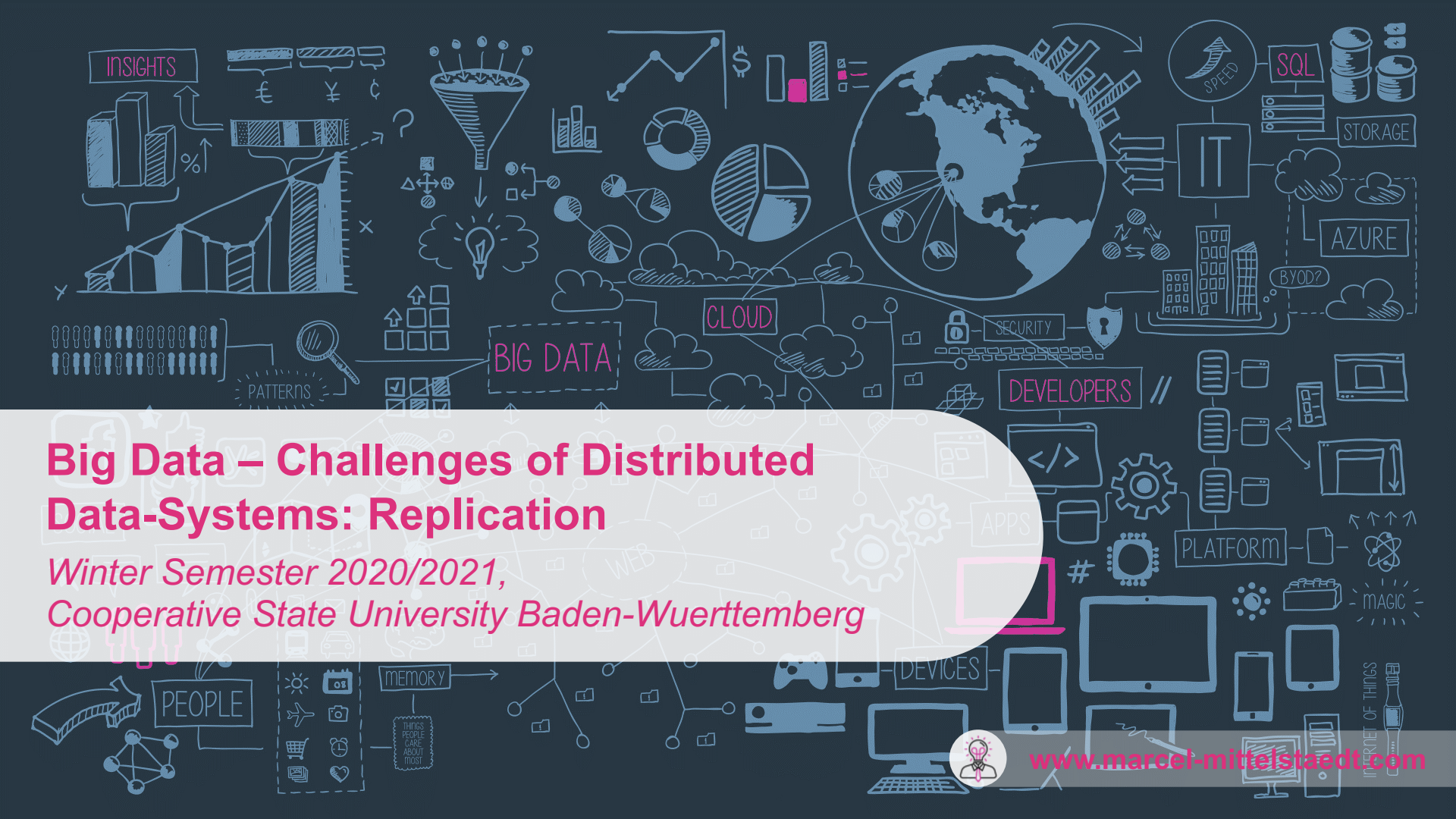


Big Data – Challenges of Distributed Data-Systems: Replication

Winter Semester 2020/2021,
Cooperative State University Baden-Wuerttemberg

Agenda – 01.03.2021

01

Presentation and Discussion: Exercise Of Last Lecture

HiveServer2, HiveQL via JDBC, Partitioning with HDFS and Hive

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 – Spark, PySpark and Jupyter. Working on IMDb Dataset.

Quick Introduction To Apache Spark, especially Spark-Shell, PySpark Shell and Jupyter Notebooks. Use PySpark Shell and/or Jupyter to solve certain problems.

04

Exercise – Use PySpark and Jupyter to work with Partitions on IMDb dataset.

Using PySpark to read/Write DataFrames and Spark Tables from/to HDFS, work with Spark DataFrames, Join and aggregate DataFrames, Plot Data

Schedule

	Lecture Topic	*HandsOn*
08.02.2021 13:00-15:45 Ro. N/A	About This Lecture, Introduction to Big Data	Setup Google Cloud, Create Own Hadoop Cluster and Run MapReduce
15.02.2021 13:00-15:45 Ro. N/A	(Non-)Functional Requirements Of Distributed Data-Systems, Data Models and Access	Hive and HiveQL
22.02.2021 13:00-15:45 Ro. N/A	Challenges Of Distributed Data Systems: Partitioning	HiveQL via JDBC, Data Partitioning (with HDFS and Hive)
01.03.2021 13:00-15:45 Ro. N/A	**Challenges Of Distributed Data Systems: Replication**	Spark, Scala, PySpark and Jupyter Notebooks
08.03.2021 13:00-15:45 Ro. N/A	ETL Workflow and Automation	Airflow
15.03.2021 13:00-15:45 Ro. N/A	Batch and Stream Processing	
22.03.2021 13:00-15:45 Ro. N/A	Practical Exam	Work On Practical Exam
29.03.2021 13:00-15:45 Ro. N/A	Practical Exam	Work On Practical Exam
12.04.2021 13:00-15:45 Ro. N/A	**Practical Exam Presentation**	
19.04.2021 13:00-15:45 Ro. N/A	**Practical Exam Presentation**	

Solution – Exercise IV

HiveServer 2, HiveQL via JDBC,
Partitioning with HDFS and Hive

Solution

Prerequisites:

- Start Gcloud instance
- Pull and start Docker image (`marcelmittelstaedt/hiveserver_base:latest`)
- Start Hadoop Cluster
- Start HiveServer2
- Download, Install and Configure JDBC Rich-client:
    - e.g. DBeaver,
    - SquirrelSQL,
    - …
- Execute all preparation and example tasks of previous HandsOn slides of last lecture

Solution

Exercise IV:

2.1 Create table `name_basics_partitioned` partitioned by column `partition_is_alive`:

```
CREATE EXTERNAL TABLE IF NOT EXISTS name_basics_partitioned (
          nconst STRING,
          primary_name STRING,
          birth_year INT,
          death_year STRING,
          primary_profession STRING,
          known_for_titles STRING
) PARTITIONED BY (partition_is_alive STRING)
STORED AS PARQUET LOCATION '/user/hadoop/imdb/actors_partitioned';
```

Solution

Exercise IV:

2.2 Use static partitioning to create and fill partition ‘alive‘

```
INSERT OVERWRITE TABLE name_basics_partitioned
partition(partition_is_alive='alive')
SELECT
          a.nconst,
          a.primary_name,
          a.birth_year,
          a.death_year,
          a.primary_profession,
          a.known_for_titles
FROM name_basics a WHERE a.death_year IS NULL
```

Solution

Exercise IV:

2.3 Use static partitioning to create and fill partition ‘dead‘

```
INSERT OVERWRITE TABLE name_basics_partitioned
partition(partition_is_alive='dead')
SELECT
          a.nconst,
          a.primary_name,
          a.birth_year,
          a.death_year,
          a.primary_profession,
          a.known_for_titles
FROM name_basics a WHERE a.death_year IS NOT NULL
```


Solution

Exercise IV:

2.4 Check Results:

```
hadoop fs -ls /user/hadoop/imdb/actors_partitioned
drwxr-xr-x   - hadoop supergroup          0 2021-02-27 17:16 /user/hadoop/imdb/actors_partitioned/partition_is_alive=alive
drwxr-xr-x   - hadoop supergroup          0 2021-02-27 17:16 /user/hadoop/imdb/actors_partitioned/partition_is_alive=dead
```


Solution

Exercise IV:

2.4 Check Results:

```
SELECT * FROM name_basics_partitioned  WHERE partition_is_alive = 'dead' LIMIT 100
```

Result

SELECT * FROM name_basics_partitioned WH | Geben Sie einen SQL-Ausdruck ein, um die Ergebnisse zu filtern (verwenden Sie Strg+ Leertaste).

	nconst	primary_name	birth_year	death_year	primary_profession	known_for_titles	partition_is_alive
1	nm0000001	Fred Astaire	1.899	1987	soundtrack,actor,miscellaneous	tt0072308,tt0053137,tt0050419,tt0031983	dead
2	nm0000002	Lauren Bacall	1.924	2014	actress,soundtrack	tt0037382,tt0071877,tt0038355,tt0117057	dead
3	nm0000004	John Belushi	1.949	1982	actor,soundtrack,writer	tt0072562,tt0080455,tt0077975,tt0078723	dead
4	nm0000005	Ingmar Bergman	1.918	2007	writer,director,actor	tt0069467,tt0050976,tt0050986,tt0060827	dead
5	nm0000006	Ingrid Bergman	1.915	1982	actress,soundtrack,producer	tt0038787,tt0077711,tt0034583,tt0038109	dead
6	nm0000007	Humphrey Bogart	1.899	1957	actor,soundtrack,producer	tt0042593,tt0037382,tt0033870,tt0034583	dead
7	nm0000008	Marlon Brando	1.924	2004	actor,soundtrack,director	tt0047296,tt0068646,tt0078788,tt0070849	dead
8	nm0000009	Richard Burton	1.925	1984	actor,soundtrack,producer	tt0057877,tt0059749,tt0061184,tt0087803	dead
9	nm0000010	James Cagney	1.899	1986	actor,soundtrack,director	tt0031867,tt0035575,tt0042041,tt0029870	dead
10	nm0000011	Gary Cooper	1.901	1961	actor,soundtrack,producer	tt0027996,tt0044706,tt0035896,tt0034167	dead

Solution

Exercise IV:

3.1 Create table `imdb_movies_and_ratings_partitioned` partitioned by column `partition_year` using fields of table `title_basics` and `title_ratings`:

```
CREATE TABLE IF NOT EXISTS imdb_movies_and_ratings_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,
            average_rating DECIMAL(2,1),
            num_votes BIGINT
) PARTITIONED BY (partition_year int) STORED AS PARQUET LOCATION '/user/hadoop/imdb/
movies_and_ratings_partitioned';
```

Solution

Exercise IV:

3.2 Use dynamic partitioning to create and fill partition `partition_year`:

```
SET hive.exec.dynamic.partition.mode=nonstrict;
INSERT OVERWRITE TABLE imdb_movies_and_ratings_partitioned partition(partition_year)
SELECT
            tb.tconst,
            tb.title_type,
            tb.primary_title,
            tb.original_title,
            tb.is_adult,
            tb.start_year,
            tb.end_year,
            tb.runtime_minutes,
            tb.genres,
            tr.average_rating,
            tr.num_votes,
            tb.start_year
FROM title_basics tb JOIN title_ratings tr ON (tb.tconst = tr.tconst)
```

Solution

Exercise IV:

3.3 Check Results:

```
hadoop fs -ls /user/hadoop/imdb/movies_and_ratings_partitioned

[…]
drwxr-xr-x   - hadoop supergroup          0 2021-02-27 17:24 /user/hadoop/imdb/movies_and_ratings_partitioned/partition_year=1874
drwxr-xr-x   - hadoop supergroup          0 2021-02-27 17:24 /user/hadoop/imdb/movies_and_ratings_partitioned/partition_year=1878
drwxr-xr-x   - hadoop supergroup          0 2021-02-27 17:24 /user/hadoop/imdb/movies_and_ratings_partitioned/partition_year=1881
drwxr-xr-x   - hadoop supergroup          0 2021-02-27 17:24 /user/hadoop/imdb/movies_and_ratings_partitioned/partition_year=1883
drwxr-xr-x   - hadoop supergroup          0 2021-02-27 17:24 /user/hadoop/imdb/movies_and_ratings_partitioned/partition_year=1885
drwxr-xr-x   - hadoop supergroup          0 2021-02-27 17:24 /user/hadoop/imdb/movies_and_ratings_partitioned/partition_year=1887
drwxr-xr-x   - hadoop supergroup          0 2021-02-27 17:24 /user/hadoop/imdb/movies_and_ratings_partitioned/partition_year=1888
drwxr-xr-x   - hadoop supergroup          0 2021-02-27 17:24 /user/hadoop/imdb/movies_and_ratings_partitioned/partition_year=1889
drwxr-xr-x   - hadoop supergroup          0 2021-02-27 17:24 /user/hadoop/imdb/movies_and_ratings_partitioned/partition_year=1890
drwxr-xr-x   - hadoop supergroup          0 2021-02-27 17:24 /user/hadoop/imdb/movies_and_ratings_partitioned/partition_year=1891
drwxr-xr-x   - hadoop supergroup          0 2021-02-27 17:24 /user/hadoop/imdb/movies_and_ratings_partitioned/partition_year=1892
[…]
```

Solution

Exercise IV:

3.3 Check Results:

```
SELECT tconst, primary_title, start_year, genres, average_rating, num_votes
FROM imdb_movies_and_ratings_partitioned
WHERE partition_year = 2020 AND title_type = 'movie' AND num_votes > 25000 ORDER BY average_rating DESC LIMIT 100
```

Result

SELECT tconst, primary_title, start_year, genres, average_ | Geben Sie einen SQL-Ausdruck ein, um die Ergebnisse zu filtern (verwenden Sie Strg+ Leertaste).

	tconst	primary_title	start_year	genres	average_rating	num_votes
1	tt10189514	Soorarai Pottru	2.020	Drama	8,6	55.837
2	tt8503618	Hamilton	2.020	Biography,Drama,History	8,5	57.348
3	tt2948372	Soul	2.020	Adventure,Animation,Comedy	8,1	179.805
4	tt8110330	Dil Bechara	2.020	Comedy,Drama,Romance	7,9	111.891
5	tt1070874	The Trial of the Chicago 7	2.020	Drama,History,Thriller	7,8	98.179
6	tt10288566	Another Round	2.020	Comedy,Drama	7,8	40.297
7	tt11464826	The Social Dilemma	2.020	Documentary,Drama	7,7	65.207
8	tt7212754	Ludo	2.020	Action,Comedy,Crime	7,6	28.350
9	tt6723592	Tenet	2.020	Action,Sci-Fi,Thriller	7,5	296.730
10	tt9620292	Promising Young Woman	2.020	Crime,Drama,Thriller	7,5	34.475
11	tt7146812	Onward	2.020	Adventure,Animation,Comedy	7,4	102.711
12	tt9484998	Palm Springs	2.020	Comedy,Fantasy,Mystery	7,4	81.009
13	tt10618286	Mank	2.020	Biography,Comedy,Drama	7,1	38.884
14	tt9686708	The King of Staten Island	2.020	Comedy,Drama	7,1	39.349
15	tt7395114	The Devil All the Time	2.020	Crime,Drama,Thriller	7,1	93.645
16	tt1051906	The Invisible Man	2.020	Drama,Horror,Mystery	7,1	169.172

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).

