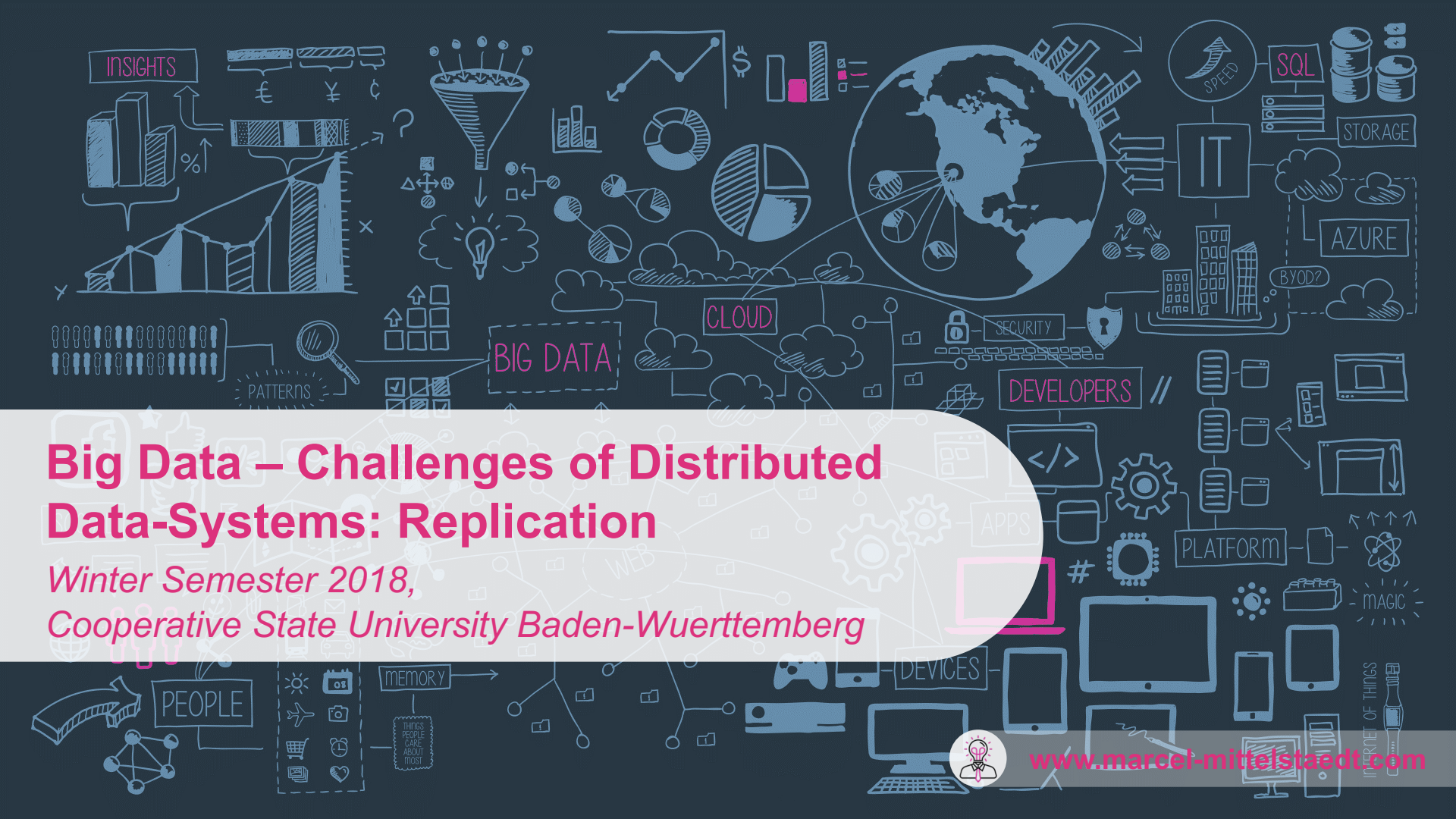


Big Data – Challenges of Distributed Data-Systems: Replication

Winter Semester 2018,
Cooperative State University Baden-Wuerttemberg

www.marcel-mittelstaedt.com

Agenda – 15.10.2018

01

Presentation and Discussion: Exercise Of Last Lecture

MapReduce, Hive, HiveQL

02

Introduction To The Challenges Of Distributed Data-Systems: Replication

Basics of Replication, Master-based, Multi-Master-based and Masterless replication, Consistency Issues and Quorums

03

HandsOn – Hive, HiveQL, Hive/HDFS Partitioning and HiveServer2

Quick Introduction To Hive, HiveQL, Hive/HDFS Partitioning and HiveQL via JDBC (HiveServer2)

04

Exercise

HDFS/Hive Partitioning, Install and Setup HiveServer2, Use Hive/HiveQL via JDBC

Schedule

	Lecture Topic	*HandsOn*
01.10.2018 16:00-19:30 Ro. 0.11	About This Lecture, Introduction to Big Data	Hadoop
08.10.2018 16:00-19:30 Ro. 0.11	(Non-)Functional Requirements Of Distributed Data-Systems, Data Models and Access	MapReduce, Hive, HiveQL
15.10.2018 16:00-19:30 Ro. 0.11	**Challenges Of Distributed Data Systems: Replication**	Hive/HDFS Partitioning, HiveServer2
22.10.2018 16:00-19:30 Ro. 0.11	Challenges Of Distributed Data Systems: Partitioning	Spark, Scala and PySpark/Jupyter
29.10.2018 16:00-19:30 Ro. 0.11	Batch and Stream Processing	MongoDB or Spark Streaming or Flink
05.11.2018 16:00-19:30 Ro. 0.11	ETL Workflow And Automation	PDI/Airflow
12.11.2018 16:00-19:30 Ro. 0.11	Work On Practical Examination	
19.11.2018 16:00-19:30 Ro. 0.11	Presentation Of Practical Examination	

Solution – Exercise 02

MapReduce in Java, Hive, HiveQL

Solution

Prerequisites:

- Download, Install and Setup Hadoop and YARN (previous lecture)
- Download, Install and Setup Apache Hive
- Start HDFS, YARN and Hive CLI

Solution

Exercise 1-4:

1. Download and unzip https://datasets.imdbws.com/name.basics.tsv.gz

```
wget https://datasets.imdbws.com/name.basics.tsv.gz
gunzip name.basics.tsv.gz
```

2. Create HDFS directory **/user/hadoop/imdb/actors/**names.tsv for file name.basics.tsv

```
hadoop fs -mkdir /user/hadoop/imdb/actors/
```

3. Create HDFS directory **/user/hadoop/imdb/actors/**names.tsv for file name.basics.tsv

```
hadoop fs -put name-basics.tsv /user/hadoop/imdb/actors/names.tsv
```


Solution

Exercise 1-4:

4. Create Hive Table `imdb_actors`:

```
hive >    CREATE EXTERNAL TABLE IF NOT EXISTS imdb_actors(
          nconst STRING,
          primary_name STRING,
          birth_year INT,
          death_year STRING,
          primary_profession STRING,
          known_for_titles STRING
          ) COMMENT 'IMDb Actors' ROW FORMAT DELIMITED FIELDS TERMINATED BY '
\t' STORED AS TEXTFILE LOCATION '/user/hadoop/imdb/actors';
```

Solution

Exercise 5:

a) How many movies are within the IMDB dataset?

```
hive >     SELECT count(*) FROM imdb_movies m WHERE m.title_type = 'movie'

499.052
```

b) Who is the oldest actor/writer/… within the dataset?

```
hive >     SELECT * FROM imdb_actors a
           WHERE a.birth_year = (SELECT MIN(birth_year) FROM imdb_actors )
```

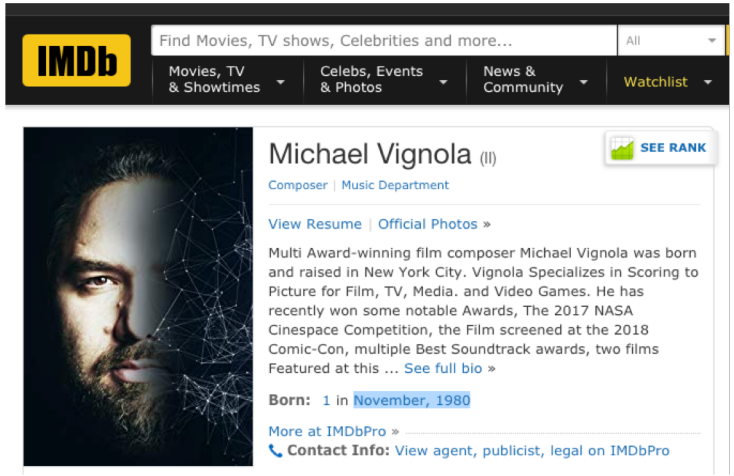
Solution

Exercise 5:

b) Who is the oldest actor/writer/… within the dataset?

a.nconst	a.primary_name	a.birth_year	a.death_year	a.primary_profession	a.known_for_titles
nm8572003	Michael Vignola	1	[NULL]	composer,music_department	tt6417824,tt4600298,tt4099244,tt6998038

Well, thats atually a bug within IMDB data:



Solution

Exercise 5:

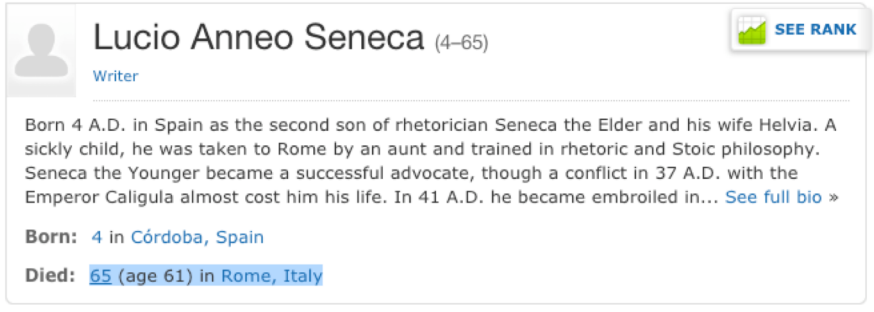
b) Who is the oldest actor/writer/… within the dataset?

Better go with:

```
hive >      SELECT * FROM imdb_actors a
            WHERE a.birth_year =
            (SELECT MIN(birth_year) FROM imdb_actors WHERE  birth_year > 1)
```

Lucio Seneca seems to be the oldest (without trash data)

a.nconst	a.primary_name	a.bi	a.death_year	
nm0194670	Célina Cély	4	[NULL]	ac
nm0784172	Lucio Anneo Seneca	4	0065	wr



Solution

Exercise 5:

c) Create a list (`m.tconst, m.original_title, m.start_year, r.average_rating, r.num_votes`) of movies which are:
- equal or newer than year 2000
- have an average rating better than 8
- have been voted more than 100.000 times

```
hive >      SELECT m.tconst, m.original_title, m.start_year, r.average_rating, r.num_votes
            FROM imdb_movies m JOIN imdb_ratings r on (m.tconst = r.tconst)
            WHERE r.average_rating > 8 and m.start_year >= 2000 and m.title_type = 'movie'
            and r.num_votes > 100000
            ORDER BY r.average_rating desc, r.num_votes DESC
```

m.tconst	m.original_title	m.start_year	r.average_rating	r.num_votes
tt0468569	The Dark Knight	2.008	9,0	1.969.110
tt0167260	The Lord of the Rings: The Return of the King	2.003	8,9	1.424.076
tt1375666	Inception	2.010	8,8	1.749.822
tt0120737	The Lord of the Rings: The Fellowship of the Ring	2.001	8,8	1.440.978
tt0167261	The Lord of the Rings: The Two Towers	2.002	8,7	1.287.434
tt0816692	Interstellar	2.014	8,6	1.213.141

Solution

Exercise 5:

d) How many movies are in list of c)?

```
hive >      SELECT count(*)
            FROM imdb_movies m JOIN imdb_ratings r on (m.tconst = r.tconst)
            WHERE r.average_rating > 8 and m.start_year >= 2000 and m.title_type = 'movie'
            and r.num_votes > 100000

86
```

Solution

Exercise 5:

e) We want to know which years have been great for cinema.
Create a list with one row per year and a related count of movies which:
- have an average rating better than 8
- have been voted more than 100.000 times
ordered descending by count of movies.

```
hive >      SELECT m.start_year, count(*)
            FROM imdb_movies m JOIN imdb_ratings r on (m.tconst = r.tconst)
            WHERE r.average_rating > 8 and m.title_type = 'movie'
            and r.num_votes > 100000
            GROUP BY m.start_year
            ORDER BY count(*) DESC
```

123 m.start_year	123 _c1
1.995	8
2.014	7
2.009	6
2.001	6
2.004	6
2.011	5
2.010	5
2.002	5
2.000	5
1.999	5
1.998	5
2.016	5
1.994	5

Introduction To The Challenges Of Distributed Data-Systems: Replication

Basics of Replication, Master-based, Multi-Master-based and Masterless replication, Consistency Issues and Quorums

Why Replication (and Partitioning)?

