

# Big Data – Challenges of Distributed Data-Systems: Replication and Partitioning

*Winter Semester 2019,  
Cooperative State University Baden-Wuerttemberg*



# Agenda – 28.10.2019

01

**Presentation and Discussion: Exercise Of Last Lecture**  
HiveQL, HDFS/Hive Partitioning, HiveServer2

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

**Introduction To The Challenges Of Distributed Data-Systems: Partitioning**  
Basics of Partitioning, Key-Range and Hash Partitioning, Partitioning of Secondary Indices, Rebalancing and Lookup of Partitions

04

**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.



# Schedule

	<i>Lecture Topic</i>	<i>HandsOn</i>
14.10.2019 13:15-18:00 Ro. 1.18	About This Lecture, Introduction to Big Data, Setup Cloud Environment (Google Cloud)	HandsOn Hadoop, Hive and Hive-QL
21.10.2019 13:15-18:00 Ro. 1.18	(Non-)Functional Requirements Of Distributed Data-Systems, Data Models and Access	Partitioning with HDFS/Hive, HiveServer2
28.10.2019 13:15-18:00 Ro. 1.18	<b>Challenges Of Distributed Data Systems: Replication and Partitioning</b>	Spark, Scala and PySpark/Jupyter
04.11.2019 13:15-18:00 Ro. 1.18	ETL Workflow and Automation, Batch Processing	Pentaho Data Integration & Airflow
11.11.2019 13:15-18:00 Ro. 1.18	Practical Exam	Work On Practical Exam
18.11.2019 13:15-18:00 Ro. 1.18	Practical Exam Presentation	



