Big Data - Labo

Tuur Vanhoutte

April 24, 2021

Contents

1 Intro

Topics:

- Linux basics + containers
- Elastic search (text search, document store)
- Linux Batch Processing & Dask
- InfluxDB (timeseries)
- Cloud services (Kafka, Kinesis, Lambda, ML services, ...)

2 NAT-ing

2.1 NAT

= Network Address Translation

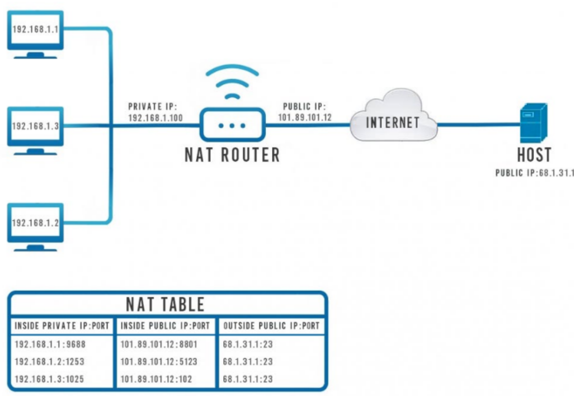


Figure 1: NAT diagram

2.1.1 The problem

- We only have one (public/private) IP-address
  - Howest: 172.23.82.60
- Connecting to a server over a network:
  - Using a protocol (HTTP) which uses TCP
  - Our server has an IP address: 172.23.82.60
  - Our server is listening at port 5000
  - $\Rightarrow$ `http://172.23.82.60:5000`
- Problem: We want to have multiple IP addresses
  - Student 1 wants to reach http://192.168.20.21:5000
  - Student 2 wants to reach http://192.168.20.22:5000
  - Student x wants to reach http://192.168.20.xx:5000

2.1.2 The solution

Translation is needed!

- 172.23.82.60:5000 should point to 192.168.20.21:5000
- 172.23.82.60:5001 should point to 192.168.20.22:5000
- 172.23.82.60:5xxx should point to 192.168.20.xx:5000

We can use any port, on both sides:

- 172.23.82.60:8000 can point to 192.168.20.21:5000
- 172.23.82.60:8000 can point to 192.168.20.21:3000

2.2 SSH Tunnel

= SSH Port Forwarding

Resource	Internal IP	Username	Password	External port	Internal port
Vyos Router	192.168.50.1	vyos	P@ssw0rd	7000	22
Storage	192.168.50.2	student	P@ssword	n.v.t.	22
SSH	192.168.50.3	student	P@ssword	7040	22
RDP	192.168.50.4	Administrator	P@ssword	7020	3389
vCenter vSphere	192.168.50.10	administrator@vsphere.local	P@ssword	7060	443
vCenter appliance	192.168.50.10	root	P@ssword	n.v.t.	5480
ESXi-00	192.168.50.11	root	P@ssword	n.v.t.	22
ESXi-01	192.168.50.12	root	P@ssword	n.v.t.	22

Figure 2: Example

Figure 3: Example: a tunnel is opened and we log into user@instance

3 Container technology

3.1 Docker

- Docker = ecosystem for creating and running containers
- Docker wants to make it possible to install and run software on any system

- Other reasons: Microservices/DevOps/Resource usage
- Docker != Container
  - Docker CLI
  - Docker Engine
  - Docker Image
  - Docker Container
  - Docker Hub
  - Docker Compose
  - Docker Swarm
  - ...

3.2 Microservices

- = A software development technique
- Structure an application as a collection of loosely coupled services
- Lightweight
- Microservices-based architectures enable continuous delivery and deployment
- `https://en.wikipedia.org/wiki/Microservices`