Availability and Redundancy: Even if some parts or nodes fail the whole data-system is able to continue working, as it can make of another replica.

Scalability and Performance: Using multiple replicas, for instance increases read performance and throughput as read queries can be distributed to any node of a replica set or even be handled concurrently by multiple nodes of the same replica set.

Reliability: Using multiple replicas stored in different data centers and locations, the data-system even continues to run during a catastrophe like an earthquake, typhoon or just a construction worker, having a bad day and cutting of the power link of one of your data-centers.

Low Latency: Keeping replicas of a data-system geographically close to users or consuming applications reduces latency (e.g. in case of a multi- national webshop, having a replica in every country).

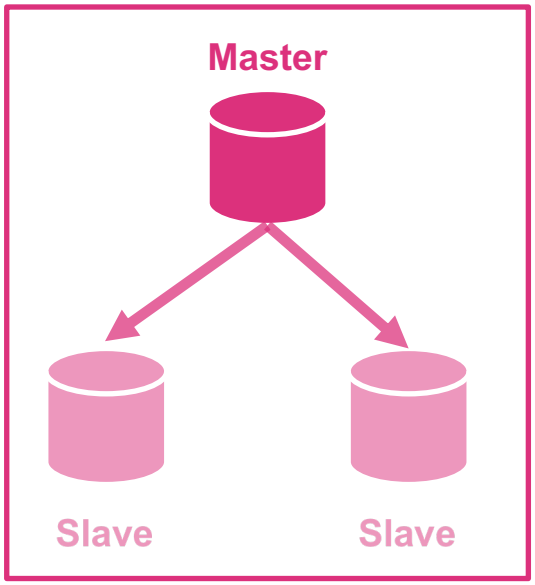
Replication vs Partitioning

	Replication	Partitioning
stores:	**copies** of the **same data** on **multiple nodes**	**subsets** (*partitions*) on **multiple nodes**
introduces:	**redundancy**	**distribution**
scalability:	**parallel IO**	**memory consumption**, certain parallel IO
availability:	**nodes** can **take** load of **failed nodes**	node **failures affect only parts** of the data

Different purposes, but usually used together

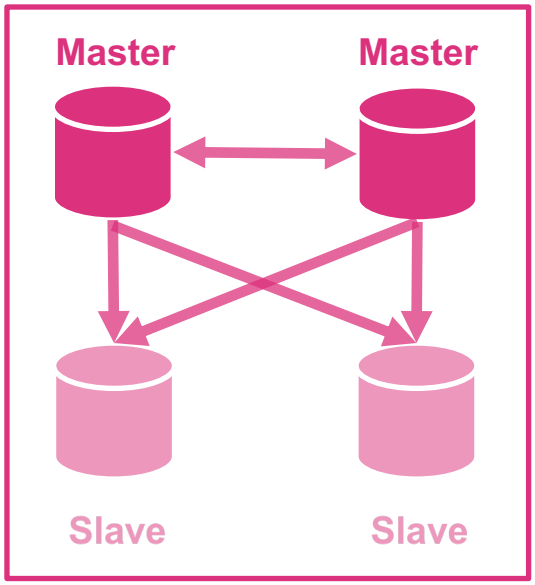
Approaches For Replication

Master-based



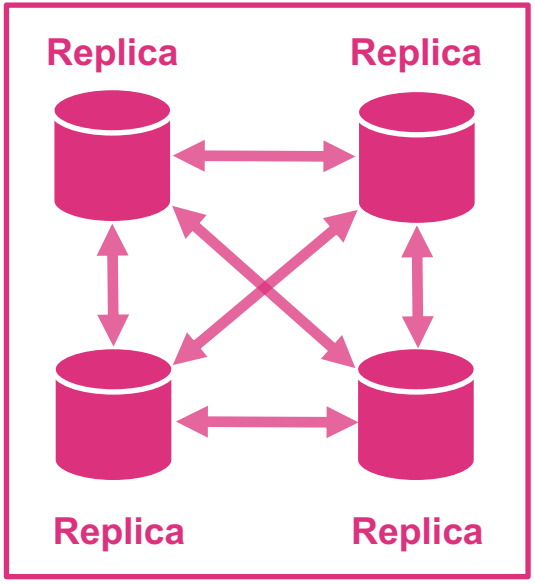
e.g. **Oracle, PostgreSQL, MySQL…**

Multi-Master-based



e.g. **CouchDB, PostgreSQL, MySQL…**

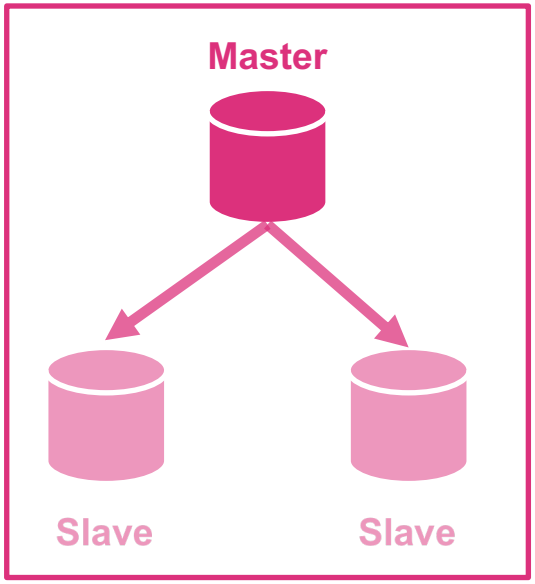
Masterless



e.g. **Cassandra, Riak, Vol demortDB, DynamoDB…**

Master Replication

Master-based



e.g. **Oracle, PostgreSQL, MySQL…**

Multi-Master-based

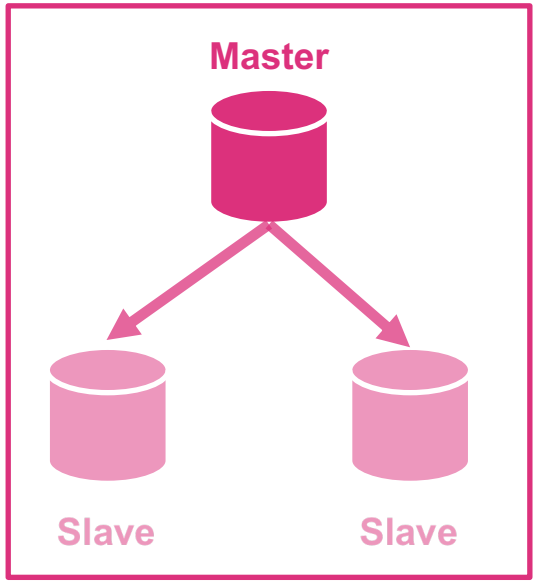
e.g. **CouchDB, PostgreSQL, MySQL…**

Masterless

e.g. **Cassandra, Riak, Vol demortDB, DynamoDB…**

Master Replication

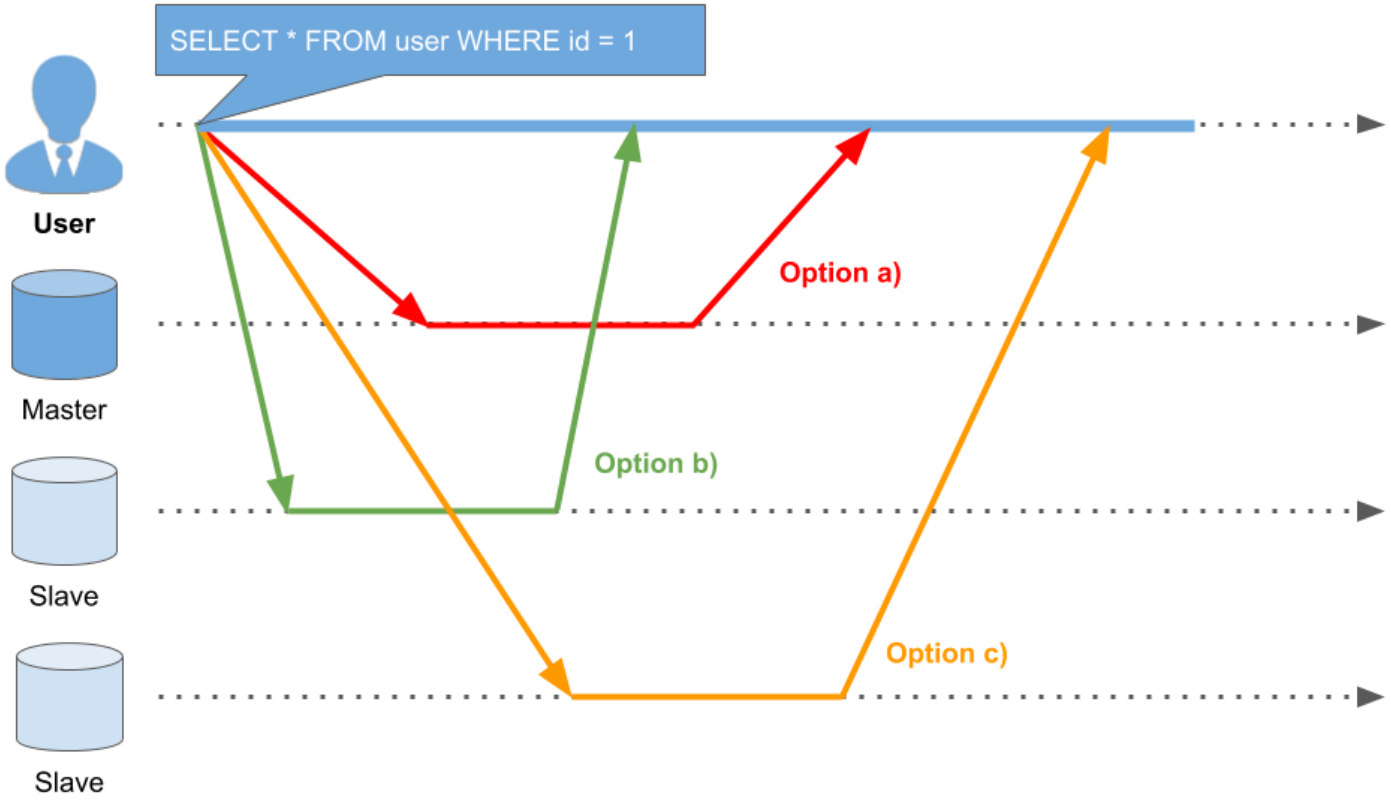
Also known as: *Primary/Secondary, Master/Slave or Single-Leader Replication*



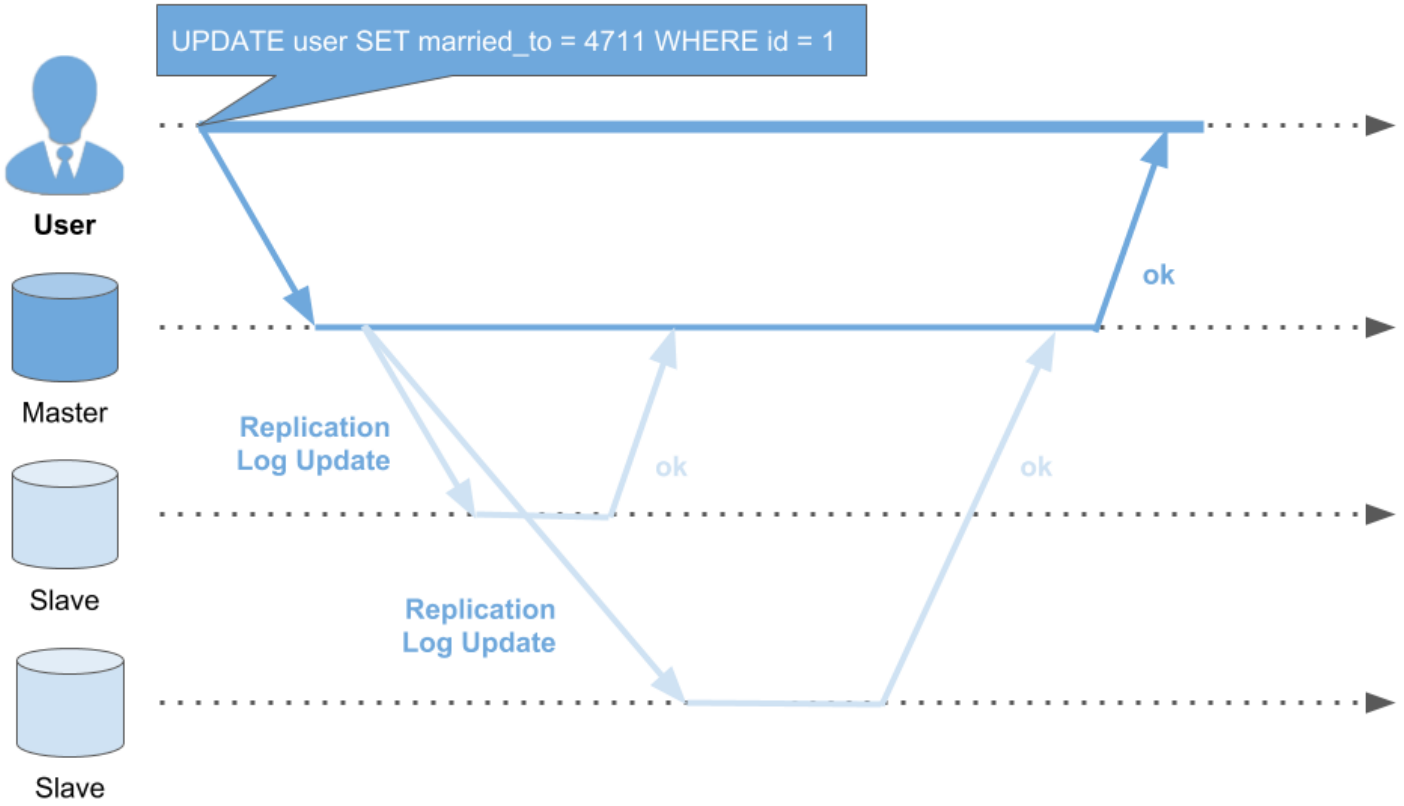
Master: (also known as *leader* or *primary*) dedicated processin g node, usually also a replica and also known as primary or leader. The master node serves read as well as write queries, is responsible for persisting changes locally and propagates changes (e.g. by using replication logs or change streams) to slave nodes.

