Bachelor Conputer Science

University of Amsterdam

Availability Analysis of PostgreSQL under Fault Injection

Ellen van Leeuwen

August 18, 2016

Supervisor(s): Raphael Poss (UvA)

Signed: Raphael Poss (UvA)

Abstract

This thesis researches the availability of PostgreSQL under fault injection in its underlying memory. Faults can happen randomly at every moment and one of the causes is cosmic radiation. Cosmic rays hit systems and some of them cause enough charge to flip a bit in for example a computer's memory. This can cause serious problems in data centres with enormous capacities of storage and RAM, because data corruption is undesirable and the more bytes of storage available, the higher the error rate. Most of these errors are correctable with error correction mechanisms however, but memory with these mechanisms is more expensive and not widely used. Since databases are an important factor in data centres, we have researched what happens when Postgres' memory is subjected to bitflips. Little is known about the way Postgres reacts to faults and that can be a problem because it is vital for the reliability that faults are detected and preferably corrected. Experiments are performed to find out the effects fault injection has before and during database tests. We find that the availability of PostgreSQL can be temporarily affected by faults, but the error detection and handling of PostgreSQL is set up in such a way that the negative effects are minimised as much as possible for both the client and host.

Contents

CHAPTER 1

Introduction

1.1 Context

Storing data is becoming ever more important in today's world. Cloud storage is widely used and cloud providers use databases to store the massive amount of data. The data storage and corresponding databases must be reliable, because the users and customers trust the fact that their data is stored safely. Also, companies make money by collecting, analysing and selling data, for them it is important that the data storage is reliable as well.

To preserve the data, it is important that the data storage is dependable and robust. For this reason, database engines often claim to be robust and this robustness contains three important aspects.

First of all, the data integrity must be guaranteed. One would not like their data to contain faults. This is especially vital when the data is significant to, for example, analyses, research or monitoring. But not only science related fields need the correct data, commercial users also want their data to be accurate. No one would like their photos or documents to become corrupt and unusable either.

Secondly, data must be consistent. An example is the database infrastructure of a bank. The money or share transactions that happen non-stop must be correct and consistent. Otherwise wrong balances can cause many problems, not only for the bank but also for the account holders or businesses involved. Also the data must eventually be consistent in the backups or other nodes of the database engine, because we would not like to have a faulty backup replace correct data or faulty data to be backed up.

The last aspect of a robust database management system is availability. A database must be available at the desired times, which often is 24/7. Many cloud services offer unlimited access to data at every moment of the day and every day of the week. People count on the data access and the unavailability of data can cost the provider a great deal of money. Nonetheless, a database engine or server can experience availability issues, like crashes or denied requests. In these cases it is vital that these problems are fixed as fast as possible or, even better, will solve themselves.

1.2 Problem

Databases and Database Management Systems (DBMS) are an important part of the software used in cloud environments. Cloud infrastructures generally ensure the availability of resources for a certain budget, instead of guaranteeing hardware stability of individual servers. So the cloud provider assures the replacement of a faulty server for free, but the data on a faulty hard drive might be (partially) lost, which makes robustness very important. There is a problem however: database products usually do not test what happens when the underlying storage starts to corrupt data. Due to this lack of testing, there is little to no knowledge about how robust these databases are when they are confronted with data errors and if they can recover from these errors.

1.3 Scope

Database providers make claims about their robustness and dependability, but there is little research performed to test these claims. This thesis will research one database in particular, namely *PostgreSQL*. However, the methodology will be applicable to other DBMS with small changes. We have chosen to focus on one of the three aspects of robustness of databases, namely *availability*. The analysis will focus on fault injection into main memory; both the stack and the heap will be targeted. These choices will be justified in Chapter 2.

1.4 Research Questions

The main research question of this project is:
How is the availability of the PostgreSQL database engine affected by the appearance of data errors in the underlying memory?

This question is further divided into subquestions. The first subquestion is:
What does PostgreSQL do to detect and correct errors and what rate of errors is detected and corrected?
The extent to which the current PostgreSQL engine is good at detecting and correcting data corruptions is unclear. There are features in the database that can help to restore corrupted data. Replication of data (storing data on multiple server nodes) and backups are helpful if these options are switched on. There are also mechanisms to check for errors and possibly correct them, which makes us expect that most of the errors will be detected and corrected when these mechanisms are used. The problem lies in the case where it is unknown to the user that an error has occurred and the system does not detect the error. When this happens one does not know that there is a problem and that an existing backup is needed to fix the error. As a result, there is an undetected error and data corruption in the database.

The second subquestion is:
What happens when errors occur in the memory of the PostgreSQL engine?
The database engine and cloud services claim a high degree of robustness and reliability. But it is unclear what the effects are if uncorrected errors occur in the underlying memory of the database. It is a possibility that one node or the whole database may crash. Another option is that the database might reset and restart. When that does not happen, transactions might be delayed or denied because of errors. It is also possible that nothing happens and the database is not affected at all by errors.

1.5 Contribution

There is barely any research or testing done in this subject, thus the results of this thesis will contribute to knowledge about how PostgreSQL acts in the case of faults occurring in its memory. This thesis will also provide information about faults, errors, their origin and the various methods for detecting and correcting them. Furthermore, this research will give the users of PostgreSQL, whether those are individual consumers or large companies with clouds, more insight into the performance of PostgreSQL when it comes to the availability of the database.

1.6 Structure of the Thesis

In Chapter 2 we will dive deeper into the background of the subject. Faults, errors, error detection and correction will be discussed, as well as how PostgreSQL deals with errors. Hereafter, in Chapter 3 we will explain the implementation of the fault injection program and the test framework. Then the experiments will be described in Chapter 4. Furthermore, Chapter 5 contains the results to the experiments and analyses these results. In Chapter 6 we will discuss these results, draw conclusions and answer the research questions and discuss future work.

CHAPTER 2

Background

In this chapter we will first discuss the used terminology regarding faults, errors and failures and their relation. Then we will elaborate on soft errors and their causes and occurrence rates. Next we will review different methods of error detection and correction. Furthermore, databases will be discussed, with the focus the PostgreSQL database and its features and claims.