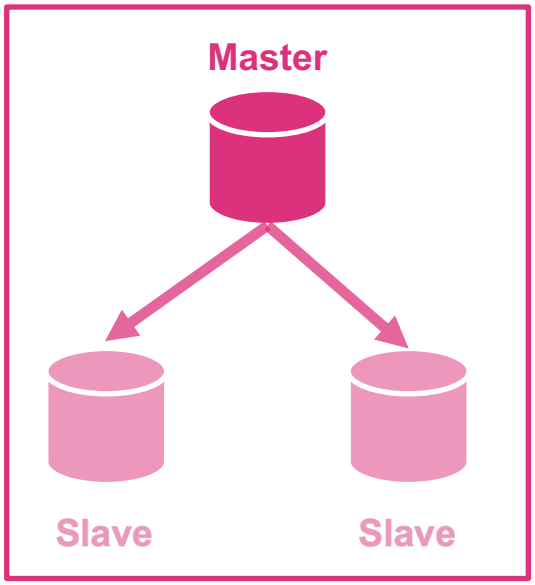
Replication vs Partitioning

	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

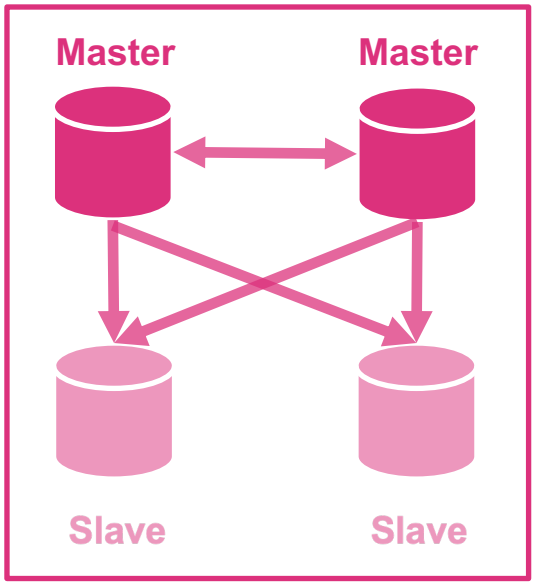
Approaches For Replication

Master-based



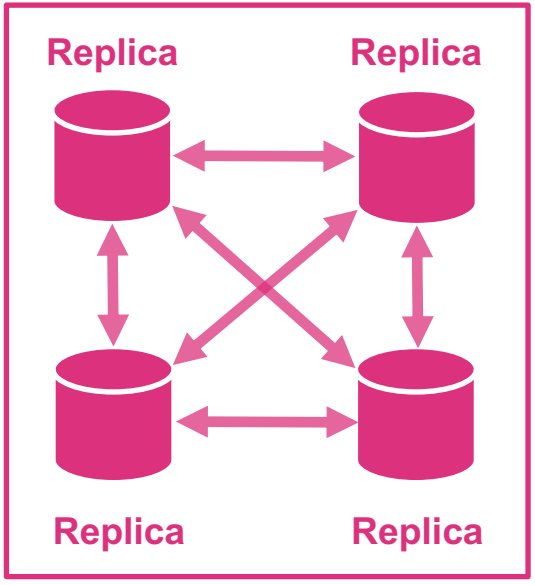
e.g. **Oracle, PostgreSQL, MySQL…**

Multi-Master-based



e.g. **CouchDB, PostgreSQL, MySQL…**

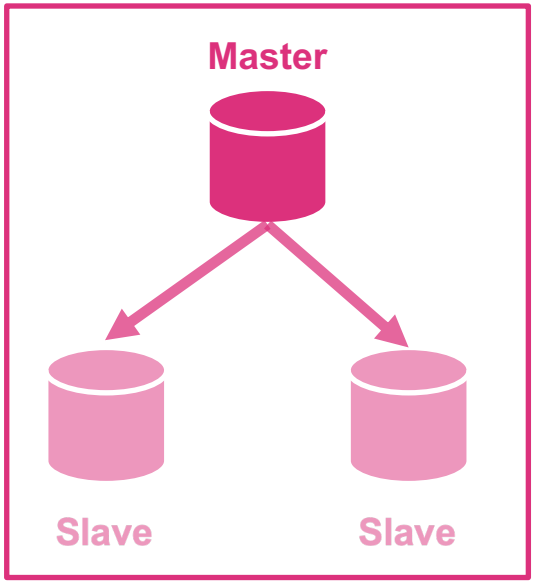
Masterless



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

Master Replication

Master-based



e.g. **Oracle, PostgreSQL, MySQL…**

Multi-Master-based

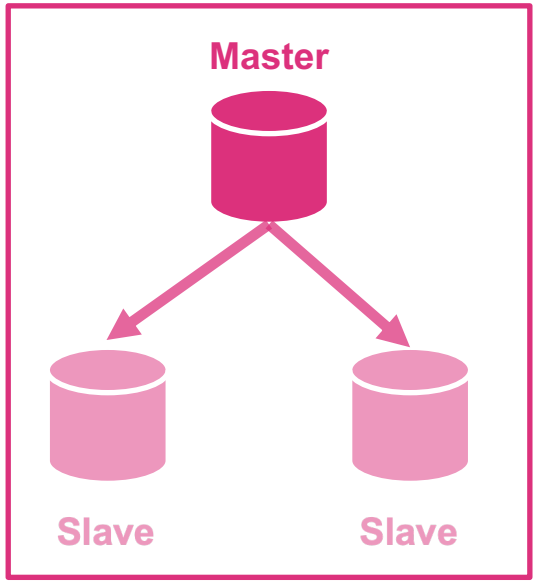
e.g. **CouchDB, PostgreSQL, MySQL…**

Masterless

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

Master Replication

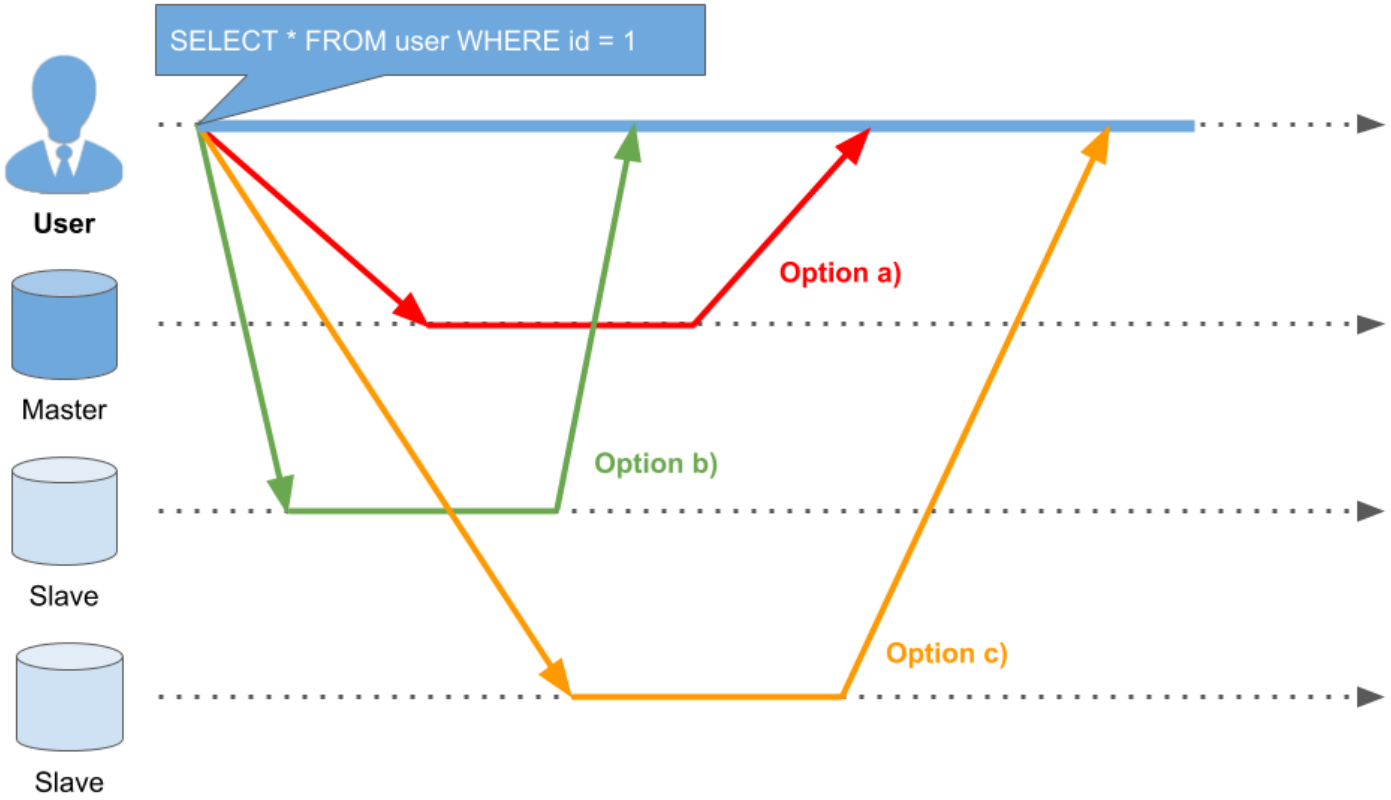
Also known as: *Primary/Secondary, Master/Slave or Single-Leader Replication*



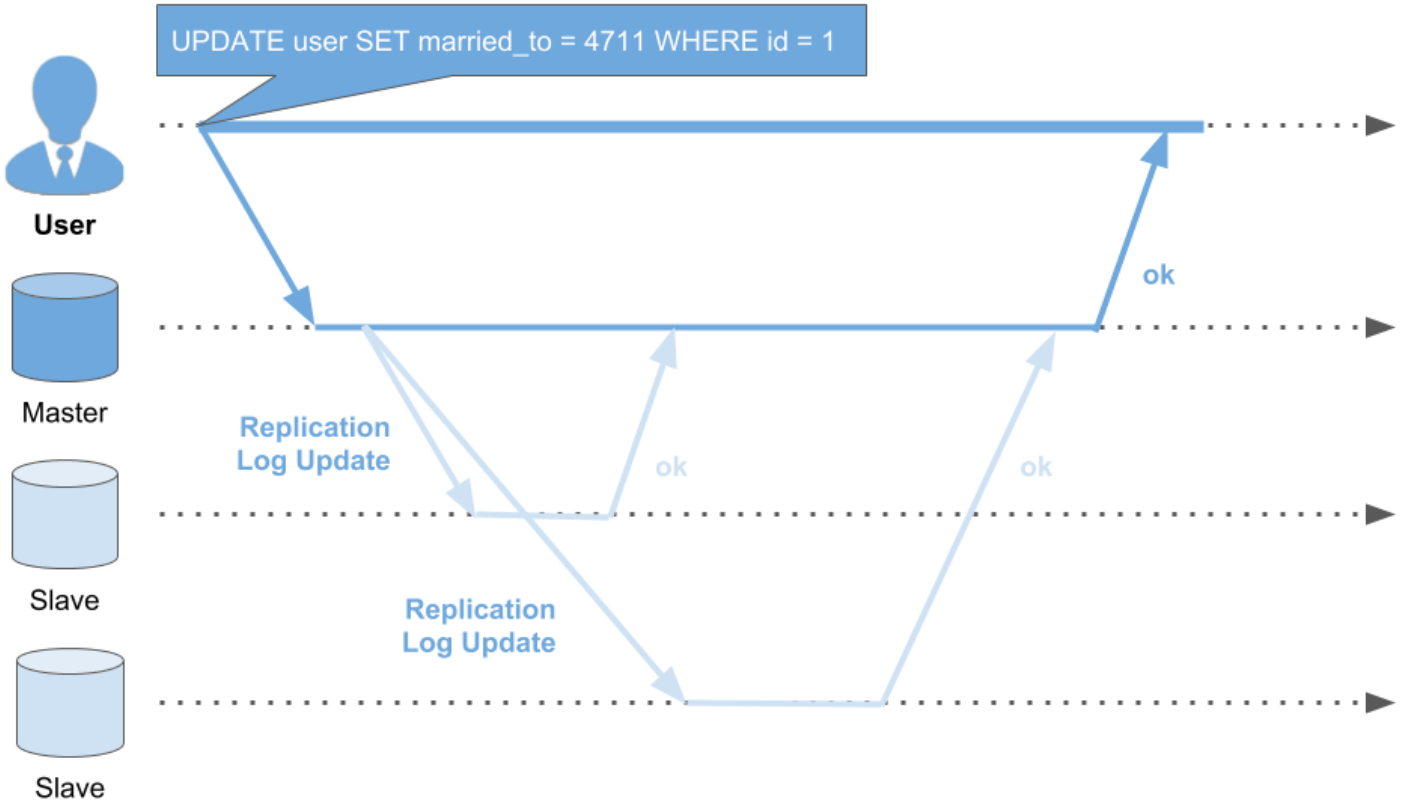
Master: (also known as *leader* or *primary*) dedicated processing node, usually also a replica and also known as primary or leader. The master node serves read as well as write queries, is responsible for persisting changes locally and propagates changes (e.g. by using replication logs or change streams)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

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

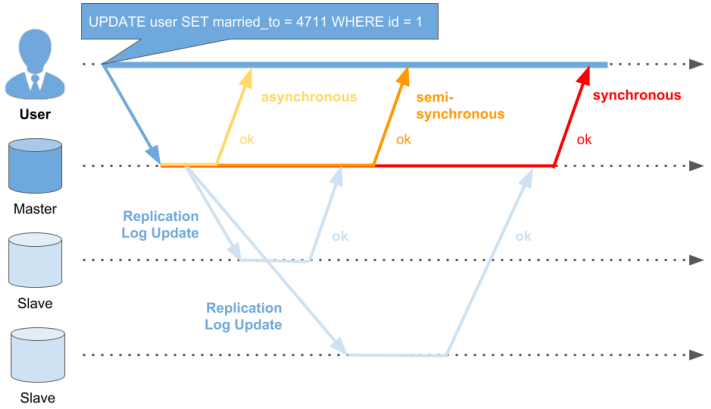
Master Replication - **Read**



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

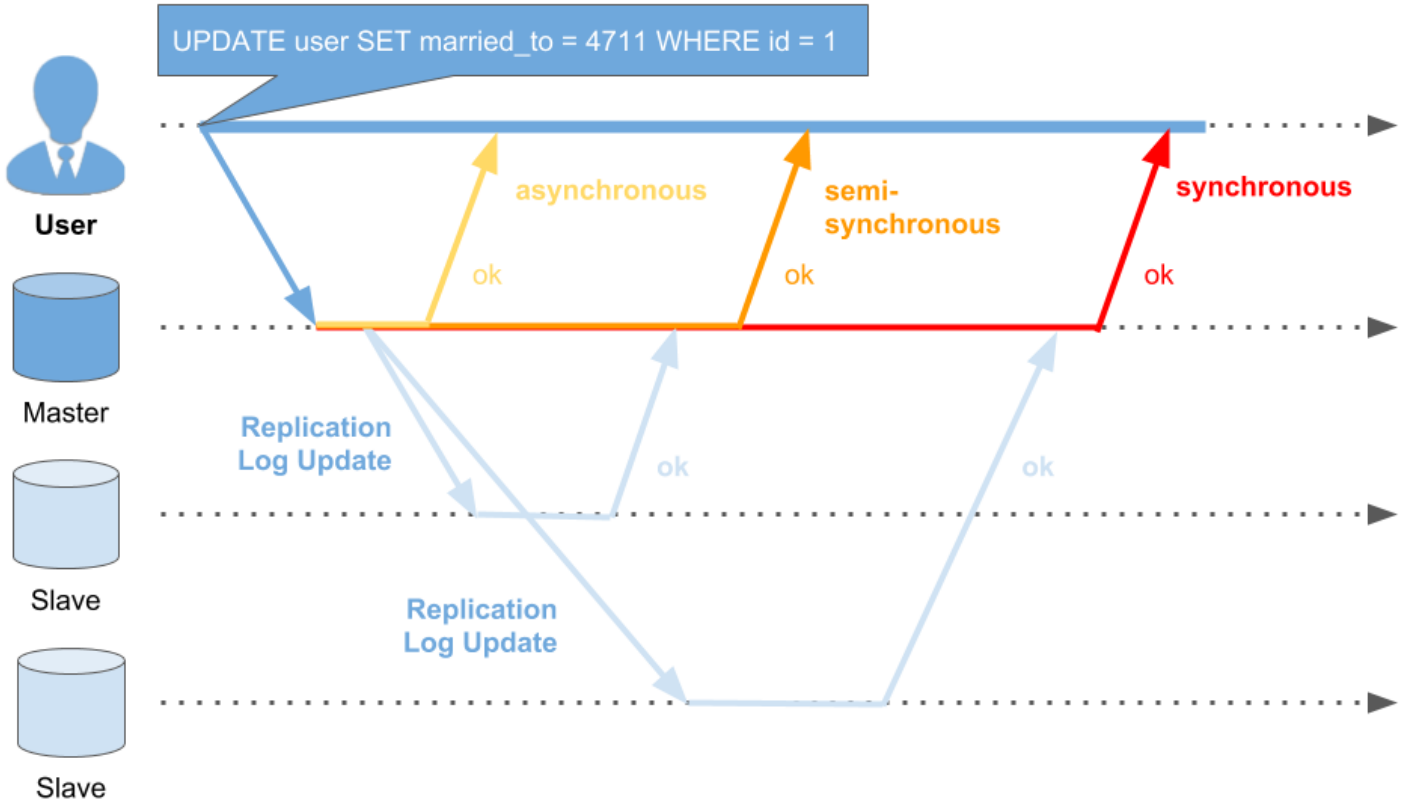
→ one slave runs synchronous to master, all other slaves asynchronous
→ If synchronous slave gets slow or crashes, another slave becomes synchronous
→ it is guaranteed, that at least two nodes store a replica which is up-to-date

