Best Practices für (verteilte) Datenbanken in Docker-Containern

Dr. Halil-Cem Gürsoy
adesso AG

@hgutwit

Über mich…

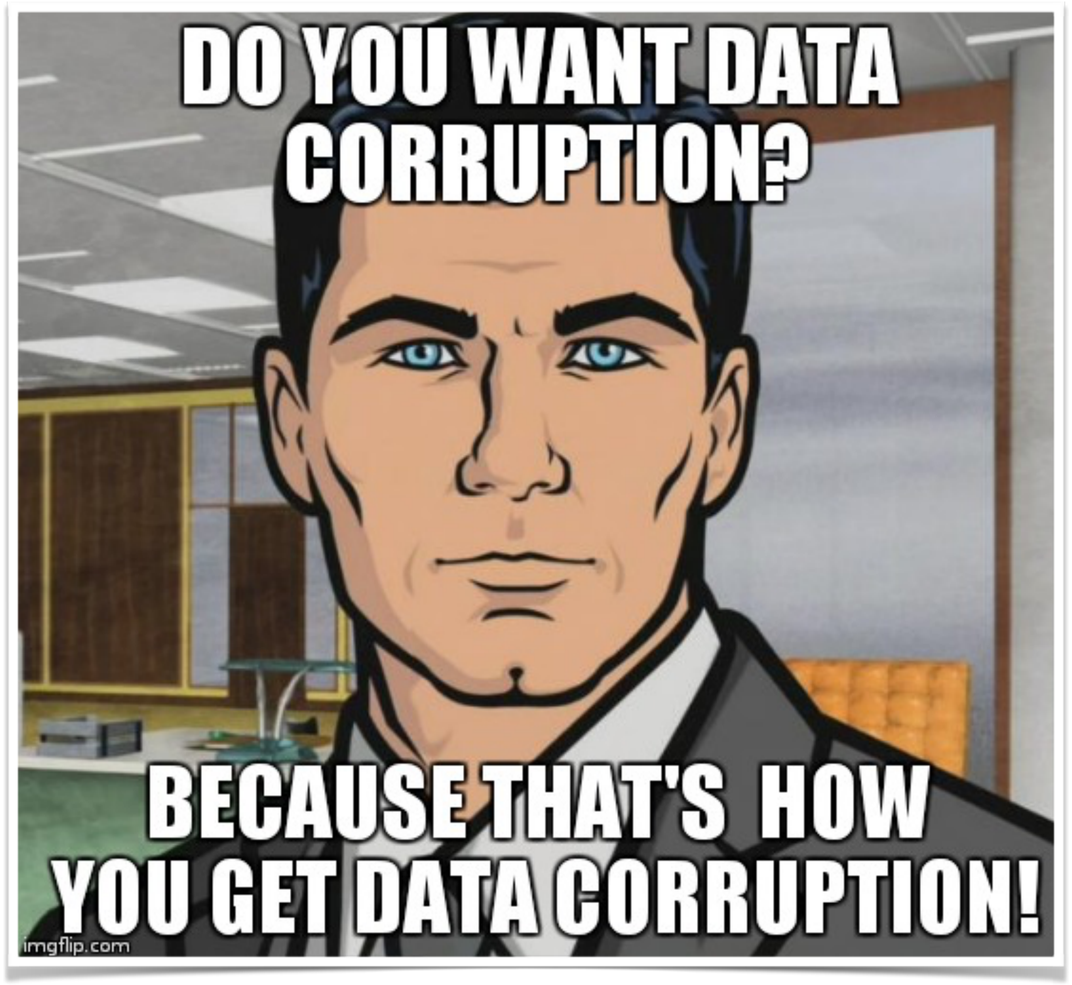
- Principal Architect @ adesso AG
- Seit > 15 Jahren Software-Entwicklung
  - davor im wissenschaftlichen Umfeld
- Verteilte Enterprise System
  - Build, Deployment & Persistenz

Eine lange Docker-Reise

- Seit Docker 0.8 (2014) produktiv mit Docker unterwegs
- Build & Testumgebungen für große agile Teams
  - Java/JEE, Testcluster, Datenbanken (NoSQL, RDBMS)
- Continuous Delivery mit Docker
  - Jenkins, Atlassian Bamboo, Docker Compose, Swarm

