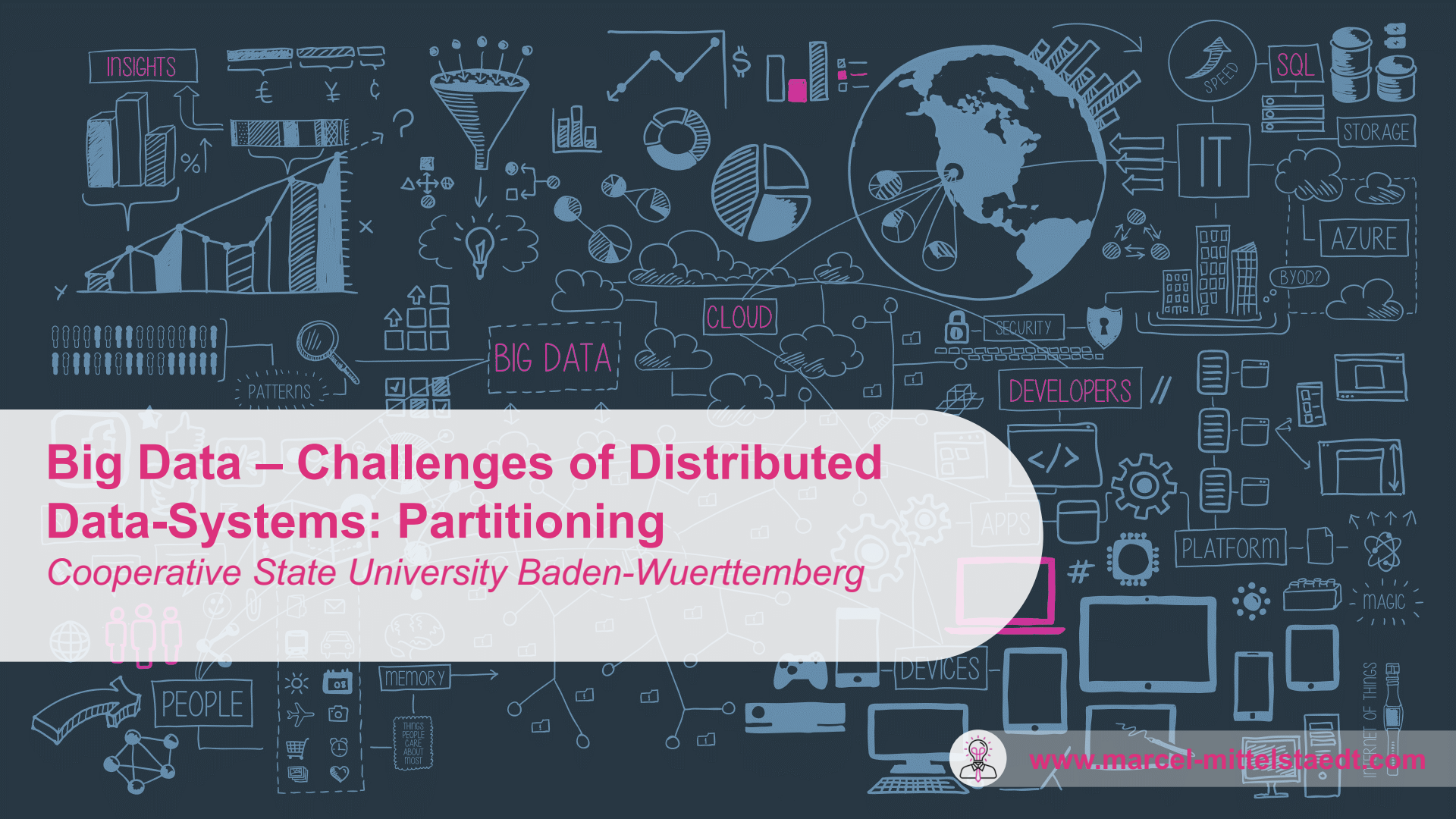


Big Data – Challenges of Distributed Data-Systems: Partitioning

Cooperative State University Baden-Wuerttemberg

Agenda – 22.10.2018

01

Presentation and Discussion: Exercise Of Last Lecture

HiveQL, HDFS/Hive Partitioning, HiveServer2

02

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

03

HandsOn – Spark, Scala, PySpark and Jupyter

A quick Introduction to Spark, Scala, PySpark and Jupyter

04

Exercise – Working with Spark, Scala, PySpark and Jupyter on IMDb dataset.

Install and configure Spark, Scala, PySpark and Jupyter as well as some practical exercises on IMDb dataset.

