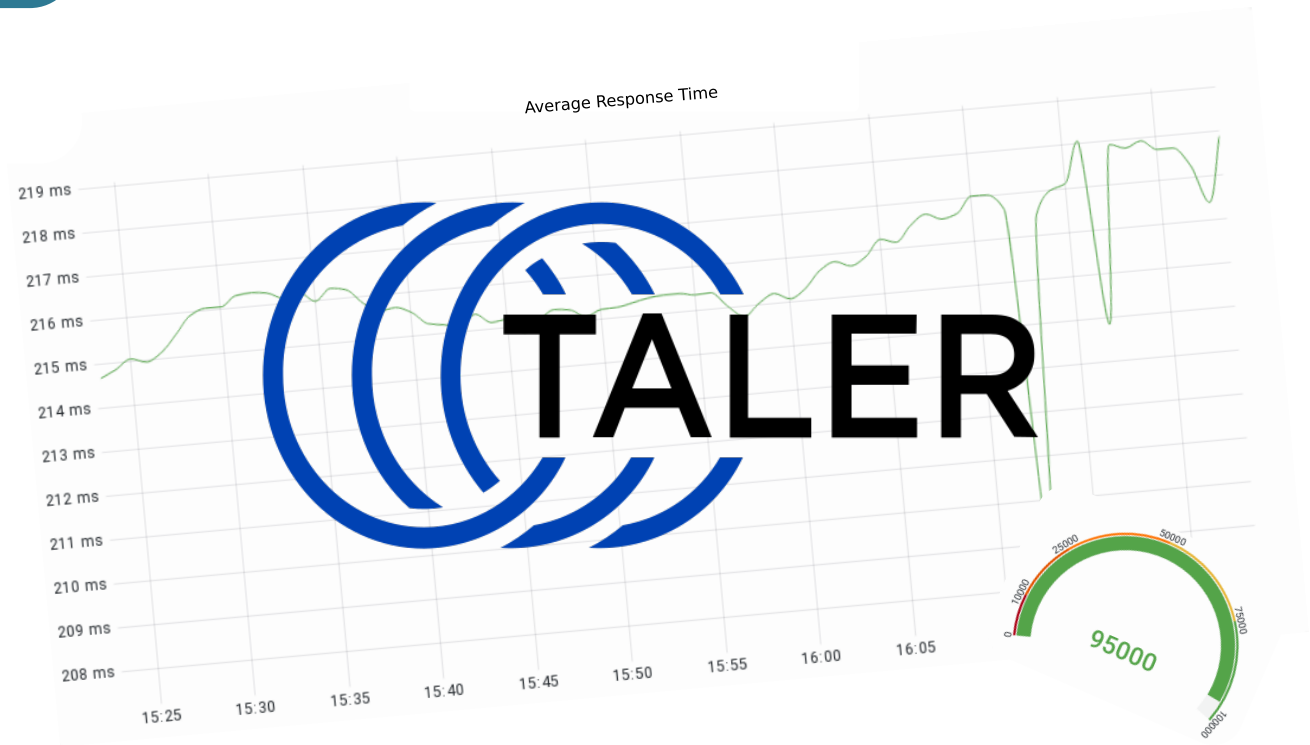
Bern University
of Applied Sciences



GNU Taler Scalability

Measuring and Improving the Performance of GNU Taler on Grid'5000

Course of study	Bachelor of Science in Computer Science
Author	Marco Boss and Christian Grothoff
Advisor	Prof. Christian Grothoff
Co-advisor	Prof. Andreas Habegger
Project partner	Taler Systems SA
Expert	Han Van der Kleij

Version 1.0 of June 15, 2022

- Engineering and Computer Science
- Computer Science

Abstract

This thesis is on the GNU Taler scalability experiments conducted on Grid'5000 in the first half of 2022, which was preceded by preparations carried out in the second half of 2021 supported by the Next Generation Internet initiative's NGI Fed4Fire+ program.

The primary goal of this study was to evaluate the scalability of GNU Taler in a real-world scenario. That is, moving away from the loopback system to a distributed network and identifying improvement opportunities therein to analyze and improve performance. While the basic framework was already known from the previous study, this work focuses on extending the framework and making further improvements to GNU Taler. This includes, among other things, the horizontal distribution of the PostgreSQL database.

We identified and fixed several bottlenecks in the GNU Taler software. We parallelized the execution of the cryptographic frontend, leaving the PostgreSQL database as the natural bottleneck. Here, we optimized queries and modified the schema to enable table partitioning.

The scalability demonstrated in our experiments shows that Taler is very capable of processing enough transactions per second to be considered an acceptable payment system. Throughout this work, we were able to increase the performance of Taler by a factor of 95, from about 300 to 28.5k transactions per second, showing that Central Bank Digital Currencies based on Taler would require only a few exchanges per continent.

Acknowledgements

We would like to thank the support staff from INRIA, which were very helpful many times when we had problems in deploying experiments on Grid'5000 or struggled with the jFed tool. We are also grateful for the support provided by NGI Fed4Fire+.

Personally, I would like to thank my two advisors, Christian Grothoff and Andreas Habegger, who helped me with hints and tips on what I could try next to find bottlenecks. As it turned out, it was more difficult than we expected.

Then I want to thank Florian Dold, the CTO of Taler Systems SA. He made a lot of improvements to the wallet clients to make the experiments feasible. The same goes for Christian Grothoff, who made most of the corrections and changes to the GNU Taler exchange software allowing me to focus on the experiments, identifying problems and improving the SQL queries.

Special thanks go to Jose E. Marchesi, who gave me many helpful tips on what we could test to find out what the problem with our performance might be.

Finally, I would like to thank everyone who responded to my inquiries on mailing lists and other channels.

Contents

1. Introduction

Today, central banks all over the globe are investigating possible designs for implementing a Central Bank Digital Currency (CBDC). Unfortunately, most of today's electronic payment systems do either not offer adequate technical privacy assurances to citizens, or are too slow to handle the expected transaction load [1].

According to a personal discussion with with Giesecke+Devrient, a payment system that is to support 500 million people for all payments should be able to handle about 100'000 Transactions Per Second (TPS). This would be sufficient to handle the combined rate of **all** currently used means of payment, such as credit cards, bank transactions and cash. Naturally, a smaller economy might work well on a much lower transaction rate. For example, the same per-capita use would imply a need of 2'000 TPS for all of Switzerland.

To assess the scalability of GNU Taler, performance experiments were carried out using INRIA's Grid'5000. Grid'5000 is a distributed testbed providing computing and storage resources distributed across France (and Luxembourg). The grid allows experimenters to reserve groups of machines for experiments giving users direct access to the local hardware and communication over shared high-speed network links. [2]

1.1. GNU Taler

GNU Taler is a privacy-friendly implementation of a payment system based on neither peer-to-peer nor blockchain technology [3]. While customers remain anonymous, merchants cannot hide their transactions to circumvent the law. Compared to Bitcoin [1], for example, Taler does not offer a new currency, but is backed by an existing currency such as CHF or EUR. [4] In our experiment case, the currency is KUDOS.

1.2. Focus

This thesis investigates how to scale the GNU Taler payment system to handle the transaction rates that might be required by a Central Bank Digital Currency. It describes the findings of experiments in a distributed setup and the resulting changes in GNU Taler with the goal of improving performance and reaching the desired scale. It mainly covers performance experiments conducted in 2022, but builds on experiments conducted in 2021 with support from NGI's Fed4Fire+.

The work is aimed at people interested in the GNU Taler performance experiments, but can also serve as a starting point for anyone who wants to work on further improving performance. Detailed resources required to perform similar experiments can be found in our Git repository at `https://git.taler.net/grid5k.git`.

[1] Bitcoin: `https://bitcoin.org/en/`

The remainder of the report is structured as follows. Chapter 2 provides a brief introduction on the other technologies that exist today and their current performance. In Chapter 3 we provide a summary about the tools which we used to run experiments on the Grid. We describe our overall experimental setup in Chapter 4. Chapter 5 provides an overview of the main achievements during this thesis, while Chapter 6 provides a more detailed analyzation at the experimental findings. Some additional results and findings which are not directly linked to improving the performance of GNU Taler, are outlined in Chapter 7. Plans for future work are discussed in Chapter 8. Finally, Chapter 9 discusses the impact on GNU Taler of the work that has been done.

Note: All URLs seen in footnotes where accessed in the period from March 2022 until June 2022

2. Related Work

2.1. Payment Systems

Central Bank Digital Currency (CBDC) research has been going strong lately. *CBDCTracker* [1] provides a nice overview of which countries are researching which payment systems, unfortunately it's not quite complete as GNU Taler, for example, is not listed anywhere. We won't go into each of these technologies here, but there are at least three that have been discussed frequently recently. They are shown here to compare some key implementation details (such as privacy) and performance with GNU Taler.

2.1.1. Project Hamilton (MIT)

In Project Hamilton, the Massachusetts Institute of Technology (MIT) is implementing a hypothetical CBDC in collaboration with the Federal Reserve Bank of Boston [5]. MIT claims to achieve nearly two million transactions per second with this payment system [6].

Project Hamilton measured its performance with uniform load balancing. However, the Bank of Japan told us that they do not view this as realistic, and that in their experiments they encounter problems especially when the load is high and non-uniform. In the real world, numerous problems have distributions that roughly follow Zipf's Law [2]. In the context of a payment system, this could for example mean that one account receives the majority of payments (e.g., 10,000), while most accounts (e.g., 10,000) receive only one payment. This would result in spiky payment processing rather than an even distribution. We asked a few individuals at central banks if they had data on actual real-world load distributions, but they could not provide us with any. Still, we believe that a Zipf distribution is closer to realistic load situations than a uniform distribution.

Project Hamilton also implements its payment system using the Unspent Transaction Outputs (UTXO) model [6]. This means that the system has to store its UTXO data forever, and keep an ever-growing list of unspent funds (in case the user lost his key or simply did not spend it), and thus could never fully migrate to new ciphers, especially if we assume that they cannot contact each owner of funds and force them to spend it, be it because of the scale of a real-world CBDC or due to privacy restrictions. Having such an ever-growing list of unspent funds is clearly a disadvantage compared to other designs which can use epochs or expiration times to limit storage growth and migrate to new technologies without loosing funds. Especially at the claimed millions of transactions per second, Project Hamilton's main scalability limit might thus be its long-term storage costs.

A public consultation conducted by the European Central Bank revealed that data protection is desired by the majority of the population (43%), followed by security (18%) [7]. Thus,

[1] CBDCTracker: `https://cbdctracker.org`
[2] Zipf's Law: `https://en.wikipedia.org/wiki/Zipf's_law`

privacy is an important aspect to consider when building a CBDC. Project Hamilton decided to consider this feature in a second phase [5]. However, features like privacy are likely to have a significant impact on the architecture and performance, further raising questions about Project Hamilton's ability to deliver the stated transaction numbers in practice.

2.1.2. Chinese Digital Yuan (e-CNY)

Since 2014, China has been researching a potential CBDC called e-CNY. By the end of 2017, commercial institutions began to be involved in the research to further drive development, and by 2021, the high-level design had been completed to the point where pilot programs could be launched in some regions of China [8].

By the end of 2021, the digital yuan had already been used to settle transactions totaling 90 billion yuan. However, this figure could already be significantly higher today, as the system was further boosted by the Winter Olympics in Beijing, where people were able to install and use the electronic wallet [9].

Last year, China published a paper on the progress and research on its digital currency. This paper states that:

> " *E-CNY follows the principle of "anonymity for small value and traceable for high value," and attaches great importance to protecting personal information and privacy. It aims to meet the public demand for anonymous small value payment services based on the risk features and information processing logic of current electronic payment system.* " [8]

However, we need to understand that the definition of anonymity in China refers to the other user(s) of the payment system. With "anonymous" payments, the shop may not learn the identity of the payer, but the e-CNY operator and thus the Chinese state has this information and thus retains full control over its population. When European citizens desired privacy in the ECB's survey, we believe that they indeed aspire towards some kind of level of separation of powers or even citizens being the sovereign, mistrust the state having full transparency over all transactions, and thus would include privacy against the CBDC operator in their stated desire for privacy.

2.1.3. E-Krona (Riksbank)

Sweden has also produced highly-cited research into suitable CBDC implementations. Their project was started back in 2017 with the goal to analyze the need for an e-krona, using a payment system based on R3 Corda which uses a distributed ledger. In 2020, the project entered a more practical phase. This phase focused on continuing the development and testing of the platform, including performance assessments. [10] [11]

While there are no real publications on the performance of their solution, we found that in Phase 1 they measured a TPS of 100, while in Phase 2 they stated that their performance was comparable to card systems [11].

The e-krona system also does not implement cryptographic privacy protections for citizens against the CBDC operators:

> " *Anonymous payments are only permitted to a limited extent according to the current anti-money laundering legislation, and only smaller amounts can be transferred anonymously at present. Accounts may not be anonymous pursuant to current legislation, which should be borne in mind when discussing potential anonymous e-kronor. It is possible that there may be anonymous e-kronor, but they would have a very limited area of use.* " [12]

This statement comes from the first phase of Riksbank's investigation of e-krona. A later report said that all payment systems leave traces, traces that *can* be erased after some basic checks. They also mentioned that privacy is a trade-off with resilience [13]. Again, we can clearly see that privacy is not their biggest concern when implementing the e-krona payment system.

2.1.4. Performance Comparison

System	Reported TPS [3]	Critique
Project Hamilton	1'700'000 [5]	Realism, Privacy
GNU Taler	28'000+	This work
e-CNY	10'000 [14]	Privacy
Visa	1'667 [15]	Privacy
e-krona	100 [4] [11]	Privacy
PayPal	193 [15]	Privacy
Ethereum	20 [15]	Mining
Bitcoin	4 [15]	Mining

Table 2.1.: TPS of different payment systems and cryptocurrencies.

2.2. PostgreSQL

Since we faced bottlenecks with the PostgreSQL database, various tools and articles were used to analyze and improve its performance. There are tools like `pgbench` [5] which can be used to benchmark the database and is provided directly by PostgreSQL. We used this tool excessively to measure the Performance of the database independent of GNU Taler. Furthermore, there are also many third-party tools. Two of them we have used: `pg_top` [6] and `pgBadger` [7]. `pg_top` provides a `top`-like interface to PostgreSQL processes, which we used, for example, to identify problems we were initially having with long-running connections. `pgBadger`, on the other hand, creates a summary of database performance in the form of an HTML report, but requires excessive logging to be enabled if we want full statistics including query analysis. This is a no-go for our high-performance application as things slow down drastically with so much logging. In the end, we only tried it once, as many of the statistics provided can also be visualized in our dashboards by queries rather than by parsing logs.

[3] Some deployed systems can likely handle significantly more transactions than reported.
[4] Measured in phase 1
[5] pgbench: `https://www.postgresql.org/docs/current/pgbench.html`
[6] pg_top: `https://pg_top.gitlab.io/`
[7] pgBadger: `https://pgbadger.darold.net/`

Nonetheless, it still provides statistics that can't be easily visualized in dashboards, such as query execution plans. But there we ended up writing our own Python tool that is better tailored to our use-case.

Apart from the tools, there are also many blog posts and research papers about the performance of PostgreSQL on GNU/Linux that we found informative. A small selection of them are:

- PostgreSQL load tuning: Blog about Linux Kernel and PostgreSQL configuration (RedHat) [16]
- PostgreSQL performance optimization: Wiki about PostgreSQL performance (PostgreSQL) [17]
- PGTune: PostgreSQL configuration generator (Community) [18]
- PostgreSQL performance benchmark: Blog about PostgreSQL 12 performance analysis (EDB) [19]
- PostgreSQL configuration: EDB blog showing some important configurations for PostgreSQL [20]

3. Grid'5000 Introduction

To perform experiments on Grid'5000, knowledge in different tools and applications is required. This background is summarized in the following sections in order to ease the setup for someone who is interested in reproducing our experiments or conducting similar research.

3.1. Grid'5000

The Grid'5000 is a large scale and flexible testbed in which research can be done in form of experiments where researchers are granted exclusive control over individual compute nodes and associated storage, but share the network. It provides about 800 computing nodes which can be used for bare-metal deployment. Normally a default environment is installed which can be used to run experiments. But for a more individual approach it is possible to create custom environments with pre-installed software. Those environments can be created by using `Kameleon`.

3.2. Experimental Overview

The first step is to either select an existing environment provided by Grid'5000 or to create a custom environment with `Kameleon`[1]. Subsequently, jFed is used to allocate resources and run specific experiments.

To run an experiment, a subset of the nodes of Grid'5000 is reserved using a `Resource Specification (RSpec)`[2] and a Kameleon environment is specified for each reserved node. Running the actual experiment in such a configuration is then facilitated using an `Experiment Specification (ESpec)`[3].

3.3. Kameleon

`Kameleon` is a tool which can be used to create customized software appliances. Where software appliances means a complete operating system image with installed tools and libraries required for an experiment. It was designed to make experiments in computer science reproducible [21]. As it is recommended for Grid'5000 research, our experiments are based on it. `Kameleon` takes a set of YAML files which define the tools to be installed and how to configure them. In the case of Grid'5000 there is already a set of default configurations

[1] Kameleon: `http://kameleon.imag.fr/index.html`
[2] jFed RSpec: `https://doc.fed4fire.eu/testbed_owner/rspec.html`
[3] jFed ESpec: `https://jfed.ilabt.imec.be/espec/`

which setup basic environments that are compatible with the grid. These base variants can then be extended with a custom software stack for individual experiments.

To build such an environment `Kameleon` and its dependencies need to be installed. A Grid'5000 template can then be instantiated in the following way:

```
$ kameleon repository add grid5000 \
  https://github.com/grid5000/environments-recipes.git


$ kameleon new debian11-custom grid5000/debian11-x64-min.yaml
```

This creates a new environment description based on a Debian 11 compatible with the nodes in Grid'5000. The created base file `debian11_custom.yaml` can then be extended to install the required software. Once the configuration is done the image can be built:

```
$ kameleon build debian11-custom
```

It is also possible to provide build time variables to customize the build. We use this feature to pass commit hashes which will be checked out before the GNU Taler binaries are built:

```
$ kameleon build debian11-custom -g variable-name:variable-value
```

Once the build has finished, the compressed image and the automatically generated environment description (`.dsc`) must be uploaded to the grid so that it can be referenced in an experiment description, such as jFed's `RSpec`. The command below shows an example of how to copy the files to the *Lille* site from Grid'5000.

```
$ scp debian11-custom.{tar.zst,dsc} <USER>@access.grid5000.fr:lille/public
```

3.4. jFed

`jFed` is an application which provides a graphical user interface (GUI) to various testbeds, including Grid'5000. Its primary operations are the reservation of resources and experiment control. There are two main concepts which need to be understood in order to start an experiment with jFed. These two aspects will be discussed below.

3.4.1. Resource Specification (RSpec)

In order to reserve nodes on the various tesbeds, an XML based configuration called a Resource Specification (`RSpec`) is requried. This can either be created via the `Topology` (GUI) or the `RSpec` (Text) editor in jFed. Basically the configuration must contain which (types of) node(s) should be requested for a job and which environment should be installed on each node.

The example below (Listing 3.1) shows the requested node from the `Dahu` cluster and the node's name for the experiment ("`DB`"). Once this node is allocated successfully, our custom image's description (`.dsc`) generated by `Kameleon` will be read (`disk_image name`) and the image will be downloaded and installed.

```
<rspec ...>
  <node client_id="DB" exclusive="true" component_manager_id=
    "urn:publicid:IDN+am.grid5000.fr+authority+am">
    <sliver_type name="raw-pc">
      <disk_image name=
        "http://public.lille.grid5000.fr/~bfhch01/debian11-custom.dsc"/>
    </sliver_type>
    <hardware_type name="dahu-grenoble"/>
    <location xmlns="http://jfed.iminds.be/rspec/ext/jfed/1"
       x="156.0" y="70.0"/>
  </node>
</rspec>
```

Listing 3.1: RSpec example snipped reserving a node in the dahu cluster with our custom image located in the public directory on Grid'5000.

3.4.2. Experiment Specification (ESpec)

After the testbed has allocated nodes to the experiment and deployed the specified environments, experimenters can log into the reserved nodes using `ssh`. However, this may be inconvenient if the number of nodes is large, and interactive use may not be ideal if the goal is to produce reproducible experiments.

The `Experiment Specification` is an additional YAML configuration file which can optionally be used to control experiments using jFed. It allows the researcher to control experiments by providing a way to upload additional data and to execute commands or scripts on the allocated node(s). It can also provide crucial metadata about a successfully allocated experiment (in the form of a file upload to the grid), such as the mapping from jFed node names to the actual nodes reserved on Grid'5000.

An example for a simplistic `ESpec` control is:

```
version: 1.0-basic
rspec:
  - bundled: taler.rspec
execute:
  - direct: |
    #!/bin/bash
    echo "Hello from the experiment"
    nodes: ["DB"]
```

Listing 3.2: Simplistic ESpec which will reserve the resources specified in the RSpec `taler.rspec` and additionally output the "Hello from the experiment" message on the node with the name "`DB`". Other execute methods could also include scripts to be called.

An ESpec can be run directly in jFed, which will try to allocate nodes from the testbed and will start the uploads/executes once the allocation was successful. Alternatively, it is possible to load the RSpec first and then subsequently execute the ESpec against the already allocated nodes.

4. Experiment Setup

All Taler-specific resources mentioned in the following sections can be found in the `grid5k` Git repository on `git.taler.net`. In particular, that repository includes all instructions necessary to recreate an experiment in the grid.

4.1. Environment

Running a job inside the grid requires an environment which can be deployed on the nodes. For an easy start, a set of default environments is provided by Grid'5000. But since installing applications at each experiment start is resource and time-consuming, custom ones, containing the required applications can be created using some already existing variants[1]. The sections below will describe the environment used for the Taler performance experiments.

4.1.1. Structure

To keep things simple, there is only one environment created for the experiments. This environment has all required tools preinstalled but unconfigured and disabled. They need to be set up and enabled on the corresponding node when an experiment is started. This ensures that even though all binaries are present, only the required ones can use scarce resources of a node.

The base variant used for this environment is a "Debian 11 with NFS". The NFS enables an automatically configured shared directory on nodes located in the same site. This shared directory will also remain accessible when an experiment ends and nodes are released. This is an important aspect, since there is information generated during an experiment which can be interesting for reproducibility or further analysis of problems. An example of such information are the logs created by the Taler applications while running.

It should be possible to re-create an expired experiment as closely as possible at a later time. To achieve this, a commit hash of each Taler binary is required when building an environment. Those commits will then be checked out at build time and the applications which get installed will be this exact version. Thus, anyone could rebuild the environment which was used in a particular experiment and reproduce it. We note that this does not extend to the core Debian system used, which may undergo updates that are not versioned. However, such changes are unlikely to have a major impact on the Taler experiments.

[1] Grid'5000 Environments: https://www.grid5000.fr/w/Getting_Started#Deploying_a_system_on_nodes_with_Kadeploy

4.1.2. Build

The environment can be built locally with a tool called `Kameleon`. Once Kameleon and its dependencies[2] have been installed the build can be started. This process can take quite some time (on a Lenovo X1 Extreme Gen 2 it took approximately 40 minutes for our custom images, which build all binaries from GNU Taler from source) and possibly requires additional user interaction. But since the experiments frequently use the latest Taler binaries built from the latest source, an image rebuild can be needed rather often. To ease this process, the build can be executed automatically with a Docker setup which can also be found in the Git repository. Using this Docker setup simplifies things as installing the dependencies to build the image can be skipped and the image gets deployed to the grid automatically. All that is required is an SSH private key with access to the grid so that the image can be copied there once its build finished.

4.2. Analysis Tools

Aside from the core system, we deployed additional tooling specific to the analysis of the experiments. This includes the tools to collect, ship, analyze and visualize the data generated by the nodes and applications as well as basic utilities to debug the experiments. Due to some prior knowledge as well as their flexibility the following tools were chosen: Prometheus, Loki, Promtail and Grafana. The following sections will explain how these tools operate together in detail.

4.2.1. Data Collection

For data collection inside the grid a combination of Prometheus[3] and Loki[4] is used. The big advantage of Prometheus is that it can monitor nodes as well as applications. Furthermore, it is also possible to write custom exporters which Prometheus can query.

A list of all (popular) Prometheus exporters, can be found on the official webpage. For our experiment, we deployed the Node[5] exporter to monitor each node's resources and activities, as well as the Nginx[6] and PostgreSQL[7] exporters as we were using PostgreSQL for the database and Nginx for the reverse proxy.

Loki in combination with Promtail[8] is responsible for collecting the logs the applications write. To limit resource usage on the experiment nodes, a single Promtail instance is used. Applications will then ship their logs to this instance by using rsyslog[9]. Loki was chosen partially because our team already had some experience with it, but there are also other reasons, such as its performance[10] and the compatibility with Grafana. The main goal of using Loki is not to display the logs in the Grafana dashboards, but to easily analyze problems and

[2] Kameleon dependencies: `https://www.grid5000.fr/w/Environments_creation_using_Kameleon_and_Puppet#Install_Kameleon_and_other_dependencies`
[3] Prometheus Homepage: `https://prometheus.io/`
[4] Loki Homepage: `https://grafana.com/oss/loki/`
[5] Node Exporter: `https://github.com/prometheus/node_exporter`
[6] Nginx Exporter: `https://github.com/nginxinc/nginx-prometheus-exporter`
[7] PostgreSQL Exporter: `https://github.com/prometheus-community/PostgreSQL_exporter`
[8] Promtail Homepage: `https://grafana.com/docs/loki/latest/clients/promtail/`
[9] Rsyslog: `https://www.rsyslog.com/`
[10] Comparing Logging Solutions: `https://crashlaker.medium.com/which-logging-solution-4b96ad3e8d21`

compute additional statistics from the logs. As logs are passed through different abstraction layers, they are also written as plain text files to the Grid'5000's NFS for manual analysis.

4.2.2. Data Analysis

To analyze the data collected by Prometheus and Loki, Grafana[11] is used. It analyzes the data from the two collectors with queries defined in the frontend dashboards. A dashboard specifies how to render subsets of the collected data.

The advantage of Grafana is that own custom dashboards can be created easily, which is crucial to adapt the output to the experiments as the analysis progresses.

Dashboards can either be imported from `grafana.com`'s public dashboard library[12] by specifying an ID, or they can be created manually. The following subsections provide brief descriptions of the dashboards used in the Taler performance experiments. Their detailed panel description can be found in Section A.1.

Overview

The overview dashboard is the entry point of the Grafana instance. It lists all other dashboards which are important to monitor a running experiment.

Transactions

This dashboard visualizes how far we have come to reach our goal of 100'000 TPS. It is capable to visualize the TPS over the whole experiment duration as well as additional statistics about the applications in use. Those are important to understand the effect and the correlation on the progression.

The goal of this dashboard is to create a feeling for the impact the different applications may have on the TPS and which we may need to improve.

Load Statistics

A more detailed or specific look at the performance of the various application nodes in relation to the load they have to handle. This load is usually represented by the number of requests generated from the clients, in this case the wallets. These requests are logged by the Nginx proxy which forwards them to the exchanges. Promtail is then responsible for counting the number of requests by analyzing these logs.

Request Statistics

Request Statistics provides an overview of the time spent on requests to the important endpoints of the exchange. It includes various statistics such as minimum, maximum, average and different percentiles for request times, as well as the total number of successful or

[11] Grafana Homepage: `https://grafana.com`
[12] Grafana Dashboards: `https://grafana.com/grafana/dashboards`

failed requests in a given time period. Most of the data seen in this dashboard is calculated by Promtail, by analyzing the logs generated by the Nginx proxy.

There are also additional panels that show the latency between the most important nodes of the experiment, such as the latency between the exchange and the database, or the one from the proxy to the exchange. Those statistics are generated by the nodes themselves using `ping`.

Logs

Many of the logs written by the applications contain statistically important data, such as request times measured by the proxy or slow queries reported by the database. These logs must therefore be available for processing. Fortunately, as mentioned earlier, this can be done easily with Promtail/Loki. However, since some things sometimes still need to be checked manually, there is this dashboard that provides a filtering function to look for specific entries in these logs. This feature was sometimes helpful for us to analyze the logs faster. Since the number of log lines displayed is limited, more detailed checks have to rely on manual analysis of the log files stored on the NFS.

Database

This dashboard is a combination of the two library dashboards 455 and 6742, plus some custom panels. It contains important information about the database, such as its configurations, transaction states, duration and failures.

Exchange

On this dashboard you can see the metrics generated directly by the exchange. It mainly shows how many signatures/checks are performed per second. But also statistics about the interaction with the Exchange database, e.g. which queries exceed the configured query execution time and how often, or the number of database serialization errors that occur at a given GNU Taler web API endpoint.

Proxy

A fairly simple dashboard that displays the connection statistics of the Nginx proxy. The basic panels are imported from the public dashboard library with ID 12708. However, there are also custom panels that display request statistics, such as the average response times read from the Nginx logs.

Nodes

To detect the resources which might limit the TPS in the experiments, the nodes hosting the applications need to be monitored too. Thanks to the Prometheus node exporter and the public dashboard 1860 this is pretty easy. In this dashboard one can see almost any statistics the kernel provides about the particular node. This includes information about CPU, memory, I/O, network, processes and others.

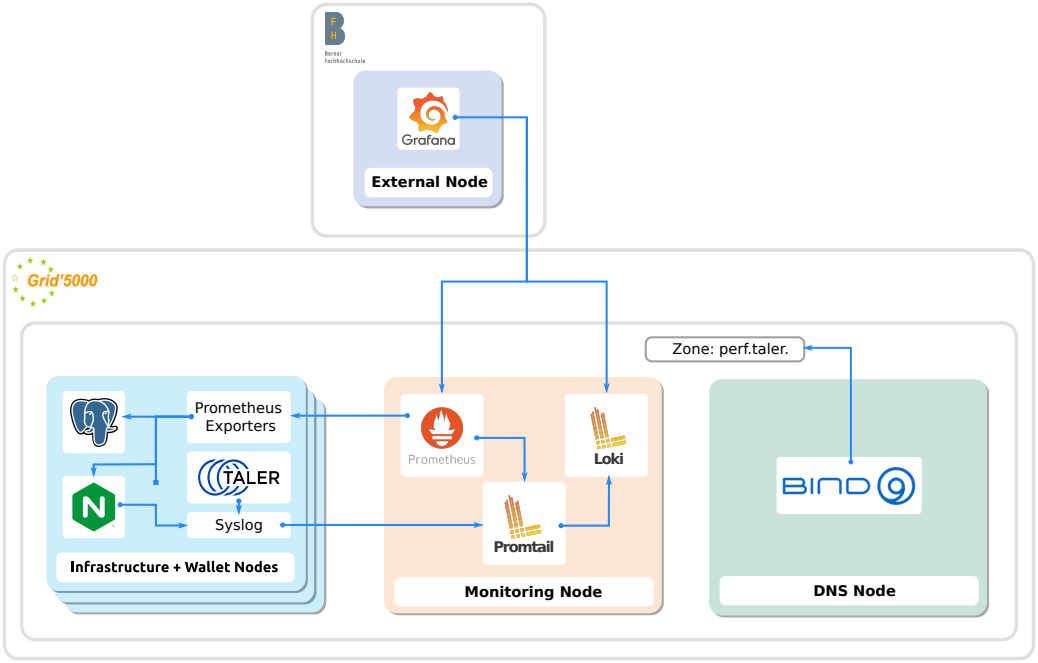


Figure 4.1.: Experiment architecture

4.3. Network System Architecture

Figure 4.1 shows the allocation of nodes inside Grid'5000. One important thing to notice is that Grafana is hosted externally. It needs to be configured with users and dashboards, where dashboards should also be accessible from outside the grid. While Grid'5000 does provide a proxy[13] where services inside the grid can be reached from outside, it would still not be very useful because the URL to access the instance would be different in each experiment. The reason for this is that it is almost impossible to reserve the same specific node for different experiments. So it would be necessary to extract the information manually and passing it to everyone who would like to access the dashboards. This in turn would only be possible for registered users of Grid'5000, since the proxy requests login credentials.

Another reason for hosting the dashboard externally is that it needs to be configured with dashboards and users. While it would be possible to do this at every experiment start via the API Grafana provides, it would still not be efficient, especially as work on dashboards does not otherwise require a running experiment on the grid. Furthermore, externally hosting the dashboard simplifies exporting data that is to be preserved.

However, the challenge we faced here is that Grafana queries its datasources (Prometheus and Loki) and thus needs to know their Grid'5000 proxy URL (which changes for each experiment). Fortunately this can be achieved by using the admin API of Grafana. Through this API we can update the datasource URLs from inside the grid, once the allocated nodes are known.

Figure 4.2 shows the initial situation we created in the first part of this work in 2021.

[13] Grid'5000 Proxy: `https://www.grid5000.fr/w/HTTP/HTTPs_access`

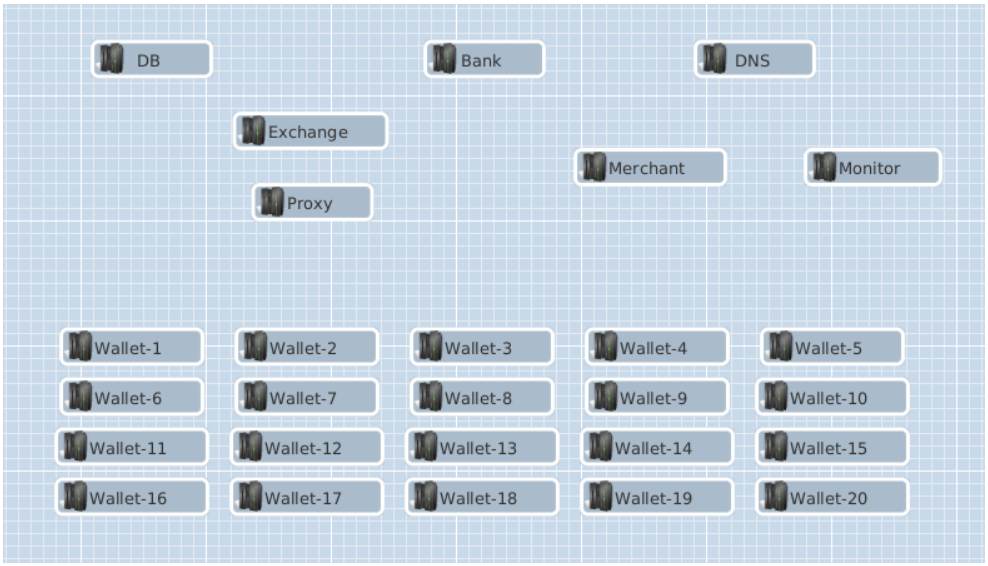


Figure 4.2.: Nodes allocated in the grid, as shown in jFed. This setup was created in the first half of this work in 2021 when we had about 300 TPS only.

There is only one *Exchange* node running all Exchange processes, such as taler-exchange-wirewatch, taler-exchange-httpd, taler-exchange-aggregator, and taler-exchange-transfer. However, the evolution of the load required that the main process (taler-exchange-httpd) run on multiple nodes and the other processes be offloaded to separate nodes. This results in an architecture as shown in Figure 4.3. The current setup allows for any number of Exchange, EProxy (exchange-proxy), Wallet, Shard, and Merchant nodes by incrementing the suffix number.

There are now multiple rspecs in the repository ('additional/rspecs') that have a different setup, for example a database without shards or different number of wallets etc. There are also minimal examples of experiments that are run just to test the setup and not the performance.

4.3.1. Selected Nodes

Since we only had *misc* access to the grid in the second part of our experiments, we had limited resources available. With our access, we were only allowed to use the *default* queue, which does not contain the most powerful nodes.

To achieve the lowest possible latency, we decided to place all backend processes in the same cluster. The most suitable one, offering enough resources but also nodes, seemed to be *Dahu* in Grenoble. This cluster offers 32 nodes with a 10Gbps network connection, 2 times an Intel Xeon Gold 6130 CPU (Skylake, 2.10GHz, 16 cores/CPU), 192 GB of RAM and one HDD plus 2 SSDs per node.

For the clients, we did not choose an explicit hardware type. They are not required to provide much performance, just enough to host some wallet processes. They are automatically assigned by the grid when an experiment is started.

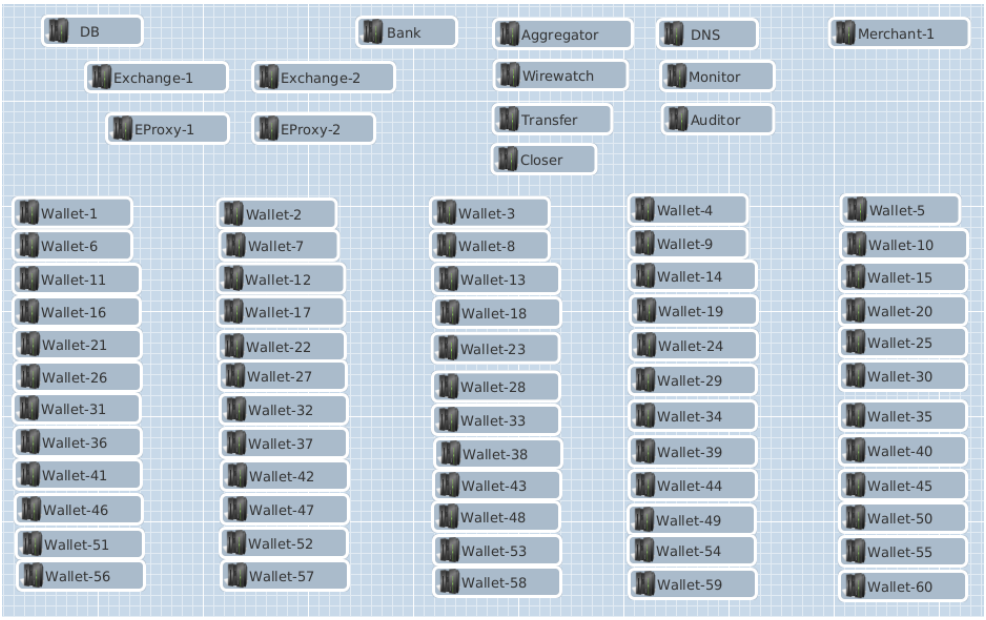


Figure 4.3.: Nodes allocated in the grid, as shown in jFed. This setup resulted from a higher throughput as some bottlenecks were fixed and more requests were issued. Multiple exchange nodes do share their key material via NFS. One is responsible for creating the keys while the others are waiting until the primary one is done.