3.2.1 Monolithic vs Microservices

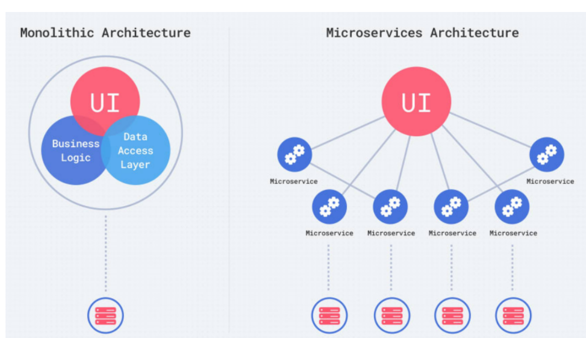


Figure 4: Monolithic architecture vs Microservices architecture

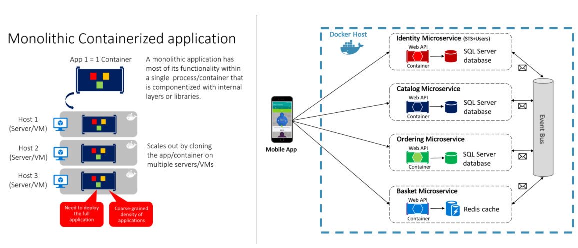


Figure 5: Monolithic Containerized application

Microservices does **not** necessarily mean containerization!

3.3 Virtualization vs Containerization

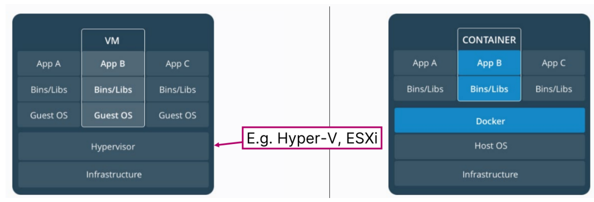


Figure 6: Virtualization vs Containerization

3.3.1 Virtualization

- = An abstraction of physical hardware turning one server into many servers
- Multiple VMs can run on the same machine
- Each VM includes a full copy of an Operating System (OS), one or more apps
- Takes a lot of space
- Can be slow to boot

3.3.2 Containerization

- = An abstraction at the app layer that packages code and dependencies together
- Multiple containers can run on the same machine, they share the OS kernel with each other, each running as isolated processes in user space.
- Takes up less space than VMs
- Boot up almost instantly

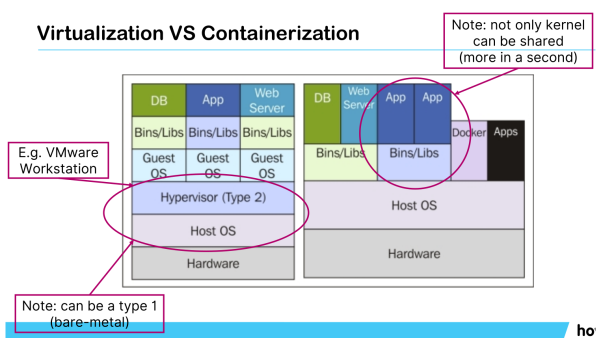


Figure 7: Schematic

3.4 Shared kernel

3.4.1 What is a kernel?

- Piece of software that offers basic functionality to the OS

- System calls: open, read, write, close, wait, exit, . . .
- A typical kernel has a few hundred system calls

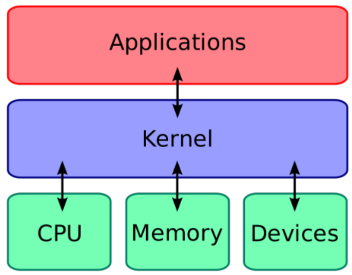


Figure 8: The kernel is the layer that communicates between hardware and applications

- Docker shares the host OS kernel
  - Host OS: Windows / MacOS / Linux
  - Shared Linux Kernel

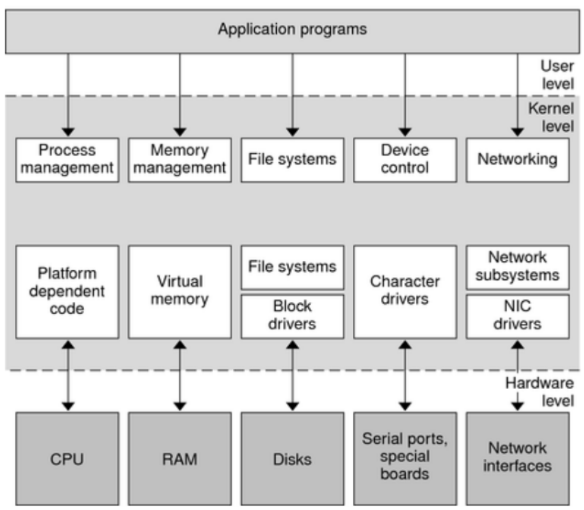


Figure 9: Kernel in detail

- The Ubuntu container requires the Linux kernel
- The Linux kernel runs in a Virtual Machine