Schedule

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

Solution – Exercise 03

HiveQL, HDFS/Hive Partitioning, HiveServer2

Solution

Prerequisites:

- Start HDFS, YARN and Hive or HiveServer2
- Execute all preparation and example tasks of previous HandsOn slides of last lecture

Solution

Exercise 2:

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

```
hive >    CREATE EXTERNAL TABLE IF NOT EXISTS imdb_actors_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 2:

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

```
hive >    INSERT OVERWRITE TABLE imdb_actors_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 imdb_actors a WHERE a.death_year IS NULL
```


Solution

Exercise 2:

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

```
hive >    INSERT OVERWRITE TABLE imdb_actors_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 imdb_actors a WHERE a.death_year IS NOT NULL
```


Solution

Exercise 2:

2.4 Check Results:

```
hadoop fs -ls /user/hadoop/imdb/actors_partitioned
drwxr-xr-x   - hadoop supergroup          0 2018-10-09 20:48 /user/hadoop/imdb/actors_partitioned/
partition_is_alive=alive
drwxr-xr-x   - hadoop supergroup          0 2018-10-09 20:54 /user/hadoop/imdb/actors_partitioned/
partition_is_alive=dead
```


Solution

Exercise 2:

2.4 Check Results:

```
select * FROM imdb_actors_partitioned where partition_is_alive = 'dead' LIMIT 100
```

Result

select * FROM imdb_actors_partitioned where p | Enter a SQL expression to filter results (use Ctrl+Space)

	imdb_actors_partiti	imdb_actors_partitioned.prima	imdb_actors_s	imdb_actors_partitio	imdb_actors_partitioned.primary_professio	imdb_actors_partitioned.known_for_titles
6	nm0000006	Ingrid Bergman	1.915	1982	actress,soundtrack,producer	tt0038109,tt0034583,tt0038787,tt0071877
7	nm0000007	Humphrey Bogart	1.899	1957	actor,soundtrack,producer	tt0034583,tt0043265,tt0037382,tt0033870
8	nm0000008	Marlon Brando	1.924	2004	actor,soundtrack,director	tt0047296,tt0078788,tt0070849,tt0068646
9	nm0000009	Richard Burton	1.925	1984	actor,producer,soundtrack	tt0061184,tt0059749,tt0087803,tt0057877
10	nm0000010	James Cagney	1.899	1986	actor,soundtrack,director	tt0029870,tt0031867,tt0035575,tt0055256
11	nm0000011	Gary Cooper	1.901	1961	actor,soundtrack,producer	tt0027996,tt0052876,tt0034167,tt0044706
12	nm0000012	Bette Davis	1.908	1989	actress,soundtrack,make_up_department	tt0056687,tt0035140,tt0031210,tt0042192
13	nm0000015	James Dean	1.931	1955	actor,miscellaneous	tt0049261,tt0045458,tt0048028,tt0048545
14	nm0000016	Georges Delerue	1.925	1992	composer,soundtrack,music_department	tt0079477,tt0069946,tt0096320,tt0091763
15	nm0000017	Marlene Dietrich	1.901	1992	soundtrack,actress,music_department	tt0055031,tt0052311,tt0040367,tt0051201
16	nm0000019	Federico Fellini	1.920	1993	writer,director,assistant_director	tt0053779,tt0071129,tt0056801,tt0047528
17	nm0000020	Henry Fonda	1.905	1982	actor,producer,soundtrack	tt0032551,tt0050083,tt0064116,tt0082846
18	nm0000021	Joan Fontaine	1.917	2013	actress,soundtrack,producer	tt0032976,tt0039504,tt0036969,tt0034248
19	nm0000022	Clark Gable	1.901	1960	actor,soundtrack,producer	tt0052278,tt0025316,tt0026752,tt0031381

Solution

Exercise 3:

3.1 Create table `imdb_movies_and_ratings_partitioned` partitioned by column `partition_year` using fields of table `imdb_movies` and `imdb_ratings`:

```
hive >     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 3:

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

```
hive >      SET hive.exec.dynamic.partition.mode=nonstrict;
            INSERT OVERWRITE TABLE imdb_movies_and_ratings_partitioned partition(partition_year)
            SELECT
                  m.tconst,
                  m.title_type,
                  m.primary_title,
                  m.original_title,
                  m.is_adult,
                  m.start_year,
                  m.end_year,
                  m.runtime_minutes,
                  m.genres,
                  r.average_rating,
                  r.num_votes,
                  m.start_year
            FROM imdb_movies m JOIN imdb_ratings r ON (m.tconst = r.tconst)
```