# Solution – Exercise IV

HiveQL, HDFS/Hive Partitioning, HiveServer2



# Solution

## Prerequisites:

- Start Gcloud instance
- Pull and start Docker image ([marcelmittelstaedt/hiveserver\\_base:latest](https://hub.docker.com/r/marcelmittelstaedt/hiveserver_base))
- 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 ORCFILE 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 2019-10-20 18:18 /user/hadoop/imdb/actors_partitioned/partition_is_alive=alive
drwxr-xr-x  - hadoop supergroup          0 2019-10-20 18:18 /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								
	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,tt0043044,tt0050419,tt0053137	dead	
2	nm0000002	Lauren Bacall	1.924	2014	actress,soundtrack	tt0117057,tt0037382,tt0038355,tt0071877	dead	
3	nm0000004	John Belushi	1.949	1982	actor,writer,soundtrack	tt0072562,tt0077975,tt0078723,tt0080455	dead	
4	nm0000005	Ingmar Bergman	1.918	2007	writer,director,actor	tt0083922,tt0050986,tt0050976,tt0069467	dead	
5	nm0000006	Ingrid Bergman	1.915	1982	actress,soundtrack,producer	tt0036855,tt0071877,tt0038787,tt0038109	dead	
6	nm0000007	Humphrey Bogart	1.899	1957	actor,soundtrack,producer	tt0033870,tt0037382,tt0034583,tt0043265	dead	
7	nm0000008	Marlon Brando	1.924	2004	actor,soundtrack,director	tt0047296,tt0068646,tt0070849,tt0078788	dead	
8	nm0000009	Richard Burton	1.925	1984	actor,producer,soundtrack	tt0087803,tt0057877,tt0059749,tt0061184	dead	
9	nm0000010	James Cagney	1.899	1986	actor,soundtrack,director	tt0031867,tt0035575,tt0029870,tt0042041	dead	
10	nm0000011	Gary Cooper	1.901	1961	actor,soundtrack,producer	tt0035896,tt0034167,tt0044706,tt0027996	dead	
11	nm0000012	Bette Davis	1.908	1989	actress,soundtrack,make_up_department	tt0056687,tt0035140,tt0031210,tt0042192	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 ORCFILE 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 2019-10-20 18:27 /user/hadoop/imdb/movies_and_ratings_partitioned/partition_year=1874
drwxr-xr-x  - hadoop supergroup          0 2019-10-20 18:27 /user/hadoop/imdb/movies_and_ratings_partitioned/partition_year=1878
drwxr-xr-x  - hadoop supergroup          0 2019-10-20 18:27 /user/hadoop/imdb/movies_and_ratings_partitioned/partition_year=1881
drwxr-xr-x  - hadoop supergroup          0 2019-10-20 18:27 /user/hadoop/imdb/movies_and_ratings_partitioned/partition_year=1883
drwxr-xr-x  - hadoop supergroup          0 2019-10-20 18:27 /user/hadoop/imdb/movies_and_ratings_partitioned/partition_year=1885
drwxr-xr-x  - hadoop supergroup          0 2019-10-20 18:27 /user/hadoop/imdb/movies_and_ratings_partitioned/partition_year=1887
drwxr-xr-x  - hadoop supergroup          0 2019-10-20 18:27 /user/hadoop/imdb/movies_and_ratings_partitioned/partition_year=1888
drwxr-xr-x  - hadoop supergroup          0 2019-10-20 18:27 /user/hadoop/imdb/movies_and_ratings_partitioned/partition_year=1889
drwxr-xr-x  - hadoop supergroup          0 2019-10-20 18:27 /user/hadoop/imdb/movies_and_ratings_partitioned/partition_year=1890
[...]
```



# Solution

## Exercise IV:

### 3.3 Check Results:

select * from imdb_movies_and_ratings_partitioned where partition_year = 2019 LIMIT 100												
Result												
select * from imdb_movies_and_ratings_partitioned   Geben Sie einen SQL-Ausdruck ein, um die Ergebnisse zu filtern (verwenden Sie Strg+ Leertaste).												
	abc_tconst	abc_title_type	abc_primary_title	abc_original_title	is_adult	start_year	en	runtime_minutes	abc_genres	average_rating	num_votes	partition_year
1	tt091490	short	Martina's Playhouse	Martina's Playhouse	0	2.019	[NULL]	20	Drama,Short	5,8	27	2.019
2	tt172112	short	Ambulans	Ambulans	0	2.019	[NULL]	11	Short	7,6	41	2.019
3	tt0172817	tvShort	Monolog trebacza	Monolog trebacza	0	2.019	[NULL]	22	Short	7,2	11	2.019
4	tt0255841	short	Bird in a Window	Bird in a Window	0	2.019	[NULL]	10	Animation,Short	7,4	23	2.019
5	tt0269235	short	Flying Nansen	Flying Nansen	0	2.019	[NULL]	11	Animation,Short	7,9	21	2.019
6	tt0302617	tvMovie	Great Bear Rainforest	Great Bear Rainforest	0	2.019	[NULL]	[NULL]	Documentary	8,2	25	2.019
7	tt0345776	tvMovie	The Patchwork Girl of Oz	The Patchwork Girl of Oz	0	2.019	[NULL]	[NULL]	Adventure,Animator	4,8	15	2.019
8	tt0385887	movie	Motherless Brooklyn	Motherless Brooklyn	0	2.019	[NULL]	144	Crime,Drama	7,8	1.135	2.019
9	tt0437086	movie	Alita: Battle Angel	Alita: Battle Angel	0	2.019	[NULL]	122	Action,Adventure,Sci	7,4	165.972	2.019
10	tt0441881	movie	Danger Close	Danger Close: The Battle of Long	0	2.019	[NULL]	118	Action,Drama,War	8,0	882	2.019
11	tt0446792	movie	Surviving in L.A.	Surviving in L.A.	0	2.019	[NULL]	[NULL]	Comedy,Drama,Rom	8,1	13	2.019
12	tt0448115	movie	Shazam!	Shazam!	0	2.019	[NULL]	132	Action,Adventure,Co	7,1	185.949	2.019
13	tt0489974	tvSeries	Carnival Row	Carnival Row	0	2.019	[NULL]	[NULL]	Crime,Drama,Fantas	8,0	22.760	2.019
14	tt0800325	movie	The Dirt	The Dirt	0	2.019	[NULL]	107	Biography,Comedy,C	7,0	31.148	2.019
15	tt0810836	movie	Dirt Music	Dirt Music	0	2.019	[NULL]	105	Crime,Drama,Roman	7,1	28	2.019





## 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

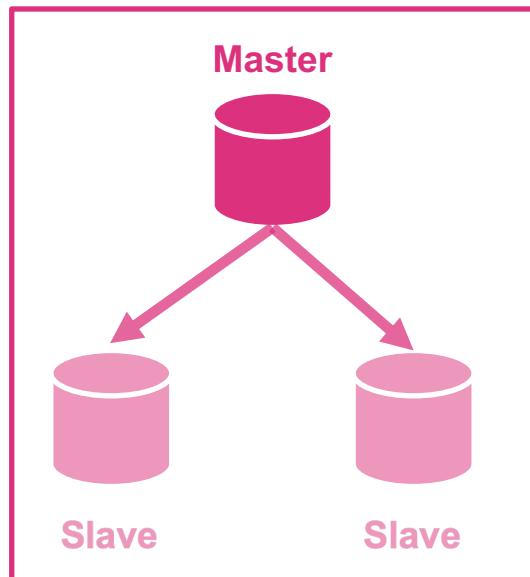
		Replication	Partitioning
stores:	<b>copies of the same data on multiple nodes</b>	<b>subsets (<i>partitions</i>) on multiple nodes</b>	
introduces:	<b>redundancy</b>	<b>distribution</b>	
scalability:	<b>parallel IO</b>	<b>memory consumption</b> , certain parallel IO	
availability:	<b>nodes can take load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

**Different purposes, but usually used together**



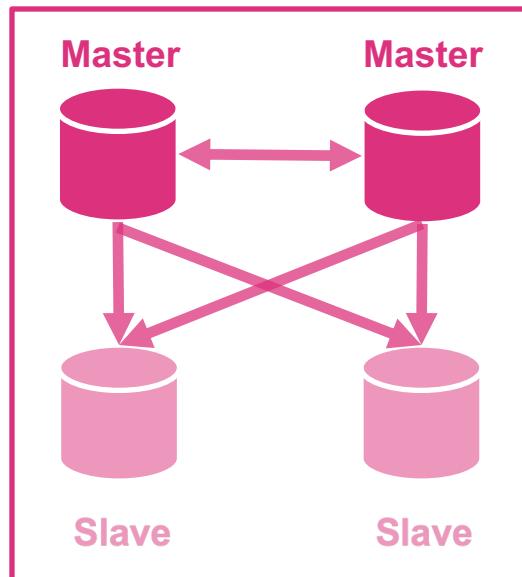
# Approaches For Replication

Master-based



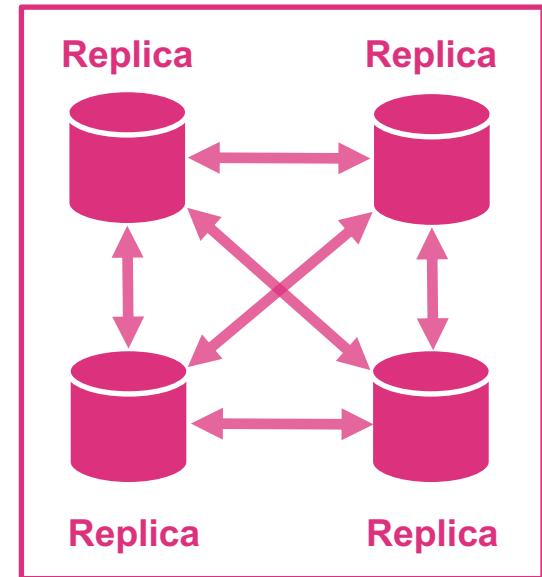
e.g. Oracle, PostgreSQL,  
MySQL...

Multi-Master-based



e.g. CouchDB, PostgreSQL,  
MySQL...

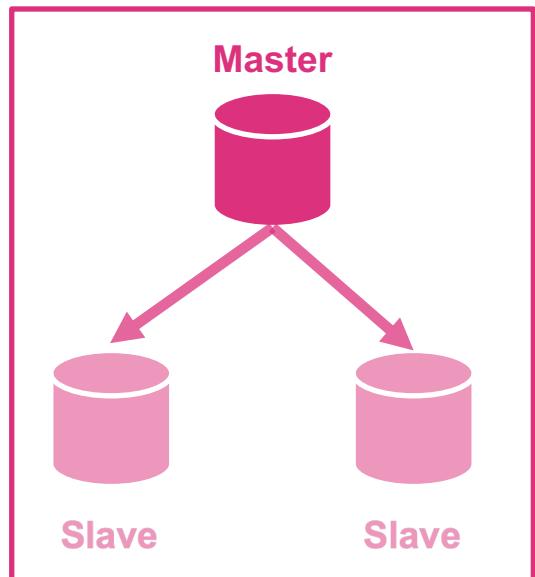
Masterless



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

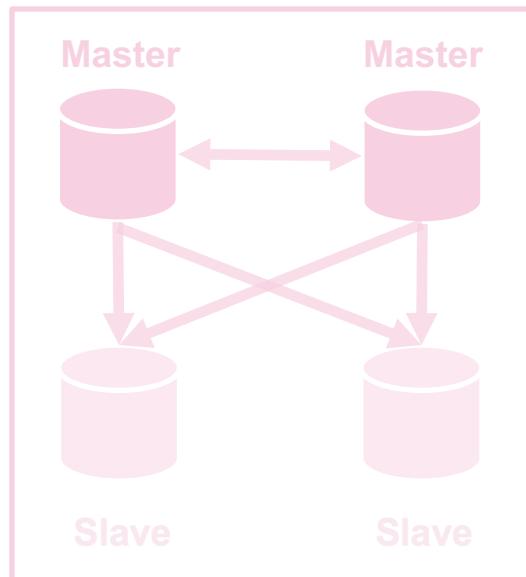
# Master Replication

**Master-based**



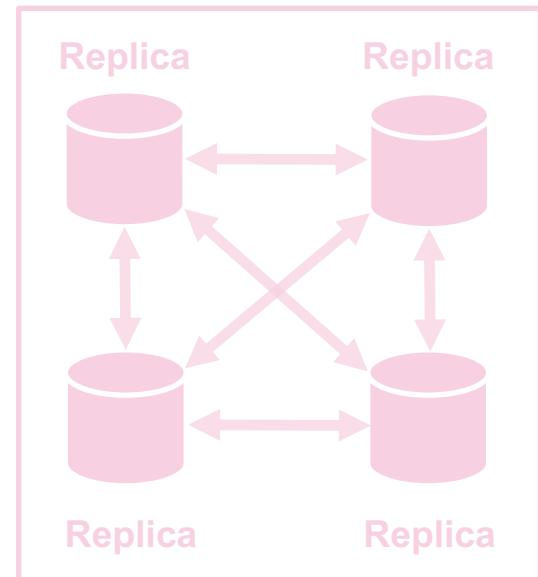
e.g. Oracle, PostgreSQL,  
MySQL...

**Multi-Master-based**



e.g. CouchDB, PostgreSQL,  
MySQL...

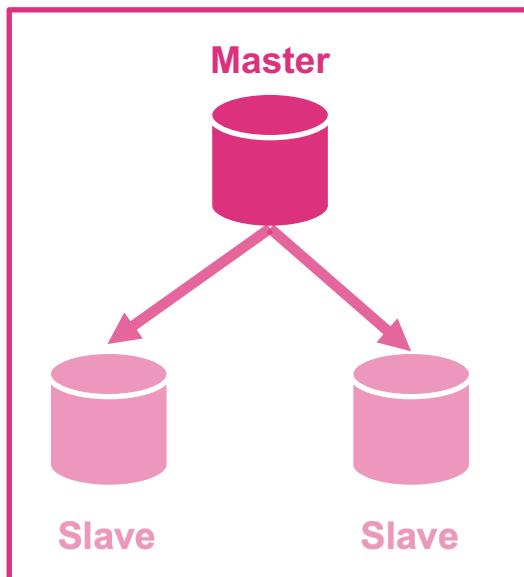
**Masterless**



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

# Master Replication

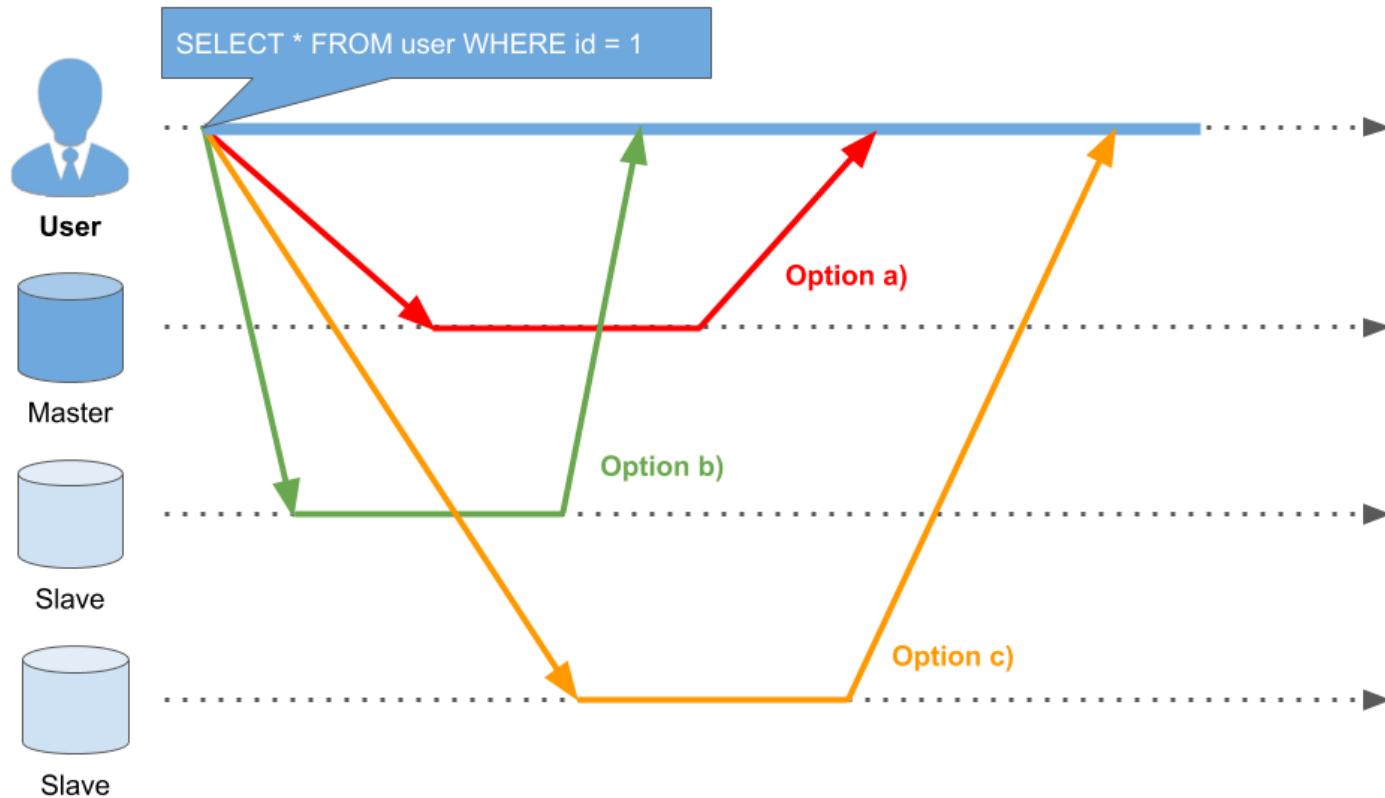
**Also known as:** Primary/Secondary, Master/Slave or Single-Leader Replication



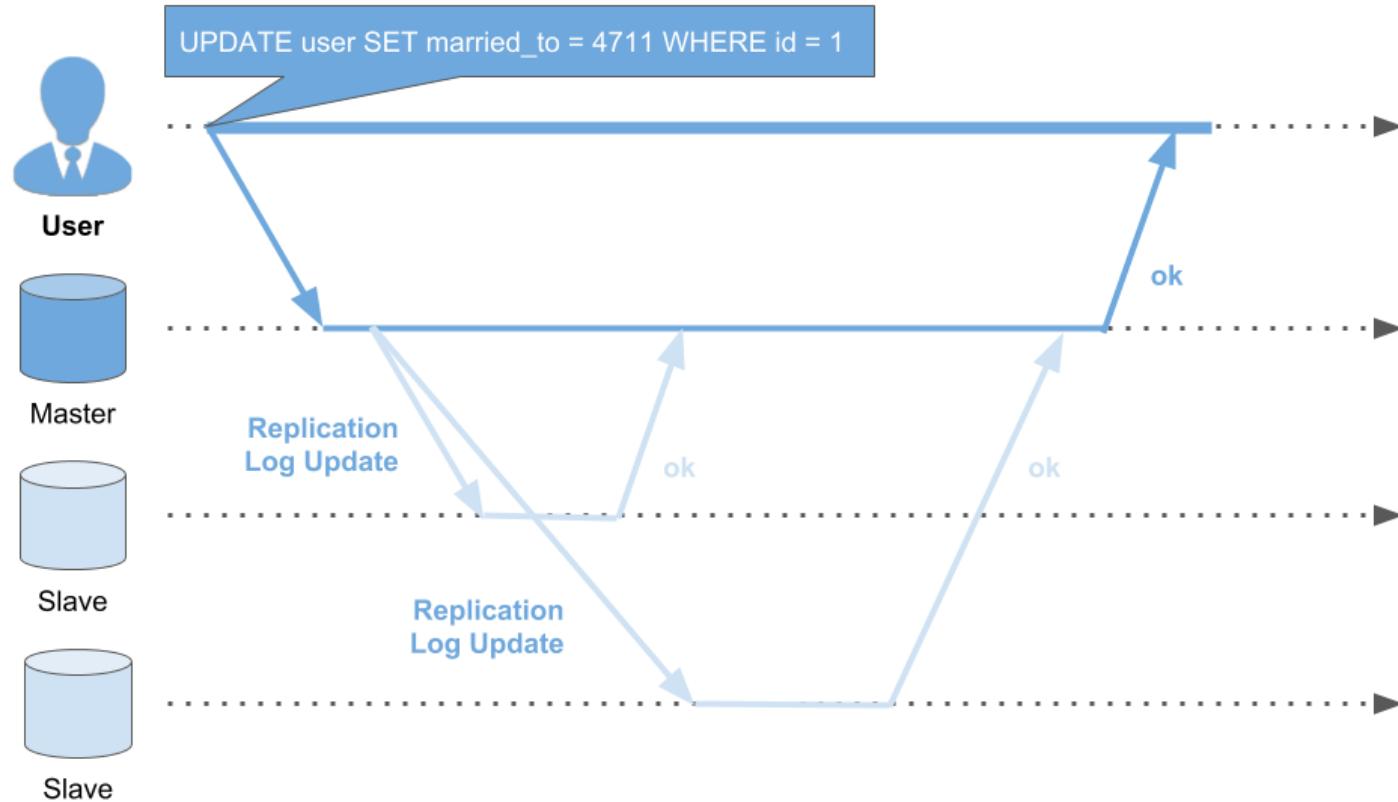
**Master:** (also known as *leader* or *primary*) dedicated processing node, usually also a replica and also known as primary or leader. The master node serves read as well as write queries, is responsible for persisting changes locally and propagates changes (e.g. by using replication logs or change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

**Slave:** (also known as *secondary*, *follower* or *hot-standby*) are dedicated 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.

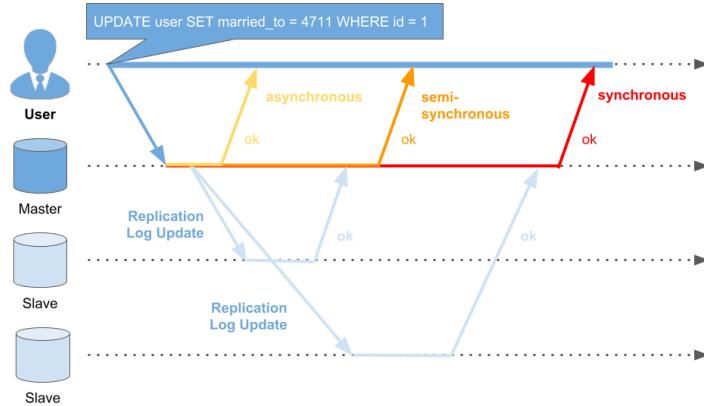
# Master Replication - Read



# 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 additional processing time to write queries

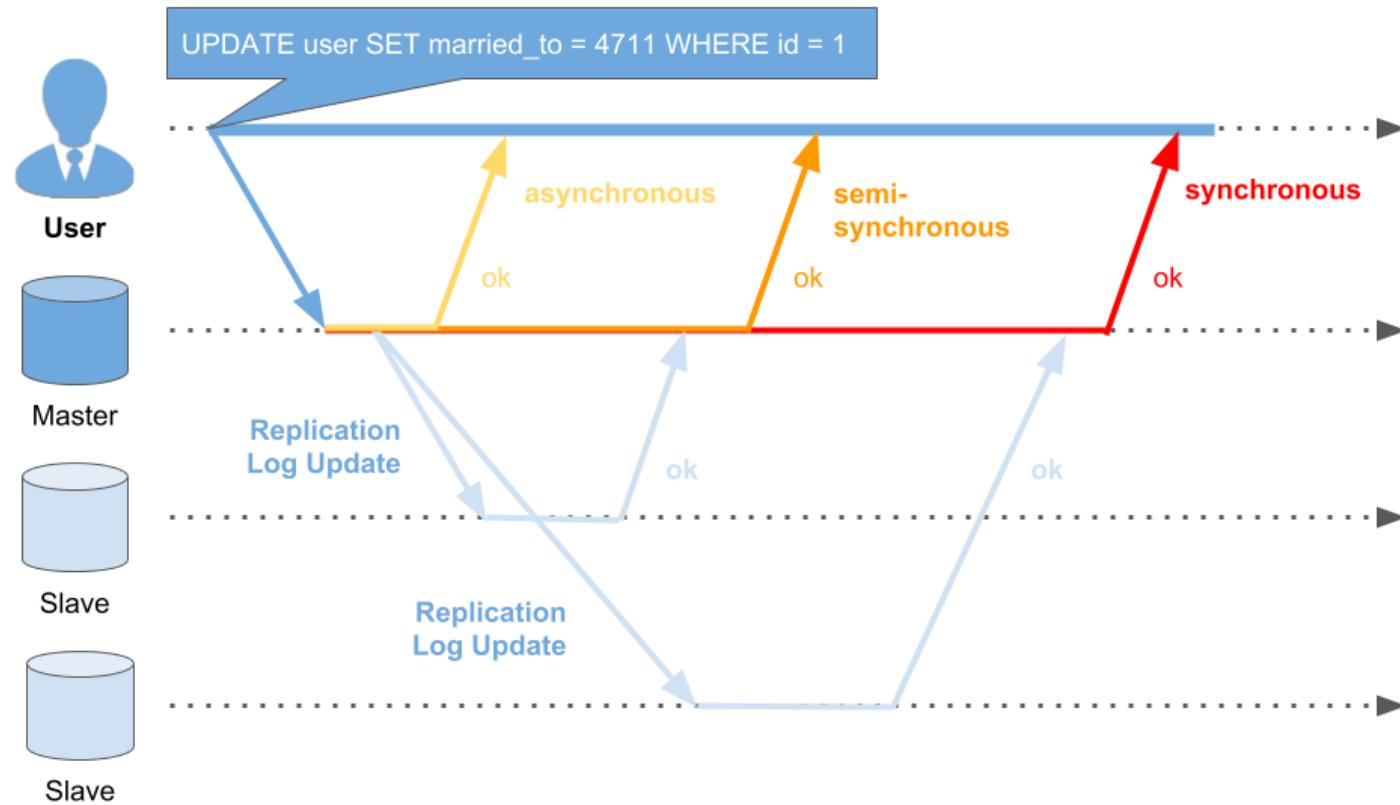
## 2. Semi-Synchronous:

- 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

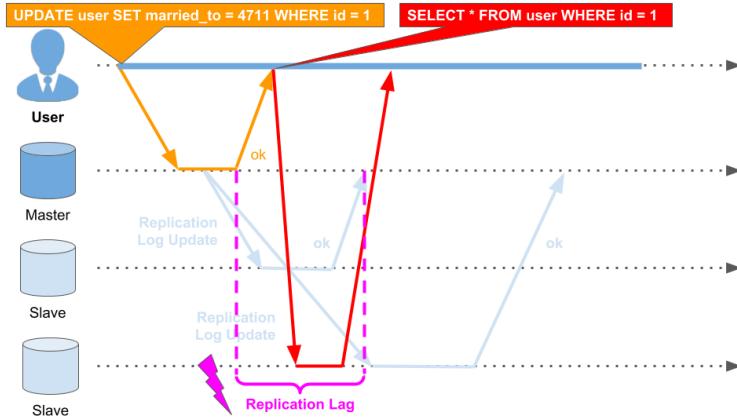
## 3. Asynchronous:

- 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 crashes 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

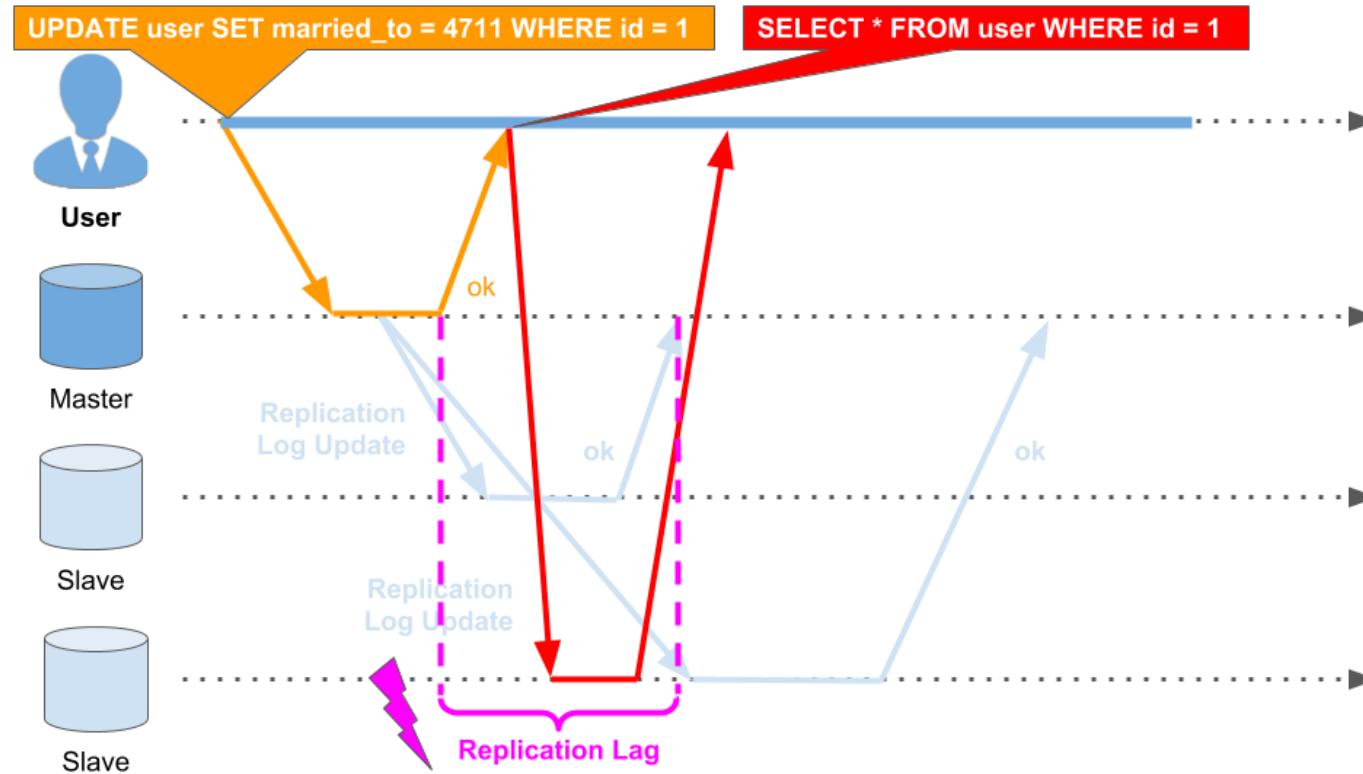
- Distribute and scale read requests by „just“ adding more slaves
- **Data locality** = put slaves geographically close to user/client-applications
- But: **vulnerable to replication lag** (timeframe when master and slaves are inconsistent, as the slaves have not yet processed the most recent write requests)

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

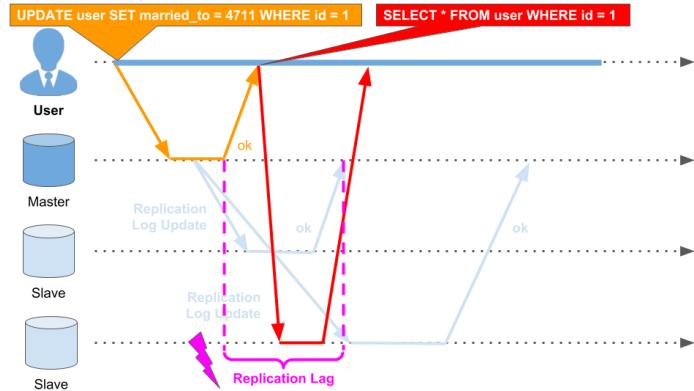
- Examples of replication lag causing a data-system to be inconsistent:

- **Reading Your Own Writes**
- **Monotonic Reads**

# Master Replication – Replication Lag



# Master Replication – Read-Your-Own-Writes



## Read-Your-Writes Consistency:

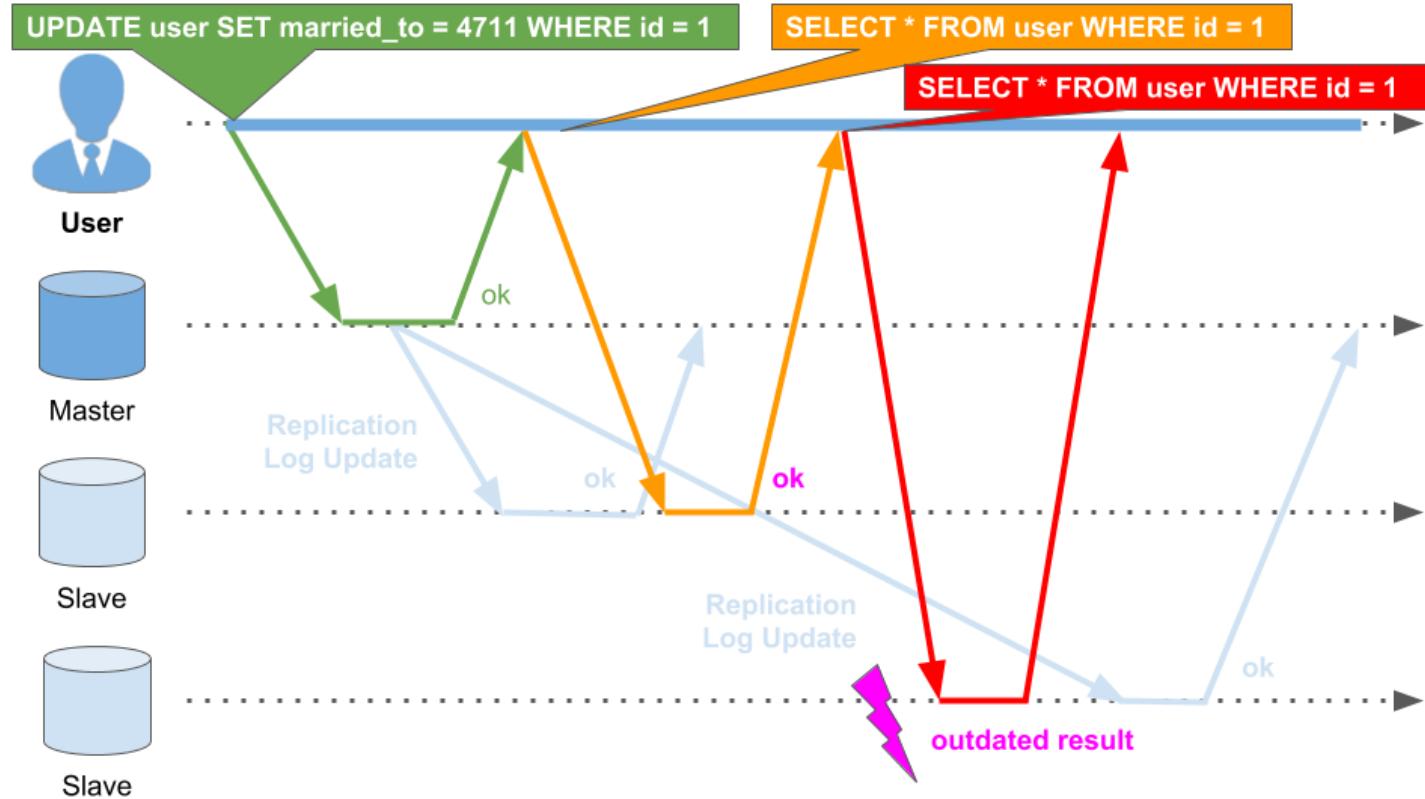
ensures that any write requests submitted by a user will directly be seen on any further read requests (by the same user, guaranteeing a user his write request has been processed successfully).

## Approaches for achieving Read-Your-Writes consistency:

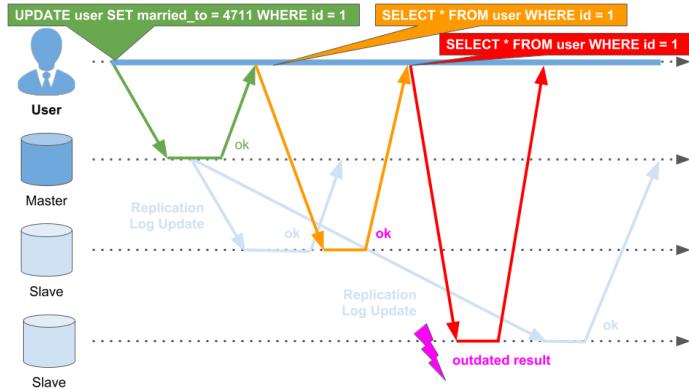
- **Read data, a user may have modified, from the master**, otherwise make use of a slave. E.g. Xing/LinkedIn/Facebook Profile profiles → those profiles can only be edited by the user itself.
- The previous approach does not work very well for data-systems where a user is able to edit almost anything 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 whether use of the 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 a 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 respond 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}})
```