„Docker SHALL NOT run databases in production, by design.“

The HFT Guy in
https://thehftguy.com/2016/11/01/docker-in-production-an-history-of-failure/



- „WHY DATABASES ARE NOT FOR CONTAINERS“ -
https://myopsblog.wordpress.com/2017/02/06/why-databases-is-not-for-containers/

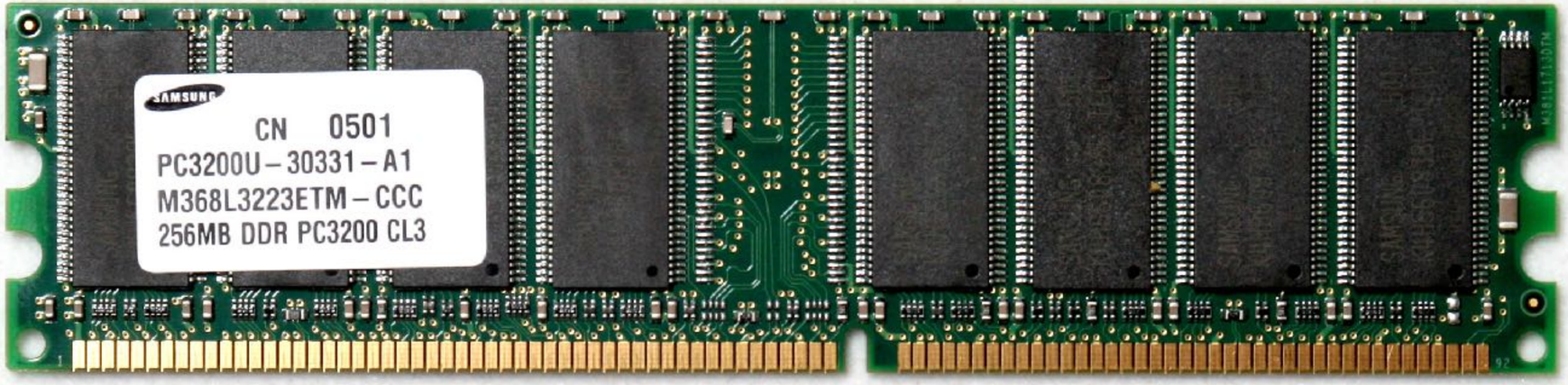
https://www.flickr.com/photos/twicepix/3902796929/

https://www.flickr.com/photos/vanmelis/14920357394

JVM

- Viele Datenbanken basieren auf Java
  - Apache Cassandra, Elasticsearch usw.
- „Funktioniert“ Java out of the Box in einem Docker Container?
  - Oracle: *„Java 8 update 131 includes a number of changes enabling better memory and processor integration between Java and Docker.“* - https://blogs.oracle.com

https://www.flickr.com/photos/bombardier/19428000/



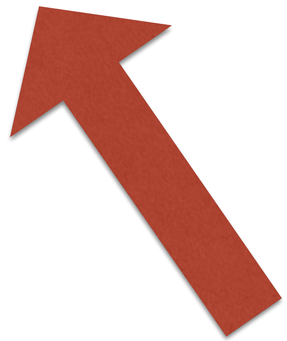
https://www.flickr.com/photos/mac_users_guide/3665683020

JVM, Memory & Docker

```
# docker run --rm store/oracle/serverjre:8 java -XX:+PrintFlagsFinal -
version |grep -i heapsize | egrep 'Initial|Max'
java version "1.8.0_131"
Java(TM) SE Runtime Environment (build 1.8.0_131-b11)
Java HotSpot(TM) 64-Bit Server VM (build 25.131-b11, mixed mode)
    uintx InitialHeapSize                                  := 33554432
    uintx MaxHeapSize                                      := 526385152
```