2.1 Terminology

What exactly are faults and what is the difference between faults, errors and failures? The terms are often used interchangeably, but they have different definitions. The next subsections will define the taxonomy [4, 22] of this research.

2.1.1 Faults

When we look at a system or service, the origin of a problem is a *fault*. An *active* fault can be the underlying cause of an error, while a *dormant* fault does not cause errors (yet). We can classify faults into three types [21]:

The first is a *transient fault*, of which a common example is a *bit-flip*. It is a fault that leads to incorrect data to be read from a memory section causing a so-called *soft error*. The fault can be repaired by rewriting correct data to the memory location. These type of faults are not an indication of a damaged device, because they occur randomly, one prevalent source being cosmic rays. Subsection 2.2 will discuss soft errors and cosmic rays into more detail.

The second type of fault is an *intermittent fault*. This fault occurs in *random intervals* and is caused by for example a slight change in voltage or temperature. Unlike transient faults, this type of fault can indicate device damage or at least malfunction. An illustration of an intermittent fault is the case in which a variable that is supposed to be set to zero by a program is not set. If the program uses empty memory, this fault will not have any consequences. However, if the memory used for the variable was non-zero, this fault can cause an error in the program or, if undetected, cause wrong output of a function and eventually an incorrect result of the whole program.

Lastly, the *hard fault* is the only *permanent* fault and is characteristic for a broken device, which needs to be repaired or replaced. A hard fault makes a memory location constantly return the wrong value. A *stuck-at bit* is an example of this fault.

The last two types of faults can be grouped into the category *permanent faults* and their resulting errors are often called *hard errors*.

This research will only focus on the transient fault and resulting soft error. This type of fault will be injected into the database memory and the rate of soft errors will be measured. This will be explained further in Chapter 3.

2.1.2 Errors

An *error* is defined as an incorrect part of the system state and is the result of an activated or external fault. Errors can be classified as undetected, uncorrected or corrected [21].

- The *undetected error* is not detected by hardware and is also called a silent or latent error. This type of error can cause silent data corruption or does no harm if the data is not used by the running service or program.
- The *uncorrected error* is detected by hardware, but cannot be corrected. A result of an uncorrected error can be process termination or even a node crash. Whether a node crashes depends on its settings and features regarding reliability.
- The *corrected error* is detected and corrected by hardware. This type of error will not influence the correct operation of a program, but it can lead to a temporary or permanent performance decrease on the node.

The existing methods for error detection and correction will be discussed in Section 2.3.

2.1.3 Failures

According to Baumann in [4], a *service failure* is an event that occurs when the delivered service differs from the correct service, where correct service is delivered when the system function is implemented by the service. An error that is undetected or not properly corrected can propagate through the system to the system interface. When the error unacceptably alters the delivered service and makes the system eventually malfunction, a failure occurs. A failure of a system can subsequently cause a permanent or transient external fault in other systems that connect with the faulty system [3].

2.1.4 Relation

The relation between faults, errors and failures is shown in Figure 2.1. As noted before, a fault causes an error after it is activated. This error can then propagate through the system to result in a failure at the system interface. It is possible that a failure does not occur until multiple errors have propagated. If a failure happens it can then be the cause for new faults in connected systems.

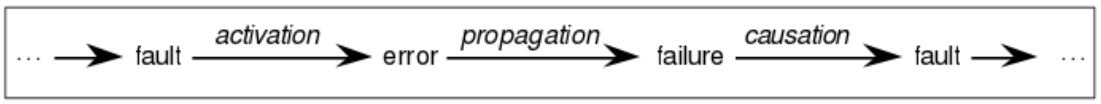

Figure 2.1: The relations between a fault, error and failure, taken from [3].

2.2 Soft errors in RAM

As explained before, a transient fault can cause a soft error. A soft error is a *single event upset* in static logic, computational logic or memory [20]. A critical charge is needed to alter the state or output of a circuit on a chip and cause an error. This charge is caused by high-energy particle strikes on the chip. These particle strikes have three different origins, which will be discussed in the following subsections.

2.2.1 Alpha particles

The first origin is *alpha particles*. These particles are helium nuclei and are decay products of traces of uranium and/or thorium coming from material contamination in wafer fabrication or chip packaging [13, 20]. Since the discovery of these particles in causation to soft errors in 1978, manufacturers have removed most of the alpha particle sources from their materials. Also shielding like internal die coating was added and chips were redesigned to make them less vulnerable to alpha particles [19]. The emission of alpha particles is however not completely eliminated. It could be done by excessively purifying the materials and the manufacturing process, but that would significantly increase the cost of the final products, without eliminating soft errors because there are two more causes for soft errors [20].

2.2.2 Cosmic rays

In 1979, Ziegler and Lanford [24] published a study about the effect of cosmic rays on computer memories. They developed a method for predicting the number of soft errors originating from cosmic rays in electronic components. A distinction is made between two types of cosmic rays; primary and secondary cosmic rays. *Primary cosmic rays* are very high-energy particles from unknown galactic origin. These particles collide with the particles in the earth's atmosphere which generates showers of *secondary cosmic rays*, which we call cosmic rays [13]. These cosmic rays consist of high-energy atomic and sub-atomic particles, including *high-energy neutrons* [20]. These neutrons are the most important cause of soft errors at terrestrial levels (i.e. sea level to 10,000 feet). The energy levels of high-energy neutrons can drop by scattering off materials in the environment resulting in *thermal neutrons*. These neutrons cannot generate soft errors most of the times, because they do not have enough energy to generate the required critical charge. In some cases however, the thermal neutron generates nuclear fission where it reacts with the material and high-energy charged particles are formed that result in soft errors.

2.2.3 Occurrence

The occurrence of soft errors, the Soft Error Rate (SER), is often measured in Failures In Time (FIT): Errors per billion (10^9) hours of use. A white paper from 2004 by Tezzaron Semiconductor [19] combined the results of many studies and concluded that a reasonable SER for modern memory devices was 1,000-5,000 FIT per Mbit. However, the reviewed studies were quite outdated. In 2009, a study by Google [18] found a staggering average SER of 25,000-75,000 FIT per Mbit of correctable errors on their DIMMs (dual in-line memory modules with DRAM integrated circuits). The yearly incidence of uncorrectable errors was 1.3% per machine and 0.22% per DIMM. This indicates that the SER is rising and as this Intel patent [11] states: "Cosmic ray induced computer crashes have occurred and are expected to increase with frequency as devices (for example, transistors) decrease in size in chips. This problem is projected to become a major limiter of computer reliability in the next decade." Smaller devices are less likely to be hit by a cosmic ray, but the components become much more sensitive to comic ray hits [25]. The critical charge needed to flip a bit decreases the smaller a component gets, but the likelihood that the hit will affect the state increases with that fact. The SER varies because it increases with memory speed and the intensity of memory use. The SER also rises with altitude, which is why NASA researches radiation hardened software and uses bigger devices [14].