3. Asynchronous:

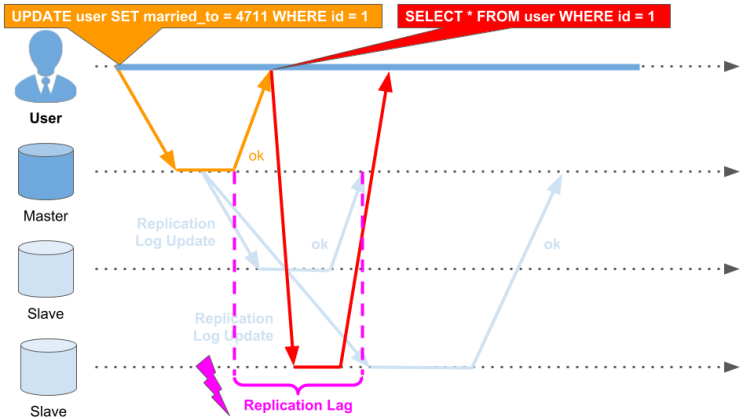
- the **master does not wait for any slaves** to report success, before reporting success to a user

→ 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

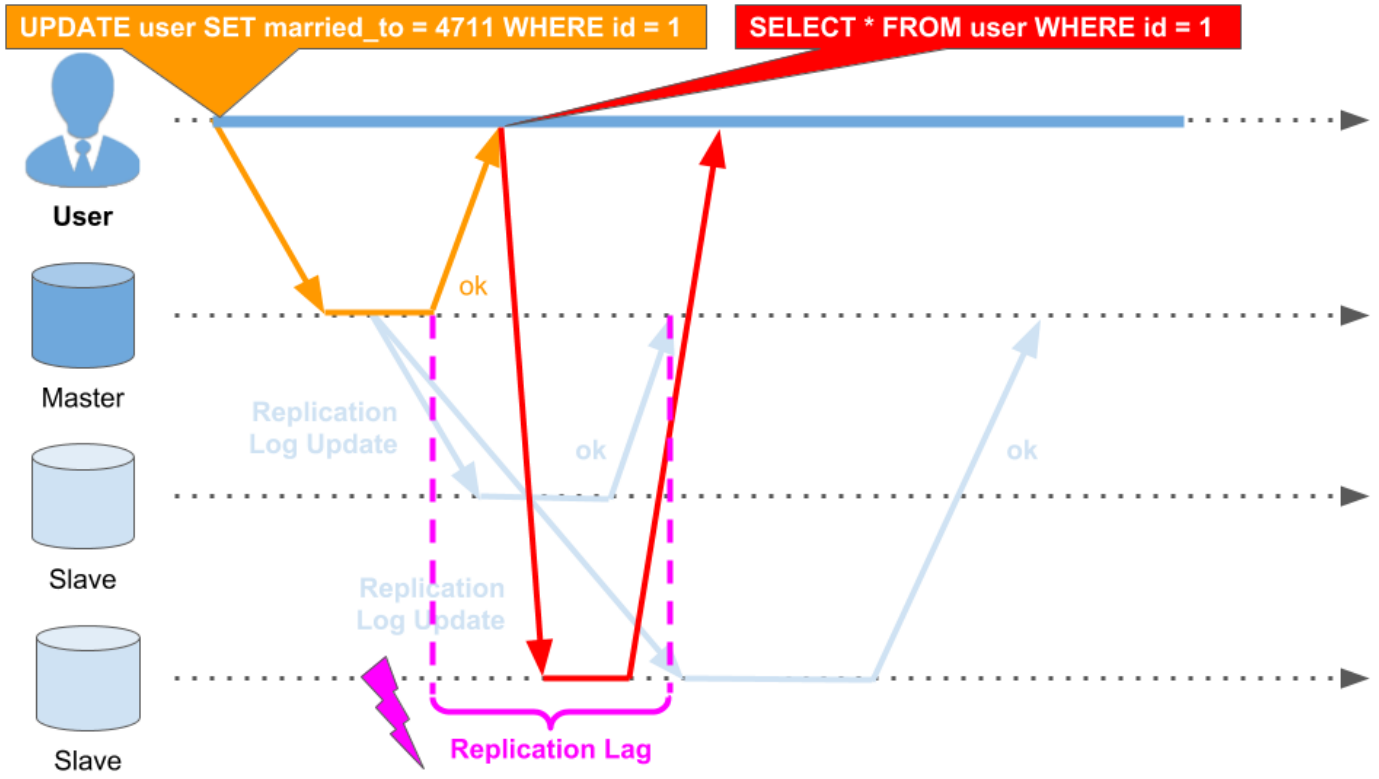
→ Data locality = put slaves geographically close to user/client-applications

→ But: **vulerable to replication lag** (timeframe when master and slaves are inconsistent, as the slaves have not yet processed the most recent write requests)

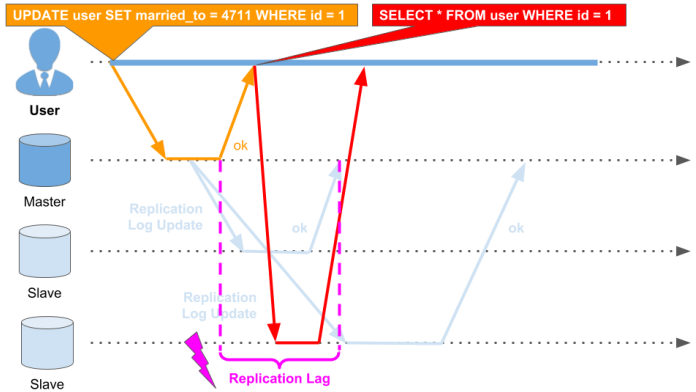
→ **replication lag** = **temporary incosistency** (usually just fractions of seconds)

- 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

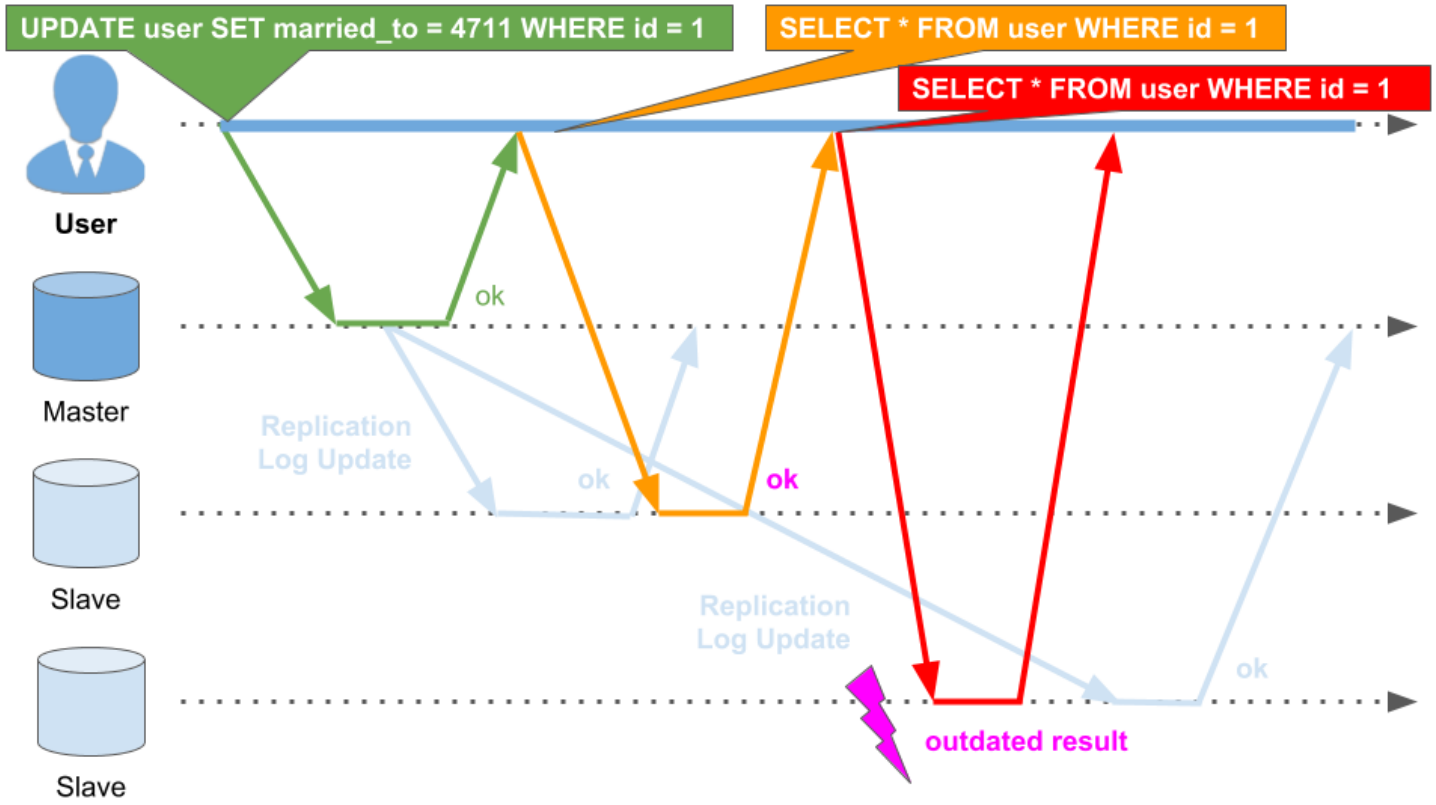


Read-Your-Writes Consistency:
ensures that any write requests submitted by a user will directly be seen on any further read requests (by the same user, guaranteeing a user his write request has been processed successfully.

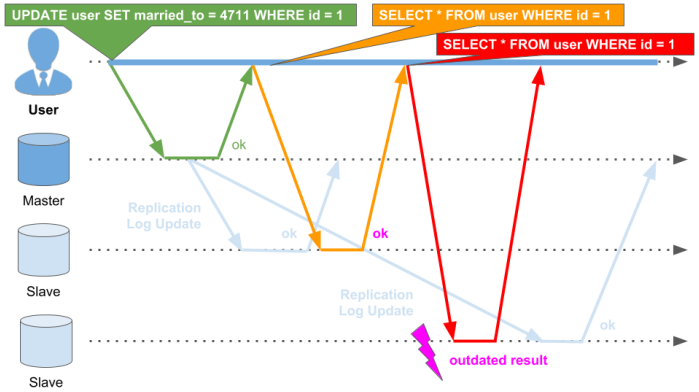
Approaches for achieving Read-Your-Writes consistency:

- **Read data**, a **user may have modified**, **from the master**, otherwise make use of a slave.
  E.g. Xing/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 on 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**.

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 **replication 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**, otherwise they will end up in different results.

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

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 to 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 statement 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 impossible
  - → increases complexity of maintenance tasks, especially rolling-upgrades of single nodes 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