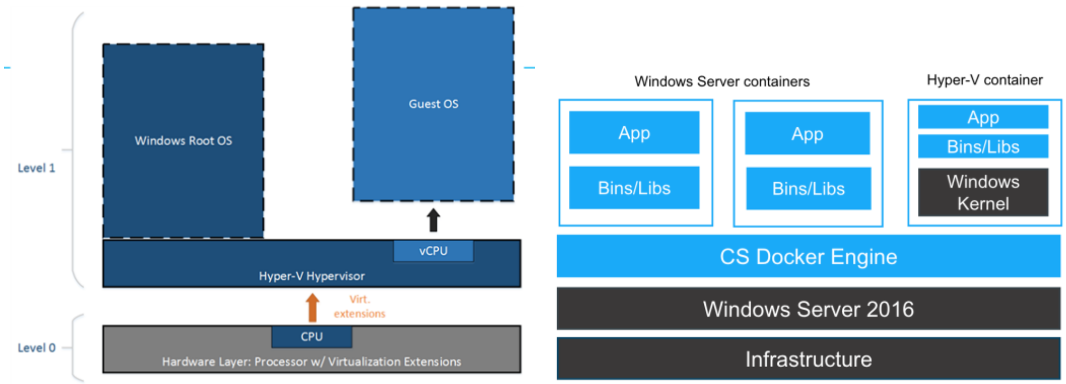


Figure 10

3.4.2 How?

Two important Linux kernel features:

- **Namespaces** are a feature of the Linux kernel that partitions kernel resources
- **cgroups** (control croups) is a Linux kernel feature that limits, accounts for, and isolates resource usage of a collection of processes

Simpler:

- Namespaces = isolating resources per process (or group of processes)
- cgroups = Limitating resource usage per process (or group of processes)

3.4.3 Namespaces

- 7 types:
  - mount, UTS, IPC, network, PID, cgroup, user
- For the process (or group of processes) it looks like there is a completely isolated set of resources

3.4.4 Containers

What is a container?

- One or more running processes (if not running anymore ⇒ container dead)
- Resources are specifically assigned to it
- The real bulding blocks: Linux kernel features
  - Namespaces
  - cgroups

3.5 Images

What is an image?

- Filesystem snapshot
- Startup command
- Layered structure **(!)**

Instance of image = container

3.5.1 Image layer

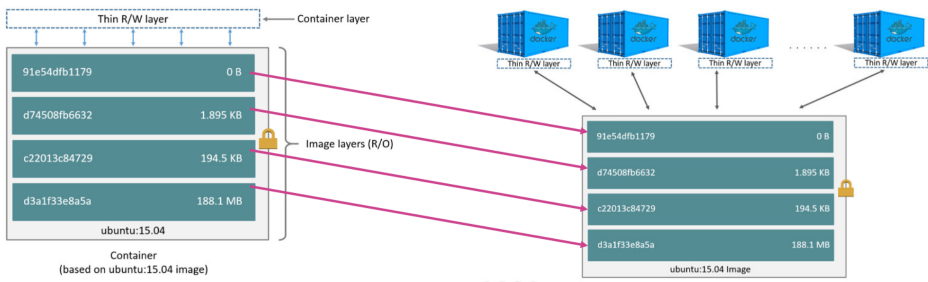


Figure 11: Image layers

- RUN, COPY, ADD
  - = new read-only layer
- Top layer = container layer
  - Writeable
- Delete container = delete container layer
  - Image will still exist
  - Peristent volumes