4.4. Experiment Scripting

4.4.1. Introduction

Ideally, one should be able to run an experiment without needing to perform any manual configuration work. To make things as easy as possible, most of the configuration and setup work was automated using GNU Bash scripts. They in turn require a simple configuration file (*.env*) which sets environment variables specifying details on how to set up the experiment.

Nodes will get a default configuration file and the allocation info, which contains the mapping between the jFed node names and the actual nodes in the grid. Based on the information they get from there they know which role to play. For each of these roles there is then a script which makes the necessary changes to the node configuration and starts the required services.

To automate these setup tasks, jFed's ESpec format is used. It allows uploading different resources, such as bash scripts, which can then be executed in a predefined order and on specified nodes. According to the ESpec documentation *it is preferable to keep ESpecs so simple that a user not familiar with the format, will be easily able to manually execute the experiment using tools that do not support ESpec [22].* This was also the goal while setting up these experiments.

Currently, ESpec is only used to upload the configuration file, the allocation info and scripts. It then executes the `setup` and `run` scripts on each node. Even though uploading is currently part of ESpec, it could be omitted later and instead the scripts that are already in the pre-installed grid5k Git repository inside the environment could be used directly.

The reason ESpec is used to upload the resources is because it is easier/faster to test changes in scripts than if they have to be pushed to Git first and then updated in an allocated experiment. The only part that is not really functional without ESpec is the node allocation info. This info can be uploaded along with the other uploads to get the mapping from jFed to the grid nodes, which is a crucial part for these experiments so that the nodes know who to communicate with. However, other tools provide other ways to communicate such information to the nodes. Only the `setup` script would then need to be extended to support the functionality of this particular script. Essentially, the following steps are performed:

For the most part, the configuration of the services themselves is done statically in files located in the Git repository. However, many of them contain placeholders to create an environment that can be easily modified. For example, the Taler `systemd` service files contain variables that can be used to change the behavior of the binaries when they are configured in the *.env* configuration file before ESpec is run.

4.4.2. Procedure

When starting an experiment, the *direct* execution of ESpec does some necessary work, such as moving the uploads from a temporary directory to the desired location. This step is done because ESpec does not allow overwriting directories during upload. Then the `setup` script is (and must be) run first. This copies the configuration files to the correct locations and changes the files that are common to all nodes. It also cleans up the shared NFS storage and (re)creates the directories needed for the experiment (e.g. the syslog directories). If configured, `setup.sh` also rebuilds the applications such as GNUnet and Taler from source before proceeding. This way, the rather long build (and copy) times for the Grid'5000 environment can be bypassed. Then, various environment variables are set up that are needed by either the scripts or the applications. Adding these variables to */etc/environment* also makes them available in each new shell. Finally, it also configures the DNS stub resolver on each node and the DNS server on the DNS node so that it is ready for the `run` script. The `run` script will (must) be started after that. It determines which role has been assigned to each node (specified in the jFed node name, which can be read from the allocation info) and registers that node with the DNS. Finally, it calls the correct script to set up the specific configurations and the required applications for that role. For convenience, we also perform other additional setup steps that are similar on each node (if the setting is desired), such as setting up the node-specific log directory in the NFS or enabling the RTT measurement to a particular node.

A role-specific script that should also be mentioned here is the one for configuring the monitor node. This one is special because the node needs to be known to our external Grafana instance in order to get the data from Loki and Prometheus. A question came from the Grid'5000 staff about how we achieved that Grafana can query the (with each new experiment different) node from the outside. As mentioned earlier, Grid'5000 provides a proxy into the grid through which ports such as 80/8080 and 443/8443 can be reached from the outside. Therefore, we instruct Loki and Prometheus to listen on one of these ports. We have two initially configured data sources (one Prometheus and one Loki) in the Grafana instance. However, the proxy requests credentials, so a registered user must add them to the basic auth configuration of the data source to make them query-able by Grafana. These data sources are then retrieved on each ESpec run and their URL is changed to match the new Grid'5000 proxy URL. All of this can be done through Grafana's admin API, which is also why our configuration file requires an admin API token. Since 2020, an issue has been opened

to allow changing datasources without admin privileges, as *everything* can be accessed with this token [14]. The source code for this particular part is shown in Listing 4.1.

```
21 # Update a data source on the external grafana instance
22 # $1: Datasource name (configured in .env)
23 # $2: Port where the datasource is listening
24 #     (http/http8080 or https/https8443)
25 # See  https://www.grid5000.fr/w/HTTP/HTTPs_access
26 function update_datasource() {
27   # Get the id of the datasource to update
28   ID=$(jq --arg name "$1" \
29           '.[] | select(.name == $name) | .id' \
30           ds.json)
31
32   # We require e.g. dahu-2.grenoble.<PORT>.proxy... as domain
33   # Extract dahu-2.grenoble from 'hostname'
34   HOST=$(hostname | cut -d "." -f 1,2 -)
35
36   # Replace the datasource's URL with our proxy domain
37   jq --arg url "https://${HOST}.${2}.proxy.grid5000.fr" \
38      --arg name "$1" \
39      '.[] | select(.name == $name) | .url = $url' \
40      ds.json | \
41   curl -X PUT -k -f -d @- \
42        -H "${AUTH_HEADER}" \
43        -H "Content-Type: application/json" \
44        -H "Accept: application/json" \
45   "${GRAFANA_API}/datasources/${ID}"
46 }
47
48 # Update the external grafana instance and tell it
49 # about the node which hosts our datasources
50 # If GRAFANA_HOST or GRAFANA_API_KEY are empty this
51 # step is skipped - requires admin level api key
52 function update_grafana() {
53   if [[ -z ${GRAFANA_HOST} || -z ${GRAFANA_API_KEY} ]]; then
54     return 0
55   fi
56   AUTH_HEADER="Authorization: Bearer ${GRAFANA_API_KEY}"
57   GRAFANA_API="${GRAFANA_HOST}/api"
58
59   # Retrieve the initially configured datasources
60   # and safe them to a file
61   # to be used later in update_datasource
62   if ! curl -k -f -H "${AUTH_HEADER}" \
63     "${GRAFANA_API}/datasources" \
64     -o ds.json ;
65   then
66     echo "Failed to retrieve datasources from Grafana"
67     exit $?
68   fi
69
70   update_datasource "${PROMETHEUS_DATASOURCE_NAME}" \
71                     "${PROMETHEUS_G5K_PROXY_PORT}"
72   update_datasource "${LOKI_DATASOURCE_NAME}" \
73                     "${LOKI_G5K_PROXY_PORT}"
74 }
```

Listing 4.1: Part of *experiment/script/monitor.sh*, which is located in the grid5k repository. Through these functions, the URLs of the data sources stored in our external Grafana instance are updated based on configuration values.

[14] Grafana datasource API permissions: `https://github.com/grafana/grafana/issues/24544`

4.4.3. Utility Scripts

In our experiments, it is also important to see what effect running multiple wallets in parallel has on the performance of the other nodes (e.g., if they scale linearly). To achieve this, it is important to be able to add and remove wallet clients while the experiment is running. While this is possible with jFed's *Multi-Command* feature, it is tedious to select only the wallet nodes, especially when there are as many nodes as we are using. Therefore, we created a separate script that can be called on any node, provided the node has been configured at least once by running ESpec. The script can be invoked by the command `taler-perf`. With the `taler-perf start <TYPE>` and `taler-perf stop <TYPE>` commands, the experimenter is able to add and remove wallet clients, as well as start and stop wirewatch and and exchange processes in a running experiment. The latter is a bit more complicated, since the proxy and the monitoring node also need to be informed and updated when exchange processes are added or removed (because each exchange-httpd process listens on a different port - see Section 4.4.4). Therefore, it is also easier if it is automated by a script that can be triggered instead of the *Multi-Command* function.

In addition, this script can also be used to rebuild selected binaries from the source code using the `taler-perf rebuild` command. This will update and rebuild the binaries on each node of the experiment (similar to `setup.sh`).

4.4.4. Systemd Templates

We had to find a way to efficiently run multiple processes of the same binary. This is especially necessary for the various exchange binaries and wallets. Initially we had considered using GNU Parallel [15], but quickly discovered that systemd [16] provided a more convenient way to do this, including starting and stopping processes on the fly without having to search for them. It was also a simple task to accomplish this, since GNU Taler was already using systemd as a service manager. By adding a '`@`' character to the name of a systemd *unit file*, it becomes a *unit template* and can be used to start multiple independent processes of the same application. Everything that comes after the `@` character when starting such a unit is interpreted as an argument and can be used to pass it to the binary, for example. For taler-exchange-httpd, we used this functionality to specify the port on which each process should listen. [17] In the template file itself, format identifiers beginning with a '%' character can be used to refer to these arguments (%i and %I) and other parameters [18]. The modification of the unit files is described with an example in Listing 4.2 and 4.3.

[15] GNU Parallel: `https://www.gnu.org/software/parallel/`
[16] Systemd: `https://systemd.io/`
[17] This is necessary because each exchange-httpd process exports independent metrics to Prometheus. For example, if we were to use port 80 only, we would not get all the metrics in a metrics request, but only those of a single and probably different process each time. Note that you would certainly use the same port in production, but retrieving the metrics from such a setup requires further work.
[18] Systemd Specifiers: `https://freedesktop.org/software/systemd/man/systemd.unit.html#Specifiers`

```
[Unit]
Description=Taler Exchange Socket
PartOf=taler-exchange-httpd.service

[Socket]
ListenStream=80
Accept=no
Service=taler-exchange-httpd.service
SocketUser=taler-exchange-httpd
SocketGroup=www-data
SocketMode=0660

[Install]
WantedBy=sockets.target
```

Listing 4.2: `taler-exchange-httpd.socket` unit file before changing to templates, this (socket) service could be started with `systemctl start taler-exchange-httpd.service`.

```
[Unit]
Description=Taler Exchange Socket at %I
PartOf=taler-exchange-httpd@%i.service

[Socket]
ListenStream=%i
Accept=no
Service=taler-exchange-httpd@%i.service
SocketUser=taler-exchange-httpd
SocketGroup=www-data
SocketMode=0660

[Install]
WantedBy=sockets.target
```

Listing 4.3: `taler-exchange-httpd@.socket` template file. This unit takes the port number to listen on as an argument so that we have independent ports for each exchange-httpd process. `%i` and `%I` are used to reference the argument after @ in the unit which is started like this: `systemctl start taler-exchange-httpd@80.service`. (This file can be found in the g5k repository under 'configs/usr/lib/systemd/system').

4.5. Persistence and Recovery

At the very beginning, we only had the logs of an experiment on the Grid'5000 NFS. They could then be downloaded after an experiment was over, in order to perform further analysis or to keep them if needed. In addition, we created Grafana snapshots of the important dashboards, for example, to create figures for the report. However, with Grafana, it can sometimes be a bit tedious to persist all panels, as sometimes not all of them are included in the snapshot. Also, you always have to be ready to take a snapshot before an experiment expires, otherwise the data, except for the logs on the NFS, will be lost when the nodes are released. So we had to think about how to keep the data in a more efficient way so that it could be viewed at a later time and also by someone else.

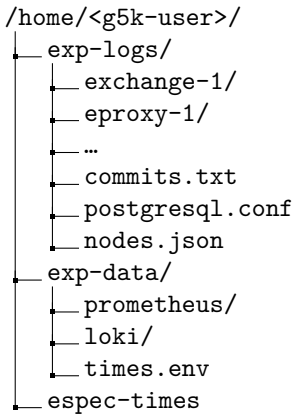


Figure 4.4.: Grid'5000 NFS at the Grenoble site. While *exp-data* and *espec-times* are present only there - respectively only where the `monitor` node is allocated, *exp-logs* is also present at other sites where nodes are allocated for our experiment. *exp-logs* contains all logs output directly from `syslog` and additional information, such as the used configuration in PostgreSQL for the master node or the commit hashes of the exchange binaries. In *exp-data*, we store the Prometheus snapshots, Loki data, and the required timestamps for experiment recovery (*times.env*).

4.5.1. Persistence

We found that Prometheus provides built-in snapshot functionality through its API [19], whereas Loki does not. Instead, we have to back up all the data created by Loki, which means almost all the logs again, but in a different format. This means that we had to choose a different strategy for each datasource. For Prometheus, we set up a service called `taler-prometheus-backup.service` that calls the script `prometheus-backup.sh` every two minutes and creates a snapshot that is stored on the NFS. Loki, in turn, is configured to write all data, including the WAL, directly to this location. In addition to this data, the start and (Prometheus-) snapshot times are also stored there, as these are needed for recovery. The directory structure on the NFS is shown and explained in Figure 4.4, these files should be persisted for later analysis.

To persist this data, the `additional/persist.sh` script can be used once an experiment is finished. The `-b` option collects the persisted files from each site and creates a compressed archive, which is then copied to the local machine and stored in `additional/archives/`. It is a good idea to clean up the NFS storage afterwards using the `-d` option. Although the directories will also be cleaned up once a new experiment starts, it is still a good idea to run the command after the backup. Since the data is deleted only at the locations where nodes are assigned, it could happen that data from an old experiment is included in a backup if the second experiment was run at a different location. These two steps are performed independently to prevent data from being lost accidentally or due to an error.

4.5.2. Recovery

To make the recovery as simple and efficient as possible, it is based on a `Docker` setup. This offers many advantages, such as the fact that no requirements other than Docker need to be installed and that there is always a working setup on different machines due to the

[19] Prometheus Snapshot: `https://prometheus.io/docs/prometheus/latest/querying/api/#snapshot`

same configuration, operating system and versions of all applications. The setup includes Grafana with the *Image Renderer plugin* [20] (to be able to create PNG plots of the recovered experiment), Prometheus and Loki.

A saved experiment can be automatically recovered using the `run.sh` script which can be found in the directory `additional/recovery`. This script takes as argument an archive or directory containing the experiment data to be recovered. It extracts this data and configures Grafana with the "Taler Performance Dashboards" from `additional/grafana` to display the data at the time of the experiment. This can be accomplished with the persisted timestamps. Otherwise, the user would have to search back through history to find the experiment data in the UI. Once configured, the experiment dashboards can be inspected on `localhost:8080` in the same interactive way as in a live experiment, i.e. one could also configure new panels afterwards to create all kinds of different plots that were not (yet) available in the live experiment dashboards.

[20] Grafana Image Renderer: `https://grafana.com/grafana/plugins/grafana-image-renderer/`

5. Performance Timeline

This section summarizes the performance improvements of GNU Taler during this work. Most of the results shown here are discussed in more detail in Chapter 6.

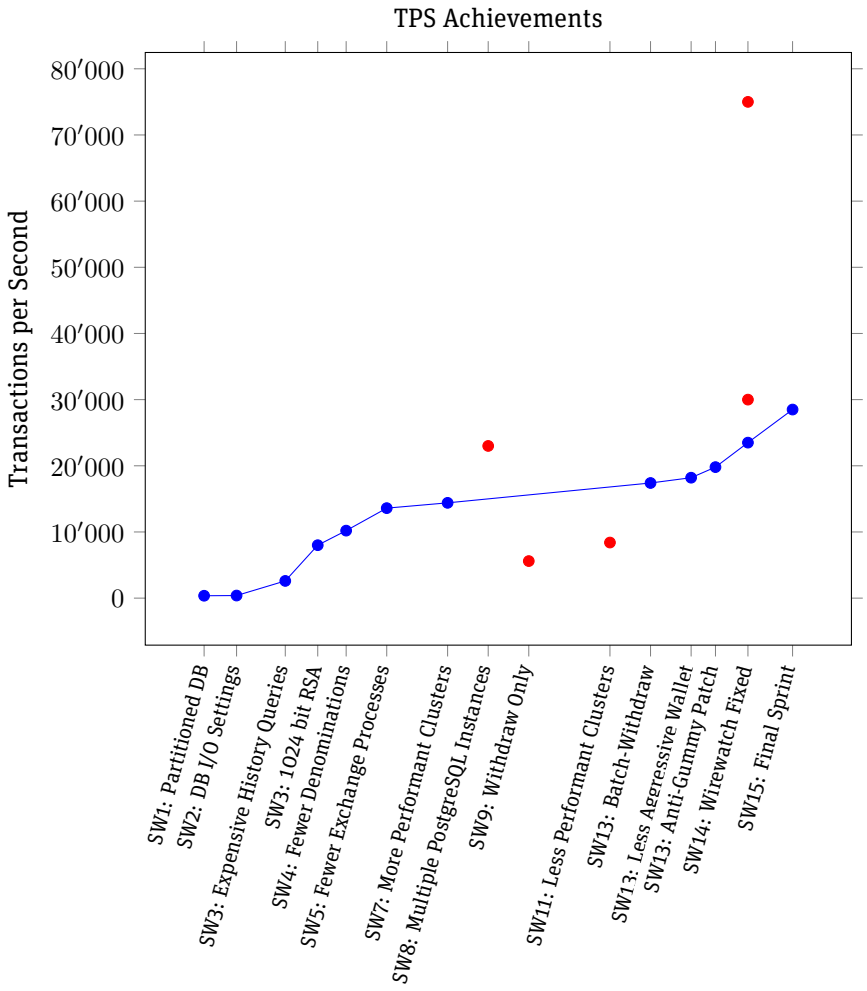


Figure 5.1.: GNU Taler TPS (withdrawals + deposits per second) achievements during the thesis. The x-axis shows the Semester Week (SW) and the changes made. The red dots mark achievements that are not so important for evaluating the performance of GNU Taler, but show what we have achieved with various attempts.

Partitioned DB

Since the end of project two (the work preceeding this one) it was clear that the database would have to be partitioned/sharded at some point. On one hand, this allows us to decrease serialization errors, and on the other hand, we can distribute the computing load horizontally.

DB I/O Settings

We had already reached the I/O limit at 300 TPS on the database node, at which time we still assumed that the drives could be the problem. Some experiments with the WAL on different hard disks quickly showed that this significantly affected the IO load. Fortunately, there are some configuration parameters for PostgreSQL that can be used to optimize IO performance. Some of these are clearly unsafe (more in Section 6.4.5), but may be useful for running experiments on the Grid'5000 platform since we have no control over the type and number of disks installed on the nodes.

Expensive History Queries

Some of the database queries that were issued each time a request was made to the `/reserves` or `/withdraw` endpoints had a serious negative effect on the overall database performance. They increased response times, serialization errors, and consequently CPU load. Originally, these queries created a total of 200 slow queries per second with a duration of about 500ms each, meaning they caused 100s of computation time on a 64-core system per second.

What was most striking is that the queries were technically not required at all in a normal withdrawal. They computed and returned the full transaction history, when only the current balance was required. For our benchmarks, they could thus be fully removed by modifying the database to track the reserve balance with the reserve. However, even then they remain necessary during error handling, especially in case of disputed balances. Thus, we still optimized these queries to ensure that they would not become an attack vector for DoS/DDoS attacks, even though they are not relevant for the measured scenario.

1024-Bit RSA

Once the expensive queries were addressed, we soon discovered that the exchange's CPU was the next bottleneck. Fortunately, we were able to identify the problem by running a simple `top` on the node, where the RSA crypto workers took up about 70% of the CPU. To investigate cryptographic performance, we ran various RSA benchmarks on the following hosts:

`Notebook`: Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz
`G5K-Dahu`: Intel(R) Xeon(R) Gold 6130 CPU @ 2.1GHz

Which resulted in the following number of signatures per second (single core):

Keysize	2048	1024
Libgcrypt	240	877
OpenSSL	2'020	13'434

Table 5.1.: Notebook RSA performance

Keysize	2048	1024
Libgcrypt	201	813
OpenSSL	1'832	12'237

Table 5.2.: G5K-Dahu RSA performance

Considering the performance results, we changed the RSA key size to 1024 bits and achieved about three times more TPS with the same number of wallets. Thus, changing the key size also resulted in a major benefit to the wallet execution time, which before was also a concern for our experiments.

Naturally, running with 1024 bits would be a concern for production. However, in production, wallet scalability is not as critical as in the experiments here, as a single device would not be used to simulate thousands of wallets. On the exchange side, we found that we are far away from network or storage bandwidth concerns, so here only the CPU performance would change. However, we found that the OpenSSL library is about 8 times faster than the Libgcrypt library used in our experiments. Thus, the increased cost from RSA 2048 should be more than offset by switching to OpenSSL.

Fewer Denominations

The reduction to 3 available denominations instead of 14 and reducing the lookahead period (for which keys are generated into the future) resulted in a significant performance improvement for the wallet clients: these changes lightened the load on the wallets to such an extent that they became sufficiently fast to generate the envisioned load for all of our experiments. We note that in practice, 10–20 denominations are quite realistic, but as before a wallet would have significantly more computational resources available in a real-world deployment compared to when it is used to drive our load-testing. Nevertheless, optimizing the wallet code to work well with a large number of denominations and a large lookahead period was identified as a direction for future work.

Fewer Exchange Processes

After running some PostgreSQL benchmarks with our configuration, we noticed that the number of (database) TPS decreases when more clients are connected to the database. When we reached 10k TPS in Taler, there were a total of 160 Exchanges connected to the database, evenly split between two nodes. After we lowered the number to 40 per host, we reached 13.6k TPS and our database TPS maximum of 65k.

However, as the TPS increased, we again noticed a shift in the optimal number of clients for best performance. In the end, the best configuration was 200 exchanges for a partitioned single-node database, while 80 exchanges was still best for sharding. However, due to these observations, we expect this numbers to change even further.

More Performant Clusters

We tested many of the nodes we had access to with our access level for performance with PostgreSQL, but except for Neowise and Yeti, none really showed much better performance

over the Dahu cluster we started with. Nodes in these clusters were typically able to achieve 85-95k TPS in PostgreSQL. Running on Neowise, we achieved 14.4k TPS (with about 80k TPS in PostgreSQL), which is not much of an improvement over 13.6k on Dahu. We also got the same TPS on the Yeti cluster. In fact, in some other experiments with the Dahu cluster, we were also able to achieve 14k+. So these fluctuations might have been caused by environmental conditions. We note that we rarely re-ran the same experiment many times to stay within the utilization limits imposed on us when using Grid'5000.

Multiple PostgreSQL Instances

To test which resource might be limiting our performance in PostgreSQL, we ran some dedicated experiments (Section 6.4.9). One of them was to run two independent PostgreSQL instances on the same node, with independent exchanges associated with them. This is not useful for performance evaluation of GNU Taler in terms of single exchange TPS, but showed that the server hosting the database is capable of taking more load. Interestingly, PostgreSQL's 65k TPS were almost evenly distributed between these two instances, while both Taler instances were able to handle almost the same amount of TPS as they did when they individually reached 65k TPS on PostgreSQL. When we ran the same experiment with three PostgreSQL instances and exchanges, we again only achieved a total of 23k TPS in Taler (with the PostgreSQL TPS evenly distributed between the three instances), but the system not being maxed out in any obvious way.

Less Performant Clusters

To explain the discoveries we made when running multiple PostgreSQL instances on the same node, we ran an experiment using a lower performing node, one that achieved about 41k TPS with `pgbench` in PostgreSQL. This was the cluster called *Chiclet*. However, in Taler we again achieved only 8.4k TPS, while the CPU on the nodes was consumed about the same as on the more powerful nodes (e.g., the db node also had only 25%). Thus, at this point, we were again left with no conclusion or explanation for the bottleneck.

Withdraw Only

When we could not reach more TPS in a full experiment, we decided to do independent audits of the operations and started an experiment where we only did withdrawals. We reached the limit at 5.6k, which given the rate of withdrawals to deposits (almost 1 to 1), was not far from the 13.6k in total. We thus concluded that the withdrawals were likely a key part of the bottleneck.

Batch Withdrawal

Because we believed and saw that the TPS in PostgreSQL was limited, we tried to minimize the number of transactions required by combining multiple small operations into larger ones. For the withdrawal, this meant that we combined several individual requests to */withdraw* that processed a single coin, and thus made individual transactions on the database, into one large batch that would withdraw all the coins at once. As serialization errors were no longer an issue, we also made an unrelated change to the refresh endpoint where we

combined multiple transactions into one larger transaction. With these changes, we were able to achieve a peak TPS of 17.5k.

In terms of the withdraw-only experiment, using batch withdrawals improved the situation modestly, to about 6.2k TPS. Since we had more detailed dashboards at that time, we saw that when we reached the maximum TPS in Taler, we also had about 6k withdrawals per second. This implied that the withdraw bottleneck was not really resolved by batching, and that even if we were not constrained anywhere else, we would not be able to get any further until the withdrawal bottleneck was resolved.

Less Aggressive Wallet

During the first attempts with batch withdrawals, we noticed that there were some requests that were not idempotent because a conflict was reported (with HTTP status code 409). It was initially unclear why, as this would mean that the same reserve was used multiple times, which the benchmark should never do. However, we found that the problem was with the wallet client implementation. When wallets did not receive a response for 200 ms, they tried the exact same request a second time! While in sequential withdraws this was not noticed, as the implementation had some logic to handle idempotent requests. The draft batch withdraw implementation did not handle this case at the time, as we wanted to assess its performance impact. Increasing the wallet timeout to 1s had several effects on the experiments. Most importantly, this resulted in an increase in our maximum TPS to 18.2k.

Anti-Gummy Patch

Ultimately, we found that the wirewatch implementation was the cause for the constantly high number of transactions per second in PostgreSQL somewhat independent of the actual transaction load, because it was adaptive: When the database was fast, wirewatch would do multiple smaller insert transactions, and if the database was slow it would instead do a single large batch insert. While batching based on load is actually desired, wirewatch reduced the batch size so aggressively that Postgres was basically always at peak load if wirewatch had any significant amount of work at all. We addressed this by forcing wirewatch to sleep for a few milliseconds whenever it had done an insert transaction below the maximum batch size.

Immediately, the database TPS dropped substantially below the 65k TPS and became more proportional to the Taler TPS.

Wirewatch Fixed

Nevertheless, the withdrawal bottleneck briefly persisted. To diagnose it, we ran an experiment using separate wirewatch processes that were responsible for different banks, with each bank having a different account for the same exchange. Using this setup, we achieved a new maximum TPS of 19.8k.

We concluded that the withdrawal process was limited to about 6k withdrawals per second because of the wirewatch process, as it is responsible for creating reserves in the database for withdrawal transfers received from the bank. Performance did not improve regardless of what shard sizes were used or how many processes were started. It turned out that the

problem was a regression in the sharding logic. Due to some execution paths, processes could lock shards way into the future and wait there for transfers, thus not handling pending operations for a while, effectively rendering the desired parallelism ineffective. After we fixed this issue, we ran the withdrawal experiment again and achieved 75k withdrawals per second (batch-withdraw), with the database performing about 15k TPS per second, far from the historical limit. Unfortunately, we have not yet been able to clearly determine why we could not achieve more withdrawals per second, as there was no other obvious bottleneck either.

We then also ran a complete experiment as usual, where we hoped to achieve much more as the 6.2k bottleneck in withdrawals was removed, but we only achieved 23.5k TPS (using batch-withdraw, reaching about 60k DB TPS), which is only a moderate improvement.

Final Sprint

In the end, we tried to tweak a few small things and were finally able to increase performance to 28.5k. The changes included several things, like increased number of exchange processes or halving the number of required database transactions in deposit. Unfortunately we were not yet able to identify which of these was most responsible for the improvement.

Peeking Into the Future

To identify further bottlenecks, we ran an experiment without refreshes. This is an expensive operation, but necessary to exchange dirty coins that were not fully spent for fresh ones to achieve unlinkability between different payments. In this experiment, the wallets withdrew a single coin in one iteration and deposited it directly, eliminating refresh. We achieved 30k TPS in the process, which is not much more than in our usual experiments where around 4000 refreshes per second happened (with 23.5k withdrawals+payments per second).

Based on our results so far, we believe that there are now still two major bottlenecks: latency, which we could only tackle by running more exchanges in parallel, but as we have seen, the number of optimal connections has changed several times, which makes it difficult to clearly identify the cause. We also still assume that the second bottleneck is the database server, which we believe is somehow ‘limited’ by the amount of TPS, since we still reach about 65k in many experiments. Here it might be helpful to implement some of the queries natively instead of using them with prepared statements or stored procedures.

6. Performance Results

Previously, the Taler system had never been benchmarked in a distributed setting: during previous benchmarks, the respective components were always running on the same node together. Also, only the exchange component had been subject to benchmarking, using as a client a simplistic wallet simulator written in C only for this purpose.

In our Grid'5000 benchmarks, we wanted to gain insights into how Taler would perform if we distributed the components over the network and used the actual production wallet logic (written in TypeScript) as it may exhibit different request patterns compared to the simplistic client simulator. Furthermore, we wanted to separate the database server from the REST front-end (which runs the cryptographic operations), moving towards a more "large scale" real-world deployment with many clients and multiple frontend and backend systems.

In doing so, we discovered a substantial number of new and sometimes unexpected issues. We will present some of the more interesting findings and possible resolutions in the following sections.

6.1. Single System Performance Baseline[1]

We started our work by optimizing the Taler exchange on a single system, measuring the various operations individually. For this, an in-memory real-time gross settlement (RTGS) emulator was written that allows us to pretend to execute bank transactions. The Taler aggregator, transfer and wirewatch tools were then updated to support load sharing where each tool could be run using multiple processes, each taking a range of the queries to be handled.

We then tested incoming wire transfers (wirewatch), outgoing wire transfers (transfer) and aggregation of transactions (aggregator) using this setup. Table 6.1 summarizes the transaction rates observed on an AMD Threadripper 1950 and an unoptimized PostgreSQL database (running on an Intel Optane SSDPED1D280GA).

Operation	TPS	Shards
Aggregator	33k	16
Transfer	62k	8
Wirewatch	50k	24

Table 6.1.: Single-host benchmarks for the Exchange RTGS integration.

We note that on the same system, Florian Dold and Christian Grothoff had years earlier

[1] Unlike the other experiments reported on in this thesis, the reported single-system experiments were done by Christian Grothoff before the start of the thesis.

benchmarked the main exchange logic to be able to handle about 1'000 TPS when running a trivial C client against the exchange on loopback.

6.2. Introduction

Note that most of the performance results shown in this section were measured without the use of batch withdrawals and without the aggregator, while using 2 partitions. If we used a different configuration, it is explicitly stated in the relevant section. For more information on why we excluded the aggregator, please consult Section 6.4.

Please also note the following: The results shown here were measured in relatively short periods of no more than two hours, since we were unable to reserve nodes on the Grid'5000 for a longer period (except on weekends or at night). Furthermore, in longer experiments storage capacities could also be exceeded as available disk space on Grid'5000 is limited. While we do not expect any surprises, we leave conducting longer experiments to see how Taler behaves during longer deployments to future work. Most results were also measured when TLS was disabled, as at higher load (20k+ TPS) 4-5 Nginx nodes are required in the Dahu cluster to process the amount of transactions. However, this should not affect performance as the load can be easily distributed by adding more reverse proxy nodes.

As described in Chapter 5 we reduced the amount of denominations to improve the performance of the wallet clients. After this change we used only the following denominations for all our experiments:

- KUDOS:1
- KUDOS:4
- KUDOS:8

It is almost certain that there would be a much different combination in a real deployment, but this should not affect the performance of the exchange in any significant way. For example, in this configuration we withdraw 3 coins to get 20 KUDOS $(2 * 8 + 4)$, while a more canonical use of 1, 2, 4, 8 and 16 KUDOS denominations would typically withdraw a 4 and a 16 KUDOS coin.

In our benchmark scripts `bench1.ts` and `bench3.ts`, we then calculated the amount of KUDOS to be withdrawn by $(deposits + 1) * amount_{deposit}$, statically configuring the amount for payments to 10 KUDOS. For 10 deposits, this would mean that 110 KUDOS would be withdrawn and then 10 times 10 KUDOS would be deposited. The number of deposits itself was randomly chosen between 1 and 20 and passed as an argument to the benchmark script. Thus, we had different amounts from 20 to 210 KUDOS to withdraw and thus also different combinations of denominations for the coins, which in the end also affects the deposits and especially the refresh operations. For example, for 20 KUDOS the system would withdraw $2 * 8 + 4$, while 50 the system would use $5 * 8$. Paying 10 KUDOS with $2 * 8$ would result in the refresh process generating $2 * 1 + 4$ in change, while paying with $8 + 4$ would result in $2 * 1$ in change.

What is a Transaction

There is always the question of which operations in a digital payment system count as one transaction. Normally, we consider a withdrawal from an ATM or a payment in a store, including receiving change, as one business transaction. For banking systems facilitating the

transfer, the debit and credit operations may be separate operations, and with split payments the situation may become even more difficult.

For our experiments, we wanted to be able to relate the key operations on the HTTP REST endpoints to the transaction counters. However, simply counting HTTP REST endpoints would not allow us to reasonably compare batch withdraw and withdraw. Thus, we decided to always count the number of coins withdrawn and deposited as transactions. As the need for refresh (change) in Taler may be surprising when comparing payment systems, we are not including the refreshing process in the TPS.[2] However, we do run the refresh process as usual in Taler.

In practice, we expect a typical Taler deployment to use about 20 different denominations of value 2^i currency units. Assuming transacted amounts are random amounts in the range of $[2^0 - 2^{20}]$ currency units, this would result in the average transaction involving roughly 10 coins. Thus, converting the reported Taler TPS into business TPS involves a division by approximately 10.

Counting coins was measured relatively easily when batch withdrawals were not yet implemented, since at that time it was just the number of successful requests to *reserves/withdraw* and *coins/deposit*. For batch withdrawals, we then added new metrics to the exchange processes that count the number of coins signed in each request to *reserves/batch-withdraw*. In the end, metrics like this were eventually implemented for other requests as well, as we found that rsyslog was getting too slow in our high load application. This change now affects the counter as well: While idempotent requests were also counted when counting HTTP requests (since they return status 200), the new metrics now ignore them, so only *real* transactions are reported.

6.3. Wallet Performance Analysis

The Taler wallet performance was originally envisioned as a load generator, but not itself an optimization target. However, even if we allocated many machines to generate the client-side load, it turned out that the original wallet performance was inadequate to generate a sufficient load, even when using the maximum of 800 servers of Grid'5000 to run wallets.

6.3.1. IndexedDB

A key reason for this is that the wallet is typically executed in a browser, and uses the browser's IndexedDB for storage. However, for the experiments we are using the NodeJS-based command-line version, which comes with its own implementation of NodeJS, which is substantially less efficient. Furthermore, while we can run several wallets per core and thus dozens of wallets per server, the IO subsystem of the host cannot sustain the resulting load. A first trivial step was to move the wallet database file onto a RAM disk. However, due to the inefficient implementation of the IndexedDB in NodeJS, this still is insufficient as the Taler TPS drop as the wallet database grows over time (as shown in Figure 6.1).

Thus, we modified the wallet to drop its state between iterations to avoid the TPS drop (Figure 6.2).

[2] If we had also counted refreshing, we would see about 4k TPS more in our experiments with 23.5k TPS.

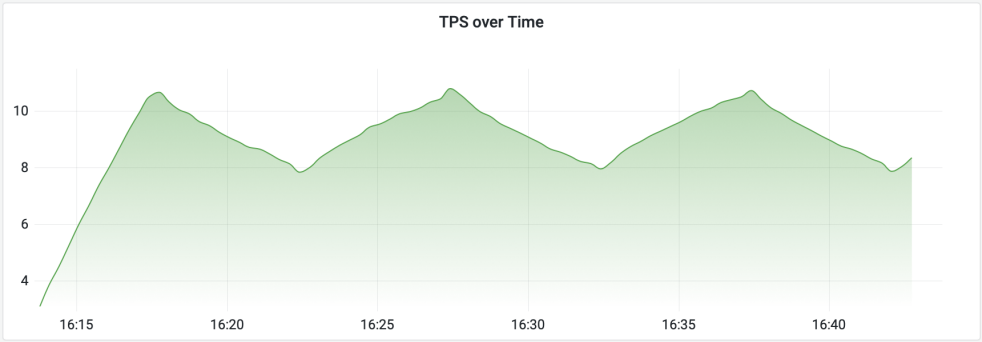


Figure 6.1.: TPS (deposit + withdraw) visualized over time when re-initializing the wallet's in-memory database after multiple iterations. It shows the impact of the growing file based json database on the overall performance.

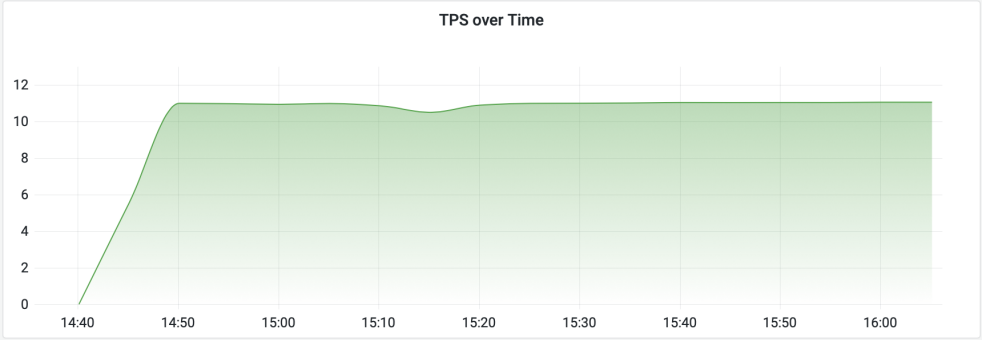


Figure 6.2.: TPS (deposit + withdraw) visualized over time when re-initializing the wallet's in-memory database after each iteration. The TPS are now stable but still on the same level even though the initialization has to be done each time.

The wallet's NodeJS implementation of IndexedDB makes heavy use of a structured clone primitive to deeply copy objects when storing/retrieving records to/from the database. We identified this bottleneck and switched to a more optimized implementation, yielding a $\approx$ 40% wall clock time improvement time in local tests.

Furthermore, we discovered that the wallet's NodeJS implementation of IndexedDB had accidental $O(n^2)$ run-time behavior when iterating over an index where multiple records have the same key. While this did not result in any observable performance improvement, we expect this improvement to become useful once we have identified more queries that would benefit from better indexing.

6.3.2. CPU Consumption

The wallet must perform certain cryptographic operations. These were initially implemented in TypeScript. As these are rather slow, we have added the ability to offload some cryptographic operations to a crypto worker written in C, which reads requests for cryptographic operations on stdin and writes the result to stdout. In local experiments, this yielded a wall clock time performance improvement of $\approx$ 20%.

About half of the cryptographic operations of the wallet involve checking cryptographic signatures made by other components. As in a benchmark those signature checks will always succeed (there are no malicious actors), we added an option to disable the signature checks.