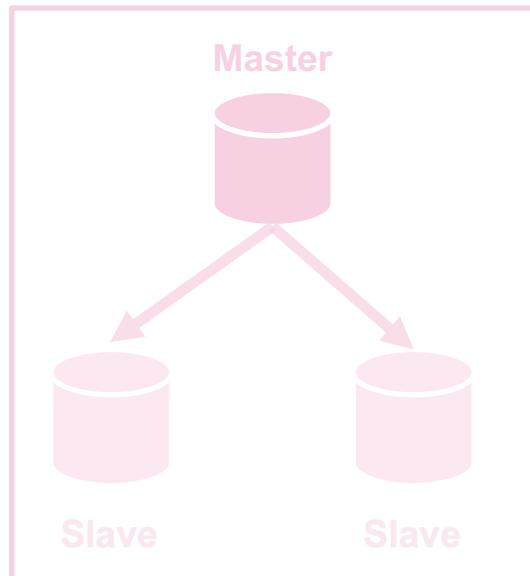
## Resulting Replication Log

```
$ 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 }
  }
}
```



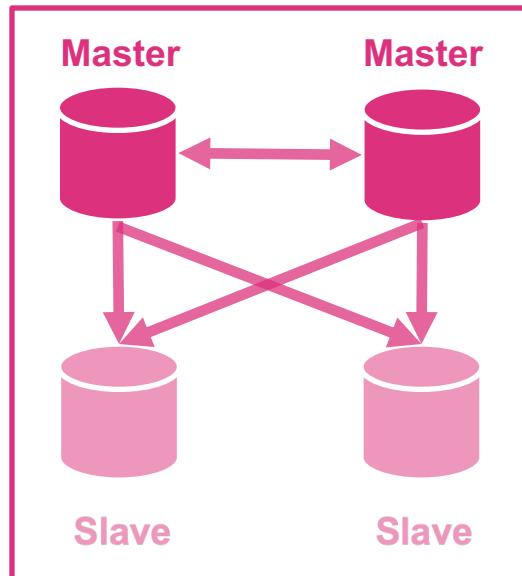
# Multi-Master Replication

Master-based



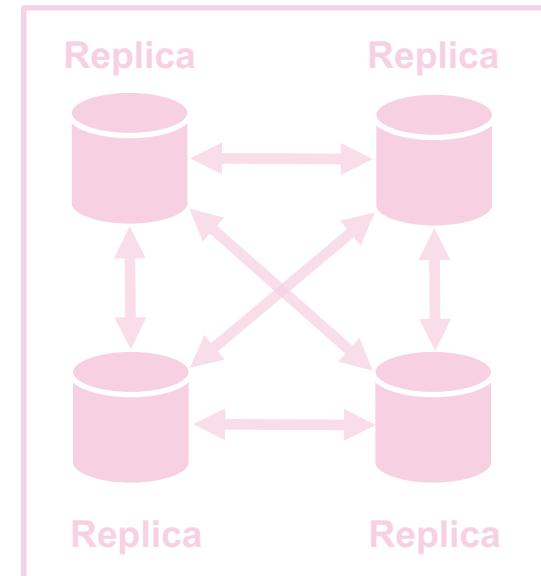
e.g. Oracle, PostgreSQL,  
MySQL...

Multi-Master-based



e.g. CouchDB, PostgreSQL,  
MySQL...

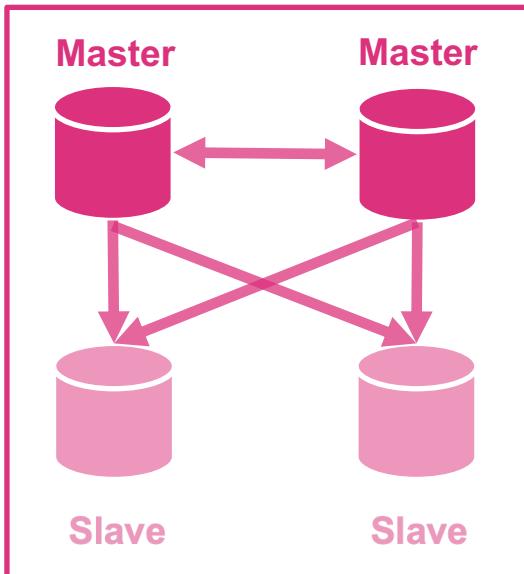
Masterless



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

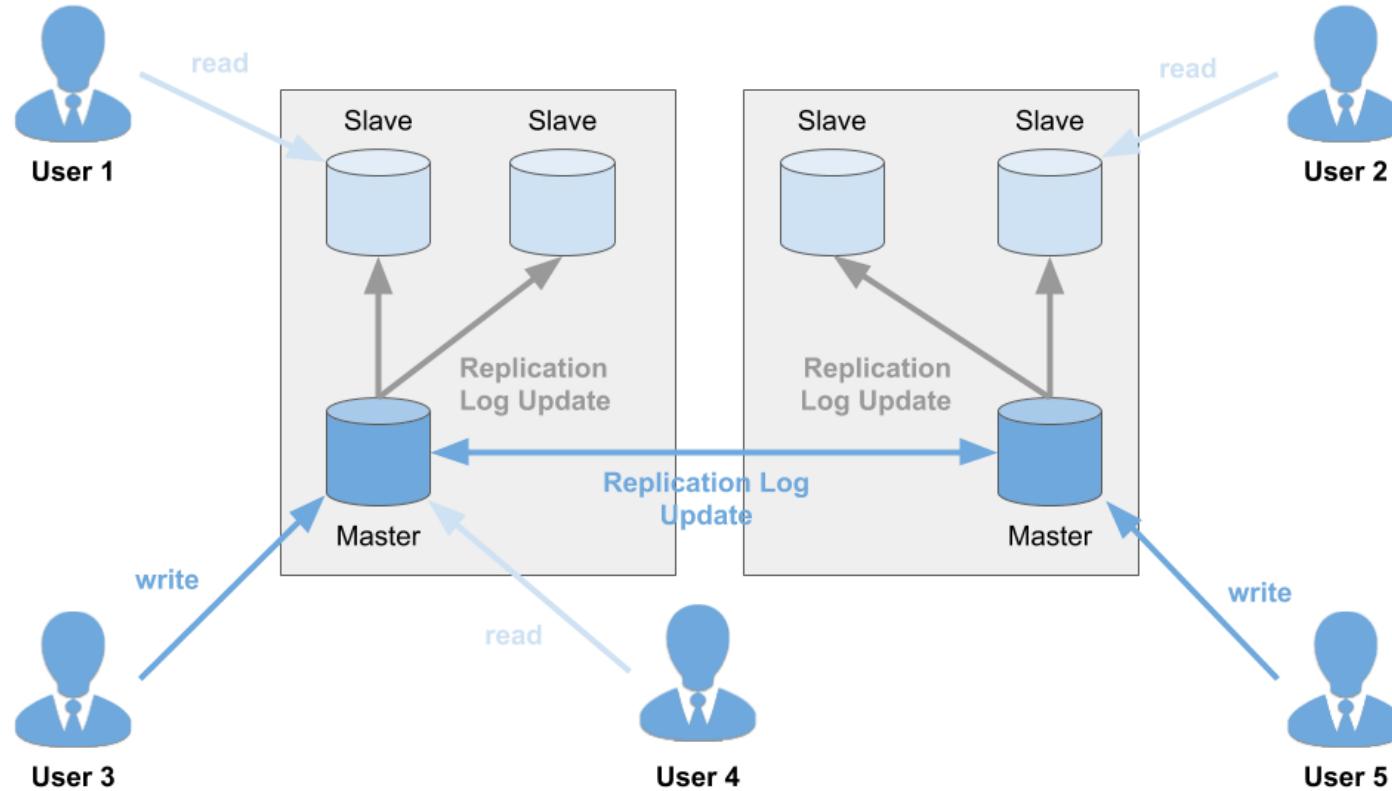
## Advantages:

- **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 simultaneously (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 conflict? **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.

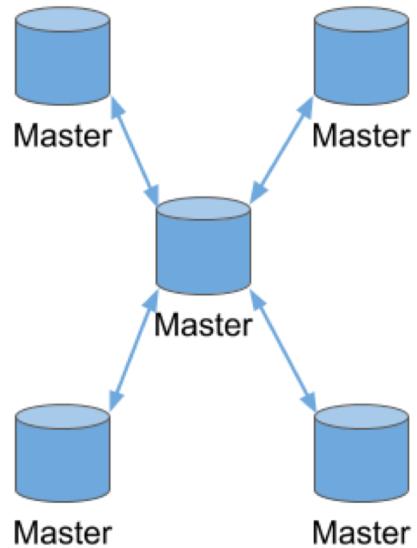
**Merge:** merge values of multiple updates together (e.g. alphabetically or in order of appearance).