Java Heap

```
# docker run -m 256m --rm store/oracle/serverjre:8 java -XX:+PrintFlagsFinal
-version |grep -i heapsize | egrep 'Initial|Max'
java version "1.8.0_131"
Java(TM) SE Runtime Environment (build 1.8.0_131-b11)
Java HotSpot(TM) 64-Bit Server VM (build 25.131-b11, mixed mode)
    uintx InitialHeapSize                                  := 33554432
    uintx MaxHeapSize                                      := 526385152
```



https://www.flickr.com/photos/helico/1568566210

Java Heap

```
# docker run -m 256m --rm store/oracle/serverjre:8 java -XX:MaxRAM=256m -XX:
+PrintFlagsFinal -version |grep -i heapsize | egrep 'Initial|Max'
java version "1.8.0_131"
Java(TM) SE Runtime Environment (build 1.8.0_131-b11)
Java HotSpot(TM) 64-Bit Server VM (build 25.131-b11, mixed mode)
    uintx InitialHeapSize                                  := 8388608
    uintx MaxHeapSize                                      := 132120576
```

Java Heap

```
# docker run -m 256m --rm store/oracle/serverjre:8 java -XX:+PrintFlagsFinal
-XX:+UnlockExperimentalVMOptions -XX:+UseCGroupMemoryLimitForHeap -version |
grep -i heapsize | egrep 'Initial|Max'
java version "1.8.0_131"
Java(TM) SE Runtime Environment (build 1.8.0_131-b11)
Java HotSpot(TM) 64-Bit Server VM (build 25.131-b11, mixed mode)
    uintx InitialHeapSize                                  := 8388608
    uintx MaxHeapSize                                      := 132120576
```

Docker & Memory

- Immer Max Heap-Size (*-Xmx*) beim Starten von Java definieren!
- Immer *-XX:MaxRAM* setzen
- Alternativ
  -XX:+UnlockExperimentalVMOptions
  -XX:+UseCGroupMemoryLimitForHeap

https://www.flickr.com/photos/tim_norris/2538415572

Java & glibc

- glibc's *malloc()* reserviert 64Mb-Blöcke („Arenas“)
  - *Adress space*, u.a. Java 8 Metaspace, Threads usw.
- Kein Problem bis zur Verwendung von *mlockall()*
  - „Arenas“ werden tatsächlich gemappt
- ElasticSearch, Apache Cassandra (via JNA)
  - Optionale Konfiguration zur Swap-Vermeidung

mlockall & Docker

- Maximale Anzahl der Arenas kann begrenzt werden
- *MALLOC_ARENA_MAX* auf niedrigen Wert setzen
  - Default: 8 * CPU-Kerne
  - Ausprobieren und messen!
- Noch besser: Swap disablen statt JNA/*mlockall*

