DevConf

www.devconf.ru

Tarantool
A no-SQL DBMS now with SQL

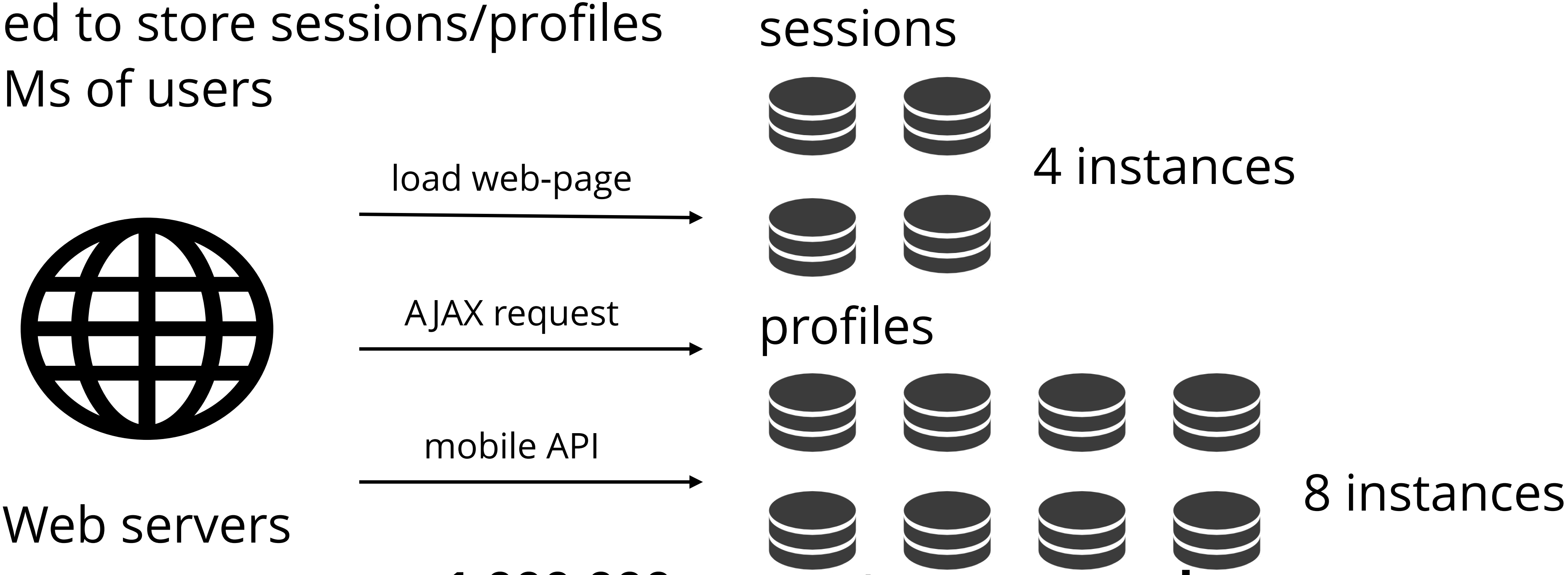
Kirill Yukhin

Agenda

- What is Tarantool?
- Performance
- Storage engines
- Scaling
- SQL
- Plans

History

- Was born @ Mail.ru group
- Used to store sessions/profiles of Ms of users



sessions

4 instances

load web-page

AJAX request

mobile API

profiles

8 instances

Web servers

> 1.000.000 requests per second

Must-have and mustn't-have

- No secondary keys, constraints etc.
- Schema-less
- Need a language. *QL is **not** must-have
    - High-speed in any sense!
    - Simple
    - Extensible
- Transactions
- Persistency
- Once again: it must be **fast**, no excuses