6.3.3. Expensive Serialization

When we changed the amount of denominations from 14 to 3, we discovered that the wallets increased a bit in performance. While this impact was at first only minor, we could create a major performance boost by also reducing `LOOKAHEAD_SIGN` in the exchange secmod configuration. This results in the exchange generating fewer denominations and signing keys into the future. This performance boost came from less json serializations the wallets had to do, as responses from the exchange became a lot smaller. With the default of 2 years the responses to */keys* were about 900kB each, while setting them to 10 hours made response sizes go down to 1.7kB. The impact of those changes can be seen in Figure 6.3.

Those changes might not be applicable in production to this extent, but there we would also have one wallet client per host, thus performance is less important there than in our experiments. The only thing which could become a problem might be the bandwidth on the exchange, but this could also be trivially distributed to multiple nodes. However, it was clear that there is room for improvement in the way the wallets process the data they receive. It would also probably not be necessary to retrieve all signatures for every denomination for the next two years in advance.

6.3.4. Less Aggressive Behavior

Because batch withdraw was initially implemented as an experimental feature without idempotency checks, we discovered a problem that we might not have found otherwise: the wallets were too aggressive in re-issuing requests to the exchange. They retried every request that was not answered within 200ms. Using our average response time of 75ms [3] plus 12ms [4] round-trip time (also average) between the wallet and Nginx proxy, we get an overall average of nearly 90ms. So there could be quite a few repetitive requests, especially when the response time increased when we were getting near our TPS limit. While this was not noticeable for sequential withdrawal requests since they were already idempotent, it showed up as a conflict for batch withdrawals. We then fixed this behavior by changing the retry timeout to 1s, which led to several changes in our experiment. Although we did not expect a large change in TPS from Taler, since the repeated requests basically had to go through the same operations as the original ones, i.e. signatures created on the exchange, database accesses, etc., we actually ended up seeing an increase to 18.2k. This was mainly, but not exclusively, due to the decrease in serialization errors, which no longer occurred as frequently, as the repeated requests hitting the same row in the database were eliminated. The decrease, in turn, had a positive impact on repeated queries, and thus on the total number of queries required to complete a transaction. Which meant that there ended up being more resources available for further queries.

Notably, this didn't just impact the backend, but also the clients. While before this fix we needed about 3000 wallets to reach 13.6k, after removing the aggressive retries, it now takes about 8400. The reason is simply that retrying a request is cheaper than issuing a

[3] Measured by the Nginx proxy and represents the complete request time from the first byte received to the last byte sent
[4] This is measured without entering the userspace, as ping requests are responded by the kernel

Figure 6.3.: TPS (withdrawals and payments per second) shown with different values for `LOOKAHEAD_SIGN`. In each experiment we started two times 600 wallets, and we can see clearly how much their speed improved.

completely fresh request, but was previously counted as a transaction just like the original request.

6.3.5. Final Performance

With these changes, we can run only approximately 500 wallets per host (approximately as there are more or less performant hosts). [5] Grid'5000 currently has around 800 nodes that could be reserved, which would yield a theoretical maximum of 400'000 wallets. In the first part of this work, we had some difficulties because each node could only host about 50 wallets, resulting in an absolute maximum of 40'000 clients. Since each of these wallets could clearly contribute less than one transaction per second to the total, it would have been impossible to reach the goal of 100'000. However, with the current wallet performance, we achieve about 1.2 transactions per second with a single wallet, with all 400'000 this would result in a theoretical maximum TPS of 480k, which is well above our target.

Thus, for subsequent experiments we can be sure that the wallets will no longer will become *the* bottleneck. [6]

6.4. Exchange Database

As mentioned earlier, the Taler exchange was previously not run against a database running on a different server. For the experiments, we co-located the database and the exchange HTTP frontend in the same cluster. We expected database transactions to take a bit longer due to the additional network latency. Beyond that, we always expected the database to become the bottleneck once we distributed the HTTP frontend over multiple machines. As expected, the database quickly became the source of many performance problems. Thus, naturally, after most of them were solved, we still suspected the database to be the problem — even when it later turned out not to be the case. Here, we spent some time in optimizing our queries, which only resulted in minor gains. Nevertheless, these optimizations were likely still helpful in boosting the final performance numbers.

Note: Most of the results shown here were measured without batch withdrawals, as they were not implemented at that time. Also, in most cases, we did not start the aggregator either. If we did something different, or something specific, it is explicitly noted.

Why we did not use the Aggregator

The reason we did not start the aggregator in most cases is that the *materialized indexes* for `deposits` were partitioned using the `HASH` method **for simplicity**. As a result, using the aggregator leads to numerous slow queries and eventually serialization errors because a single transaction hits most if not all partitions.

The correct way to partition for the aggregator would be to use a `RANGE` method, which should result in conflict-free transactions. However, further work would be required to create dynamic range partitions by date (timestamp) on the *wire* and *refund* deadline. These would

[5] In our mainly used load situation as described in Section 6.2

[6] There are still venues for improvement, which might have an impact on the performance, but the current implementation is good enough to reach the 100k TPS in Grid'5000.

need to be created dynamically, e.g. by a cronjob, to get new partitions for future time ranges where new rows are inserted. Since the aggregator only touches partitions that are dated in the past, e.g., because it waits for the refund deadline to pass before initiating the actual transfer, this would ensure that it only hits a partition that is no longer touched by any other action. This method is expected to eliminate the serialization errors. Furthermore, as soon as the aggregator finishes its work on these partitions, they could be safely deleted. While we are convinced this would work, implementing dynamic partition generation, especially with sharding, is non-trivial and thus was left for future work.

6.4.1. Connections

During long-lasting experiments the database showed a quite high memory usage until its node eventually crashed and rebooted at some point. This behavior is visualized in Figure 6.4 just at the point the RAM was filled. Subsequently, PostgreSQL would occupy all the available swap space until the node would shut-down and reboot.

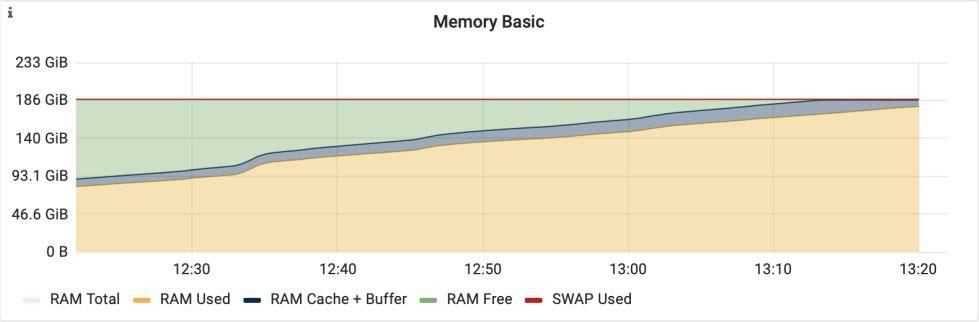


Figure 6.4.: The database's node memory usage with the same connections opened during whole experiment duration.

In the beginning it was not clear why this happened, since the configuration parameter `shared_buffers` was set to approximately a third of the total memory available. Which, according to the documentation, is the amount of memory available for all processes belonging to one database instance[7]. But as it seems this is not the only way that PostgreSQL allocates memory:

> "*Backend processes start out around 5 MB in size but may grow to be much larger depending on the data they are accessing." [23].*

> "*The most likely cause for this is that PostgreSQL keeps a per-connection cache of metadata about all the database objects (tables, indexes, etc.) it has touched during the connection lifetime. There is no upper limit on how big this cache can get, and no mechanism for expiring it. So if you have hundreds of thousands or millions of these things, and a long-lived connection will eventually touch each one of them, then their memory usage will continuously grow." [24]*

Both of those statements initially found through an article of *Italo Santos* [25] lead us to identify the cause of the database server crash.

[7] PostgreSQL Glossary: `https://www.postgresql.org/docs/13/glossary.html#GLOSSARY-SHARED-MEMORY`

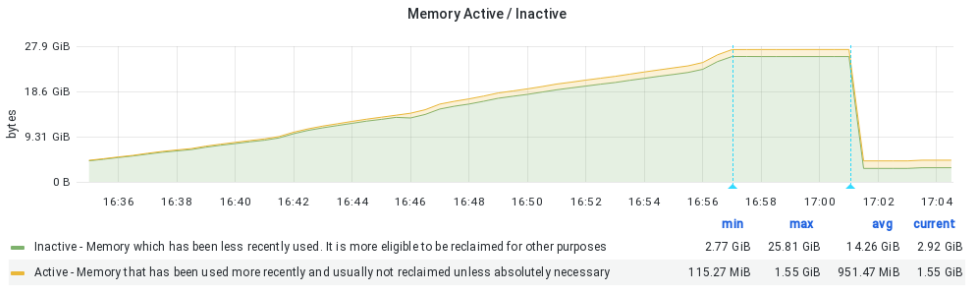


Figure 6.5.: Verification of the research about long-lasting connections in PostgreSQL. While wallets are doing withdrawals and deposits the memory usage continuously grows. Once all those wallets are stopped (first blue line), the memory is still allocated but constant. It is then finally freed when all processes having a connection to the database, such as the exchange, are killed (second blue line).

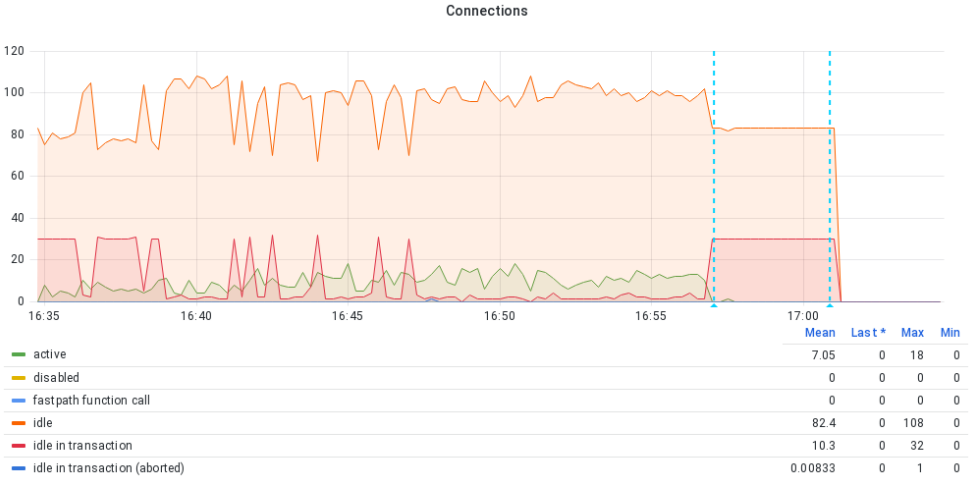


Figure 6.6.: The same verification run as in Figure 6.5 showing the connection statistics. Stopping all wallets lets the connections stay open but without any in an active state. Through those two figures it is clear that the connection's memory use only gets larger when they are used. Stopping the exchange processes closes all connections and thus releases the allocated memory (see Figure 6.5).

The exchange, as well as other Taler processes like wirewatch or aggregator, all use a persistent connection to the database. For each of those connections, PostgreSQL creates a separate process called *backend*. While doing work on the database these processes maintain an internal cache which occupies memory other than the one of the shared buffers. Thus, they will grow until there is no more memory left (since there is no configuration option in PostgreSQL which would prevent this behavior).

According to the findings above it would seem that when there is no work to be done for the backends, their memory usage should stay more or less constant. Since they do not hit any resources of the database and thus their cache does not grow. Finally, their memory should be freed once the connection which they originate from is closed. To testify those assumptions, an experiment is carried out in the grid. The results of this can be seen in Figure 6.5 and 6.6.

Some of the resources used to find the issue state that one solution would be to use a connec-

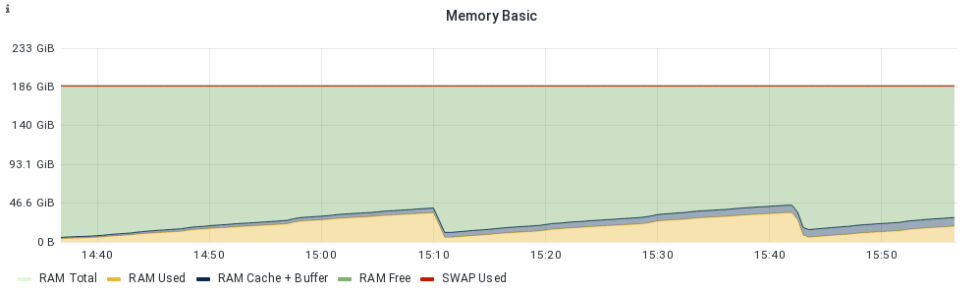


Figure 6.7.: The database's node memory usage with periodically closing connections due to exchange restarts.

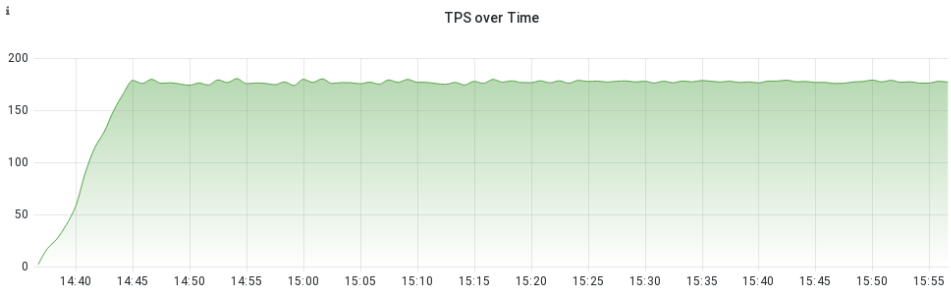


Figure 6.8.: Taler TPS in the same run as in Figure 6.7. Exchange restarts seem to be implemented correctly, without a visible impact on the performance.

tion pooling mechanism, such as PgBouncer[8]. We briefly investigated the use of PgBouncer, but it quickly became clear that PgBouncer can not help with this particular issue: The only connection pooling strategy[9] which would make sense is *transaction*, since *session* would be just the same as when using the connections without pooling and *statement* is not possible since Taler uses transactions with multiple statements. But while using *transaction* pooling the exchanges fail on creating or using prepared statements.

Thus, the solution we ultimately deployed is quite different: The exchange now has a new configuration parameter called *MAX_REQUESTS*. This parameter is an upper limit for the total number of requests clients can drop off at the exchange. Once this limit is reached the exchange process will kill itself. This naturally also releases the database connection and frees the memory allocated by the connection's backend. `systemd` is configured to automatically start a replacement exchange process should one commit suicide.

In order to keep the TPS stable when those restarts happen, the killing must be implemented carefully, and thus it happens the following way:

Once the maximum allowed requests are reached, the exchange will close the listen socket and call `fork()`. The child process will then finish processing all of the currently active client connections, so that ongoing requests are completed normally. The exchange parent process will then exit, which will cause a replacement to be started by `systemd`. While the restarting happens quite instantly, the forked child, handling client connections will probably finish later. Once they do so, the child will also terminate. Taler uses `systemd`

[8] PgBouncer Homepage: `https://www.pgbouncer.org/`
[9] PgBouncer Pool Modes: `https://www.pgbouncer.org/config.html#pool_mode`

socket activation [10], hence `systemd` always has an open `listen` socket, which ensures that whenever a client initiates a new connection there is a process listening on the port. This way, restarting of the exchange does not result in any failed requests, thus minimizing the impact of the restarts on system performance.

Running the same experiment again with the exchange suicide implemented, shows that the memory is freed once the connection is closed. Figure 6.2 shows the database's memory usage in this case. While the memory shows quite clear when it is released, the TPS still stay stable (Figure 6.8) because the clients can finish their current actions without disruption.

However, with further experimentation and various corrections to the query performance, we found that memory usage no longer increased as much. This means that the parameter *MAX_REQUESTS* could actually be set to a large number (likely millions or billions of requests) in a production deployment.

6.4.2. Slow Queries

In our experiments, we found how important it is for good database server performance to address slow queries. When many slow queries are executed on the node, CPU utilization can increase sharply. These findings are also shown in Figure 6.9 and 6.10.

An extreme example of such a query was a missing unique constraint and thus index for `wire_targets`. The query took about 400-500 ms, but was executed almost 40'500 times in just 20 minutes, which meant about 5 CPU hours in the same period of time.

During several experiments, we discovered several slow queries. Quite early in the experiments, there was a slow query that took more than 1.5 seconds to complete in some cases. Fortunately, this high duration was fixed fairly quickly due to a missing index in the table `refresh_revealed_coins` that was originally present but then lost due to a schema change.

However, while fixing one of these queries could be beneficial to the TPS for a while, other slow queries occurred when the load increased. Some similar cases as in `refresh_revealed_coins` occurred as soon as we started partitioning the database. Indexes were lost or forgotten to be added in each partition. Since it is not possible to get unique constraints that are not part of the partition key of the parent table, they have to be added in each partition afterwards. This was solved by simple SQL functions using `EXECUTE FORMAT` like the one shown in Listing 6.1.

[10] systemd.socket: `https://www.freedesktop.org/software/systemd/man/systemd.socket.html`

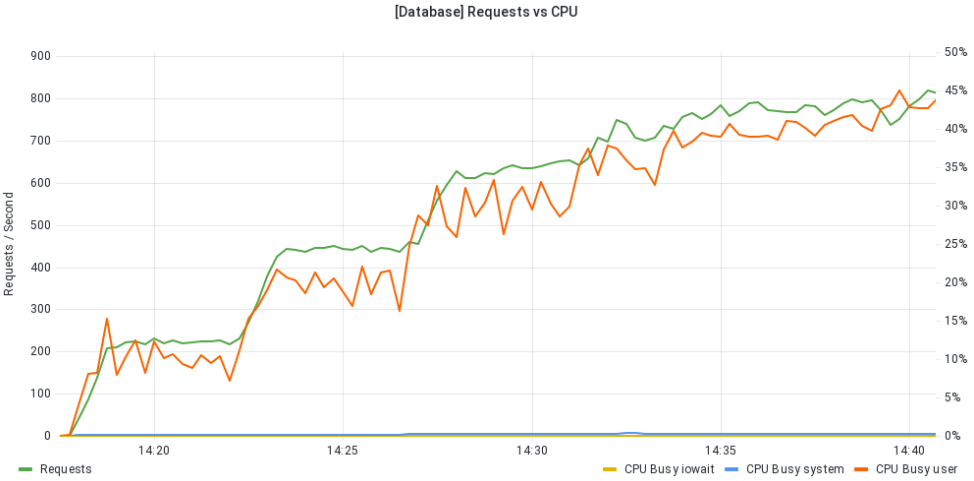


Figure 6.9.: The CPU usage of the database server compared to the total number of requests to the Nginx proxy during a slow query in the exchange withdrawal process.

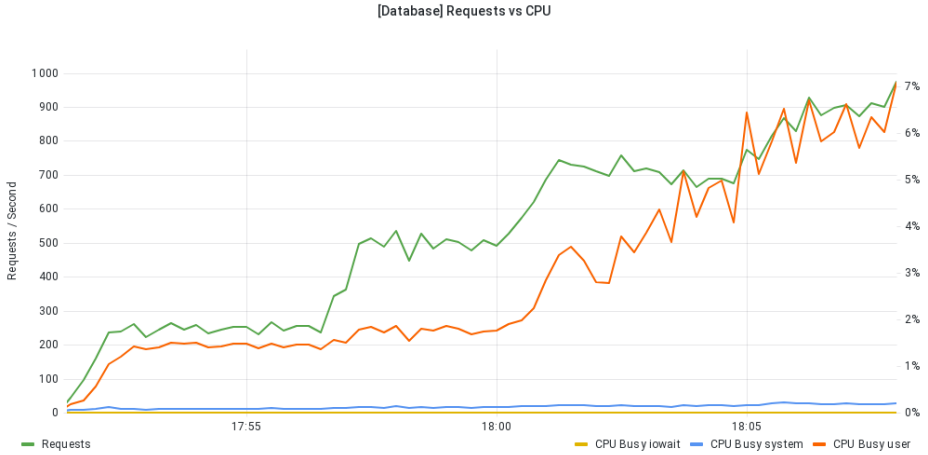


Figure 6.10.: The CPU usage of the database server compared to the total number of requests to the Nginx proxy when the slow query in the withdrawal procedure is fixed. Compared to Figure 6.9, we dropped to 5% CPU utilization at 800 queries, while we had about 40% before.

```
CREATE FUNCTION add_constraints_to_wire_targets_partition(
  IN partition_suffix VARCHAR
)
RETURNS void
LANGUAGE plpgsql
AS $$
BEGIN
  EXECUTE FORMAT (
    'ALTER TABLE wire_targets_' || partition_suffix || ' '
     'ADD CONSTRAINT '
      'wire_targets_' || partition_suffix || '_serial_id_key '
      'UNIQUE (wire_target_serial_id)'
    );
END
$$;
```

Listing 6.1: Example *add_constraint* function for partitions of *wire_targets*. This function adds constraints to a partition which cannot be created on the (partitioned) main table anymore.

These functions are called once for each partition/shard and add any required constraints that could not be retained in the parent table. However, due to the partitions, slow queries occurred again, which had not occurred before. Some queries with suboptimal `JOIN` or `WHERE` clauses required searching all partitions, sometimes multiple tables. An example of such an expensive query is `recoup_by_reserve`, where each `JOIN` is based on a unique constraint, but none of them is used as a partition key. This leads to the need to fully scan the partitions of each table, which can be very expensive in the case of sharding (see more on this in Section 6.5). However, this was not the only problem with this query. While it was initially executed in about 100-300 ms, its duration suddenly increased to 1.5-3 sec. We started to examine the execution plan with `EXPLAIN ANALYZE`, which showed that PostgreSQL hugely overestimated the number of rows expected to be returned. These estimates will be very bad for performance because:

> " *[...] the query planner uses row count estimates to choose between different query implementations with very different performance profiles. If those estimates are a long way out, then the query planner can make some bad choices, leaving your query running very slowly indeed.* " [26].

There might be many factors that lead to this miscalculation. We focused on *dead tuples*, as we did not expect any and some articles suggested that they might indicate reasons why the query planer produces bad estimates.

6.4.3. Dead Tuples

We found that the `known_coins` table had a significant number of dead tuples (about 10% of the amount of live ones). This seemed suspicious at first, given that we were not doing deletes on the database during our experiments. But looking at PostgreSQL's `UPDATE` statement documentation, the dead tuples make sense:

> " *In PostgreSQL, an UPDATE or DELETE of a row does not immediately remove the old version of the row. [...] the row version must not be deleted while it is still potentially visible to other transactions. But eventually, an outdated or deleted row version is no longer of interest to any transaction. The space it occupies must then be reclaimed for reuse by new rows, to avoid unbounded growth of disk space requirements. This is done by running VACUUM.* " [27]

Thus, a UPDATE could be considered a DELETE followed by a INSERT. These 'row versions' are also referred to as *tuples*, where the old versions (the 'deleted' ones) are called *dead tuples*. Since known_coins, for example, is updated during many operations (such as deposit and melt), there are quickly many dead tuples.

Although dead tuples do not have a direct effect on the query planner, they might indicate one thing: stale statistics. Too many dead tuples indicate that the statistics are outdated, which in turn affects the query planner, possibly resulting in a bad plan. However, they also affect the amount of disk space used [27] [28]. A common solution is to use the *autovacuum daemon*, which updates these statistics and also frees the dead tuples [29]. However, the default settings may be insufficient for tables that change frequently or grow rapidly. The schedule can therefore be customized with default_statistics_target [11] and the *autovacuum_** [12] parameters on a global or table-by-table basis. We have tried this but soon realized that it does not have a significant effect on the query plans or the performance for our experiments.

We also learned that the occurrence of *dead tuples* could be minimized with so-called *HOT* updates where the rows are updated place, i.e. in the same page the old row becomes a pointer to the new row (HOT chain). This has two main advantages: no indexes need to be recalculated, and *dead tuples* are removed without having to perform VACUUM [13], possibly improving PostgreSQL's performance a bit. However, they are also only possible if the UPDATE statement does not include any indexed row [30], and require additional storage space since extra space must be kept free for updates in the pages. To enable hot updates, the storage parameter fillfactor [14] must be evaluated and set appropriately for each desired table.

We made sure that the two relevant updated tables, reserves and known_coins, did not have any index on updated rows and thus satisfied the requirements for HOT updates. However, we could not see any immediate performance improvement. While some sources state that it takes some time to become observable [30], we are sure that it depends on the amount of data processed, which in our case would certainly be enough. We assumed that we could not see any direct impact because we did not have a resource at the limit.

Later we added custom queries to our PostgreSQL exporter for Prometheus that made user table statistics available for dashboards. Through these metrics we saw why we had no noticeable difference in neither I/O load nor PostgreSQL performance: most of our updates were already HOT (see Figure 6.11).

6.4.4. Conclusion

We found out that the PostgreSQL query planner can be difficult to understand sometimes. Despite our efforts, we could not find a way for it to consistently optimize the SQL queries in their "natural" form. While ad-hoc optimizations by the PostgreSQL planner may generally be desirable for one-off queries, consistent performance is much more crucial for a secure system than peek performance or ease of development.

[11] PostgreSQL Statistics: https://www.postgresql.org/docs/13/runtime-config-query.html#GUC-DEFAULT-STATISTICS-TARGET

[12] PostgreSQL Autovacuum: https://www.postgresql.org/docs/13/runtime-config-autovacuum.html

[13] Because the HOT chain is updated by different DB transactions, e.g. by a SELECT [30].

[14] PostgreSQL fillfactor: https://www.postgresql.org/docs/13/sql-createtable.html#SQL-CREATETABLE-STORAGE-PARAMETERS

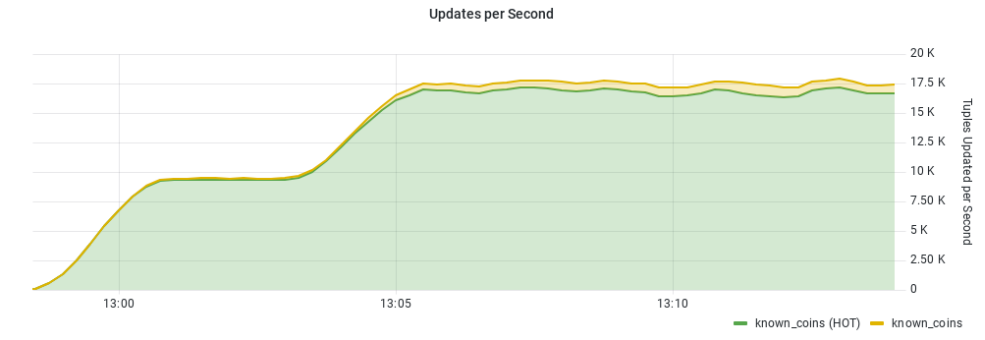


Figure 6.11.: Updates per second in the `known_coins` table, the table with the most changes. Without adjusting the `fillfactor`, we already see 16'700 HOT compared to 700 non-HOT updates per second, while we have a total TPS of 21k in Taler. It is clear that changing the fillfactor on this table cannot have a significant impact. (For the `reserves` table wee see no HOT updates at the default settings, while there are 1000 updates per second in total)

Since we were not able to achieve a consistently good query planner execution by configuration, we decided to go ahead and try to create different queries to improve the query performance by explicitly writing the queries in a way that largely eliminates unreliable query planner performance. The results of these reformulations are explained in Section 6.5. Here, we shall simply note that initial attempts to write queries with explicit join clauses [15] to enforce some order of execution were unsuccessful, as even with explicit joins the query planner still performed queries in unexpected and ultimately suboptimal ways.

6.4.5. I/O Load

At the beginning of the benchmarking (at around 300 TPS in Taler), the database node showed a very high I/O load. This surprised us a bit since the node hosting the database (Dahu) uses fast SSDs. According to the datasheet of the main SSD[16] the disk should be able to handle about 400MB/s sequential reads/writes, 90K IOPS random reads, and 10K IOPS random writes. But the dashboards showed that these values were not achieved in the slightest: with only 400 wallets, there was an I/O utilization of 80%, but only 406 IO write-operations and 6.5MB written per second while the read utilization was almost zero.

Several approaches were taken, such as swapping the WALs or temporary tables to another disk, enabling large pages, or increasing the WAL buffer size. Unfortunately, none of these approaches provided any relief, but we found that the problem was in the WAL.

Initially, we suspected that the disks might be worn out to some degree given the load they must endure in a testbed like Grid'5000, or that the datasheet values were generated under 'laboratory conditions'. However, separate hard drive benchmarks using the flexible IO tester[17] (*fio*) showed them to be just fine. By digging further into the PostgreSQL documentation, we found out the following configuration parameters [18] [19], which might be helpful for reducing the IO load:

[15] Explicit Joins: `https://www.postgresql.org/docs/13/explicit-joins.html`
[16] Samsung SSD `https://www.samsung.com/semiconductor/ssd/enterprise-ssd/MZ7KM240HMHQ/`
[17] fio: `https://fio.readthedocs.io/en/latest/fio_doc.html`
[18] PostgreSQL Resource Consumption: `https://www.postgresql.org/docs/13/runtime-config-resource.html`
[19] PostgreSQL WAL: `https://www.postgresql.org/docs/13/runtime-config-wal.html`

- `bgwriter_flush_after`:
  Force data to be written to the underlying OS storage when the specified amount is reached (default 512kB).
- `backend_flush_after`:
  Same as `bgwriter_flush_after` but for the *backend*[20] process.
- `effective_io_concurrency`:
  Number of allowed concurrent IO Operations for PostgreSQL, this value may be set higher for SSDs.
- `fsync`:
  Ensure that data is written to disk via a `fsync` (or similar) system call. Disabling this might provide a performance benefit in exchange with reliability.
- `synchronous_commit`:
  Defines when PostgreSQL returns 'sucess' to the clients. E.g. only when the data in the WAL is persistent (synchronous) or earlier already (asynchronous), default is on.
- `wal_compression`:
  Compress WAL pages before writing to disk, might decrease IO but increase CPU load.
- `wal_sync_method`:
  Specifies the systemcall for *fsync*.
- `full_page_writes`:
  Write entire pages to disk to prevent inconsistent pages after a system crash, default is on.
- `min/max_wal_size`:
  The boundaries for the WAL size, the configured amount of data can be written until it gets written to disks by a checkpoint.

We conducted experiments to find out which ones could have the biggest impact. However, we did not forget that some of them could be dangerous in a live system in terms of data loss. For example, if we disable `fsync`, there could be unrecoverable data corruption in case of a server crash. However, the parameter that seemed most promising (with the greatest impact on performance without too much risk of data corruption) was `synchronous_commit` (read more about this in section 6.4.8). While the documentation led us to believe that this might only be useful for client performance, since the data still has to be written to disk, it turned out to be the most efficient. However, it also turned out that even this parameter can lead to large data losses:

> " *The loss will be less than two times the wal_writer_delay in most cases. But it can be up to three times in the worst-case.* " [31]

With a default value of 200ms for `wal_writer_delay` this could result in a total of 60'000 Transactions (Taler) to be lost in the worst case (assuming we reached the 100'000 TPS) which is clearly intolerable for a banking application. However, it significantly reduced the IO load (see Figure 6.12), which was not entirely clear since no source indicated that it would reduce I/O. However, we suspect it reduces the load because the WAL does not have to be forced to disk for every commit [32]. Therefore, we disabled this setting because we also do not have a real influence on what disks are installed on the nodes in Grid'5000. There

[20] PostgreSQL backend: `https://www.postgresql.org/docs/13/glossary.html#GLOSSARY-BACKEND`

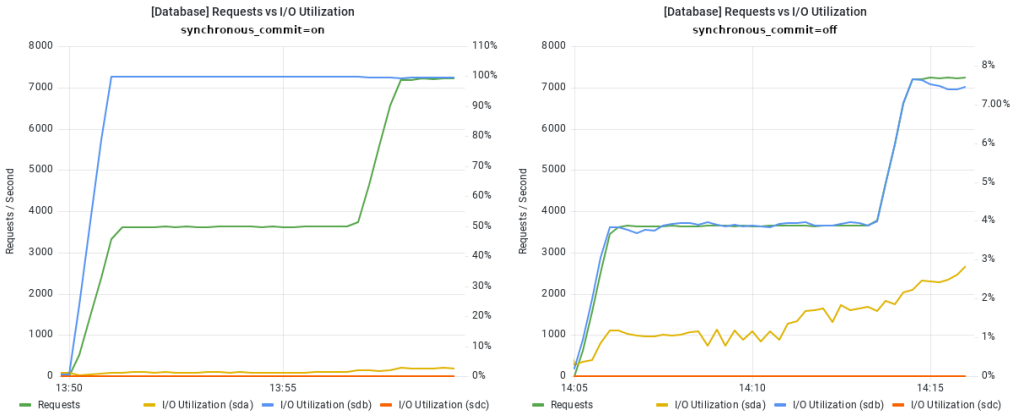


Figure 6.12.: These two figures show how the `synchronous_commit=off` setting affected the load on the disks. In both cases, the WAL was stored on *sdb* and we started 400 wallets twice (400 wallets: 2.3k and 800 wallets: 4.6k TPS on the exchange) with the only difference being the value of `synchronous_commit`. It is clear to see that asynchronous commits significantly reduce the I/O load, while the number of requests per second on the exchange remain the same.

is a good article by EDB which shows the performance impact of different values for this setting [33]. For `fsync`, however, it is not recommended turning it off, even for performance experiments, since it is not applicable in production (which was also pointed in multiple responses on the PostgreSQL mailing list [34]).

6.4.6. Serialization Errors

At the start of the experiments, the database exhibited a high number of serialization errors. We added code to export serialization failure details for each request type to the Taler software, which shows that serialization errors are directly linked to drops in the number of requests (see Figure 6.13) as some requests run into timeouts after experiencing repeated serialization failures. Figure 6.15 shows that the number of slow queries per second also increases when the serialization errors are high.

Some occurrences of the serialization errors from different endpoints were addressed by changing the PostgreSQL transactions. The most effective transformation was to replace 'SELECT and if not exists INSERT' patterns with an 'INSERT ON CONFLICT DO NOTHING otherwise SELECT' approach.

We also considered that we are seeing more serialization errors because transactions running on another node last longer due to the network latency. To address this, we rewrote some of the most critical transactions as stored procedures, thereby eliminating the need for additional round-trips. However, the effect of this transformation was minor.

That said, we still saw a high number of serialization errors for the */withdraw* endpoint, which sometimes reached an average of 3 serialization errors per request. That is despite these requests should actually never be conflicting.

In the end, we had taken several measures that helped us reduce these errors. These included partitioning the tables, which helped eliminate the need to access entire tables, but also rewriting queries and restructuring tables to fit these new partitions. Equally helpful

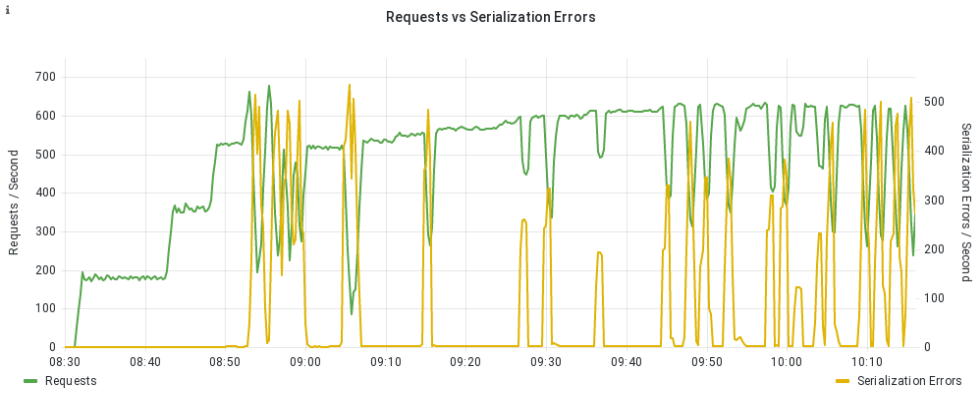


Figure 6.13.: All HTTP request processed by the Nginx proxy per second compared to the serialization errors (SE) per second logged by the database. It clearly shows the impact of the errors on the number of requests and ultimately on the total number of withdrawals and payments per second (TPS, see below).

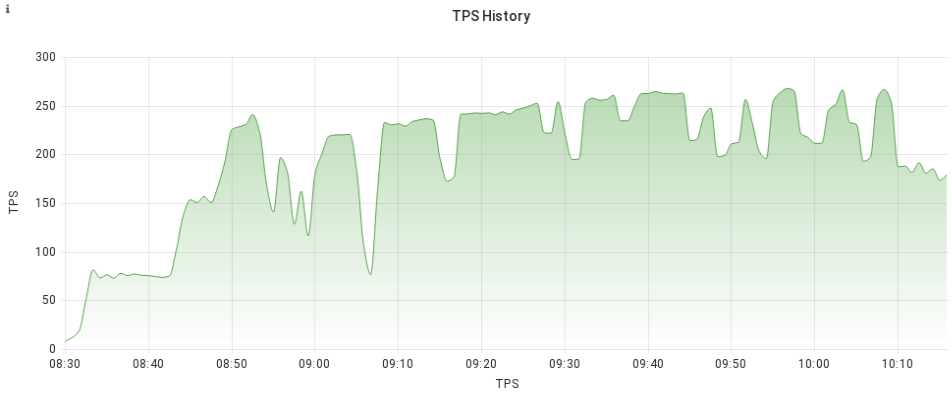


Figure 6.14.: Taler TPS affected by the serialization errors shown in Figure 6.13.

