Using Docker and ZFS to speed up development time

Matic Adamič
University of Primorska
Faculty of Mathematics, Natural Sciences and Information Technologies
89202071@student.upr.si

Aleksandar Tošić
University of Primorska
Faculty of Mathematics, Natural Sciences and Information Technologies
aleksandar.tosic@upr.si

May 2023

Abstract

Testing is an important part of software development. However, testing can be time consuming especially when the software depends heavily on a database. Moreover, testing the software is greatly interconnected with the state of the database. Tests have requirements regarding the state of the database. In this paper we propose an architecture that aims to speed up the testing phase in software development, when testing is done via traditional databases and often time consuming. The solution takes care of generating new isolated database instances, which can be queried, re-used or restarted at any given time. These database instances are time and space efficient. We argue this leads to faster testing, due to isolated instances that can be used in testing in parallel, and therefore speeds up the entire development time, helping teams solve bugs and deliver software updates faster.

Keywords— ZFS, Docker, database, development

1 Introduction

Development pipelines, such as Continuous integration and Continuous deployment (CI/CD) are an essential part of modern software development. They offer a way for developers to work on new features and fix problems in existing ones. CI is a practice almost all software development teams use to update their code base. It can generally be viewed as a process of three main steps: 1) make code-base changes, 2) test those changes, 3) merge changes.

Two of these steps, keeping track of code base changes and merging, usually involve a version control system (for an example Git). The in-between step, testing, is usually the step that differs the most between different development teams. This is usually due to the nature of the product that is being developed. Many software products that must and still are being maintained to this day suffer from what the industry call "legacy code-bases". Legacy code-bases are usually pieces of fully developed software, which can be very complex. It can sometimes be hard to integrate them into what are considered today's best testing practices.

In order to test such programs, it's considered good practice to not only test individual parts of the program (class-level tests), but also the application as a whole, usually known as integration testing. Such a test must include a database server instance, which would mimic what actually happens in a production environment. Keep in mind, such programs can depend on which database is used; the program might work with MySql, but not with an Oracle database.

We propose a solution for legacy code-bases which perform any kind of batch processing on potentially very large databases. The solution is a system which allows one to generate multiple database instances for a specific database, which all share the same underlying data. Each such instance is independent of one another and can be started, stopped or re-started at any given time. Managing an instance (starting and stopping) is independent of the size of the database,and is space efficient. Two main pieces of technologies that are used in order to achieve this system are ZFS and Docker.

We collect data from a a medium sized company, a team of 6 developers, and track their requests for database instances, over a period of 3 weeks, to see how much time and space is saved by using this system.

Finally, we present the results of collected data and compare it to a more primitive system, that we'll refer to as a baseline. In section 2 we propose the architecture of the system and give a high level overview. In section 2.1 we give a brief introduction to ZFS and mention the features that are most important to this system. In the following section 2.2 we give a short introduction to Docker and it's role in the system. In section 3 we present results of using the system in practice and in section 5 we conclude tour findings and shortcomings and pitfalls of this system.

2 Architecture

The two main pieces of this system are ZFS and Docker. What follows are brief introductions to both technologies, with a focus on key features that enable us to implement the proposed system. In section 2.3 we show how, in theory, ZFS and Docker can be combined to generate the proposed database instances.

2.1 ZFS

ZFS, known as the Zettabyte File System or the Z File System, is an advanced local file system and volume manager. It offers many features, such as data replication, duplication, compression, and scalability, as it can address up to 256 quadrillion zettabytes ($256 * 10^{24}$ TB) [2, 4]. For considerations of this work, we will focus on two most important ones, that will allow us to implement the wanted system. Those are copy-on-write and snapshots.

2.1.1 CoW (Copy on Write)

ZFS offers immutable datasets. Most file systems usually perform content replacement in-place, in order to save space. ZFS however, gives you the ability to create a mutable dataset, from an immutable dataset. All writes go through the mutable dataset, where all changes are written to new blocks. Meanwhile, the root data, that is the immutable dataset, is left completely unchanged.

It is possible to have multiple, simultaneous mutable datasets from the same immutable set. Each new mutable dataset is space efficient, as it only contains the blocks that have been changed. Creating a mutable dataset is fast, since all ZFS has to do is copy some light-weight data structures that point to some blocks on the disk. These mutable datasets can be "snapshotted" as the undergo changes, which in turn makes them immutable [2, 3].