Tarantool: Bird's Eye View

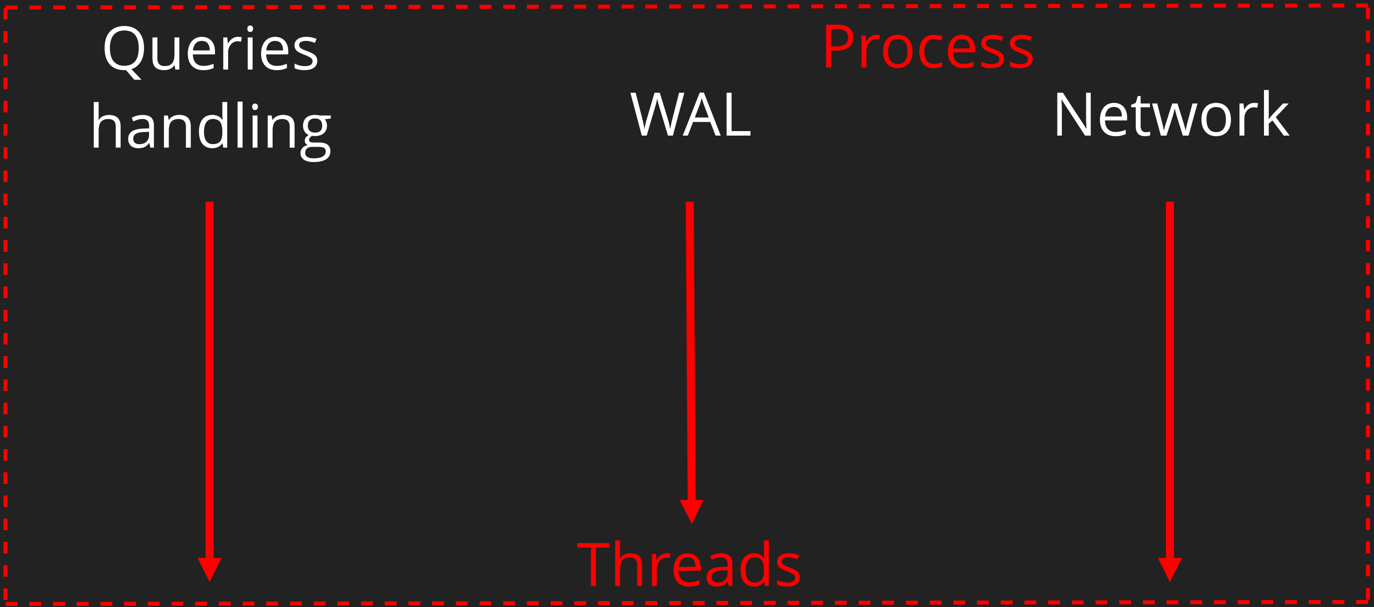
- No need for cache: It is **in-memory**
- But still DBMS: persistency and transactions
  - It regards **ACID**
- Single threaded: It is **lock-free**
- Easy: imperative language is on board: Lua
  - It **JIT**s
  - It's easy to program business
- It scales: **Replication** and sharding

DBMS + Application Server

C, Lua, SQL, Python, PHP, Go, Java, C# ...

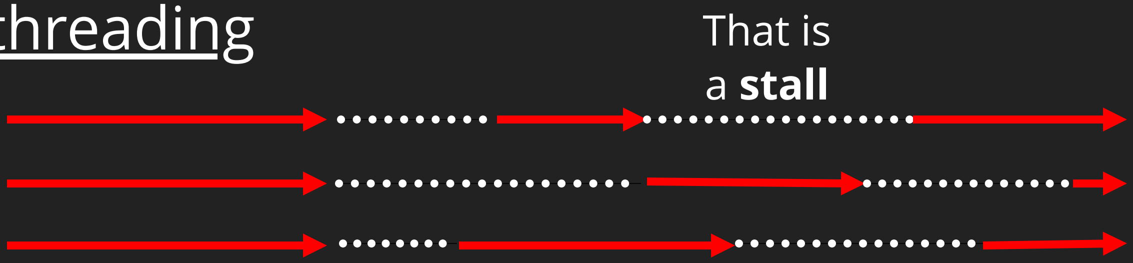
Persistent in-memory and disk storage engines

Stored procedures in **C, Lua, SQL**



Coöperative multitasking

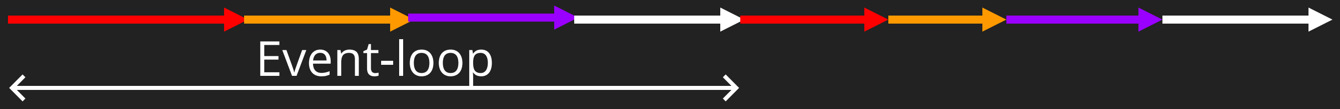
Multithreading



☹

- Losses on caches coherency support
- Losses on locks
- Losses on long operations