**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)

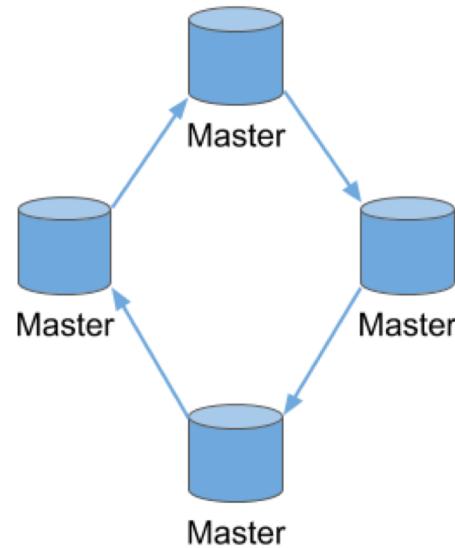


# Multi-Master Replication - Topologies

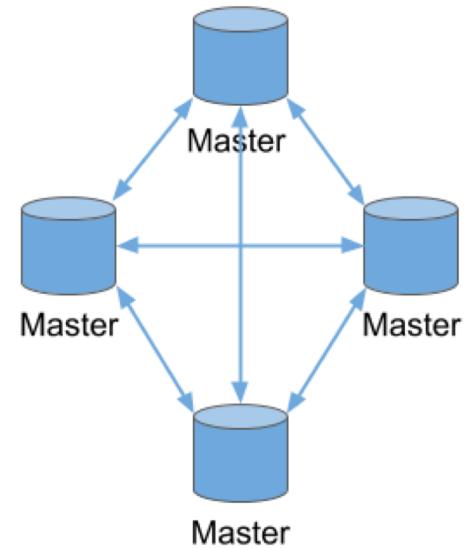
**Star Topology**



**Circle Topology**

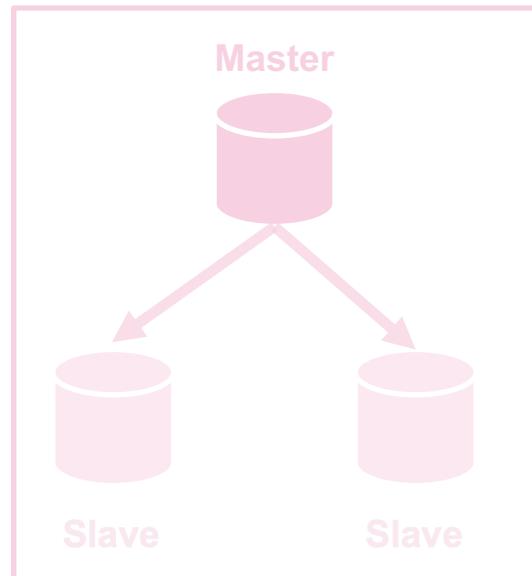


**All-To-All Topology**



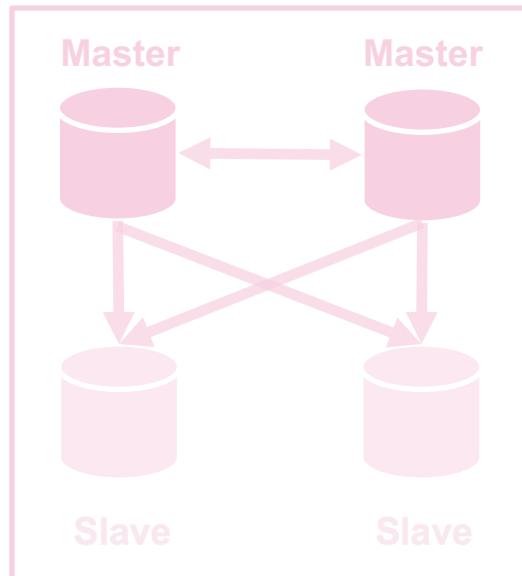
# Masterless Replication

Master-based



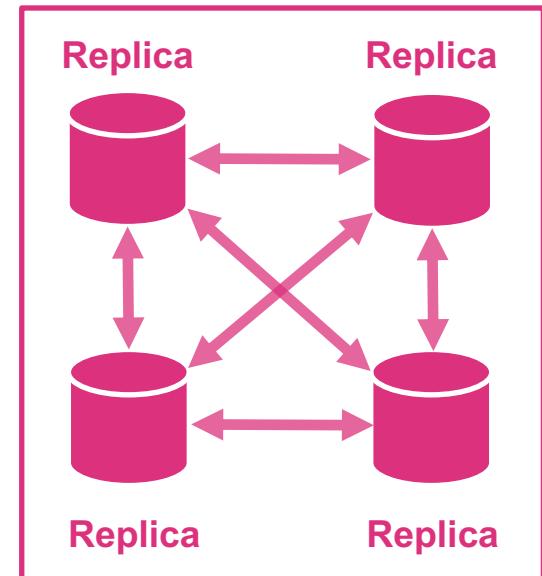
e.g. Oracle, PostgreSQL,  
MySQL...

Multi-Master-based



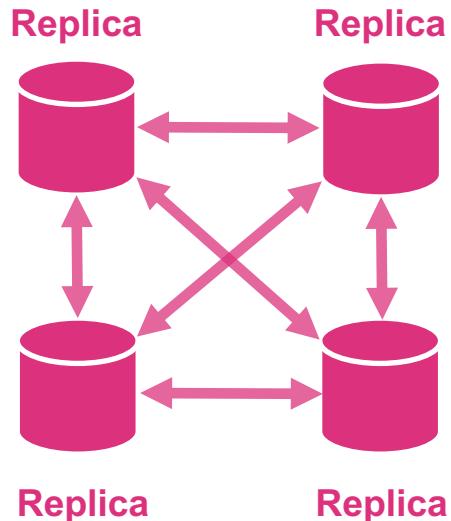
e.g. CouchDB, PostgreSQL,  
MySQL...

Masterless



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

# Masterless 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*)

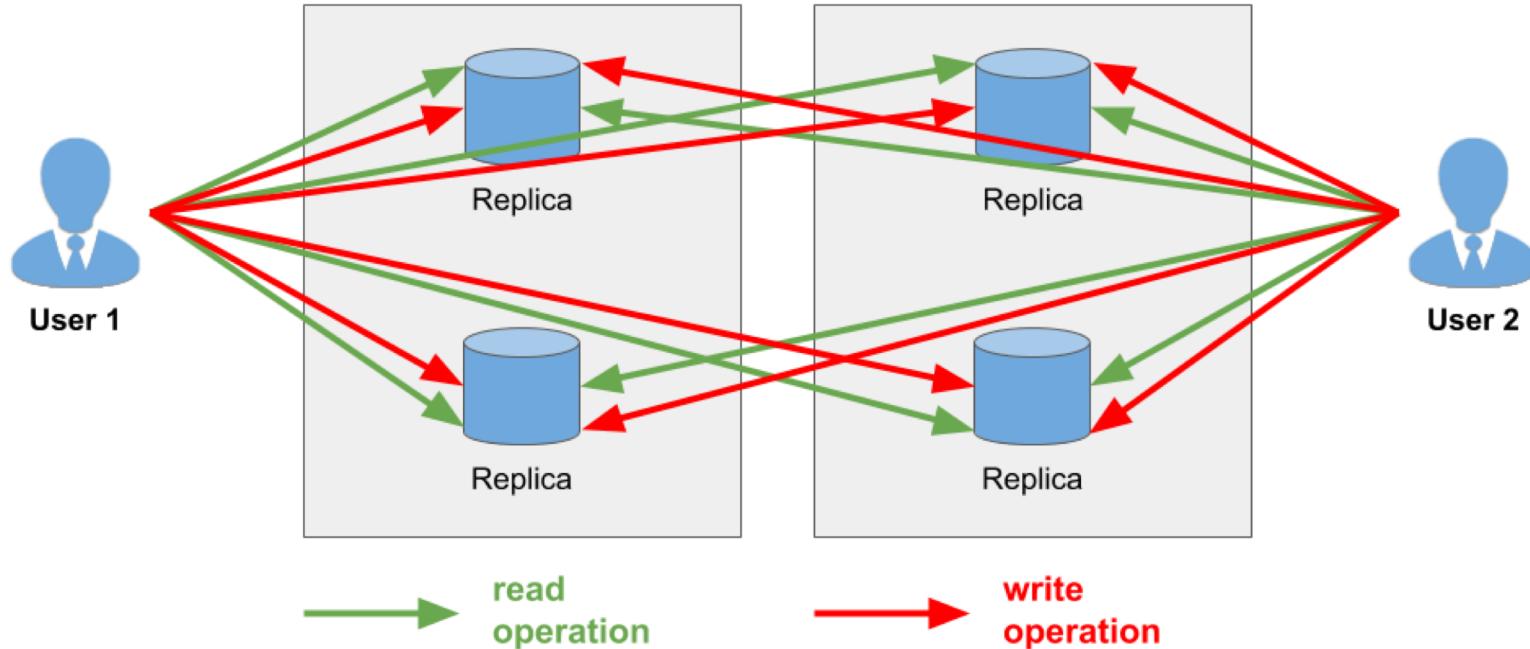
## Disadvantages:

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

**Also known as:**

*Leaderless Replication*

# 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

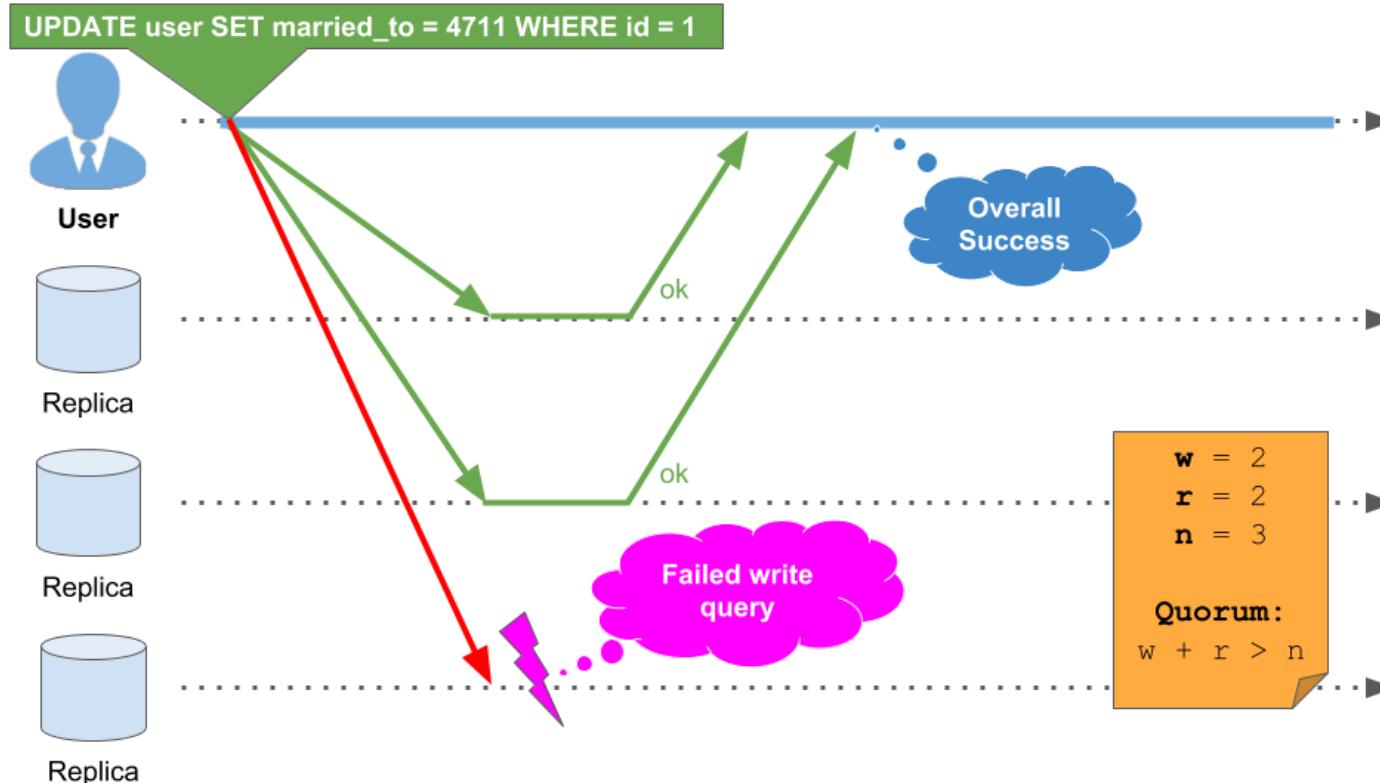
- able to run on **multiple datacenters** (with respect to *quorum*)

# Masterless Replication- Quorums

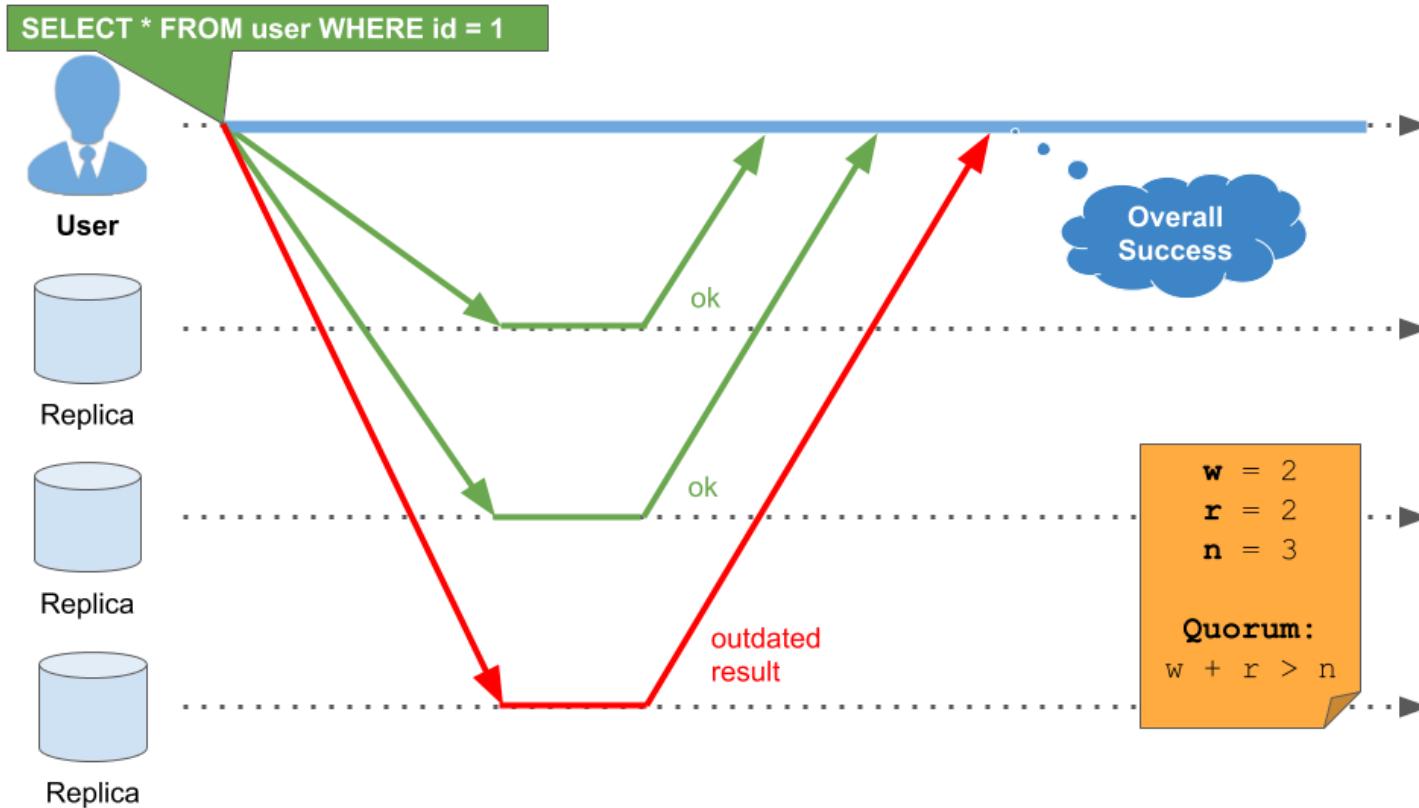
- **r** and **w** are configurable depending on *read/write performance* and *fault-tolerance* you want to achieve
  - smaller **r** = faster reads, slower writes
  - smaller **w** = faster writes, slower reads
- Number of node failures a data-system can handle, is calculated by:
  - **n - r = number** 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 *Cassandra*) 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 succeed 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*.



# Introduction To The Challenges Of Distributed Data-Systems: Partitioning

Basics of Partitioning, Key-Range and Hash Partitioning,  
Partitioning of Secondary Indices, Rebalancing and Lookup of  
Partitions



# Why Partitioning (and Replication)?

Partitioning is the process of continuously **dividing data into subsets** and **distributing it to several nodes** within a data-system. Usually **each record or document** within a partitioned data-system is distributed and **directly assigned to certain partition**. Partitioning serves the purpose of, e.g.:

**Scalability and Performance:** Distributing data to multiple nodes, for instance increases read/write performance and throughput as read/write queries can be distributed to multiple nodes and handled concurrently. In this way it is possible parallelize IO (disk), computing power (CPU) as well as scale the memory usage needed to run a certain operation on a part of the dataset.