2.1.2 Snaphots

As a consequence of the CoW feature, snapshots represent the state of a dataset in time. They can be seen as "commits" of work on a mutable dataset. Snapshots of a dataset follow each other sequentially in time, as each new snapshot contains accumulated changes from a previous snapshot, thus creating a tree structure that represents the history of a dataset. History is a tree structure due to possibility of branching out to multiple snapshots, from a given snapshot, and not just one. The useful feature of snapshots is that it is possible to create mutable simultaneous datasets from any one of them, meaning one has access to the history of a dataset [2, 3].

The diagram 1 shows an example of a file and it's snapshot history of changes. Not that in ZFS this entire dataset occupies 7 KB, which contains 5 different versions of the same underlying file, compared to a regular file system, which would need to copy this data 5 times, in total occupying 18 KB.

This is mainly because ZFS snapshots share the same underlying data blocks, which are indexed by the immutable datasets. ZFS only writes new blocks when mutable datasets are changed, data is added or deleted. In the example, mutable datasets are denoted with blue, and their practical size is 0 KB on the disk, since no data has been mutated.

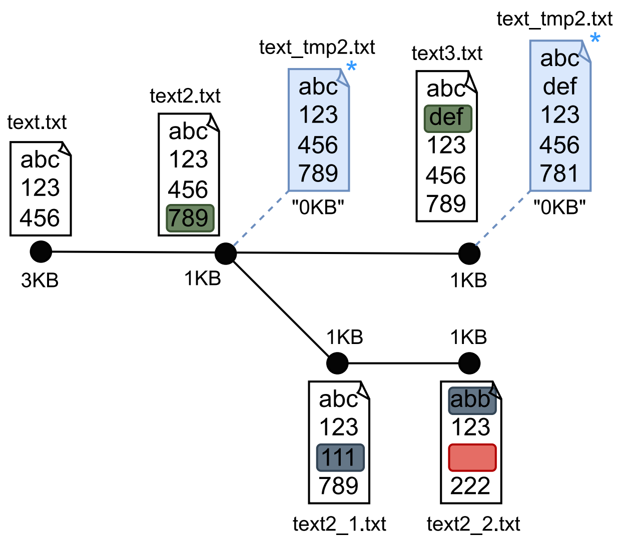


Figure 1: History of a *text.txt* file and its history of snapshots. Blue files indicate mutable datasets, which can modify the contents of a file.

2.2 Docker

Docker is platform that offers OS-level virtualization in packages known as containers. This allows developers to package their application and all it's dependencies in a fixed environment - a *container*. A container acts as a light-weight virtual machine, with it's own filesystem, networking and environment. Containers are started from *images*, which is read-only template which define instructions on how containers must be started. They also act as snapshots and are starting points, from which a containers are started from. Once an image is defined, multiple containers can started from it [1].

Docker has a few key features, which contribute to the solution we have talked above:
Speed: starting containers usually takes only a few seconds.
Scalability: Due to resource efficiency and fast start-up times, it is possible to start or stop a lot of containers in a short period of time.
Isolation: all containers are isolated from each other, as each container contains it's own filesystem and processing resources [5].

2.3 Creating a database instance

In order to create a database instance, a mutable ZFS dataset is required. In our case, a dataset is a regular database file, which will be imported to the database server. The database server will be running inside a docker container. The majority of database containers are already available on the official docker image repository. Such images can be used and modified slightly to fit the specific use cases.

Each container can be run with an external volume. In order for the database server, inside the container, to access the database file, the container instance must be started with such external volume. This external volume in this case will be ZFS's mutable dataset. Note that there is a 1:1 relationship between database containers and mutable ZFS datasets that are mounted to them. Only one container should be accessing a given mutable dataset. If one needs multiple instances of the same database, then the same amount of mutable datasets must be created with ZFS.

In short, a database instance can be created by the following steps:

1. Create a dataset from the database file
2. Create a mutable dataset from the database dataset
3. Run a docker container with the mutable dataset as the external volume
4. Instruct the database server inside the container to look for a database in the external volume (is usually prepared in advance)

5. Connect to the database/container

When shutting down an instance of a database, the container must be stopped and ZFS mutable dataset must be destroyed afterwards. A high-level view of this system can be seen in figure 2, where three developers are working simultaneously on 2 different database versions.