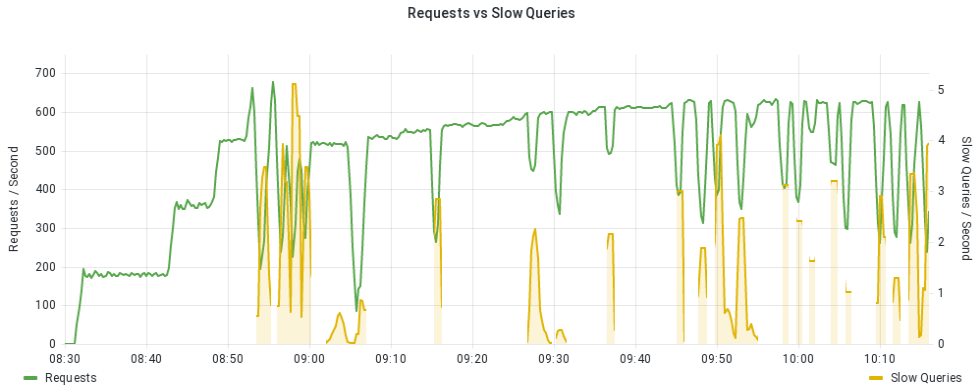


Figure 6.15.: This figure shows the number of slow queries relative to the number of HTTP requests processed in the same run as Figure 6.13. Slow queries seem to occur mostly at the same time when serialization errors occur. Looks like these two occur together.

was the addition of indexes where they were missing, as too many sequential scans of an entire table can have a negative impact on serialization errors [35]. Towards the end of the work, we were able to reduce the serialization errors on */withdraw* to almost zero. We accomplished this by introducing batch withdrawals. Before this change, the wallet sent a separate request sequentially for each coin in a reserve to withdraw only that coin. Among other things, this caused the row in the database with the reserve for this coin to be hit multiple times, namely for all coins of this reserve. If this happened too fast we got serialization errors when accessing this row. This was even more extreme when the wallet was yet too aggressive, retrying a request after 200ms would even more increase the probability for two queries hitting the same row. After this aggressive behavior was fixed we could reduce serialization errors on withdraw from the 30/s seen in Figure 6.16 to below 3/s (Figure 6.17). For batch withdrawals, they will now withdraw all coins at once in one big request to *batch-withdraw*. Which means that the reserve is now directly set to zero in a single database transaction and thus is only hit once, which makes serialization errors no longer possible. The results of this change can be seen in Figure 6.18.

With all these changes we have now achieved and shown that regular transactions of Taler are not conflicting, as designed.

6.4.7. Number of Database Transactions

Since we were consistently hitting the limit in terms of number of TPS in PostgreSQL and saw no significant number of serialization errors, we tried ways to reduce the total number of database transactions by running larger transactions. An example for such a change was the introduction of batch-withdraw, which initially brought only a minor improvement in performance. We also found that changes like this did not only reduce I/O load as expected, but sometimes even further reduced the number of serialization errors.

In going through possible places to minimize transactions, we found an interesting case. Queries issued during the *refresh-reveal* phase were not part of a transaction block, but were issued separately, with auto-commits performed for each query. There is no clear number of how many queries were issued separately, as it is a loop over all *fresh coins*. We ran two similar experiments with the same nodes assigned to eliminate fluctuations that can occur when using different nodes. We found that we saved about 5k TPS with PostgreSQL, while we gained about 1k TPS with Taler (see Figure 6.19). However, this was not the only improvement we noticed. In fact, we also gained in I/O utilization on the WAL disk (Figure 6.20) without any real change in CPU load (Figure 6.21), with I/O load explained by the fact that fewer transactions need to be written to the log. However, the overall impact on TPS in PostgreSQL and Taler was small for the change in *refresh-reveal*. This was to be expected since there are fewer transactions per second involving refresh than ordinary withdrawals and payments.

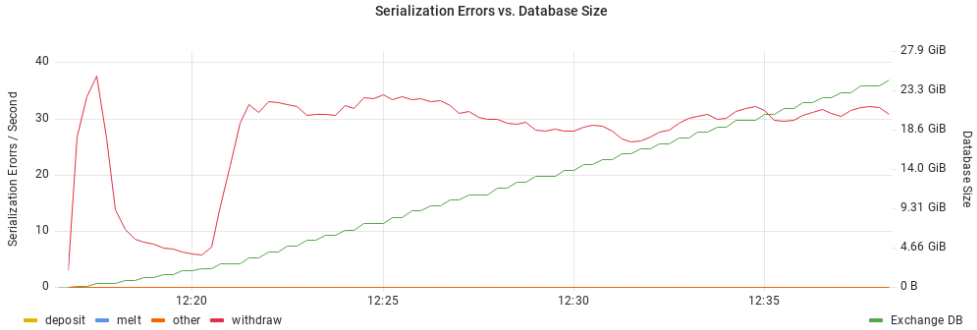


Figure 6.16.: Number of serialization errors compared to the total size of the exchange database before implementing batch withdrawals. It can be seen that serialization errors are highest for */withdraw*. This figure was recorded when about 6200 requests per second were made to this endpoint, where about 30 of them failed due to errors. This is still low compared to what we saw at the beginning, but we were able to reduce it to below 3/s at the end due to the less aggressive behavior of the wallet clients.

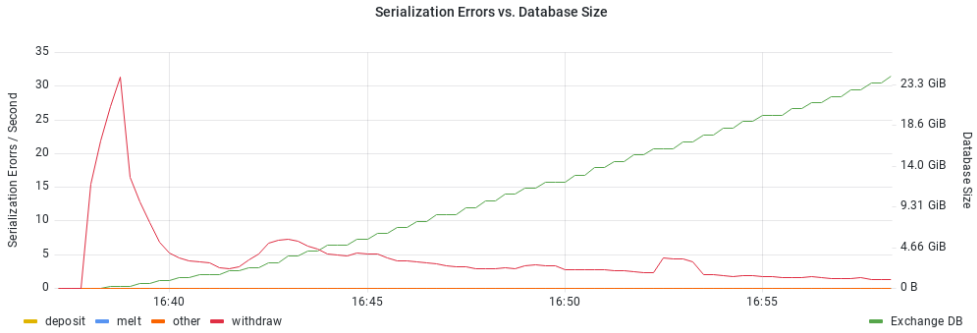


Figure 6.17.: Number of serialization errors compared to the total size of the exchange database when the wallets were adjusted to wait longer before repeating a request. At the time this number was obtained, there were about 5600 requests per second to */withdraw*. Fewer requests are being made because fewer (probably none) are repeated by the clients, partly because the wallets are less aggressive, but also because there are fewer serialization errors. It is clear to see that there are fewer serialization errors than in Figure 6.16.

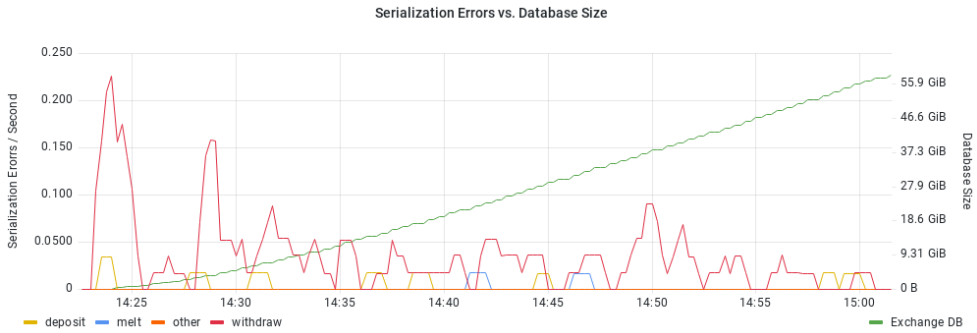


Figure 6.18.: Number of serialization errors compared to the total size of the exchange database when batch withdrawals are used. We see that there are almost no more serialization errors anymore; for 900 requests per second to *batch-withdraw* and about 6300 withdrawn coins per second we see only 0.05 (about average) serialization errors per second. Again a clear improvement compared to Figure 6.16 and 6.17.

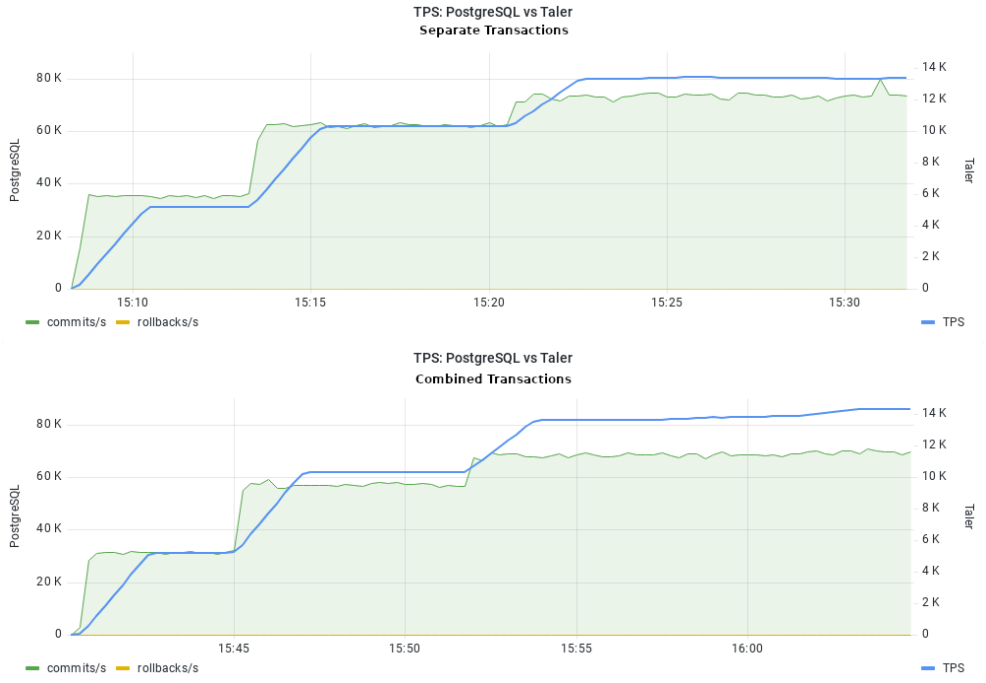


Figure 6.19.: The TPS of PostgreSQL compared to those of Taler. The first figure shows the performance before we combined the auto-committed refresh transactions into one large one. In both cases, we initially started 2400 wallets (in three steps) and then only 30 at a time until we reached the maximum possible TPS in Taler. At the time we maxed out in the first one, we had about 2310 requests to */refresh-reveal*. We can see that the performance of Taler did not improve anymore without the combined transactions, while in the second case we reached up to 14k+ TPS. At this point, we had about 2512 requests to */refresh-reveal*, which shows that our change did indeed make a difference. This is even more clear when looking at the TPS of PostgreSQL, those were reduced from about 74k to 68k (at the Taler TPS maximum). However, this was not the only improvement. As we can see in Figure 6.20, the I/O load for the disk where we stored the WAL was also reduced somewhat.

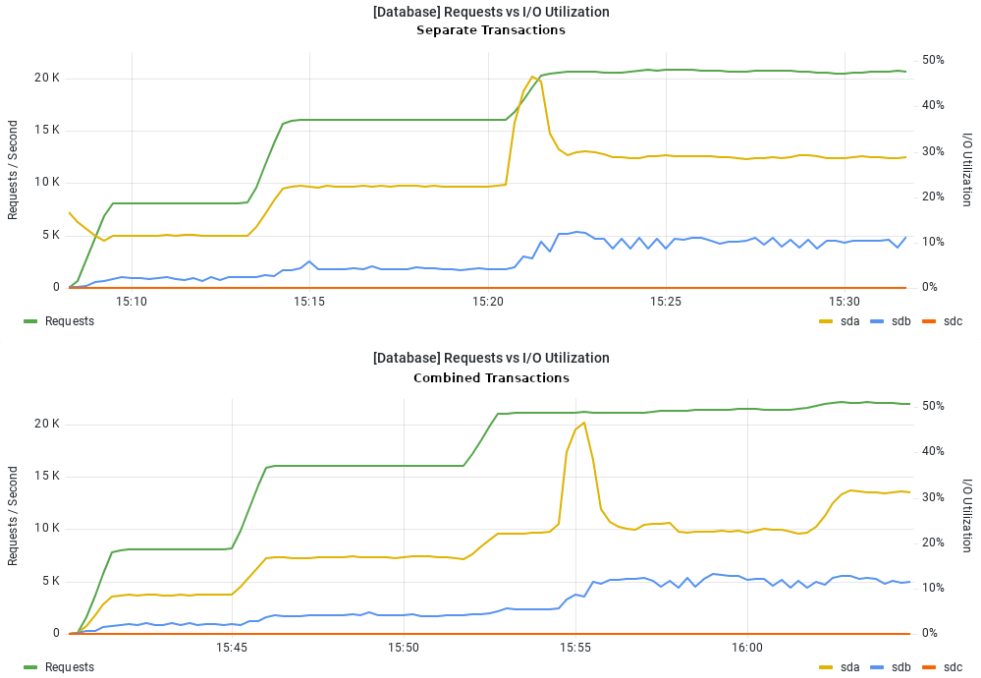


Figure 6.20.: The I/O load compared to the total number of requests per second in exactly the same experiment as in Figure 6.19, where the second figure shows the run where the transactions have already been combined. We see that combining multiple transactions into one also reduces the I/O load on the WAL disk. This is because the WAL is now also written only once instead of n times. However, we could not explain why exactly it increases again by so much towards the end.

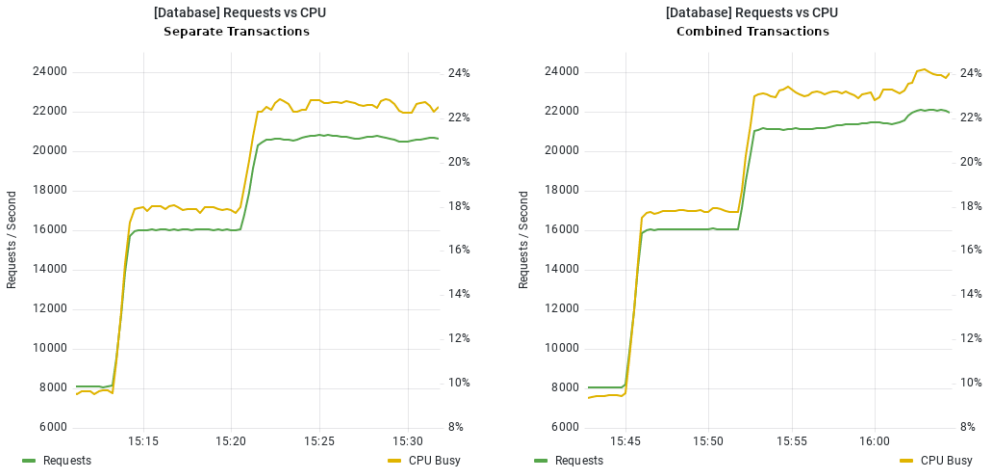


Figure 6.21.: The CPU usage of the database node compared to the total number of requests to the Nginx proxy. This is shown for a subset of the time of the experiment of Figure 6.19 and 6.20 to show it more clearly. We can see that the CPU load did not increase after combining the refresh transactions for the same number of requests. This shows us that by making changes like this, we won't suddenly get a CPU bottleneck.

6.4.8. PostgreSQL Benchmark

As already mentioned we believed to be facing a non-trivial bottleneck with PostgreSQL, when we had fixed the major problems described in previous sections. We were unable to reach more than roughly 65k TPS (DB) while CPU, I/O and network showed no signs of limitation. To track down the problem, we tried many things, which are discussed below.

Towards the end of the experiments, however, we found out that the problem was not in the database anymore but in the wirewatch, which showed us that we had been looking in the wrong place for a long time (more below). After fixing the 'bug' in the wirewatch, we were sure that we were bound with the *withdraw* operation and not with the TPS in PostgreSQL. Nevertheless, this section describes what we tried to better understand PostgreSQL and its performance.

The PostgreSQL software stack provides some additional utilities that can be used to perform various tasks. One of them is the `pgbench` utility. It can be used to benchmark a PostgreSQL instance. Therefore, we have performed some benchmarks with our system, which we have chosen to host the database.

Hardware

As mentioned earlier we used the *Dahu* cluster in Grenoble [21]. A node there has the following specifications:

- 4 * Intel Xeon Gold 6130 (Skylake, 2.10GHz, 2 CPUs/node, 16 cores/CPU)
- 12 * 16GiB DIMM DDR4 Synchronous Registered (Buffered) 2666 MHz (0.4 ns)
- 1 * SSD SATA Samsung MZ7KM240HMHQ0D3 (used for PostgreSQL data)
- 1 * SSD SATA Samsung MZ7KM480HMHQ0D3 (used for PostgreSQL WAL)

Initialization

To start a PostgreSQL benchmark, the database must first be initialized. This can be done with the following command:

```
$ pgbench -i -s 500 -U postgres postgres
```

Benchmarks

To benchmark our PostgreSQL instance, we ran the following command with a variation in number of clients (`-c`) from a remote system:

```
$ pgbench -t 10000 -c 64 -j 32 -h db.perf.taler -U postgres postgres
```

This runs 64 clients on 32 threads, with each client executing 10'000 transactions. Where each transaction includes one `INSERT`, one `SELECT` and three `UPDATE`s [36].

[21] Dahu: https://www.grid5000.fr/w/Grenoble:Hardware#dahu

Figure 6.22 and 6.23 show the result when we ran the benchmarks against the default and a custom configuration of PostgreSQL, respectively. The custom configuration used can be found in the appendix.

Many of the benchmarks found on the Internet use the `-S` flag to perform read-only benchmarks, resulting in a much higher TPS. However, since the Exchange database is very write-intensive (many updates and inserts), we did not focus on read-only benchmarks. However, for comparison with Internet resources, we ran a benchmark with 80 clients, which resulted in a total of 507k TPS.

What we see in these graphs is that PostgreSQL slows down the more clients are connected, we also saw this in our experiments. While we received only 10k TPS (payments and withdrawals) with 160 exchanges connected, it was 13.6k with 80. This looked strange at first, as the benchmarks from EDB [22] or Percona [23] (and others) reach their maximum TPS at many more clients (in the hundreds - but also with more CPUs sometimes). But the PostgreSQL wiki says that it is optimal to set `max_connections` to twice the number of physical CPUs (without Hyper-Threading) [37]. This would be consistent with what we observed for the nodes in Dahu, as the nodes there have 32 physical cores with 2 threads each. However, while we made improvements and increased performance, the number of clients that proved best for performance first increased to 120 and finally to 200 with partitioning, while it remained at 80 with a sharded setup. [24]

The amount of TPS in PostgreSQL was almost exactly the same in our experiments, with the difference that the experiments only used 25% of the CPU, while pgbench used 75%. A different behavior was observed for disk I/O, where pgbench had about 5-10% IO utilization, while the exchange had 45-50%. In both cases, however, neither system appeared to be the possible bottleneck. This led us to spend quite a bit of time trying to figure out what the bottleneck in the database could be, since it can't be a coincidence that both applications achieve the same amount of TPS? But no matter what we did, we could not get the amount of PostgreSQL TPS to increase. We tried many configuration combinations, such as disabling huge pages, halving/doubling some configuration values (e.g. `max_parallel_workers` or `shared_memory`), but none of them had a significant impact on performance. The only parameters that made things run faster were `fsync=off` or `synchronous_commit=off`. They were tested individually with the default configuration and in combination with a high value for `min` resp. `max_wal_size`:

Configuration	Default WAL Size	WAL Size (min=10GB, max=20GB)
Default	33k (100%)	48k (100%)
synchronous_commit=off	49k (100%)	65k (30%)
fsync=off	62k (30%)	65k (10%)

Table 6.2.: PostgreSQL TPS affected by configuration values (pgbench with 64 clients). Values in braces show the average IO utilization shown by `iostat -xm 1`

Clearly this table shows that the main bottleneck was I/O at that time and that removing

[22] EDB pgbench: `https://www.enterprisedb.com/blog/pgbench-performance-benchmark-postgresql-12-and-edb-advanced-server-12`

[23] Percona pgbench: `https://www.percona.com/blog/2017/01/06/millions-queries-per-second-postgresql-and-mysql-peaceful-battle-at-modern-demanding-workloads/`

[24] The number of optimal clients in sharding might also be different now, because the last experiment we ran with a sharded setup was at the time when we had 120 connections in a partitioned setup and reached about 23.5k TPS in thaler.

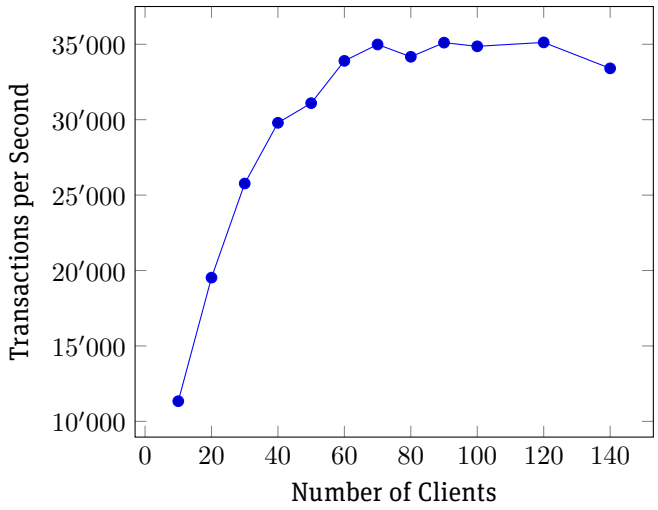


Figure 6.22.: PostgreSQL TPS (average of 3) measured with `pgbench` when using the default `postgresql.conf` and a node in the Dahu cluster.

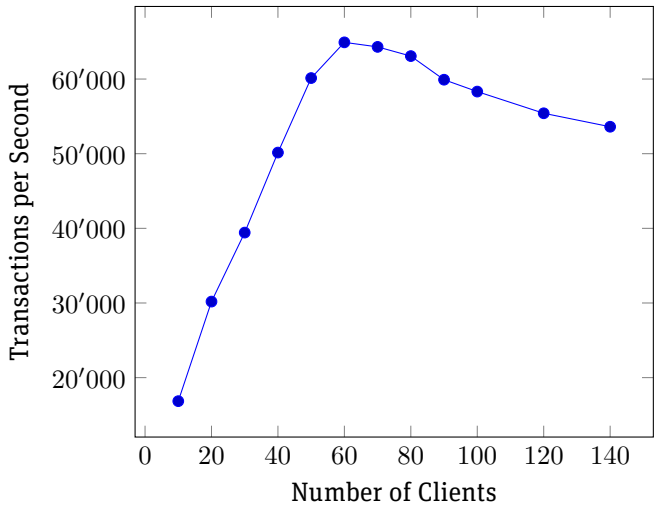


Figure 6.23.: PostgreSQL TPS (average of 3) measured with pgbench with the custom configuration which can be found in the appendix (Listing A.4), this was done on the same node as in Figure 6.22.

some I/O operations resulted in more TPS. To verify that I/O cannot be the problem anymore the data (including WAL) directory of PostgreSQL was placed in memory. But this did also not increase the TPS any further. `max_wal_size` can be set high enough in order to eliminate 'forced' checkpoints, so that they occur on a regular basis (configured with `checkpoint_timeout`) and thus do not impact performance too much [38].

Insertion: What are WAL and Checkpoint?

To understand the above section, it may be helpful to know what WAL and checkpoints are and do, respectively. The Write Ahead Log is a concept to ensure data integrity, because what would happen if we write an insert (commit) directly to disk and the database server crashes during it? To prevent this, an insert is first written to the WAL, which is basically a log file that contains all the changes that need to be persisted in the database files. At some point, however, this data must be written to the real database files. This is done by a checkpoint that takes entries from the WAL and writes them to disk so that the WAL space can be reused. In the event of a crash, all changes that were not written by a checkpoint can be restored from the WAL. [39] [40]

Discussion

It could be that the dataset was too small, resulting in many serialization errors or locks on rows/tables that slow things down. However, changing the size of the dataset at initialization (`-i -s N`) did not result in any significant change in the performance. This gave us the idea that the problem might lie elsewhere, possibly in the operating system. One guess was also that the operating system's lock concurrency might be the problem, which would be *easy* to debug with the `linux-perf` tools. However, to record lock statistics, one would need to build a custom kernel that had the statistics enabled [41]. It would also probably be a lot of work to get this customized kernel into the grid to simply run some experiments. But even without these statistics, we could exclude lock conflicts from the list of possible bottlenecks, since they would certainly be visible as sleeping processes in the output of `vmstat`. However, the output of `vmstat 1` (Section A.5) shows no such processes (column b), indicating that 'no' processes are actually blocked and waiting for a lock to be released.

Another factor that could potentially limit performance would be locks or wait events within PostgreSQL. But these could also be ruled out, since various queries, such as:

```
SELECT * FROM pg_locks
  WHERE granted IS false;
```

and

```
SELECT wait_event, count(*)
  FROM pg_stat_activity
  WHERE state='idle in transaction'
  GROUP BY wait_event;
```

have not yielded anything that would indicate a solution [42] [43] (statistics like these are also shown in the dashboards).

Unable to identify the issue, we created posts on the mailing lists of PostgreSQL [34] and Grid'5000 [25], the PostgreSQL Slack channel [26] and StackOverflow [27]. Most answers falsely suggested that I/O was the problem, and (unhelpfully) that we should definitely not disable `fsync` (which was so at this time). But we already ruled out I/O as the problem, because there are no faster disks than `RAM`, which we used in some of our experiment to exclude I/O from the list of possible bottlenecks.

6.4.9. Stress-Testing the Database Node

Since we could not find a solution to our (what we believed) performance problem with PostgreSQL, we ran several additional experiments to help us find our non-trivial bottleneck. Most of them involved pushing the available resources, which in most cases are the cause of a performance problem, to the limit during a maxed out experiment. Since our resources were obviously not running at maximum, we hoped that this might lead to new insights, since it is unusual that performance cannot be improved when there are still enough resources available. The following sections briefly describe what we did.

I/O

We have made many attempts to see if I/O could be our bottleneck, such as using NVME disks (with nodes of the Yeti cluster) or moving all data to memory. However, as we have already explained, none of these measures resulted in a performance increase. However, to really rule it out, we went one step further and pushed disk utilization to the extreme by using `fio` during a running experiment. The results of this experiment show that the hard disks are quite capable of taking more load than they are exposed to in our benchmarks, see Figure 6.24.

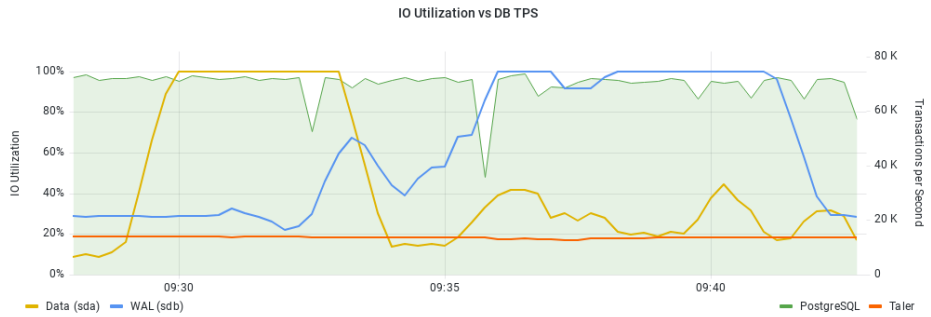


Figure 6.24.: The flexible io tester (fio) is running on the disks of our database node while we are at the TPS limit of PostgreSQL. We can clearly see that even when the disks are fully utilized, we still have about the same TPS. Thus, we can (once again) rule out the disks as possible bottlenecks.

[25] No link because it requires a login

[26] PostgreSQL Slack: `https://postgres-slack.herokuapp.com/`

[27] StackOverflow Entry: `https://stackoverflow.com/questions/71631348/postgresql-bottleneck-neither-cpu-network-nor-i-o`

CPU

Since we only had about 25% CPU load, we were pretty sure that this couldn't be the bottleneck, especially since we have also already seen that PostgreSQL is equally fast when we reached 75% load while using `pgbench`. Nevertheless, we also added load to this resource using a third-party tool called `cpuburn` [28] during an experiment. While there was a noticeable drop in PostgreSQL's performance (shown in Figure 6.25), this is nothing unexpected; after all, the CPU is now heavily loaded by other additional processes. Therefore, even after this experiment, we were sure that the CPU was not our main problem either.

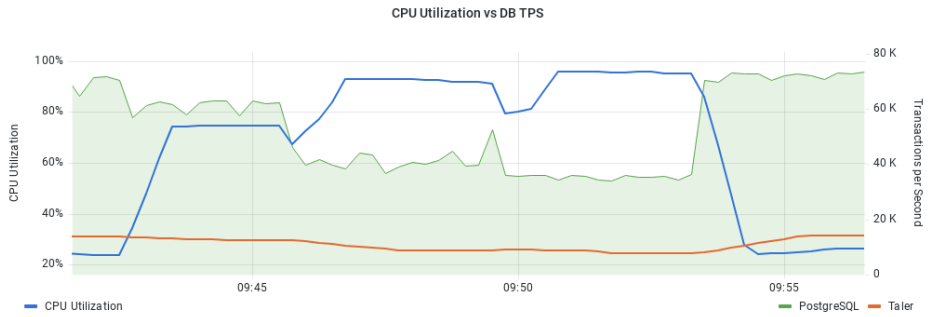


Figure 6.25.: `cpuburn` is used to fully utilize the CPU. Although this is the first effect we can really observe, it is still nothing that is unexpected as there are more processes now.

Network

As with the other resources, we also loaded the network while an experiment was running. To do this, we used `iperf` [29] to send large amounts of data from the exchange nodes to the database node. Again, we can exclude the resource from the possible bottlenecks, as we see in Figure 6.26 that the load generated by the additional network traffic did not really affect PostgreSQL's performance.

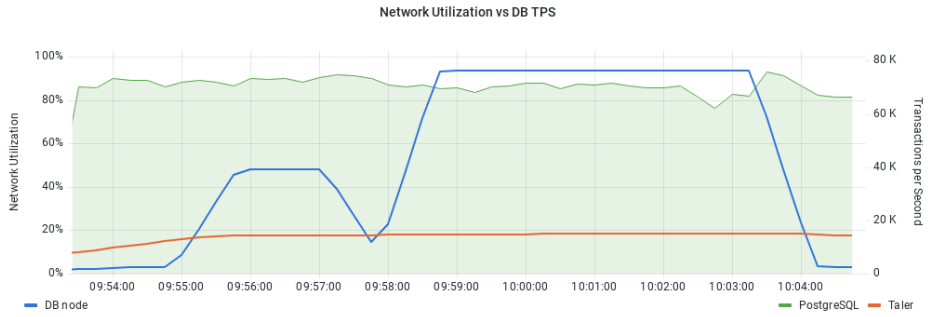


Figure 6.26.: The `iperf` utility is run to keep the network busy. As we can see, the effect is small, but this is clearly not a bottleneck either.

[28] cpuburn: `https://patrickmn.com/projects/cpuburn/`
[29] iperf: `https://iperf.fr/`

Multiple PostgreSQL Instances

Since we did not reach anywhere near full utilization for any of the most common bottlenecks on any of the DB nodes we tested, we wanted to see what would happen if we hosted two PostgreSQL instances on the same node. To do this, we initialized two PostgreSQL servers with independent data on different ports. The data was still stored on the same disks, but both instances used independent directories. Then we initialized two exchange nodes, each communicating with one of the PostgreSQL instances. The only thing that was then shared was the hardware of the database node, that of the clients and the bank, of which there was only one. The clients were also configured to ensure an even distribution of operations across the exchanges. The idea was to find out if the server hosting the database is capable of taking more load and if there might be a bottleneck in PostgreSQL or due to a misconfiguration.

In the first run, we found that the TPS on PostgreSQL was more or less spread across the two instances. Whereas with a single instance we had about 65-70k at full load, we now had two instances with a maximum TPS of 35-45k. However, we were surprised by another fact: our dashboards showed that both instances of Taler reached 23k TPS (peak) together, almost double what we normally had (13k). And all this while on the DB node, the main resources also almost doubled. Next, we ran the same experiment also with three instances, which resulted in the database TPS again being evenly spread across the three instances running on the same hardware. However, we again achieved 'only' around the same TPS in Taler as with two instances, despite the CPU load still suggesting that we would have more resources on all nodes available to achieve more.

While puzzled by this behavior, we tested various other things, which eventually lead us towards finding the bug in the wirewatch code (see Section 6.4.10).

Less Performant Nodes

Since we were obviously able to achieve more TPS in Taler with less possible TPS in PostgreSQL, we thought it might be possible to achieve the same TPS in Taler on a less powerful node. So we searched the grid for a node that could only achieve about 45k TPS with PostgreSQL and found that there were such nodes in the Grisou [30] cluster. However, unlike our discoveries with multiple instances, we were only able to achieve 8.1k TPS in Taler, while we achieved 42k TPS with PostgreSQL, so this still did not explain our previous findings. It led us to believe that we might be limited by the PostgreSQL implementation, unless we missed a crucial configuration option. However, we could probably rule this out, since we did not receive any hints in our support requests about a different configuration than the one we already tried.

CentOS

To also rule out a bottleneck that could come from the operating system, we created a second image based on centos-8 with `kameleon` exclusively for the database node. [31] But even this change was not helpful and showed that it is perfectly fine to use debian-11 for the database (and our other nodes).

[30] Grisou: `https://www.grid5000.fr/w/Nancy:Hardware#grisou`

[31] The image definition is also included in the Git repository on taler.net and in the Docker build setup. It can be built with the `--centos` flag when using Docker, or as shown in Chapter 3 for manual builds.

Artificial Network Delay

After all the experiments described above, there was still something to investigate. It could be that latency between the database and the database clients, such as exchange or wirewatch, is causing the TPS to stagnate. To test the impact of latency on our performance, we ran a simple experiment where we created artificial network latency by using *traffic control* (`tc`) [32] on the database node. We did this again while Taler was running at its TPS limit, adjusting the round-trip time several times (see Figure 6.27). The result was quite interesting: while PostgreSQL was immediately affected by any additional latency, we saw that Taler was not much affected by it. This raised the question of why Taler is still so fast when it is actually limited to a third of the original number of queries per second, because we have also already seen that this does not work on nodes reaching less PostgreSQL TPS in general.

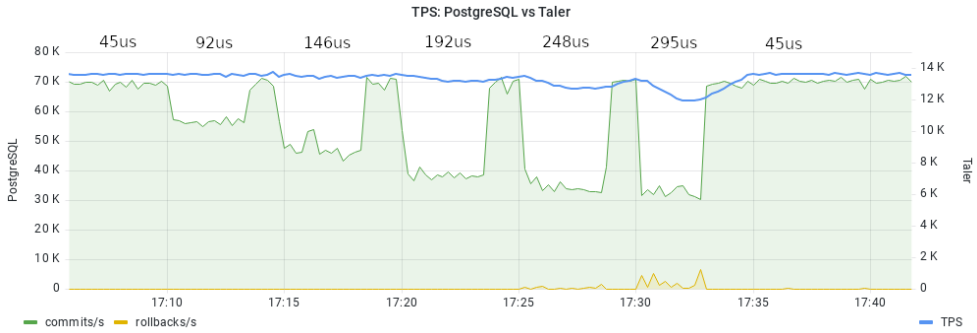


Figure 6.27.: PostgreSQL TPS compared to Taler TPS, while we added various artificial network latencies to the database node using *traffic control* (`tc`). The resulting round-trip times, measured with `ping` between the exchange and the database, are shown in the figure (45us are default with no additional delay). It can be clearly seen that the TPS in PostgreSQL is extremely affected by the additional latency, while Taler does not show a real impact until 4 to 5 times the default round-trip time. But even then, the change is not remarkable compared to what PostgreSQL sees.

This would mean that if the delay is shorter, more queries are issued by a component that are not directly helpful to the TPS in Taler. To investigate this, we stopped the closer and transfer processes to see if PostgreSQL's TPS changed, but according to our expectations, there was no change since they shouldn't have done much anyway. This leaves wirewatch as a possible cause. However, the problem with wirewatch is that this service is required for the experiments to work, since it creates the reserves in the database once it receives the money from the bank. Therefore, we had to find the problem by looking at the code, and found that wirewatch was actually the cause of the TPS fluctuations in PostgreSQL.

6.4.10. Exchange Wirewatch

Normally, wirewatch queries the bank for incoming transfers (withdrawals) with a window size of 1024 transactions and places their reserve in the database. If wirewatch queries the bank too fast, it gets only a subset of these transactions, say 10, for these ten it then performs the transaction in the DB, and once this transaction is finished, it fetches the next transfers from the bank. Now if we add a delay to the DB, that interaction with the DB will take longer, which will cause it to wait longer to query the bank again. So ultimately, we get

[32] Traffic control: `https://en.wikipedia.org/wiki/Tc_(Linux)`

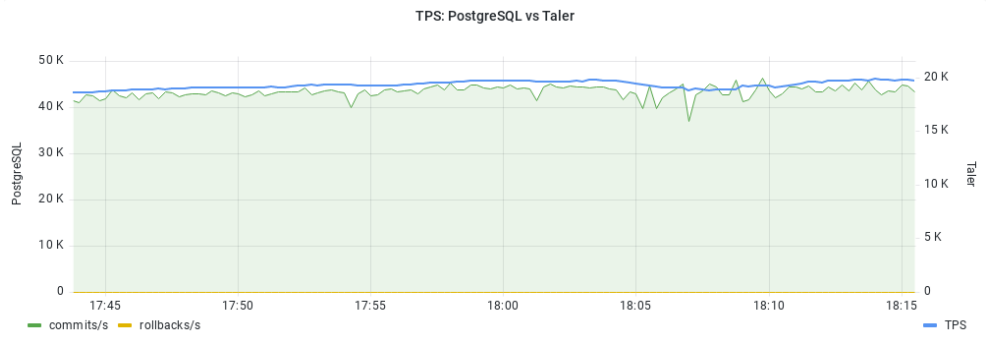


Figure 6.28.: The TPS of PostgreSQL compared to that of Taler when we set the wirewatch to wait before issuing transactions in PostgreSQL. We can see that the total TPS in PostgreSQL is now well below the 65k. Note, however, that we have two wirewatchers for different accounts on independent banks running here, which we did to hopefully identify other issues with the wirewatch implementation.

more transfers per request to the bank, which in turn leads to fewer database transactions. The solution then was to wait a certain amount of time in wirewatch before it starts database interaction, or at least wait except when the window of 1024 inserts is filled. This helped us reduce the number of TPS on the DB well below 65k (Figure 6.28). This change eventually showed us that we could have a lot more transactions in PostgreSQL, so wirewatch is probably the next bottleneck to investigate, as we had now clearly reached the limit of capacity on withdraws since we could now only reach about 6.2k withdrawals per second in experiment doing only reserve creation and withdraw.

Withdrawals Only

Fortunately, this withdrawal bottleneck was resolved fairly quickly. The wirewatch processes allowed some code execution paths that locked shards far into the future, waiting there for transfers that may not have come during our experiments, so the actual transfers were pending for a while. Once we identified and fixed the problem, we were able to achieve about 75k withdrawals per second in batch-withdraw with our *normal* withdrawal amount calculation, while not being anywhere near the DB TPS limit (see Figure 6.29). The next issue, if the withdrawals increase more, might probably become I/O load again soon (Figure 6.30). However, while we achieved much lower DB TPS in some cases, it could again be very different in others (see Figure 6.32 and Figure 6.33). Of course, this means that there is probably still a bug in Taler somewhere, but we would not reach that many withdrawals per second in our usual experiments with the current performance anyway, since we reach the DB TPS earlier when deposits and updates are also included. Nonetheless, this is an issue which must be addressed in the near future.