3.6 Docker is lightweight

- Shared kernel
- Container has no OS
- Less disk space ⇒ sharing layers
- Small community images
  - ex: Alpine Linux (small, simple, secure)
- Current Docker version is using runC (previously LXC = Linux Containers)
  - runC = tooling (written in Go) that makes it possible to create and run containers
  - runC = CLI to 'easily' access kernel features such as cgroups and namespacing
  - runC = successor of libcontainer (developed by Docker)
  - Open-sourced ⇒ better community
  - runC implements 'Open Container Initiative Runtime Specification'

– https://github.com/opencontainers/runtime-spec

Docker is ‘nothing more’ than an ecosystem about creating & running containers

3.7 Using Docker

(see slides 40-55 in 02_big_data_01_containers.pdf for basic commands)

3.7.1 View layers

With the command ‘docker history <image | container id>’ you’ll get an overview of the layers of an image.

- Every RUN, COPY, ADD adds a new read-only layer
- Make Dockerfile more efficient ⇒ create less layers

3.7.2 Make Dockerfile more efficient

Our Dockerfile, before optimalisation:

```
FROM python:3.9.1-alpine3.13
WORKDIR '/app'
RUN apk add --no-cache linux-headers g++
RUN pip install Flask # we can replace these two lines by:
RUN pip install uwsgi # RUN pip install -r requirements.txt
COPY ./ ./
RUN addgroup -S uwsgi && adduser -S uwsgi -G uwsgi
USER uwsgi
CMD ["uwsgi", "--ini", "app.ini"]
```

After optimalisation:

```
FROM python:3.9.1-alpine3.13
WORKDIR '/app'
RUN apk add --no-cache linux-headers g++
# the addgroup and adduser commands can be higher up
RUN addgroup -S uwsgi && adduser -S uwsgi -G uwsgi
# first, we copy the requirements.txt file
COPY ./requirements.txt ./
# then we install ALL packages
RUN pip install -r requirements.txt
# then we copy the remaining files
COPY ./ ./
USER uwsgi
CMD ["uwsgi", "--ini", "app.ini"]
```

3.7.3 Connecting to a database in a different container

Use ‘ip a’ to find the correct ip to use in this command:

```
docker run -p 8080:8080
    -e POSTGRES_PASSWORD=student_password
```

```
    -e POSTGRES_USER=student_user
    -e POSTGRES_DATABASE=labo
    -e POSTGRES_PORT=5432
    -e POSTGRES_HOST=ip-van-je-vm  # change this ip
    -e PORT=8080
    jouw-naam/api                  # change this
```

4 Sharding

- Index = collection of documents
- Document = data in JSON format
- Shard = A piece of an index. Index is "sharded" in blocks, a block = shard
- Primary shard = Document is primarily indexed (written) to a primary shard
- Replica shard = an asynchronous copy of the primary shard

4.1 Create index

```
{
    "settings": {
        "number_of_shards": 2,
        "number_of_replicas": 1
    }
}
```

How many shards in total: **4**

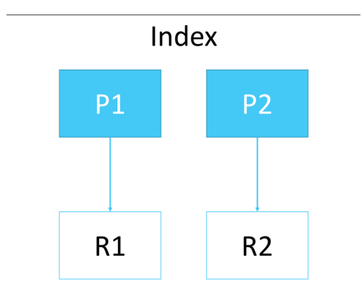


Figure 12: 4 shards: 2 primary shards with 1 replica each

4.2 Health

Health exists at shard, index and cluster level!

4.2.1 Shard health

- Green = all shards are allocated
- Yellow = all primaries are allocated but at least one replica is not
- Red = at least one primary shard is not allocated in the cluster

4.2.2 Index health

= status of the worst shard in that index

4.2.3 Cluster health

= status of the worst index in the cluster

4.3 Shard allocation

Shards states:

- Unassigned = master did not assign the shard (yet)
  - Or master is not able to assign the shard
- Initializing = master did assign the shard, creating. . .
- Started = shard is fully operational
- Relocating = shard is moving
  - Imbalance, new nodes, removed nodes, . . .