Solution

Exercise 3:

3.3 Check Results:

```
hadoop fs -ls /user/hadoop/imdb/movies_and_ratings_partitioned
[…]
drwxr-xr-x   - hadoop supergroup          0 2018-10-09 21:14 /user/hadoop/imdb/movies_and_ratings_
partitioned/partition_year=2012
drwxr-xr-x   - hadoop supergroup          0 2018-10-09 21:14 /user/hadoop/imdb/movies_and_ratings_
partitioned/partition_year=2013
drwxr-xr-x   - hadoop supergroup          0 2018-10-09 21:14 /user/hadoop/imdb/movies_and_ratings_
partitioned/partition_year=2014
drwxr-xr-x   - hadoop supergroup          0 2018-10-09 21:14 /user/hadoop/imdb/movies_and_ratings_
partitioned/partition_year=2015
drwxr-xr-x   - hadoop supergroup          0 2018-10-09 21:14 /user/hadoop/imdb/movies_and_ratings_
partitioned/partition_year=2016
drwxr-xr-x   - hadoop supergroup          0 2018-10-09 21:14 /user/hadoop/imdb/movies_and_ratings_
partitioned/partition_year=2017
drwxr-xr-x   - hadoop supergroup          0 2018-10-09 21:14 /user/hadoop/imdb/movies_and_ratings_
partitioned/partition_year=2018
```


Solution

Exercise 3:

3.3 Check Results:

```
select * FROM imdb_movies_and_ratings_partitioned mrp where mrp.partition_year = '2018' LIMIT 100
```

Result

select * FROM imdb_movies_and_ratings_partitionec | Enter a SQL expression to filter results (use Ctrl+Space)

mrp.tit	mrp.primary_title	mrp.original_title	mrp.is	mrp.start	mrp	mrp.r	mrp.genres	mrp.a	mrp.num	mrp.partition_year
movie	The Other Side of the Wind	The Other Side of the Wind	0	2.018	[NULL]	122	Comedy,Drama	7,5	174	2.018
movie	In Memoriam Alfons Vranckx	Le Saigneur est avec nous	0	2.018	[NULL]	90	Action,Adventure,Comedy	6,4	13	2.018
tvMovie	The Patchwork Girl of Oz	The Patchwork Girl of Oz	0	2.018	[NULL]	[NULL]	Adventure,Animation,Famil	5,1	14	2.018
movie	Nappily Ever After	Nappily Ever After	0	2.018	[NULL]	98	Comedy,Drama,Romance	6,5	2.354	2.018
movie	A Million Little Pieces	A Million Little Pieces	0	2.018	[NULL]	113	Drama	8,2	79	2.018
movie	Duel of Legends	Duel of Legends	0	2.018	[NULL]	[NULL]	Action,Drama,Sport	5,9	34	2.018
movie	Dukun	Dukun	0	2.018	[NULL]	108	Horror,Thriller	6,9	238	2.018
movie	Suspiria	Suspiria	0	2.018	[NULL]	152	Fantasy,Horror,Mystery	6,5	673	2.018
movie	Farming	Farming	0	2.018	[NULL]	107	Drama	7,5	40	2.018
movie	Death Wish	Death Wish	0	2.018	[NULL]	107	Action,Crime,Drama	6,4	37.212	2.018
movie	Beautiful Boy	Beautiful Boy	0	2.018	[NULL]	112	Biography,Drama	7,2	412	2.018
movie	Where Hands Touch	Where Hands Touch	0	2.018	[NULL]	122	Drama,Romance,War	5,1	101	2.018
movie	Den of Thieves	Den of Thieves	0	2.018	[NULL]	140	Action,Crime,Drama	7,0	52.408	2.018
movie	London Fields	London Fields	0	2.018	[NULL]	118	Crime,Mystery,Thriller	5,4	101	2.018
movie	The Guernsey Literary and Pot	The Guernsey Literary and Pota	0	2.018	[NULL]	124	Drama,History,Romance	7,4	12.586	2.018

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 pur pose 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:	**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

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:

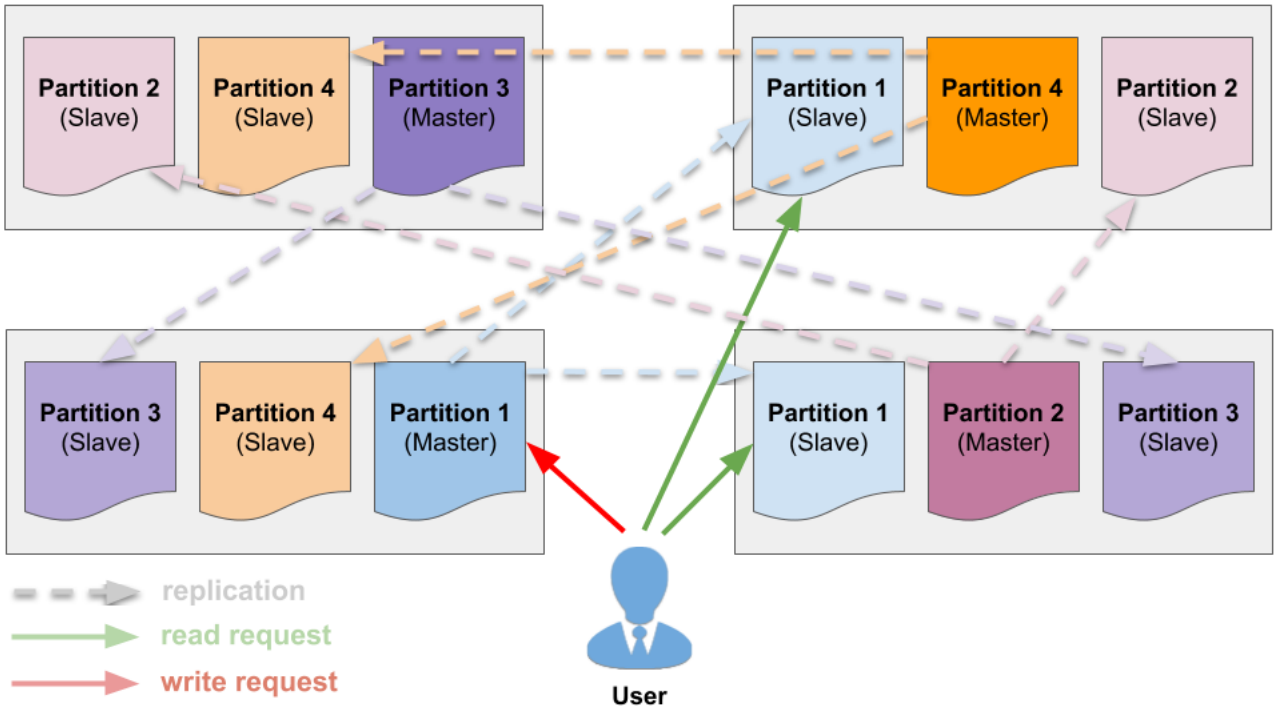
- ***shards/sharding*** - (e.g. *MongoDB*, *ElasticSearch* or *RethinkDB*)
- ***Vnodes/Virtual Nodes*** - (e.g. *Riak* or *Cassandra*)
- ***region*** - (e.g. *HBase*)
- ***tablet*** - (e.g. *BigTable*)
- ***vBucket*** - (e.g. *Couchbase*)

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 through put). 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 partioning****.* ***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 choosen 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 linearily in terms of space utilization and request throug hput.

Partitioning – Key-Value Data (Approaches)

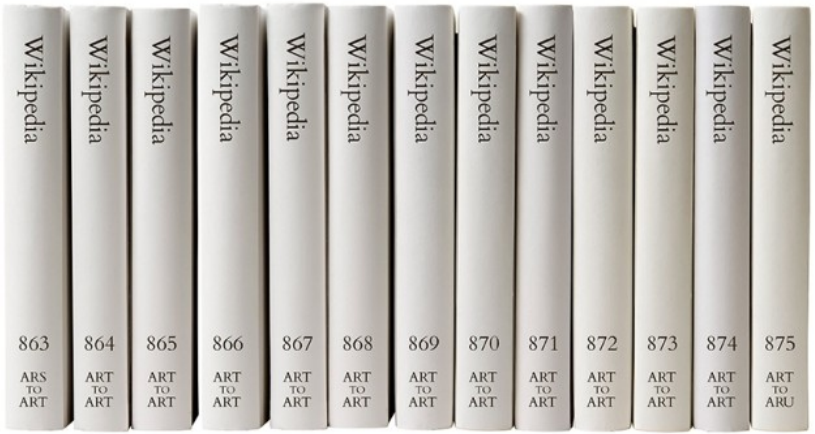
Key Range Partitioning: Derive a partition by determining whether a key is inside a certain value range.
→ *More details on next slides.*

Partitioning By Hash Value Of A Key: Derive a partition by a certain hash of a given key to achieve a mor e 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 Smart-phone. → *As no data-intensive system makes use of partitioning by list (as it is very improper to provide even data distributio n), 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** modulo **p** (*n = number of inccoming data records, p = 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 continues 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 dat a-system) just by using the key of the record.