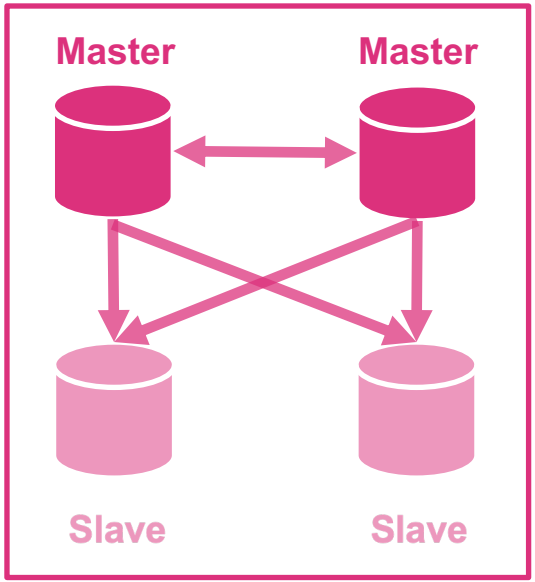
```
$ use local
switched to db local
$ db.oplog.rs.find()
{
  "ts" : { "t" : 1534616696000, "i" : 1 },
  "h" : NumberLong("1342870845645633201"),
  "op" : "i",
  "ns" : "test.user",
  "o" : {
    "_id" : ObjectId("4cb35859543cc1f4f9f7f85d"),
    "user_id" : 1
  }
}
{
  "ts" : { "t" : 1534616699000, "i"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



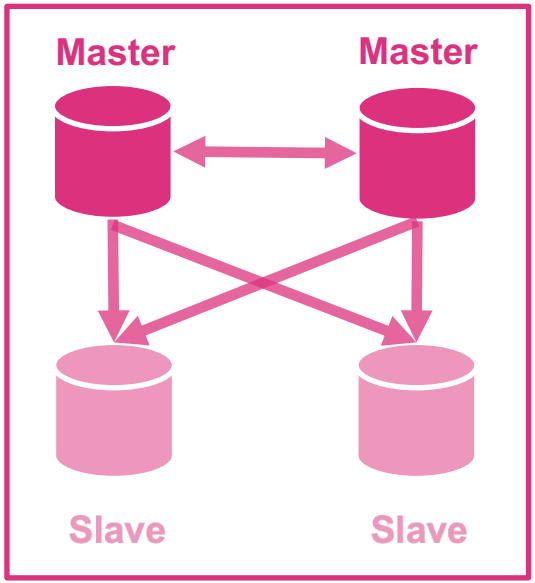
e.g. **CouchDB, PostgreSQL, MySQL…**

Masterless

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

Multi-Master Replication

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

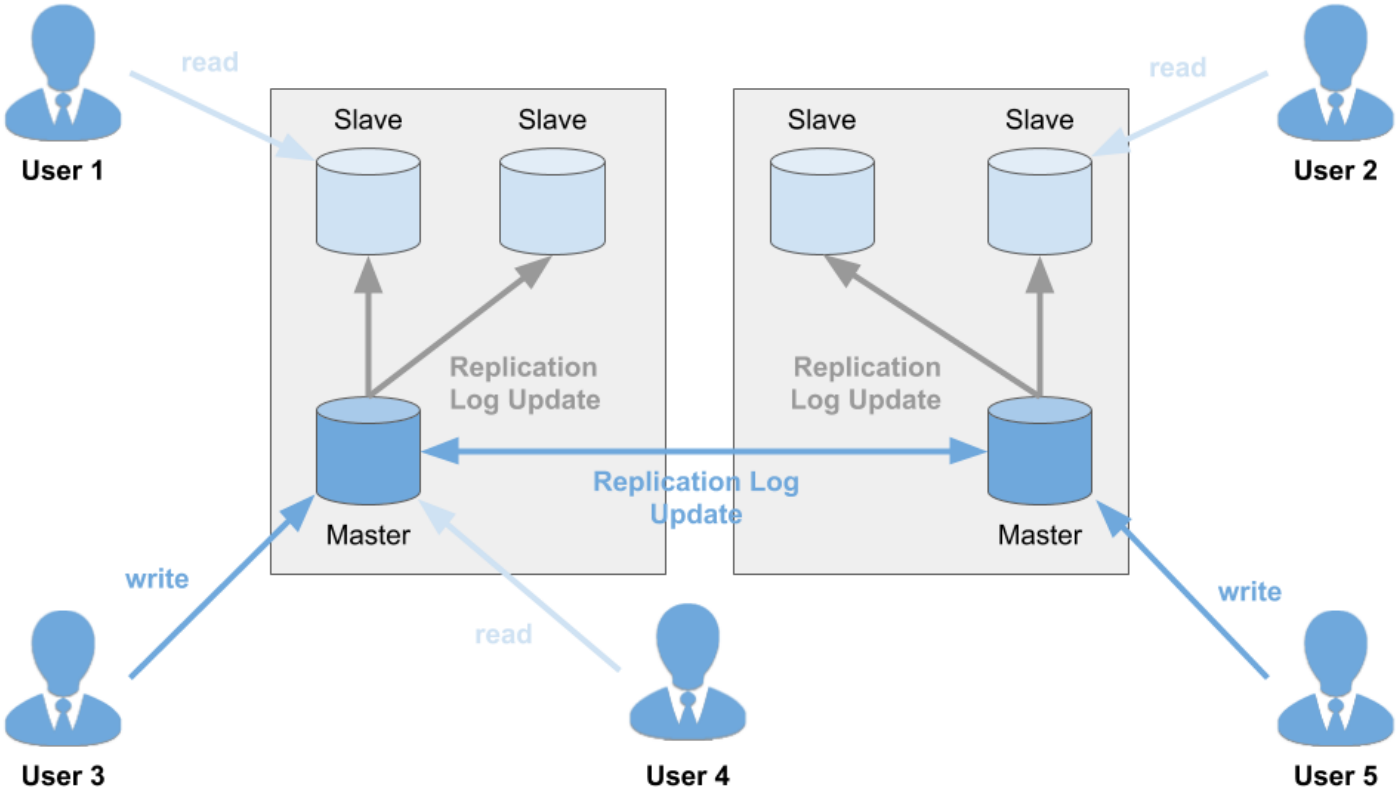


Setup:
- **multiple master** nodes (which accept write requests)
- query and replication processing similar to master-based replication

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

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:**

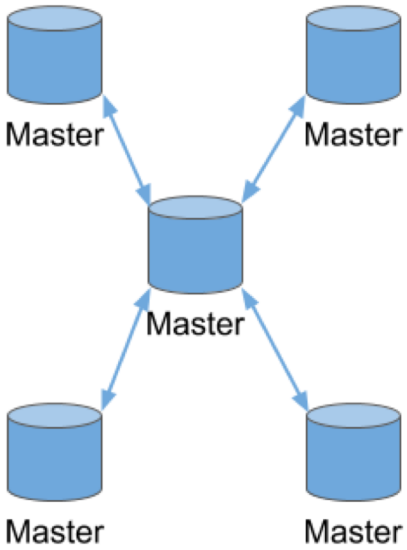
LWW: only accept the most recent operation by e.g. unique operation ID, timestamp or both (*Last-Write-Wins*). It is important to notice that this approach is highly vulnerable to data loss.

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

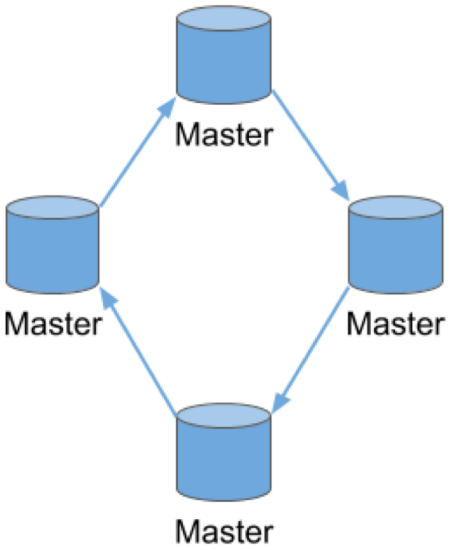
Application Managed: persist the conflict in a way, that it preserves all information without loss and let the application, using the data-system, or rather the user of the application take care about it later on (e.g. implemented by SVN or Git)

Multi-Master Replication - **Topologies**

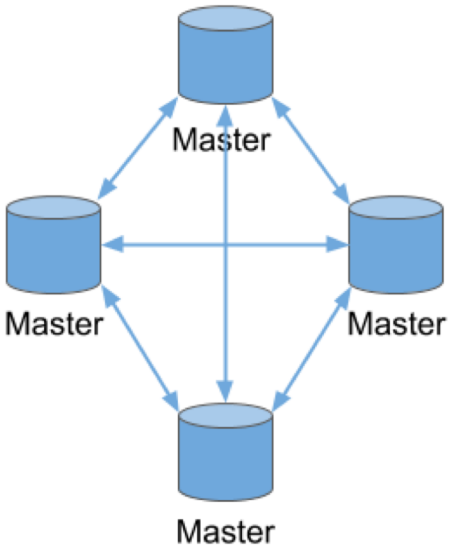
Star Topology



Circle Topology



All-To-All Topology



Masterless Replication

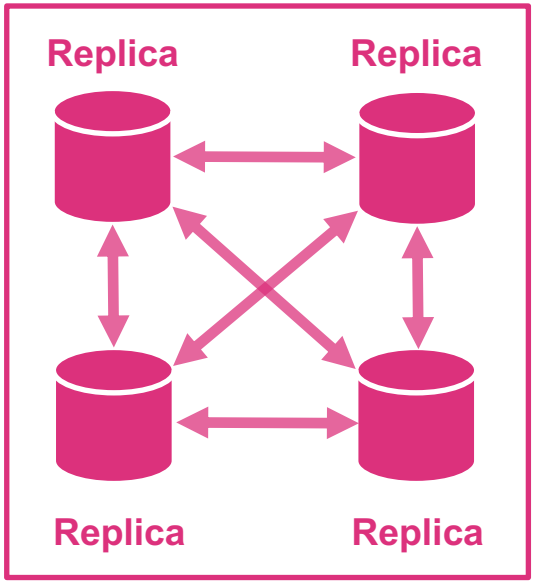
Master-based

e.g. **Oracle, PostgreSQL, MySQL…**

Multi-Master-based

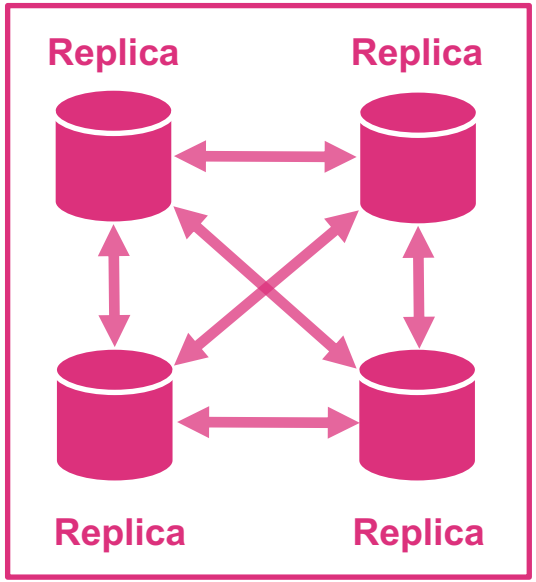
e.g. **CouchDB, PostgreSQL, MySQL…**

Masterless



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

Masterless Replication



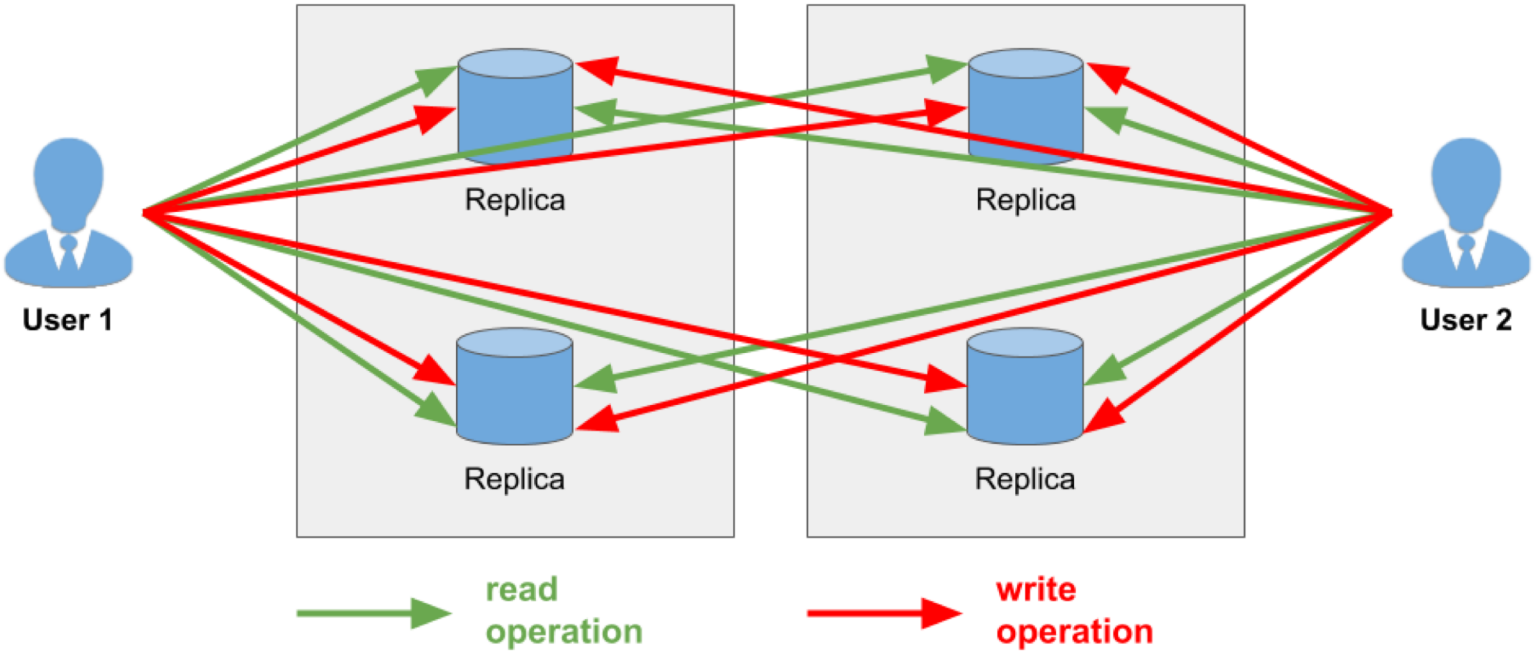
Also known as:
Leaderless Replication

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 result, as at least one of the replica will have the most recent value

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

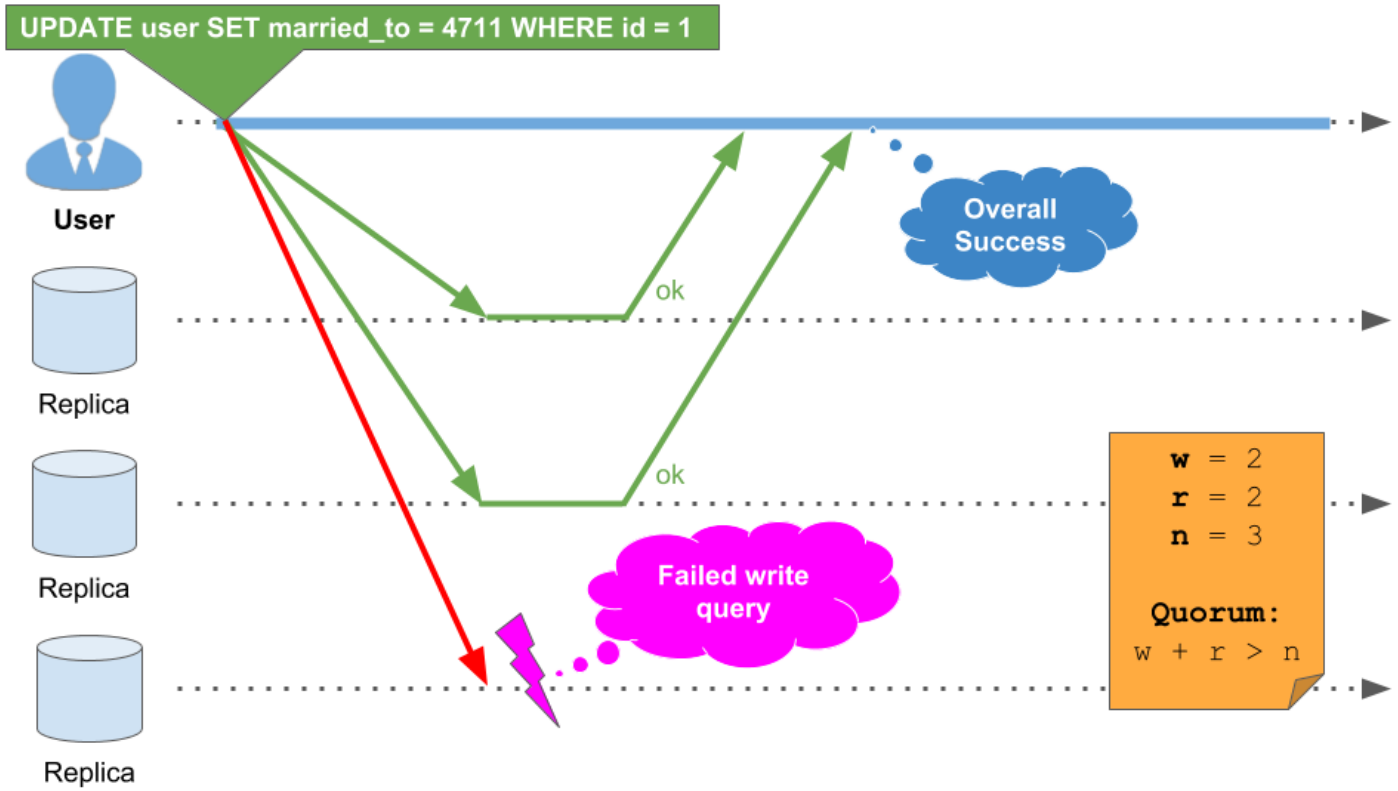
Masterless Replication- **Quorums**

- **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

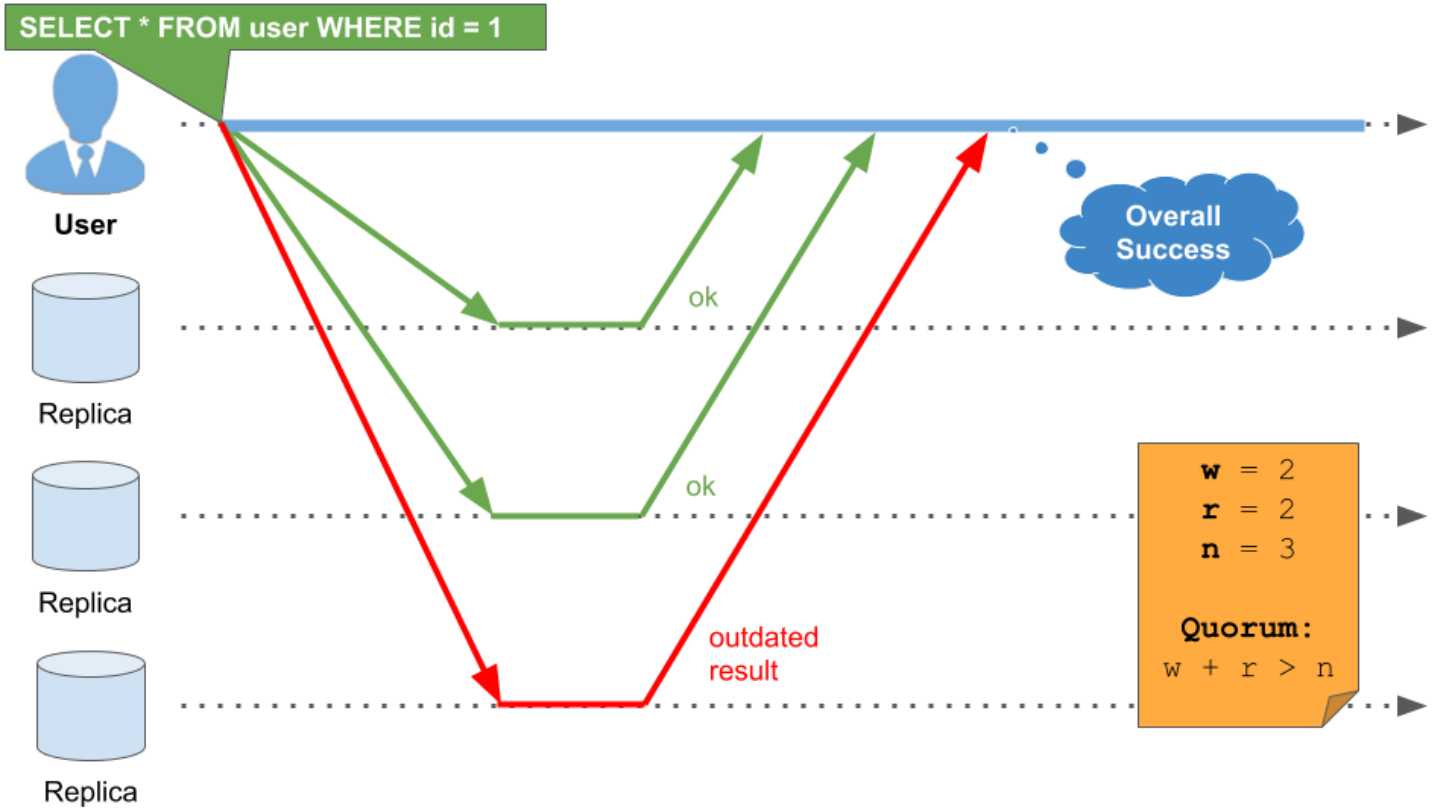
- Number of node failures a data-system can handle, is calculated by:
  - → **n - r = number** of nodes tolerated to be unavailable for read requests
  - → **n - w = number** of nodes tolerated to be unavailable for write requests

- 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.

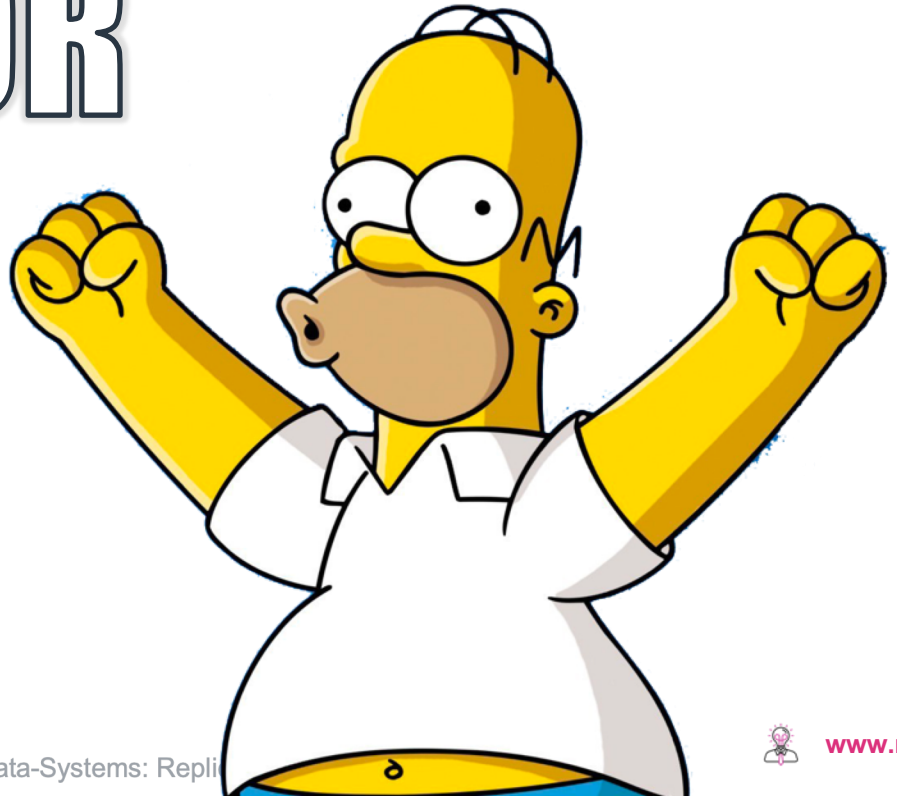
Node Failure: If a node previously processed a new value, crashed and comes back online and is **restored** from a node with the **old value**, the **quorum might be violated** as the number of nodes storing the new value might be < w.