Fibers

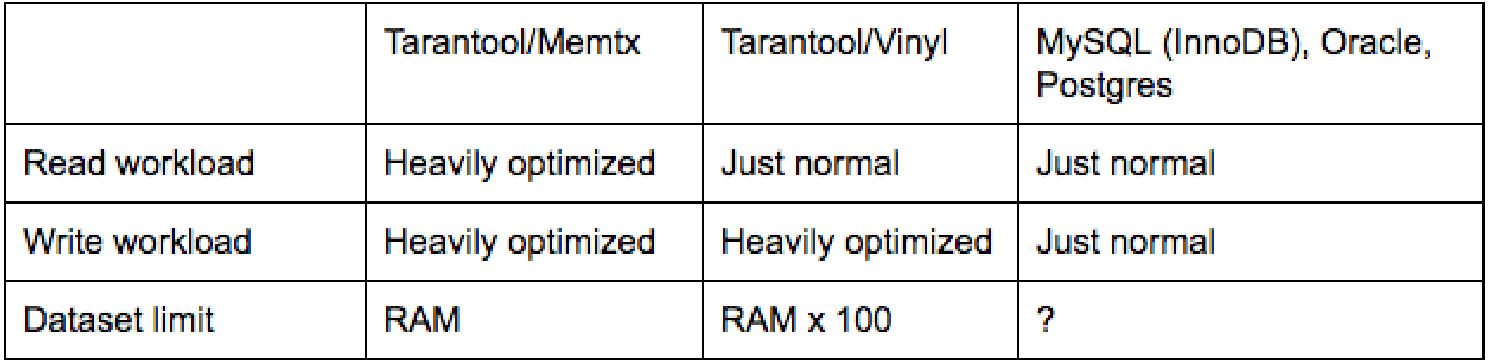


☺

- Thread is always busy
- Lock-free
- Single core - no coherency issues at all

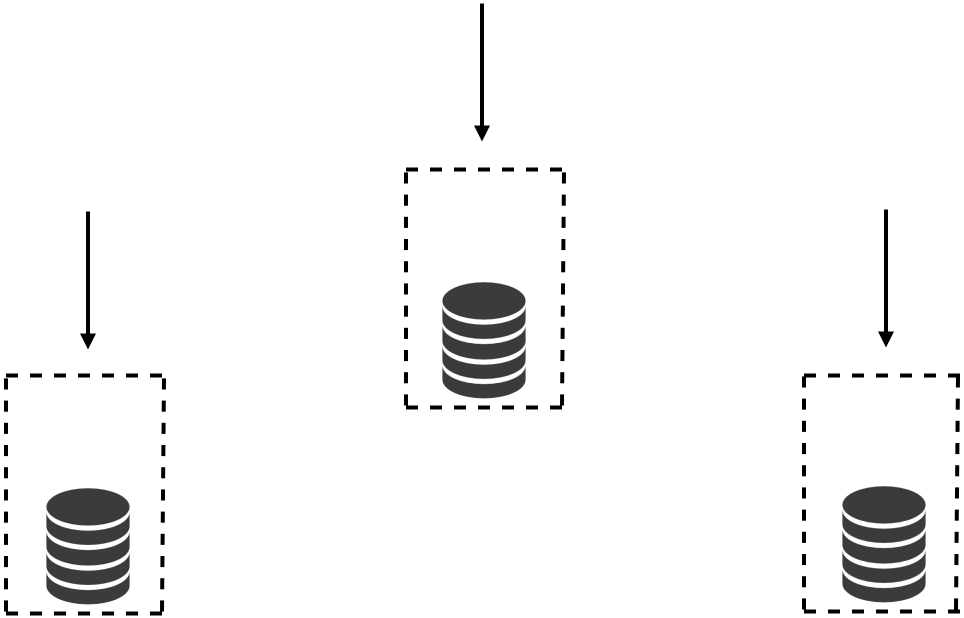
Vinyl

- In-memory is OK, but not always enough
- Write-oriented: LSM tree
- Same API as memtx
- Transactions, secondary keys



	Tarantool/Memtx	Tarantool/Vinyl	MySQL (InnoDB), Oracle, Postgres
Read workload	Heavily optimized	Just normal	Just normal
Write workload	Heavily optimized	Heavily optimized	Just normal
Dataset limit	RAM	RAM x 100	?

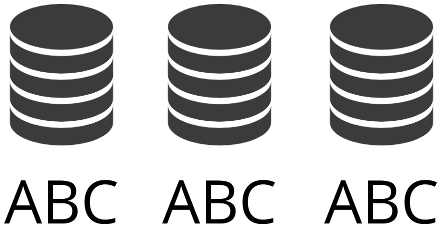
Why?