Memory… MongoDB

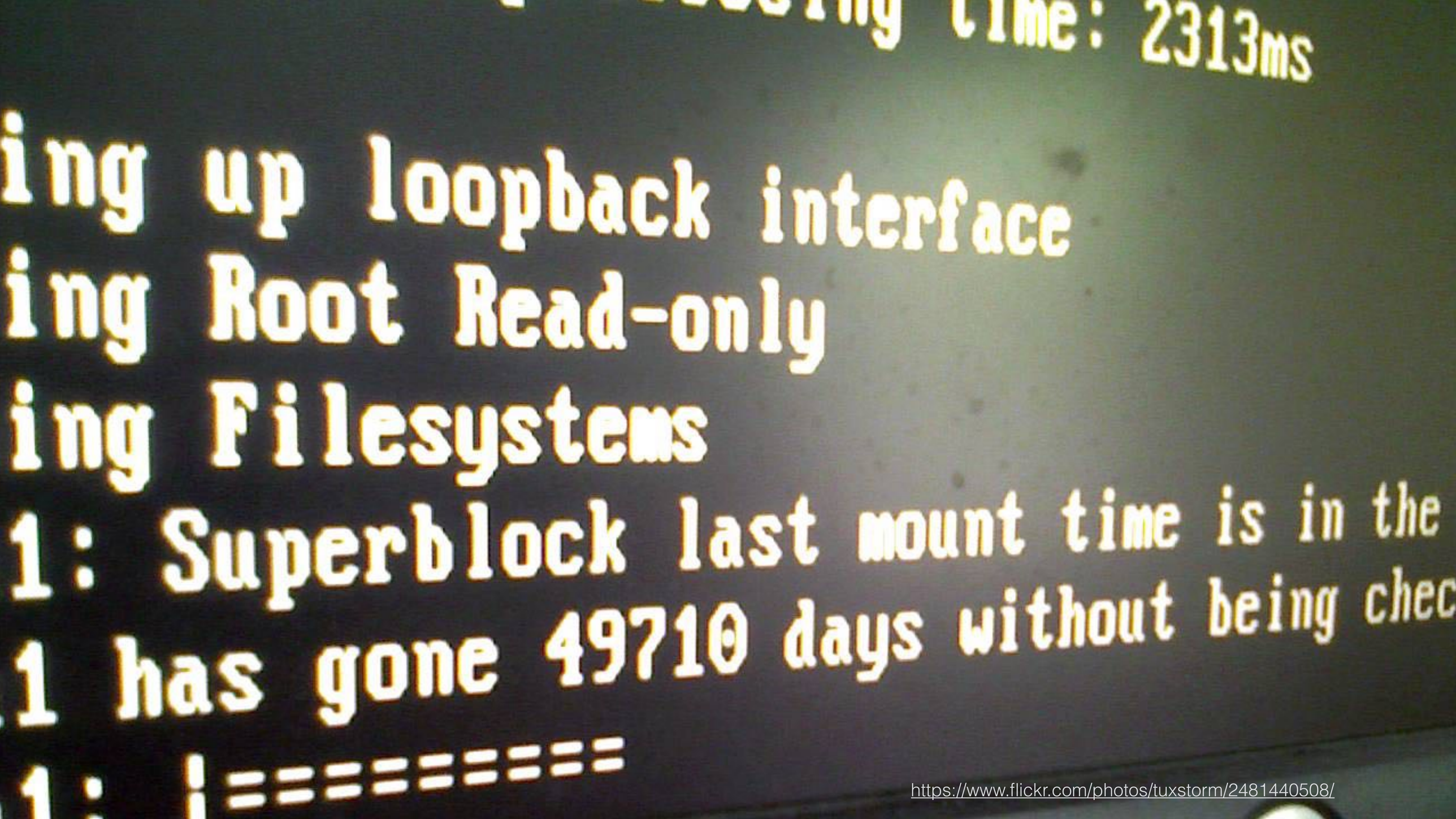
- Nicht nur Probleme mit Java-Datenbanken
- MongoDB Wired Tiger Engine berechnet internen Cache
- Mit *storage.wiredTiger.engineConfig.cacheSizeGB* manuell setzen (*-wiredTigerCacheSizeGB*)
- Faustregel: *-wiredTigerCacheSizeGB = ContainerMem / 2 - 1GB*

Memory… Oracle

- Oracle XE / Oracle DB benötigt großen Shared memory
- Default für */dev/shm* in Docker 64Mb, XE benötigt 1Gb
- Mit *--shm-size=xx* erhöhen
- *SGA_TARGET*, *PGA_AGGREGATE_TARGET* und *MEMORY_TARGET* sinnvoll setzen!
  - Achtung: ohne *MEMORY_TARGET* kann es größere Performance-Einbrüche geben!

∑ DB’s & Memory

- Container Memory Limits setzen
- JVM: MaxRAM und ggfs. MALLOC_ARENA_MAX nutzen
- Datenbank-spezifischen Memory-Konfigurationen sinnvoll und dynamisch setzen



https://www.flickr.com/photos/tuxstorm/2481440508/

Datenbank-Files

- Werden wirklich „persistente“ Datenbank-Files benötigt?
- Verteilte Datenbanken mit Replikation, wiederherstellbare Daten usw.
- Kann der Wegfall eines Knotens einfach kompensiert werden?
- „Wegwerf-Instanzen“ können Sinn machen
  - Trotzdem an Backups denken!