To see the impact of batch-withdrawals, now that the obvious 6.2k withdrawal bottleneck was gone, we also ran an experiment with sequential withdraws again, which can be seen in Figure 6.31. Clearly batch-withdrawals have a huge impact on the performance, and we hope to see similar results once batch deposit is implemented.

As we ran several experiments with different modifications, we were able to identify more aspects in the wallet code which could be improved, and some rather unexpected ones in the monitoring setup. The identified issues found in these experiments are shown in Sec-

tion 6.4.10.

Note that in the figures mentioned, which represent experiments with withdrawals only, 200 exchange processes are distributed over five nodes. Since we again found an increased optimum in the number of processes, we increased them in small steps from 120 (which yielded about 50k withdrawal TPS in Taler) to 200, which seemed optimal. However, we must also say that we may have made a mistake by starting the exchange processes in running experiments. We 'accidentally' left the number of processes at 200 and ran an additional experiment with deposit, where we again achieved more Taler TPS, while we assumed 120 as optimal - this number we also determined at that time by starting during an experiment. However, since starting an exchange process is expensive (due to DB initialization) it may actually slow down a running experiment rather than increasing it.

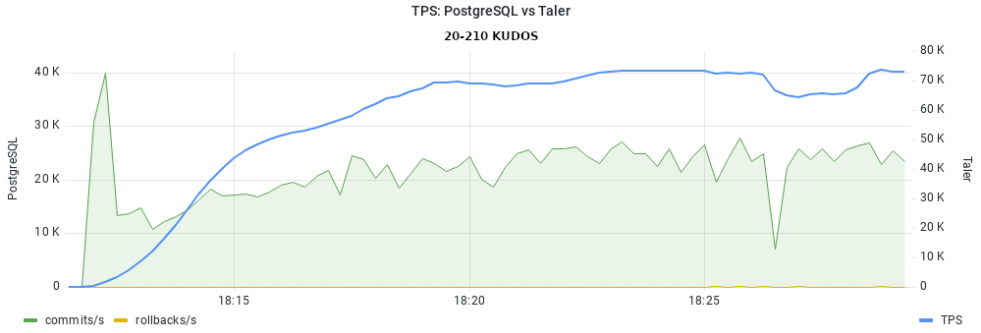


Figure 6.29.: TPS of PostgreSQL and Taler when performing an experiment with batch withdrawals. This figure was created when our usual withdrawal amount calculation was used (20-210 KUDOS) - 200 exchange and 8 wirewatch processes running. Obviously, the DB TPS are not the problem here with only 25k. But we still could not achieve more than 75k withdrawals per second - the most used nodes were five exchanges on 50% CPU and one DB node on 40% CPU, average latencies reported by nginx where 2.25s for *reserves* and 700ms for *batch-withdraw* (4'800 requests per second each).

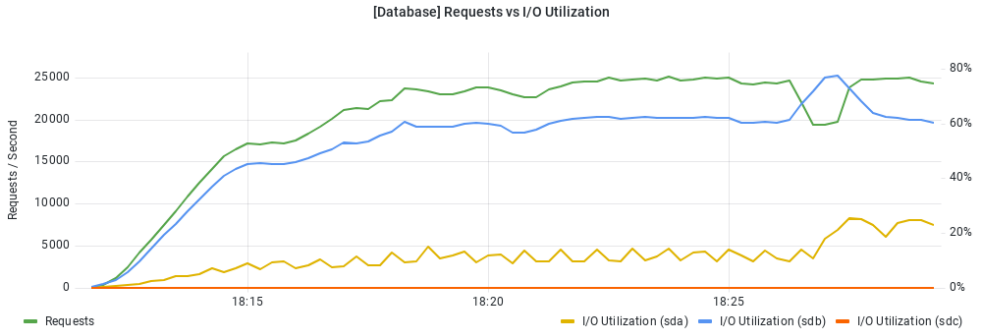


Figure 6.30.: The I/O load in the experiment shown in Figure 6.29, we will obviously be limited by IO again soon (in Grid'5000) if the TPS of withdrawals continue to increase.

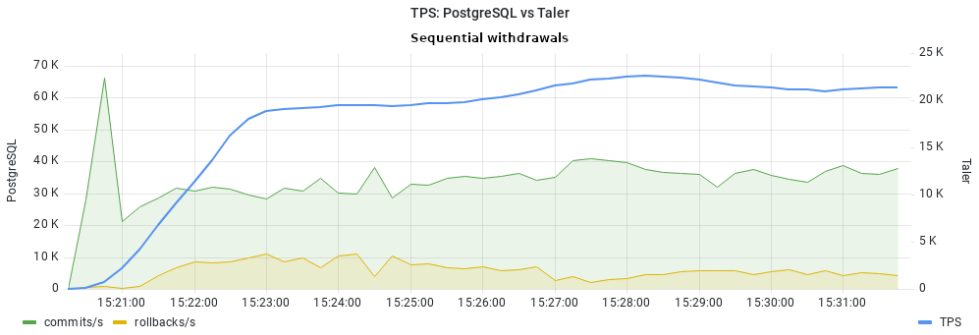


Figure 6.31.: An experiment configured the same as in Figure 6.29 with the only exception that we disabled batch-withdraw and made the withdrawals sequential again. The rollbacks in PostgreSQL's TPS reflect the serialization errors at that point, so you can clearly see that batch withdrawals are indeed crucial for performance. As we can only reach about 22k Taler TPS compared to 75k with batch-withdrawals.

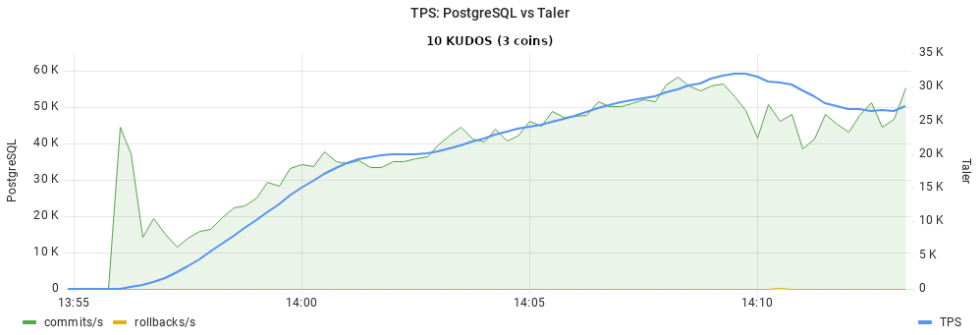


Figure 6.32.: In our attempt to further address the wirewatch problem, we conducted another experiment with batch withdrawals. In this experiment, we used a constant amount for the withdrawal (10 KUDOS), resulting in three coins. In fact, we were able to achieve much less TPS in Taler than in Figure 6.29, while in PostgreSQL we had more relative TPS (which is explained by the fact that there are more single withdrawal operations and thus more requests triggering DB transactions). But we identified quite different issues again: the wallet clients once more became the bottleneck (Section 6.4.10) and we noticed that our metrics might be an issue too (Figure 6.38). During this experiment, we had about 10'000 requests to *reserves* and *batch-withdraw* ($*3\ coins \approx 30k\ withdrawals$) with a corresponding latency of 1.5s and 70ms (at 30k Taler TPS).

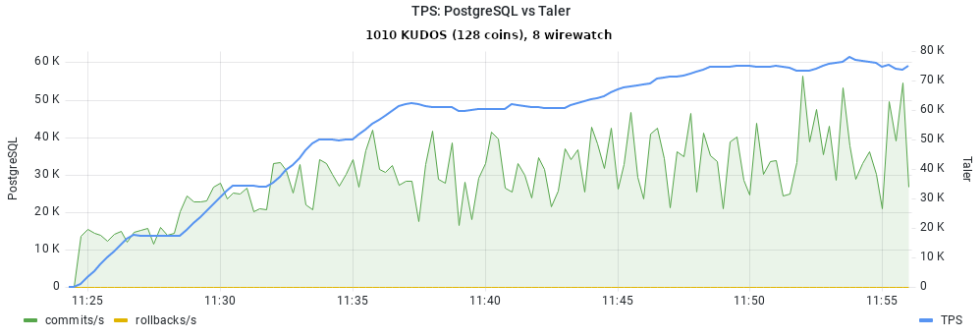


Figure 6.33.: We ran the same experiment as in Figure 6.32 again with a larger constant amount of 1010 KUDOS (128 coins), we now again have fewer DB transactions (on average) compared to the previous experiment, since a single request to *batch-withdraw* now receives 128 coins. In theory, however, this should result in much higher withdrawals per second compared to DB TPS, as Figure 6.29 shows fewer DB transactions with about the same amount of Taler TPS, while making many more requests - here it is about 590 requests to *reserves* and *batch-withdraw* with latencies of 1.6 and 1.2s, respectively. In this figure where eight wirewatch processes running, which we reduced to one in Figure 6.34, which resulted in a completely different figure.

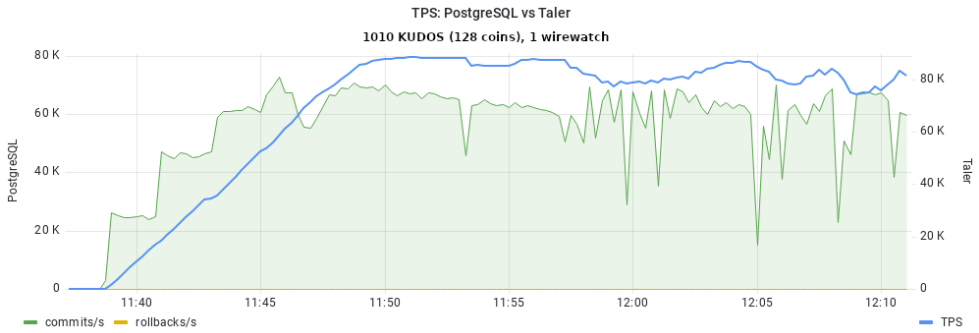


Figure 6.34.: The same experiment as in Figure 6.34 but with a single wirewatch process rather than eight. We can see that the DB TPS are at our limit again while we would again expect much less than in Figure 6.29, since there are only about 675 requests to *reserves* and *batch-withdraw* respectively, withdrawing 128 coins each. This and the previous figures probably indicate a further bug in the wirewatch logic which must be inspected soon. (Reported latencies by Nginx: 700ms on *reserves* and 1.3s on *batch-withdraw*)

Complete Experiments

Subsequent full experiments let us reach 23k+ TPS in Taler, with the DB again very close to our historical limit (Figure 6.35). However, only 120 Exchange processes were connected at that time and four wirewatch processes were running. Together with a change in the exchange, halving the number of DB transactions in the deposit operation, we eventually achieved 28.5k TPS with Taler while having only two wirewatch but 200 exchange processes running (Figure 6.36). Unfortunately, there was not enough time left in this work to figure out which change was most important in increasing the TPS. Therefore, these experiments may need to be repeated incrementally to see what factor was most useful.

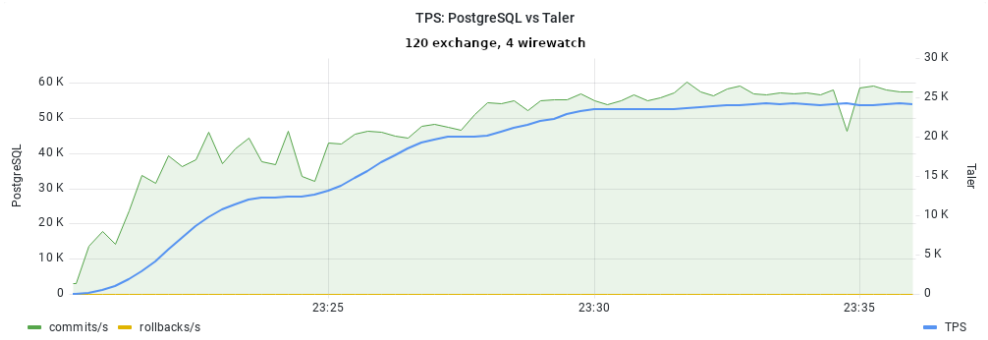


Figure 6.35.: The TPS of Taler relative to the TPS of PostgreSQL in a full experiment (not just withdrawals). While we were able to get much more withdrawals, we were not able to increase the TPS in Taler much, as we again came very close to the limit of PostgreSQL. At the time of this experiment there were 4 wirewatch and 120 exchange processes running.

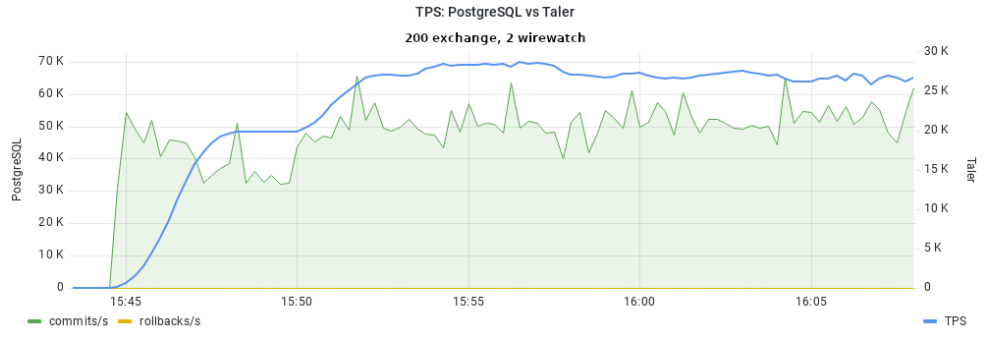


Figure 6.36.: The same experiment again with a different configuration: only 2 wirewatches, but 200 exchanges running. Together with a change in the exchange logic to halve the number of DB transactions, we finally reached 28.5k TPS. We can see that there are fewer TPS than in Figure 6.35, unfortunately we haven't yet figured out which of the changes (more exchanges, fewer wirewatchers, or halving the DB transactions) contributed most to the increase in TPS. But before we can verify this, we probably need to identify the remaining wirewatch problem first.

Identified Bottlenecks

During the experiment shown in Figure 6.32, we were able to locate three additional opportunities for improvement, which fortunately did not really affect our results so far. They are explained below.

Figure 6.37 shows the memory of a wallet node during the experiment in Figure 6.32. At about 14:10, several wallet nodes ran out of memory and restarted. This can be seen directly from Taler's TPS, which decreases around the same time. Perhaps we could have achieved a few more TPS in Taler if this had not happened. However, we decided not to check this because, first, the load of the experiment is not realistic and, second, we had already allocated 70 client nodes. Fortunately, this has not yet affected previous experiments, as we never had that many wallet processes running [33]. Although full experiments have not yet reached this point, it could still become problematic as we examined older experiments and found that memory was constantly growing. One solution we have found is to reduce the number of iterations a wallet process performs so that it restarts more often and frees the memory, but it may still be necessary to make the processes lighter.

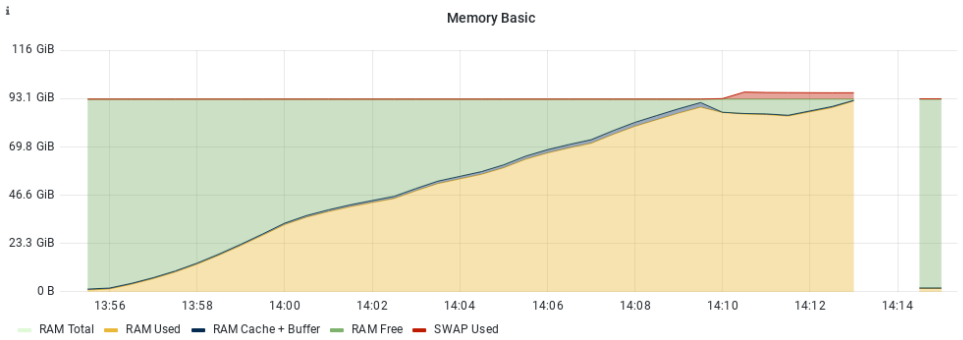


Figure 6.37.: The memory usage of a wallet node during the experiment in Figure 6.32. We had 70 client nodes on which we had to launch about 900 wallets each, since many more were needed to achieve more TPS (as fewer coins were withdrawn per request). At roughly the same time the wallets run out of memory we see the Taler TPS in Figure 6.32 decrease. Although the load is not very realistic, we still see this as new issue for improvement in the wallet clients.

The other, more surprising issue we noticed was a difference in the number of requests reported by the Nginx servers and those reported by Promtail (Figure 6.38). The number of requests from Nginx is simply a counter of the Nginx process being exported to Prometheus, where we cannot see the actual URLs. Therefore, we used Promtail to evaluate the request statistics per endpoint from the Nginx logs. These statistics were used, among other things, to calculate the number of Taler TPS (excluding batch withdrawals, for which we added metrics to the exchange). At first, we thought Promtail was the problem, since a single process was responsible for all logs from our experiments. But when we couldn't find anything in the logs, we looked on the Nginx nodes. There we found that indeed `rsyslog` is the problem [34]:

```
Jun  9 14:03:14 dahu-20 rsyslogd: main Q:Reg: high activity -
  starting 1 additional worker thread,
  currently 1 active worker threads.
  [v8.2102.0 try https://www.rsyslog.com/e/2439]
```

At the exact time this log message appeared (in `/var/log/messages`), the reported number of requests started to diverge (Figure 6.38). We quickly went through the past snapshots to see if this problem has been around for a while. Fortunately, it hasn't been a relevant problem yet. However, since it could obviously become one soon we started adding new

[33] In this experiment, there were 900 per node, which was required due to a completely different load situation

[34] We first log locally to `rsyslog`, since the logs are also sent to the NFS and not only to Promtail. Although Nginx supports logging to remote servers, it was not possible to log directly to Promtail, as we got errors trying to do so.

metrics to the exchange processes. These count the number of non-idempotent operations to get around the unreliable logging statistics. However, the logs are still important to compute various statistics for request times, etc., but for these we would not need *all* the log lines and could analyze them using a sample of logs [35]. This would in turn also reduce the snapshot sizes for recovering experiments, since the Nginx logs are usually the largest.

The discovery of this problem also led us, in turn, to another problem in the wallet clients. Based on the number of requests reported, we found that the command-line version of the clients were not caching the results of the `wire`, `terms`, and `keys` endpoints. They were requesting these endpoints again on each new iteration, which generated a tremendous amount of unnecessary network traffic for Nginx. Fortunately, Nginx had respected the `Expires` headers and did not forward the requests to the exchange. In the experiments here (withdrawing 3 coins) we see 3.2 Gb/s of bandwidth on the Nginx nodes compared to 900 Mb/s on the exchanges, clearly a huge difference and thus also an issue to further improve client performance.

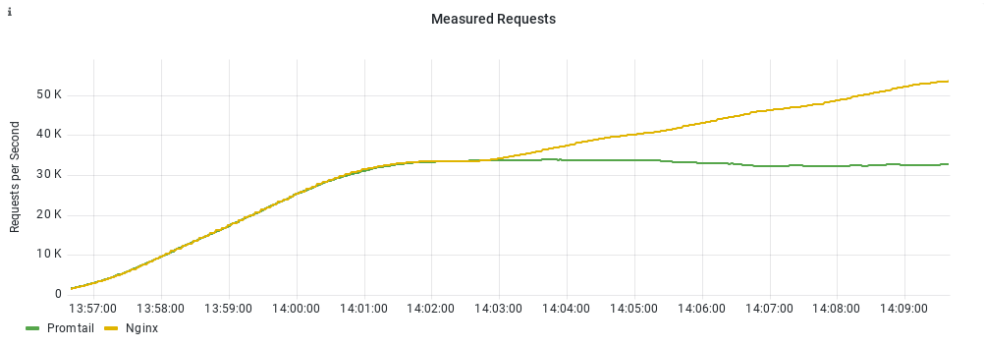


Figure 6.38.: The deviating number of requests per second observed in our experiments. This is, of course, a problem when measuring the TPS for Taler, since these are calculated from the logs. Fortunately, we identified this problem before it became problematic for our experiments. In the experiment here (Figure 6.32), we had many more requests because only 3 coins were withdrawn in a request (about 10,000 per second for each endpoint). Fortunately, all previous results are still valid as we were able to verify this in our snapshots.

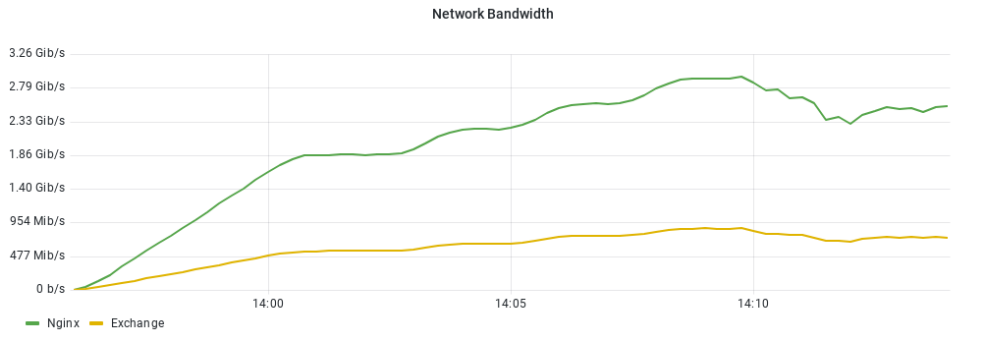


Figure 6.39.: The sum of network traffic on the respective nodes hosting wallets and Nginx proxies. While Nginx apparently caches the responses to `wire`, `terms`, and `keys`, the CLI wallets do not yet do so. Fixing this issue could further improve client performance (at least).

[35] Nginx conditional logging: `https://www.nginx.com/blog/sampling-requests-with-nginx-conditional-logging/`

6.4.11. Conclusion

We assumed too early that PostgreSQL could be the bottleneck. While we believe this to be true in some cases, e.g. when we reach 65k DB TPS in our experiments, we now know that this is not true for experiments where we only perform withdrawals. Since there, after an 'obvious' wirewatch bottleneck has been fixed, we no longer reach the DB limit. However, there are still bottlenecks and behaviors that are not yet fully identified, e.g. why we still achieve much different DB TPS than expected (e.g. in withdraw only). In the experiments conducted, we also identified that there is quite a difference in the optimal number of exchange clients connected to the DB, while it was 120 for complete experiments for some time, it increased again and is now 200. It could well be that we are limited by the number of exchange processes in our experiments, since we may need more, but are limited by the optimal number of connections to the DB in the various loading situations. Again, we have plenty of room for further investigation. Unfortunately, this does not include testing another database implementation, as the combination of features required in Taler is only available in PostgreSQL. But that's not the only reason: there is simply too much SQL code that would have to be changed to make it work in another DB. At least this was not an option for this work yet.

6.5. Partitioning and Sharding

Partitioning in PostgreSQL is a way to split large tables into smaller ones. While the main table is called *partitioned table* and has no physical memory, the smaller tables are then called *partitions* and that is where the data is actually stored. This partitioning can be done by defining a distribution key (also called a sharding or partitioning key), which PostgreSQL uses to decide which partition to write data to. This should ultimately improve query performance by allowing smaller tables to be searched instead of the full table (if partition pruning is enabled - which is default [36]). Declarative partitioning is used in PostgreSQL to create such partitioned tables. This declaration must include one of the partitioning methods (`Range`, `List` or `Hash`) and a partition key. [44]

An example of a partitioned table is shown in Listing 6.2, its visualization in Figure 6.40.

```
CREATE TABLE taler (
    id INTEGER,
    value VARCHAR(20)
) PARTITION BY RANGE(id);

CREATE TABLE taler_1
  PARTITION OF taler
  FOR VALUES FROM (1) TO (10);

CREATE TABLE taler_2
  PARTITION OF taler
  FOR VALUES FROM (11) TO (20);
```

Listing 6.2: Simple partitioned table that shows how partitions (using the range method) can be created with PostgreSQL's declarative partitioning. All rows inserted into the table `taler` are actually inserted into one of `taler_1` or `taler_2`, depending on which `id` is used in the inserted row.

[36] Partition pruning: https://www.postgresql.org/docs/14/runtime-config-query.html#GUC-ENABLE-PARTITION-PRUNING

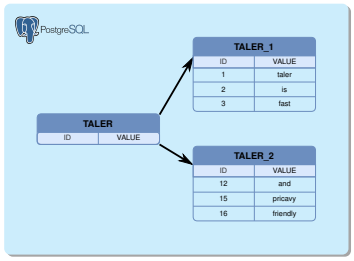


Figure 6.40.: PostgreSQL's partitioning visualized. The table 'TALER' is divided into several smaller partitions, which can be located on different hard disks.

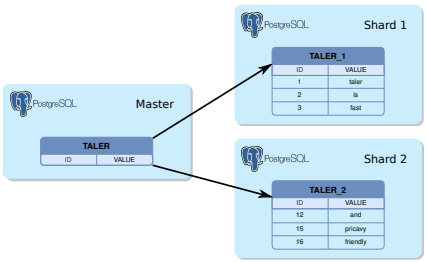


Figure 6.41.: The natural extension of partitioning is called sharding. It includes the relocation of partitions to separate servers to ease the load on the master node.

Declarative partitioning is a relatively new concept in PostgreSQL (introduced in version 10) and offers another advantage besides better query performance, namely that partitions can be stored on separate disks [45].

Sharding is based on the same concept, with the difference that the partitions are now located on a different server and are therefore also called *foreign tables*. Thus, a PostgreSQL database can now be distributed horizontally, which further increases storage capacity. However, there are also some disadvantages, namely that there are no real `UNIQUE` constraints except for the distribution key. `UNIQUE` constraints have to be added to each partition separately, which results in the respective columns being unique only for each of the partitions and not for the whole table [44]. Also, there is no longer a useful option for foreign keys, as we cannot reference a table on another database server. This is only partially useful, as it would still be possible to point to a partition or replicated table on the same node, but we cannot be sure that the data we need is there (except for replicated tables).

While there are multiple extensions which can be used to distribute a PostgreSQL database across multiple nodes, we decided to use to relatively new feature of PostgreSQL itself, called the Foreign Data Wrapper (`postgres_fdw`) [37]. Other extensions are at risk of being discontinued, such as Postgres-XL [38] (although it is not officially stated, the last commit on sourceforge [39] is from 2016), or `pg_shard` [40] which was integrated in the Citus extension [41]. Most of the time, however, such extensions don't support the latest version of PostgreSQL until they've made a new patch, so they don't support the latest features either, which could become interesting. There are other extensions, but the problem remains the same.

6.5.1. Implementation

Citus offers a considerable number of features, one of which is the automatic creation of child tables [42]. However, this function is not available with `postgres_fwd`. Therefore, we had to implement a similar function ourselves. We did this with simple SQL, mainly using functions called when initializing the Taler database from `taler-exchange-dbinit`. The following flags were added to the utility:

[37] postgres_fdw: `https://www.postgresql.org/docs/13/postgres-fdw.html`
[38] Postgres-XL: `postgres-xl.org`
[39] Postgres-XL Repo: `https://sourceforge.net/projects/postgres-xl/`
[40] pg_shard: `https://github.com/citusdata/pg_shard`
[41] Citus: `https://citusdata.com`
[42] Citus table creation: `https://docs.citusdata.com/en/stable/develop/api_udf.html#create-distributed-table`

Master Node Flags

- `-P <NUM>`:
  Create partitioned tables with `NUM` partitions.
- `-F`:
  Create foreign servers instead of partitions (including foreign tables with suffixes `1-NUM` pointing to the remote tables on the shard nodes).

Shard Node Flags

- `-S <NUM>`:
  Create tables on a shard with index `NUM` (creates tables with suffix `NUM`).
- `-R <NUM>`:
  Drop the database on a shard node with index `NUM` (drops all tables with suffix `NUM`).

To avoid duplicate code, we created a function template (shown in Listing 6.3 and 6.4) that allows us to create *partitioned tables* and *partitions* from the same table definition.

```
CREATE OR REPLACE FUNCTION create_partitioned_table(
   IN table_definition VARCHAR
  ,IN table_name VARCHAR
  ,IN main_table_partition_str VARCHAR
  ,IN shard_suffix VARCHAR DEFAULT NULL
)
RETURNS VOID
LANGUAGE plpgsql
AS $$
BEGIN

  IF shard_suffix IS NOT NULL THEN
    table_name=table_name || '_' || shard_suffix;
    main_table_partition_str = '';
  END IF;

  EXECUTE FORMAT(
    table_definition,
    table_name,
    main_table_partition_str
  );

END
$$;
```

Listing 6.3: Method added to common-OOO1.sql to create partitioned tables and tables on the shard nodes (if `shard_suffix` is NULL, then a partitioned table is created with the partitioning method passed with `main_table_partition_str`, else the table on the shard is created). The usage is shown in Listing 6.4.

```
CREATE OR REPLACE FUNCTION create_table_wire_targets(
  IN shard_suffix VARCHAR DEFAULT NULL
)
RETURNS VOID
LANGUAGE plpgsql
AS $$
BEGIN

  PERFORM create_partitioned_table(
```

```
    'CREATE TABLE IF NOT EXISTS %I'
      '(wire_target_serial_id BIGINT '
      '   GENERATED BY DEFAULT AS IDENTITY'
      ',wire_target_h_payto '
      '   BYTEA PRIMARY KEY CHECK (LENGTH(wire_target_h_payto)=32)'
      ',payto_uri VARCHAR NOT NULL'
      ',kyc_ok BOOLEAN NOT NULL DEFAULT (FALSE)'
      ',external_id VARCHAR'
    ') %s ;'
    ,'wire_targets'
    ,'PARTITION BY HASH (wire_target_h_payto)'
    ,shard_suffix
  );

END
$$;
```

Listing 6.4: The new method for creating tables in the Taler database, shown here as an example with the table `wire_targets`. This method can now be called on the master and the shard node(s). If the `shard_suffix` is NULL, the partitioned table on the master node is created, otherwise a partition is created on the shard with index `shard_suffix` (as shown in Listing 6.3).

Since not all methods and tables are needed on each node, we have written separate SQL files which are concatenated into SQL files for the respective nodes (shard and master) as soon as `make` is called. For example, the master must create the partitioned tables with a partition method, the foreign partitions and the foreign servers that host these partitions. However, only the tables that will serve as partitions must be created on the shards.

To initialize a sharded database the following steps must be performed:

Master Node

Before the distributed database can be initialized, some configuration values need to be set. We decided to take a simple approach by using the same credentials and domain for all shards. This should not be a problem since the databases will probably not be accessible from the outside, but only through an Exchange on the same network. Therefore, we added the following configuration options to `taler.conf`:

- `SHARD_DOMAIN` (section `exchange`):
  The domain in which all shards are located. The shards must all match the pattern shard-<N>.SHARD_DOMAIN, where N is the index of the respective shard. This index must then also be used when initializing the shard database.
- `SHARD_REMOTE_USER` (secret section `exchangedb-postgres`):
  The username that can be used to log in to the shard and access the partitions; this must be the same for all shards and must be allowed for at least the remote master node. [43]
- `SHARD_REMOTE_USER_PW` (secret section `exchangedb-postgres`):
  The password used to log in to the shard and access the partitions; it must also be the same for all shards.

Once these values are configured, the following command can be run to create two foreign servers and thus partitioned tables with two partitions each. Note that the command must

[43] PostgreSQL remote access: `https://www.postgresql.org/docs/current/auth-pg-hba-conf.html`

be run as user `postgres` because `postgres_fdw` initially requires superuser privileges. The correct permissions are granted to `taler-exchange-httpd` by the SQL code during initialization.

```
$ sudo -u postgres taler-exchange-dbinit -P 2 -F
```

This will create two foreign servers (shard-1.SHARD_DOMAIN and shard-2.SHARD_DOMAIN) and all the tables including their foreign partitions named ‹table-name›_1 and ‹table-name›_2. However, the foreign tables do not exist yet and must be created on their respective shards:

Shard Node(s)

This step can be done before or after the step that must be performed on the main node. It is only important that it is performed before a query to the database is executed, otherwise the query will fail.

shard-1:

```
$ sudo -u taler-exchange-httpd taler-exchange-dbinit -S 1
```

shard-2:

```
$ sudo -u taler-exchange-httpd taler-exchange-dbinit -S 2
```

The database is ready for use as soon as this command has been successfully executed on all shards (plus the one on the master).

6.5.2. Results

While partitioning seemed to work well, we ran into problems with sharding. First, the overall TPS performance dropped. While we had about 13k TPS in Taler with a partitioned DB, it was only about 9k when using the same architecture for sharding. Some initial experiments showed a full network bandwidth utilization of 10 Gbps, with only about 300 TPS in Taler, rendering it unusable. Our investigation of the problematic queries revealed that there was one particular issue:

Suboptimal Queries

Sharding may be difficult to set up, it requires quite a bit of work to transition a non-partitioned schema to a partitioned one. This includes picking the right sharding (distribution) keys for each table. While most sources propose to take tenant based solution, this cannot be applied to Taler, as it does not have a concept of accounts.

Tables should be partitioned in a way that queries require access to the smallest number of partitions, usually achieved by having the WHERE clause should select on the partition key. If this is not the case, PostgreSQL may need to scan each partition to compute the result, reducing performance [44].

To enable partition-wise joins, the distribution keys must be those that appear in the join condition and be exactly aligned, otherwise multiple partitions may need to be scanned to compute the join.

Denormalization

A first optimization we performed was to denormalize the database. For example, the `wire_targets` table had to be partitioned by `h_payto`, but some joins (and references) used the `wire_target_serial_id` column. As a result, those queries had to scan each partition to get the correct result. We changed the queries to use `h_payto` in joins instead of the serial number, replacing the serial number by the `h_payto` in referring tables and in one place in the protocol. To reduce the impact this had on storage space, we also changed the length of `h_payto` to 32 bytes instead of 64.

However, it turned out that this kind of transformation did not always improve performance as desired, and that the network load was still too high. The problem was the way PostgreSQL performs joins (see Listing 6.5). It may select all rows from a table and then remove the ones that don't match once both results are available, even if filtering one half of the join first would have resulted in much better utilization of indices and singleton joins. We first tried to fix the queries by enforcing join sequences with explicit joins [44], hoping to be able to filter results before the joins, but PostgreSQL continued to execute the queries in a suboptimal order.

Materialized Common Table Expressions

Ultimately, we were able to fix the problem by using *Common Table Expressions* (CTEs) that define temporary tables that exist only for the executed query [46]. The new query execution with CTE is shown in Listing 6.6. Although the same could be achieved with sub-queries, we chose to use CTEs because they are generally more readable and by using `MATERIALIZED` we force PostgreSQL to execute the CTE first instead of "optimizing" it inline. We must disable inline optimization as that would often result in the same inefficient query plan that we get with sub-queries or non-materialized CTEs [47]:

> " *However, if a WITH query is non-recursive and side-effect-free (that is, it is a SELECT containing no volatile functions) then it can be folded into the parent query, allowing joint optimization of the two query levels. By default, this happens if the parent query references the WITH query just once, but not if it references the WITH query more than once. You can override that decision by specifying MATERIALIZED to force separate calculation of the WITH query, [...]* " [46]

Using CTEs with `MATERIALIZED` brings the necessary determinism to query execution, so that we can be sure that a query plan is not slowed down by a bad estimate or unexpectedly bad decisions by the query planner. Of course, this change is often only possible after the denormalization described above has been performed, allowing the correct partitions to be hit. Thus, with the MATERIALIZED approach, PostgreSQL accesses only one partition for one or more where clauses and retrieves only the required rows from the remote partition, drastically reducing the network load.

Figure 6.42 shows the effects of the query rewrites on a sharded database.

[44] Explicit joins: `https://www.postgresql.org/docs/current/explicit-joins.html`

Figure 6.42.: The network traffic on the master node in a sharded database before and after the queries with materialized CTEs. Before, it is constantly increasing because entire tables (which are constantly growing) are fetched from the respective shards. The fix clearly allows the network load to be kept constant. Both figures were created when 10 wallets were running and when we had 53 TPS in Taler.

```
EXPLAIN ANALYZE
SELECT kyc_ok, wire_target_serial_id
  FROM reserves_in
  JOIN wire_targets ON (wire_source_h_payto = wire_target_h_payto)
WHERE reserve_pub='\xdf6bceef72ae1f37ccb7acae2ada7bf596df1f6fccd1f8526e2c43709836fbba';
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------
Nested Loop  (cost=0.41..10.48 rows=1 width=9) (actual time=0.086..0.093 rows=1 loops=1)
Join Filter: (reserves_in.wire_source_h_payto = wire_targets.wire_target_h_payto)
Rows Removed by Join Filter: 1
->  Index Scan using reserves_in_1_pkey on reserves_in_1 reserves_in
    (cost=0.41..8.43 rows=1 width=33) (actual time=0.058..0.060 rows=1 loops=1)
      Index Cond: (reserve_pub = '\xdf6bceef72ae1f37ccb7acae2ada7bf596df1f6fccd1f8526e2c43709836fbba'::bytea)
->  Append  (cost=0.00..2.03 rows=2 width=41) (actual time=0.013..0.022 rows=2 loops=1)
      ->  Seq Scan on wire_targets_1
          (cost=0.00..1.01 rows=1 width=41) (actual time=0.012..0.013 rows=1 loops=1)
      ->  Seq Scan on wire_targets_2
          (cost=0.00..1.01 rows=1 width=41) (actual time=0.005..0.006 rows=1 loops=1)
Planning Time: 0.598 ms
Execution Time: 0.234 ms
```

Listing 6.5: Query execution of a join by PostgreSQL. It shows that both tables involved are scanned before the join is actually executed - this can be seen by the `Rows Removed by Join Filter` statement. This is fatal for sharding if joins are not executed on the shard nodes, which is not possible here since the tables involved are not sharded on the same column. So PostgreSQL has to get the tables from the shards to the master node and then execute the join locally. If this happens multiple times, the network capacity is quickly exceeded. Clearly this query was not contributing much in either case as the whole table contains two rows only, but the one shown in the appendix (Listing A.1 and A.2) is. (The query plan shown here is executed on a partitioned rather than a sharded database, but the result remains the same) - Note: Explain shows an inverted tree and must be read from inside out (innermost node is executed first) [a].