For more information we recommend checking out the referenced sources and additionally this talk from 2005 by Ray Heald and its extensive references [12].

2.3 Error detection and correction in RAM

There are different ways for detecting and correcting errors [5]. The most effective and simplest technique to handle soft memory errors is to add a *parity* bit to every data word (a fixed-size piece of data). A parity bit is a check bit representing the total number of ones (including the parity) in a word. If the number of ones (without the parity) is odd, the parity bit is set to 1 (even) and if the number is even, the parity bit is set to 0 (odd). When the data is used, the parity of the data is compared to the parity bit. If a single bit is flipped, the values will not match and the error is detected. However, with any even number of bit flips the errors cannot be detected using this method, since the parity of the data will match the parity bit again. This is one of the disadvantages of parity checking, another is the fact that the detected error cannot be corrected, since there is no way to know which of the bits was flipped. Despite these downsides, parity checking implements soft error detection for a minimal cost regarding memory width and circuit complexity, because only one bit is added to each word.

The aforementioned drawbacks of parity bit checking are addressed by *error-correcting codes (ECC)*. Error-correcting memory is DRAM (dynamic random-access memory) with provided ECC protection. On error-correcting memory, errors can be detected and corrected by "adding extra bits to each data vector and encoding the data so that the information distance between any two possible data vectors is at least three" [5]. The favoured error correction scheme is the *single-error-correction, double-error-detection (SECDED)* scheme. Additional circuitry and extra parity bits can generate larger information distances so that more errors can be detected and corrected. An example is *chip-kill*, it can correct up to four neighbouring bits instantly [18]. Systems with ECC enabled see a significant decrease in the number of failures, as most of the soft errors occurring are single-bit errors. However, this protection has a few downsides, namely the design complexity is more expensive, extra memory is required and the access, parity checks and error correction cause latency [5].

2.4 Other error locii in computer systems

There are other locii in computer systems besides RAM where errors occur. Cosmic rays can also cause errors in hard drives, in contrast to the errors in RAM, these errors are permanent. Errors can also occur in the RAM within the hard drive controller, in data paths (connecting components) and in caches. The data paths and caches are often protected by some type of ECC and the memory near the processor is often shielded as well. There are multiple software solutions for errors in hard drives. RAID (redundant array of independent disks) can be used, file systems have error correction and the operating system has fault detection as well.

These locii all have some type of protection against errors and as explained in Section 2.3, there is protection for memory as well. However, this protection is not always enabled in for example cloud servers. Of the three largest cloud providers, Google Cloud Platform does not use ECC memory. This is in contrast to the practice of Microsoft Azure. A post on their blog published in March of 2015 [8] is titled:

> Microsoft Azure uses Error-Correcting Code memory for enhanced reliability and security

While Amazon Web Services states on one of their FAQ pages [1]:

> Q: Does Amazon EC2 use ECC memory?
> In our experience, ECC memory is necessary for server infrastructure, and all the hardware underlying Amazon EC2 uses ECC memory.

So Amazon uses ECC memory in their EC2 servers but it is unclear if they use it in their other products. Microsoft uses ECC memory at least since 2015 and Google does not mention ECC memory at all. The fact that ECC memory is not even (fully) used by all three largest cloud providers makes it probable that other (smaller) cloud providers do not utilise ECC memory either. This can be considered to be a bad trade-off between hardware cost and data and application security. This makes researching errors in RAM very interesting and that is why we decided to choose that focus.

2.5 Databases

A database is a collection of related data indexed in tables. A client can use a database management system (DBMS) to communicate with the database to store, delete and update records. The database can also be queried to combine and extract information from related tables. Databases are susceptible to faults, because they contain data that is stored on hard drives and in memory. Databases also have active and idle processes running in memory managing the database and performing background tasks. The chosen database for our research is PostgreSQL. What is interesting about Postgres is the fact that it has multiple processes running in memory, which would possibly make it extra sensitive to faults compared to other DBMS.

2.5.1 PostgreSQL

PostgreSQL [15] is an object-relational database management system (ORDBMS). It is open-source, cross-platform and runs on many operating systems. Postgres implements most of the core features of the 2011 stable SQL standard, is ACID (Atomicity, Consistency, Isolation, Durability) compliant and supports binary large objects, including pictures, sounds and video. PostgreSQL supports asynchronous replication, online backups and write-ahead logging for fault tolerance. PostgreSQL has many more features, but the aforementioned features are relevant for this research.

PostgreSQL is process-based, which means that it starts one Postgres instance per connection. The postmaster is also always running and there are multiple helper processes working. Examples are the mandatory writer, WAL writer and checkpointer processes and optional logger, autovacuum launcher and stats collector processes.

PostgreSQL has made some statements regarding reliability and stability, a few of them are shown below.
This PostgreSQL Wiki article [23] about data integrity:

> PostgreSQL has always been strict about making sure data is valid before allowing it into the database, and there is no way for a client to bypass those checks.

One of the advantages stated by PostgreSQL [16] is:

> Legendary reliability and stability
> Unlike many proprietary databases, it is extremely common for companies to report that PostgreSQL has never, ever crashed for them in several years of high activity operation. Not even once. It just works.

Reading these statements sparks our interest about how reliable and robust Postgres really is and that is what we hope to find out with this research.

CHAPTER 3

Implementation

This chapter will explain the implementation that has been created to make the experiments and results possible.

3.1 Software

A few different software packages are used for this research. To start off with the database, `PostgreSQL 9.5` is installed. Every script and program is implemented in `Python 3.5.1`. Postgres is compatible with Python, among many other programming languages and there is a library interface available for Python. This library interface is `psycopg2 2.6.2`. This package allows the user to connect to a database, create databases and tables, execute queries and perform other related actions.