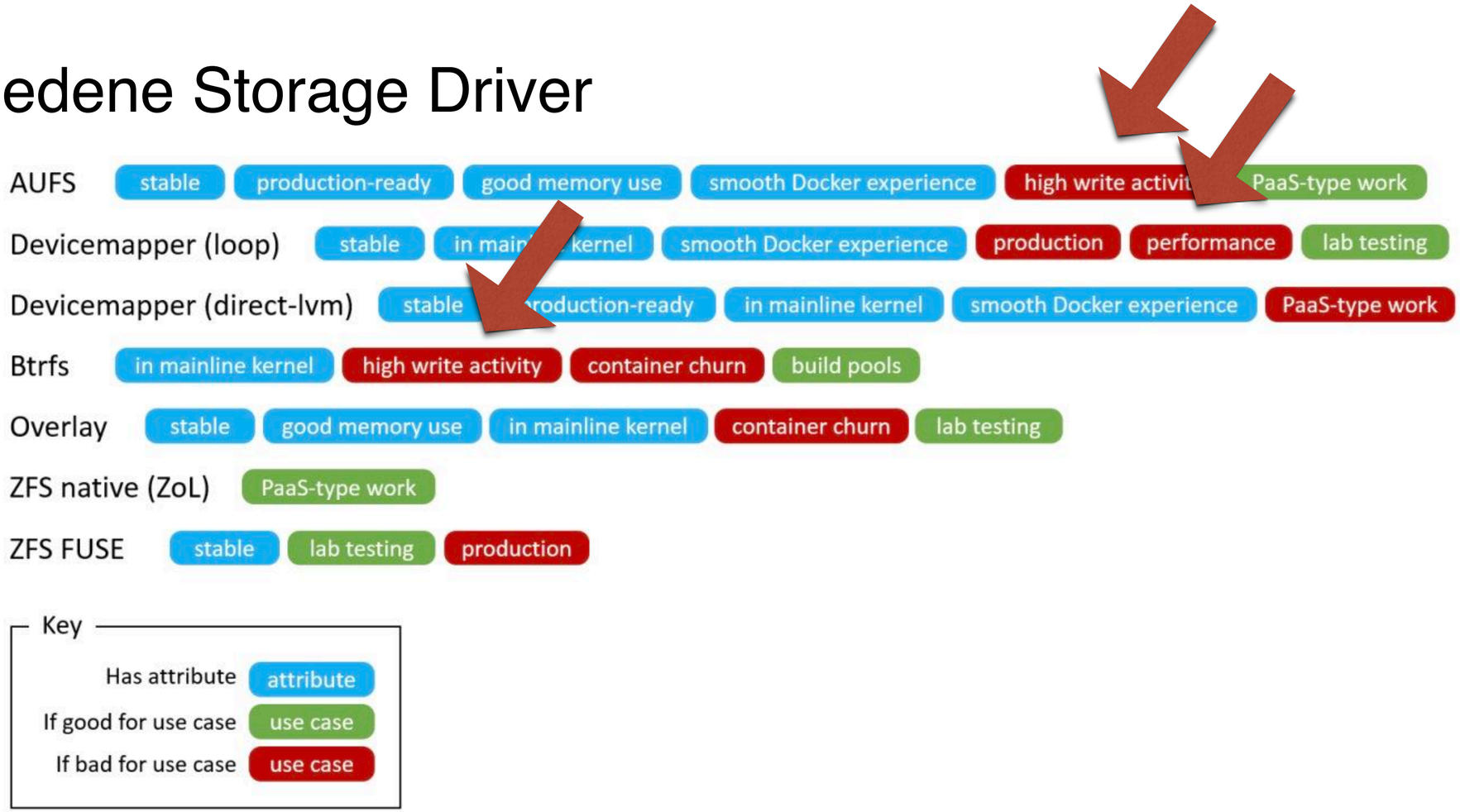
Storage Driver

- Default Storage Driver (Docker EE):
  - CentOS: devicemapper
  - Oracle Linux: devicemapper
  - RHEL: devicemapper
  - SLES: btrfs
  - Ubuntu: aufs3

https://success.docker.com/Policies/Compatibility_Matrix

In Container Files

- Verschiedene Storage Driver



aus https://docs.docker.com/engine/userguide/storagedriver/selectadriver/#future-proofing