Horizontal

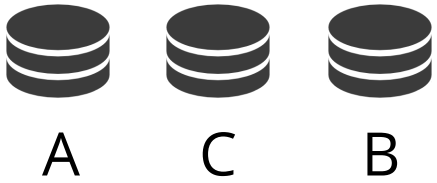
Horizontal scaling

Replication



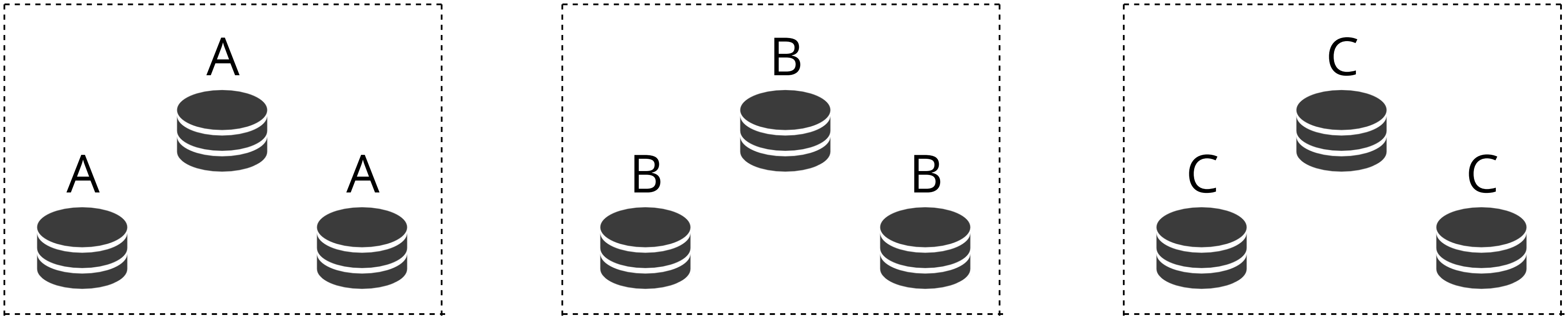
Scaling computation **and fault tolerance**