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*.

Break

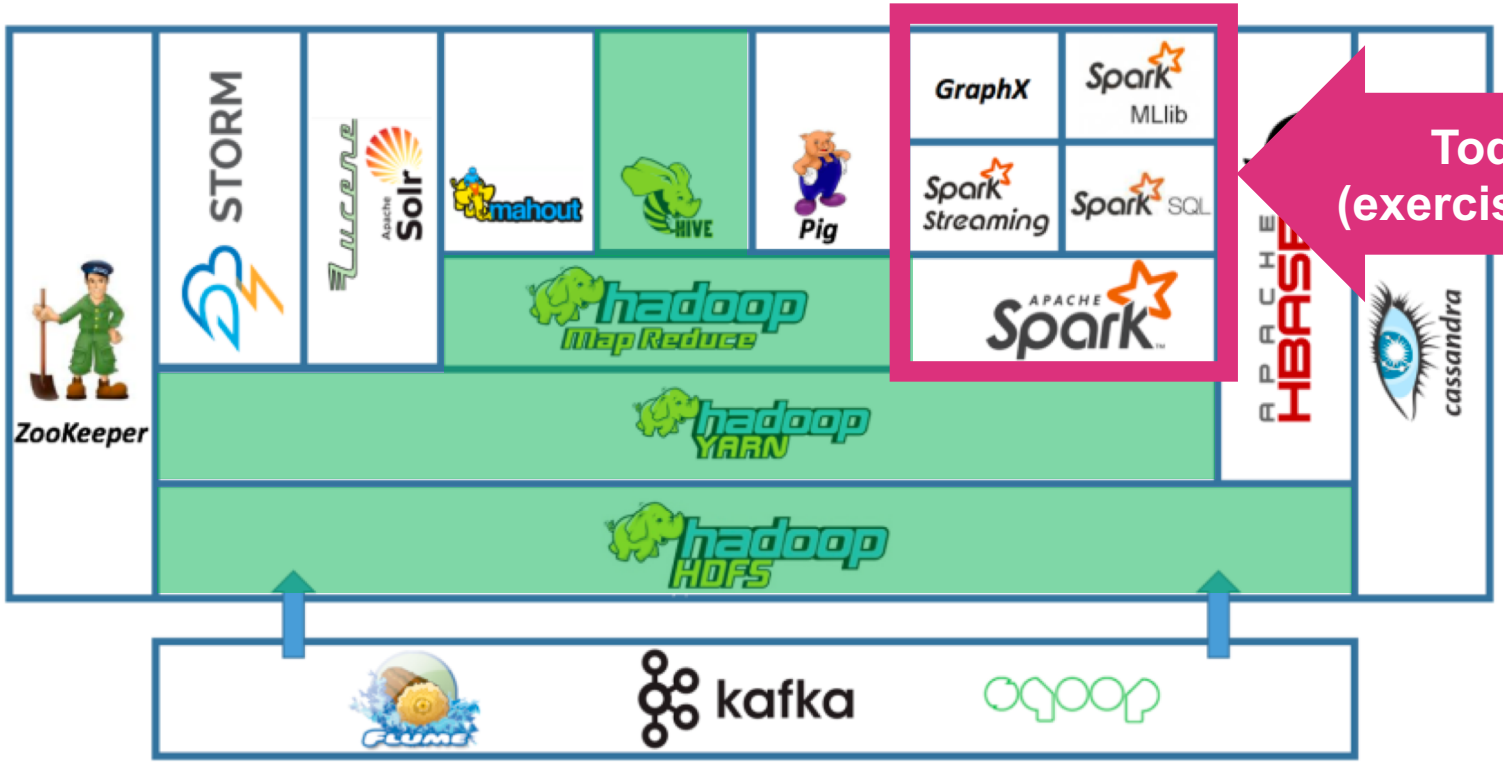
TIME FOR A BREAK



HandsOn – **Spark, Scala, PySpark and Jupyter**

A quick Introduction to Spark, Scala, PySpark and Jupyter

The Hadoop Ecosystem



Spark

API Languages: Java Scala Python R

APIs and Libraries: Spark SQL | Spark Streaming | Mlib | GraphX

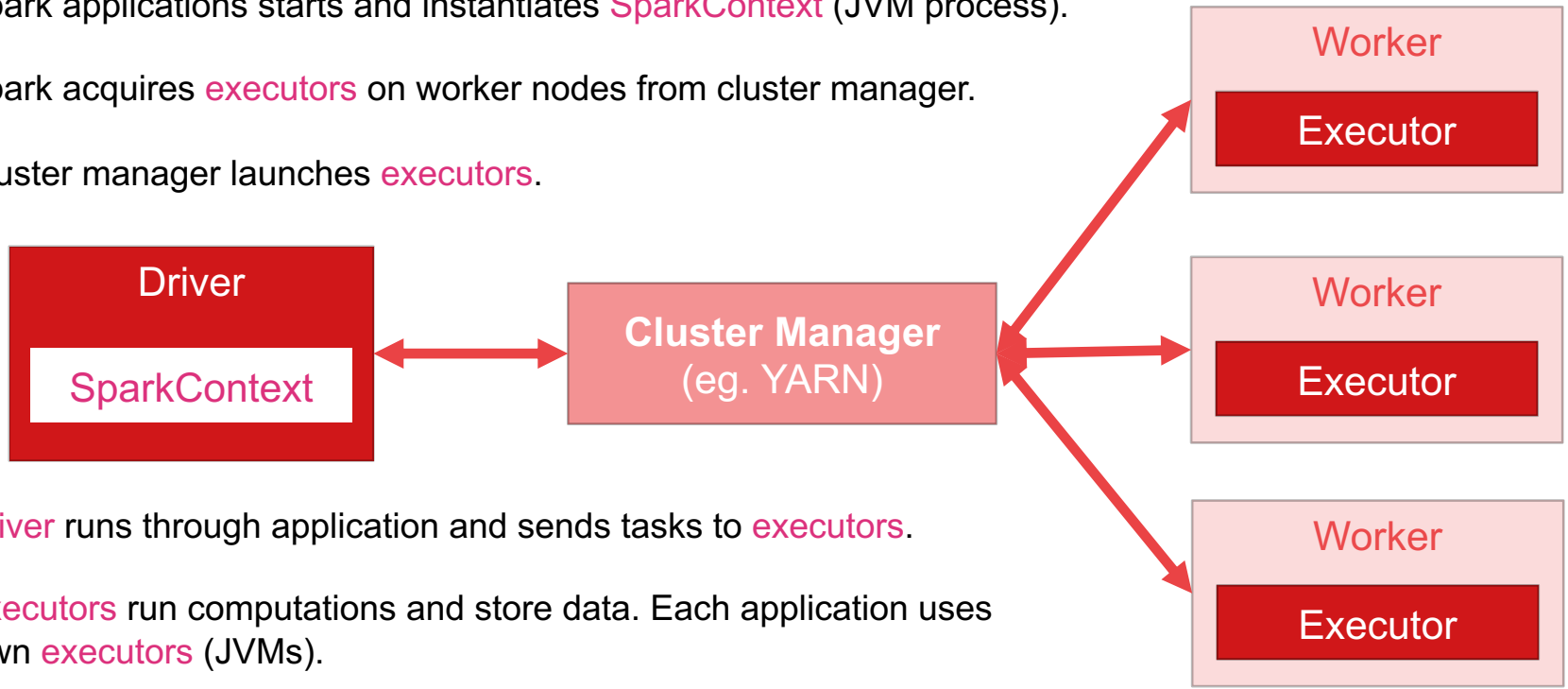
Spark Core

Ressource/ Cluster Manager: Standalone | YARN | Mesos | Kubernetes

Data Source APIs: HDFS, ElasticSearch, Amazon S3, Azure Data Lake, Redis, Riak, Couch-base, MongoDB, Hive, Avro, CSV, Parquet, JDBC DB…

Spark – Execution Process

1. Spark applications starts and instantiates SparkContext (JVM process).
2. Spark acquires executors on worker nodes from cluster manager.
3. Cluster manager launches executors.



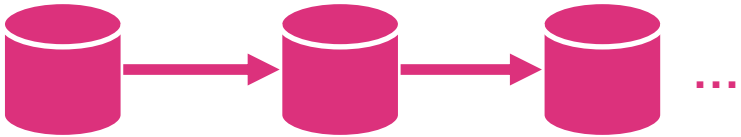
4. Driver runs through application and sends tasks to executors.
5. Executors run computations and store data. Each application uses its own executors (JVMs).
6. If any worker crashes, ist taks will be send to another executor.

Spark – RDDs, DataFrame and DataSet

	RDD (2011)	DataFrame (2013)	DataSet (2015)
Supported by:	Scala, Java, Python	Scala, Java, Python	Scala, Java
Idea:	- RDD = immutable distributed collection of data - partitioned across nodes of cluster - can be operated in parallel with a low-level API	- based on RDD - immutable distributed collection of data - but organized into columns - higher level abstraction	- based on DataFrame API, but type-safe - immutable distributed collection of data - high-level API - converted into optimized RDD
Typed:	typed, no schema	untyped, schema	typed, schema
Use for:	unstructured data	semi-structured and structured data	structured data

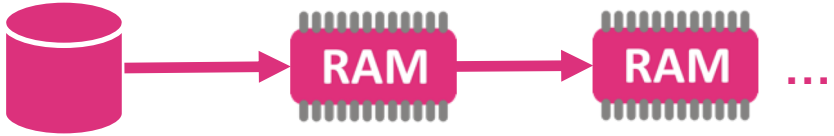
Hadoop MapReduce vs Spark

Hadoop MapReduce



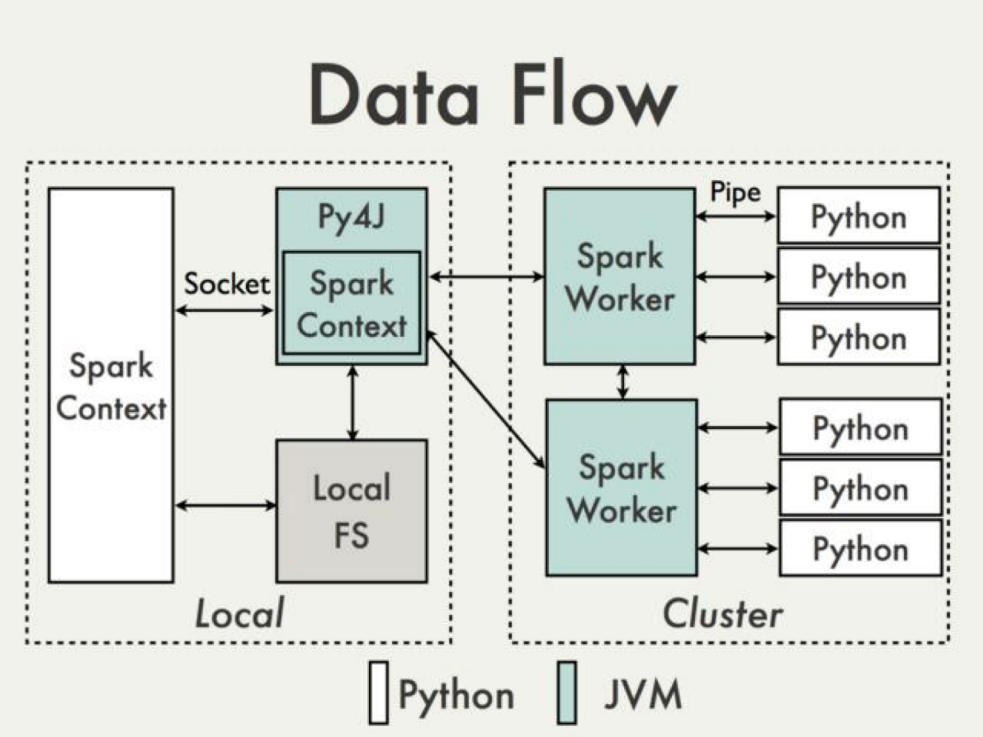
- performs read from HDFS (HDD) before every computation
- performs write on HDFS (HDD) after every computation
- using distributed disk space

Spark



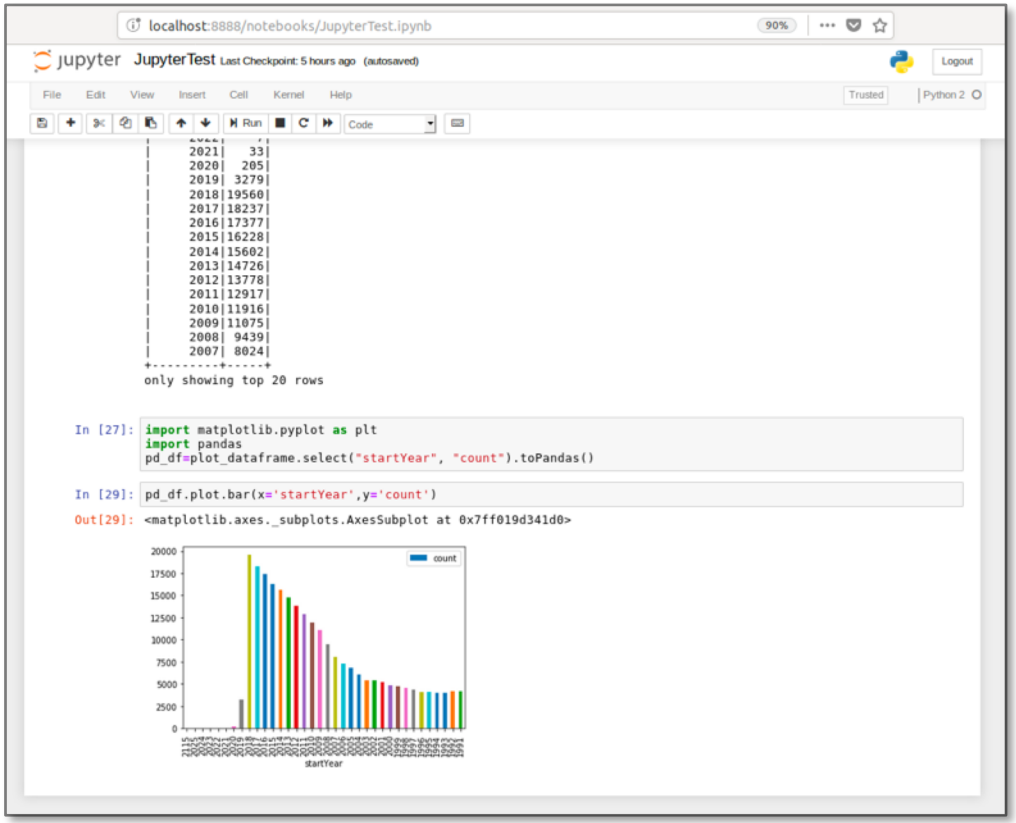
- performs read from RAM before e very computation (except first)
- performs write on RAM after every computation
- using distributed **memory**

PySpark



- built on-top of Sparks Java API
- data is processed in Python and cached/ shuffled within the JVM
- Spark executors on the cluster start Python interpreter to execute user code
- A Python RDD corrresponds to an RDD in
- the local JVM
- e.g. **sc.textFile()** in Python will call JavaSparkContext **textFile()**

Jupyter (Notebooks)



- Interactive Web IDE
- Kernel-based Notebooks
- Open-source
- Create and share:
  - code,
  - visualizations and
  - narrative text like documentation
- Supports:
  - Python
  - R
  - Scala
  - …
- Works well with Spark

Exercises Preparation I

Start Hadoop Cluster and Test Spark Shell
(Word Count Example)

Start Gcloud VM and Connect

1. Start Gcloud Instance:

```
gcloud compute instances start big-data
```

2. Connect to Gcloud instance via SSH (on Windows using Putty):

```
ssh hans.wurst@XXX.XXX.XXX.XXX
```


Pull and Start Docker Container

1. Pull Docker Image:

```
docker pull marcelmittelstaedt/spark_base:latest
```

2. Start Docker Image:

```
docker network create --driver bridge bigdatanet
docker run -dit --name hadoop \
          -p 8088:8088 -p 9870:9870 -p 9864:9864 -p 10000:10000 \
          -p 8032:8032 -p 8030:8030 -p 8031:8031 -p 9000:9000 \
          -p 8888:8888 --net bigdatanet \
          marcelmittelstaedt/spark_base:latest
```

3. Wait till first Container Initialization finished:

```
docker logs hadoop

[…]
Stopping nodemanagers
Stopping resourcemanager
Container Startup finished.
```


Start Hadoop Cluster

1. Get into Docker container:

```
docker exec -it hadoop bash
```

2. Switch to hadoop user:

```
sudo su hadoop
```

```
cd
```

3. Start Hadoop Cluster:

```
start-all.sh
```


Add Test text file to HDFS (Faust 1)

1. Download test Text File (Faust_1.txt):

```
wget https://raw.githubusercontent.com/marcelmittelstaedt/BigData/master/exercises/winter_semester_2020-2021/01_hadoop/sample_data/Faust_1.txt
```

2. Upload file to HDFS:

```
hadoop fs -put Faust_1.txt /user/hadoop/Faust_1.txt
```


Start Spark (on Yarn)

1. Start Spark Shell:

```
spark-shell --master yarn
```

```
Spark context Web UI available at http://localhost:4040
Spark context available as 'sc' (master = yarn, app id = application_1572177196643_0001).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.3.4
      /_/

Using Scala version 2.11.8 (OpenJDK 64-Bit Server VM, Java 1.8.0_222)
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```


Start Spark – WordCount Example (Scala)

1. Execute Word Count Example in Scala:

```
scala> val text_file = sc.textFile("/user/hadoop/Faust_1.txt")
scala> val words = text_file.flatMap(line => line.split(" "))
scala> val counts = words.map(word => (word, 1))
scala> val reduced_counts = counts.reduceByKey((count1, count2) => count1 + count2)
scala> val sorted_counts = reduced_counts.sortBy(- _._2)
```

```
scala> sorted_counts.take(10)

res0: Array[(String, Int)] = Array(("",1603), (und,509), (die,463), (der,440), (ich,435), (Und,
400), (nicht,346), (zu,319), (ist,291), (ein,284))
```

2. Save results to HDFS:

```
scala> sorted_counts.saveAsTextFile("/user/hadoop/Faust_1_WordCounts_Scala.txt")
```


Start Spark – WordCount Example (Scala)

3. Get results from HDFS to local filesystem:

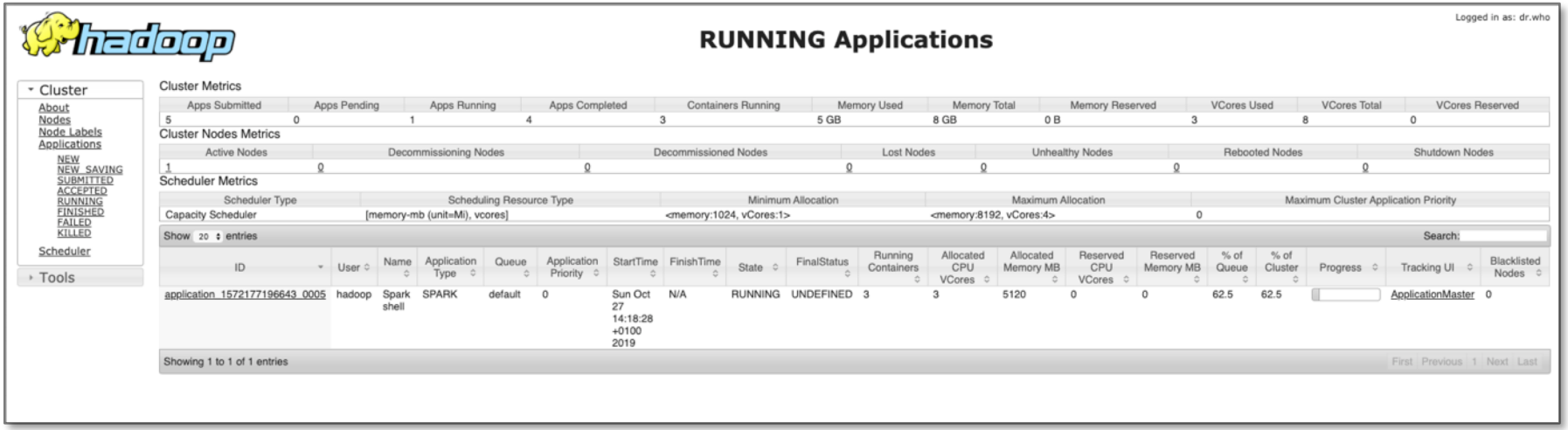
```
hadoop fs -get /user/hadoop/Faust_1_WordCounts_Scala.txt/part-00000 Faust_1_WordCounts_Scala.txt
```

4. Check Result:

```
head -10 Faust_1_WordCounts_Scala.txt

(,1603)
(und,509)
(die,463)
(der,440)
(ich,435)
(Und,400)
(nicht,346)
(zu,319)
(ist,291)
(ein,284)
```

Start Spark (on Yarn) – WordCount Example

5. See Spark Shell Container Running on Yarn http://xxx.xxx.xxx.xxx:8088/cluster :



Exercises Preparation II

Test PySpark Shell (Word Count Example)

Start PySpark (on Yarn) – Test Install

1. As PySpark is already installed, start PySpark Shell and execute previous example as Python code:

```
pyspark --master yarn
```

```
Python 3.6.8 (default, Oct  7 2019, 12:59:55)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 2.3.4
      /_/

Using Python version 3.6.8 (default, Oct  7 2019 12:59:55)
SparkSession available as 'spark'.
>>>
```

PySpark – WordCount Example (Python)

1. Execute Word Count Example in Python:

```
>>> text_file = spark.read.text("/user/hadoop/Faust_1.txt").rdd.map(lambda r: r[0])
>>> words = text_file.flatMap(lambda line: line.split(" "))
>>> counts = words.map(lambda word: (word, 1)).reduceByKey(lambda a,b:a +b)
>>> output = counts.collect()
>>> sorted_output = sorted(output,key=lambda x:(-x[1],x[0]))
```

```
>>> sorted_output[:10]

[('', 1603), ('und', 509), ('die', 463), ('der', 440), ('ich', 435), ('Und', 400), ('nicht', 34
6), ('zu', 319), ('ist', 291), ('ein', 284)]
```

2. Save results to HDFS:

```
>>> counts.saveAsTextFile("/user/hadoop/Faust_1_WordCounts_Python.txt")
```


PySpark – WordCount Example (Python)

3. Get results from HDFS to local filesystem:

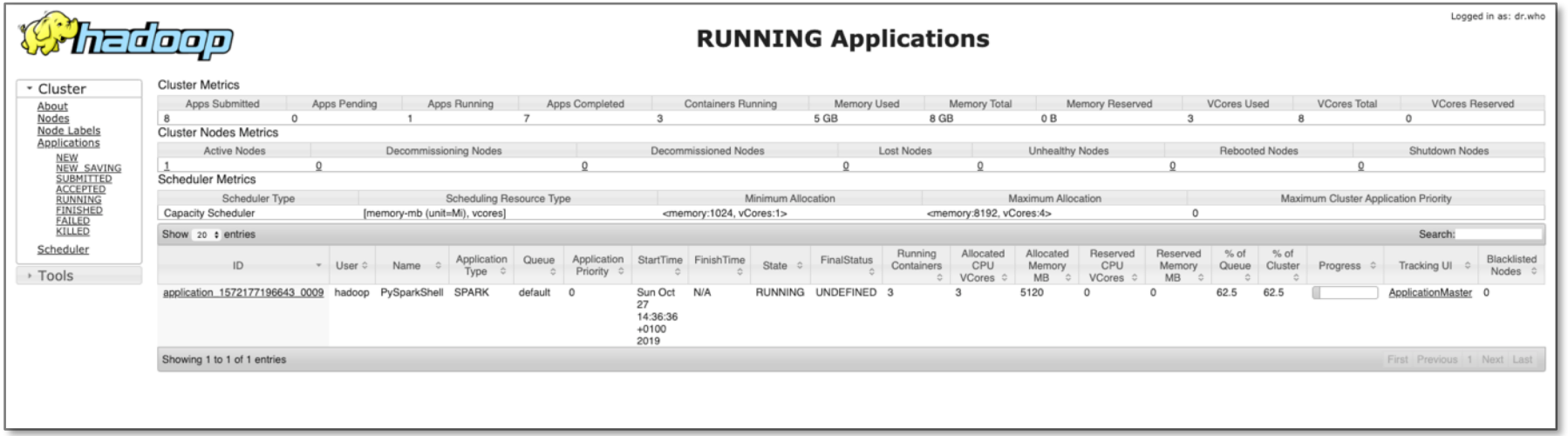
```
hadoop fs -get /user/hadoop/Faust_1_WordCounts_Python.txt/part-00000 Faust_1_WordCounts_Python.txt
```

4. Check Result:

```
head -10 Faust_1_WordCounts_Python.txt

('Johann', 1)
('Wolfgang', 1)
('von', 133)
('Goethe:', 1)
('Faust,', 8)
('Der', 130)
('Tragödie', 1)
('erster', 2)
('Teil', 6)
('', 1603)
```

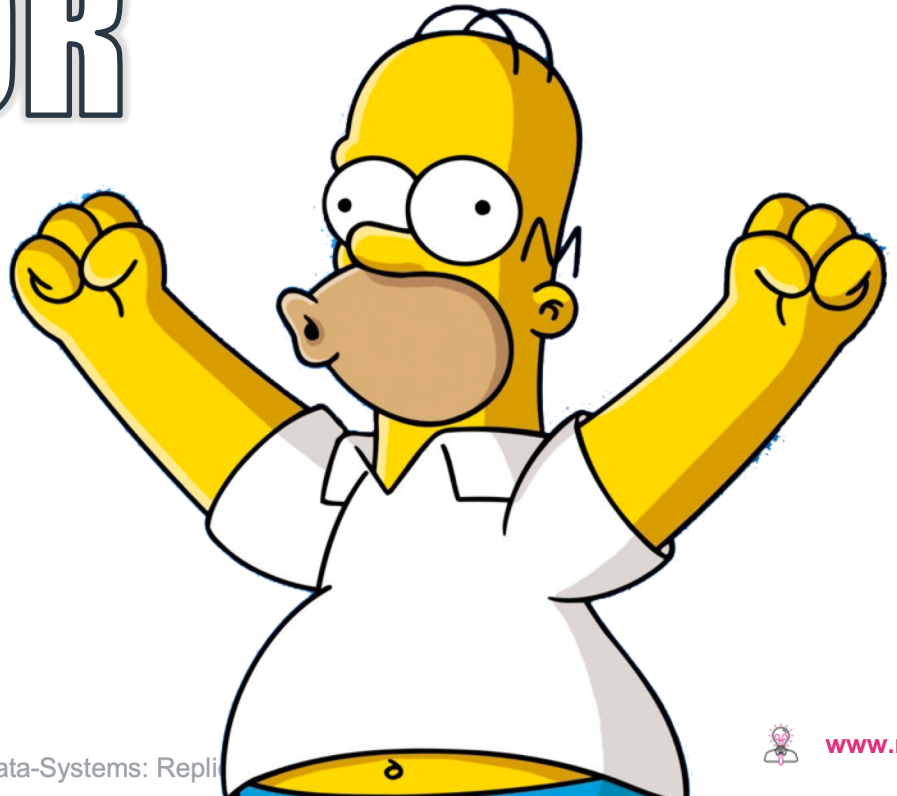
PySpark (on Yarn) – WordCount Example

5. See PySpark Shell Container Running on Yarn http://xxx.xxx.xxx.xxx:8088/cluster :



Break

TIME FOR A BREAK



Exercises Preparation III

Start and work with Jupyter Notebooks
(on PySpark)