**Low Latency:** Using partitioning it is possible to place data close to where it is used (user or consumer applications).

**Availability:** Even if some nodes fail, only parts of the data are offline.



# Replication vs Partitioning

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

**Different purposes, but usually used together**



# Partitioning – Avoid Confusion On Terms

To avoid confusion on the term partition or partitioning, let's list some other terms, you might have heard and which are frequently used synonymously:

- **shards/sharding** - (e.g. *MongoDB*, *ElasticSearch* or *RethinkDB*)
- **Vnodes/Virtual Nodes** - (e.g. *Riak* or *Cassandra*)
- **region** - (e.g. *HBase*)
- **tablet** - (e.g. *BigTable*)
- **vBucket** - (e.g. *Couchbase*)

**Partitioning** and **Replication** are usually used together, especially when building data-intensive applications, as datasets are too big to be stored on a single server or replica, and benefits of replication are required (e.g. redundancy, fault-tolerance or high read/write throughput). This can be achieved by storing partitions of a data set on multiple replica nodes.

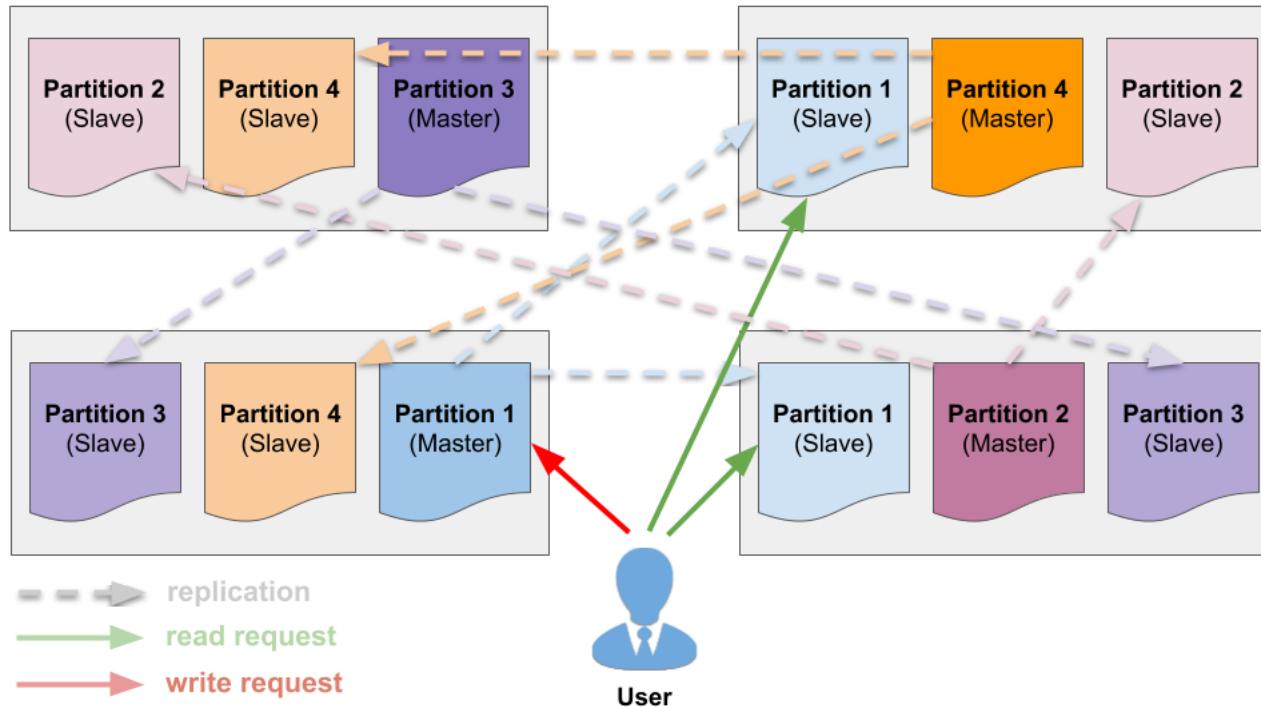


# Partitioning – Avoid Confusion On Terms



As **horizontal** and **vertical partitioning** are mixed up sometimes, it is important to notice: when we speak about **partitioning** within this lecture, we mean **horizontal partitioning**. **Vertical partitioning** is an **approach of traditional relational databases**, usually done by splitting datasets into multiple entities (e.g. tables or databases) and using references (e.g. to achieve normalization).

# Partitioning – An Example



- Partitioning and Replication
  - Using Master-based Replication
  - Each node is master for a certain partition
  - Each partition has 2 slave nodes
- Ensure HA

# Partitioning – Key-Value Data

## 2 Purposes of Partitioning:

- **distributing a dataset,**
- more important: **distribute related load** (read/write queries) evenly among several nodes of a data-system

## This requires:

- a **wise way of determining the partition** of a certain row or document → as it **directly affects the performance of a data-system**
- an **improper chosen distribution key** may cause:
  - **some nodes** to be **idle and/or empty** and
  - a **single node** to be the **processing bottleneck** and hitting its space limitations as all read/write requests end up on that single node
- an **appropriate distribution key** will **distribute the data evenly** and enable the data-system to (theoretically) scale linearly in terms of space utilization and request throughput.



# Partitioning – Key-Value Data (Approaches)

**Key Range Partitioning:** Derive a partition by determining whether a key is inside a certain value range.  
→ More details on next slides.

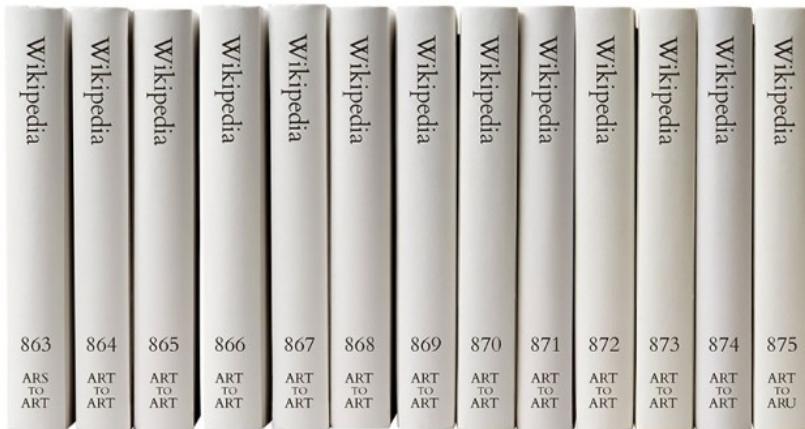
**Partitioning By Hash Value Of A Key:** Derive a partition by a certain hash of a given key to achieve a more even data distribution. → More details on next slides.

**Partitioning By List:** Every partition to be used has an assigned list of values. A related partition is derived from the input dataset by checking whether it contains one of those values. For instance all rows containing iPhone, Samsung Galaxy and HTC One within a column `device_type` are assigned to partition Smartphone. → As no data-intensive system makes use of partitioning by list (as it is very improper to provide even data distribution), we wont discuss this approach in detail.

**Round-Robin Partitioning:** A very simple approach, which ensures even data distribution. For instance assignment to a partition can be achieved by  $n \text{ modulo } p$  ( $n = \text{number of incoming data records}$ ,  $p = \text{number of partitions}$ ). → As no data-intensive system makes use of round-robin partitioning (as for instance the direct access to an individual data record or subset usually requires accessing the whole dataset), we wont discuss this approach in detail.



# Partitioning – Key-Range Partitioning



**Key Range partitioning** is done by:

- defining continuous ranges of keys
  - assigning each range to a certain partition
- If you are aware of the boundaries of each key range, you can easily derive a partition belonging to a certain data record (and in this way node of a data-system) just by using the key of the record.

- This approach can be compared with an encyclopedia, which is partitioned into books, of which everyone stores a certain range of articles partitioned by the first letters of the name of the article.
- For instance the article “Arsenal F.C.” will be found in partition 863 “ARS” - “ART”.
- For instance used by: **RethinkDB** (called *ranged sharding*)

# Partitioning – Key-Range Partitioning

## Strengths:

- Simple
- Range Lookups

## Weaknesses:

- **Datasets Are Changing:** A *key range* partitioning which was suitable in the past might not be appropriate in the future. Expensive rebalancing or even repartitioning might be needed somewhere. For instance web server log files partitioned per ranges of the URL (`/products/[A-B]`, `/products/[C-D]` ... `/products/[Y-Z]`) maybe improper in the future, as some products will have heavier traffic than other products over time (*load skew*).
- **Hotspots:** Keys that seem very appropriate in terms of even distribution at first sight, e.g. partitioning of webserver logfiles over time (by using timestamp of data record), create hotspots as all write requests end up on the same partition (e.g. *today*), a single partition (and node(-s)) will underly heavy load whereas other partitions or rather nodes are idle (*load skew*).
- **Query Performance:** As you do not know the size of a partition beforehand, query performance is unpredictable as well as partition pruning and partitionwise joins are more complex and less efficient.



# Partitioning – Hash Partitioning

**Hash partitioning** is used to spread data efficiently and evenly among several certain partitions. This is achieved by:

- splitting data in a randomized way
  - rather than by using information provided within the dataset (e.g. IDs) or
  - derived by arbitrary factors (e.g. time of data receival)
- The hash value itself is derived by a hash function (on a certain key of a data record) and is used to determine the partition a data records should be saved on.



**Hash Function** is a function which takes input data of arbitrary size and usually provides an output of fixed size. The output of a hash function is called hash, hash value or digest. A hash function needs to be deterministic and uniform. Common use cases for hash functions are cryptography, checksums and partitioning.

# Partitioning – Hash Partitioning (Hash Functions)

**Hash functions** are commonly used for partitioning, as they are:

- **deterministic** - as we need to be able to find records to be saved later on and
- **uniform** - as we want to distribute the data as evenly as possible among the set of available partitions and nodes, even if the inputs of the hash function (e.g. data record key) are very similar.

Unlike cryptography, **partitioning makes use of hash functions** that are **not cryptographilly strong** (as this is not needed) but fast and less CPU consuming.  
Examples of commonly used hash function for distributed data-systems are:

- **MD5** - For instance used and supported by *MySQL* and *Cassandra*.
- **MurmurHash** - For instance supported by *Cassandra*.
- **SHA1** - For instance used and supported by *Riak*.
- **CRC32** - For instance used and supported by *Couchbase*.



# Partitioning – Hash Partitioning (Hash Functions)

As an example for **determinism** and **uniformity** let's take a quick look at **MD5** and how the hash value changes:

- when a single **character is added** to the input value,
- when just a single **character** of the input value is **changed** or
- the **same input** value is **hashed twice**

```
1 marcel$ md5 -s abc
2 MD5 ("abc") = 900150983cd24fb0d6963f7d28e17f72
3 // add a single character ("d")
4 marcel$ md5 -s abcd
5 MD5 ("abcd") = e2fc714c4727ee9395f324cd2e7f331f
6 // change a single character ("d" to "e")
7 marcel$ md5 -s abce
8 MD5 ("abce") = b9c4fe92c2a30ef69833ac8f53eebcec
9 // hash again with same input value
10 marcel$ md5 -s abce
11 MD5 ("abce") = b9c4fe92c2a30ef69833ac8f53eebcec
```

Code Snippet 2.7: Bash Output - *MD5 Hash For Several Input Values*



# Partitioning – Hash Partitioning (Hash Modulo)

**Hash Modulo Partitioning** = take the calculated *hash value V* and calculate  $V \text{ modulo } N$  (*number of partitions*).

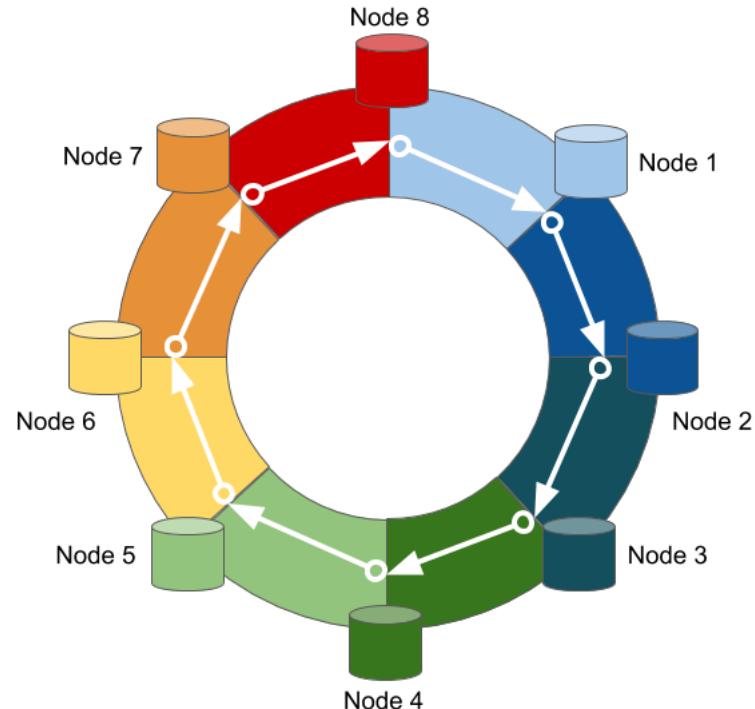
- This allows the data-system to easily distribute and receive records to and from a given number of partitions.
- Major disadvantage in terms of **scalability** and **operability**.
  - Adding nodes results in different partition assignments for a lot of records (depending on size of N), which will require to **shuffle and reassign already saved data again** among all partitions.
- Nevertheless for instance **elasticsearch** makes use of it, a partition (called shard) is derived by: ***shard = hash(routing) % number\_of\_primary\_shards***
- The number of shards for an *index* can increased (by *\_split*) or decreased (by *\_shrink*) some time after creation but this is not a trivial task and will usually require recreating the same or even creating a new index.



# Partitioning – Hash Partitioning (Consistent Hashing)

- assign a *range of hashes* to every partition (and in this way node)
- every record will be stored and read from the partition in charge for a given *range of hash values*.
- imagine **consistent hashing** as a *ring of keys*
- each node ( $N_i$ ) is in charge for serving all hash values  $v$  in between:
  - $i$  and
  - the position  $j$  of its clockwise predecessor  $N_j$

$$j < v < i$$



# Partitioning – Hash Partitioning (Consistent Hashing)

## Strengths:

- Adding/Removing a node → only  $c/N$  keys need to be re-distributed
  - $c$  is the **count of hash values** and
  - $N$  is the **number of nodes** within the data-system

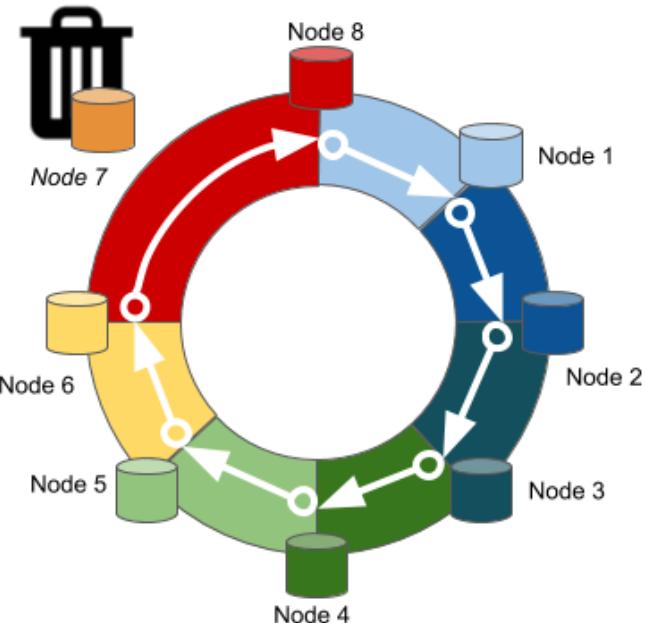
## Weaknesses:

- Range Queries (as keys are randomly distributed among data system)
- e.g. MongoDB provides *key-range partitioning* as well as *hash partitioning* to efficiently serve both use cases

## Used by e.g.:

- Cassandra
- Riak
- VoldemortDB
- DynamoDB

## Example: Removing a Node



# Partitioning – Partitioning Of Secondary Indices

**Secondary Index** = Index in addition to primary index, which:

- is used to accelerates queries
- may not identify records uniquely
- used to lookup records with a certain value/attribute
- **needs to be partitioned as well**

→ Remember previous Facebook Profile example (Primary Key = User ID)  
→ What if you want to acclerate the lookup of cities and comanies? Add a secondary index!