Slave: (also known as *secondary*, *follower* or *hot-standby*) are dedicated for serving read queries only. They receive changes f rom the master node and update their local replica accordingly to and in the same order as the replica- tion log provided by the master.

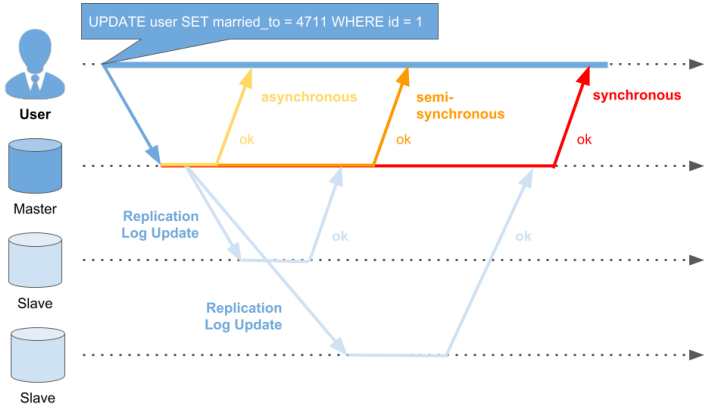
Master Replication - **Read**



Master Replication - **Write**



Master Replication – Write Synchrony



1. Synchronous:

- master waits for **all slaves** to succeed before reporting success to the user

→ all *slaves* have up-to-date copy of *master*
→ any *slave* can take over, if *master* crashes
→ Latency (like network) will directly effect write performance of the data-system, as the master waits for all slaves
- even if everything is running fine, it's slow, as slaves add additonal processing time to write queries

2. Semi-Synchronous:

- the **master waits** for **one slave** to report success before reporting success to the user

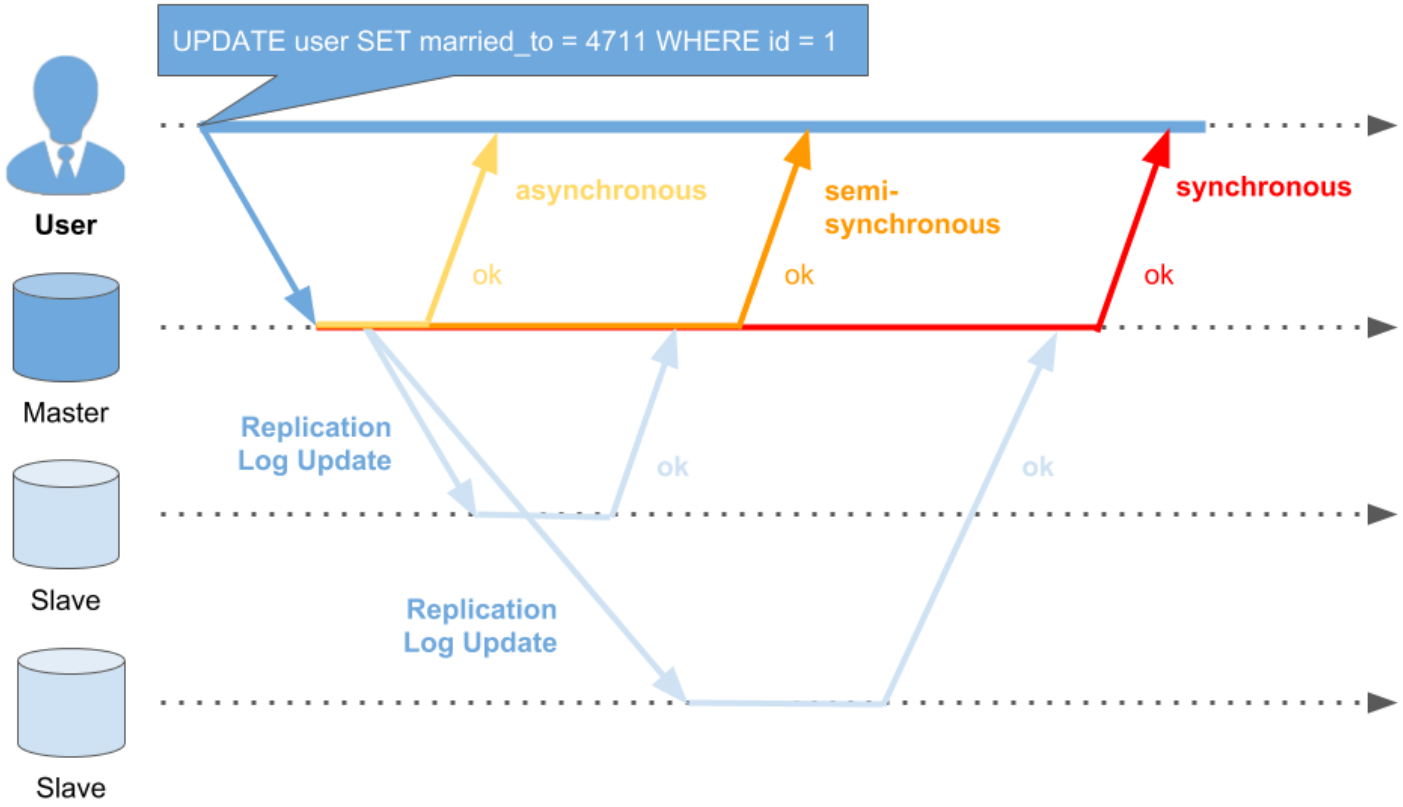
→ one slave runs synchronous to master, all other slaves asynchronous
→ If synchronous slave gets slow or crashes, another slave becomes synchronous
→ it is guaranteed, that at least two nodes store a replica which is up-to-date

3. Asynchronous:

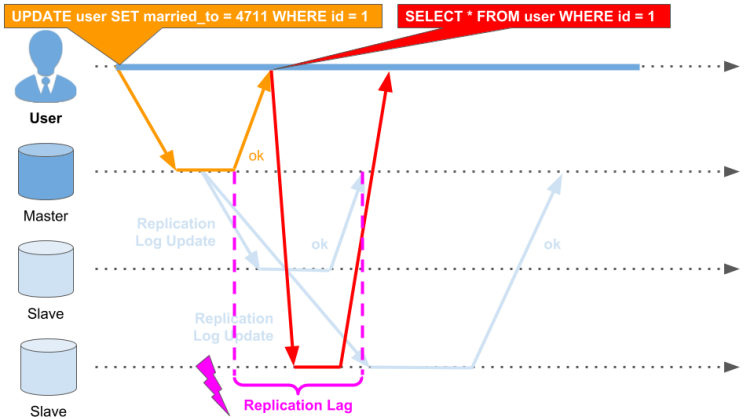
- the **master does not wait for any slaves** to report success, before reporting success to a user

→ not guaranteed to ensure durability
→ If the master crashs and cannot be re- covered, any write requests processed but not replicated to slaves, are lost
→ write-performance is incredibly fast

Master Replication - Synchrony



Master Replication – Replication Lag



Asynchronous and **semi-synchronous replication** are great for scalability of read-intensive data-systems/applications

→ Distribute and scale read requests by „just“ adding more slaves

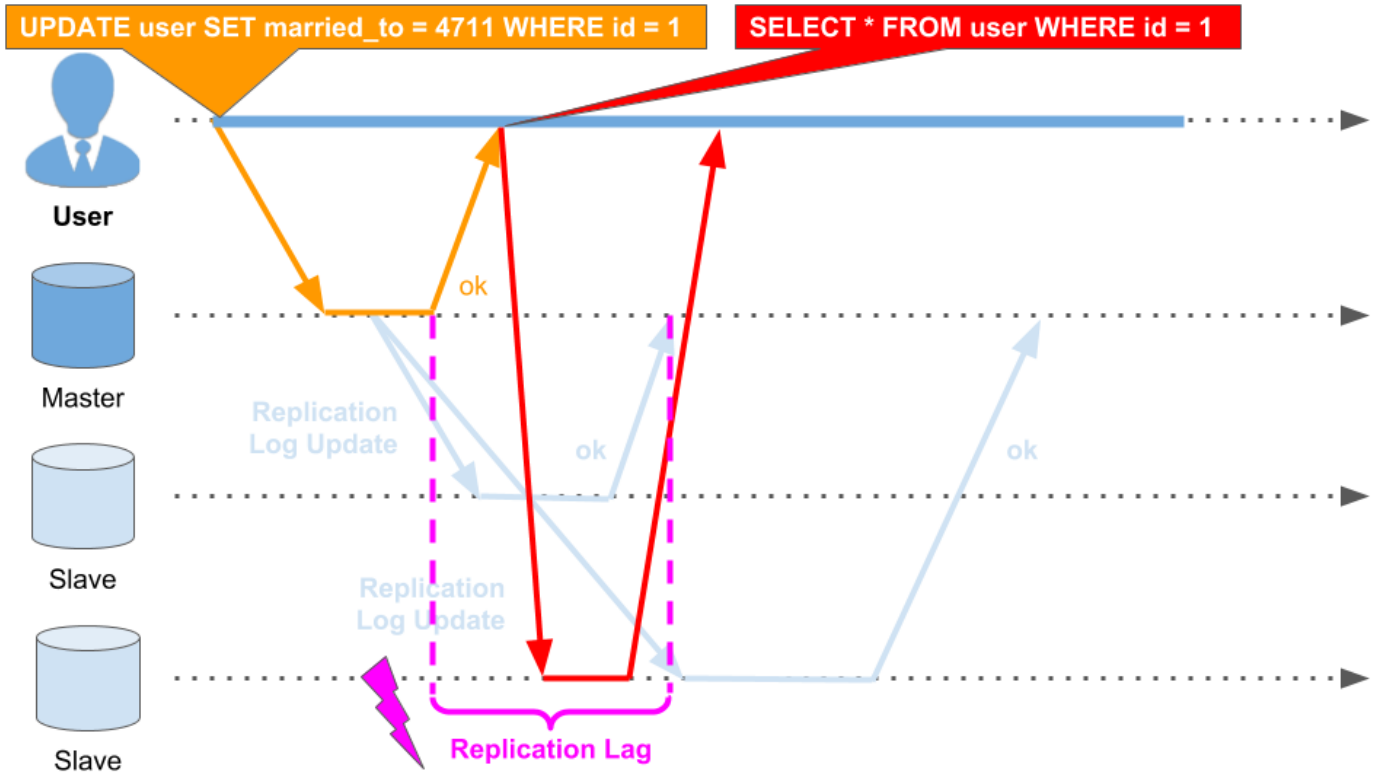
→ Data locality = put slaves geographically close to user/client-applications

→ But: **vulerable to replication lag** (timeframe when master a nd slaves ar inconsistent, as the slaves have not yet process ed the most recent write requests)

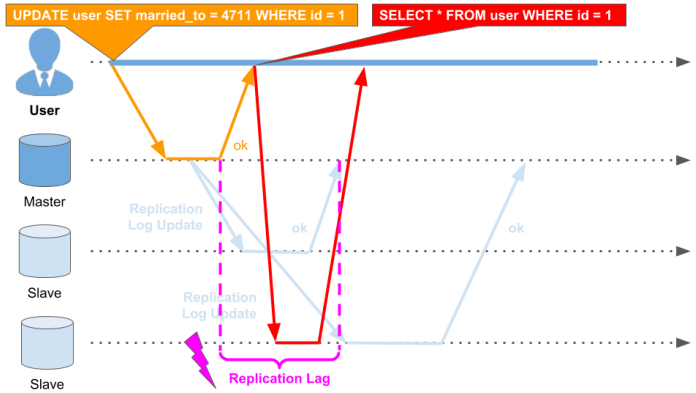
→ **replication lag** = **temporary incosistency** (usually just fractions of seconds)

- Examples of replication lag causing a data-system to be inconsistent:
  - ***Reading Your Own Writes***
  - **Monotonic Reads**