4.3.1 Unassigned

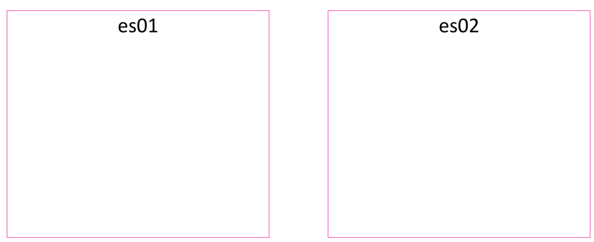


Figure 13: No shards assigned yet

4.3.2 Initializing

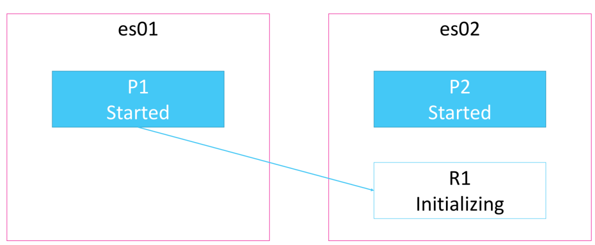


Figure 14: Creating shards

4.3.3 Started

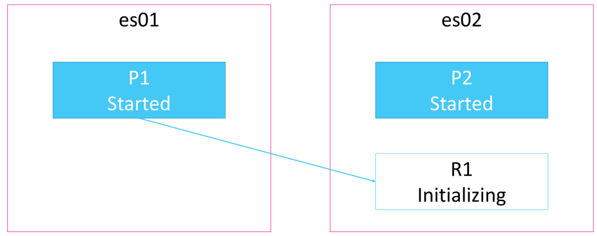


Figure 15: The primary shards have been started, replica 1 is initializing. **Cluster status = yellow**

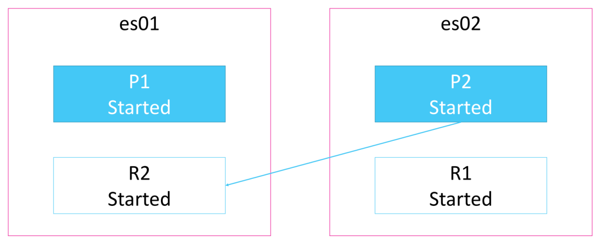


Figure 16: **Cluster status = green**

What if one of the node fails?

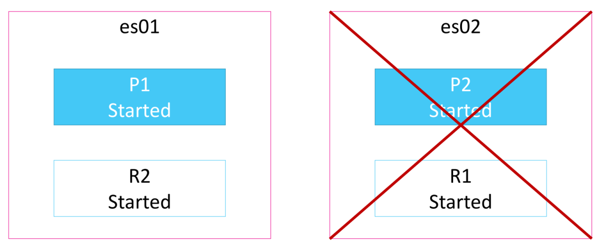


Figure 17: Situation when one node fails

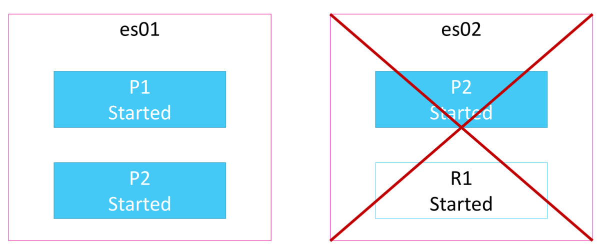


Figure 18: After some time, R2 will become a primary shard. **Cluster status = yellow**