Another important tool that is used is `ptrace`. Ptrace, an abbreviation of process trace, is a system call that allows a process, the tracer, to observe and control the execution of another process, the tracee, and examine and change the memory and registers of the tracee [17]. The ptrace system call is available in several Unix and Unix-like operating systems, such as Linux and Mac OS X. Ptrace is wrapped in a Python library called `python-ptrace 0.9` to enable ptrace to be used in Python programs. We can connect to a process with the PtraceDebugger and access and manipulate its memory contents with this package.

Furthermore, we need to setup the tests, run the tests and monitor the results and database. *Apache JMeter* [2] is a tool whose implementation meets all those requirements. A user can create a Test Plan, in which it can set up a Thread Group with the desired number of threads (users) and the loop count, schedule the execution and more settings. A Config Element can be attached to the thread group which in this case is a Java Database Connection (JDBC) Connection Configuration. Here we can set up the database connection and configure the connection pool. Then a Sampler is added to the Thread Group, for these experiments a JDBC Request is needed. In this Request we define the SQL Query used in the particular test. Lastly, Listeners can be added to the Thread Group. We use the View Results Tree listener to view the query results and set up the output file to write the results to.

3.2 Fault injection

The fault injection program is the core of the implementation. To run the program, three parameters have to be supplied: the name of the database, the memory target and the number of faults to inject. The program uses the database name to find the corresponding database process identifier (PID). Then Ptrace is used to add the PID to the PtraceDebugger, which results in a process that can be used in the rest of the program. Next, the memory mapping of the process is read and the mappings corresponding with the given memory target, which can be stack or heap, are saved. The next stage is acquiring the memory addresses of the designated target. This is performed by matching a pattern to the memory mapping line(s) to extract the

16 bit hexadecimal addresses. Figure 3.1 shows an example of the memory mapping lines. As shown, there can be more than one set of addresses and if that is the case, the addresses are merged into one set. The addresses will be converted to 16 bit integers for convenience.

```
0x0000000001b82000-0x0000000001bbf000 => [heap] (rw-p)
0x0000000001bbf000-0x00000000023e7000 => [heap] (rw-p)
```

Figure 3.1: Example of the lines corresponding to the heap in the memory mapping of a database process.

After requiring the addresses, everything is ready for the actual fault injection. As mentioned before, the type of fault we are going to inject is the transient fault or bit-flip. First, a random address is chosen in the range of the start and end address of the desired memory location. Then one byte is read from that address and the bit-flip operation is executed on one random bit from that byte. This is done using the following XOR operation:

$$\text{new\_byte} = \text{xor}(\text{ord}(\text{old\_byte}),\ 2^{\text{randint}(0,\ 8\ -\ 1)}).$$

Subsequently, the new byte is written to the same memory location. This process is repeated the given number of times.

There is however one observation, because we are injecting faults into the stack and heap and these locations are constantly used by the database, a portion of the bit flips are not actually stored. While performing or right after the bit flip, the memory location may already be overwritten with new data, which will eliminate the bit flip. Since we cannot lock the memory location to do the reading and writing of the byte, this cannot be prevented. However, this is exactly the way it works in real life. Cosmic rays cause bit flips, but these can be overwritten right away or happen on memory locations that are not currently used by the database processes. So the fact that not all bit flips will actually affect the database will correspond with the real life cosmic ray effects.

3.3 PostgreSQL setup

A few settings had to be changed in order to be able to connect to the database remotely. First we edit the file `postgresql.conf`. The line `listen_addresses='localhost'` is changed to `listen_addresses='*'` to enable the database to listen to hosts other than localhost. Another change in this file is the `max_connections=100` line. We increase this variable from 100 to 120 because we want to be able to connect 100 clients as well as open a psql connection to the database at the same time (the reason will be explained in Section 3.6).

Furthermore, one other file needs to be adapted, namely `pg_hba.conf`. In this file we add the IP-address of the other VM so that it can connect to the database to perform requests. We make sure that md5 is set so that we can only connect with a password.

The last action that is required to make the database setup remotely accessible is the resetting of the password for the postgres user. We execute the query `ALTER ROLE postgres PASSWORD '[password]'` in psql on the localhost and then the PostgreSQL setup is ready.

3.4 Database setup

3.4.1 Dataset

We use one dataset and query workload for this research. The dataset is a Multispectral Image Database (hereafter referred to as *ms_data*) made available to the research community [7]. This dataset was created for another study and contains 32 folders with in total 992 pictures of PNG format. The original filename and binary file data of each picture is added to one table in a Postgres database and the size of the filled table is 390MB.

3.4.2 Dataset insertion

We have written a fairly universal Python script to insert data into a database. The script first creates a database with a given name or connects to the database if it already exists. There are two functions present in this script. The first creates a table for every folder in the given data path with the same name as the folder. It then inserts the pictures in that folder into the table. The second function creates one table with the given name of the database and inserts every picture in the given data path into that table.

3.5 Restoring data

An important last part of the implementation is restoring the original data. We want to prevent faulty data to be used in the next experiments. We have initialised the PostgreSQL service on an empty *Btrfs* partition on a virtual machine. Btrfs is a fairly new filesystem for Linux with various advanced features while focusing on fault tolerance, repair and easy administration [6]. One of Btrfs' features are subvolumes, which are separate internal filesystem roots. These subvolumes can be created on the filesystem, where they are shown as folders. The subvolumes can be mounted as well to use only that part of the data. The original partition can always be remounted which will show all of the subvolumes again.

Another feature of Btrfs are snapshots. A snapshot is a subvolume itself and an exact copy of a selected subvolume. The snapshot does not use storage capacity until a file in the original subvolume is changed. Then the snapshot stores a copy of the original version of the file, while the other files remain mapped to the unchanged original files. Snapshots can also contain folders, so they can be used to create regular backups of data on subvolumes.

In our implementation, a snapshot is used to restore the original data. We created a subvolume on the Btrfs partition which was used as the data directory when the database service was initialised. We setup PostgreSQL and inserted the dataset as described in the former sections and as soon as the setup was complete, we created a snapshot of the data directory. This snapshot is mounted and it is the basis for every experiment. At the end of a run, the Postgres service is stopped, the partition is unmounted and the snapshot is mounted on that location. This provides the next run with a fully correct data directory to start the database service on again.