Master Replication – Replication Lag



Master Replication – Read-Your-Own-Writes



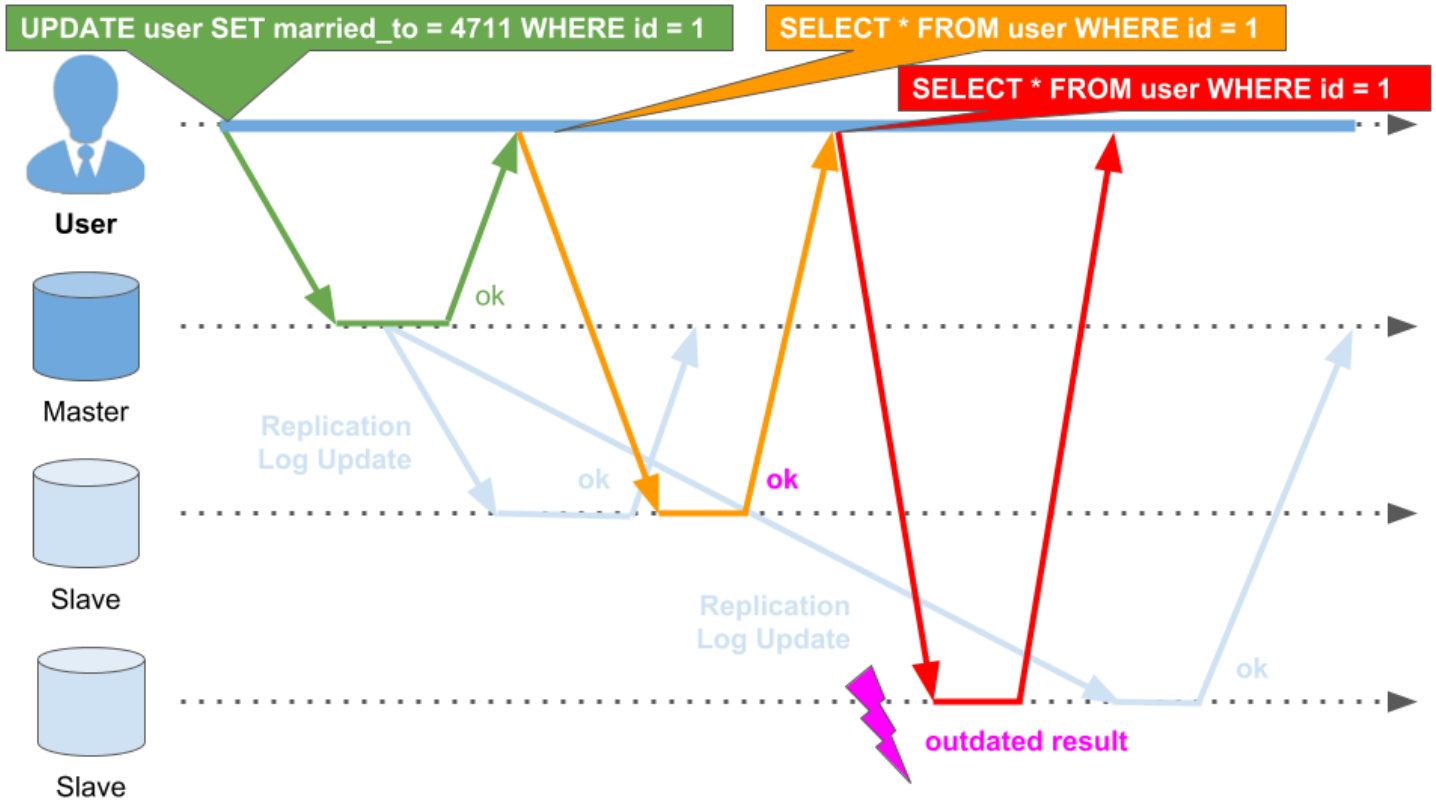
Read-Your-Writes Consistency:
ensures that any write requests submitted by a user will directly be seen on any further read requests (by the same user, guaranteeing a user his write request has been processed successfully.

Approaches for achieving Read-Your-Writes consistency:

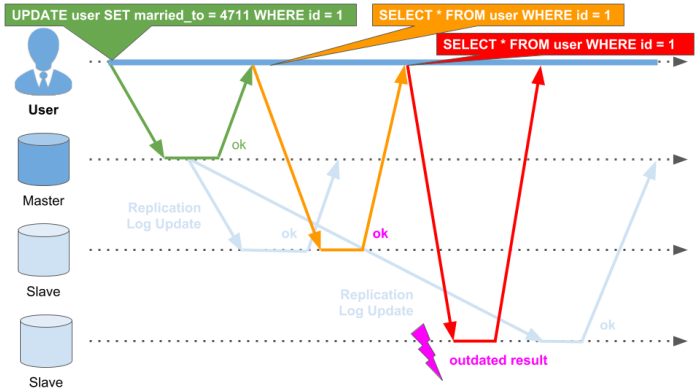
- **Read data**, a **user may have modified**, **from the master**, otherwise make use of a slave. E.g. Xin g/LinkedIn/Facebook Profile profiles → those profiles can only be edited by the user itself.
- The previous approach does not work very well for data-systems where a user is able to edit almost anything of a data-system, as this would cause any read request to end up oon the master.
  Time or replication state could be a valuable criteria to decide wether use of he master or a slave is appropriate, e.g. if an update request is newer than X seconds or less than the replication lag**, read from master** node **otherwise** make use of a **slave node**.

www.marcel-mittelstaedt.com

Master Replication – **Monotonic Reads**



Master Replication – Monotonic Reads



Monotonic Reads Consistency:

A user is reading from 2 different slaves, one with less and one with more replication lag, whereas the last one is still missing a replication update. Monotonic Reads Consistency ensures that this kind of divergence does not happen - a user will not read less recent data after reading new data .

Approaches for achieving Monotonic Reads consistency:

- a user needs to always read from same replica (master or slave) within a session
- (different user can read from different replica nodes)
- E.g. determine replica by *user id* `modulo` the *number of replica nodes*

Master Replication – **Adding New Slave Nodes**

1. Create Snapshot

- take a snapshot of a master or slave node
- usually achieved by tools of a data system (e.g. *Ops Manager* in case of *MongoDB*) or Ops storage system tool like *LVM* or *Amazon EBS* in case of *EC2* instances

→ last one requires stop of data-system

→ plain snapshots require a lot of space as they contain indices, storage padding and fragmentation

2. Copy Snapshot

- copy Snapshot to new replica slave node
- for instance automatically by using tools of the data-system or in a lot of cases even rsync or cp is used and recommended (e.g. MySQL).

Master Replication – Adding New Slave Nodes

3. Process Replication Log

- the new slave node connects to the master node and processes all dataset changes happened since the snapshot used, was created.
→ this is usually done by using a *replication log* of the *master node*
→ oldest entry within the *replication log* needs to be less recent than or equal to the creation time of the snapshot
→ a *slave* which is too far behind the replication log (*stale slave*) would lead to data loss

4. Go-Live

- as soon as the new slave successfully processed the replication log and is up-to-date, it can start working again like any slave node
- serve read requests and processing changes from the master as they happen

Master Replication – Outages Of Nodes

Slave Outage:

- slave nodes make use of **repliation logs** to stay in-sync with the master
- Those replication logs and processing state (usually
  - timestamp,
  - number or
  - position of event within replication log)

stored on the slave, e.g as

- ***OpLog*** collection *local.oplog.rs* in case of MongoDB)
- **relay logs** in case of MySQL relay logs

- If a slave node crashes and gets back up again or recovers from an network issue, it:
  - can just start right away using the **replication log** where it stopped
  - **connect to the master** and **request all dataset changes** that have happened during the time of outage
- As soon as the new slave successfully processed all missing data changes and is up-to-date, it works like before

Master Replication – Outages Of Nodes

Master Outage:

1. Determining Master Failure

- e.g. done by defining and using timeouts (*heartbeat*)

→ if a node does not respon within a defined timeframe, it is supposed to be dead

2. Evaluating New Master

- several approaches, most common: choose new master by the majority of the remaining nodes or a controller node (e.g. an *Arbiter* node in case of MongoDB)
- best candidate: node containing most-recent updates of old master node
- vulnerable to *split-brain scenario* (if node failure was caused by network outage)

3. Reconfiguration

- as soon as *new master* node is elected, all clients need to send their write requests to the *new master*
- all *slave nodes* need to receive the replication log from the new master as well

Master Replication – Replication Logs

Statement Based:

- most-simple approach
- master node processes each query and afterwards adds them to the replication log (e.g. `INSERT`, `UPDATE`, `DELETE`…)
- statements are later executed by slave nodes
- Pitfalls:
  - How to handle **non-deterministic** functions within a write request (like `RAND()`, `USER()` or `NOW()` used in an SQL query) or external functions like StoredProcedures, Triggers or user-defined functions?
    → The master node would need to **replace** those **non-deterministic functions** with deterministic values (like the return value of the called function).
  - What if statements depend on other data, like in an `UPDATE` ... `WHERE` ... statement.
    → This requires all statements to be executed **within the exact same order**, oth- erwise they will end up in different results.

- For instance previously used by MySQL and still supported in a mixed-approach (with row-based replication)

www.marcel-mittelstaedt.com

Master Replication – Replication Logs

WAL *(Write-Ahead-Log)*:

- the master logs all IO data changes to a **WAL** (e.g. (re-)writes of disk blocks, appends t o files, etc.), writes the WAL to disk and sends it to all slaves
  → when a slave processes this log file, it builds an exact same copy of the data structures as found at the master.
- **significantly reduces IO** (disk writes) → only the log file needs to be flushed to disk to guarantee that a transaction is committed, rather than every state-ment or data file changed by the transaction
- Highly dependent on storage engine (which byte has changed within which disk block)
  - → different versions of a data-system or storage engine on master and slave nodes i mpossible
  - → increases complexity of maintenance tasks, especially rolling-upgrades of single n odes within a data-system are not possible any more
  - → makes downtimes inevitable

- For instance used by ArangoDB or PostgreSQL

Master Replication – Replication Logs

Logical Log Replication

- logical row-based replication (decoupled from storage engine)
- Replication log contains records describing changes of a dataset in a row-based way, e.g.:
  - **`INSERT`**: The log contains one record with all values for each inserted row.
  - **`UPDATE`**: The log contains one record for each updated row as well as all new values and an information to uniquely identify the updated row (e.g. primary key).
  - **`DELETE`**: The log contains one record for each row to be deleted as well as inform ation to uniquely identify the row to be deleted (e.g. primary key).
- For instance used by MongoDB

Master Replication – Replication Logs

Update on MongoDB collection

```
$ use test
 switched to db test
$ db.user.insert({user_id:1})
$ db.user.update({user_id:1}, {$set : {married_to:4711}})
```

Resulting Replication Log

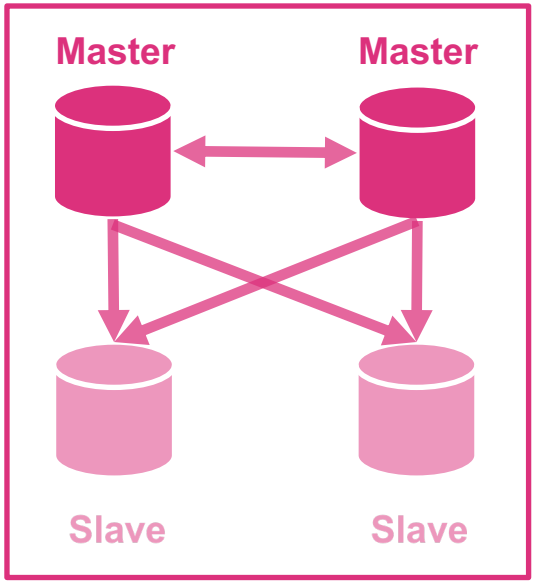
```
$ use local
switched to db local
$ db.oplog.rs.find()
{
  "ts" : { "t" : 1534616696000, "i" : 1 },
  "h" : NumberLong("1342870845645633201"),
  "op" : "i",
  "ns" : "test.user",
  "o" : {
    "_id" : ObjectId("4cb35859543cc1f4f9f7f85d"),
    "user_id" : 1
  }
}
{
  "ts" : { "t" : 1534616699000, "i" : 1 },
  "h" : NumberLong("1233487572903545434"),
  "op" : "u",
  "ns" : "test.user",
  "o2" : {
    "_id" : ObjectId("4cb35859543cc1f4f9f7f85d")
  },
    "o" : {
      "$set" : { "married_to" : 4711 }
    }
}
```