# Partitioning – Local Secondary Indices

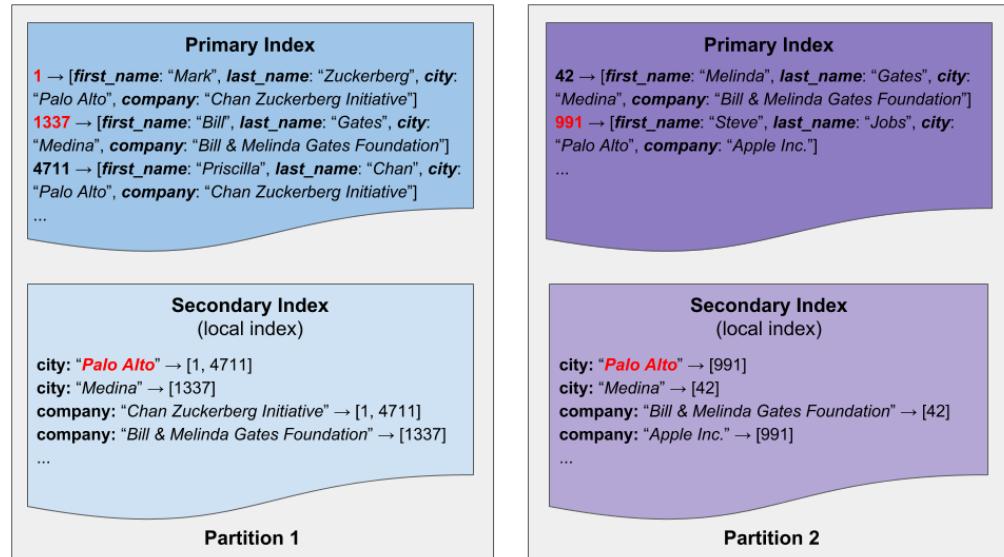
- **LSI = Local Secondary Index**
- every partition manages its own index
- all pointers reference only to local data items

## Strengths:

- maintaining *local index* requires less overhead than *global index*
- INSERT/DELETE/UPDATE performed locally

## Weaknesses:

- maintaining *local index* will compete with *local workload* and affect throughput
- expensive `SELECT: scattering and gathering` as well as related overhead will probably affect read request performance



## E.g. provided by:

- Riak
- Cassandra
- DynamoDB



# Partitioning – Global Secondary Indices

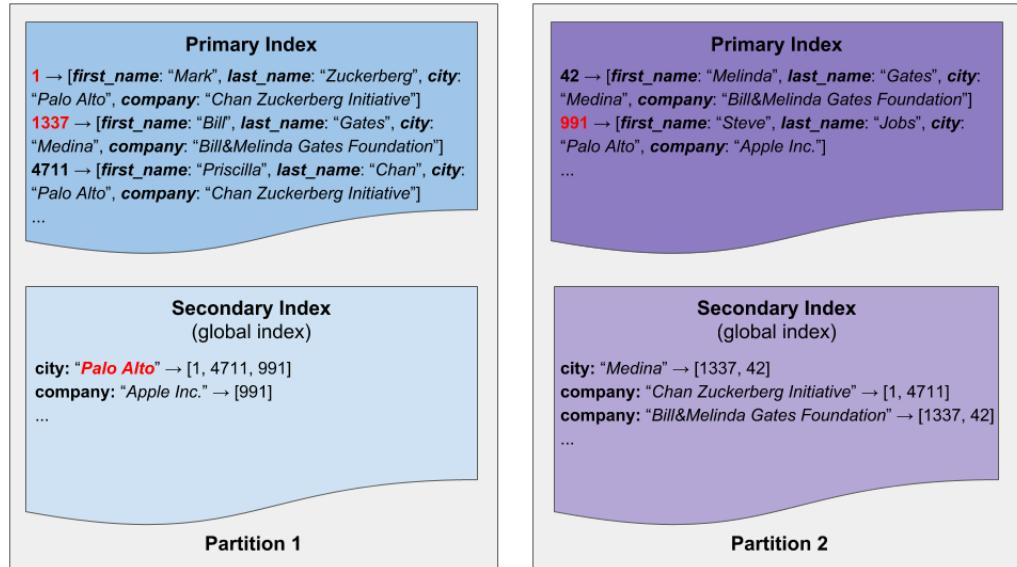
- **GSI = Global Secondary Index**
- Index is partitioned, distributed and stored among several nodes independently of local data records
- pointers reference to local but also remote data records

## Strengths:

- SELECT statements need to query only one index partition
- No scattering and gathering (like using LSI)

## Weaknesses:

- maintaining *global index* requires more overhead than a *local index* → *INSERT/DELETE/E/UPDATE* statements require remote updates
- usually weakens read-consistency as indices update take more time → DynamoDB **eventual consistency**



## E.g. provided by:

- Oracle
- MS SQL Server
- Couchbase
- DynamoDB

# Partitioning – Rebalancing Partitions

## Why?

- Node Failure → other nodes need to take over
- Query load increases → more CPUs and RAM required
- Data Size increases → more RAM and disk space required

## Goals:

- **Minimal Data Shuffle:** Move as less data as possible around nodes during rebalancing
- **Evenly Distribution:** Data distribution should be even after rebalancing.
- **No Availability Impact:** Rebalancing should have as less impact as possible on a running data-system, requiring no downtime.



# Partitioning – Rebalancing Partitions (Hash Modulo)

## Approach:

- hash % n

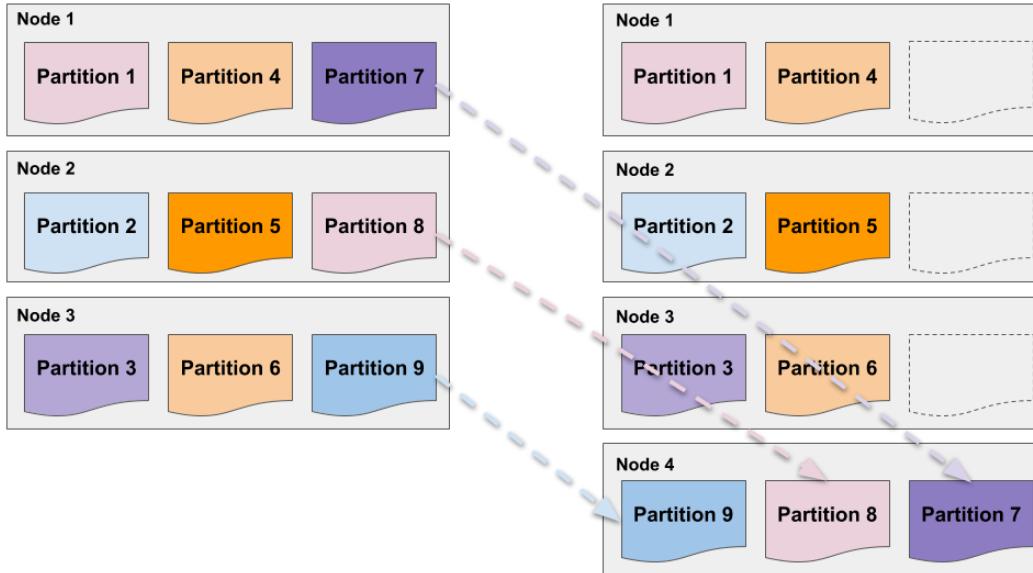
## Contra:

- stupid
- requires shuffling a lot of data, as assignment of partitions to nodes changes significantly

## Used by e.g.:

- shouldn't be used

# Partitioning – Rebalancing Partitions (Fixed Number of Partitions)



## Approach:

- Create more partitions than there are nodes on initial setup
- new nodes „steal“ partitions from old ones

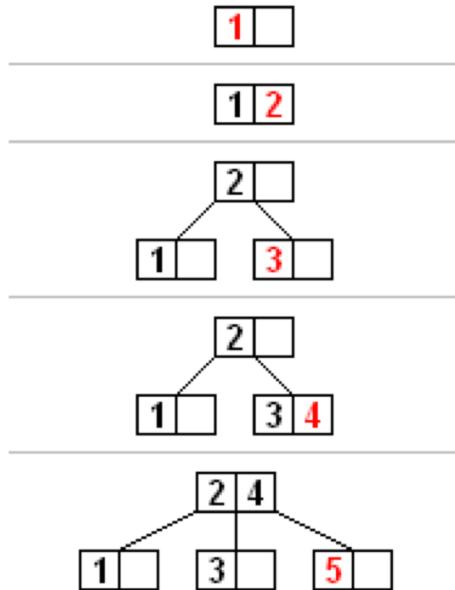
## Pro/Contra:

- partition/node mappings change
- but for less partitions  
→ only partial rewrite of partitions

## Used by e.g.:

- Voldemort
- Riak
- ElasticSearch
- Couchbase

# Partitioning – Rebalancing Partitions (Dynamic No. Of Partitions)



## Approach:

- Create certain number of initial partitions
- Idea similar to B-Trees
- If a partition exceeds a threshold in size, it gets split
- If a partition falls below a threshold in size, merge it with another
- new nodes „steal“ partitions from old ones

## Pro/Contra:

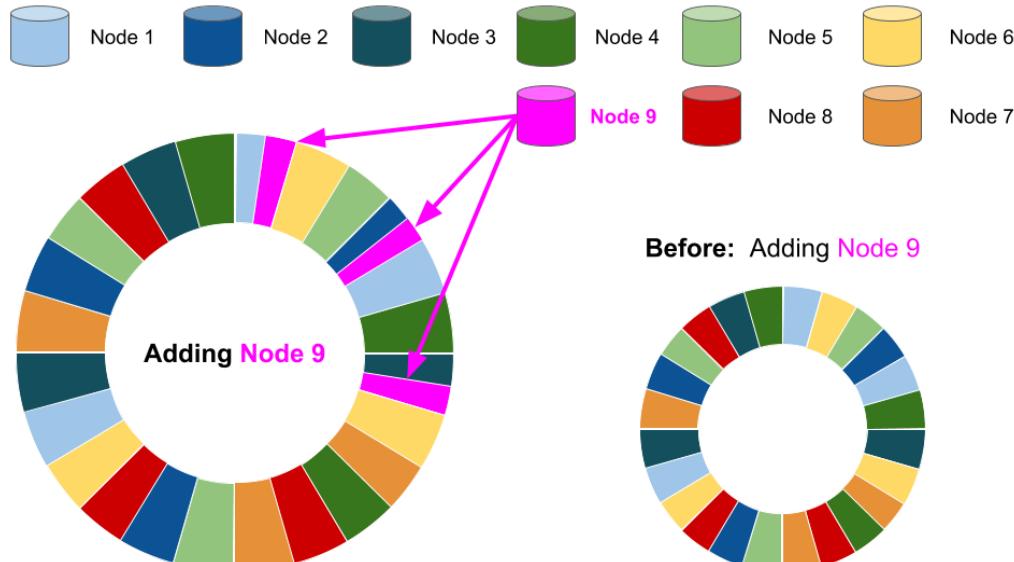
- evenly distribution
- continues overhead during runtime

## Used by e.g.:

- MongoDB
- RethinkDB



# Partitioning – Rebalancing Partitions (Fixed Partition No. Per Node)



## Approach:

- create fixed number of partitions at initial setup
  - new nodes randomly split partitions of old nodes
- steal half of partitions of old nodes

## Pro/Contra:

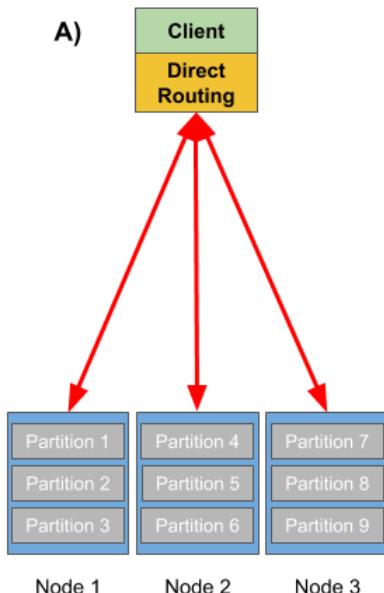
- works well for range partitioning (hash or keys)
- load may be temporarily unbalanced but will be even again over time

## Used by e.g.:

- Cassandra

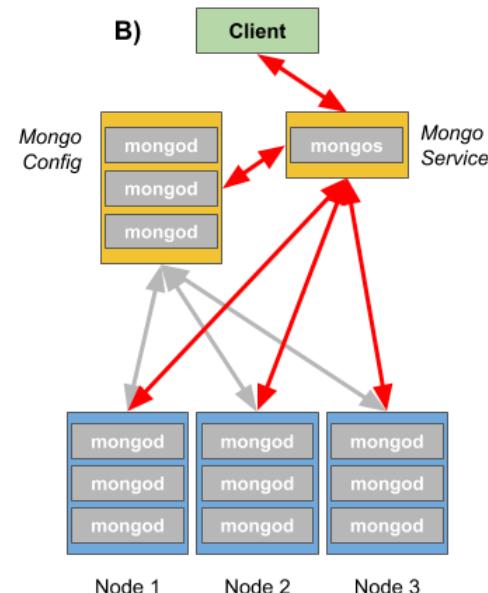
# Partitioning – Partition Lookup

Direct Routing



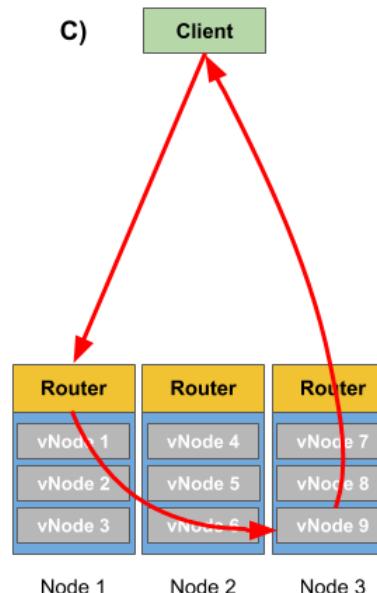
e.g. Microsoft SQL Server

Dedicated Routing Tier



e.g. MongoDB

Ask-Any-Node



e.g. Riak, Cassandra

Legend:



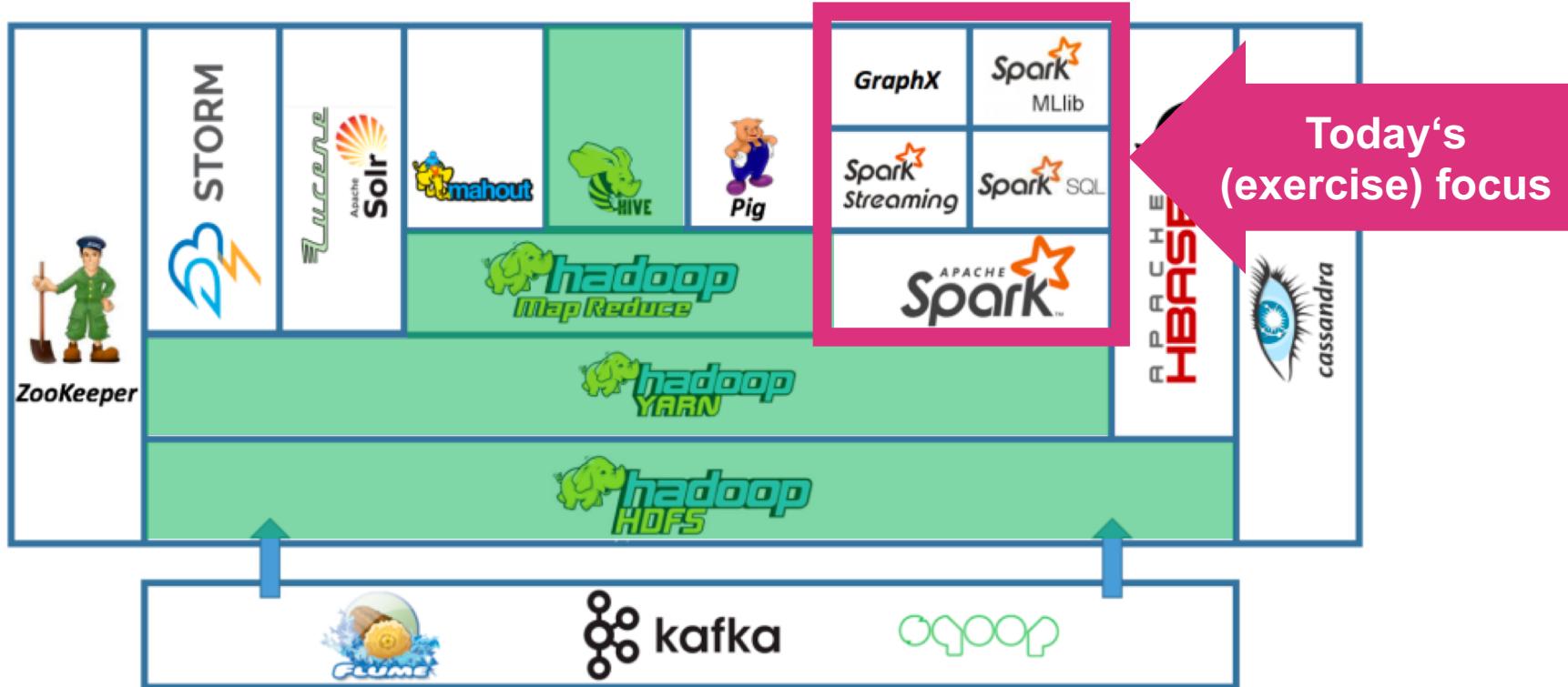


# HandsOn – Spark, Scala, PySpark and Jupyter

A quick Introduction to Spark, Scala, PySpark and  
Jupyter



# The Hadoop Ecosystem

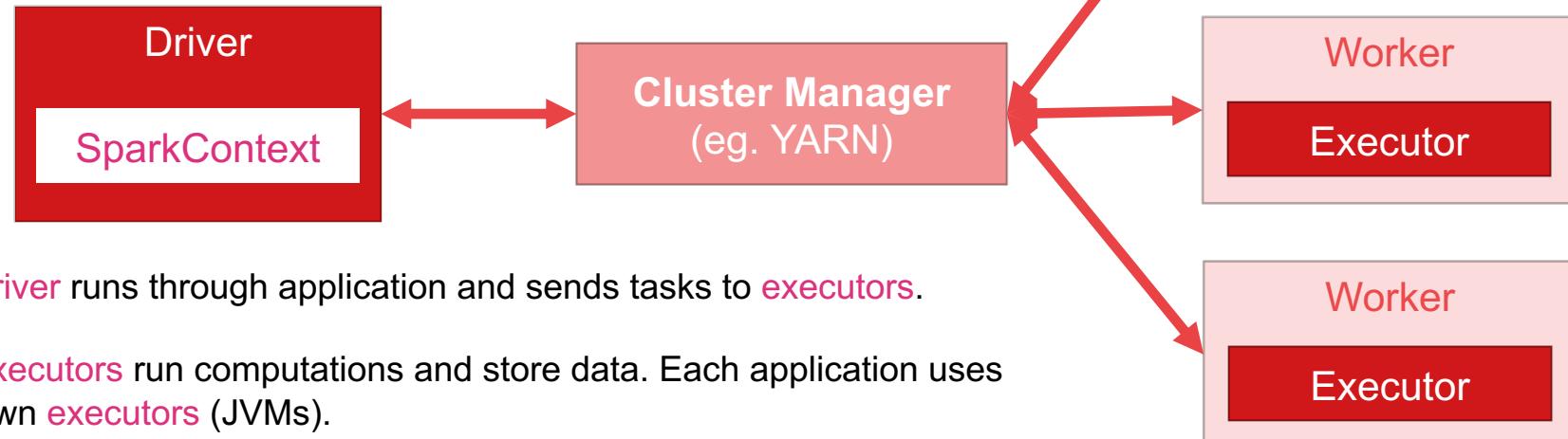


# Spark

<b>API Languages:</b>	Java	Scala	Python	R
<b>APIs and Libraries:</b>	Spark SQL	Spark Streaming	Mlib	GraphX
<b>Spark Core</b>				
<b>Ressource/ Cluster Manager:</b>	Standalone	YARN	Mesos	Kubernetes
<b>Data Source APIs:</b>	HDFS, ElasticSearch, Amazon S3, Azure, Redis, Riak, Couchbase, MongoDB, SQL(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 tasks will be send to another **executor**.

# Spark – RDDs, DataFrame and DataSet

**Supported by:**

**RDD**  
(2011)

Scala, Java, Python

**Idea:**  
- RDD = immutable **distributed** collection of data  
- **partitioned across nodes** of cluster  
- can be **operated in parallel** with a low-level API

**Typed:** typed, no schema

**Use for:** unstructured data

**DataFrame**  
(2013)

Scala, Java, Python

- based on RDD  
- immutable **distributed** collection of data  
- but **organized into columns**  
- higher level abstraction

untyped, schema

semi-structured and structured data

**DataSet**  
(2015)

Scala, Java

- based on DataFrame API, but type-safe  
- immutable **distributed** collection of data  
- high-level API  
- converted into optimized RDD

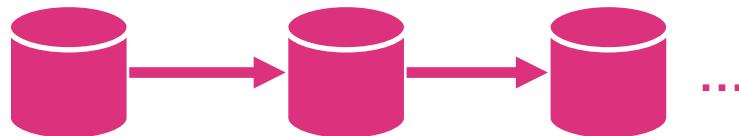
typed, schema

structured data



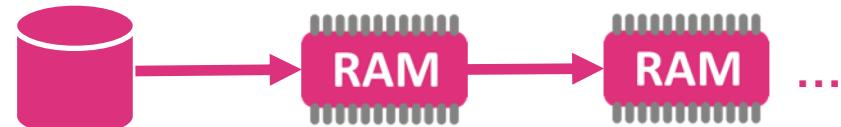
# Hadoop MapReduce vs Spark

## Hadoop MapReduce



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

## Spark

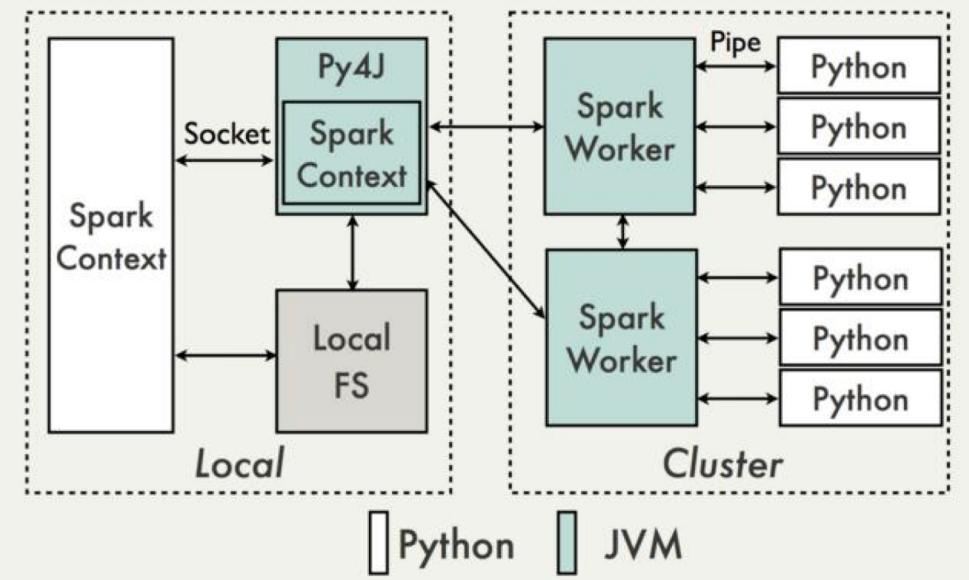


- performs **read** from **RAM** before **every computation (except first)**
- performs **write** on **RAM** after **every computation**
- using distributed **memory**

# PySpark

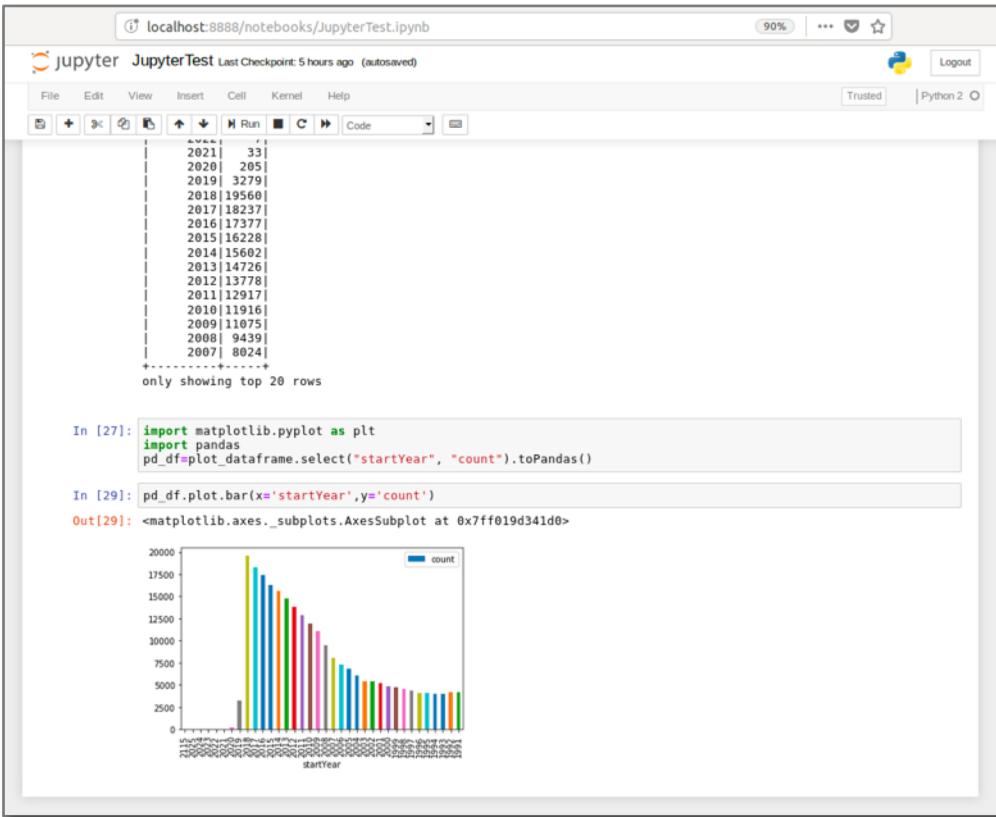


## Data Flow



- 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 corresponds to an RDD in the local JVM
- e.g. **sc.textFile()** in Python will call JavaSparkContext **textFile()**

# Jupyter (Notebooks)



The screenshot shows a Jupyter Notebook interface running on localhost:8888/notebooks/JupyterTest.ipynb. The notebook displays a table of data and two code cells.

**Data Table:**

startYear	count
2021	33
2020	205
2019	3279
2018	19560
2017	18237
2016	17377
2015	16228
2014	15602
2013	14726
2012	13778
2011	12917
2010	11916
2009	11075
2008	9439
2007	8024

only showing top 20 rows

**In [27]:**

```
import matplotlib.pyplot as plt
import pandas
pd_df=plt_dataframe.select("startYear", "count").toPandas()
```

**In [29]:**

```
pd_df.plot.bar(x='startYear',y='count')
```

**Out[29]:**

```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff019d341d0>
```

A bar chart titled 'startYear' showing the count for each year from 2007 to 2021. The y-axis ranges from 0 to 20,000. The bars are colored in a gradient from blue to red.

- 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 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_2019-2020/01_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
```

```
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(- _.value)
```

```
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)
(un,d,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> :

The screenshot shows the Hadoop YARN web interface with the title "RUNNING Applications". The left sidebar includes a logo, navigation links for Cluster, Node Labels, Applications (with a table of status), and Tools. The main content area displays various metrics and a table of running applications.

**Cluster Metrics:**

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	VCores Used	VCores Total	VCores Reserved
5	0	1	4	3	5 GB	8 GB	0 B	3	8	0

**Cluster Nodes Metrics:**

Active Nodes	Decommissioning Nodes	Decommissioned Nodes	Lost Nodes	Unhealthy Nodes	Rebooted Nodes	Shutdown Nodes
1	0	0	0	0	0	0

**Scheduler Metrics:**

Scheduler Type	Scheduling Resource Type	Minimum Allocation	Maximum Allocation	Maximum Cluster Application Priority
Capacity Scheduler	[memory-mb (unit=Mi), vcores]	<memory:1024, vCores:1>	<memory:8192, vCores:4>	0

**Running Applications Table:**

ID	User	Name	Application Type	Queue	Application Priority	StartTime	FinishTime	State	FinalStatus	Running Containers	Allocated CPU Vcores	Allocated Memory MB	Reserved CPU Vcores	Reserved Memory MB	% of Queue	% of Cluster	Progress	Tracking UI	Blacklisted Nodes
application_1572177196643_0005	hadoop	Spark shell	SPARK	default	0	Sun Oct 27 14:18:28 +0100 2019	N/A	RUNNING	UNDEFINED	3	3	5120	0	0	62.5	62.5		ApplicationMaster	0

Showing 1 to 1 of 1 entries



# 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
```

```
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]))
```

```
>>> print(sorted_output[:10])
[(',', 1603), ('und', 509), ('die', 463), ('der', 440), ('ich', 435), ('Und', 400), ('nicht', 346), ('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> :

 **RUNNING Applications** Logged in as: dr.who

**Cluster Metrics**

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	VCores Used	VCores Total	VCores Reserved
8	0	1	7	3	5 GB	8 GB	0 B	3	8	0

**Cluster Nodes Metrics**

Active Nodes	Decommissioning Nodes	Decommissioned Nodes	Lost Nodes	Unhealthy Nodes	Rebooted Nodes	Shutdown Nodes
1	0	0	0	0	0	0

**Scheduler Metrics**

Scheduler Type	Scheduling Resource Type	Minimum Allocation	Maximum Allocation	Maximum Cluster Application Priority
Capacity Scheduler	[memory-mb (unit=M), vcores]	<memory:1024, vCores:1>	<memory:8192, vCores:4>	0

Show 20 entries Search:

ID	User	Name	Application Type	Queue	Application Priority	StartTime	FinishTime	State	FinalStatus	Running Containers	Allocated CPU VCores	Allocated Memory MB	Reserved CPU VCores	Reserved Memory MB	% of Queue	% of Cluster	Progress	Tracking UI	Blacklisted Nodes
application_1572177196643_0009	hadoop	PySparkShell	SPARK	default	0	Sun Oct 27 14:36:36 +0100 2019	N/A	RUNNING	UNDEFINED	3	3	5120	0	0	62.5	62.5	<input type="checkbox"/>	ApplicationMaster	0