Sharding



Scaling computation **and data**

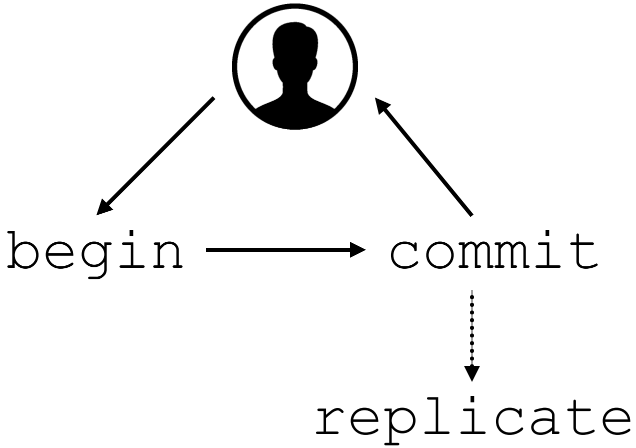
Replication and sharding



Scaling computation, **data and fault tolerance**

Replication

Asynchronous



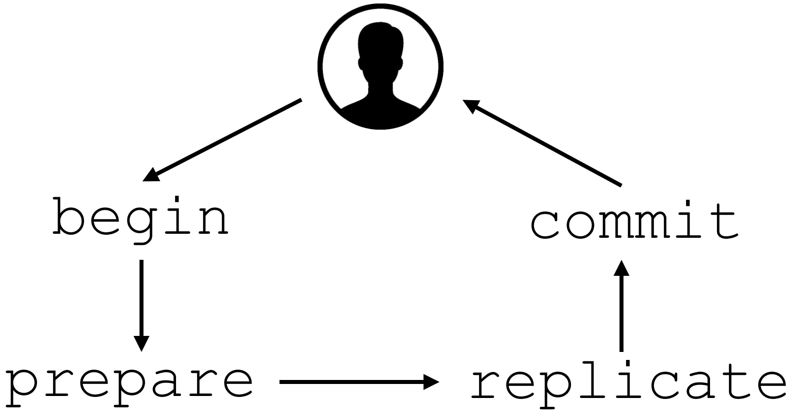
begin → commit

replicate

Commit is not waiting for replication to succeed

- + Faster
- − Replicas might lag, conflict

Synchronous



begin

prepare → replicate

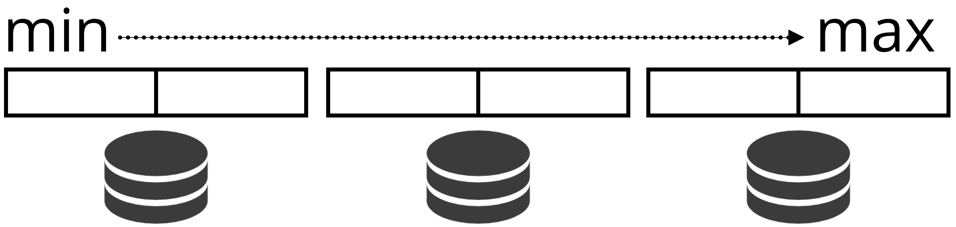
commit

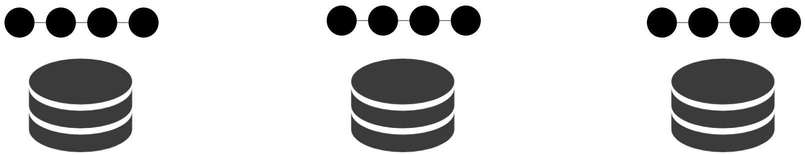
Two phase commit. To succeed, need to replicate to N nodes

- + More reliable
- − Slower, complicated protocols

Sharding

Ranges Decide where to store? hash





Found range where the key belongs -> found the node

Cockroach DB

mongoDB

Calculated hash of the key -> found the node

TARANTOOL

- + Best
- − Complicated
- − Usually useless

- + Good enough
- ? Complex resharding
- − Complex queries not fast

Resharding problem

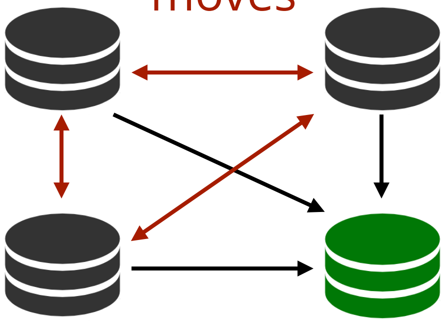
$$shard\_id(key) : key \rightarrow \{shard_1, shard_2, ..., shard_N\}$$

Change N leads to change of shard-function

$$shard\_id(key1) \neq new\_shard\_id(key)$$

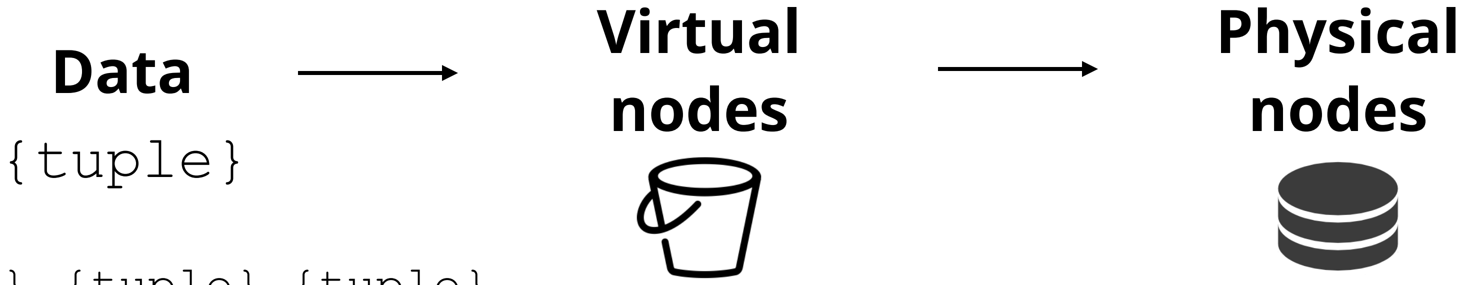
- ⊖ Need to re-calculate shard-functions for all data
- ⊖ Some data might move on one of old nodes