Multi-Master Replication

Master-based

e.g. **Oracle, PostgreSQL, MySQL…**

Multi-Master-based



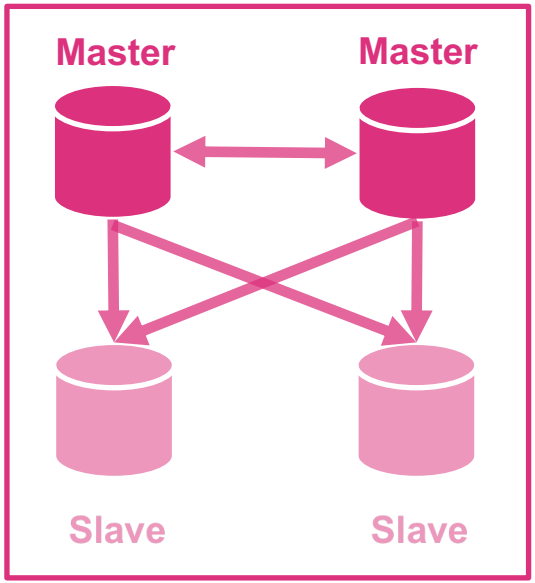
e.g. **CouchDB, PostgreSQL, MySQL…**

Masterless

e.g. **Cassandra, Riak, Vol demortDB, DynamoDB…**

Multi-Master Replication

Also known as: *Multi-Leader, Active/Active or Master/Master Replication*



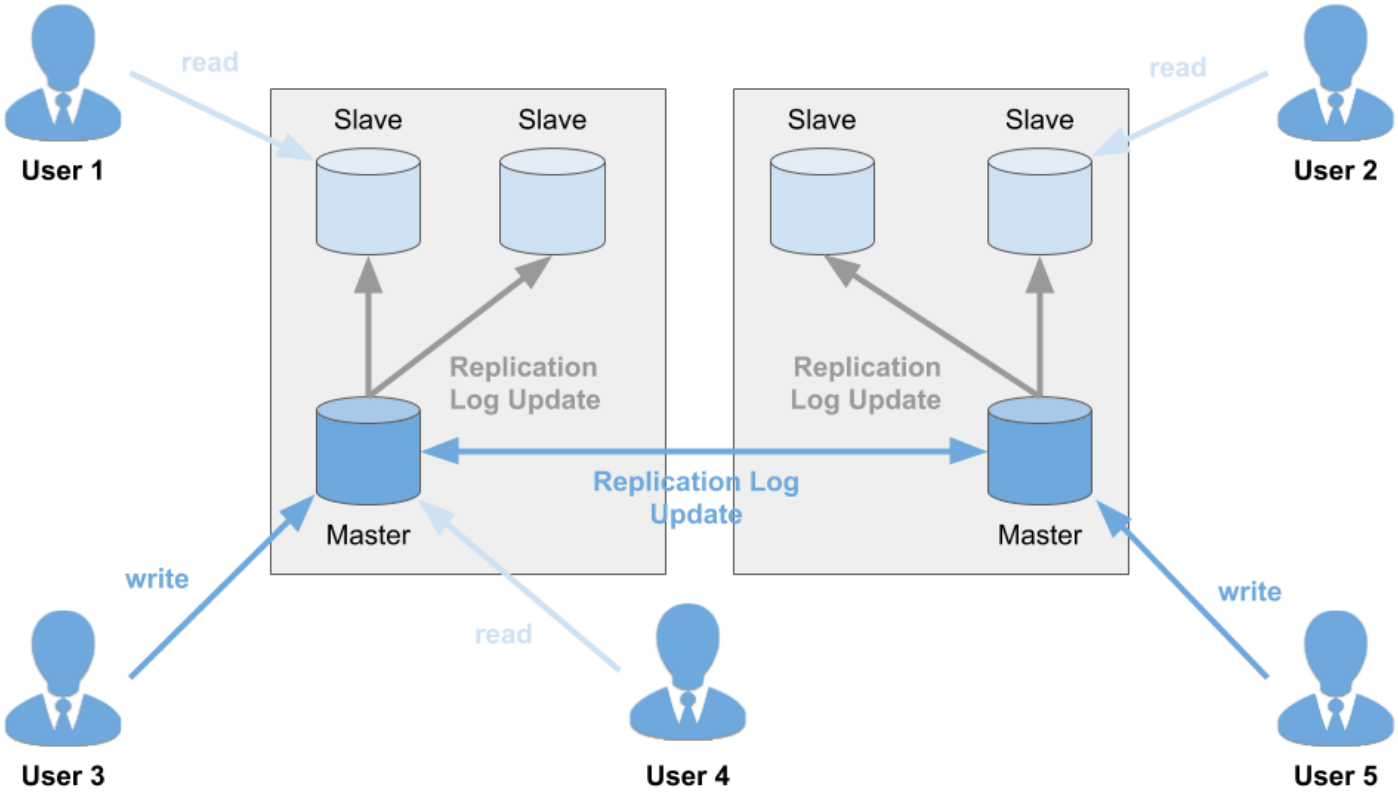
Setup:
- **multiple master** nodes (which accept write requests)
- query and replication processing similar to master-based replication

Advantages:
- **fault tolerance** (ability to mitigate faults of master nodes)
- **write performance** and horiz. scalability (parallel writes)
- able to run on **multiple datacenters** (e.g. one master in each of them)

Disadvantages:
- **write conflicts** possible (requires conflict resolution)
- complexity

www.marcel-mittelstaedt.com

Multi-Master Replication – **Example**



Multi-Master Replication - **Conflicts**

- **same record/dataset** is being **edited in parallel** using **multiple master nodes**

→ e.g. editing the same line of a document within Google Docs simultanously (user 1 on a master node based in Frankfurt and user 2 on a master node in Berlin)
→ Later asynchronous replication between the two master nodes will conflict (this won't happen if you are using a master replication).
→ **Possible solution:** application needs to ensure both users are pushing their write request to the same master.

But what if the application is not able to prevent those confict? **Approaches:**

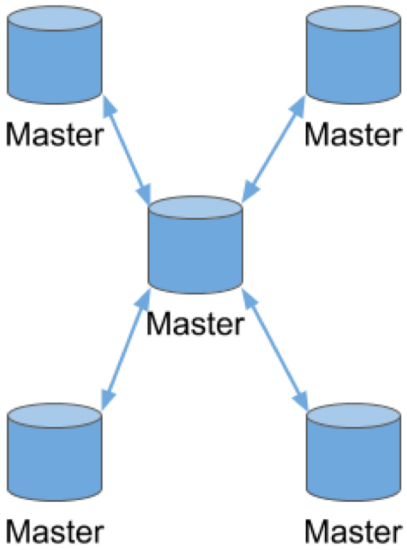
LWW: only accept the most recent operation by e.g. unique operation ID, timestamp or both (*Last-Write-Wins*). It is important to notice that this approach is highly vulnerable to data loss.

Merge: merge values of multiple updates together (e.g. alphabetically or in order of appearance).

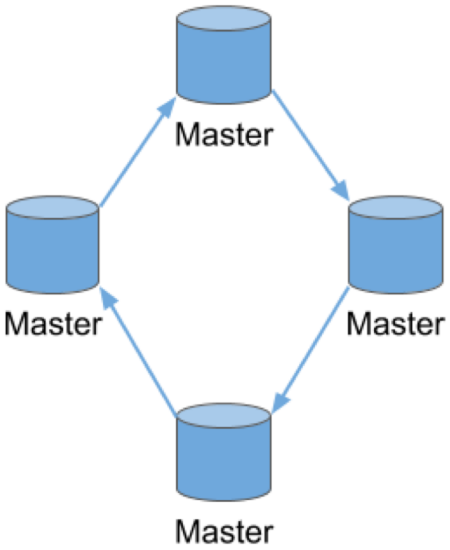
Application Managed: persist the conflict in a way, that it preserves all information without loss and let the application, using the data-system, or rather the user of the application take care about it later on (e.g. implemented by SVN or Git)

Multi-Master Replication - **Topologies**

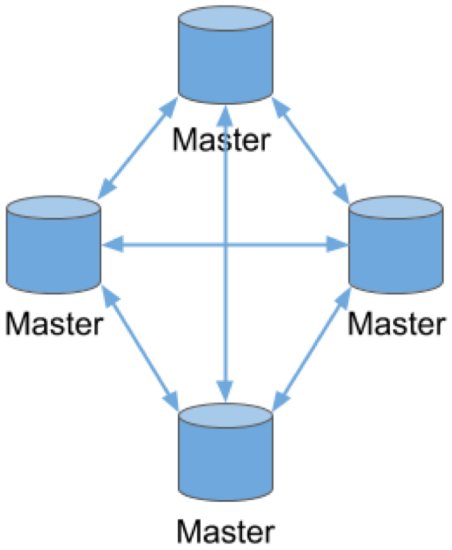
Star Topology



Circle Topology



All-To-All Topology



Masterless Replication

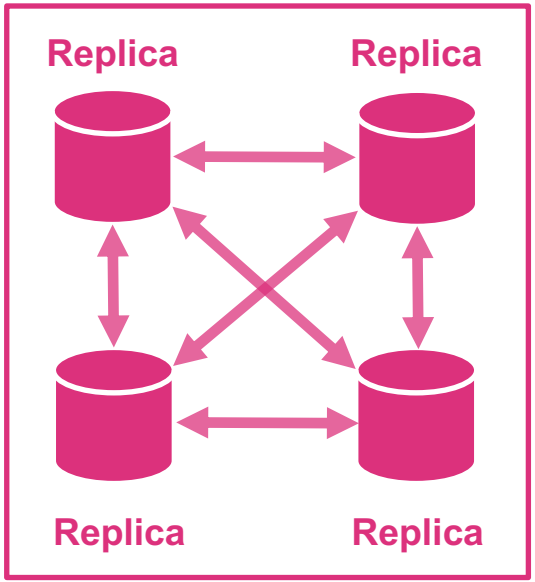
Master-based

e.g. **Oracle, PostgreSQL, MySQL…**

Multi-Master-based

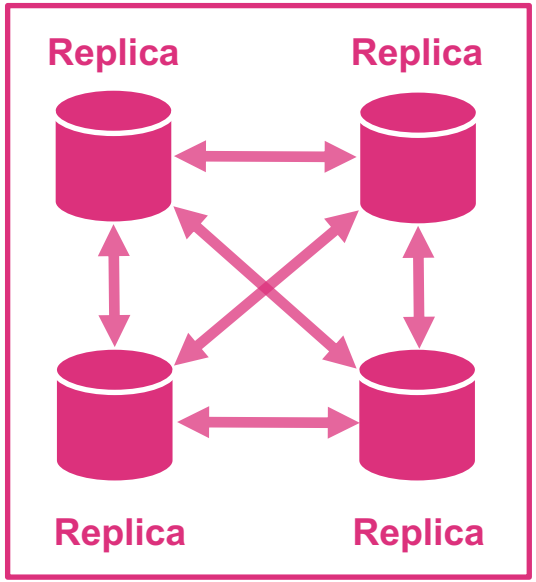
e.g. **CouchDB, PostgreSQL, MySQL…**

Masterless



e.g. **Cassandra, Riak, Voldemort, DynamoDB…**

Masterless Replication



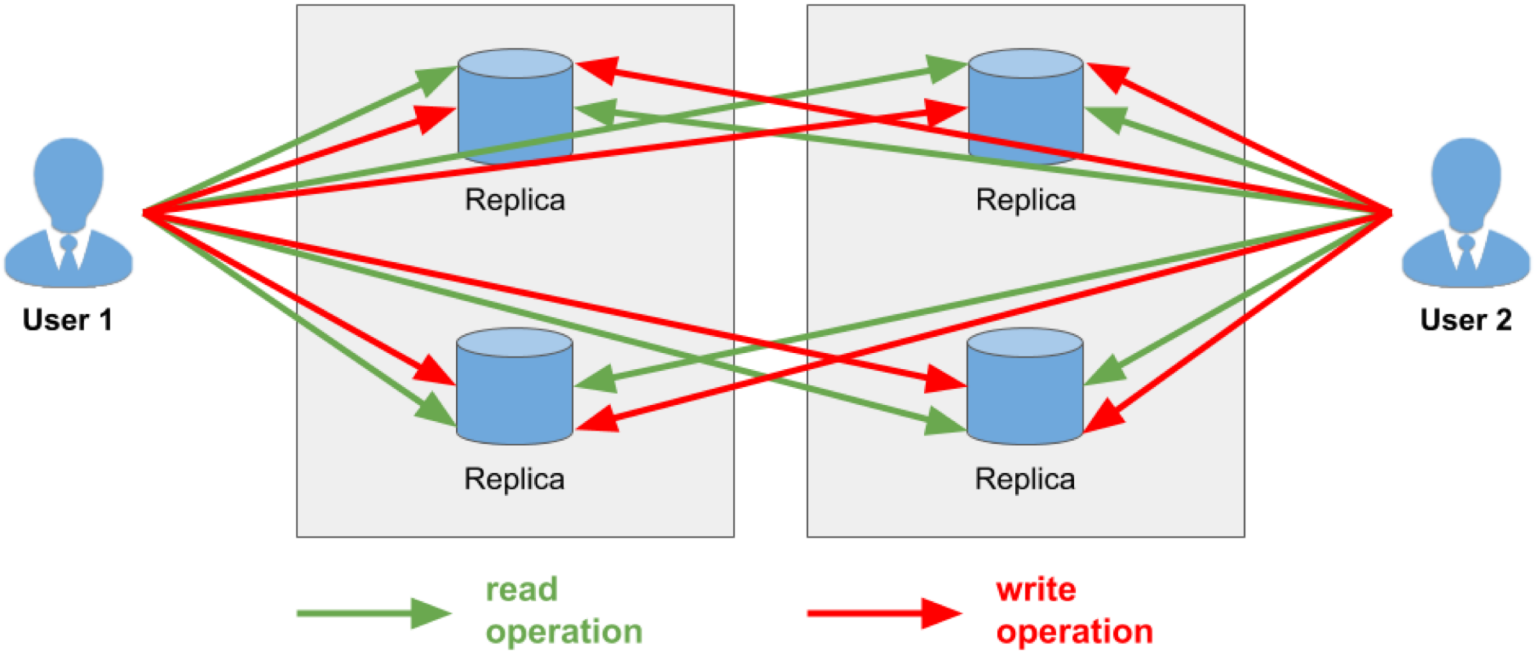
Also known as:
Leaderless Replication

Setup:
- **no leader** (all nodes/replicas accept write requests)
- makes use of **gossip protocol** (all replica nodes run local processes which periodically match their states)
- queries are sent to multiple nodes of data-system
  → if a certain number of nodes succeeded, the query is considered successful