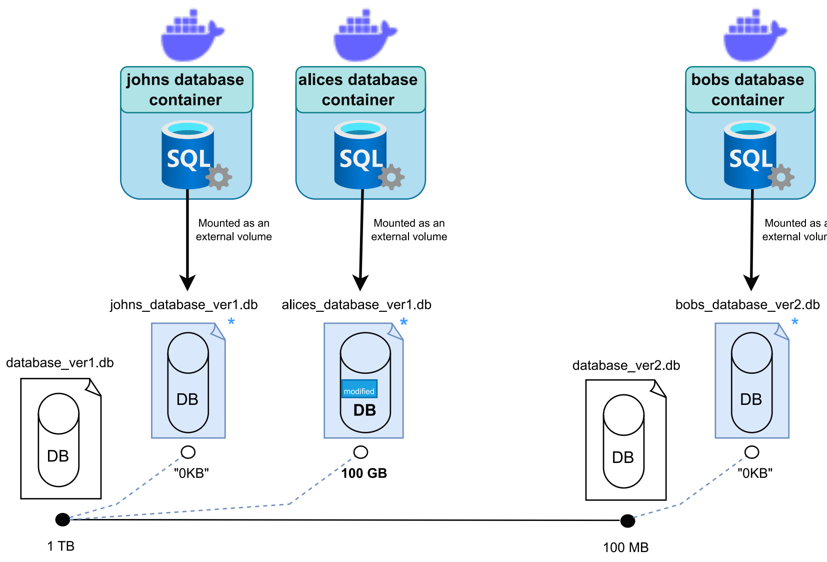


Figure 2: Overview of the ZFS+Docker system. John and Alice are working from the same database version, where John hasn't updated its own database, but Alice has accumulated 100GB worth of changes. Only Alice can see her changes to the database. Bob is working on his own database, which has been modified by a 100MB worth of data compared to the database version Alice and John are working on.

Table 1: Daily averages for all databases and comparison between systems

	size	**instantiated**	**total data copied**	**baseline**	**ZFS+Docker**
psi	1 760 GB	6.4	11 264 GB	6h 15 min	1 min 23 s
psk	285 GB	2.06	587 GB	20 min	27 s
psr	411 GB	3.26	1 334 GB	45 min	42 s
phr	252 GB	3.73	822 GB	27 min	48 s
pmk	50 GB	2.13	107 GB	3 min	28 s
pba	178 GB	2.66	473 GB	16 min	35 s
total	2 936 GB	20.24	14 487 GB	8h 6min	4 min 23 s

3 Results

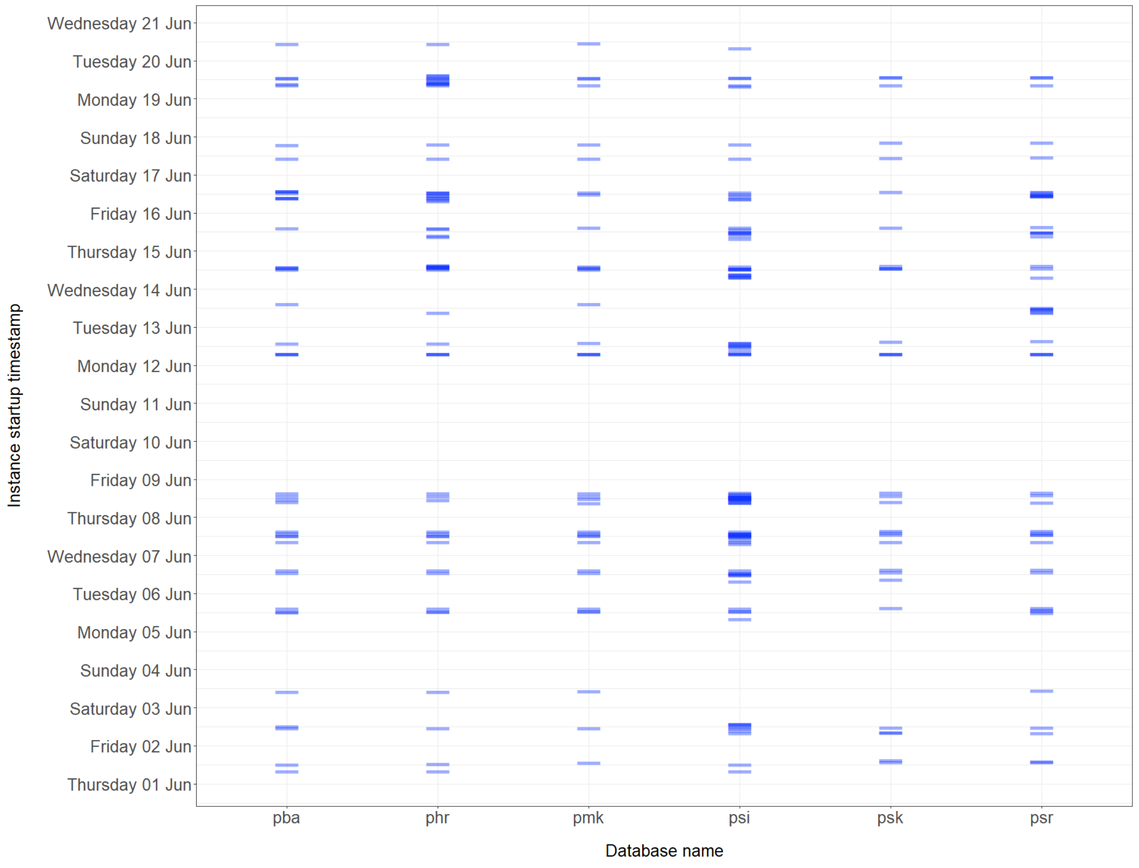
We gather results from a team of 6 developers over a period of 3 weeks for 6 different databases. We log each time a database instance was created. Graph for instance requests for every database can be seen in figure 3.