… but not in Tarantool land



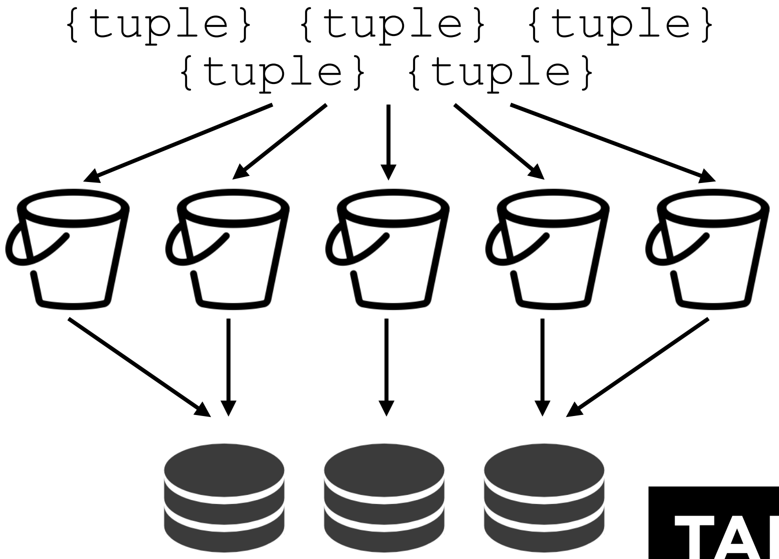
Useless data moves

Virtual sharding



Data → **Virtual nodes** → **Physical nodes**

`{tuple}`



$$shard\_id(key) = \{bucket_1, bucket_2, ..., bucket_N\}$$

# 🪣 = const >> # 🛢

Shard-function is **fixed**

TARANTOOL

Couchbase

Sharding

- Ranges
- Hashes
- Virtual buckets

Having a range or a bucket, how to find where it is stored physically?

1. Prohibit re-sharding
2. Always visit all nodes
3. Implement proxy-router!

Why SQL?

```
CREATE TABLE t1 (id INTEGER PRIMARY KEY, a INTEGER, b INTEGER, c INTEGER)
CREATE TABLE t2 (id INTEGER PRIMARY KEY, x INTEGER, y INTEGER, z INTEGER)
```

```
SQL> SELECT DISTINCT(a)
       FROM t1, t2
       WHERE t1.id = t2.id
         AND t2.y > 1;
```

Why SQL?

```
CREATE TABLE t1 (id INTEGER PRIMARY KEY, a INTEGER, b INTEGER, c INTEGER)

CREATE TABLE t2 (id INTEGER PRIMARY KEY, x INTEGER, y INTEGER, z INTEGER)
```

```
function query()
    local join = {}
    for _, v1 in box.space.t1:pairs({}, {iterator='ALL'}) do
        local v2 = box.space.t2:get(v1[1])
        if v2[3] > 1 then
            table.insert(join, {t1=v1, t2=v2})
        end
    end
    local dist = {}
    for _, v in pairs(join) do
           if dist[v['t1'][2]] == nil then
            dist[v['t1'][2]] = 1
        end
      end
    local result = {}
    for k, _ in pairs(dist) do
        table.insert(result, k)
    end
    return result end
```

SQL Features

- Trying to be subset of ANSI
- Minimum overhead of query planner
- ACID transactions, SAVEPOINTs
- left/inner/natural JOIN, UNION/EXCEPT, subqueries
- HAVING, GROUP BY, ORDER BY
- WITH RECURSIVE
- Triggers
- Views
- Constraints
- Collations

Perspectives

- Onboard sharding
- Synchronous replication
- SQL: more types, JIT, query planner

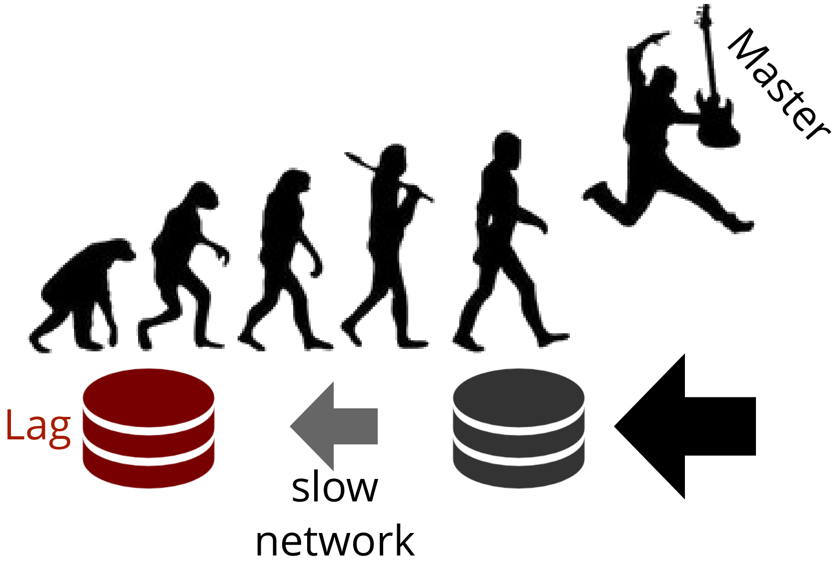
Sharding	✓ Tarantool VShard
Replication	✓ Synchronous/Asynchronous
In-memory	✓ memtx engine
Disk	✓ vinyl engine, LSM-tree
Persistency	✓ Both engines
SQL	✓ ANSI
Stored procedures	✓ Lua, C, SQL
Audit logging	✓ Yes
Connectors to DBMSes	✓ MySQL, Oracle, Memcached
Static build	✓ for Linux
GUI	✓ Cluster management
Unprecedented performance	✓ 100.000 RPS per instance - easy!