Advantages:
- **fault tolerance** (tolerates multiple failed or slow nodes)
- **write performance** and horiz. scalability (parallel writes)
- able to run on **multiple datacenters** (with respect to *quorum*)

Disadvantages:
- **consistency, complexity** and additional **tasks** clients need to handle (e.g. quorum, conflict resolution, consistency)
- most data-system are key/value stores

Masterless Replication – Example



Masterless Replication- **Quorums**

Quorum:

- a *quorum* is the minimum number of votes that a read/write request to a distributed data-system has to obtain in order to be sure the requests is successful (and the result consistent)

Reasonable quorum:

- every read and write request must be processed and confirmed by at least **r nodes** (read) or acknowledged by at least **w nodes** (write):

$$r + w > n$$

→ sure to get an up-to-date resul, as at least one of the replica will have the most-recent value

- able to run on **multiple datacenters** (with respect to *quorum*)

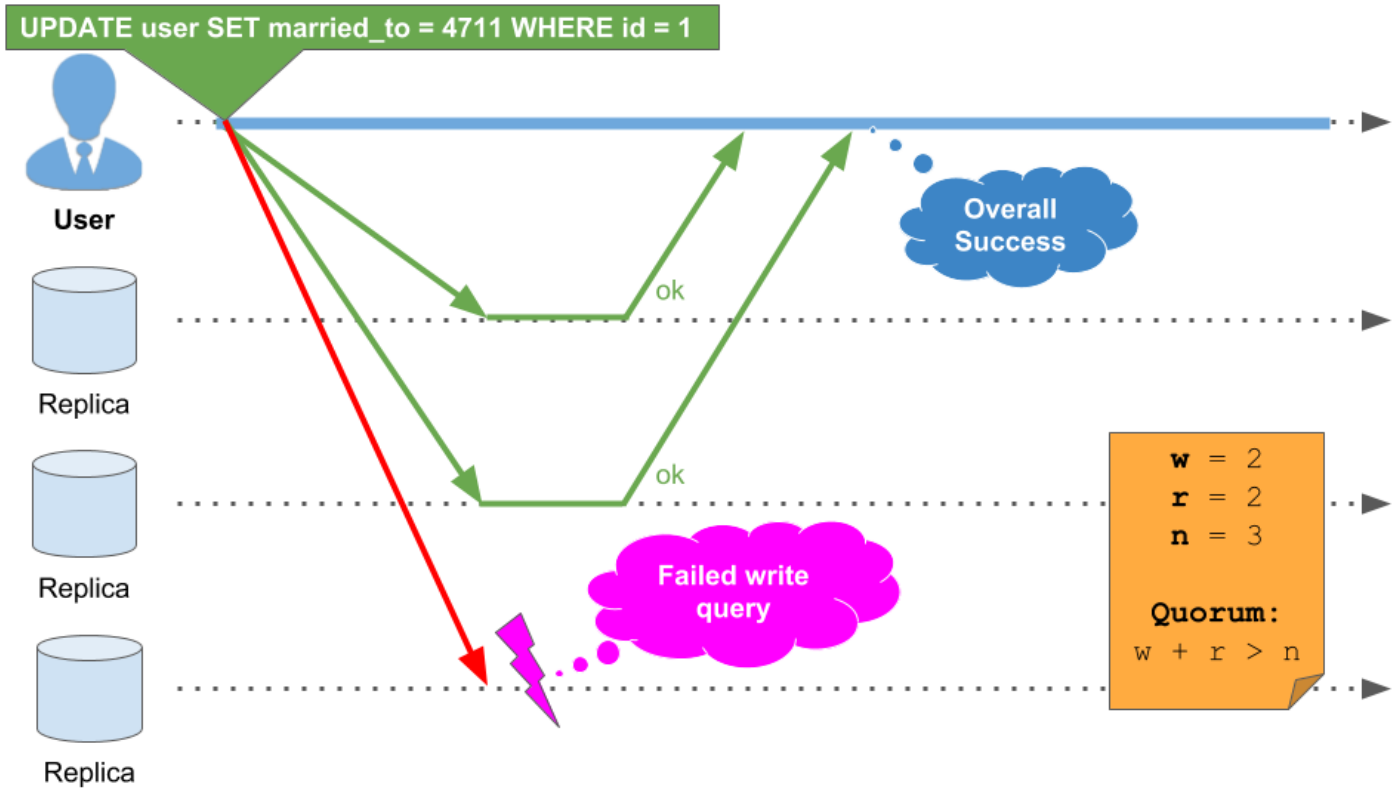
Masterless Replication- **Quorums**

- **r** and **w** are configurable depending on *read/write performance* and *fault-tolerance* you want to achieve
  - → smaller **r** = faster reads, slower writes
  - → smaller **w** = faster writes, slower reads

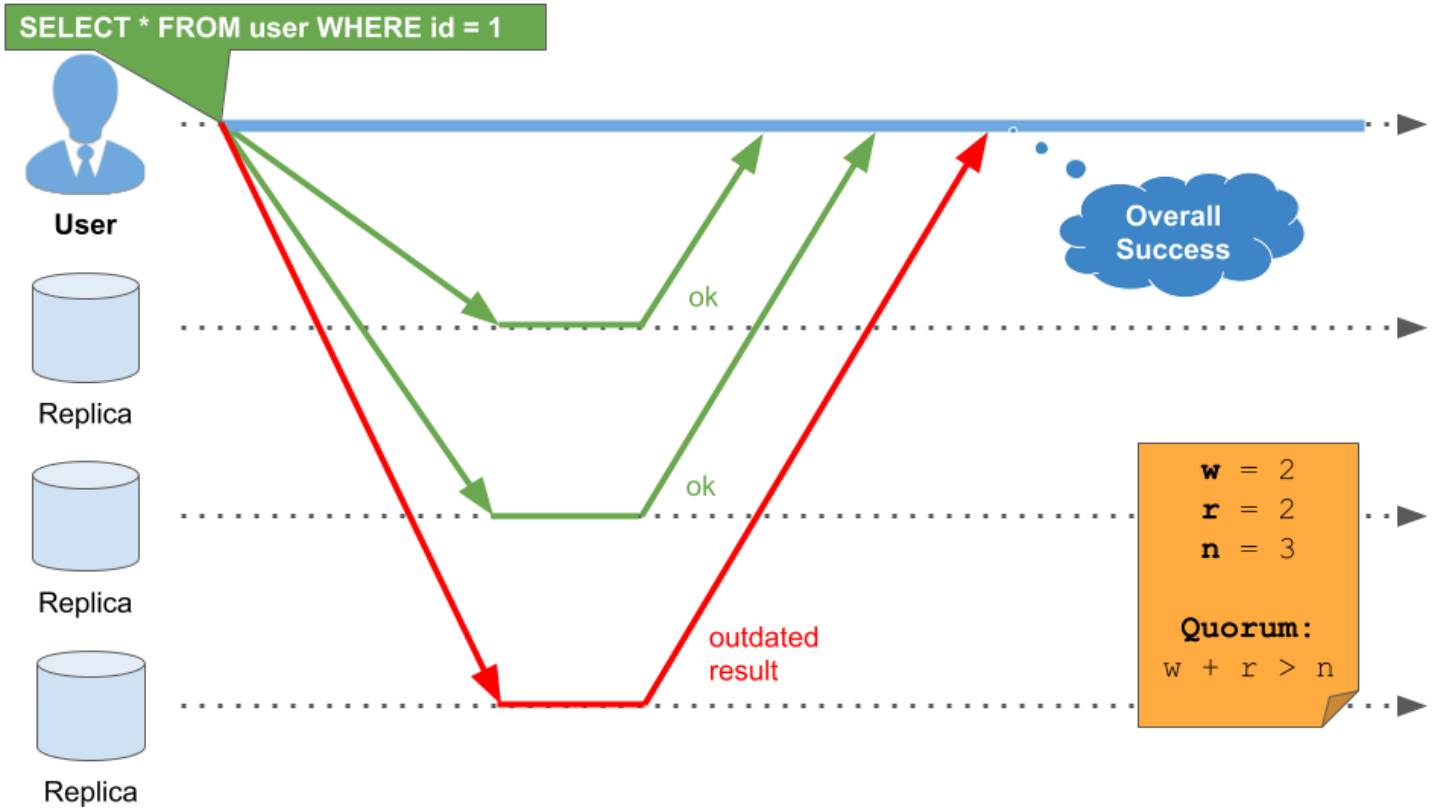
- Number of node failures a data-system can handle, is calculated by:
  - → **n - r = number** of nodes tolerated to be unavailable for read requests
  - → **n - w = number** of nodes tolerated to be unavailable for write requests

- For instance, a data-system of **5 nodes** (n = 5, r = 3 and w = 3) is able to tolerate 2 unavailable nodes.

Masterless Replication- **Quorum Write**



Masterless Replication- **Quorum Read**



Masterless Replication – **Eventual Consistency**

Read Repair: If a user application or controller node executes a read requests and recognizes a replica node with a *stale* value, the user application or controller sends it the most recent value afterwards.
For instance used by *Cassandra*, *Riak* and *VoldemortDB*.

Anti-Entropy Repair: Some data-systems are running background processes or provide tools (e.g. *nodetool repair* in case of *Casssandra*) which run in background constantly looking for differences between different replica nodes and copying data from one node to another to keep everything up-to- date.
For instance used by *Cassandra* and *Riak* but not supported by *VoldemortDB*.

Hinted Handoff: If a node is unable to process a particular write request, the user application or coordinator node (which executed the write request) preserves the data to be written as a set of hints. As soon as the faulty node comes back online, the user application or coordinator triggers a repair process by handing off hints to the faulty node to catch up with the missed writes.
For instance used by Cassandra and Riak but not supported by VoldemortDB.

Masterless Replication- Limitations of Quorums

Concurrent Writes: If two write requests are executed concurrently, it is **not clear which one happened first**, as both are executed from different controller nodes or applications. Both requests need to be merged, for instance based on a timestamp (*last-write-wins*), which is highly vulnerable to *clock skew*, causing **older values to overwrite more recent ones**.
For instance *Cassandra* is based on *last-write-wins* whereas *Riak* requires the admin to choose whether to make use of *last-write-wins* or *handle write conflicts within the application* using the data-system.