→ This approach can be compared with an encyclopedia, which is partitioned into books, of which every-one 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 appropriat e in the future. Expensive rebalancing or even repartitioning might be needed somewhen.
  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 impredictable 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 usua lly provides an output of fixed size. The output of a hash function is called has h, hash value or digest. A hash function needs to be deterministic and unifor m. 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 modulo N`** (*number of partitions*).

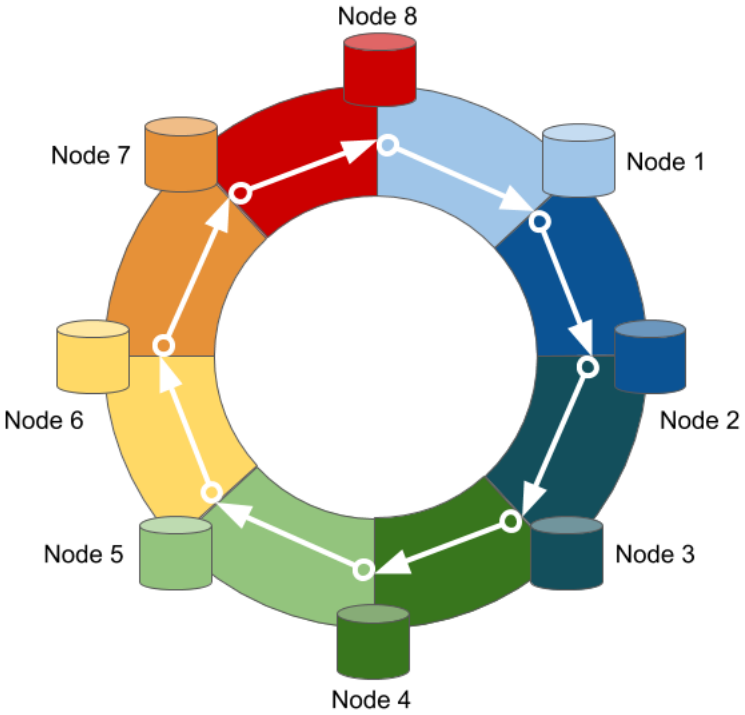
→ This allows the the data-system to easily distribute and receive records to and from a given number of partitions.
→ Major disadvantage in terms of **scalability** and **operability**.
→ Adding nodes results in different partition assignments for a lot of records (depending on size of N), which will 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 (***Ni***) is in charge for serving all hash values ***v*** in between:
  - ***i*** and
  - the position ***j*** of its clockwise pre-decessor ***Nj***

`J < v < i`



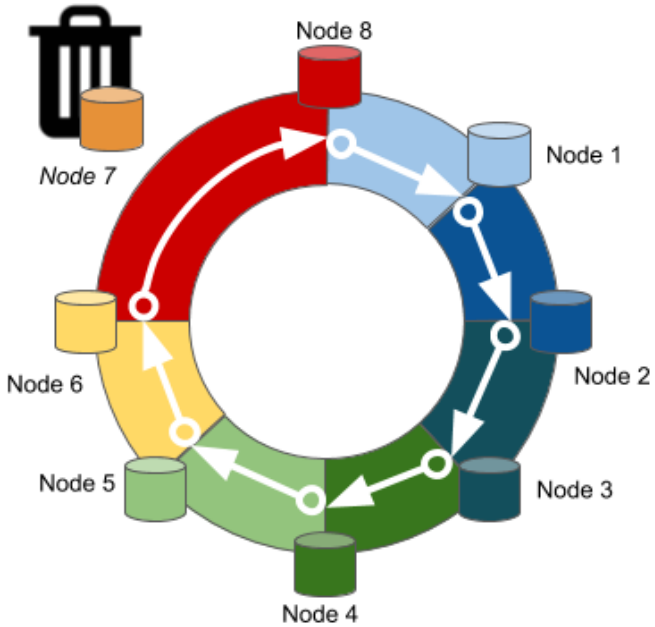
Partitioning – **Hash Partitioning (Consistent Hashing)**

Strenghts:
- 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 amon 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 seco ndary index!