[a] Explain: https://www.postgresql.org/docs/13/using-explain.html#USING-EXPLAIN-BASICS

```
EXPLAIN ANALYZE
WITH reserves_in AS MATERIALIZED (
  SELECT wire_source_h_payto
    FROM reserves_in
  WHERE reserve_pub='\xdf6bceef72ae1f37ccb7acae2ada7bf596df1f6fccd1f8526e2c43709836fbba'
)
SELECT kyc_ok, wire_target_serial_id
  FROM wire_targets
WHERE wire_target_h_payto = (
  SELECT wire_source_h_payto
    FROM reserves_in
);
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------
Append  (cost=8.45..10.48 rows=2 width=9) (actual time=0.091..0.095 rows=1 loops=1)
 CTE reserves_in
   ->  Index Scan using reserves_in_1_pkey on reserves_in_1 reserves_in
       (cost=0.41..8.43 rows=1 width=33) (actual time=0.060..0.063 rows=1 loops=1)
         Index Cond: (reserve_pub = '\xdf6bceef72ae1f37ccb7acae2ada7bf596df1f6fccd1f8526e2c43709836fbba'::bytea)
  InitPlan 2 (returns $1)
    ->  CTE Scan on reserves_in reserves_in_1
        (cost=0.00..0.02 rows=1 width=32) (actual time=0.065..0.068 rows=1 loops=1)
  ->  Seq Scan on wire_targets_1
      (cost=0.00..1.01 rows=1 width=9) (never executed)
        Filter: (wire_target_h_payto = $1)
  ->  Seq Scan on wire_targets_2
      (cost=0.00..1.01 rows=1 width=9) (actual time=0.013..0.015 rows=1 loops=1)
        Filter: (wire_target_h_payto = $1)
Planning Time: 0.643 ms
Execution Time: 0.276 ms
```

Listing 6.6: Query execution when using CTEs, we can now clearly see that `reserves_in` is scanned first and then the result is used in the second phase to select from `wire_targets`. Due to our previous change to include `h_payto` as join key instead of the serial_id only one partition is hit then. Thus drastically reducing the network load.

Materialized Indices

Similar problems occurred with queries such as `recoup_by_reserve`, however those could not easily be fixed with this approach, as some joins cannot use the partition key. Thus, we created something we named *materialized indexes*.

Let's take the tables `reserves_out` and `reserves`, which are used in this query, as an example. `reserves_out` is partitioned by `h_blind_ev` while `reserve_pub` is used for `reserves`. If we now join those two tables like it was done initially in `recoup_by_reserve`:

```
SELECT *
  FROM reserves
  JOIN reserves_out USING (reserve_uuid)
WHERE reserve_pub=$1;
```

Listing 6.7: Example query which performs bad because partition keys do not align and thus hits all partitions.

we will still end up scanning each partition of `reserves_out` even if we used materialized CTEs, as `h_blind_ev` is not present in `reserves` (which is the partition key of `reserves_out`). The usual approach would be to define an additional index on `h_blind_ev`, but PostgreSQL does not allow indices to span partitions. However, the join would still have to hit multiple partitions.

To overcome this problem, we added additional tables. We call these *materialized indexes* as the tables are basically an index from the desired key to the partition key. These materialized index tables contain two columns, one of each table, and provide a mapping for the partitions. In the example above, the table looks basically like this:

```
CREATE TABLE reserves_out_by_reserve (
  reserve_uuid INT8,
  h_blind_ev BYTEA
) PARTITION BY HASH (reserve_uuid);
```

Listing 6.8: The table we called *materialized index* to align partitions of reserves and reserves_out.

These materialized indexes are populated and unpopulated by triggers on INSERT, UPDATE and DELETE in the respective main table (in this case `reserves_out`). Originally, we had hoped that when joining these three tables, we could limit the number of partitions affected, since the mapping is now given with the materialized index, e.g.:

```
WITH res AS MATERIALIZED (
  SELECT *
    FROM reserves
  WHERE reserve_pub=$1
)
SELECT *
FROM reserves_out ro
  JOIN (
    SELECT *
      FROM reserves_out_by_reserve
    WHERE reserve_uuid = (
      SELECT reserve_uuid
        FROM res
    ) robr
) ON (ro.h_blind_ev = robr.h_blind_ev);
```

Listing 6.9: Example query showing the usage of the materialized indexes to better match partitions.

However, at that time we did not know about PostgreSQL's join behavior with distributed tables.

Dealing With Non-Singleton Results

Specifically, since `reserves_out_by_reserve` returns multiple rows, we cannot use a direct filter with equality. But once we do a `JOIN` or use `WHERE IN`, PostgreSQL does a join which always scans all partitions (as shown before). However, when a direct filter with equality can be used, the number of partitions hit and rows received can be minimized.

So we had to find another way for queries like this to hit only the needed partitions. We solved this with SQL functions like the one shown in Listing 6.10.

```
CREATE OR REPLACE FUNCTION do_recoup_by_reserve(
  IN res_pub BYTEA
)
RETURNS TABLE
(
  denom_sig            BYTEA,
  denominations_serial BIGINT,
  coin_pub             BYTEA,
  coin_sig             BYTEA,
  coin_blind           BYTEA,
  amount_val           BIGINT,
  amount_frac          INTEGER,
  recoup_timestamp     BIGINT
)
LANGUAGE plpgsql
AS $$

DECLARE
  res_uuid BIGINT;
  blind_ev BYTEA;
  c_pub    BYTEA;

BEGIN

  SELECT reserve_uuid
    INTO res_uuid
    FROM reserves
```

```
    WHERE reserves.reserve_pub = res_pub;

    FOR blind_ev IN
      SELECT h_blind_ev
        FROM reserves_out_by_reserve
      WHERE reserves_out_by_reserve.reserve_uuid = res_uuid
    LOOP

      SELECT robr.coin_pub
        INTO c_pub
        FROM recoup_by_reserve robr
      WHERE robr.reserve_out_serial_id = (
        SELECT reserves_out.reserve_out_serial_id
          FROM reserves_out
        WHERE reserves_out.h_blind_ev = blind_ev
      );

      RETURN QUERY
        SELECT kc.denom_sig,
               kc.denominations_serial,
               rc.coin_pub,
               rc.coin_sig,
               rc.coin_blind,
               rc.amount_val,
               rc.amount_frac,
               rc.recoup_timestamp
        FROM (
          SELECT *
          FROM known_coins
          WHERE known_coins.coin_pub = c_pub
        ) kc
        JOIN (
          SELECT *
          FROM recoup
          WHERE recoup.coin_pub = c_pub
        ) rc USING (coin_pub);

    END LOOP;
  END;
  $$;
```

Listing 6.10: The function we created to perform the query `recoup_by_reserve`. This implementation is in our opinion the most efficient method in a partitioned database. Our general approach using `MATERIALIZED` does not work here because we would have to perform a join instead of a single filter with equality, which as we know fetches all rows from a shard. Therefore, this function splits the join into several single filter statements with a loop whose results are returned at the end as a table. You might think that the last part of the function could be pushed down to the shard for a join, since both `recoup` and `known_coins` are partitioned by `coin_pub`. However, PostgreSQL only pushes queries down if they contain a `WHERE` clause, which unfortunately we don't have here [48].

Validation

First experiments using this query showed that it works well, but we have to keep in mind that in the experiments the `recoup` table was always empty, which means that we still have to investigate the performance when there is actually data. However, it is not so easy to debug function calls with PostgreSQL, because in `auto_explain` or `EXPLAIN ANALYZE` generally only the function call is logged, not its content. However, it can still be logged by setting the proper configuration:

```
LOAD 'auto_explain';
SET auto_explain.log_min_duration = 0ms;
SET auto_explain.log_nested_statements TO true;
SET auto_explain.log_verbose TO true;

SELECT * FROM do_recoup_by_reserve('\x...');
```

Listing 6.11: These are the steps that need to be performed to log the statements executed within an SQL function in PostgreSQL - shown here in an interactive PostgreSQL session with `psql`. The generated output of `auto_explain` can then be read from the PostgreSQL logs. Note that the queries are logged as independent queries but with a `CONTEXT` referring to the function they belong to.

Slow Master Node

When we first tried sharding, we found that we achieved less TPS than using a single-node database. While we were hoping to gain some performance, we only got to about 9k TPS in Taler [45] (with 20k DB TPS in the master PostgreSQL instance and a CPU load of about 20%). However, we were able to identify the problem relatively quickly by running the following query on the shard servers:

```
SELECT wait_event, wait_event_type, application_name
  FROM pg_stat_activity
WHERE state='idle in transaction'
  AND wait_event is not NULL
  AND datname='taler-exchange';

 wait_event | wait_event_type | application_name
------------+-----------------+------------------
 ClientRead | Client          | postgres_fdw
 ClientRead | Client          | postgres_fdw
 ClientRead | Client          | postgres_fdw
 ClientRead | Client          | postgres_fdw
...
```

Listing 6.12: The query executed to see what is slowing down our database in a sharded setup.

The output shows that the shards are mostly waiting in state `ClientReads`, which in PostgreSQL means that the shards are waiting for the master to provide input [49].

We believe that the reason is that PostgreSQL is not able to push entire transactions to shards based on some partition key, which would ease the load on the master node. As various queries inherently involve two or more partitions, the master node remains responsible for organizing the process of fetching the required rows from different shards and merging them. Consequently, the shards are not used to offload a significant part of the computation, and mostly wait for simple queries to be delegated to them. Consequently, due to the extra latency required to communicate with the shards, sharding lowers the TPS seen in Taler.

However, we should stress that we introduced horizontal distribution not only to improve performance, but also to increase the storage capacity of our database. After all, in a production system running a large number of transactions for many years might reach the limit of hard disks that can be installed on a single node. Thus, it is important that sharding is possible to scale the storage capacity, even if it does not help scale the transaction rate.

[45] At the time we had about 13.6k in a partitioned setup

6.5.3. Final Performance

While we have found that sharding is not as powerful as we would have liked, it can still be handy if one considers the amount of data that has to be stored. We also managed to fix major problems with improving the queries to the point where only the necessary shards are hit and there the data is filtered before transfer. This way we can ensure that our implementation scales as well as possible with a single database master.

The final results of our approach are shown in Table 6.3. Most of the TPS were measured with nodes in the Dahu cluster, but for some we had to use Gros because Dahu does not provide enough nodes. In these experiments, we also changed the number of connected clients (exchanges) to see the impact on TPS. We found that for a partitioned DB with a single node, 120 was best, while 80 proved to be the best performing for the sharded setup. [46] This is also the configuration used to measure the numbers below. Please also note that these numbers were measured using a rather short time window and not over a longer duration.

Partitions	Single Node (Partitioned)	Distributed (Sharded)
0	23.4k	
1	23.4k	11.1k
2	23.4k	10.8k
4	23.2k	10.1k
8	22.8k	9.8k
16	22.8k	8.8k
32	21.5k	6.7k [47]
64	17.5k	-

Table 6.3.: The scalability of the database implementation of GNU Taler is shown by the number of TPS in Taler (using batch-withdraw - number of coins signed on each successful request to */batch-withdraw* plus number of successful requests to */deposit*). We can clearly see that partitioning scales quite well, while sharding is fine up to 8 shards. However, the problem with sharding is not the queries as we could not see many slow queries above 50ms in both cases, but the master node that has to deliver data to the shards and fetch data from there to perform further actions.

We can see that the overhead of sharding is quite remarkable when comparing the TPS for just one partition. With sharding, it is less than 50% of that of a single node. Therefore, it might be necessary to investigate further possibilities for better partitioning of the database to achieve a query push-down to the shard nodes.

[46] These numbers may look different now that performance has increased and we have determined that 200 exchanges is optimal when we reach 28k+ TPS with Taler.

[47] Measured using the Gros cluster, for comparison: with 16 shards we achieved 7.2k TPS there.

7. Additional Results

7.1. Transaction-Load Distribution

Many non-uniform real-world distributions of human-created systems follow Zipf's Law. We suspect that as a first approximation this may also be true for banking transactions, or at least that Zipf's Law yields a distribution suitable for realistic benchmarking of payment systems. When applying Zipf's law to transactions, this means that if there are 100 merchants, there will be one merchant who receives say 100 payments over a certain period of time. Then, the second-largest merchant would receive half of what the first receives, i.e. 50, the third a third (33), and so on. [50]

In order to obtain performance results that reflect real-world problems, we have extended the wallet clients with another benchmark that can generate such a distribution. This is particularly interesting because this distribution leads to a spiky rather than a uniform transaction load. The algorithm will randomly select the merchant to pay at each deposit, weighted by Zipf's law. An example distribution generated by this algorithm is shown in Figure 7.1 and 7.2.

As expected, we have not noticed any change in GNU Taler's performance with this payment receiver distribution. [1]

The results of these experiments are shown in Figure 7.3, Figure 7.4, and Figure 7.5. Note, however, that the achieved TPS is now affected by the poorly performing aggregator (of which there was only one at the time those figures were recorded), which pushes our TPS to about 18-19k. [2] For non-uniform selection, we randomly selected a merchant ID from 1000 IDs using the Zipf algorithm described above and a random algorithm (`Math.random()`).

The reason we did not see an impact on performance is mainly because we are not using an account-based system, where a row in the database would need to be updated with the new balance for each transaction. The problem with the account-based approach is that when a merchant receives a payment, their balance and the customer's balance must be updated. While this is not a problem for the customer since it is only one transaction, it can be problematic for the merchant row: if the merchant receives many payments at once, all but one will fail due to database serialization errors. In this case, the transactions would have to be retried but may fail again if other transactions involving the same merchant are again being attempted at the same time. This may result in transactions repeatedly failing which can drastically degrade performance.

However, this is not a problem with the implementation of the Taler exchange, since the deposits are not made to an account, but by updating a coin, which means that there are no conflicts at that time. Later, many deposits are aggregated into a single transfer to the target account. However, here the query is partitioned by time from concurrent incoming

[1] In the majority of the experiments, we even had only one merchant account which received all payments.
[2] This was measured when we reached about 23.5k Taler TPS.

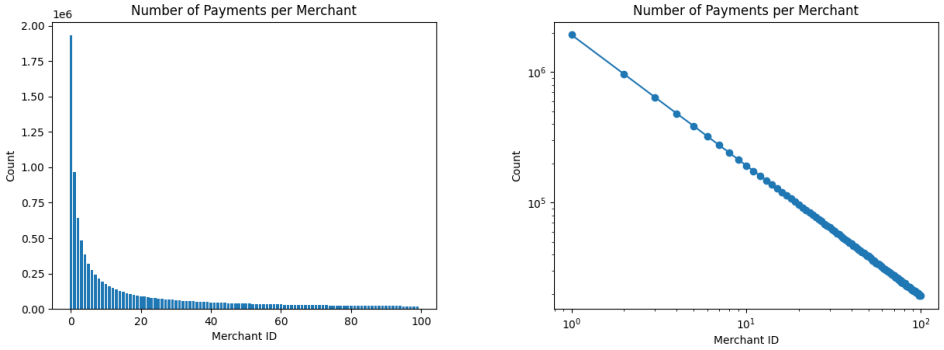


Figure 7.1.: Result of the algorithm used in bench3, which randomly selects a merchant weighted according to Zipf's law. These two figures show the same result of ten million draws from 100 merchants on a linear and log-log scale. The y-axis (*Count*) reflects the number of times a merchant with a given ID (x-axis) was selected to be paid. The result of the integration in the benchmarks is shown in Figure 7.2.

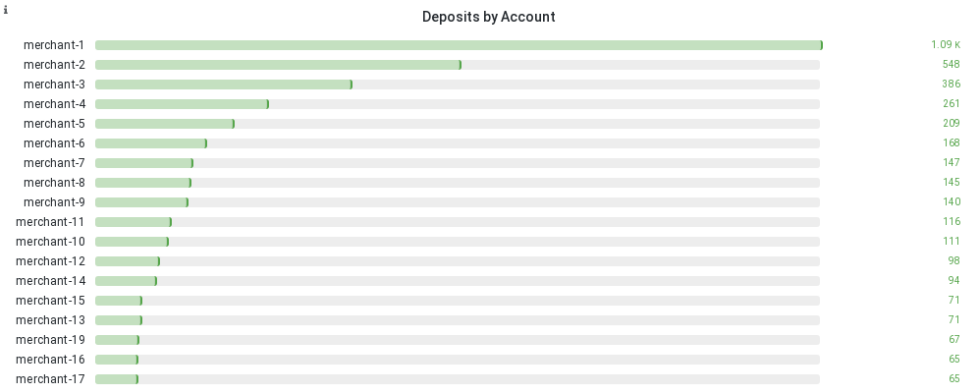


Figure 7.2.: Number of payments (deposits) to each account in a benchmark. We see that the Zipf distribution of the algorithm we created works as expected. The information to display this panel in Grafana is retrieved from the logs of the taler-fakebank, which records all transfers initiated by the taler-exchange-transfer.

transactions and runs per merchant, thereby again ensuring that there is no possibility of a conflict.

There is still one more task to complete which have not yet done, as the aggregator query is not yet optimized and the partitions are not yet set up optimally: We saw that the aggregator was able to achieve about 33k TPS on a single system. We would like to see how it performs with different payment distributions combined with different numbers of shards and processes, as this is more likely to affect the performance of how quickly a merchant receives their payments than the TPS of the system (which we have shown is not affected), as the aggregator is sharded by account, among other things. This means that it will process an account until it is finished aggregating that account. Multiple accounts with different transaction amounts may therefore affect the completion time.

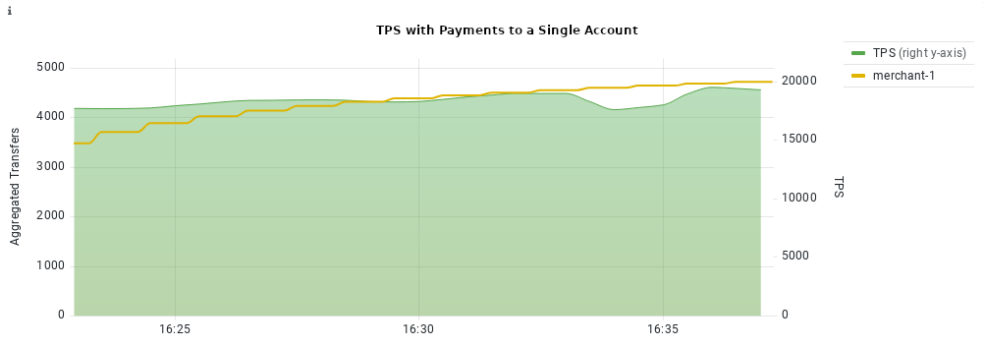


Figure 7.3.: The TPS when all payments are made to a single account (merchant-1). In general, all of our experiments have been like this, making this our starting point for the next set of graphs. The merchant counts show the number of payments to each merchant that are logged by the bank once the transfer process initiates the aggregated transfers created by the Aggregator.

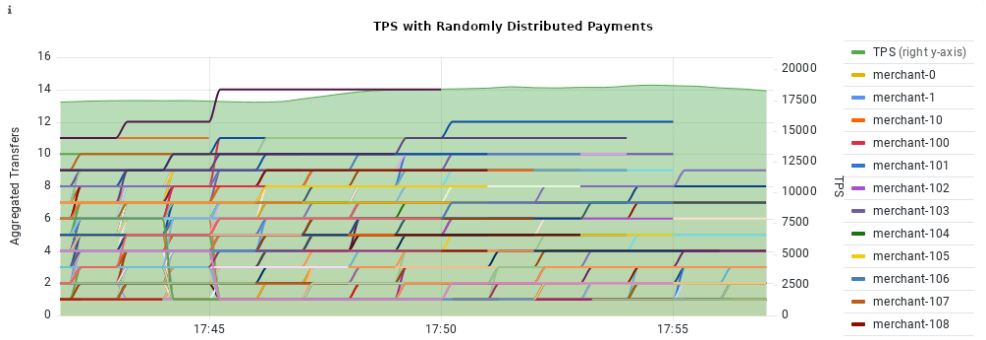


Figure 7.4.: The TPS given a random selection from 1000 accounts to pay to. It can be seen that there is no difference in the TPS (except for a negligible difference due to the use of different nodes in a separate allocation). The same is true for the Zipf distribution in Figure 7.5.

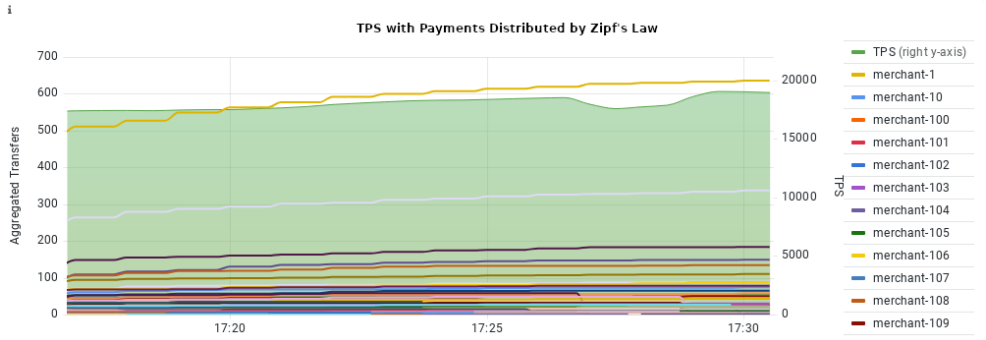


Figure 7.5.: The TPS when using a random Zipf's Law weighted selection from 1000 accounts to pay to. As in Figure 7.4, *this* has no effect on the TPS.

7.2. Auditor Inclusion

To see how well Taler and especially the PostgreSQL database work with replication, we also included the Auditor. To do this, we enabled logical replication [3] to another node in Grid'5000. As we had hoped, the auditor did not cause any performance degradation; there was a slight, almost imperceptible increase in CPU load on the DB node, shown in Figure 7.7, and slightly more network load, visible in Figure 7.8. However, the TPS for Taler remained the same, which is reflected in the referenced figures by the number of requests per second. But we could see that the replication is not that fast to be replicated in real time, which would mean that the auditor does not always see the latest changes (see Figure 7.6). This could possibly be remedied by further research, such as explained in a post on severalnines [4], but this is no longer part of this work. However, under real-world conditions, we would probably not need to handle the TPS peaks for an extended period of time, but only for short periods of time, such as Black Friday. Hence, the replication should be able to keep up except for some peak loads.

Note that this experiment was performed on nodes in the Gros [5] cluster, which are similar in performance to the nodes in Dahu (measured with pgbench), since no free node was available there at that time. We also reached the TPS maximum of about 13.6k in Taler, with a slightly higher CPU utilization (see figures), which still makes this experiment comparable to the others.

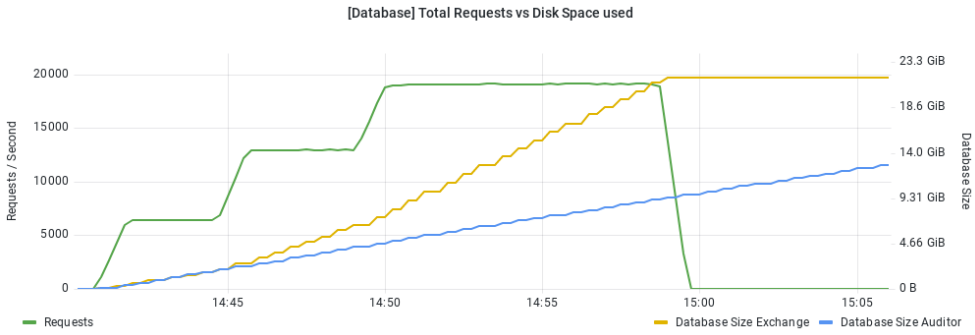


Figure 7.6.: The DB sizes of Exchange and Auditor compared to the number of requests per second. It is clear that Auditor lags behind Exchange in replication when we are at maximum TPS in Taler. This is easy to see because the replication process still continues when the Exchange DB has long since stopped growing. Some configuration values of PostgreSQL might provide a possibility to fix this, but this remains a task for further work.

7.3. Loki Performance

At some point, as the experiments reached higher TPS, we realized that our dashboards were getting too slow. The bottleneck was quickly identified as Loki. It had to parse large chunks of log volume every time an update was requested and compute the metrics we needed. This quickly hit a timeout when we reached about 10k TPS and Loki had to parse gigabytes of

[3] Logical Replication: `https://www.postgresql.org/docs/current/logical-replication.html`
[4] Blog about PostgreSQL replication lag: `https://severalnines.com/database-blog/what-look-if-your-postgresql-replication-lagging`
[5] Gros cluster: `https://www.grid5000.fr/w/Nancy:Hardware#gros`

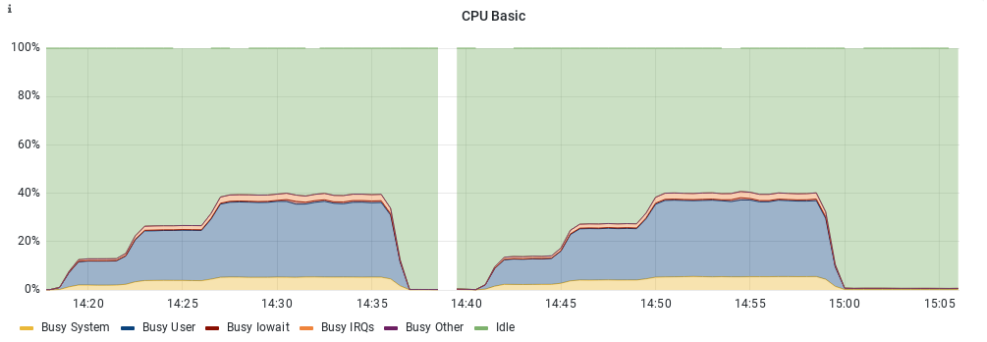


Figure 7.7.: The left side of this figure shows the CPU load of the Exchange database node when there is no replication, while on the right side replication to the Auditor server is running. Almost no change is visible, but the load has increased by a maximum of 1%. This shows us that replication should not be a problem even with higher CPU load. Note that this experiment was placed in the Gros cluster rather than Dahu, which is why we can see a CPU load of 40% rather than 25%.

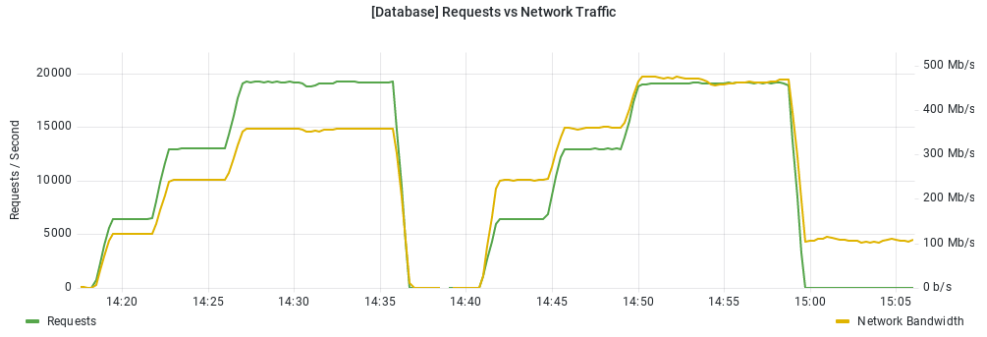


Figure 7.8.: The right part of this figure shows the increase in the total bandwidth of the DB node, which has increased by 120 Mb/s compared to the left side, where no replication takes place. Note that most of this increase is limited to outgoing traffic, as replication data is sent to the auditor but not much additional data is actually received. This figure also clearly shows that the number of requests per second is the same in both cases (with the same number of wallets running), which also reflects the TPS in Taler.

proxy logs to calculate TPS counts and request time statistics. While researching, we found out that this problem can be solved quite easily. Promtail can be configured to calculate custom statistics by calculating them right when the log lines are received using *pipeline stages* (Appendix Listing A.3). These metrics can then in turn be retrieved by Prometheus, while the actual logs continue to be sent on to Loki.

7.4. PostgreSQL Query Analysis

When we had multiple slow queries to analyze and improve, it quickly became tedious to search through PostgreSQL's entire log file to find the right query and understand its execution plan created by `auto_explain`. To simplify this process, we wrote a Python script that automatically scans the logs for log statements that indicate queries and their parameters. Prior to this script, we had to dump the entire database, find the parameters needed for a query by searching the database, and then manually run the queries to customize them. The

script was then written to automate this process and analyze the performance of the various prepared statements. Since many of them require parameters to be manually searched, we could not simply read the statements from the C source code. Therefore, we decided to write verbose PostgreSQL logs containing all executed instructions with parameters. Thus, all that is needed for the script is a pre-filled database and logs that contain the necessary output. To achieve this, we pre-filled the database with some data by running an experiment. Then we disabled unnecessary logging and enabled statement logging with `log_statement=all`. Then we ran only one wallet to get a sample of queries. Once the database and logs were saved locally, the script could be run. To still manually check some queries, it is possible to specify the query instead of a log file. Or one could simply change a statement of interest in the log directly.

The script summarizes various statistics extracted from `EXPLAIN ANALYZE`, such as how many partitions were hit by a query, how long it took, or which non-indexed scans were run. This makes it easier to compare two different queries with the same output result. The analysis of the queries for sharding was particularly interesting, since we did not have *the* perfect sharding key and had to optimize the queries to address as few shards as possible to minimize network load. An example output can be seen in Listing 7.1.

For more detailed analysis, the script generates a file for each query executed, containing the full plan in a more readable form as `EXPLAIN ANALYZE`.

However, this script is not as effective for functions because the logs contain only the function call statement.[6] The execution of explain analyze for such an instruction therefore does not contain any usable information other than the query execution time.

```
name                 t (ms)      act-rws   sub-q-rws   prts     n-idx
4_get_reserves_out   0.176       0         5           3 (2)    1
5_recoup_by_reserve  1056.125    0         4           2 (8)    3
6_close_by_reserve   0.056       0         0           1 (2)    1
7_call_withdraw      7.626       1         0           0 (0)    0
```

Listing 7.1: Sample output from explain-analyze.py listing the number and name of the query, the time spent, the actual rows returned, the total number of rows returned by subqueries, the number of partitions hit, and the total number of non indexed scans. The numbers in parentheses indicate the number of partitions that are hit when the entire query is run, e.g. when some filters have a match (however, this also include the number of partitions filtered by our materialized CTE approach as described in Section 6.5).

[6] Unfortunately, there is no similar option here as `log_nested_statement` in explain analyze. With `log_statements`, PostgreSQL only logs the explicitly executed statements.

8. Future Work

We will now summarize key points that might enable a single exchange deployment to reach our original goal of 100,000 TPS, based on the various findings in this thesis and avenues we had to leave unexplored.

8.1. Exchange

OpenSSL RSA

As we have seen, the current implementation of RSA cryptography is too slow for the high-performance application of Taler, especially already for the load generation logic in the wallets. While we were able to reduce the key sizes to 1024 bits in our experiments, one would want at least 2048 bits in production. To limit the number of exchange nodes required to handle the 2048 bit load and to be able to generate load with 2048 bits with a reasonable amount of resources, we suggest changing the implementation and to either use the much faster OpenSSL implementation instead of Libgcrypt, or to modify Libgcrypt taking inspiration from OpenSSL.

Batch-Withdraw Parallelization

The current implementation of batch withdrawal is sequential, meaning that the RSA signatures are performed sequentially, even if many CPU cores are available. Thus, latency of batch withdrawals could likely be significantly reduced by parallelizing the process to sign coins.

Key Sharing

Since we need multiple exchange nodes to achieve the desired performance, we need to distribute the key material among the nodes. One exchange node must be responsible to generate keys, and the others must then receive this material to accomplish their task. Since the experiments lasted only a short time, we could simply exchange the keys there via NFS and instruct all but one exchange to wait until the keys were generated before starting.

However, this is not a production-quality design, as the keys rotate when the exchanges are used for an extended period of time. We will need to implement rotation in a sane way where one exchange is explicitly made responsible for this task, while the others will be instructed to securely obtain keys from the master exchange. Ideally, there would also be a fallback mechanism that handles the case where the master exchange is not operational at the time the keys need to be rotated.

Metrics

Currently, the metrics are reported by each individual Exchange process, which is why each httpd process has to listen on its own port. Such a setup is not suitable for production use, where metrics like TPS are certainly of interest. Since these are now measured by the exchange, it might be a good idea to implement metrics counters in a shared store where all exchange processes can increment the same counter. This would allow a single port for all processes and also prevent counter resets due to exchange restarts.

8.2. Exchange Database

Query Performance

Although we were able to improve the query performance significantly, there is always room for improvement. The current implementation is good as far as slow queries are concerned, as there were only a few over 50ms in experiments.

Nevertheless, one should consider implementing queries using PostgreSQL's C language functions [1] to further improve performance. This may reduce the CPU load on the database, and nail down access partitions on shards, and possibly even allow us to push down more of the load to the shards.

However, such a C implementation should probably be reserved for rare situations where the SQL implementation is too slow, as it would significantly increase the required effort to properly deploy an exchange.

Sharding

We have achieved that sharding now works with constant network load and acceptable performance. However, we had hoped to achieve better performance than with a single node since we could offload operations to shards. While we were able to distribute IO, we were not able to improve general performance: since the master node is still responsible for joins it has to fetch data (sometimes multiple times) from the shards and merge them locally. Additionally, we lost TPS due to the newly added latency to the shards. By making further adjustments to the schema and queries, one might be able to move the queries entirely to the shards, which could lighten the load on the master, and thus hopefully improve performance.

Batch Deposit

We have introduced batch withdrawal and shown that it is beneficial for TPS and I/O load. Changes like this can also be applied to other endpoints, especially deposit, which currently deposits each coin sequentially. Like with batch withdraw, such a change may also slightly reduce bandwidth as the individual requests do contain some redundant information.

[1] C language functions: `https://www.postgresql.org/docs/current/xfunc-c.html`

Aggregator

As described earlier, we did in most cases not use the aggregator because the queries were not yet aligned and optimized for partitions. Currently, there is a large query (called "aggregate") that does everything in one big SQL statement, but therefore can be rather slow, scans multiple partitions and may generate serialization errors without proper partitions.

This query needs to be placed in a function like we did with `recoup_by_reserve` (or directly in a C language function) and rewritten to prevent it from unnecessarily retrieving tables from all partitions before joins. Further, we need to change the partition method on materialized indexes for the deposit helper tables (`deposits_by_ready` and `deposits_for_matching`). The partition method is currently `HASH` for simplicity, and should be changed to `RANGE`. These tables are partitioned by deadlines. The deadlines are timestamps signal to the aggregator when a transfer to a merchant can be initiated, for example when the refund period has expired. This means that the aggregator will only handle rows of a table where the deadline is in the past. If we now create the partitions by range on this deadline, we ensure that the aggregator only hits rows that are no longer accessed by any other operation from the exchange. Furthermore, after the aggregator is done, the respective partition can be deleted, because it will not be used anymore.

However, if we now switch to range, partitions must be created continuously for future transactions, since we cannot generate partitions for all possible time intervals into the future. The number of partitions, the duration of the deadline respectively, is also configurable, as this determines how long a merchant has to wait for his payment after the wire deadline. A small interval would result in a potentially large number of partitions. The creation and deletion of these partitions must therefore run automatically in the background. This could be solved by an SQL function that is triggered when entries would be placed into a special "future" partition (beyond the normal range of partitions). It would then have to create the required partiton(s) and migrate all entries from the "future" partition into the newly created partitions.

8.3. Auditor

While we have included the auditor in our experiments and shown that it does not have a large impact on Taler's performance, we have found that replication lags behind the main database in high load situations. This is probably not a problem for most cases, since a payment system is not expected to be constantly at peak load, improving replication speed might still be interesting, especially in combination with shards. By replicating individual shards, one might be able to improve the replication speed, and thus the timeliness of the auditor.

Another issue is that the current implementation of the auditor itself is limited in its parallelism. Thus, even if the database replication were faster, the CPU utilization of the auditor itself would likely cause the auditor to lag behind the exchange. Here, parallel verification of signatures may go a long way towards improving auditor performance.

Finally, we did not investigate to what extent the auditor's SQL queries themselves might be suboptimal. In particular, the auditor makes use of serial numbers of records in tables to organize its work. This could be good as it may enable the auditor to efficiently download work from the various shards in parallel, or bad if a query unnecessarily hits many shards.

Thus, the SQL queries of the auditor should also be investigated.

8.4. Additional Transactions

At present, the experiments have not yet been carried out with all types of Taler transactions. Specifically, refunds and recoup operations have not yet been taken into account. While we do not expect a fundamental change, these transactions may again require specific optimizations to be performed. Naturally, it would be good to know how (un)common these operations would be in practice to ensure that the benchmark is realistic.

Another key issue to be considered is transactions that arise when handling (client) failures. For example, when a client attempts to double-spend a coin, the exchange generates an error message that proves the double-spending. Some of the queries to generate error messages involve returning complex data structures, and hence the queries can sometimes be rather complicated (see our discussion on recoup-by-reserve, which is an example for such a query). While errors should be rare in normal production, an adversary trying to perform a denial-of-service attack may deliberately issue requests that result in expensive queries. Thus, *all* queries of the system should eventually be checked for worst-case performance, and not only the “happy path” queries we considered in this work.

8.5. Wallet Clients

We have found in many cases that the wallets do not work well with longer signature periods and more denominations, as they cannot check the retrieved data in a constant time. This should not be a problem for real deployments, as a customer is likely to have only one wallet running at a time. Still, making them more powerful especially for a large number of denominations would be a place for improvement. Specifically, it might be helpful to only retrieve the denominations that are urgently required, rather than all of them for the next two years.

We also measured the time it takes a wallet to perform the withdraw and deposit operations. To do this, we took an idle node with no task and ran the script bench1 with different amounts for withdrawal and deposit while having 23k TPS in Taler. In doing so, we measured the following results:

KUDOS	Withdraw	Batch-Withdraw	Deposit
8	1.8 s	1.8 s	2.6 s
10	2 s	2 s	2.6 s
20	2 s	2 s	2.6 s
50	2.2 s	2 s	2.7 s
100	2.8 s	2 s	2.8 s
200	3.8 s	2 s	3.5 s

Table 8.1.: The time taken by the wallets to execute each operation with the specified KUDOS amount. Note that for bench1 this includes multiple HTTP requests for all operations as well as refreshing in the deposit phase. All of these numbers were measured when we had an average of 1.63 ms for reserves, 82 ms for batch withdrawals (which were used to generate the main load), and 65 ms for deposits (plus 69 ms for coins-melt and 76 ms for refresh-reveal) as Nginx response times.