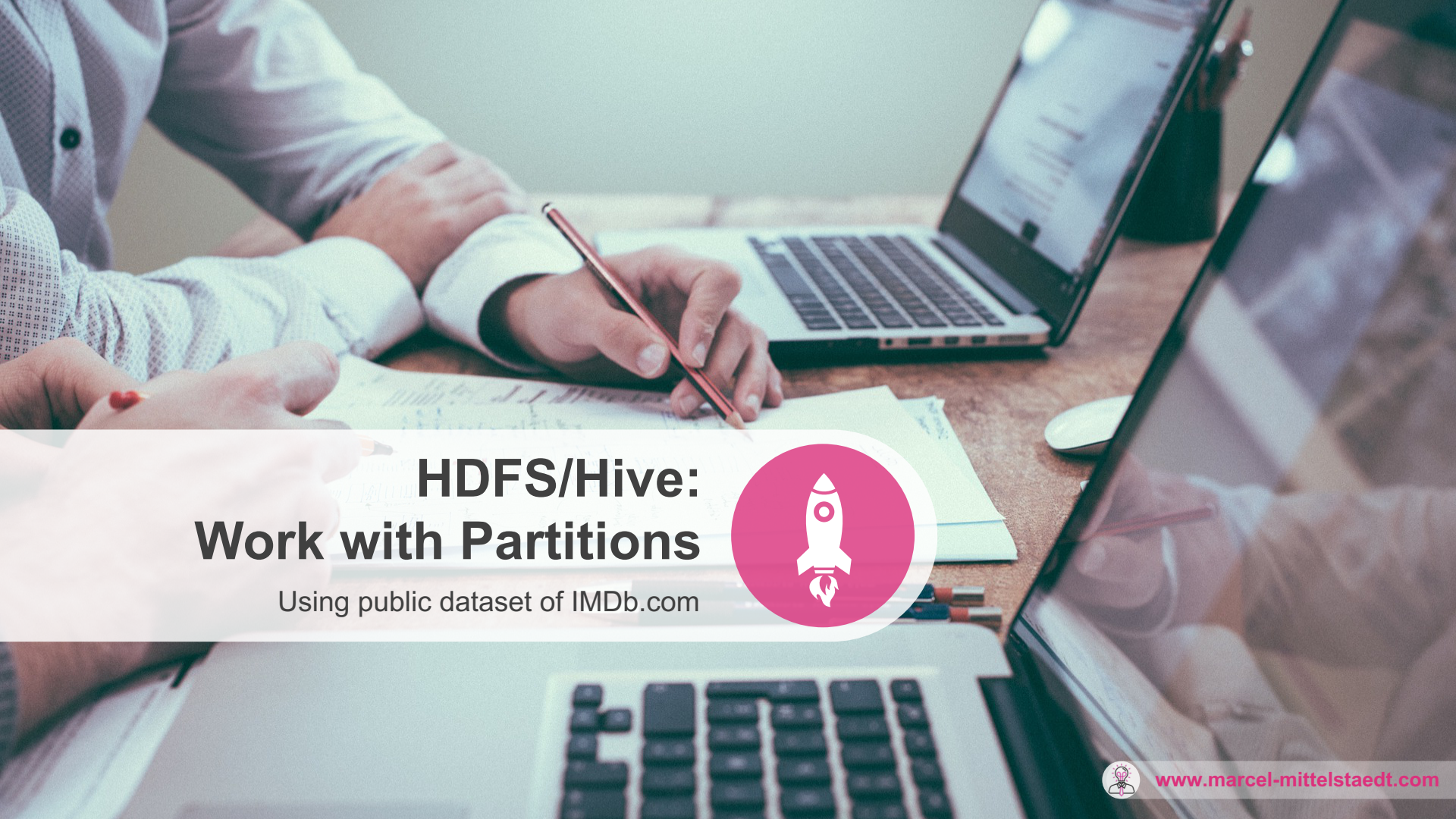
Concurrent Reads And Writes: If **read and write requests** (regarding the same value) happen at the **same time** it is unclear whether the read request returns the **stale or the new value**. As the write request may succeeded only on some nodes during execution of the read query, the new value might be under-represented, causing the old value to win the quorum.

Node Failure: If a node previously processed a new value, crashed and comes back online and is **restored** from a node with the **old value**, the **quorum might be violated** as the number of nodes storing the new value might be < w.

Masterless Replication- **Limitations of Quorums**

Sloppy Quorum and Hinted Handoff: There are cases like network or datacenter outages causing some user or consumer applications being cut off from some nodes of a data-system (while the unreachable nodes are still online). In this case it is possible that some user or consumer applications won't be able to achieve a quorum. If the data-systems contains more than n nodes, it needs to make a crucial decision here, whether to ignore all requests that cannot achieve a quorum or still accept write requests and just write them to some nodes outside of n to ensure write availability and durability. A sloppy quorum still requires r and w nodes, but those do not need to be one of the original n nodes. As soon as the network or datacenter outage is fixed, all writes processed by nodes outside of n are sent to the appropriate nodes inside of n (*Hinted Handoff*).
Using this approach it is no longer guaranteed a data-system will provide the most recent value as read and write requests may not overlap on r and w nodes. This could also be mitigated by using versioning of values, like *vector clocks* (e.g. used by *DynamoDB*).
For instance *sloppy quorums* are enabled by default within *Riak* and disabled by default within *Cassandra*.

HandsOn – HDFS/Hive Partitioning and HiveServer2

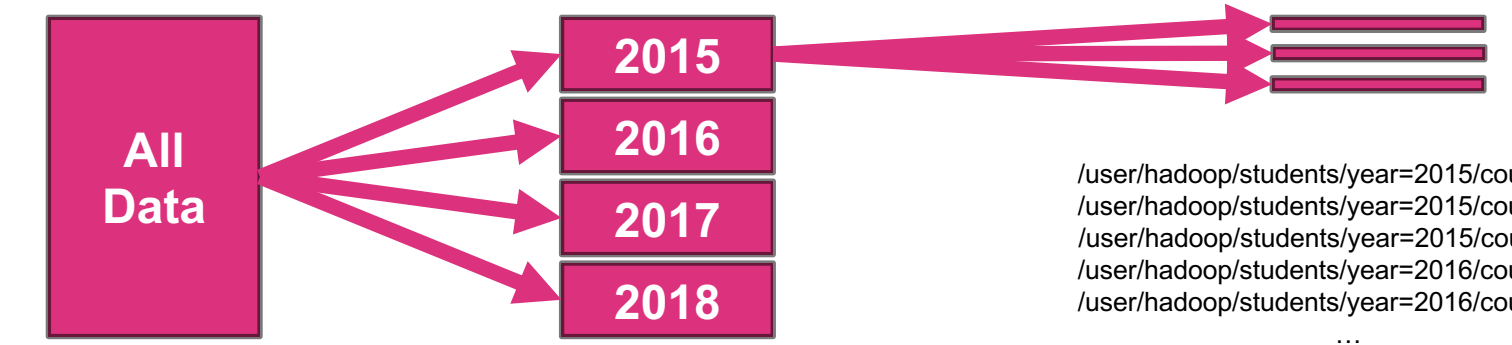


HDFS/Hive: Work with Partitions

Using public dataset of IMDb.com

HDFS/Hive - **Partitioning**

- Partitioning of data distributes load and speeds up data processing
- A table can have one or more partition columns, defined by the time of creating a table (`CREATE TABLE student(id Int, name STRING) PARTITIONED BY (year STRING)… STORED AS TEXTFILE LOCATION '/user/hadoop/students';`)
- partitioning can be done either **static** or **dynamic**
- each distinct value of a partition column is represented by a **HDFS directory**



www.marcel-mittelstaedt.com

Static Partitioning – **Create Partitioned Table**

1. Create partitioned version of table `imdb_ratings`: **`imdb_ratings_partitioned:`**

```
hive >    CREATE TABLE IF NOT EXISTS imdb_ratings_partitioned(
                tconst STRING,
                average_rating DECIMAL(2,1),
                num_votes BIGINT
          ) PARTITIONED BY (partition_quality STRING)
          STORED AS ORCFILE LOCATION '/user/hadoop/imdb/ratings_partitioned';
```

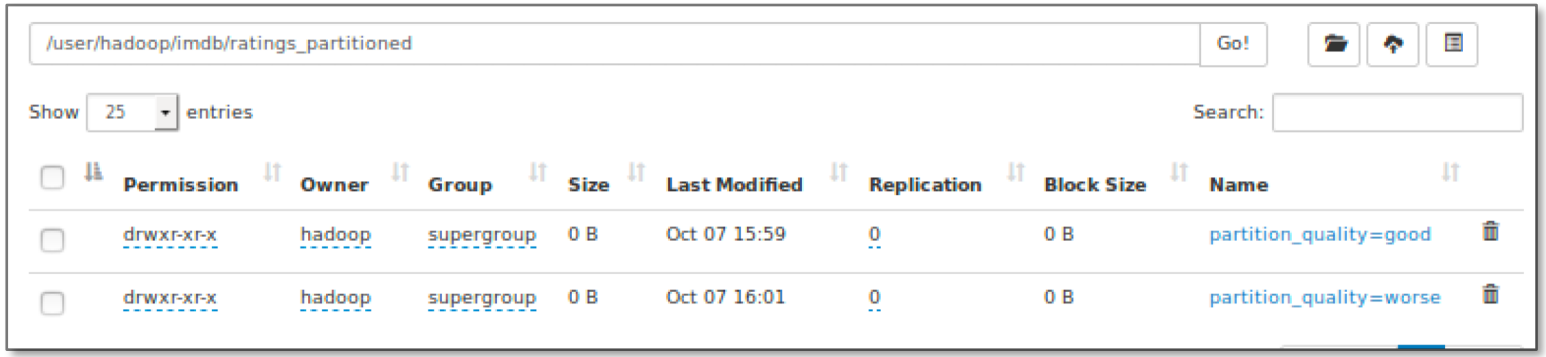

Static Partitioning – INSERT Into Table via Hive

1. Migrate and partition data of table `imdb_ratings` to table **`imdb_ratings_partitioned:`**

```
INSERT OVERWRITE TABLE imdb_ratings_partitioned partition(partition_quality='good')
SELECT r.tconst, r.average_rating, r.num_votes FROM imdb_ratings r WHERE r.average_rating >= 7;

INSERT OVERWRITE TABLE imdb_ratings_partitioned partition(partition_quality='worse')
SELECT r.tconst, r.average_rating, r.num_votes FROM imdb_ratings r WHERE r.average_rating < 7;
```

2. Check Success on HDFS:



www.marcel-mittelstaedt.com

Static Partitioning – **INSERT Into Table via Hive**

3. Check Success via Hive:

```
select distinct average_rating from imdb_ratings_partitioned where partition_quality = 'good';

7.0
7.1
7.2
7.3
7.4
7.5
7.6
7.7
[...]
9.0
9.1
9.2
9.3
9.4
9.5
9.6
9.7
9.8
9.9
```


Dynamic Partitioning – **Create Partitioned Table**

1. Create partitioned version of table `imdb_movies`: **`imdb_movies_partitioned:`**

```
hive >  CREATE TABLE IF NOT EXISTS imdb_movies_partitioned(
          tconst STRING,
          title_type STRING,
          primary_title STRING,
          original_title STRING,
          is_adult DECIMAL(1,0),
          start_year DECIMAL(4,0),
          end_year STRING,
          runtime_minutes INT,
          genres STRING
) PARTITIONED BY (partition_year int)
STORED AS ORCFILE
LOCATION '/user/hadoop/imdb/name_partitioned';
```

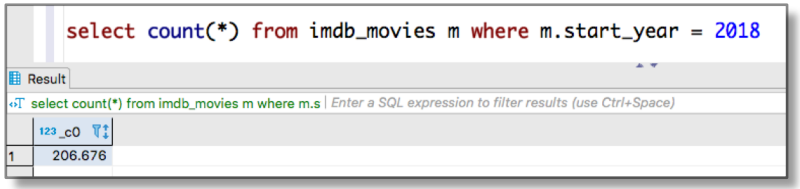
Dynamic Partitioning – INSERT Into Table via Hive

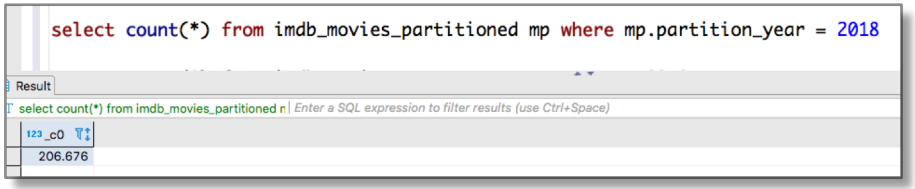
1. Migrate and partition data of table `imdb_movies` to table **`imdb_movies_partitioned:`**

```
set hive.exec.dynamic.partition.mode=nonstrict; -- enable dynamic partitioning

INSERT OVERWRITE TABLE imdb_movies_partitioned partition(partition_year)
SELECT m.tconst, m.title_type, m.primary_title, m.original_title, m.is_adult,
m.start_year, m.end_year, m.runtime_minutes, m.genres,
m.start_year -- last column = partition column
FROM imdb_movies m
```