Partitioning – **Local Secondary Indices**

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

Strenths:

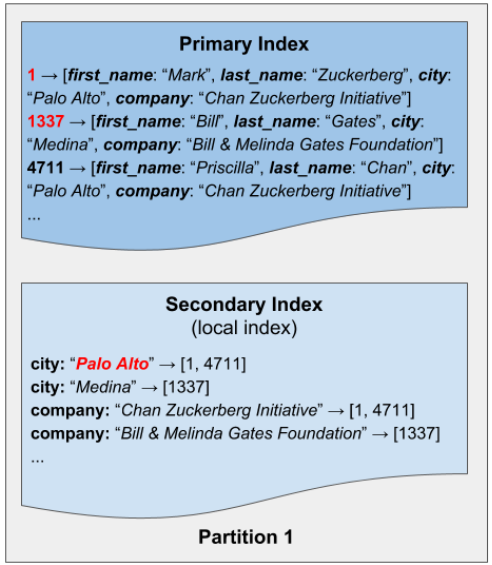
→ maintaining *local index* requires less over-head than *global index*

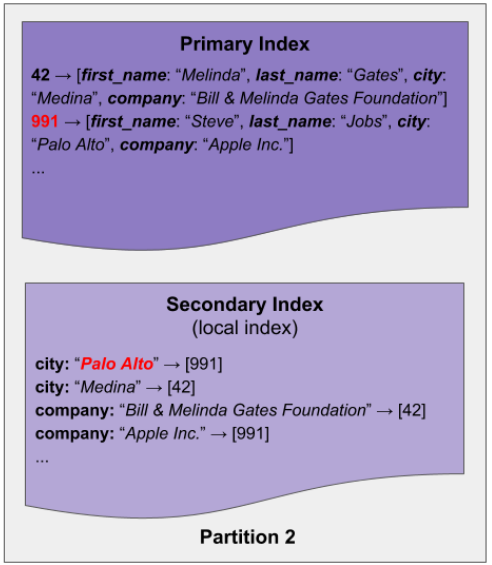
→ `INSERT`/`DELETE`/`UPDATE` performed locally

Weaknesses:

→ maintaining *local index* will compete with *local workload* niand 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

Strenths:

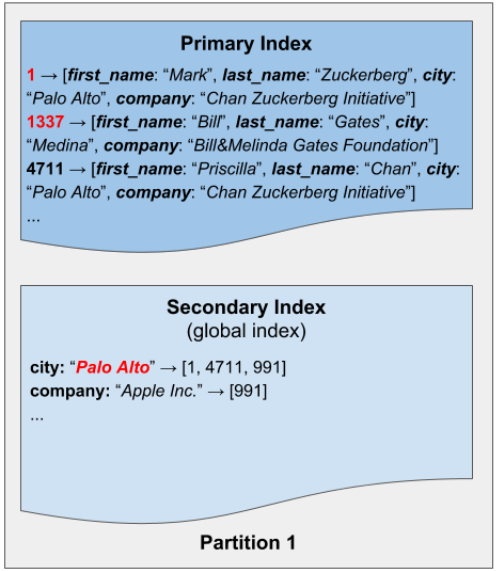
→ `SELECT` statements need to query only one index partition

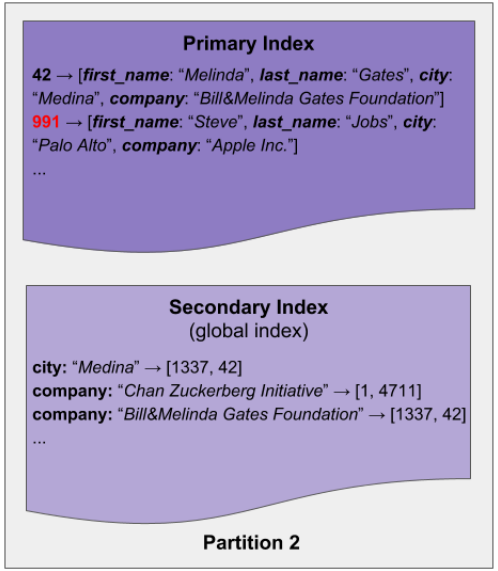
→ No *scattering and gathering (like using LSI)*

Weaknesses:

→ maintaining *global index* requires more overhead than a *local index* → `INSERT/DELETE/UPDATE` statements require remote updates

→ usually weakens read-consistency as indices updates 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 rebalacning.
- No Availability Impact: Rebalancing should have as less impact as possible on a running data-system, requiring no downtime.