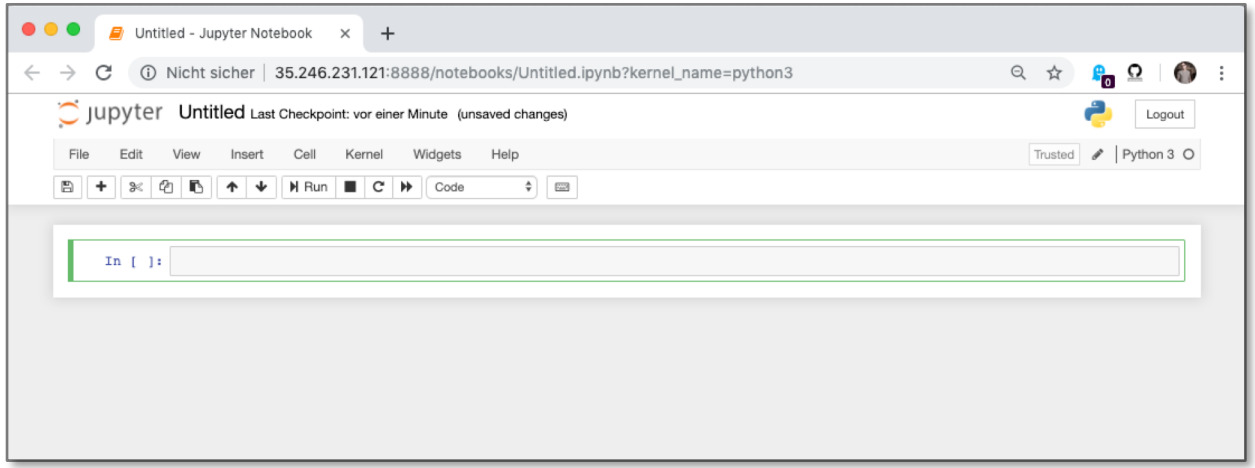
Start Jupyter

1. Start Jupyter Notebook

```
jupyter notebook

[I 14:02:39.790 NotebookApp] Writing notebook server cookie secret to /home/hadoop/.local/share/jupyter/runtime/notebook_cookie_secret
[I 14:02:40.957 NotebookApp] Serving notebooks from local directory: /home/hadoop
[I 14:02:40.957 NotebookApp] The Jupyter Notebook is running at:
[I 14:02:40.957 NotebookApp] http://e0f4472dcb12:8888/?token=76b68a3c700b415790b019e07a3dd46a0d068c153a732d27
[I 14:02:40.957 NotebookApp]  or http://127.0.0.1:8888/?token=76b68a3c700b415790b019e07a3dd46a0d068c153a732d27
[I 14:02:40.957 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[W 14:02:40.979 NotebookApp] No web browser found: could not locate runnable browser.
[C 14:02:40.980 NotebookApp]

    To access the notebook, open this file in a browser:
        file:///home/hadoop/.local/share/jupyter/runtime/nbserver-8624-open.html
    Or copy and paste one of these URLs:
        http://e0f4472dcb12:8888/?token=76b68a3c700b415790b019e07a3dd46a0d068c153a732d27
     or http://127.0.0.1:8888/?token=76b68a3c700b415790b019e07a3dd46a0d068c153a732d27
```


Start Jupyter

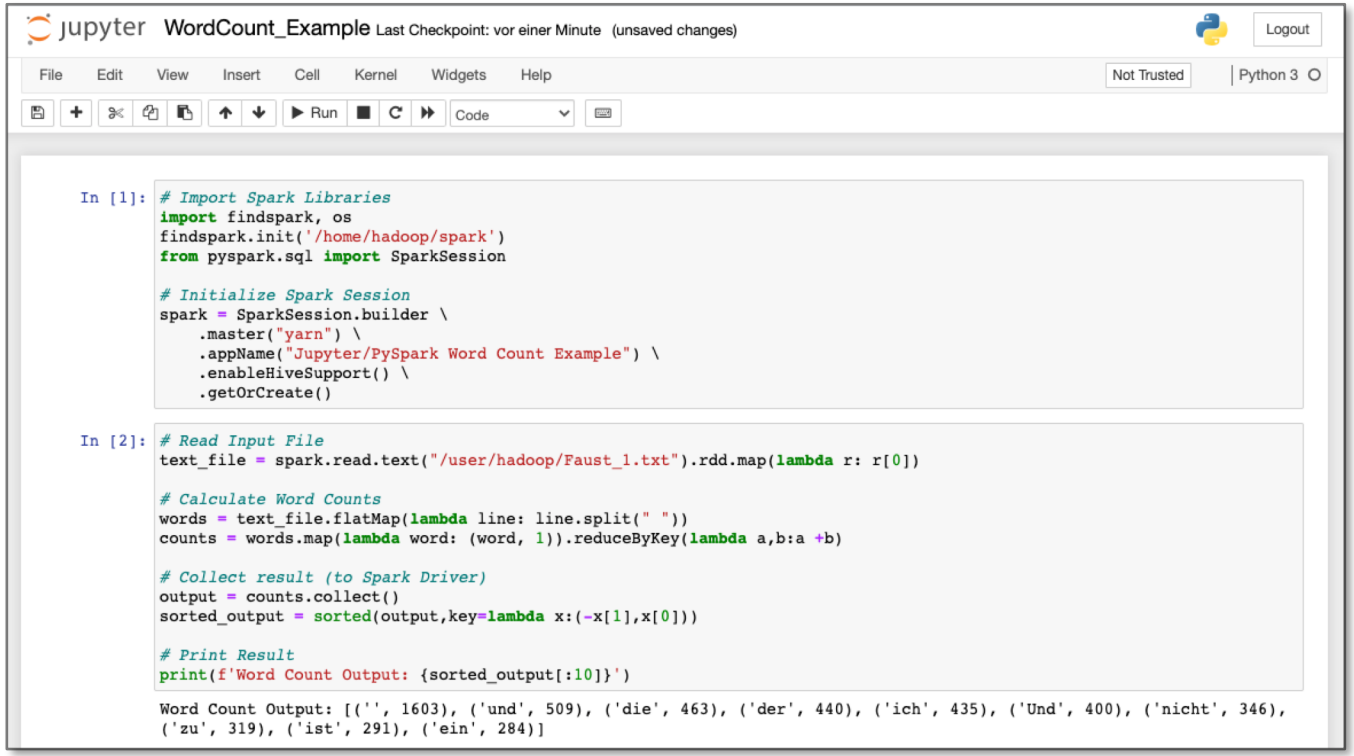
2. Open Notebook in Browser: http://XXX.XXX.XXX.XXX:8888/?token=[XYZXYZXYZ]



Use Jupyter (Word Count Example)

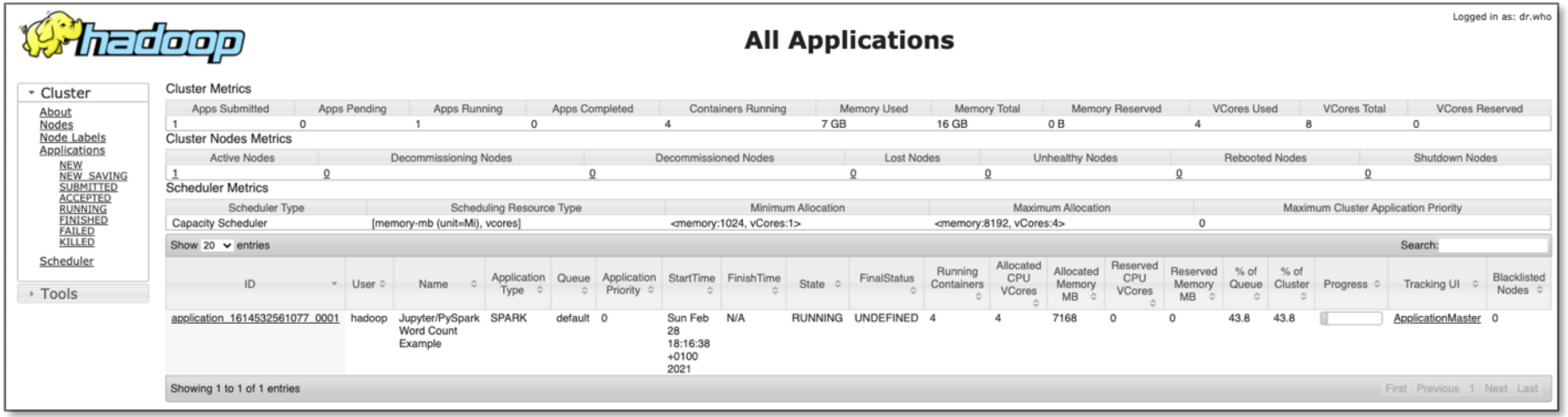
1. Execute previous Word Count example

https://github.com/marcelmittelstaedt/BigData/tree/master/exercises/winter_semester_2020-2021/04_spark_pyspark_jupyter/WordCount_Example.html



Use Jupyter (Word Count Example)

2. See **Jupyter** PySpark Container Running on Yarn http://xxx.xxx.xxx.xxx:8088/cluster :



Get some data…

1. Get some IMDb data:

```
wget https://datasets.imdbws.com/title.basics.tsv.gz && gunzip title.basics.tsv.gz
wget https://datasets.imdbws.com/title.ratings.tsv.gz && gunzip title.ratings.tsv.gz
```

2. Put them into HDFS:

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

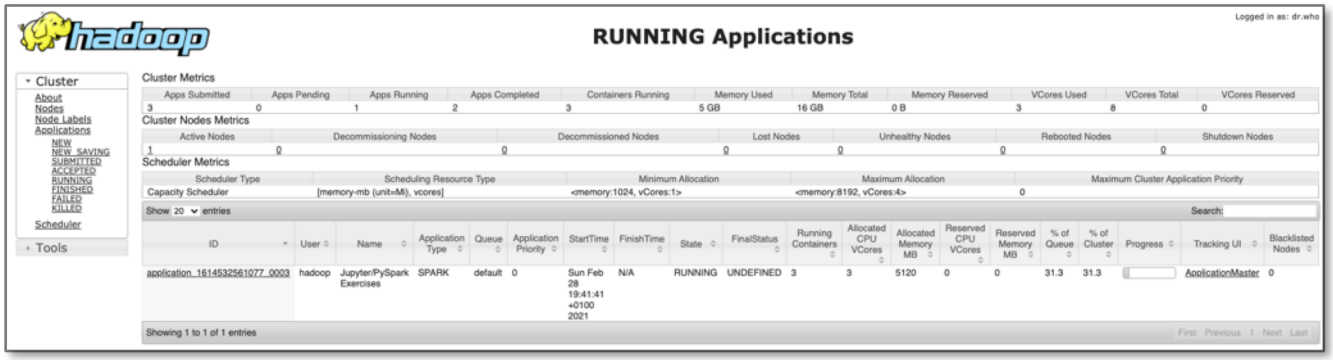
```
hadoop fs -mkdir /user/hadoop/imdb/title_basics
hadoop fs -mkdir /user/hadoop/imdb/title_ratings
```

```
hadoop fs -put title.basics.tsv /user/hadoop/imdb/title_basics/title.basics.tsv
hadoop fs -put title.ratings.tsv /user/hadoop/imdb/title_ratings/title.ratings.tsv
```

PySpark Operations

1. Initialize Spark Session:

```
In [1]: # Import Spark Libraries
        import findspark, os
        findspark.init('/home/hadoop/spark')
        from pyspark.sql import SparkSession

        # Initialize Spark Session
        spark = SparkSession.builder \
            .master("yarn") \
            .appName("Jupyter/PySpark Exercises") \
            .enableHiveSupport() \
            .getOrCreate()
```



https://github.com/marcelmittelstaedt/BigData/tree/master/exercises/winter_semester_2020-2021/04_spark_pyspark_jupyter/Exercises.html

PySpark Operations

2. Basic PySpark operations: Read Files from HDFS into DataFrames:

```
In [2]: # Read IMDb title basics CSV file from HDFS
        df_title_basics = spark.read \
            .format('csv') \
            .options(header='true', delimiter='\t', nullValue='null', inferSchema='true') \
            .load('/user/hadoop/imdb/title_basics/title.basics.tsv')

In [3]: # Print Schema of DataFrame
        df_title_basics.printSchema()

root
 |-- tconst: string (nullable = true)
 |-- titleType: string (nullable = true)
 |-- primaryTitle: string (nullable = true)
 |-- originalTitle: string (nullable = true)
 |-- isAdult: string (nullable = true)
 |-- startYear: string (nullable = true)
 |-- endYear: string (nullable = true)
 |-- runtimeMinutes: string (nullable = true)
 |-- genres: string (nullable = true)


In [4]: # Print First 3 Rows of DataFrame Data
        df_title_basics.show(3)

+---------+---------+--------------------+--------------------+-------+---------+-------+--------------+--------------
-------+
|   tconst|titleType|        primaryTitle|       originalTitle|isAdult|startYear|endYear|runtimeMinutes|
genres|
+---------+---------+--------------------+--------------------+-------+---------+-------+--------------+--------------
-------+
|tt0000001|    short|           Carmencita|           Carmencita|      0|     1894|     \N|             1|   Documentar
y,Short|
|tt0000002|    short|Le clown et ses c...|Le clown et ses c...|      0|     1892|     \N|             5|     Animatio
n,Short|
|tt0000003|    short|      Pauvre Pierrot|      Pauvre Pierrot|      0|     1892|     \N|             4|Animation,Com
edy,...|
+---------+---------+--------------------+--------------------+-------+---------+-------+--------------+--------------
-------+
only showing top 3 rows
```

https://github.com/marcelmittelstaedt/BigData/tree/master/exercises/winter_semester_2020-2021/04_spark_pyspark_jupyter/Exercises.html

www.marcel-mittelstaedt.com

PySpark Operations

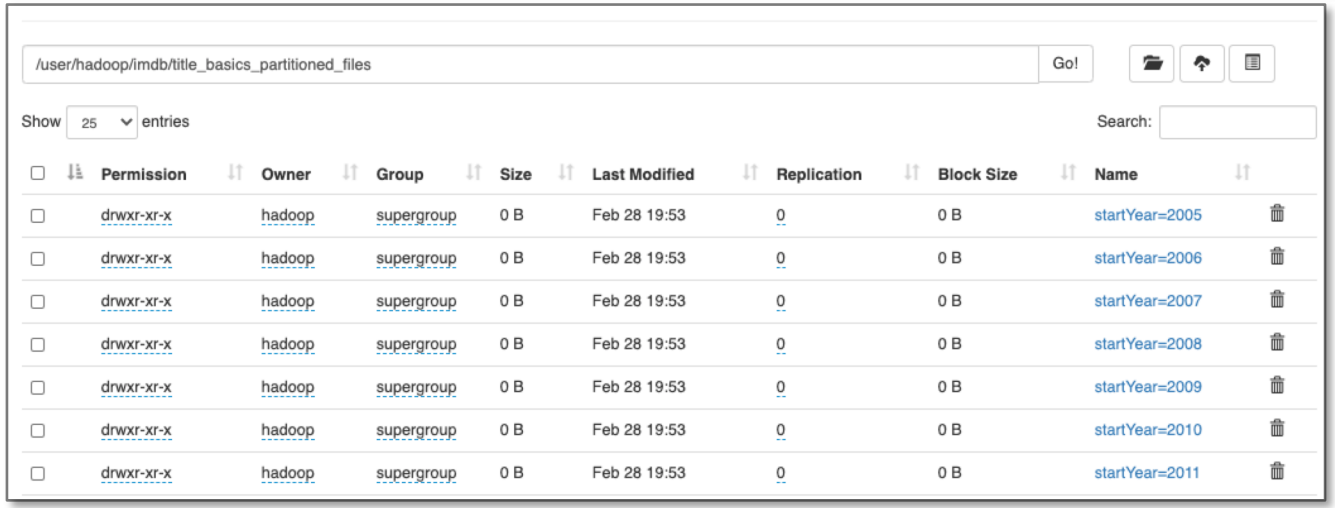
3. Basic PySpark: Operations on DataFrames (aggregations…):

```
In [5]: # Get Number of Rows of a DataFrame
        df_title_basics.count()

Out[5]: 7656314

In [6]: # Groups and Counts: Get column titleTypes values with counts and ordered descending
        from pyspark.sql.functions import desc
        df_title_basics \
            .groupBy("titleType") \
            .count() \
            .orderBy(desc("count")) \
            .show()

+------------+-------+
|   titleType|  count|
+------------+-------+
|    tvEpisode|5556805|
|       short| 796470|
|       movie| 569437|
|       video| 296405|
|    tvSeries| 202321|
|     tvMovie| 130185|
|tvMiniSeries|  36080|
|   tvSpecial|  31590|
|   videoGame|  27416|
|     tvShort|   9602|
|   audiobook|      1|
|     episode|      1|
| radioSeries|      1|
+------------+-------+


In [7]: # Calculate average Movie length in minutes
        from pyspark.sql.functions import avg, col
        df_title_basics \
            .where(col('titleType') == 'movie') \
            .agg(avg('runtimeMinutes')) \
            .show()

+------------------+
|avg(runtimeMinutes)|
+------------------+
|  89.62651045204976|
+------------------+
```