4.3.4 Relocating

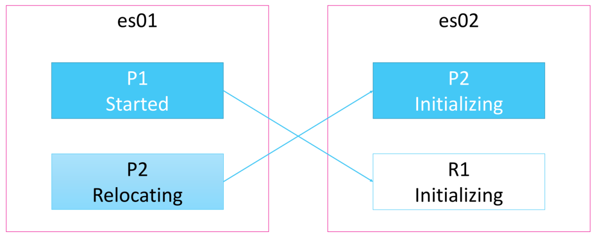


Figure 19: After es02 is restored, P2 gets relocated to its previous node

4.4 Change number of replicas

```
{
    "index": {
        "number_of_replicas": 0
    }
}
```

How many shards in total? **2**

Figure 20: 2 shards total: 2 primary shards, 0 replicas each

4.4.1 Health when one fails

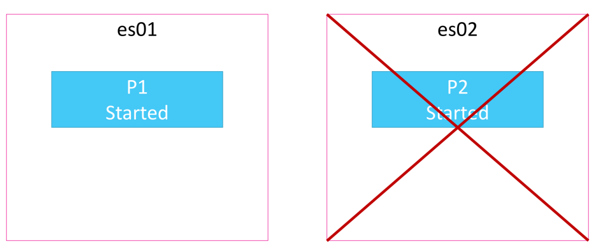


Figure 21: **Cluster status = Red**

4.5 Caveat: single node cluster

- Bootstrap checks = important settings are checked

- discovery.type=single-node
- If a node is already part of a cluster:
  - Unique node ID
  - Unique cluster ID
  - Not easy to create a new cluster

5 Linux batching + Dask

5.1 Python & data engineering/science

- Many tools, libraries (numpy, pandas)
- Not very scalable ⇒ parallellisation
- Threads/processes possible, but complex and not ideal
- What if it doesn't fit in memory?
  - To disk?
  - Possible, but complex! Some operations require everything in memory

5.2 Spark vs Dask

5.2.1 Spark

- Complex, learning curve!
- Complete 'engine', clustering
- Streaming engine
- Written in Java: uses the Java Virtual Machine (JVM) ⇒ not very accessible
- Standalone

5.2.2 Dask

- Simpler (especially if you know Python)
- Lightweight, even useful when only 1 node
- More flexible, but less performance
- Integration with other libraries
- 'The Python version of Spark'

6 TICK Stack

The TICK stack is an acronym for a platform of open source tools built to make collection, storage, graphing, and alerting on time series data incredibly easy.

The tools:

- Telegraf
- InfluxDB
- Chronograf
- Kapacitor

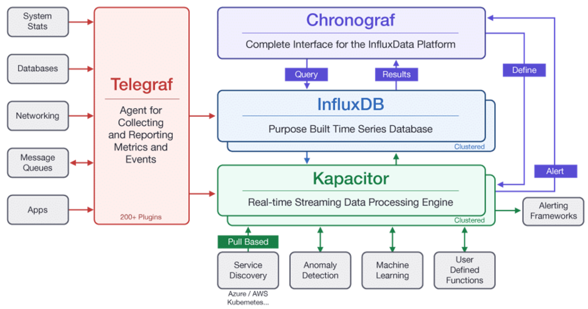


Figure 22: The components of the TICK Stack

6.1 Telegraf

- = Agent for collecting and reporting metrics and events
- Has inputs and outputs

6.2 InfluxDB

= Purpose built time series database

- Open source
- Simple HTTP API (POST, GET) with client libraries
- Somewhat similar to classic SQL, there are two versions:
  - V1: SQL & Flux: SELECT * FROM measurement WHERE tag=value
  - V2: Flux, less like SQL, better for time series data:

```
# Flux
from(bucket: "bucket")
|> range(start: v.timeRangeStart, stop: v.timeRangeStop)
|> filter(fn: (r) => r["_measurement"] == "test")
```

6.2.1 Key concepts

- Line protocol = a text-based format that provides the measurement, tag set, field set, and timestamp of a data point:

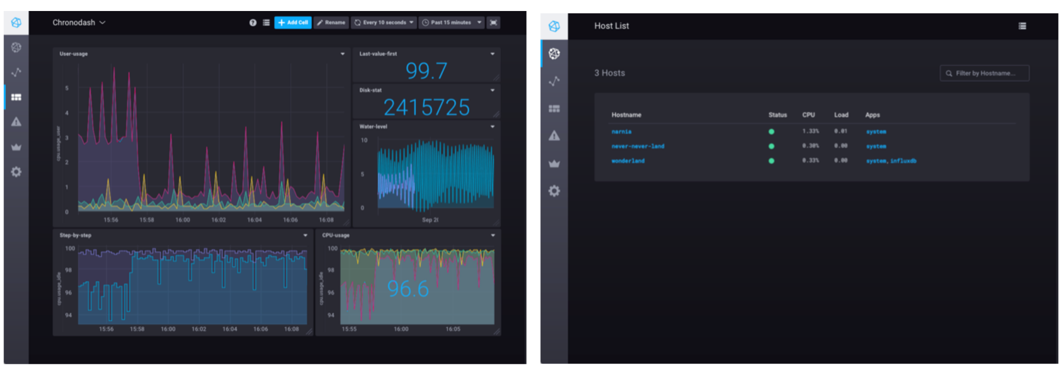
```
weather,location=us-midwest temperature=79,humidity=49 1591711854359
weather,location=us-midwest temperature=82,humidity=50 1591711787540
    |    -------------------- --------------------------  |
    |             |             |                          |
    |             |             |                          |
```

```
+-----------+--------+-+---------+-------------+---------+
|measurement|,tag_set| |field_set|             |timestamp|
+-----------+--------+-+---------+-------------+---------+
```

- Measurement = data that belongs together
- Timestamp = UNIX format
- Tags / Fields = key:value
- Tag = metadata
  - Tags are indexed
  - ‘Fields’ where you want to query on
  - Only strings!
- Field = data
  - Fields are not indexed
  - Floats, integers, strings, and booleans
- Tag set = set of tags
- Field set = set of fields

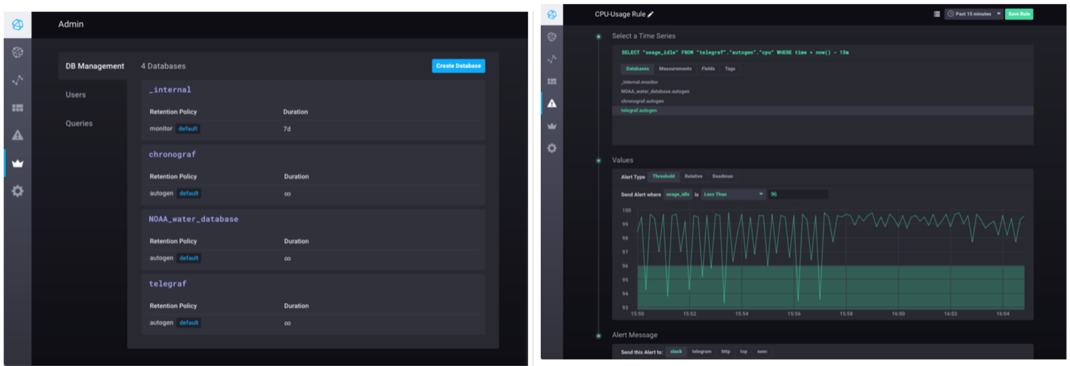
6.3 Chronograf

= A visualization tool



Data Visualization   Infrastructure Monitoring

Figure 23: Data visualization and Infrastructure Monitoring



Database Management

Alert Management

Figure 24: Database management and alert management

6.4 Kapacitor

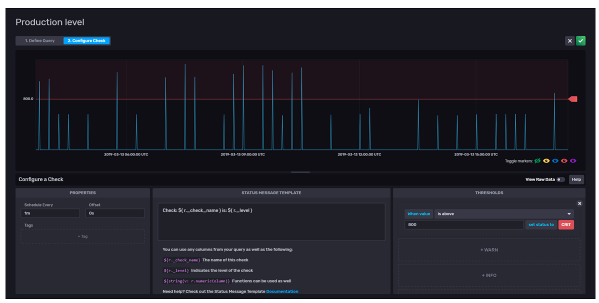


Figure 25

6.5 Deployment models

V2

- Open source version (OSS)
  - No clustering
  - No out-of-the-box replication
- Enterprise version: expensive, contact sales
- Cloud version: cheaper, usage based
- Chronograf and InfluxDB: one component
- Multi-tenant focus