Showing 1 to 1 of 1 entries First Previous 1 Next Last



# 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`

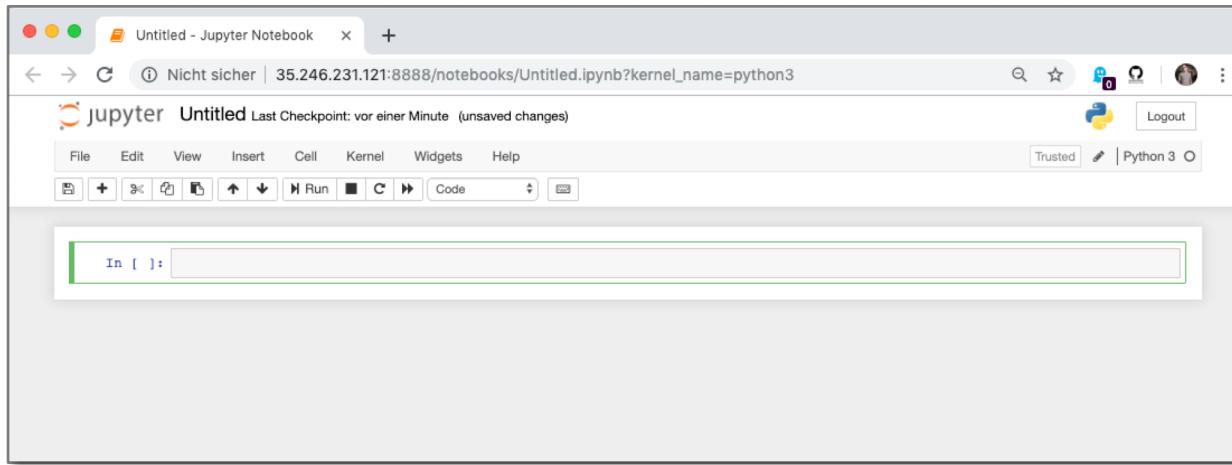
Or copy and paste one of these URLs:

`http://e0f4472dcb12:8888/?token=76b68a3c700b415790b019e07a3dd46a0d068c153a732d27`  
or `http://127.0.0.1:8888/?token=76b68a3c700b415790b019e07a3dd46a0d068c153a732d27`



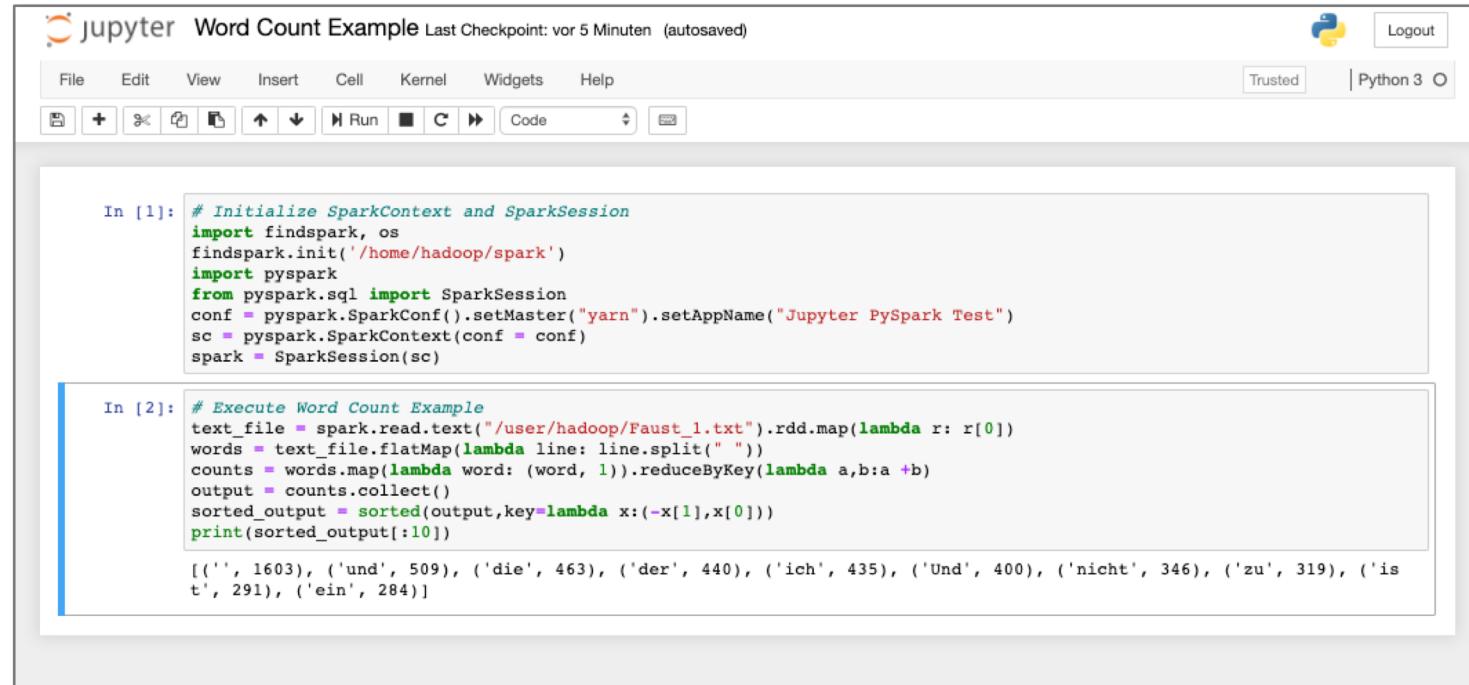
# Start Jupyter

2. Open Notebook in Browser: <http://XXX.XXX.XXX.XXX:8888/?token=XYZXYZXYZ>



# Use Jupyter (Word Count Example)

## 1. Execute previous Word Count example



The screenshot shows a Jupyter Notebook interface with the title "jupyter Word Count Example Last Checkpoint: vor 5 Minuten (autosaved)". The notebook has two code cells:

```
In [1]: # Initialize SparkContext and SparkSession
import findspark, os
findspark.init('/home/hadoop/spark')
import pyspark
from pyspark.sql import SparkSession
conf = pyspark.SparkConf().setMaster("yarn").setAppName("Jupyter PySpark Test")
sc = pyspark.SparkContext(conf = conf)
spark = SparkSession(sc)

In [2]: # Execute Word Count Example
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]))
print(sorted_output[:10])
```

The output of the second cell is:

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



# Use Jupyter (Word Count Example)

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

Logged in as: dr.who

## RUNNING Applications

Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	VCores Used	VCores Total	VCores Reserved
4	0	1	3	3	5 GB	8 GB	0 B	3	8	0

Cluster Nodes Metrics

Active Nodes	Decommissioning Nodes	Decommissioned Nodes	Lost Nodes	Unhealthy Nodes	Rebooted Nodes	Shutdown Nodes
1	0	0	0	0	0	0

Scheduler Metrics

Scheduler Type	Scheduling Resource Type	Minimum Allocation	Maximum Allocation	Maximum Cluster Application Priority
Capacity Scheduler	[memory-mb (unit=Mi), vcores]	<memory:1024, vCores:1>	<memory:8192, vCores:4>	0

Show 20 entries Search:

ID	User	Name	Application Type	Queue	Application Priority	StartTime	FinishTime	State	FinalStatus	Running Containers	Allocated CPU VCores	Allocated Memory MB	Reserved CPU VCores	Reserved Memory MB	% of Queue	% of Cluster	Progress	Tracking UI	Blacklisted Nodes
application_1572185574197_0004	hadoop	Jupyter PySpark Test	SPARK	default	0	Sun Oct 27 15:45:26 +0100 2019	N/A	RUNNING	UNDEFINED	3	3	5120	0	0	62.5	62.5	<input type="checkbox"/>	ApplicationMaster	0

Showing 1 to 1 of 1 entries First Previous 1 Next Last



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

```
In [5]: # Read title.basics.tsv into Spark dataframe
imdb_title_basics_dataframe = spark.read.format('csv').options(
    header='true', delimiter='\t', nullValue='null', inferSchema='true')\
    .load('/user/hadoop/imdb/title_basics/title.basics.tsv')

In [6]: imdb_title_basics_dataframe.printSchema() # Print Schema of title_basics dataframe

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

In [7]: imdb_title_basics_dataframe.show(5) # Show first 5 rows of title_basics dataframe
+-----+-----+-----+-----+-----+-----+
| 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,...|
|tt0000004| short| Un bon bock| Un bon bock| 0| 1892| \N| \N| Animatio
n,Short|
|tt0000005| short| Blacksmith Scene| Blacksmith Scene| 0| 1893| \N| 1| Comed
y,Short|
+-----+-----+-----+-----+-----+-----+
only showing top 5 rows
```

[https://github.com/marcelmittelstaedt/BigData/tree/master/exercises/winter\\_semester\\_2019-2020/04\\_spark\\_pyspark\\_jupyter/PySparkExamples.html](https://github.com/marcelmittelstaedt/BigData/tree/master/exercises/winter_semester_2019-2020/04_spark_pyspark_jupyter/PySparkExamples.html)



[www.marcel-mittelstaedt.com](http://www.marcel-mittelstaedt.com)

# PySpark Operations

## 2. Basic PySpark: Operations on DataFrames (filter, aggregate, sort...):

```
In [5]: imdb_title_basics_dataframe.count() # show number of rows within title_basics dataframe  
Out[5]: 6262637
```

```
In [11]: # Get column titleTypes values with counts and ordered descending  
from pyspark.sql.functions import desc  
imdb_title_basics_dataframe.groupBy("titleType").count().orderBy(desc("count")).show()
```

titleType	count
tvEpisode	4392190
short	707983
movie	533869
video	245253
tvSeries	173337
tvMovie	120824
tvMiniSeries	28114
tvSpecial	25479
videoGame	24322
tvShort	11266

```
In [28]: # Calculate average Movie length in minutes  
from pyspark.sql.functions import avg  
imdb_title_basics_dataframe.filter(imdb_title_basics_dataframe['titleType']=='movie')\\  
.agg(avg('runtimeMinutes')).show()
```

avg(runtimeMinutes)
88.34293031750659

[https://github.com/marcelmittelstaedt/BigData/tree/master/exercises/winter\\_semester\\_2019-2020/04\\_spark\\_pyspark\\_jupyter/PySparkExamples.html](https://github.com/marcelmittelstaedt/BigData/tree/master/exercises/winter_semester_2019-2020/04_spark_pyspark_jupyter/PySparkExamples.html)



# PySpark Operations

## 3. PySpark SQL: Join DataFrames:

```
In [12]: # Read title.ratings.tsv into Spark dataframe  
imdb_title_ratings_dataframe = spark.read.format('csv').options(\n    header='true', delimiter='\t', nullValue='null', inferSchema='true')\\  
.load('/user/hadoop/imdb/title_ratings/title.ratings.tsv')
```

```
In [13]: imdb_title_ratings_dataframe.printSchema() # Print Schema of title_ratings dataframe  
  
root  
|-- tconst: string (nullable = true)  
|-- averageRating: double (nullable = true)  
|-- numVotes: integer (nullable = true)
```

```
In [14]: imdb_title_ratings_dataframe.show(5) # Show first 5 rows of title_ratings dataframe  
  
+-----+-----+  
| tconst|averageRating|numVotes|  
+-----+-----+  
| tt0000001|      5.6|     1543|  
| tt0000002|      6.1|      186|  
| tt0000003|      6.5|     1201|  
| tt0000004|      6.2|      114|  
| tt0000005|      6.1|     1921|  
+-----+-----+  
only showing top 5 rows
```

```
In [15]: # JOIN Data Frames  
title_basics_and_ratings_df = imdb_title_basics_dataframe.join(imdb_title_ratings_dataframe, \\  
imdb_title_basics_dataframe.tconst == imdb_title_ratings_dataframe.tconst)
```

```
In [25]: top_tvseries=title_basics_and_ratings_df.filter(title_basics_and_ratings_df['titleType']=='tvSeries')\\  
.filter(title_basics_and_ratings_df['numVotes'] > 200000)\\  
.orderBy(desc('averageRating'))\\  
.select('originalTitle', 'startYear', 'endYear', 'averageRating', 'numVotes')  
top_tvseries.show(5)
```

```
+-----+-----+-----+-----+  
| originalTitle|startYear|endYear|averageRating|numVotes|  
+-----+-----+-----+-----+  
| Breaking Bad|   2008|  2013|       9.5| 1271805|  
| Game of Thrones| 2011|  2019|       9.4| 1598911|  
| Rick and Morty| 2013|     \N|       9.3|  296206|  
| The Wire|   2002|  2008|       9.3|  254371|  
| Avatar: The Last ...| 2005|  2008|       9.2|  200077|  
+-----+-----+-----+-----+  
only showing top 5 rows
```

[https://github.com/marcelmittelstaedt/BigData/tree/master/exercises/winter\\_semester\\_2019-2020/04\\_spark\\_pyspark\\_jupyter/PySparkExamples.html](https://github.com/marcelmittelstaedt/BigData/tree/master/exercises/winter_semester_2019-2020/04_spark_pyspark_jupyter/PySparkExamples.html)



# PySpark Operations

## 4. Basic PySpark: Save DataFrames as File to HDFS:

```
In [27]: top_tvseries.write.format('parquet')\ # Could also be CSV, but parquet requires less space
          .partitionBy('startYear')\ # partition data by startYear (not mandatory but useful)
          .mode('overwrite')\ # overwrite existing data
          .save('/user/hadoop/imdb/top_tvseries') # where to save
```

### Browse Directory

/user/hadoop/imdb/top\_tvseries

Go!



Show 25 entries

Search:

Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
-rw-r--r--	hadoop	supergroup	0 B	Oct 27 17:16	1	128 MB	_SUCCESS
drwxr-xr-x	hadoop	supergroup	0 B	Oct 27 17:16	0	0 B	startYear=1989
drwxr-xr-x	hadoop	supergroup	0 B	Oct 27 17:16	0	0 B	startYear=1994
drwxr-xr-x	hadoop	supergroup	0 B	Oct 27 17:16	0	0 B	startYear=1997
drwxr-xr-x	hadoop	supergroup	0 B	Oct 27 17:16	0	0 B	startYear=1999
drwxr-xr-x	hadoop	supergroup	0 B	Oct 27 17:16	0	0 B	startYear=2001
drwxr-xr-x	hadoop	supergroup	0 B	Oct 27 17:16	0	0 B	startYear=2002



# PySpark Operations

## 5. PySpark SQL: Save DataFrames as (temporary Hive) Table:

```
In [17]: title_basics_and_ratings_df.select('originalTitle', 'titleType', 'startYear', \
                                         'endYear', 'numVotes', 'averageRating')\
                                         .write.saveAsTable('movies_and_ratings')
```

## 6. PySpark SQL : Read from (temporary Hive) Table:

```
In [18]: result_df = spark.sql("""SELECT originalTitle, averageRating FROM movies_and_ratings WHERE
                                         numVotes > 200000 AND titleType= 'movie' AND averageRating > 8.5 AND startYear > 2010
                                         ORDER BY averageRating DESC LIMIT 10""")
                                         ).show(10)
```

originalTitle	averageRating
Joker	8.9
Interstellar	8.6



# PySpark Operations

## 7. Basic Python: Plot results:

```
In [6]: good_movies_df = title_basics_and_ratings_df.filter(title_basics_and_ratings_df['titleType']=='movie')\
    .filter(title_basics_and_ratings_df['numVotes'] > 200000) \
    .filter(title_basics_and_ratings_df['startYear'] > 1990) \
    .groupBy('startYear') \
    .count() \
    .orderBy('startYear')

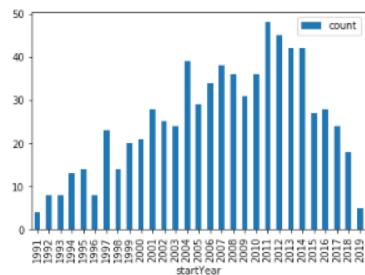
good_movies_df.show(5)
```

startYear	count
1991	4
1992	8
1993	8
1994	13
1995	14

only showing top 5 rows

```
In [11]: import matplotlib.pyplot as plt
import pandas
pandas_dataframe = good_movies_df.select('startYear', 'count').toPandas()
pandas_dataframe.plot.bar(x='startYear', y='count')
```

```
Out[11]: <matplotlib.axes._subplots.AxesSubplot at 0x7fd4e21a2240>
```



# Exercises

Use PySpark Shell or Jupyter Notebooks on  
PySpark to solve exercises



# PySpark Exercises - IMDB

1. Execute Tasks of previous HandsOn Slides
2. Use PySpark or Jupyter on PySpark to answer following questions:
  - a) How many **tv series** are within the IMDB dataset?
  - b) Who is the **youngest** actor/writer/... within the dataset?
  - 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 better than 8
    - have been voted more than 100.000 timessave result (DataFrame) back to HDFS as CSV File.



# PySpark Exercises - IMDB

2. Use PySpark or Jupyter on PySpark to answer following questions:

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

e) create following plot with result of c)  
(plot visualizes the amount of good  
movies per year since 2001)