3.6 Testing

The global steps used in the testing scripts are described in the following roadmap:

Step 1 Log into Postgres server with an SSH connection and start the PostgreSQL service

Step 2 Set up a psql connection to the database using a second screen

Step 3 Execute the fault injection script

- ⇒ An extra second screen is started in which step 3 is executed for the local experiments with fault injection during the test
- ⇒ A sleep of 5 seconds is added in the remote experiment with fault injection during the test

Step 3 Execute the JMeter testplan

- ⇒ An SSH connection to the second VM is set up first for the remote experiments

Step 4 Log into Postgres server with an SSH connection and stop the PostgreSQL service

Step 5 Unmount the Btrfs partition

Step 6 Mount the snapshot to restore original data

Note When second screens are used, the screens are closed as soon as they are no longer needed

CHAPTER 4

Experiments

Each test inserts 0, 1, 10, 100, 250, 500 and 1000 bit flips on the stack or the heap of the database process. In one experiment, the testplan is run *after* the aforementioned fault injection and in another experiment the fault injection is run *during* the execution of the testplan. The fault injection is done in one instant in about one second and not distributed along the test.

4.1 Setup

The experiments are run on a server on the *Google Cloud Platform* [10] with a *Btrfs partition* of 4GB. The data directory that Postgres uses is stored on the aforementioned partition. Furthermore, JMeter contains the test plan, executes the queries and monitors and logs the response into a specified CSV file. In two of the four tests, JMeter is run on another virtual machine to enable remote testing of the database.

4.2 Test

The Select test is performed on the ms_data database. The query in Figure 4.1 selects one random file name and corresponding data from all of the pictures in the database. The random ordering was chosen to prevent cached results from being used and to have the full database accessed, since faults can occur anywhere in the stack or heap memory.

```
SELECT orig_filename, file_data
FROM ms_data
ORDER BY random()
LIMIT 1
```

Figure 4.1: SELECT query performed on the ms_data database.

This query is executed one hundred times by one hundred clients every time the test is run. The ramp-up period in which to start the threads is set to zero or one second, so all of the users start querying the database almost instantaneously. These settings are present in the Testplan that is executed by JMeter. One experiment totals 100,000 requests for each number of bitflips, as the experiments are repeated ten times for each number of bitflips.

CHAPTER 5

Results and Analysis

5.1 Fault injection before test

The first two experiments are set up so that the fault injection is finished before the JMeter test is started.

5.1.1 Local experiment

The first experiment was run locally on the server. The test plan that was used was described in Chapter 4, however, the ramp-up period to start the threads was one instead of zero seconds in this test. The duration of each run was between 2.45 and 3.02 minutes.

Table 5.1: Number of errors per run (10,000 requests), per number of bitflips on the two memory locations.

	Number of bitflips						
Run	0	1	10	100	250	500	1000
1	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0
4	0	0	0	0	1	0	0
5	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0
Total	0	0	0	0	1	0	0

(a) Fault injection on stack

	Number of bitflips						
0	1	10	100	250	500	1000
1	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
1	0	0	0	0	0	0

(b) Fault injection on heap

Tables 5.1a and 5.1b show the number of errors per run per number of bitflips on the stack and heap respectively in this experiment. Both errors were `java.sql.SQLException: Error preloading the connection pool` errors. We assume that these errors occured because the maximum number of connections of 100 was reached while there was still one thread to be started. It is possible that the psql connection used right before the test was not closed fast enough, which causes a request of one of the 100 clients to be denied. These two errors were so sporadic and at the start of both runs, that we conclude that these errors are most likely not caused by the fault injection. Especially since one of the two happened during a run without injection of faults.

5.1.2 Remote experiment

The second experiment was run remotely on another VM. The duration of each run was between 2.28 and 2.41 minutes, which made it slightly faster than the locally run test. The reason for this is more memory congestion happening when running the tests on the server instead of on the client remotely. The testplan that was used is exactly the one described in Chapter 4

In this experiment, we found that no errors were reported to the client. Every run ran perfectly in terms of the abundance of errors under the different amounts of bitflips injected on the stack and heap. This strengthens the conclusion from the previous subsection that those two errors were probably not caused by the fault injection.

Now when we take a look at the log of the host, the PostgreSQL service. It turns out that errors actually have occurred. These errors did not show up in the test because Postgres detected them in the few (milli)seconds between the fault injection and the test. This makes sure that the client has no knowledge and inconvenience from these errors, while they are reported to the host. There were 59 errors reported in the log file, as described in Table 5.2.

Table 5.2: The number of errors and their descriptions from the PostgreSQL log file generated in this experiment.

Amount	Error description
44	`server process was terminated by signal 11: Segmentation fault`
3	`server process was terminated by signal 7: Bus error`

2	`stack smashing detected: postgres:`
	`postgres ms_data_1 [local] startup terminated.`
	`server process was terminated by signal 6: Aborted`

3	`Error in 'postgres: postgres ms_data_1 [local] idle':`
	`free(): invalid next size (normal).`
	`server process was terminated by signal 6: Aborted`

2	`Error in 'postgres: postgres ms_data_1 [local] idle':`
	`free(): invalid pointer.`
	`server process was terminated by signal 6: Aborted`

1	`Error in 'postgres: postgres ms_data_1 [local] idle':`
	`double free or corruption (out).`
	`server process was terminated by signal 6: Aborted`

1	`Error in 'postgres: postgres ms_data_1 [local] idle':`
	`free(): invalid next size (fast).`
	`server process was terminated by signal 6: Aborted`

1	`Error in 'postgres: postgres ms_data_1 [local] idle':`
	`corrupted double-linked list.`
	`server process was terminated by signal 6: Aborted`

1	`Error in 'postgres: postgres ms_data_1 [local] idle':`
	`free(): corrupted unsorted chunks.`
	`server process was terminated by signal 6: Aborted`

1	`Inconsistency detected by ld.so.` `server process exited with exit code 127`

Due to the random injection of fault into the stack or heap, the errors are various. The segmentation fault was very common, but it is interesting that Postgres even has a 'stack smashing detection' which prevents the startup of the database. This will prevent damage being done to the database which is a good thing.