Given the request time reported by Nginx and the round-trip times of about 12 ms to the proxy and the bank, we can see that much more time is needed to perform the actions on the clients, which probably indicates that they can be further improved here as well. However, we also found that the wallets do not yet cache the responses to `wire`, `terms`, and `keys`. These queries are also made in every withdrawal. Eliminating these requests could at least reduce withdrawal times.

8.6. Merchant

In our experiments, we did include various merchant accounts to which we made payments, but the payments were not yet made to the individual merchants. This feature requires the wallets in the benchmark to create orders and pay those orders directly through the merchant. While the merchant is already included in the setup, and ready to create orders, the Wallets are not capable of creating these orders at the moment. However, we anticipate that the merchants will become a bottleneck fairly quickly, as there are likely to be a lot of unoptimized queries that need to be addressed.

9. Conclusion

Performance tuning is always a game of whack-a-mole, where removing one obstacle immediately leads to the next, with the danger of looking in the wrong place. Still, we identified and fixed various bottlenecks and made several improvements in the GNU Taler software, such as introducing batch withdrawals. But there were also a lot of problems with a rather unexpected component: the load-generating clients. The wallets kept introducing new problems that affected our experiments more than expected, leaving them in need of improvement almost as much as the main logic of the exchange. We have now also found that our monitoring system is reaching its limits with excessive logging, certainly a problem that not everyone struggles with. That said, the database is a key issue for Taler scalability, though we expected that too soon.

We have shown that GNU Taler can already process nearly 30,000 transactions per second with a single exchange, using a setup that is as close to reality as possible. This includes the distributed setup, non-uniform load generation, and state-of-the-art encryption methods such as TLS. However, the experiments themselves were not yet very realistic, lasting only about an hour or two and therefore showing more of what Taler can handle in peak load situations. In some places, we were even forced to relax the requirements a bit to further improve performance even with limited resources, for example, the RSA key size. But in the end, we identified many aspects that will help to improve performance even more in future refinements.

Nevertheless, we already know that Taler scales well in a distributed setup and that a real-world deployment would require only a few exchanges per continent. In Europe, this could mean one exchange for each country in a well-connected city (e.g., Paris, Frankfurt, Zurich), all audited by a trusted authority such as the European Central Bank. Such a facility would certainly reach more than 100'000 TPS, but at the cost of some privacy, since payments could then be linked to countries.

Declaration of Authorship

I hereby declare that I have written this thesis independently and have not used any sources or aids other than those acknowledged.

All statements taken from other writings, either literally or in essence, have been marked as such.

I hereby agree that the present work may be reviewed in electronic form using appropriate software.

June 15, 2022

M. Boss

Bibliography

[1] Nir Kshetri. The economics of central bank digital currency [computing's economics]. *Computer*, 54(6):53–58, 2021. Available at https://ieeexplore.ieee.org/document/9447413 [07.12.2021].

[2] Grid'5000. Grid'5000 introduction, 2022. Available at https://www.grid5000.fr [13.06.2022].

[3] Florian Dold. *The GNU Taler System: Practical and Provably Secure Electronic Payments. PhD Thesis*. PhD thesis, University of Rennes 1, 2019. Available at https://taler.net/papers/thesis-dold-phd-2019.pdf [04.06.2022].

[4] GNU Taler. Gnu taler: Features, 2022. Available at https://taler.net/en/features.html [04.06.2022].

[5] Jim Cunha, Robert Bench, James Lovejoy, Cory Fields, Madars Virza, Tyler Frederick, David Urness, Kevin Karwaski, Anders Brownworth, Neha Narula. Project hamilton phase 1 a high performance payment processing system designed for central bank digital currencies. Technical report, Federal Reserve Bank of Boston and Massachusetts Institute of Technology Digital Currency Initiative, Feb 2022. Available at https://www.bostonfed.org/-/media/Documents/Project-Hamilton/Project-Hamilton-Phase-1-Whitepaper.pdf [05.05.2022].

[6] James Lovejoy, Cory Fields, Madars Virza, Tyler Frederick, David Urness, Kevin Karwaski, Anders Brownworth, Neha Narula. A high performance payment processing system designed for central bank digital currencies. Technical report, Federal Reserve Bank of Boston and Massachusetts Institute of Technology Digital Currency Initiative, Feb 2022. Available at https://static1.squarespace.com/static/59aae5e9a803bb10bedeb03e/t/61fc25f91a0df9037488eb7d/1643914745989/Hamilton.Whitepaper-2022-02-02-FINAL2.pdf [15.04.2022].

[7] ECB. Eurosystem report on the public consultation on a digital euro. Technical report, European Central Bank, Apr 2021. Available at https://www.ecb.europa.eu/pub/pdf/other/Eurosystem_report_on_the_public_consultation_on_a_digital_euro~539fa8cd8d.en.pdf [15.04.2022].

[8] People's Bank of China. Progress of research & development of e-cny in china. Technical report, People's Bank of China, Jul 2021. Available at http://www.pbc.gov.cn/en/3688110/3688172/4157443/4293696/2021071614584691871.pdf [05.05.2022].

[9] Ashutosh Pandey. China heats up digital currency race with the e-cny debut at olympics, Feb 2022. Available at https://www.dw.com/en/china-heats-up-digital-currency-race-with-e-cny-debut-at-olympics/a-60701261 [05.05.2022].

[10] Sveriges Riskbank. e-krona, 2022. Available at https://www.riksbank.se/en-gb/payments--cash/e-krona/ [05.05.2022].

[11] Sveriges Riskbank. e-krona pilot phase 2. Technical report, Sveriges Riskbank, Apr 2022. Available at https://www.riksbank.se/globalassets/media/rapporter/e-krona/2022/e-krona-pilot-phase-2.pdf [05.05.2022].

[12] Sveriges Riskbank. e-krona pilot phase 1. Technical report, Sveriges Riskbank, Apr 2021. Available at https://www.riksbank.se/globalassets/media/rapporter/e-krona/2021/e-krona-pilot-phase-1.pdf [05.05.2022].

[13] Stig Johansson Hanna Armelius, Gabriela Guibourg and Johan Schmalholz. E-krona design models: pros, cons and trade-offs, 2020. Available at https://www.riksbank.se/globalassets/media/rapporter/pov/artiklar/engelska/2020/200618/2020_2-e-krona-design-models-pros-cons-and-trade-offs.pdf [08.05.2022].

[14] Ananya Kunar. A report card on china's central bank digital currency: the e-cny, Jan 2022. Available at https://www.atlanticcouncil.org/blogs/econographics/a-report-card-on-chinas-central-bank-digital-currency-the-e-cny/ [05.05.2022].

[15] Daniela Mechkaroska, Vesna Dimitrova, and Aleksandra Popovska-Mitrovikj. Analysis of the possibilities for improvement of blockchain technology. , pages 1–4, 11 2018. Available at https://www.researchgate.net/publication/330585021_Analysis_of_the_Possibilities_for_Improvement_of_BlockChain_Technology [16.05.2022].

[16] Andre Rocha. Postgresql load tuning on red hat enterprise linux, Apr 2022. Available at https://www.redhat.com/en/blog/postgresql-load-tuning-red-hat-enterprise-linux [24.04.2022].

[17] Postgres. Postgresql wiki, Sep 2020. Avaliable at https://wiki.postgresql.org/wiki/Performance_Optimization [24.04.2022].

[18] leOpard. Pgtune, Apr 2022. Available at https://pgtune.leopard.in.ua [24.04.2022].

[19] Kumar Vibhor. Pgbench: Performance benchmark of postgresql 12 and edb advanced server 12, May 2020. Available at https://www.enterprisedb.com/blog/pgbench-performance-benchmark-postgresql-12-and-edb-advanced-server-12 [24.04.2022].

[20] Swapnil Suryawanshi. Comprehensive guide on how to tune database parameters and configuration in postgresql, Dec 2019. Available at https://www.enterprisedb.com/postgres-tutorials/comprehensive-guide-how-tune-database-parameters-and-configuration-postgresql [24.04.2022].

[21] Cristian Ruiz, Salem Harrache, Michael Mercier, and Olivier Richard. Reconstructable Software Appliances with Kameleon. *Operating Systems Review*, 49(1):80–89, 2015. PDF: https://hal.archives-ouvertes.fr/hal-01334135/file/Reconstructable_software_appliances_with_kameleon.pdf.

[22] imec. *jFed Experiment Specification*. Available at https://jfed.ilabt.imec.be/espec/ [08.12.2021].

[23] brandur. How to manage connections efficiently in postgres, or any database, Oct 2018. Available at https://brandur.org/postgres-connections#concurrency-limits [07.12.2021].

[24] jjanes. How can i find the source of postgresql per-connection memory leaks?, Nov 2018. Available at https://dba.stackexchange.com/a/222815 [07.12.2021].

[25] Italo Santos. Postgresql out of memory, Feb 2020. Available at https://italux.medium.com/postgresql-out-of-memory-3fc1105446d [07.12.2021].

[26] David Conlin. Row count estimates in postgres, Dec 2018. Available at https://www.pgmustard.com/blog/2018/12/14/row-count-estimates-in-postgres [27.04.2022].

[27] PostgreSQL. *Routine Vacuuming*, 2022. Available at https://www.postgresql.org/docs/current/routine-vacuuming.html#VACUUM-FOR-SPACE-RECOVERY [06.03.2022].

[28] Ankit Shukla. How to solve a bloated postgres database, Apr 2020. Available at https://www.enterprisedb.com/blog/postgres-pulse-insights-its-still-slow-solving-bloated-postgres-database [05.05.2022].

[29] PostgreSQL. *Routine Vacuuming*. Available at https://www.postgresql.org/docs/13/routine-vacuuming.html#VACUUM-FOR-STATISTICS [27.04.2022].

[30] Laurenz Aube. Hot updates in postgresql for better performance, Sep 2020. Available at https://www.cybertec-postgresql.com/en/hot-updates-in-postgresql-for-better-performance/ [27.04.2022].

[31] Jobin Augustine. Postgresql synchronous_commit options and synchronous standby replication, 2022. Available at https://www.percona.com/blog/2020/08/21/postgresql-synchronous_commit-options-and-synchronous-standby-replication/ [07.03.2022].

[32] PostgreSQL. Write ahead log, 2022. Available at https://www.postgresql.org/docs/13/runtime-config-wal.html#GUC-SYNCHRONOUS-COMMIT [04.05.2022].

[33] Thomas Munro. Cheat sheet: Configuring streaming postgres synchronous replication, May 2017. Available at https://www.enterprisedb.com/blog/cheat-sheet-configuring-streaming-postgres-synchronous-replication [04.05.2022].

[34] Multiple. Postgresql mailing list, 2022. Available at https://www.postgresql.org/message-id/VI1PR04MB31338E811E0896D00D85B69B97E19@VI1PR04MB3133.eurprd04.prod.outlook.com [03.05.2022].

[35] PostgreSQL. *Transaction Isolation*, 2022. Available at https://www.postgresql.org/docs/13/transaction-iso.html [09.05.2022].

[36] PostgreSQL. *pgbench*, 2022. Available at https://www.postgresql.org/docs/13/pgbench.html#id-1.9.4.10.9.2 [27.04.2022].

[37] PostgreSQL. Number of database connections. Available at https://wiki.postgresql.org/wiki/Number_Of_Database_Connections#How_to_Find_the_Optimal_Database_Connection_Pool_Size [04.05.2022].

[38] Dave Page. Tuning max_wal_size in postgresql, Mar 2022. Available at https://www.enterprisedb.com/blog/tuning-maxwalsize-postgresql [04.05.2022].

[39] Hans-Jürgen Schönig. Postgresql: What is a checkpoint?, Feb 2021. Available at https://www.cybertec-postgresql.com/en/postgresql-what-is-a-checkpoint/ [04.05.2022].

[40] PostgreSQL. Write-ahead logging (wal), 2022. Available at https://www.postgresql.org/docs/13/wal-intro.html [04.05.2022].

[41] The Linux Kernel. Lock statistics, 2022. Available at https://www.kernel.org/doc/html/latest/locking/lockstat.html#configuration [04.05.2022].

[42] PostgreSQL. The statistics collector, 2022. Available at https://www.postgresql.org/docs/current/monitoring-stats.html [04.05.2022].

[43] PostgreSQL. pg_locks, 2022. Available at https://www.postgresql.org/docs/current/view-pg-locks.html [04.05.2022].

[44] PostgreSQL. Table partitioning, 2022. Available at https://www.postgresql.org/docs/13/ddl-partitioning.html [01.05.2022].

[45] Rajkumar Raghuwanshi. How to use table partitioning to scale postgresql, Mar 2020. Available at https://www.enterprisedb.com/postgres-tutorials/how-use-table-partitioning-scale-postgresql [28.04.2022].

[46] PostgreSQL. *WITH Queries (Common Table Expressions)*, 2022. Available at https://www.postgresql.org/docs/current/queries-with.html [03.05.2022].

[47] PostgreSQL. Controlling the planner with explicit join clauses, 2022. Available at https://www.postgresql.org/docs/13/explicit-joins.html [03.05.2022].

[48] PostgreSQL. *Remote Query Optimization*, 2022. Available at https://www.postgresql.org/docs/13/postgres-fdw.html#id-1.11.7.42.13 [13.05.2022].

[49] PostgreSQL. *The Statistics Collector*, 2022. Available at https://www.postgresql.org/docs/current/monitoring-stats.html#WAIT-EVENT-CLIENT-TABLE [11.05.2022].

[50] Wikipedia Community. Zipf's law, 2022. Available at https://en.wikipedia.org/wiki/Zipf%27s_law [31.05.2022].

List of Figures

List of Tables

Listings

Glossary

API Application Programmable Interface, type of software interface .

Bash Bourne again shell, popular on linux systems .

CLI Command Line Interface, text based frontend for interactice access to applications.

Datasource Term used by Grafana for storage backends providing time series data (see `https://grafana.com/docs/grafana-cloud/fundamentals/intro-to-datasources/`) .

DDoS Distributed Denial of Service, similar to DoS, but the attack originates from many different sources.

Dead Tuple Tuples (rows) in PostgreSQL which were deleted and are no longer referenced .

DNS Domain Name Service, decentralized naming system used to identify computers reachable through the internet .

DoS Denial of Service, cyber-attack to make a system unavailable for its users.

ESpec Experiment Specification, yaml file which describes the experiment steps in jFed. (see `https://jfed.ilabt.imec.be/espec/`) .

HOT Heap Only Tuple, term used by PostgreSQL (see: `https://github.com/postgres/postgres/blob/master/src/backend/access/heap/README.HOT`) .

Hyper-Threading Implementation by Intel to improve parallelization of computations by adding a logical CPU core for each physical one .

jFed A java based framework for testbed federation. Used to run experiments in various tesbeds via a graphical user interface. (see `https://jfed.ilabt.imec.be/`) .

Kameleon A tool to generate custom operating system images with selected tools and libraries installed. (see `http://kameleon.imag.fr/index.html`) .

Materialized Index Additional 'translation tables' we created for better matching of partitions in our queries .

NFS Network File System, distributed file system protocol .

NGI Next Generation Internet (`https://www.ngi.eu/`).

NVME Non Volatile Memory Express, relatively new storage access protocol optimized for high througput .

Partition Subset of a table which is split up based on a key. Often used to lower number of serialization errors, or to offload the partitions onto seperate disks to reduce IO load. (see `https://www.postgresql.org/docs/13/ddl-partitioning.html`) .

ping A commandline utility to measure round trip times between nodes by sending ICMP echo reqeusts to the corresponding network hosts .

RSpec Resource Specification, xml files used to describe resources in jFed. (see `https://doc.fed4fire.eu/testbed_owner/rspec.html`) .

RTT Round Trip Time, the duration from when a client sends a request to a server until it receives a response, in our case measured with `ping` .

Shard Same principle as partition, with the difference that a shard, also known as foreign table is, located on a different database node. (see `https://www.postgresql.org/docs/13/PostgreSQL-fdw.html`) .

systemd System and service management component of Linux operating systems .

UI User Interface, space of interaction between humans and machines.

WAL Write Ahead Log, ensures data integrity of a database .

A. Appendix

A.1. Dashboards

The following few sections will explain the custom Grafana dashboard panels used in the experiments. Most of them are documented in the dashboards themselves, they do not show this in the plots. This information can thus be found here.

A.1.1. Transactions

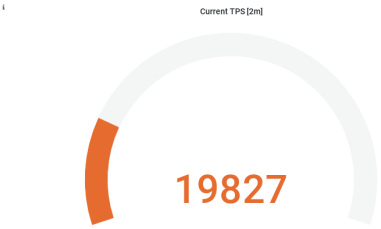


Figure A.1.: The current TPS measured in a fixed interval of 2 minutes. Without batch-withdraw, it is calculated from all successful requests (HTTP status 200) to */withdraw* and */deposit* logged by the nginx proxy. If batch-withdraw is used, withdrawals per second are retrieved from the exchange metric `taler_exchange_batch_withdraw_num_coins`.

Figure A.2.: In case of batch-withdraw, this is the number of coins withdrawn per second, measured by the exchange processes. Else it is the number of successful requests to */withdraw* which were logged by the proxy. All calculated in a fixed interval of 5 minutes.

Figure A.3.: Number of successful requests to */deposit* which were logged by the proxy. Calculated in a fixed interval of 5 minutes.

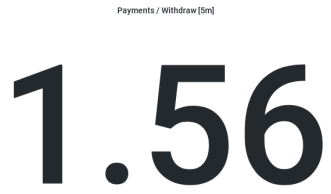


Figure A.4.: The relation deposits to withdrawals per second in a fixed 5-minute interval.

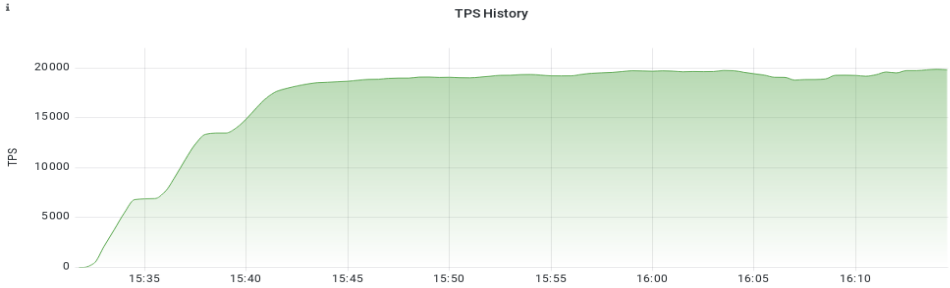


Figure A.5.: History of the TPS in Taler. With the same calculation criteria as in Figure A.1. The numbers are calculated over a fixed interval of 2 minutes.

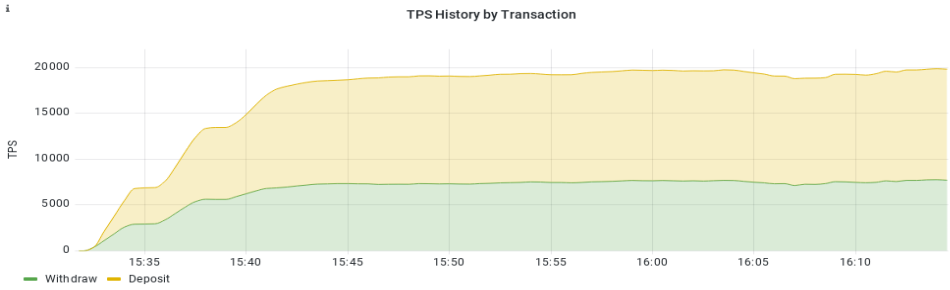


Figure A.6.: History of the TPS in Taler separated by transaction, this is basically the same as the one above, only that we can see which transaction made how much TPS.

All the dashboards shown above were created when we had not yet identified rsyslog as a problem. When we added new metrics to Exchange, we also added new dashboards using those metrics instead of the ones from Promtail. Nonetheless, the dashboards that are displayed are still there in a collapsed row in Grafana. Indeed, they are still needed to inspect snapshots, for example.

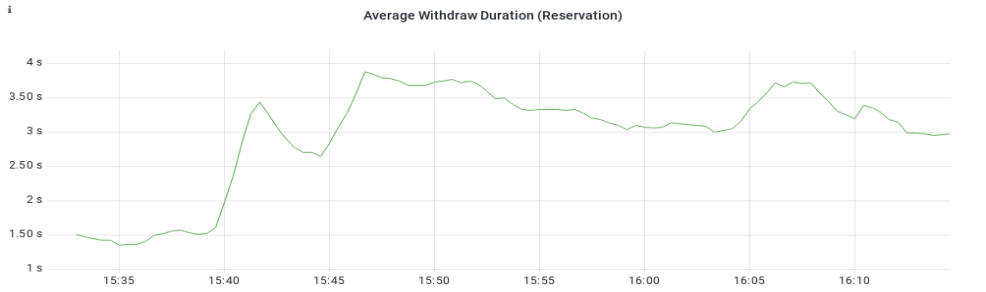


Figure A.7.: The average duration which a wallet needs to complete a whole withdrawal (all coins). Logged by every hundredth wallet on each node by the bench1 and bench3 implementations.

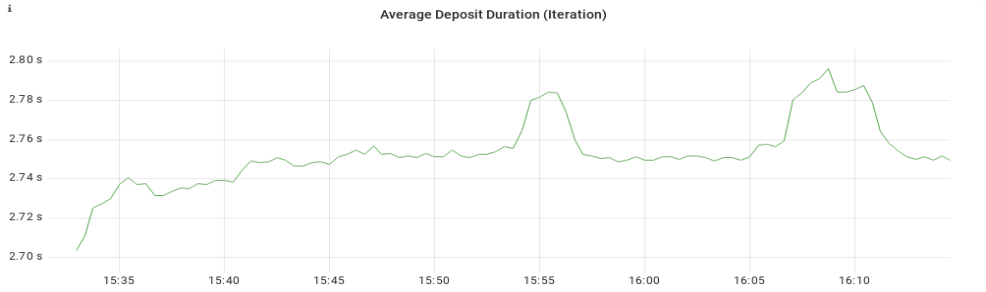


Figure A.8.: The average duration which a wallet needs to complete a whole deposit (all coins per deposit) in an iteration. This is also logged by every hundredth wallet on each wallet node.

Figure A.9.: Number of running wallets over time. Important to visualize the relation of wallets to TPS.

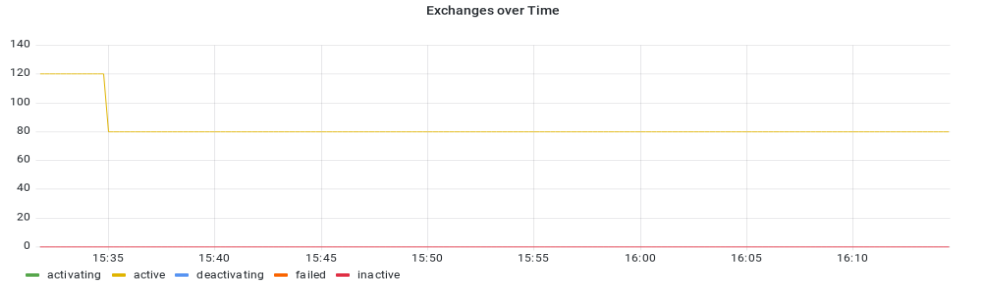


Figure A.10.: Number of running exchanges over time. Important to visualize the relation of exchanges to TPS.

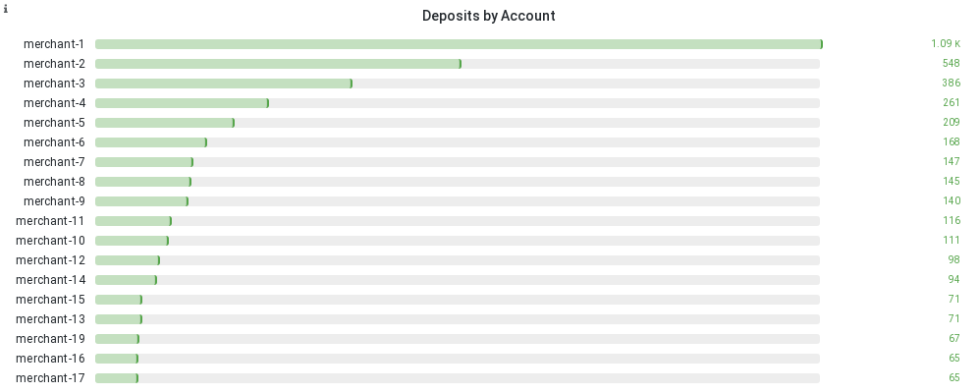


Figure A.11.: Number of payments made to the account in question. This only works when bench3 is used and aggregator plus transfer are running, as the data is retrieved from the bank logs, which logs the transfers when they were aggregated by the aggregator and finally transferred by the transfer process.

There are also further panels in this dashboard, showing the number of processes, number of nodes, withdraw technique (batch/sequential), number of partitions or the state of the processes. They do not need further explanation.

A.1.2. Exchange

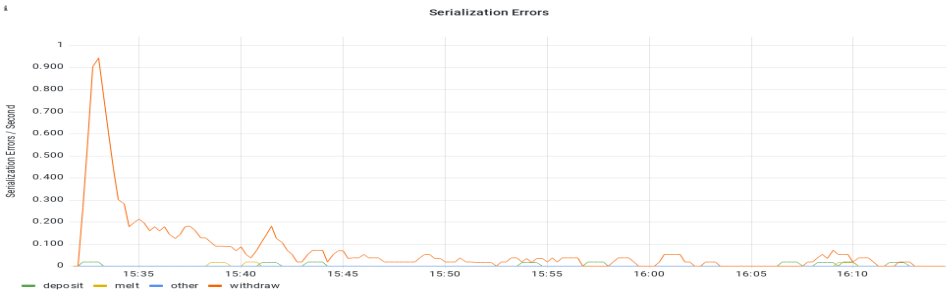


Figure A.12.: The rate of serialization errors occurring by type of request. This information is taken from the exchange's metrics endpoint.

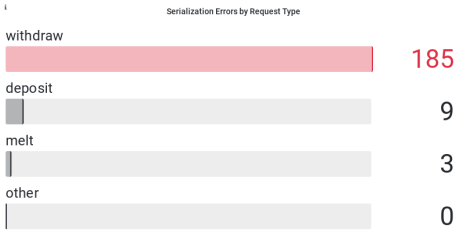


Figure A.13.: Total number of serialization errors by type of request. This information is also taken from the exchange's metrics endpoint. Which means it will sometimes be reset when the process restarts.

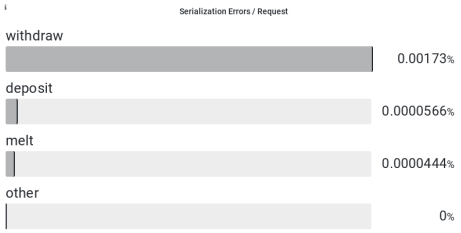


Figure A.14.: Serialization errors per request in percent. Calculated based on the information of the exchange's metrics endpoint and thus also reset when the processes restart.

Figure A.15.: Number of slow queries per second, calculated from the database's logs, which are preprocessed by Promtail. It is taken from `log_min_duration_statement` based on the filter `duration:` so make sure the setting is enabled.

Slow Query Occurences [5m]

Query	Count
SELECT out_exchange_timestamp AS exchange_timestamp,out_balance_ok AS balance_ok,out_conflict...	252
INSERT INTO refresh_revealed_coins (melt_serial_id ,freshcoin_index ,link_sig ,denominations_serial ,c...	231
WITH dd (denominations_serial ,coin_val ,coin_frac) AS (SELECT denominations_serial ,coin_val ,coin_...	35
SELECT out_denom_unknown AS denom_unknown,out_conflict AS conflict,out_nonce_reuse AS nonce...	10
SELECT out_balance_ok AS balance_ok,out_zombie_bad AS zombie_required,out_noreveal_index AS n...	1
SELECT end_row FROM work_shards WHERE job_name=$1 ORDER BY end_row DESC LIMIT 1;	1
SELECT pg_database.datname,tmp.mode,COALESCE(count,0) as count	1

Figure A.16.: Slow queries in detail. Shows how often a query has taken longer than the configured duration in `log_min_duration_statement` in the last 5minutes, which is set to 50ms in this case. Only works if the setting is enabled.

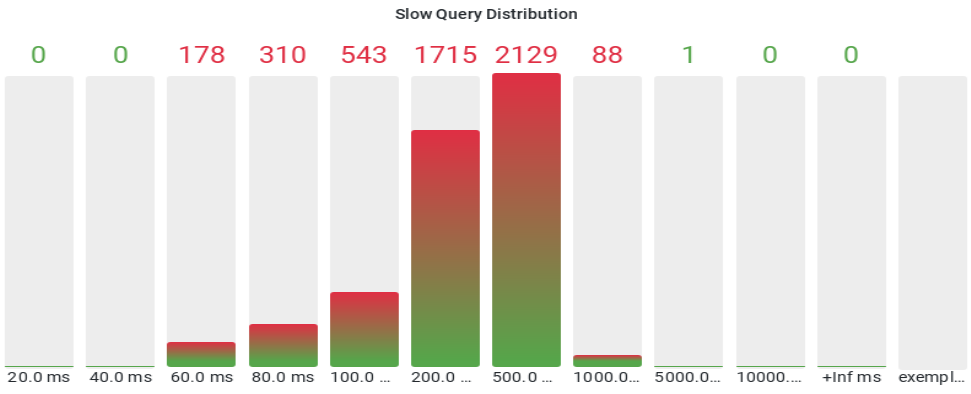


Figure A.17.: Distribution of slow query durations in the displayed time period. The histogram is calculated by Promtail based on the `log_min_duration_statement` logs of the database. There are also quantile and max/min average (etc.) panels which are not shown here.

Figure A.18.: Number of signatures created per second by the exchange processes, this is retrieved from the custom metric endpoint of the exchange processes.

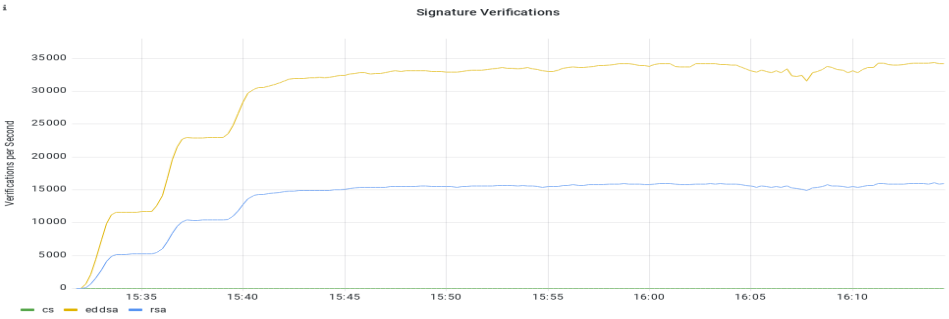


Figure A.19.: Number of signature verified per second by the exchange processes, like above this is also retrieved from the custom metric endpoint.

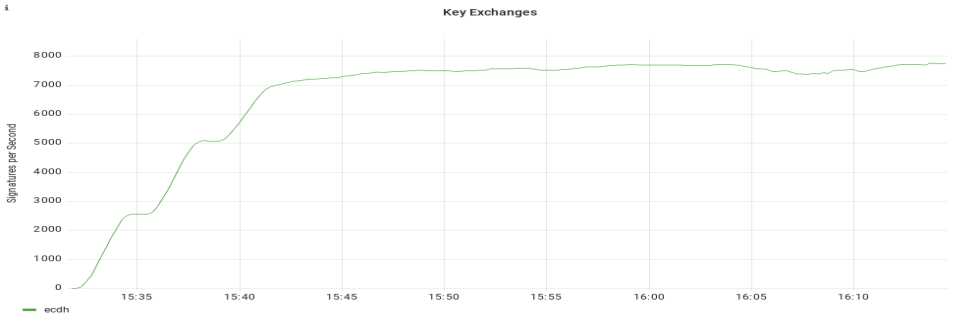


Figure A.20.: Number of key exchanges the exchange processes have done per second. Retrieved from the custom metric endpoint.

A.1.3. Load Statistics

Any information regarding load, i.e. the total number of HTTP requests/s is taken from the Nginx proxy's Prometheus exporter. Which means all requests are counted, including failed or the ones which Prometheus itself creates. The latter one is negligible since this is only one request every 5s, as configured in the Prometheus scrape settings.

Exchange

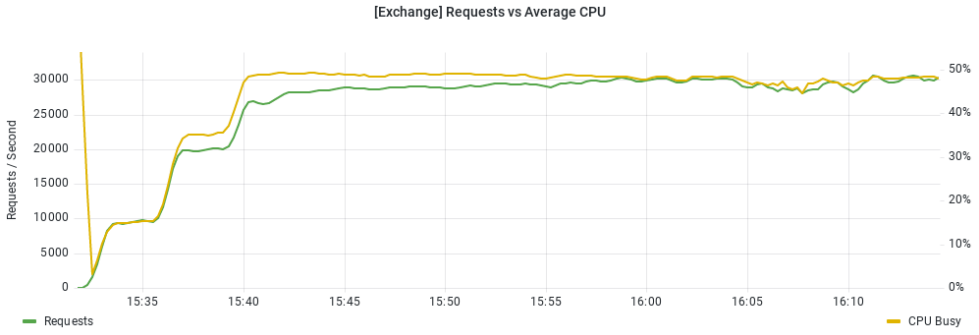


Figure A.21.: Number of HTTP requests per second in relation to the CPU load of the exchange node(s).

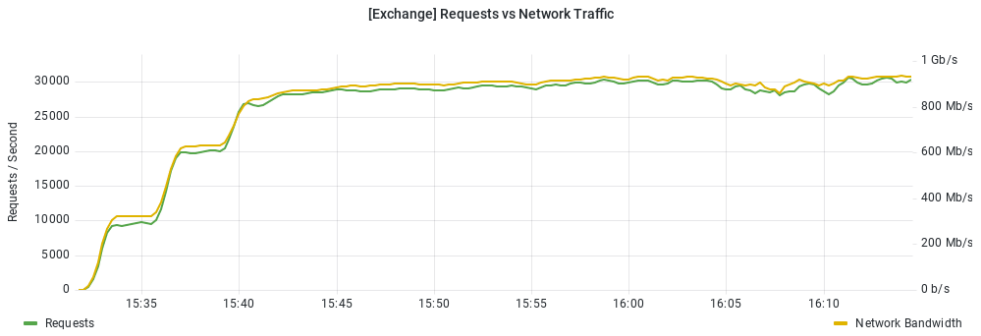


Figure A.22.: Number of bytes transmitted and received over the network on the exchange nodes, in relation to number of HTTP requests.

Database

The same panels as for the exchange are also available for the database node, plus the following ones:

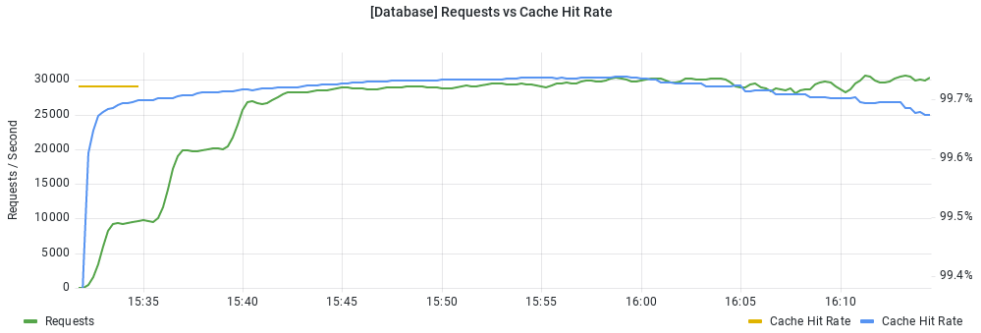


Figure A.23.: Number of HTTP requests per second in relation the database's cache hit rate.

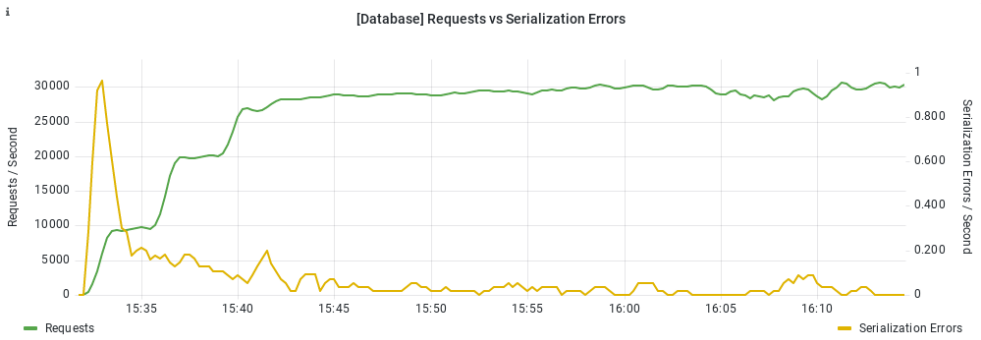


Figure A.24.: Number of HTTP requests per second in relation to number of serialization errors per second.

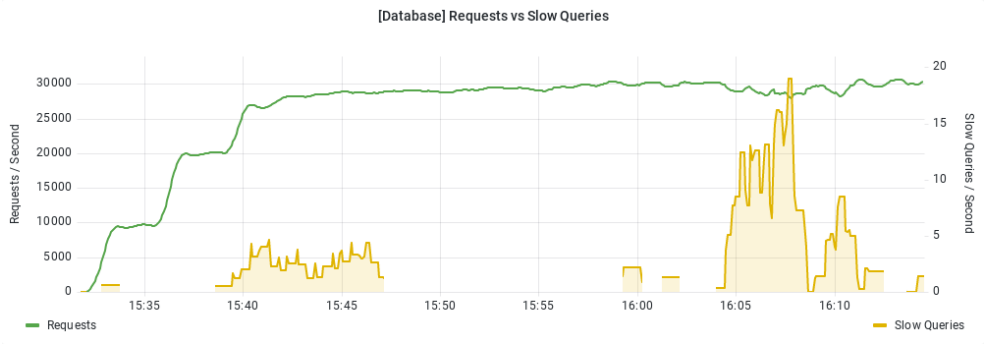


Figure A.25.: Number of HTTP requests per second in relation to number of slow queries per second.