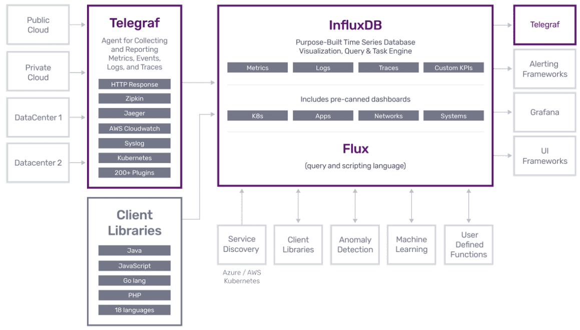
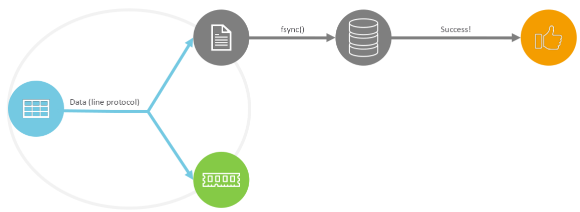


Figure 26: InfluxDB 2.0: a better graphic

6.6 Architecture of the TICK stack

6.6.1 Write Ahead Log (WAL)



- Disk optimized format (fast writes ↔ slow queries)
  - Not optimized for fast queries ⇒ memory!
- In case of a crash: replay WAL (durability ↑)
- What if we have more data than memory?
  - Out-of-memory errors (OOM)
- InfluxDB v1 vs v2 & OSS vs cloud:
  - V1 & V2 OSS: flat, simple file
  - V2 cloud: Kafka

6.6.2 Time Structured Merge Tree (TSM)

- A data structure optimized for storage and fast time-series queries
- Compressed data in columnar format
- Easy memory-mapping
- Similar to Log Structured Merge Tree (LSM)
- Field values are grouped by series key, ordered by time
- Series key = measurement, tag set and (a single) field key

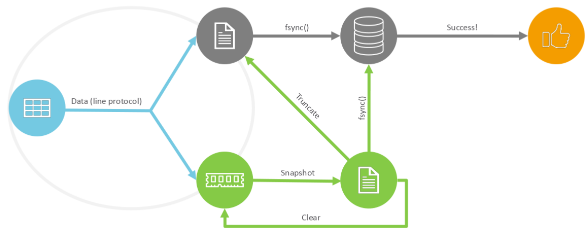


Figure 27: TSM + WAL

- Fast(er) queries: only read required series
- Compression: saved data is smaller than original (more data per node)
- Columnar: easy for memory-mapping
  - Data is cached for a limited time (solves OOM)
- What if we have many series keys (high cardinality)
  - Finding the right data will be slow!

6.6.3 Time series index (TSI)

- A data structure optimized for storage and fast query of series keys
  - TSM stores the data grouped by the series key
  - TSI stores the series keys grouped by measurement, tag and field key
- TSI answers two questions:
  - What measurements, tags and fields exists?
  - Given a measurement, tag, field, what series key exists?
- TSI stores the index in memory and on disk
  - Memory = page cache (least recently used memory)
  - Disk: writes to a WAL, compaction in the background

6.6.4 Sharding

V1:

- Directory with WAL, TSM and TSI files
- Retention policy (on database level)
- Each shard has a start and endtime
- Scalability . . . but only for InfluxDB Cloud / InfluxDB Enterprise

V2:

- Sharding in V1 has much overhead: WAL, TSM and TSI / shard
  - Too much redundant data, especially for the TSI
  - Too many writes

- Not everyone needs a retention policy
- Sharding is now implemented as a block, like in most other database systems (in OSS only 1 shard)

6.7 Pitfalls, tips & tricks

- Tips for optimal (write) performance:
  - Order your timestamps
  - Order your tags alphabetically
  - Use the right precision: seconds, milliseconds, microseconds or nanoseconds
  - Write in bulks (less fsync's)
- Duplicates: measurement, tag set & timestamp
- Tags vs. Fields
- V2 is a great product, but:
  - Documentation is far from complete
  - Bugs in client libraries, e.g. precision is neglected
  - Quick release cycle / bug fixes
- V1 vs. V2, OSS vs. Cloud vs. Enterprise