5.2 Fault injection during test

The second two experiments are set up so that the fault injection is started during the execution of the JMeter testplan.

5.2.1 Local experiment

This experiment is run on the database server, so a second screen is started in which the fault injection is executed, while at the same time the JMeter testplan is executed. The duration of each run was between 2.45 and 3.02 minutes.

Table 5.3: Number of errors per run (10,000 requests), per number of bitflips on the two memory locations.

	Number of bitflips						
Run	0	1	10	100	250	500	1000
1	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0
3	0	0	0	145	0	0	169
4	0	0	0	0	0	0	0
5	0	0	0	0	0	0	119
6	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0
9	0	0	0	0	0	162	0
10	0	0	0	0	0	0	0
Total	0	0	0	145	0	162	288

(a) Fault injection on stack

Number of bitflips						
0	1	10	100	250	500	1000
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

(b) Fault injection on heap

Tables 5.3a and 5.3b show the number of errors per run per number of bitflips on the stack and heap respectively in this experiment. The fault injection on the heap has not caused any errors, while there were errors reported in four runs with fault injections on the stack.

The errors all occurred within the first 30 seconds of a run. There were two different types of errors observed:

- 400 errors were I/O errors, specifically: `org.postgresql.util.PSQLException: An I/O error occurred while sending to the backend.`
- 195 errors were recovery mode errors, namely: `org.postgresql.util.PSQLException: FATAL: the database system is in recovery mode`.

The I/O errors are caused by the termination of all active connections, the requests cannot be accepted by the service. The sending of the requests continues while the database is still reinitialising and recovering from the previous error, this causes the recovery mode errors. The database will accept connections again once the recovery has finished.

It is interesting to see that there are errors occurring with faults on the stack, but not with faults on the heap. An explanation can be that the heap is almost seven times larger than the stack, so the probability that a bitflip on the heap will affect the database is smaller compared to the stack.

5.2.2 Remote experiment

In this experiment, the JMeter test is run remotely on another VM. A second screen is started in which an SSH connection is made to the VM and the fault injection is run. While another SSH connection is made to the database server and the JMeter test is begun. However, a 5 second delay is added before the fault injection, to ensure that it would happen during the JMeter test. This is done because the SSH connections might slow down the start of the JMeter test, while the fault injection would then already take place. The duration of each run was between 2.28 and 2.41 minutes.

Table 5.4: Number of errors per run (10,000 requests), per number of bitflips on the two memory locations.

	Number of bitflips						
Run	0	1	10	100	250	500	1000
1	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0
4	0	0	0	0	0	0	100
5	0	0	0	0	0	0	100
6	0	0	0	105	0	0	0
7	0	0	0	0	0	0	100
8	0	0	0	0	0	0	154
9	0	0	0	0	104	0	0
10	0	0	0	0	0	100	0
Total	0	0	0	105	104	100	454

(a) Fault injection on stack

	Number of bitflips						
	0	1	10	100	250	500	1000
1	0	0	0	100	120	100	100
2	0	0	0	100	100	111	120
3	0	0	101	110	100	100	100
4	0	0	110	140	132	113	0
5	0	0	102	143	100	134	115
6	0	100	0	0	112	0	100
7	0	0	101	105	108	116	100
8	0	0	0	100	100	149	100
9	0	0	0	100	102	100	104
10	0	0	0	0	100	0	100
Total	0	100	414	898	1074	923	939

(b) Fault injection on heap

In this experiment a lot of errors occurred both with bitflips on the stack and the heap. This is very different from the previous test, in which no errors occurred with bitflips on the heap. The added delay plays a big role in this change. It is very well possible that most of the faults were already detected before the JMeter test began, because it has a slight delay starting up. In this experiment, the JMeter test is already running before the fault injection is performed, so the faults have to be detected and corrected during the test. The errors again all occurred within the first 30 seconds of a run. At the end of the experiment 700 I/O errors and 63 recovery mode errors were reported with bitflips on stack. While bitflips on the heap caused a total of 4000 I/O errors and 348 recovery mode errors reported with bitflips on heap.

There is a trend visible of increased numbers of errors with increased numbers of bitflips, but for the heap it seems limited with a number of bitflips between 100 and 250. The reason is that the errors are caused by just one fault, which is why most runs experience 100 errors or slightly more. When an error is detected, the corresponding server process is terminated and subsequently all other active server processes are terminated. “The postmaster has commanded this server process to roll back the current transaction and exit, because another server process exited abnormally and possibly corrupted shared memory.” When all processes are terminated, the database is reinitialised and automatically recovered. This causes 100 requests to fail at once. In the cases that there are more than 100 requests that fail, the database was still in recovery mode and this causes every request to fail until the recovery is done. When the database is recovered, every new request is accepted again.

The results of run 6 with 1 bitflip on the heap prove that one fault can cause serious errors. This one bitflip caused an error where Postgres responded to by terminating the current 100 running processes, which caused one percent of the total amount of requests to fail.

It is interesting to see what errors the injected faults have actually caused. Table 5.5 shows the amounts and types of the 7 errors caused by the faults injected into the stack. Postgres detected four of the seven errors as stack smashing, which we obviously did, so that is very much correct. The other three errors were segmentation faults, which is a very common error as we saw earlier in this chapter.

Table 5.5: The number of errors and their descriptions from the PostgreSQL log file generated in the stack part of the experiment.

Amount	Error description
3	`server process was terminated by signal 11: Segmentation fault`

4	`stack smashing detected: postgres:`
	`postgres ms_data_1 [local] startup terminated.`
	`server process was terminated by signal 6: Aborted`

Table 5.6 shows the amounts and descriptions of the 40 errors caused by the injected faults into the heap. Again the fault injection has caused various errors, with the segmentation fault again being the dominant one.

Table 5.6: The number of errors and their descriptions from the PostgreSQL log file generated in the heap part of the experiment.