OverlayFS

- Probleme mit *O_DIRECT*-Flag (Umgehung des System-Caches)
- Einige Datenbanken nutzen *O_DIRECT*
  - z.B. Oracle, MongoDB, MySQL (InnoDB)
- OverlayFS ist unter Docker EE nicht supported!
- OverlayFS meiden & aktuell nicht produktiv nutzen

Devicemapper

- RedHat & Derivate: „muss“ für Docker EE, aber nicht default
- Zwei Modi: *direct-lvm* & *loop-lvm*
- *loop-lvm* nicht für Produktion geeignet (eine große Datei…)
- *direct-lvm* mit LVM Thinpools
- LVM Snapshots für *Copy on Write* - Layer = Snapshot
  - schlechte Schreib-Performance

Volumes

- Dateien/Verzeichnisse in Container gemapped
- Kein „eigenes“ Filesystem, kein zwischengeschalteter Treiber
- DB-Files liegen „nativ“ im Host-Filesystem - „persistente Files“ unabhängig vom Container-Lebenszyklus
- Nachteil: Aufwändiger im Management
- „Distributed Volumes“ über NFS, AS3, …

„The best storage driver to run your production will be the one with which you and your team have the most extensive operational experience.“

–Jérôme Petazzoni, Docker Inc.

https://www.flickr.com/photos/zunami/3780220955/

Netzwerke…

- Kommunikation zw. Containern über Overlay-Network bei Deployments mit Hilfe von Docker Stack
- Overlay-Network nutzt u.a. *iptables* und *vxlan*
- Wie hoch ist der Performance-Impact bei der Nutzung von Overlay?

Overlay

- Eigene Messungen mit *iperf* zeigten keine Performance-Einbrüche
- Messen von MySQL mit *sysbench* keine messbaren Abweichungen
  - aber sehr CPU-Abhängig - bei ausgelasteter CPU Performance-Einbruch der I/O bei gesättigten Devices
  - u.a. *VXLAN* und *iptables* konkurrieren mit Prozessen um CPU
- Vorsicht: Overlay sehr sensibel bei „semi-stabilen“ Netzwerken mit höheren Raten an Package Lost / Re-Transmissions

Mehr Netzwerk…

- Interfaces in Containern mit Overlay-Netzwerk im Swarm Mode liefern mehrere IP’s
- Problem für z.B. Cassandra wenn *listen_adress* nicht gesetzt ist
  - Start-Script muss angepasst werden um „richtige“ IP zu ziehen
- Analoge Probleme auch bei Elasticsearch, MongoDB usw.

Discovery

- Wie finden sich Cluster-Mitglieder?
- „Discovery“ in Swarm Mode?
- Externes System wie *Consul* mit DB Cluster-Infos fahren
- Beim Starten der DB über Script eintragen und alle Cluster-Member auslesen
  - Tools wie *consul-template* oder *envconsul* unterstützen dabei!

Bitte beachten…

- Bei großen Swarm-Clustern dauert der Aufbau des Overlay-Netzwerkes
  - VOR dem *docker stack deploy* das Netzwerk erstellen
- Firewall?
  - Docker Swarm „overruled“ *--iptables=false* Option
  - Diskussion s. #4737

https://www.flickr.com/photos/venndiagram/5331296718

Security

- Docker-Prozesse laufen by default unter *root*
- *USER*-Anweisung in Dockerfile, *-u* bei *docker run*
- MySQL -> *mysqld_safe -user xyz*
- Andere Möglichkeit: User Namespace (*—userns-remap*)
- ‚Old school' Lösung: technische User mit festen UID/GID auf Docker Hosts bei Provisionierung anlegen und in Container nutzen

Wrapper-Scripte & *docker stop*

- Oft werden Wrapper/bash-Scripte verwendet um Datenbanken im Hintergrund zu starten
- *docker stop* -> *SIGTERM* an den „Hauptprozess“
  - Datenbank bekommt kein Signal aber Namespace wird abgeräumt = Datenbank kaputt
- besser: mit *trap* SIGTERM abfangen und DB sauber herunter fahren
- oder *exec* nutzen

http://www.flickr.com/photos/an_untrained_eye/6630719431