PySpark Operations

4. Basic PySpark: Save PySpark DataFrame back to HDFS (as partitioned parquet files):

```
In [8]: # Save Dataframe back to HDFS (partitioned) as Parquet files
        df_title_basics.repartition('startYear').write \
            .format("parquet") \
            .mode("overwrite") \
            .partitionBy('startYear') \
            .save('/user/hadoop/imdb/title_basics_partitioned_files')
```



Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
drwxr-xr-x	hadoop	supergroup	0 B	Feb 28 19:53	0	0 B	startYear=2005
drwxr-xr-x	hadoop	supergroup	0 B	Feb 28 19:53	0	0 B	startYear=2006
drwxr-xr-x	hadoop	supergroup	0 B	Feb 28 19:53	0	0 B	startYear=2007
drwxr-xr-x	hadoop	supergroup	0 B	Feb 28 19:53	0	0 B	startYear=2008
drwxr-xr-x	hadoop	supergroup	0 B	Feb 28 19:53	0	0 B	startYear=2009
drwxr-xr-x	hadoop	supergroup	0 B	Feb 28 19:53	0	0 B	startYear=2010
drwxr-xr-x	hadoop	supergroup	0 B	Feb 28 19:53	0	0 B	startYear=2011

PySpark Operations

5. Basic PySpark: Save PySpark DataFrame back to HDFS (as table and partitioned parquet files):

```
In [9]: # Save Dataframe back to HDFS (partitioned) as EXTERNAL TABLE and Parquet files
        df_title_basics.repartition('startYear').write \
            .format("parquet") \
            .mode("overwrite") \
            .option('path', '/user/hadoop/imdb/title_basics_partitioned_table') \
            .partitionBy('startYear') \
            .saveAsTable('default.title_basics_partitioned')
```

Spark Table

/user/hadoop/imdb/title_basics_partitioned_table

Show 25 entries

Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
drwxr-xr-x	hadoop	supergroup	0 B	Feb 28 19:54	0	0 B	startYear=2
drwxr-xr-x	hadoop	supergroup	0 B	Feb 28 19:54	0	0 B	startYear=2006
drwxr-xr-x	hadoop	supergroup	0 B	Feb 28 19:54	0	0 B	startYear=2007
drwxr-xr-x	hadoop	supergroup	0 B	Feb 28 19:54	0	0 B	startYear=2008
drwxr-xr-x	hadoop	supergroup	0 B	Feb 28 19:54	0	0 B	startYear=2009
drwxr-xr-x	hadoop	supergroup	0 B	Feb 28 19:54	0	0 B	startYear=2010
drwxr-xr-x	hadoop	supergroup	0 B	Feb 28 19:54	0	0 B	startYear=2011

```
In [10]: spark.sql('SHOW TABLES').show(10, False)
+--------+-----------------------+-----------+
|database|tableName              |isTemporary|
+--------+-----------------------+-----------+
|default |title_basics_partitioned|false      |
+--------+-----------------------+-----------+
```

PySpark Operations

6. Basic PySpark: Interact with Spark Tables (using plain Spark SQL):

```
In [12]: # Read External Spark table using plain Spark SQL
         df = spark.sql('SELECT tconst, primaryTitle, startYear FROM default.title_basics_partitioned WHERE startYear = 2020')
         # Print Result
         df.show(3)

         +---------+--------------------+---------+
         |   tconst|        primaryTitle|startYear|
         +---------+--------------------+---------+
         |tt0060366|A Embalagem de Vidro|     2020|
         |tt0062336|El Tango del Viud...|     2020|
         |tt0065392|  Bucharest Memories|     2020|
         +---------+--------------------+---------+
         only showing top 3 rows
```

Same Result!

7. Basic PySpark: Interact with Spark Tables (using Spark programmatically):

```
In [11]: # Read External Spark table in programmatical way
         df = spark.table('default.title_basics_partitioned') \
             .where(col('startYear') == '2020') \
             .select('tconst', 'primaryTitle', 'startYear')
         # Print Result
         df.show(3)

         +---------+--------------------+---------+
         |   tconst|        primaryTitle|startYear|
         +---------+--------------------+---------+
         |tt0060366|A Embalagem de Vidro|     2020|
         |tt0062336|El Tango del Viud...|     2020|
         |tt0065392|  Bucharest Memories|     2020|
         +---------+--------------------+---------+
         only showing top 3 rows
```

PySpark Operations

8. PySpark SQL: Join Spark DataFrames:

```
In [13]: # Read title.ratings.tsv into Spark dataframe
df_title_ratings = spark.read \
    .format('csv') \
    .options(header='true', delimiter='\t', nullValue='null', inferSchema='true') \
    .load('/user/hadoop/imdb/title_ratings/title.ratings.tsv')

In [14]: # Print Schema of title_ratings dataframe
df_title_ratings.printSchema()

root
 |-- tconst: string (nullable = true)
 |-- averageRating: double (nullable = true)
 |-- numVotes: integer (nullable = true)

In [15]: # Show first 3 rows of title_ratings dataframe
df_title_ratings.show(3)

+---------+-------------+--------+
|   tconst|averageRating|numVotes|
+---------+-------------+--------+
|tt0000001|          5.7|    1685|
|tt0000002|          6.0|     208|
|tt0000003|          6.5|    1425|
+---------+-------------+--------+
only showing top 3 rows
```

```
In [16]: # JOIN Data Frames
joined_df = df_title_basics.join(df_title_ratings, df_title_basics.tconst == df_title_ratings.tconst)

In [17]: # Print Schema of joined DataFrame
joined_df.printSchema()

root
 |-- tconst: string (nullable = true)
 |-- titleType: string (nullable = true)
 |-- primaryTitle: string (nullable = true)
 |-- originalTitle: string (nullable = true)
 |-- isAdult: string (nullable = true)
 |-- startYear: string (nullable = true)
 |-- endYear: string (nullable = true)
 |-- runtimeMinutes: string (nullable = true)
 |-- genres: string (nullable = true)
 |-- tconst: string (nullable = true)
 |-- averageRating: double (nullable = true)
 |-- numVotes: integer (nullable = true)

In [18]: # Show Frist 3 Rows of Joined DataFrame
joined_df.show(3)

+---------+---------+--------------------+--------------------+-------+---------+-------+--------------+--------------------+---------+-------------+--------+
|   tconst|titleType|        primaryTitle|       originalTitle|isAdult|startYear|endYear|runtimeMinutes|              genres|   tconst|averageRating|numVotes|
+---------+---------+--------------------+--------------------+-------+---------+-------+--------------+--------------------+---------+-------------+--------+
|tt0000658|    short|The Puppet's Nigh...|Le cauchemar de F...|      0|     1908|     \N|             2|Animation,Short|tt0000658|          6.4|     184|
|tt0001732|    short|The Lighthouse Ke...|The Lighthouse Ke...|      0|     1911|     \N|            \N|       Drama,Short|tt0001732|          7.1|       8|
|tt0002253|    short|          Home Folks|          Home Folks|      0|     1912|     \N|            17|       Drama,Short|tt0002253|          3.7|       6|
+---------+---------+--------------------+--------------------+-------+---------+-------+--------------+--------------------+---------+-------------+--------+
only showing top 3 rows
```

PySpark Operations

9. Basic PySpark: Filtering, Ordering and Selecting: Get Top 5 TV Series

In [19]:
```
top_tvseries = joined_df \
    .where(col('titleType') == 'tvSeries') \
    .where(col('numVotes') > 200000) \
    .orderBy(desc('averageRating')) \
    .select('originalTitle', 'startYear', 'endYear', 'averageRating', 'numVotes')

# Print Top 5 TV Series
top_tvseries.show(5)
```

```
+--------------------+---------+-------+-------------+--------+
|       originalTitle|startYear|endYear|averageRating|numVotes|
+--------------------+---------+-------+-------------+--------+
|        Breaking Bad|     2008|   2013|          9.5| 1470997|
|            The Wire|     2002|   2008|          9.3|  286432|
|     Game of Thrones|     2011|   2019|          9.3| 1775327|
|      Rick and Morty|     2013|     \N|          9.2|  377428|
|Avatar: The Last ...|     2005|   2008|          9.2|  249236|
+--------------------+---------+-------+-------------+--------+
only showing top 5 rows
```

PySpark Operations

10. Basic PySpark: Add/Calculate Columns:

In [20]:
```
from pyspark.sql.functions import when, lit

# Add a calculated column: classify movies as being either 'good' or 'worse' based on average rating
df_with_classification = joined_df \
    .withColumn('classification',
                when(col('averageRating') > 8, lit('good')) \
                .otherwise(lit('worse'))) \
    .select('primaryTitle', 'startYear', 'averageRating', 'classification')

# Print Result
df_with_classification.show(3, False)
```

```
+----------------------+---------+-------------+--------------+
|primaryTitle          |startYear|averageRating|classification|
+----------------------+---------+-------------+--------------+
|The Puppet's Nightmare|1908     |6.4          |worse         |
|The Lighthouse Keeper |1911     |7.1          |worse         |
|Home Folks            |1912     |3.7          |worse         |
+----------------------+---------+-------------+--------------+
only showing top 3 rows
```

PySpark Operations

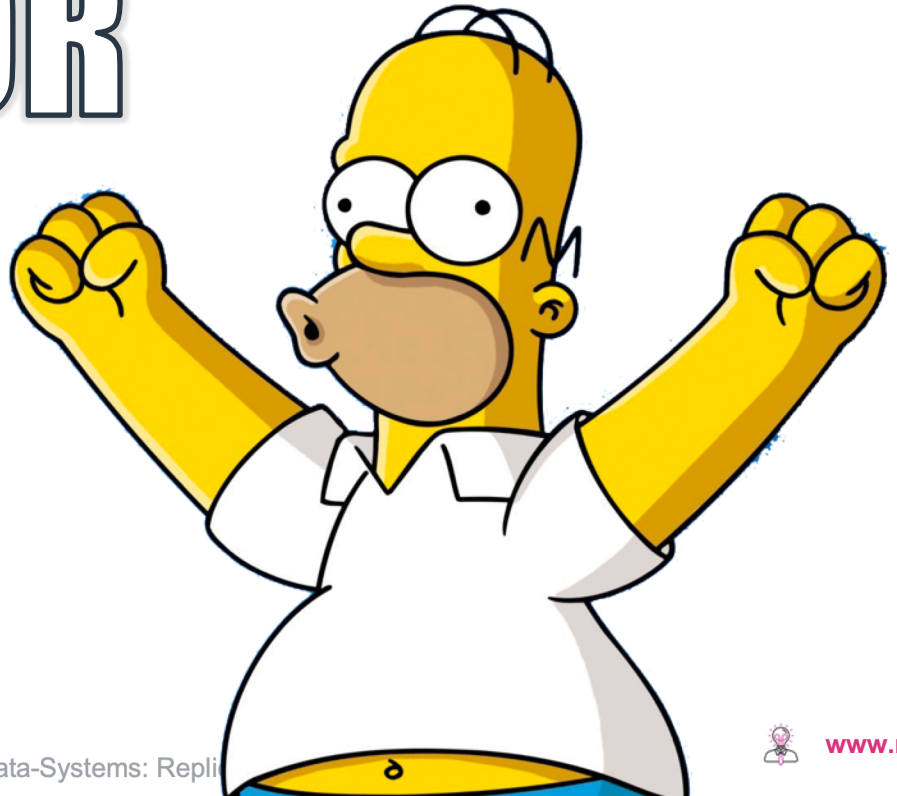
11. Basic PySpark/Python: Plot Data:

In [21]:
```
# Plot data: good movies per year
import matplotlib.pyplot as plt
import pandas

# Create DataFrame to be plotted
good_movies = df_with_classification \
    .select('startYear', 'classification') \
    .where(col('classification') == 'good') \
    .where(col('startYear') > 2000) \
    .groupBy('startYear') \
    .count() \
    .sort(col('startYear').asc())

# Convert Spark DataFrame to Pandas DataFrame
pandas_df = good_movies.toPandas()

# Plot DataFrame
pandas_df.plot.bar(x='startYear', y='count')
```

Out[21]: <AxesSubplot:xlabel='startYear'>

Break

TIME FOR A BREAK



Exercises

Use PySpark Shell or Jupyter Notebooks on PySpark to solve exercises

PySpark Exercises - **IMDB**

1. Execute Tasks of previous HandsOn Slides

2. Create External Spark Table `title_ratings` on HDFS containing data of IMDb file *title.ratings.tsv*

3. Create External Spark Table `name_basics` on HDFS containing data of IMDb file *name.basics.tsv*

4. Use PySpark to answer following questions:

    a) How many **movies** and how many **TV series** are within the IMDB dataset?

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

PySpark Exercises - **IMDB**

4. Use PySpark to answer following questions:

    c) Create a list (`tconst, original_title, start_year, average_rating, num_votes`) of movies which are:
    - equal or newer than year 2010
    - have an average rating equal or better than 8,1
    - have been voted more than 100.000 times

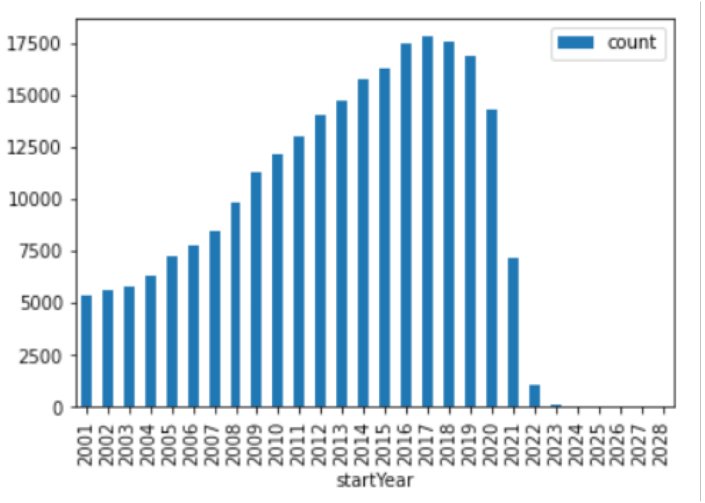
    d) Save result of c) as external Spark Table to HDFS.?

5. Create a Spark Table **`name_basics_partitioned`**, which:
    - contains all columns of table `name_basics`
    - is partitioned by column `partition_is_alive`, containing:
        - „*alive*“ in case actor is still alive
        - „*dead*“ in case actor is already dead

PySpark Exercises - **IMDB**

6. Create a partitioned Spark table `imdb_movies_and_ratings_partitioned`, which:
   - contains all columns of the two tables `title_basics_partitioned` and `title_ratings` and
   - is partitioned by start year of movie (create and add column `partition_year`).

7. Create following plot, which visualizes:
   - the amount of movies (type!)
   - per year
   - since 2000



Stop Your VM Instances

DON'T FORGET TO STOP YOUR VM INSTANCE!

```
gcloud compute instances stop big-data
```

Well Done

WE'RE DONE FOR ...TODAY