Спасибо!

https://tarantool.io

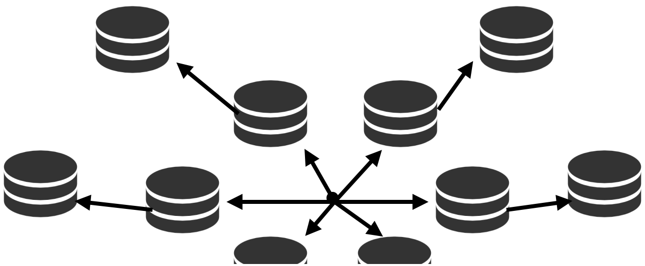
https://github.com/tarantool/tarantool

Lag of async replicas



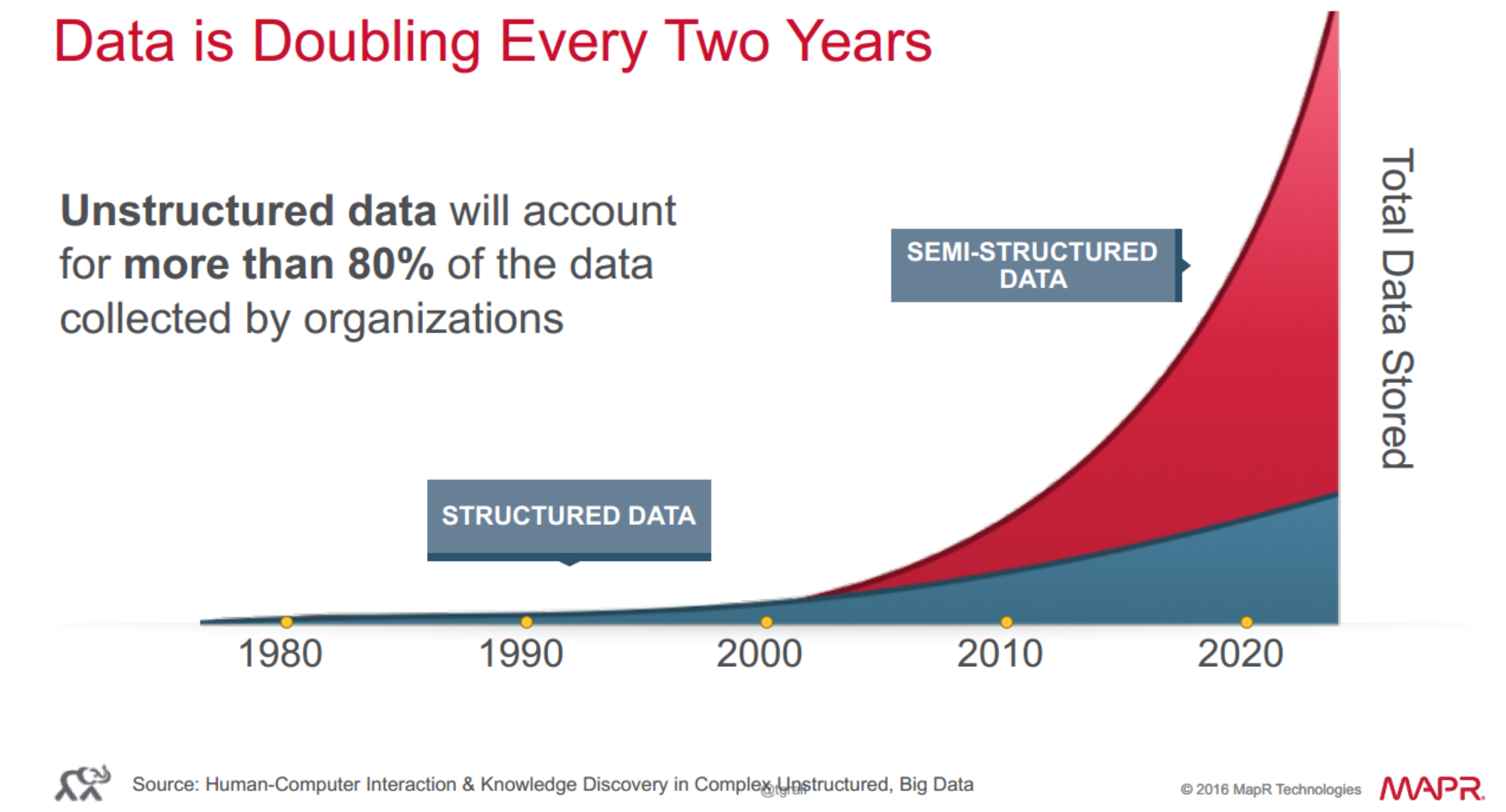
- Re-send of lost changes
- Rejoin

Complex topologies



- Support of arbitrary topologies

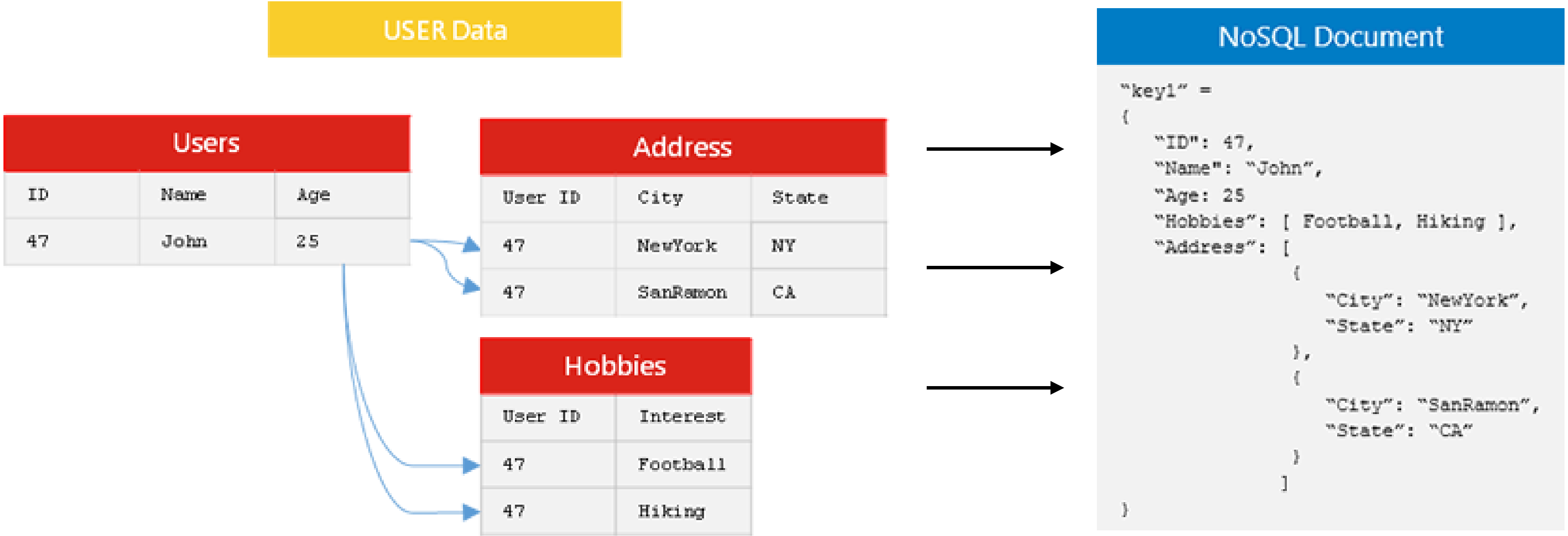
Multikey & JSON in Tarantool



Data is Doubling Every Two Years

Unstructured data will account for **more than 80%** of the data collected by organizations

Source: Human-Computer Interaction & Knowledge Discovery in Complex Unstructured, Big Data



USER Data

Users

ID	Name	Age
47	John	25

Address

User ID	City	State
47	NewYork	NY
47	SanRamon	CA

Hobbies

User ID	Interest
47	Football
47	Hiking

NoSQL Document

```
“key1” =
{
    “ID": 47,
    “Name": “John”,
    “Age: 25
    “Hobbies”: [ Football, Hiking ],
    “Address”: [
                {
                    “City”: “NewYork”,
                    “State”: “NY”
                },
                {
                    “City”: “SanRamon”,
                    “State”: “CA”
                }
               ]
}
```

Storing data in the JSON format is also a natural way to store data than in rows and columns