2. Check Success via Hive:





Dynamic Partitioning – **INSERT Into Table via Hive**

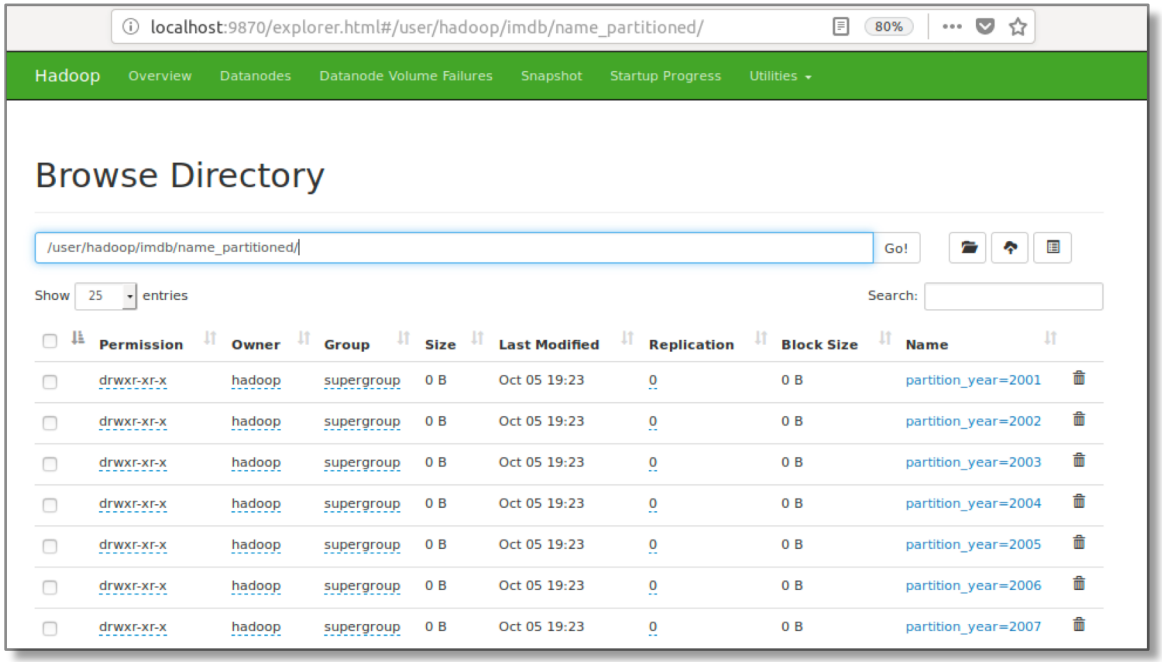
3. Check Success on HDFS:

```
hadoop fs -ls /user/hadoop/imdb/name_partitioned
[…]
drwxr-xr-x   - hadoop supergroup          0 2018-10-05 19:23 /user/hadoop/imdb/name_partitioned/partition_year=2011
drwxr-xr-x   - hadoop supergroup          0 2018-10-05 19:23 /user/hadoop/imdb/name_partitioned/partition_year=2012
drwxr-xr-x   - hadoop supergroup          0 2018-10-05 19:23 /user/hadoop/imdb/name_partitioned/partition_year=2013
drwxr-xr-x   - hadoop supergroup          0 2018-10-05 19:23 /user/hadoop/imdb/name_partitioned/partition_year=2014
drwxr-xr-x   - hadoop supergroup          0 2018-10-05 19:23 /user/hadoop/imdb/name_partitioned/partition_year=2015
drwxr-xr-x   - hadoop supergroup          0 2018-10-05 19:23 /user/hadoop/imdb/name_partitioned/partition_year=2016
drwxr-xr-x   - hadoop supergroup          0 2018-10-05 19:23 /user/hadoop/imdb/name_partitioned/partition_year=2017
drwxr-xr-x   - hadoop supergroup          0 2018-10-05 19:23 /user/hadoop/imdb/name_partitioned/partition_year=2018
drwxr-xr-x   - hadoop supergroup          0 2018-10-05 19:23 /user/hadoop/imdb/name_partitioned/partition_year=2019
drwxr-xr-x   - hadoop supergroup          0 2018-10-05 19:23 /user/hadoop/imdb/name_partitioned/partition_year=2020
drwxr-xr-x   - hadoop supergroup          0 2018-10-05 19:24 /user/hadoop/imdb/name_partitioned/partition_year=2021
drwxr-xr-x   - hadoop supergroup          0 2018-10-05 19:22 /user/hadoop/imdb/name_partitioned/partition_year=2022
drwxr-xr-x   - hadoop supergroup          0 2018-10-05 19:22 /user/hadoop/imdb/name_partitioned/partition_year=2023
drwxr-xr-x   - hadoop supergroup          0 2018-10-05 19:24 /user/hadoop/imdb/name_partitioned/partition_year=2024
drwxr-xr-x   - hadoop supergroup          0 2018-10-05 19:22 /user/hadoop/imdb/name_partitioned/partition_year=2025
drwxr-xr-x   - hadoop supergroup          0 2018-10-05 19:22 /user/hadoop/imdb/name_partitioned/partition_year=2115

hadoop fs -ls /user/hadoop/imdb/name_partitioned/partition_year=2018Found 1 items
-rw-r--r--   1 hadoop supergroup    5371795 2018-10-05 19:23 /user/hadoop/imdb/name_partitioned/partition_year=2018/000
007_0
```

Dynamic Partitioning – INSERT Into Table via Hive

4. Check Success via HDFS Web Browser:



Exercises I

HDFS/Hive: Work with Partitions

HDFS/Hive Partitioning Exercises - **IMDB**

1. Execute Tasks of previous HandsOn Slides

2. Create a (*statically*) partitioned table **`imdb_actors_partitioned`**, which:
   - contains all columns of table imdb_actors
   - is statically partitioned by `partition_is_alive`, containing:
     - „*alive*“ in case actor is still alive
     - „*dead*“ in case actor is already dead

   Load all data from `imdb_actors` to table **`imdb_actors_partitioned`**

3. Create a (*dynamically*) partitioned table **`imdb_movies_and_ratings_partitioned`**, which:
   - contains all columns of the two tables `imdb_movies` and `imdb_ratings` and
   - is partitioned by start year of movie (create and add column `partition_year`).

   Load all data of `imdb_movies` and `imdb_ratings` to table:
   `imdb_movies_and_ratings_partitioned`

Exercises II Preparation

Setup HiveServer2 For Remote Connections

Setup HiveServer2

1. Edit Hive Config and add (*hive/conf/hive-site.xml*):

```
<property>
          <name>hive.server2.thrift.min.worker.threads</name>
          <value>3</value>
</property>
<property>
          <name>hive.server2.thrift.max.worker.threads</name>
          <value>5</value>
</property>
<property>
          <name>hive.server2.thrift.port</name>
          <value>10000</value>
</property>
<property>
          <name>hive.server2.thrift.bind.host</name>
          <value>localhost</value>
</property>
```

Setup HiveServer2

2. Update Hadoop Config and add (*hadoop/etc/hadoop/core-site.xml*):

```
<property>
          <name>hadoop.proxyuser.hadoop.hosts</name>
          <value>*</value>
 </property>
<property>
          <name>hadoop.proxyuser.hadoop.groups</name>
          <value>*</value>
</property>
```

3. Restart DFS and YARN:

```
stop-all.sh
start-dfs.sh
Start-yarn.sh
```


Setup HiveServer2

4. Start HiveServer2:

```
hive/bin/hiveserver2

2018-10-02 16:19:08: Starting HiveServer2
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/home/hadoop/hive/lib/log4j-slf4j-impl-2.10.0.jar!/org/slf4
j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/home/hadoop/hadoop/share/hadoop/common/lib/slf4j-log4j12-1
.7.25.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.apache.logging.slf4j.Log4jLoggerFactory]
Hive Session ID = b8d1efb3-fc8c-4ec8-bdf0-6a9a41e2ddaa
Hive Session ID = 32503981-a5fd-497e-b887-faf3ec1e686e
Hive Session ID = 00f7eab4-5a29-4ce4-ad97-e90904d9206f
Hive Session ID = 100e54c5-14c6-4acc-b398-040152b08ebf
[…]
```


Connect To HiveServer2 via JDBC

1. Download JDBC SQL Client, e.g. *DBeaver*:

Mac OSX:

```
wget https://dbeaver.io/files/dbeaver-ce-latest-installer.pkg
```

Linux (Debian):

```
wget https://dbeaver.io/files/dbeaver-ce_latest_amd64.deb
```

Linux (RPM):

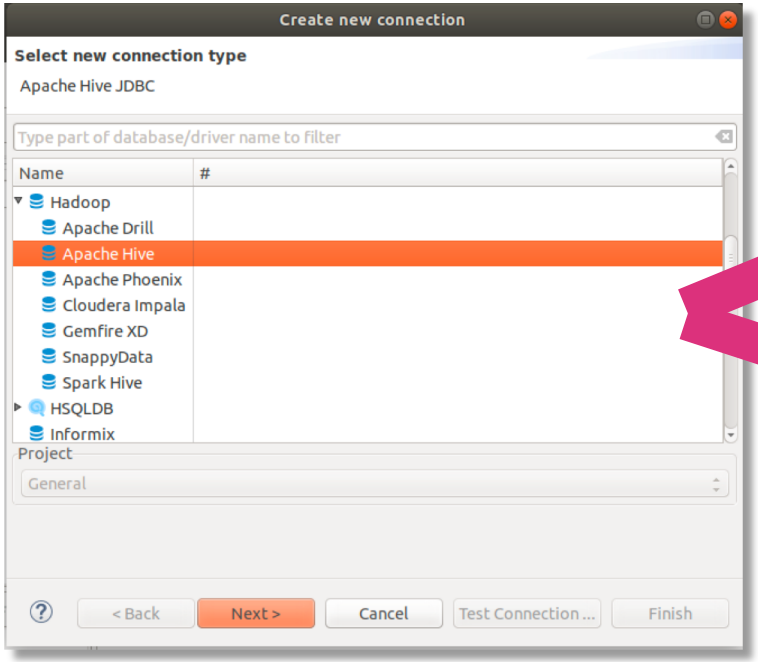
```
wget https://dbeaver.io/files/dbeaver-ce-latest-stable.x86_64.rpm
```

Windows:

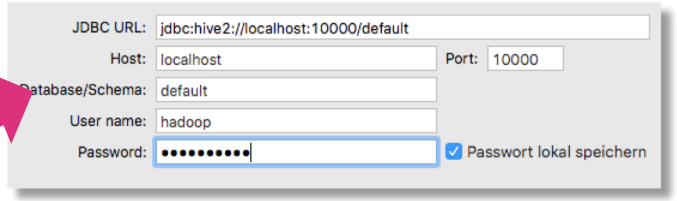
```
wget https://dbeaver.io/files/dbeaver-ce-latest-x86_64-setup.exe
```


Connect To HiveServer2 via JDBC

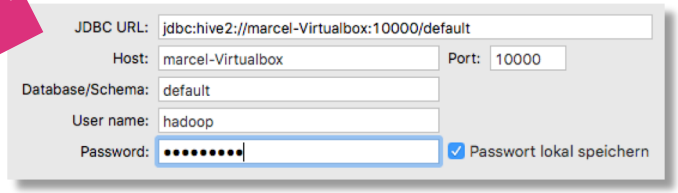
2. Configure Connection To Hive Server:



From **inside** VM:

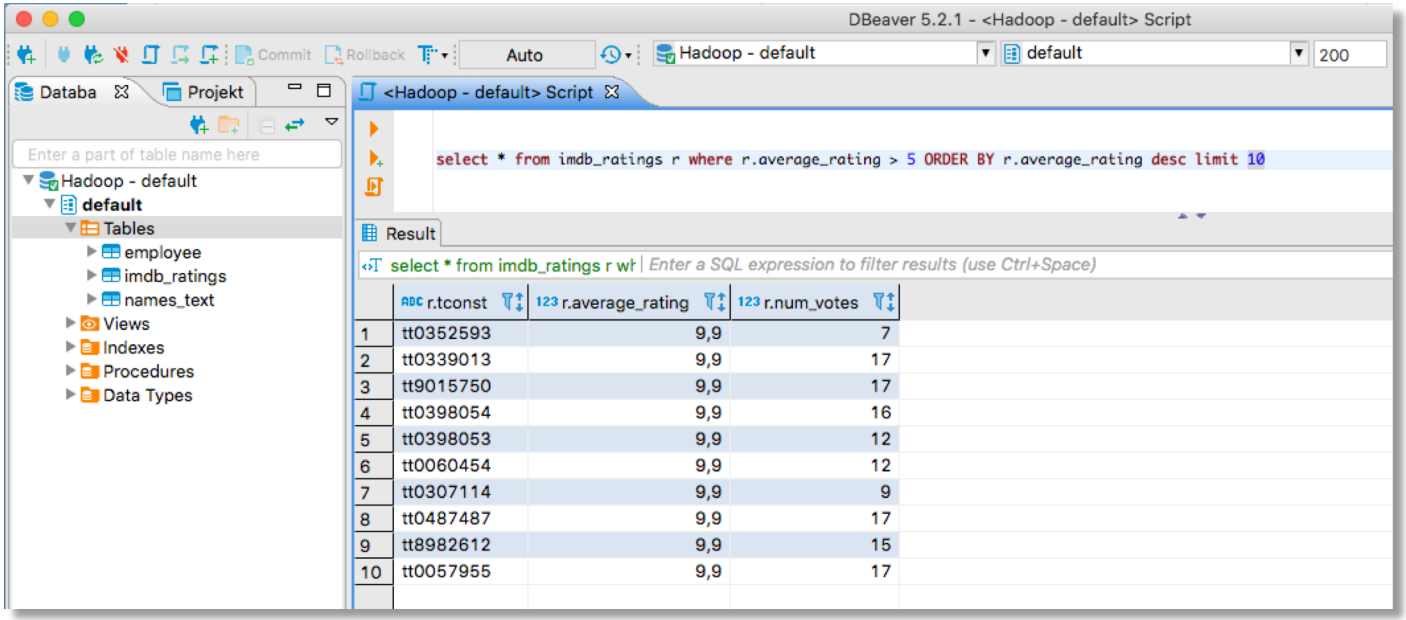


From **outside** VM:



Connect To HiveServer2 via JDBC

3. Query something, e.g.:



Exercises II

HiveServer2 For Remote Connections

HiveServer2 Exercises

1. Execute Tasks of previous HandsOn Slides.

2. Run any query.