Amount	Error description
23	`server process was terminated by signal 11: Segmentation fault`
2	`server process was terminated by signal 7: Bus error`

5	`Error in 'postgres: postgres ms_data_1 [local] idle':`
	`double free or corruption.`
	`server process was terminated by signal 6: Aborted`

5	`Error in 'postgres: postgres ms_data_1 [local] idle':`
	`corrupted double-linked list.`
	`server process was terminated by signal 6: Aborted`

2	`Error in 'postgres: postgres ms_data_1 [local] idle':`
	`free(): invalid next size (normal).`
	`server process was terminated by signal 6: Aborted`

1	`Error in 'postgres: postgres ms_data_1 [local] idle':`
	`free(): invalid pointer.`
	`server process was terminated by signal 6: Aborted`

1	`Error in 'postgres: postgres ms_data_1 [local] idle':`
	`free(): invalid next size (fast).`
	`server process was terminated by signal 6: Aborted`

1	`Error in 'postgres: postgres ms_data_1 [local] idle':`
	`munmap_chunk(): invalid pointer.`
	`server process was terminated by signal 6: Aborted`

5.3 Latency effects

We have also looked at the effects errors have on the average latency of requests in a run. "JMeter measures the latency from just before sending the request to just after the first response has been received. Thus the time includes all the processing needed to assemble the request as well as assembling the first part of the response, which in general will be longer than one byte." [9]

We have calculated the average latency for every run in the last experiment. We found that the average latency per request in non error runs was 3.795 milliseconds. Meanwhile the average latency per request in error runs was 4.891 milliseconds. This is an increase of 1.096 milliseconds or 28.88%. The data and calculations can be viewed in Appendix A.

CHAPTER 6

Conclusion and Discussion

6.1 Conclusion

First of all we wondered what PostgreSQL does to detect and correct errors and what rate of errors is detected and corrected. We found that Postgres has elaborate checks to detect many different types of errors. PostgreSQL does not seem to correct these errors however. In our experiments we found that PostgreSQL detected all errors. As soon as PostgreSQL detects an error in one of its processes, it terminates that process and subsequently commands all other active processes to roll back the current transaction and exit. This prevents corruption in shared memory to affect other processes and create more errors. Possible other errors that would have been able to affect the database are eliminated due to this procedure. When all active processes are terminated, the database goes into recovery mode and is reinitialised. When the database is recovered, it is ready to accept new connections and allow new requests to be performed.

Aside from the denial of all active connections and their transactions when an error has been detected, a delay is also a result of an error. The average latency of a request when an error has occurred in the test is 28.88% higher than the average latency of a request in a run without errors. This answers the second subquestion stated in Chapter 1.

Finally to answer the main research question. The availability of the PostgreSQL database engine is only slightly affected by the appearance of data errors in the underlying memory. When an error is detected, all active connections and their transactions fail, but the database recovers in an instant and new connections and transactions are accepted as soon as the database is out of recovery mode. This method protects other processes from becoming faulty as well and it only has a relatively small effect on the clients.

6.2 Discussion

It must be noted that the bitflips injected in these experiments are not very common. Cosmic rays do cause faults on terrestrial systems, but nowhere near the amount of faults injected in the tests. We injected larger amounts of faults to get results faster. Because it would take many more experiments to see small amounts of faults cause errors. However, we did see one case in which one bitflip caused an error. So it is proven that even though the probability is low, the possibility exists that a cosmic ray will cause noticeable errors in running systems.

Another setting to speed up the experimenting is injecting all faults at once. This increases the probability that one of those faults causes an error. For example when many stack addresses are hit in a short period of time, Postgres responds with a ‘stack smashing detected’ error. It is not probable that this error would happen in a natural situation, only in case multiple cosmic rays hit the stack specifically, which is very unlikely. However, there were many other types of errors detected that can be caused by random cosmic ray hits. It is the question what would happen if the faults were introduced distributed over a longer period of time. Since one fault can cause errors, it is probable that multiple sets of errors occur in the database engine. This could result in more errors per run, but the likelihood of errors to occur in the first place will decrease.

6.3 Future work

There are a few extra possibilities for further testing the PostgreSQL database engine and its vulnerability to errors. We can do experiments with different databases and with write, update and join queries as well as different select (read) queries. Even more varying numbers of bitflips can be injected and the fault injection can be done distributed during tests instead of all at once. Another point of interest is the effect enabled replication would have on preventing and possibly correcting errors. Finally the experiments can be performed on other cloud services, to see if the error rate might be lower. On cloud services with ECC memory, the faults might be detected even before they could cause errors in PostgreSQL processes.

Acknowledgements

First and foremost, I would like to thank my supervisor Raphael Poss for guiding me through the whole process of this graduation project. It has been quite a long ride and I am very thankful for his assistance, informative talks and helpful comments during the research and writing of this thesis. He was more positive and complimenting on my work than I could have imagined and that gave me extra confidence that I actually was going to make it to the finish line.

Furthermore, I would like to thank my family from the bottom of my hart for motivating me and supporting me until the end. My mum, dad and brothers were always interested and helped me get through this process and finish the thesis in time.

I am also very grateful to my friends Imke, Seyla, Robin and Willem for reading and commenting on my thesis, always spotting the flaws I did not see anymore and providing helpful suggestions for improving the text even more. Thanks to Vincent as well for providing me with his extensive knowledge of PostgreSQL and Alyssa and Erik for their assistance in helping me to make my code do what it was supposed to do.

Bibliography

[1] *Amazon EC2 FAQs*. URL: https://aws.amazon.com/ec2/faqs/ (visited on 08/03/2016).

[2] *Apache JMeter™*. URL: http://jmeter.apache.org/index.html (visited on 07/25/2016).

[3] Algirdas Avižienis, Jean-Claude Laprie, and Brian Randell. "Dependability and Its Threats: A Taxonomy". In: *Building the Information Society*. Springer US, Aug. 2004, pp. 91–120. DOI: 10.1007/978-1-4020-8157-6_13.

[4] Algirdas Avižienis et al. "Basic concepts and taxonomy of dependable and secure computing". In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (Jan. 2004), pp. 11–33. DOI: 10.1109/tdsc.2004.2.

[5] Robert Baumann. "Soft Errors in Advanced Computer Systems". In: *IEEE Design & Test of Computers* 22.3 (May 2005), pp. 258–266. ISSN: 0740-7475. DOI: 10.1109/mdt.2005.69.