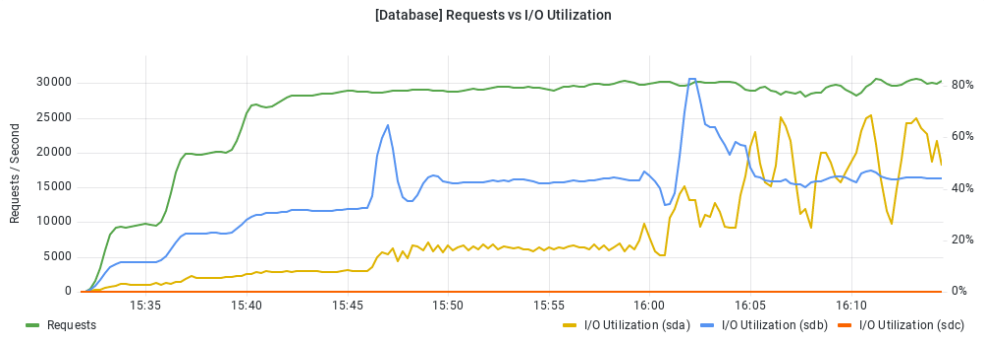


Figure A.26.: Number of HTTP requests per second in relation to the resulting IO load on the database node(s).

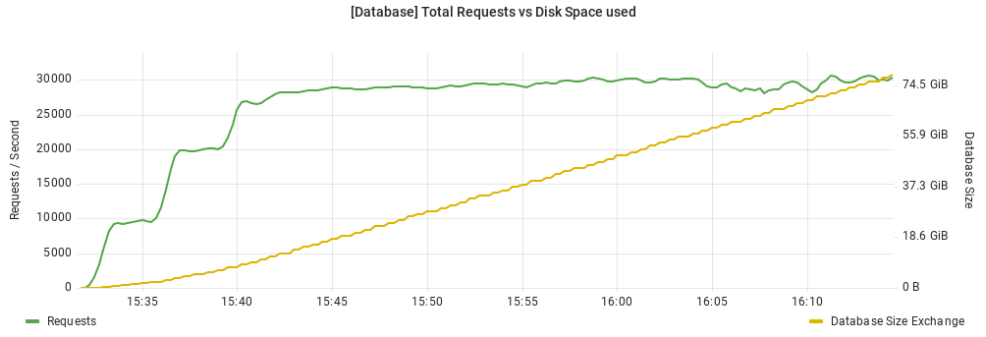


Figure A.27.: Total number of handled HTTP requests by the proxy in relation to the size of the database.

Other

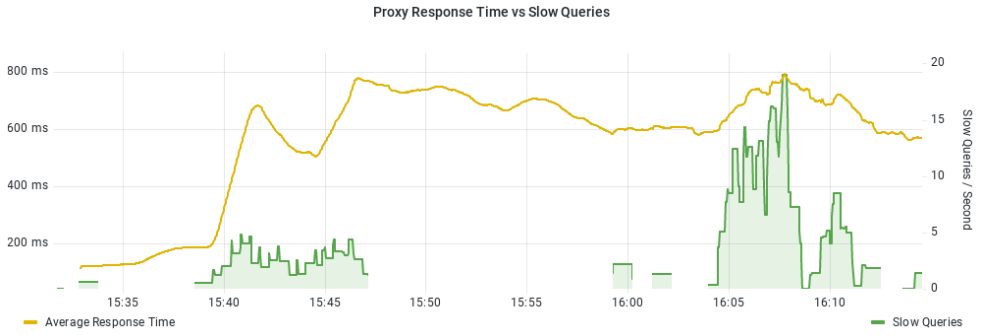


Figure A.28.: The average response time of the proxy for all HTTP requests, taken from its logs, compared to the number of slow queries per second.

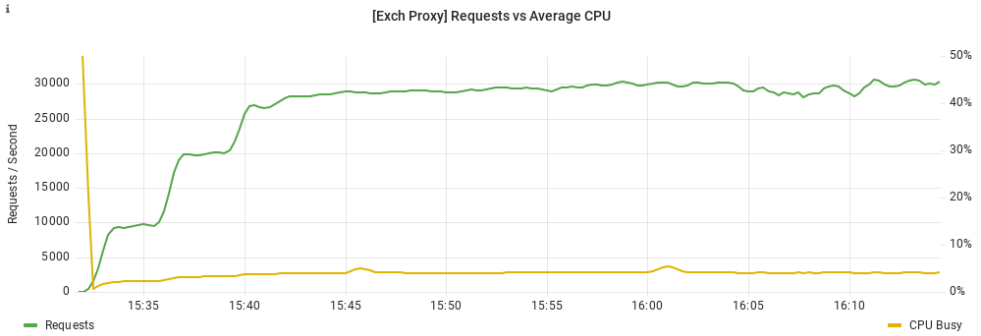


Figure A.29.: The average CPU usage of all exchange proxy nodes compared to the total number of requests.

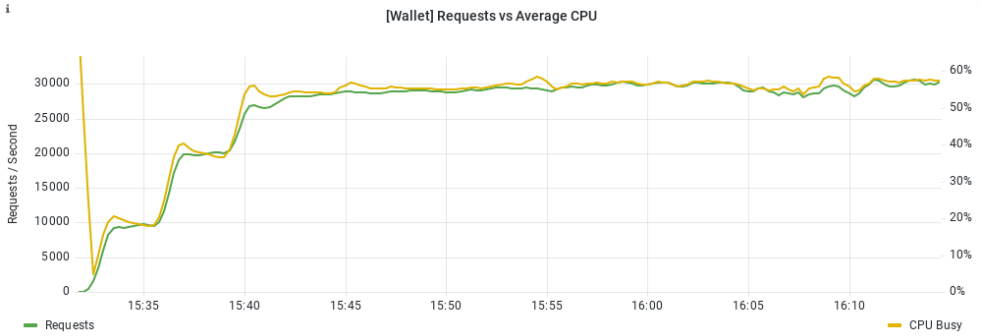


Figure A.30.: The average CPU usage of all wallet nodes compared to the total number of requests.

A.1.4. Request Statistics

The following described panels are repeated for every important endpoint. The data they visualize is retrieved from the Nginx proxy's logs. There are different statistics which are not

further explained here available, namely average, minimum, maximum, median, 90th/99th percentile and the standard deviation of the duration of a request. All of those, plus the number of successful and failed requests are calculated in an adjustable interval which can be set in the top left corner of the dashboard. Times are taken from the `request_time`[1] log entry of Nginx.

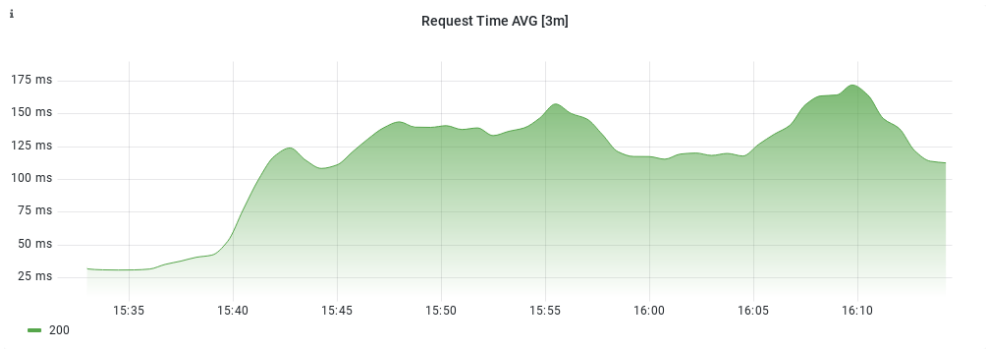


Figure A.31.: Request time average for successful requests (HTTP status 200), calculated from the `request_time` log entry of the Nginx proxy.

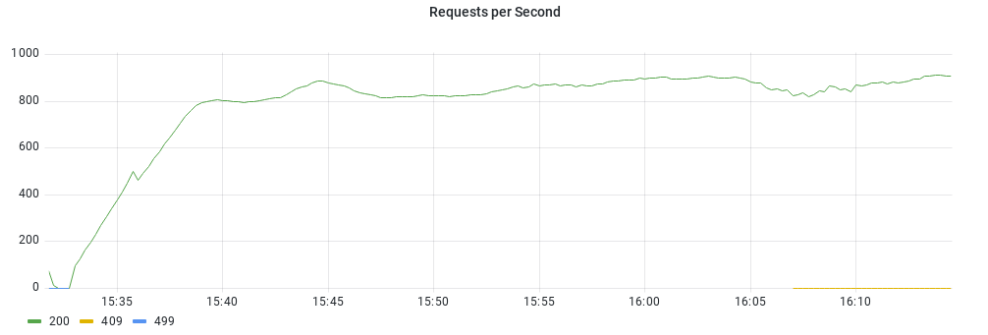


Figure A.32.: Number of requests by status code in the configured interval. Taken from the Nginx logs.

A.1.5. Database

We have imported this dashboard from the library, but in addition we have added some custom metrics to help us gain further insight into the performance of our database. We extended the prometheus-postgres-exporter with custom queries and finally added the following custom panels to the dashboard:

[1] Nginx Request Time: `https://nginx.org/en/docs/http/ngx_http_log_module.html#var_request_time`

Wait Events

Type	Event	Count per Second
Client	ClientRead	9.98
LWLock	SerializableFinishedList	0.691
LWLock	SerializableXactHash	0.118
LWLock	PredicateLockManager	0.0500
Activity	AutoVacuumMain	0
Activity	BgWriterHibernate	0
Activity	BgWriterMain	0
LWLock	BufferContent	0

Figure A.33.: Current number of wait events per second by type.

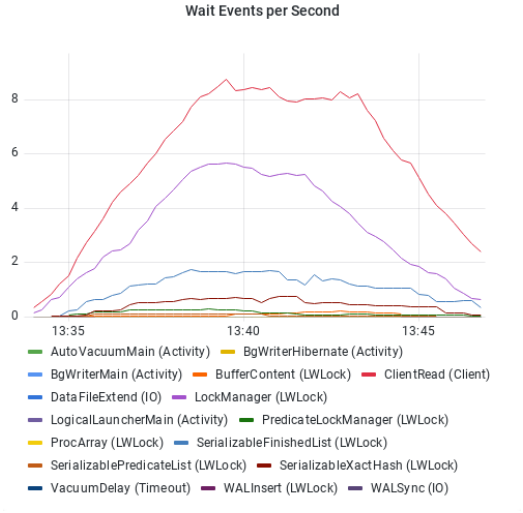


Figure A.34.: Number of wait events per second over time.

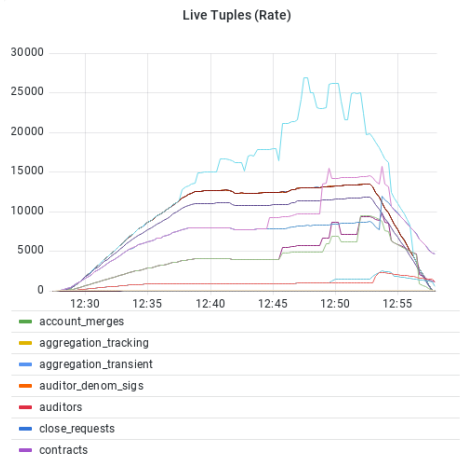


Figure A.35.: Rate of live tuples in each table in the taler-exchange database.

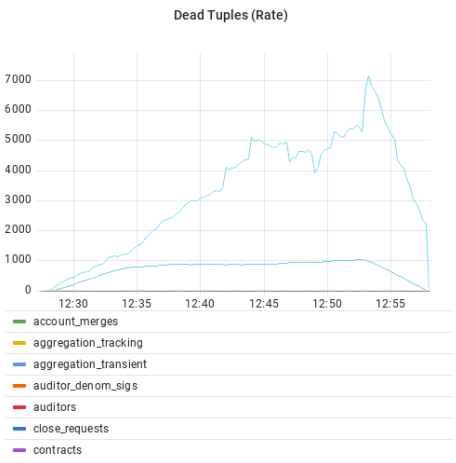


Figure A.36.: Rate of dead tuples in each table in the taler-exchange database.

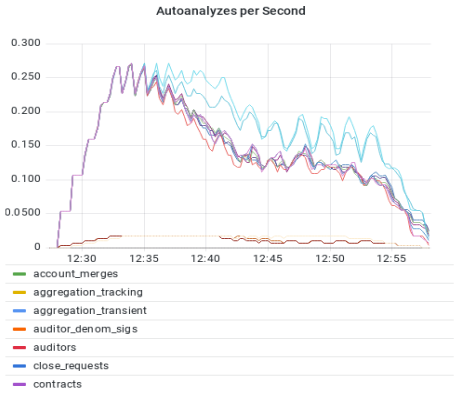


Figure A.37.: Number of times the tables in the exchange database were analyzed by autoanalyze (per second).

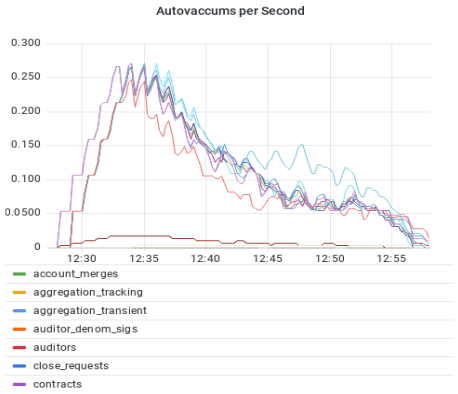


Figure A.38.: Number of times the tables in the exchange database were vacuumed by autovacuum (per second).

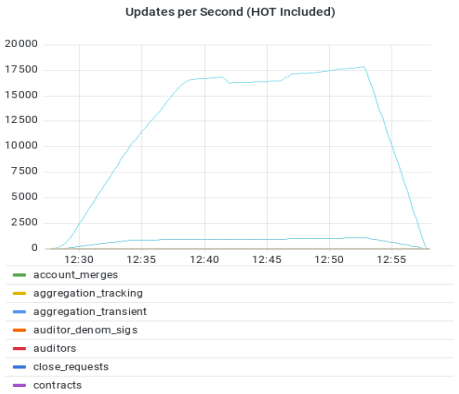


Figure A.39.: Number of updates per second on each table in the exchange database, this includes HOT updates.

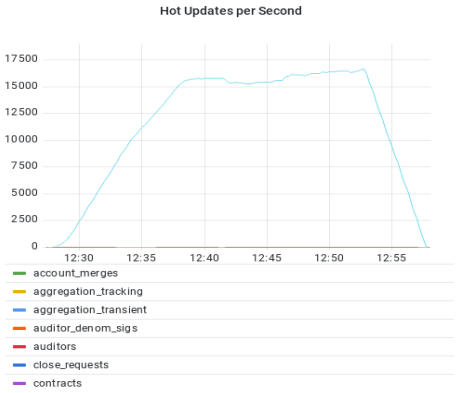


Figure A.40.: Number of HOT updates per second on each table in the exchange database.

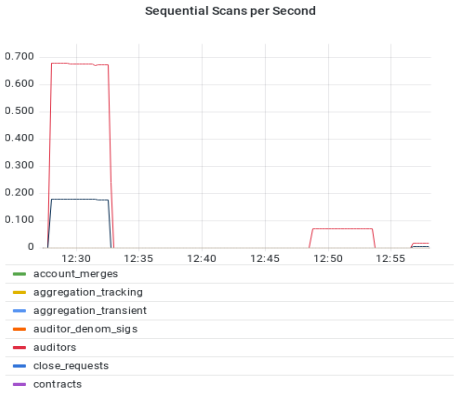


Figure A.41.: Number of sequential scans on tables in the exchange database (per second).

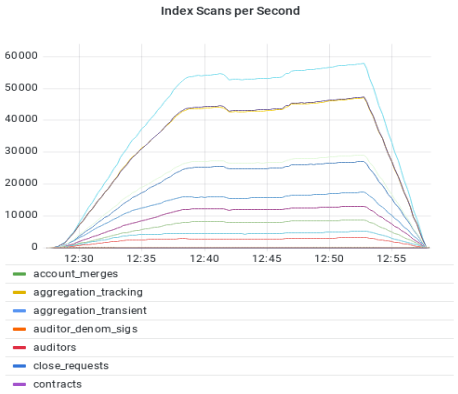


Figure A.42.: Number of index scans on tables in the exchange database (per second).

A.2. Partitioning and Sharding


```
Foreign Scan on public.known_coins_1 kc_1
(cost=100.00..135.59 rows=853 width=72) (actual time=1.292..16.353 rows=3677 loops=1)
  Output: kc_1.age_commitment_hash, kc_1.coin_pub, kc_1.denominations_serial
  Remote SQL: SELECT denominations_serial, coin_pub, age_commitment_hash FROM public.known_coins_1
Foreign Scan on public.known_coins_2 kc_2
(cost=100.00..135.59 rows=853 width=72) (actual time=0.815..15.568 rows=3723 loops=1)
  Output: kc_2.age_commitment_hash, kc_2.coin_pub, kc_2.denominations_serial
  Remote SQL: SELECT denominations_serial, coin_pub, age_commitment_hash FROM public.known_coins_2
Foreign Scan on public.known_coins_3 kc_3
(cost=100.00..135.59 rows=853 width=72) (actual time=0.948..15.970 rows=3586 loops=1)
  Output: kc_3.age_commitment_hash, kc_3.coin_pub, kc_3.denominations_serial
  Remote SQL: SELECT denominations_serial, coin_pub, age_commitment_hash FROM public.known_coins_3
Foreign Scan on public.known_coins_4 kc_4
(cost=100.00..135.59 rows=853 width=72) (actual time=1.170..16.815 rows=3615 loops=1)
  Output: kc_4.age_commitment_hash, kc_4.coin_pub, kc_4.denominations_serial
  Remote SQL: SELECT denominations_serial, coin_pub, age_commitment_hash FROM public.known_coins_4
```

Listing A.1: Important part of the execution plan of `get_melt` before materialized CTEs were used in queries. Its clear that the amount of rows returned by each shard (seen in `actual time=... rows=X`) is too high. In combination with other queries and the total load we reached the network capacity almost immediately. The plan after rewriting the query can be seen in Listing A.2. The explanation can be found Section 6.5

```
Foreign Scan on public.known_coins_1
(cost=100.00..120.74 rows=4 width=72) (never executed)
  Output: known_coins_1.age_commitment_hash, known_coins_1.coin_pub, known_coins_1.denominations_serial
  Remote SQL: SELECT **removed** FROM public.known_coins_1 WHERE ((coin_pub = $1::bytea))
Foreign Scan on public.known_coins_2
(cost=100.00..120.74 rows=4 width=72) (never executed)
  Output: known_coins_2.age_commitment_hash, known_coins_2.coin_pub, known_coins_2.denominations_serial
  Remote SQL: SELECT **removed** FROM public.known_coins_2 WHERE ((coin_pub = $1::bytea))
Foreign Scan on public.known_coins_3
(cost=100.00..120.74 rows=4 width=72) (never executed)
  Output: known_coins_3.age_commitment_hash, known_coins_3.coin_pub, known_coins_3.denominations_serial
  Remote SQL: SELECT **removed** FROM public.known_coins_3 WHERE ((coin_pub = $1::bytea))
Foreign Scan on public.known_coins_4
(cost=100.00..120.74 rows=4 width=72) (actual time=1.150..1.150 rows=1 loops=1)
  Output: known_coins_4.age_commitment_hash, known_coins_4.coin_pub, known_coins_4.denominations_serial
  Remote SQL: SELECT **removed** FROM public.known_coins_4 WHERE ((coin_pub = $1::bytea))
```

Listing A.2: Important part of the execution plan of `get_melt` after rewriting queries to use materialized CTEs. Compared to Listing A.1 we can clearly see that only one partition is hit and that we have to transmitt a lot rows less. (**removed** was added that the plan fits on one page and is a placeholder for the columns selected).

A.3. Promtail


```
pipeline_stages:
- match:
    selector: '{job="proxy"}'
    stages:
    - regex:
        # Regular expression to extract values and labels for prometheus
        expression: '.*uri=/(?P<ep>[a-zA-Z]+)(?:/\w+)? (?:/(?P<act>[a-zA-Z]+))? s=(?P<status>\d{3}).*'
    - template:
        # Make just one label by using go templates syntax out of endpoint (ep) and action (act)
        # which were extracted by the regex above
        source: endpoint
        template: '{{ printf "%s-%s" .ep .act | trimSuffix "-" }}'
    - labels:
        # define which extracted / computed values to add as labels
        endpoint:
        status:
    - metrics:
        # define which metrics to expose to prometheus
        total_requests:
          prefix: 'taler_requests_'
          type: Counter
          description: "Total Requests"
          config:
            match_all: true
            action: inc
```

Listing A.3: Example of Promtail configuration to calculate and deploy a custom metric named `taler_requests_total_requests` for Prometheus. The values are extracted from our Nginx logs using regular expressions. While this metric will be deprecated as rsyslog seems to be too slow, others are still required to calculate request time statistics for example.

A.4. PostgreSQL

A.4.1. Configuration Used During pgbench Benchmarks

```
shared_buffers=62GB
effective_cache_size=140GB
huge_pages=on
min_wal_size=4GB
max_wal_size=16GB
wal_buffers=1GB
work_mem=2GB
maintenance_work_mem=4GB
checkpoint_completion_target=0.9
checkpoint_timeout = 15min
random_page_cost=1.1
bgwriter_flush_after = 2MB
effective_io_concurrency = 200
fsync = off
synchronous_commit = off
full_page_writes = on
max_worker_processes=64
max_parallel_workers=64
max_parallel_workers_per_gather=12
max_parallel_maintenance_workers=12
```

Listing A.4: Custom PostgreSQL configuration used on a node in the Dahu cluster during pgbench benchmarks.

A.4.2. Final Configuration

```
listen_addresses='*'

log_destination=syslog
syslog_ident='taler-database'

log_error_verbosity=terse

log_min_duration_statement=50

auto_explain.log_min_duration='50ms'
auto_explain.log_verbose=true
auto_explain.log_nested_statements=true
auto_explain.log_analyze=true
auto_explain.log_buffers=true
auto_explain.log_wal=true

shared_preload_libraries='pg_stat_statements,auto_explain'

join_collapse_limit=1

log_autovacuum_min_duration=0
default_statistics_target=300
autovacuum_vacuum_cost_limit=400
autovacuum_vacuum_scale_factor=0.1
autovacuum_vacuum_threshold=1000
autovacuum_analyze_threshold=50
autovacuum_analyze_scale_factor=0.1

shared_buffers=49163965kB
effective_cache_size=147491895kB
```

```
huge_pages=on

min_wal_size=20GB
max_wal_size=200GB
wal_buffers=1GB

checkpoint_completion_target=0.9
checkpoint_timeout = 15min
checkpoint_flush_after = 2MB
random_page_cost=1.1

bgwriter_flush_after = 2MB
backend_flush_after = 2MB

effective_io_concurrency = 200
fsync = on
synchronous_commit = off

wal_compression = off
wal_sync_method = fsync

full_page_writes = on

max_worker_processes=64
max_parallel_workers=64
max_parallel_workers_per_gather=10
max_connections=500

max_parallel_maintenance_workers=12

max_locks_per_transaction=85

work_mem=2GB
maintenance_work_mem=4GB
idle_in_transaction_session_timeout=60000

enable_partitionwise_join=on
enable_partitionwise_aggregate=on

data_directory='/tmp/postgresql/13/main'
```

Listing A.5: The final configuration for PostgreSQL which we used in the Dahu cluster.

A.5. Performance Analysis

```
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 1  0      0  92009     25  18316    0    0     3   318  405   22 26 11 63  0  0
 0  0      0  92009     25  18316    0    0     0     0  261  320  0  0 100  0  0
 0  0      0  92009     25  18316    0    0     0 46616  216  257  0  0 100  0  0
45  0      0  92260     25  17996    0    0     0     0 509124 781029 35 13 52  0  0
44  1      0  92252     25  17998    0    0     0 82192 743920 1140285 51 20 30  0  0
50  0      0  92247     25  18003    0    0     0 65564 740366 1152212 52 18 30  0  0
41  0      0  92243     25  18005    0    0     0     0 736052 1138409 50 20 30  0  0
35  0      0  92240     25  18009    0    0     0     0 734265 1138483 50 20 30  0  0
36  0      0  92238     25  18012    0    0     0   264 741315 1152409 52 19 29  0  0
43  0      0  92236     25  18016    0    0     0    36 740374 1147759 51 20 29  0  0
46  0      0  92231     25  18020    0    0     0 163856 737491 1166498 53 19 28  0  0
47  0      0  92228     25  18023    0    0     0     0 741582 1171814 52 19 28  0  0
43  0      0  92223     25  18027    0    0     0 61584 741537 1168119 53 19 28  0  0
46  0      0  92220     25  18030    0    0     0    36 739695 1167456 53 19 28  0  0
43  0      0  92216     25  18034    0    0     0   228 741992 1150333 52 20 29  0  0
38  0      0  92214     25  18036    0    0     0 147464 740589 1166289 52 19 28  0  0
41  0      0  92209     25  18042    0    0     0     0 737148 1162946 52 19 28  0  0
44  0      0  92207     25  18044    0    0     0  2480 741757 1173128 53 19 28  0  0
39  0      0  92205     25  18049    0    0     0    48 740404 1170644 53 19 28  0  0
52  0      0  92201     25  18051    0    0     0   292 739032 1159037 52 19 28  0  0
42  1      0  92198     25  18054    0    0     0 20072 740101 1165594 52 20 28  0  0
51  0      0  92194     25  18059    0    0     0 45464 738055 1165382 53 19 28  0  0
41  0      0  92190     25  18062    0    0     0     0 742838 1172377 53 19 28  0  0
45  0      0  92185     25  18067    0    0     0    36 740704 1174534 53 19 28  0  0
50  0      0  92182     25  18069    0    0     0    92 741691 1150716 52 19 28  0  0
42  0      0  92177     25  18073    0    0     0    28 740220 1168488 53 18 28  0  0
44  0      0  92174     25  18077    0    0     0     0 738818 1164769 53 19 28  0  0
46  0      0  92172     25  18080    0    0     0     0 740720 1169902 53 19 28  0  0
46  0      0  92166     25  18083    0    0     0 90404 592524 945810 42 15 43  0  0
45  0      0  92161     25  18087    0    0     0  3884 746310 1159898 52 19 28  0  0
49  0      0  92157     25  18090    0    0     0    20 747242 1177750 53 19 28  0  0
36  0      0  92152     25  18094    0    0     0     0 744477 1173832 53 19 28  0  0
46  0      0  92149     25  18098    0    0     0     0 746194 1172700 53 19 28  0  0
39  0      0  92147     26  18101    0    0     0 49768 745651 1177462 54 18 28  0  0
43  0      0  92143     26  18105    0    0     0   212 744968 1161110 53 19 28  0  0
43  0      0  92138     26  18109    0    0     0     0 743223 1176960 54 19 28  0  0
43  0      0  92135     26  18112    0    0     0 81920 745168 1173574 53 19 28  0  0
48  0      0  92132     26  18116    0    0     0     0 743174 1169255 53 19 28  0  0
48  0      0  92129     26  18120    0    0     0    68 592295 933445 42 15 43  0  0
41  0      0  92124     26  18123    0    0     0    76 740354 1162221 52 19 28  0  0
49  0      0  92120     26  18125    0    0     0     0 738456 1158291 53 19 28  0  0
39  0      0  92117     26  18129    0    0     0 147536 740735 1162479 52 20 28  0  0
49  0      0  92113     26  18133    0    0     0     0 737209 1165532 53 20 28  0  0
49  0      0  92111     26  18137    0    0     0    40 741185 1168133 54 19 28  0  0
45  0      0  92110     26  18140    0    0     0  4000 740693 1141945 52 20 28  0  0
42  0      0  92105     26  18144    0    0     0     0 741857 1168830 53 19 28  0  0
43  0      0  92102     26  18147    0    0     0     8 742546 1168867 54 18 28  0  0
43  0      0  92101     26  18150    0    0     0 147456 741941 1166646 53 19 28  0  0
41  1      0  92097     26  18154    0    0     0 64192 740052 1169040 53 19 28  0  0
48  0      0  92094     26  18158    0    0     0 27484 737224 1139511 52 20 28  0  0
47  0      0  92087     26  18162    0    0     0     0 740821 1165037 53 19 28  0  0
54  0      0  92059     26  18164    0    0     0     4 737109 1155098 53 19 27  0  0
38  0      0  92051     26  18170    0    0     0 147456 701847 1075069 55 20 25  0  0
35  0      0  92064     26  18174    0    0     0    44 723153 1125736 54 19 27  0  0
48  0      0  92056     26  18179    0    0     0 53008 734590 1134838 52 19 29  0  0
46  0      0  92053     26  18183    0    0     0     0 741595 1166891 53 19 28  0  0
46  0      0  92049     26  18186    0    0     0     0 740196 1170838 54 19 27  0  0
31  1      0  92045     26  18191    0    0     0 98304 741800 1170076 54 18 28  0  0
44  0      0  92043     26  18194    0    0     0 49188 733352 1173652 53 20 27  0  0
43  0      0  92028     26  18198    0    0     0   116 733522 1151497 53 20 27  0  0
44  0      0  92037     26  18201    0    0     0     0 730364 1137665 53 19 27  0  0
37  0      0  92035     26  18204    0    0     0     0 742348 1164945 53 19 28  0  0
44  0      0  92031     26  18208    0    0     0     0 739273 1165044 53 19 28  0  0
52  0      0  92028     26  18211    0    0     0 147888 739274 1164496 53 19 28  0  0
50  0      0  92024     26  18215    0    0     0   144 739684 1145210 53 19 28  0  0
50  0      0  92020     26  18219    0    0     0     0 742847 1167779 54 18 28  0  0
38  0      0  92016     26  18223    0    0     0     0 738079 1166580 53 19 28  0  0
36  0      0  92013     26  18226    0    0     0     0 742687 1171101 54 18 27  0  0
48  0      0  92009     26  18229    0    0     0 147500 741536 1166846 53 19 28  0  0
```

```
40  0      0  92006     26  18233    0    0     0 94600 740746 1147102 52 20 28  0  0
45  0      0  92001     26  18238    0    0     0     0 741119 1163851 53 19 28  0  0
48  0      0  91999     26  18241    0    0     0     0 740995 1167197 53 19 28  0  0
35  0      0  91996     26  18244    0    0     0     0 742235 1165666 53 19 28  0  0
44  1      0  91993     26  18248    0    0     0 49192 741392 1164506 53 19 28  0  0
43  1      0  91990     26  18251    0    0     0 124876 743695 1144639 52 19 29  0  0
48  0      0  91987     26  18255    0    0     0 24864 737759 1159383 52 20 28  0  0
44  0      0  91983     26  18258    0    0     0     0 740224 1164983 53 19 28  0  0
43  0      0  91980     26  18262    0    0     0     0 741742 1168140 54 19 27  0  0
18  0      0  91976     26  18267    0    0     0    36 737449 1162293 53 19 28  0  0
49  0      0  91973     26  18269    0    0     0 147576 741462 1148048 52 20 28  0  0
43  0      0  91969     26  18274    0    0     0     0 742408 1168332 54 19 27  0  0
43  0      0  91966     26  18277    0    0     0     0 738803 1164992 53 19 28  0  0
39  0      0  91963     26  18280    0    0     0     4 737891 1159372 52 19 28  0  0
43  0      0  91962     26  18283    0    0     0    40 741888 1166835 53 19 28  0  0
48  0      0  91958     26  18287    0    0     0 164144 738677 1145900 52 20 28  0  0
46  0      0  91955     26  18291    0    0     0     0 740956 1165789 53 19 28  0  0
44  0      0  91952     26  18295    0    0     0     0 741055 1166460 53 19 28  0  0
44  0      0  91948     26  18299    0    0     0     8 739414 1165698 53 19 28  0  0
46  0      0  91945     26  18301    0    0     0    48 743218 1165277 53 19 28  0  0
36  0      0  91941     26  18305    0    0     0   208 736320 1134425 51 20 29  0  0
47  0      0  91936     26  18309    0    0     0 239096 739799 1159730 52 19 28  0  0
45  0      0  91932     26  18312    0    0     0     0 742477 1167618 53 20 28  0  0
45  0      0  91928     26  18316    0    0     0     0 736442 1159690 52 20 28  0  0
47  0      0  91926     26  18319    0    0     0    76 737145 1157620 52 20 28  0  0
48  0      0  91921     26  18323    0    0     0    64 739999 1146323 52 19 29  0  0
50  0      0  91918     26  18326    0    0     0 197176 739590 1159797 52 19 28  0  0
50  0      0  91915     26  18330    0    0     0     0 740533 1166111 53 19 28  0  0
52  0      0  91911     26  18334    0    0     0     0 739776 1161328 52 20 28  0  0
42  0      0  91907     26  18338    0    0     0    60 590783 929545 41 16 43  0  0
39  0      0  91904     26  18341    0    0     0  4248 744434 1161062 52 19 29  0  0
41  1      0  91900     26  18345    0    0     0 114688 741817 1163511 53 19 28  0  0
14  0      0  91928     26  18349    0    0     0 32768 598242 996868 43 15 42  0  0
 0  0      0  91951     26  18349    0    0     0     0 41914 84357  3  1 96  0  0
 0  0      0  91952     26  18349    0    0     0    36  174  204  0  0 100  0  0
 0  0      0  91954     26  18349    0    0     0   276 7897 13403  0  0 99  0  0
 0  0      0  91954     26  18349    0    0     0     0 1911 3678  0  0 100  0  0
 0  0      0  91954     26  18349    0    0     0 147456  330  351  0  0 100  0  0
```

Figure A.43.: `vmstat` output while running pgbench against a database in the Dahu cluster, note that most of these metrics can be seen in the node-exporter dashboards too.

A.6. Exchange Database Schema

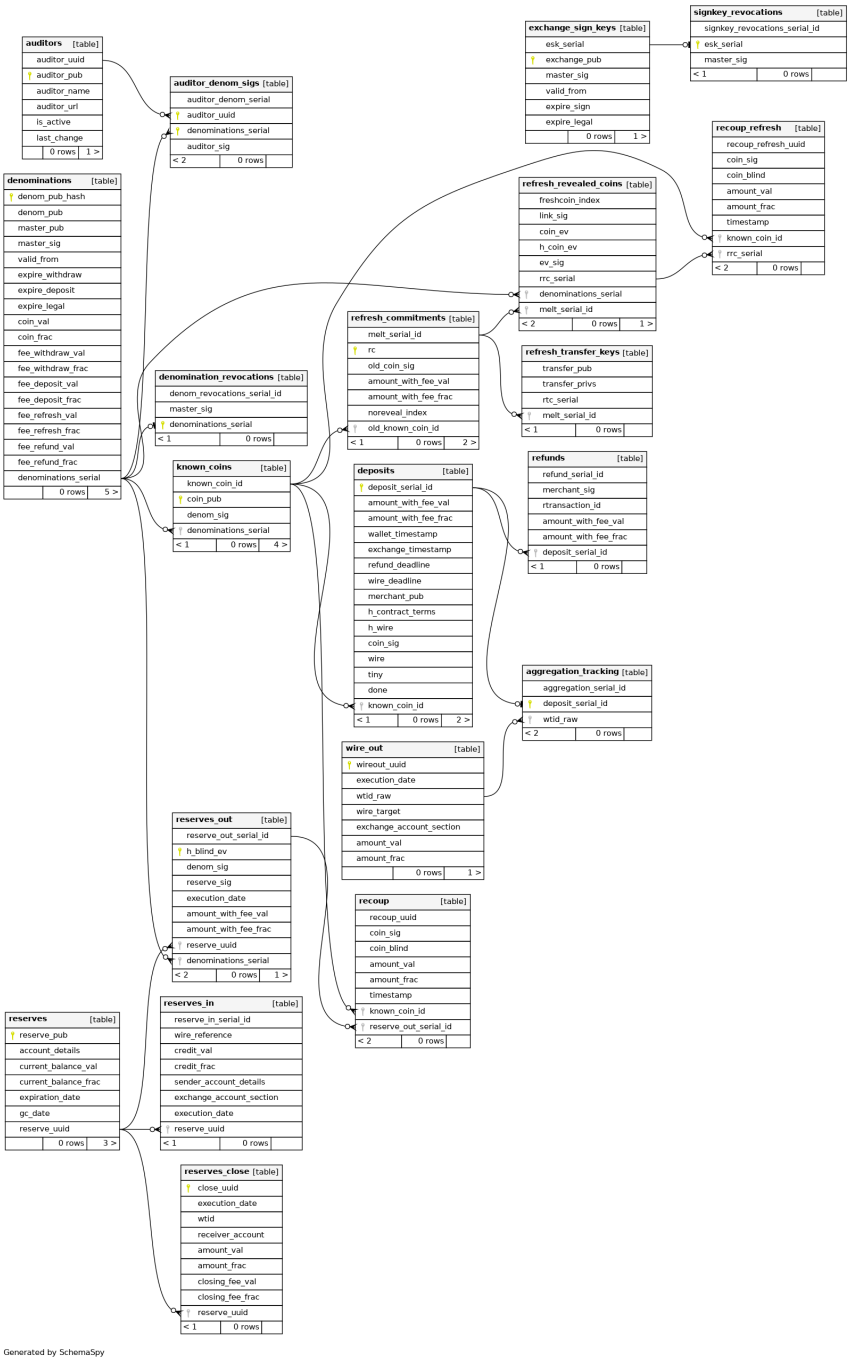


Figure A.44.: The database schema before we started partitioning the database. This figure was taken from `https://docs.taler.net`. After adding partitions, most of the foreign keys have disappeared, resulting in a schema without references, but apart from the additional materialized indexes, the schema has remained the same.

A.7. Thesis Assignment

Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Bachelor Thesis

for	Marco Boss
Subject Area	Computer Science
Supervisor	Christian Grothoff

GNU Taler Scalability

Introduction

Currently, central banks all over the globe are investigating possible implementation of a Central Bank Digital Currency (CBDC). Unfortunately, most of today's electronic payment systems do either not offer adequate technical privacy assurances to citizens, or are too slow to handle the expected transaction load.

Task

One of the goals of the GNU Taler project is to provide a free software reference implementation for a privacy preserving CBDC.

The Task of the Bachelor's Thesis is to assess and improve the scalability of GNU Taler with the goal of 100'000 transactions per second (TPS). While the setup was part of previous work, the main focus is now on improving the performance by identifying and fixing further bottlenecks, such as the used Postgres Database:

- Extend distributed setup of GNU Taler (horizontally)
- Distribute database load (horizontally)
- Assess the performance with experiments on the Grid'5000 platform

Stretch goals include broadening the setup to include additional Taler components in the setup:

- the merchant backend
- the auditor
- the "sync" wallet backup service

Advisor:	Head of Department:
16. Februar 2022	16. Februar 2022
Christian Grothoff	
Christian Grothoff (16. February 2022 19:37 GMT+1)	