We calculate the amount of data that must effectively be copied if the *baseline* system was used. The *baseline* system being the most primitive way to implement database instances: copy a database file on disk and import it as another database in your database server.

The total instance requests per database were calculated from gathered data, where we excluded the requests made over the weekend, as those are usually made by automated systems and not by developers. We also calculate the amount of time spent creating instances for the baseline system assuming disk I/O speeds at 400MB/s. Time taken for an instance to be created by ZFS and Docker is assumed to be 13 seconds, as that is the average speed of instantiation on the development teams database server - around 3 seconds for existing container shutdown and snapshot clone to be deleted, 3 seconds for generating a mutable database dataset, and the rest being the container startup and for the database server inside the container to be fully initialized.

Comparison between the proposed system and the baseline is seen in table 1. Note that the baseline system must actually copy in total 14.4TB of data every day, meanwhile the ZFS and Docker system does not need to copy any data when an instance is created.

Figure 3: Graph of requests for database instances over a period of 3 weeks



4 Further research

Both Docker and ZFS bring some overhead by running databases in such configuration. There are a few key things to consider when setting up a system like this. First, the size of ZFS data blocks can be adjusted. This is important, as it plays a key role in how fast a mutable dataset will grow as changes are made to the database, specially when changes are "small", but happen to be scattered across different blocks.

Second, is measuring the impact on the SQL server performance as it is running inside a container, and not natively. Usually, Dockers performance is acceptable, but in the case of heavy data processing applications, whose performance might be critical, such a thing must be considered.

5 Conclusion

We presented a system for fast database instatiation and compared it to a baseline system. Results show that this system is much faster and is more space efficient. Another benefit of this system is that it scales independently of the database size, since ZFS does not copy any data when mutable datasets are created.

We showed that on average, in a team of 6 developers, about 20 database instances are requested every day. That totals to over 14TB of data "replication" everyday. The proposed system is orders of magnitude faster, in creating database instance and scales independently of the database size, while also having a minimal space footprint compared to the baseline system.

Another thing to consider, would be to modify this system by only having one container for multiple databases. Currently, there is no way to add external volumes to a running container, so one would have to commit and stop the running container which is "hosting" multiple databases, then start it again by appending another volume. Since this is tedious and harder to manage, and would prevent prevent instance independence, since all database users would have to wait for container to restart. Because running multiple containers and SQL servers inside them is not too expensive, some overhead is acceptable - each database instance having it's own container and SQL server inside.

References

[1] Charles Anderson. Docker [software engineering]. *IEEE Software*, 32(3):102–c3, 2015.

[2] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. The zettabyte file system. In *Proc. of the 2nd Usenix Conference on File and Storage Technologies*, volume 215, 2003.

[3] Open-e. Zfs essentials – copy-on-write snapshots. https://www.open-e.com/blog/copy-on-write-snapshots/.

[4] Oracle. Oracle solaris zfs administration guide. https://docs.oracle.com/cd/E23823$_0$1/$html$/819 $-$ 5461/$zfsover - 2.htmlgayou$.

[5] Babak Bashari Rad, Harrison John Bhatti, and Mohammad Ahmadi. An introduction to docker and analysis of its performance. *International Journal of Computer Science and Network Security (IJCSNS)*, 17(3):228, 2017.