[6] *Btrfs Wiki Main Page*. URL: https://btrfs.wiki.kernel.org/index.php/Main_Page (visited on 08/17/2016).

[7] *CAVE — Projects: Multispectral Image Database*. URL: http://www1.cs.columbia.edu/CAVE/databases/multispectral/ (visited on 07/25/2016).

[8] Scott Field. *Microsoft Azure uses Error-Correcting Code memory for enhanced reliability and security*. Mar. 16, 2015. URL: https://azure.microsoft.com/en-us/blog/microsoft-azure-uses-error-correcting-code-memory-for-enhanced-reliability-and-security/ (visited on 08/03/2016).

[9] *Glossary*. URL: http://jmeter.apache.org/usermanual/glossary.html (visited on 08/17/2016).

[10] *Google Cloud Platform*. URL: https://cloud.google.com/ (visited on 08/03/2016).

[11] Eric C Hannah. "Cosmic Ray Detectors for Integrated Circuit Chips". Patent US7309866 B2. Dec. 18, 2007. URL: https://www.google.com/patents/US7309866.

[12] Ray Heald. "How Cosmic Rays Cause Computer Downtime". In: *IEEE Reliability Society SCV Meeting*. 2005. URL: http://www.ewh.ieee.org/r6/scv/rl/articles/ser-050323-talk-ref.pdf.

[13] J. G. Llaurado. "Commentary III - Can cosmic rays interfere with computer memories?" In: *International Journal of Bio-Medical Computing* 12.4 (July 1981), pp. 275–281. DOI: 10.1016/0020-7101(81)90040-4.

[14] Peter Mehlitz and John Penix. "Expecting the Unexpected - Radiation Hardened Software". In: *Infotech@Aerospace*. American Institute of Aeronautics and Astronautics (AIAA), Sept. 26, 2005. DOI: 10.2514/6.2005-7088.

[15] *PostgreSQL: About*. URL: https://www.postgresql.org/about/ (visited on 07/25/2016).

[16] *PostgreSQL: Advantages*. URL: https://www.postgresql.org/about/advantages/ (visited on 07/25/2016).

[17] *ptrace - process trace*. URL: http://man7.org/linux/man-pages/man2/ptrace.2.html (visited on 07/29/2016).

[18] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. "DRAM Errors in the Wild: A Large-scale Field Study". In: *Communications of the ACM* 54.2 (Feb. 1, 2011), pp. 100–107. ISSN: 00010782. DOI: 10.1145/1897816.1897844.

[19] Tezzaron Semiconductor. "Soft Errors in Electronic Memory - A White Paper". In: 2004. URL: http://www.tezzaron.com/media/soft_errors_1_1_secure.pdf.

[20] Charlie Slayman and Mike Silverman. "Soft Errors in Modern Memory Technology". In: *CMSE Conference 2011*. 2011. URL: http://www.opsalacarte.com/pdfs/Tech_Papers/DRAM_Soft_Errors_White_Paper.pdf.

[21] Vilas Sridharan and Dean Liberty. "A Study of DRAM Failures in the Field". In: *2012 International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '12. IEEE Computer Society Press, Nov. 2012, 76:1–76:11. ISBN: 978-1-4673-0804-5. DOI: 10.1109/sc.2012.13.

[22] Vilas Sridharan et al. "Memory Errors in Modern Systems: The Good, The Bad, and The Ugly". In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery (ACM), Mar. 15, 2015, pp. 297–310. ISBN: 978-1-4503-2835-7. DOI: 10.1145/2694344.2694348.

[23] *Why PostgreSQL Instead of MySQL: Comparing Reliability and Speed in 2007 - PostgreSQL wiki*. URL: https://wiki.postgresql.org/wiki/Why_PostgreSQL_Instead_of_MySQL:_Comparing_Reliability_and_Speed_in_2007 (visited on 07/25/2016).

[24] J. F. Ziegler and W. A. Lanford. "Effect of Cosmic Rays on Computer Memories". In: *Science* 206.4420 (Nov. 16, 1979), pp. 776–788. ISSN: 0036-8075, 1095-9203. DOI: 10.1126/science.206.4420.776.

[25] J. F. Ziegler et al. "IBM Experiments in Soft Fails in Computer Electronics (1978–1994)". In: *IBM Journal of Research and Development* 40.1 (Jan. 1996), pp. 3–18. DOI: 10.1147/rd.401.0003.

APPENDIX A

Latency Calculations

Remote experiment – injection during test

Stack	Number of bitflips						
Run	0	1	10	100	250	500	1000
1	37537	40199	34929	42702	28424	38540	38674
2	38850	45165	36313	39266	29498	39265	42069
3	41962	42449	39607	42813	38330	40250	44885
4	37403	35819	31640	35471	30643	34887	52016
5	34996	47194	38603	44073	28012	43855	50199
6	40006	32942	33493	46890	39962	38797	31297
7	36076	27247	39061	41634	46504	40332	50065
8	37160	36730	46688	39164	35654	32947	45297
9	41993	39361	40627	36104	57236	38135	40127
10	44991	35698	33040	44540	25315	46564	28320
Average	39097.4	38280.4	37400.1	41265.7	35957.8	39357.2	42294.9

Heap	Number of bitflips						
Run	0	1	10	100	250	500	1000
1	43211	26964	40316	48437	51970	43762	55704
2	42206	40788	47180	59274	46320	42477	43019
3	40015	25056	51543	41789	34084	54890	41223
4	44039	27157	53398	43583	51568	53261	41312
5	31961	37506	54951	45724	40197	54372	34897
6	44790	49802	35952	35954	49573	36532	53535
7	38930	46992	47144	50330	58437	48792	52480
8	38479	34140	36198	52367	49678	48807	49022
9	44531	41315	29607	49243	44330	47321	53375
10	40244	36504	35394	40058	43202	44479	50328
Average	40840.6	36622.4	43168.3	46675.9	46935.9	47469.3	47489.5

Average error latency	48912.57447	Average error latency per request	4.891257447
Average no error latency	37953.36559	Average no error latency per request	3.795336559

Marked cells contain latencies in error runs		Difference	1.095920888
Unmarked cells contain latencies in no error runs		Increase %	28.87545994