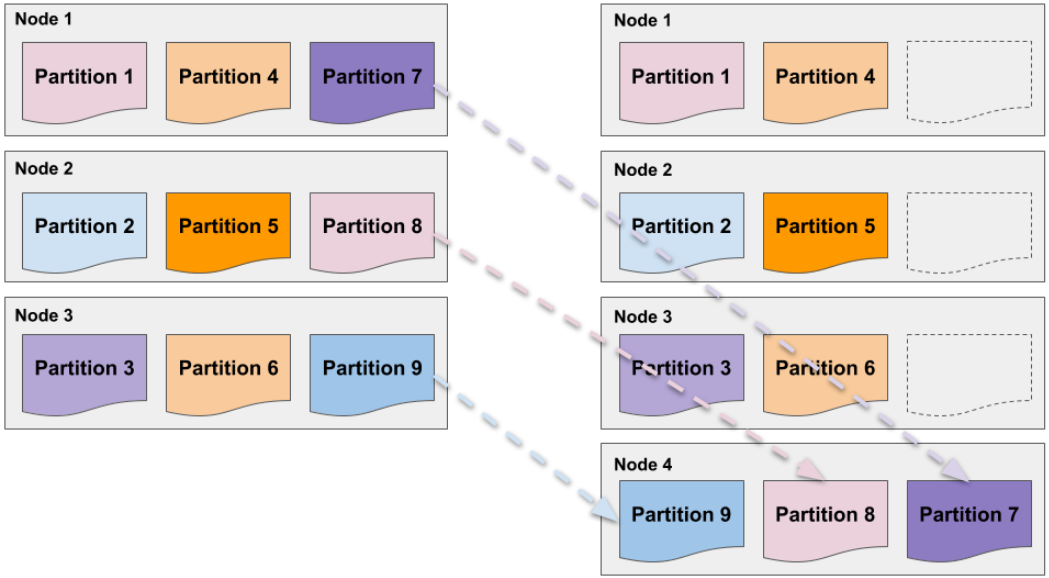
Partitioning – **Rebalancing Partitions (Hash Modulo)**

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:

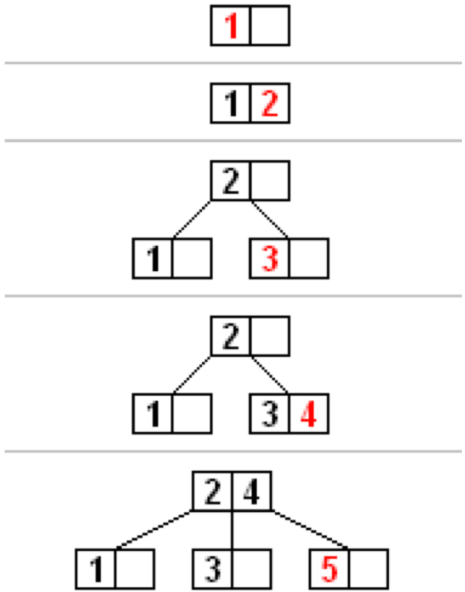
- partition/node mappings change
- but for less partitions

→ only partial rewrite of partitions

Used by e.g.:

- Voldemort
- Riak
- ElasticSearch
- Couchbase

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

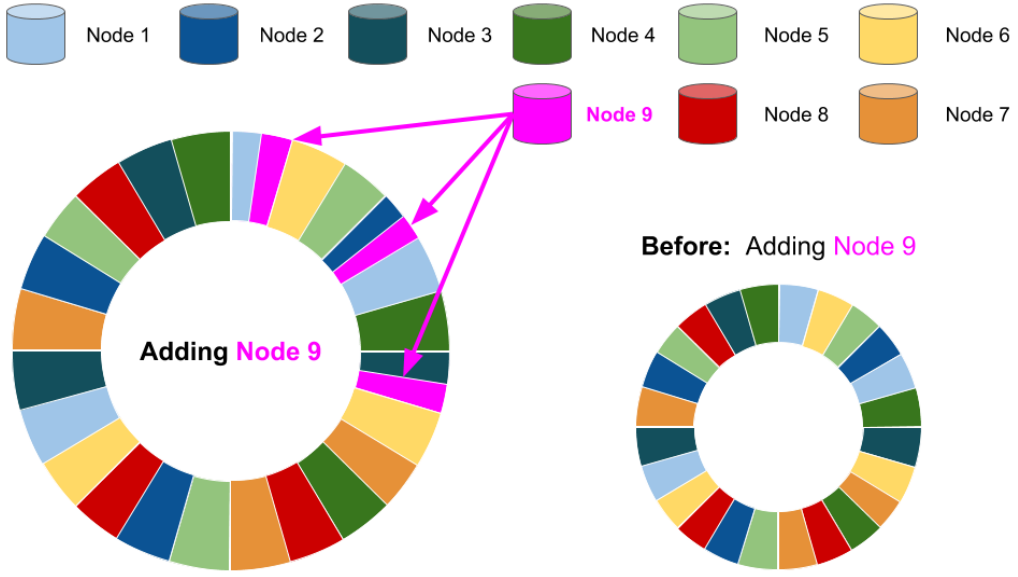


Approach:
- Create certain number of initial partitions
- Idea similar to B-Trees
→ If a partition exceeds a threshold in size, it gets split
→ If a partition falls below a threshold in size, merge it with another
- new nodes „steal“ partitions from old ones

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

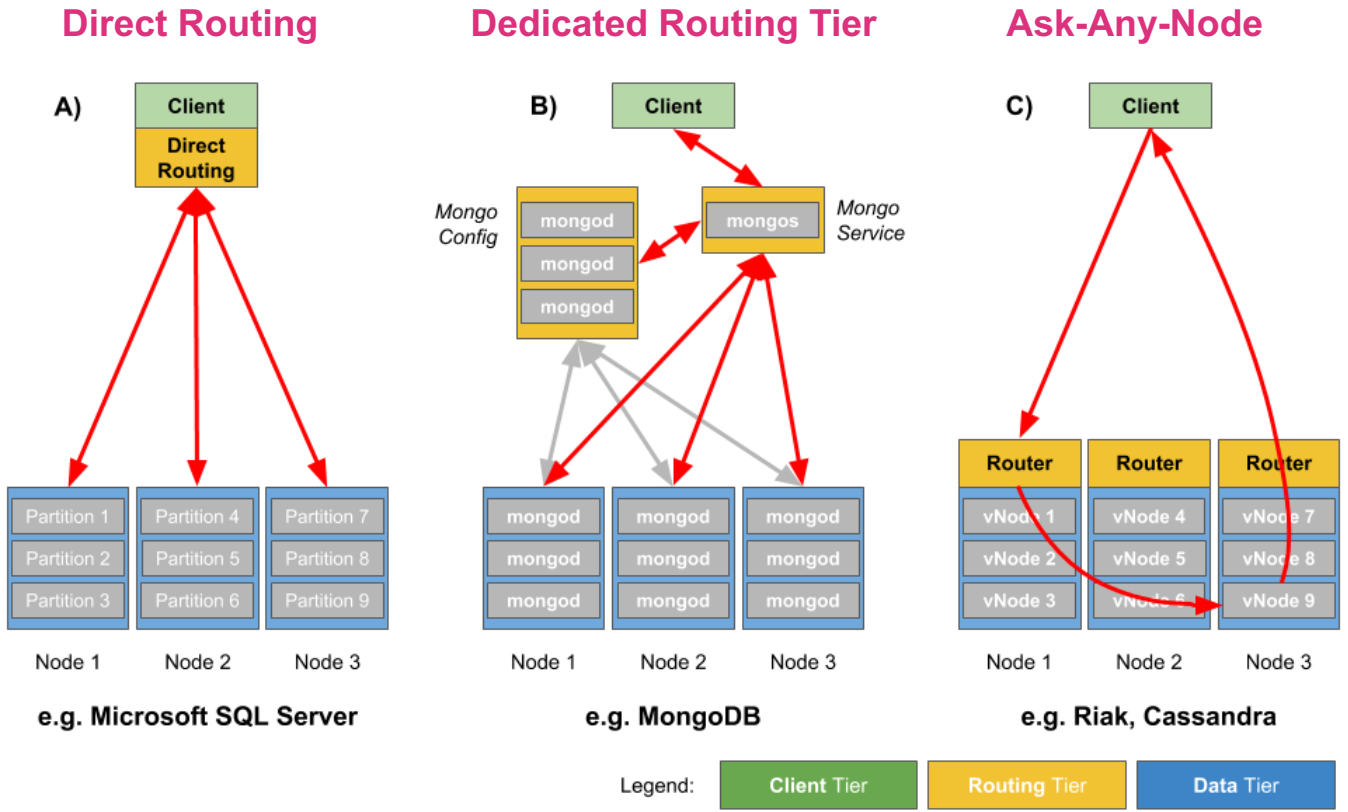
Pro/Contra:

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

Used by e.g.:

- Cassandra

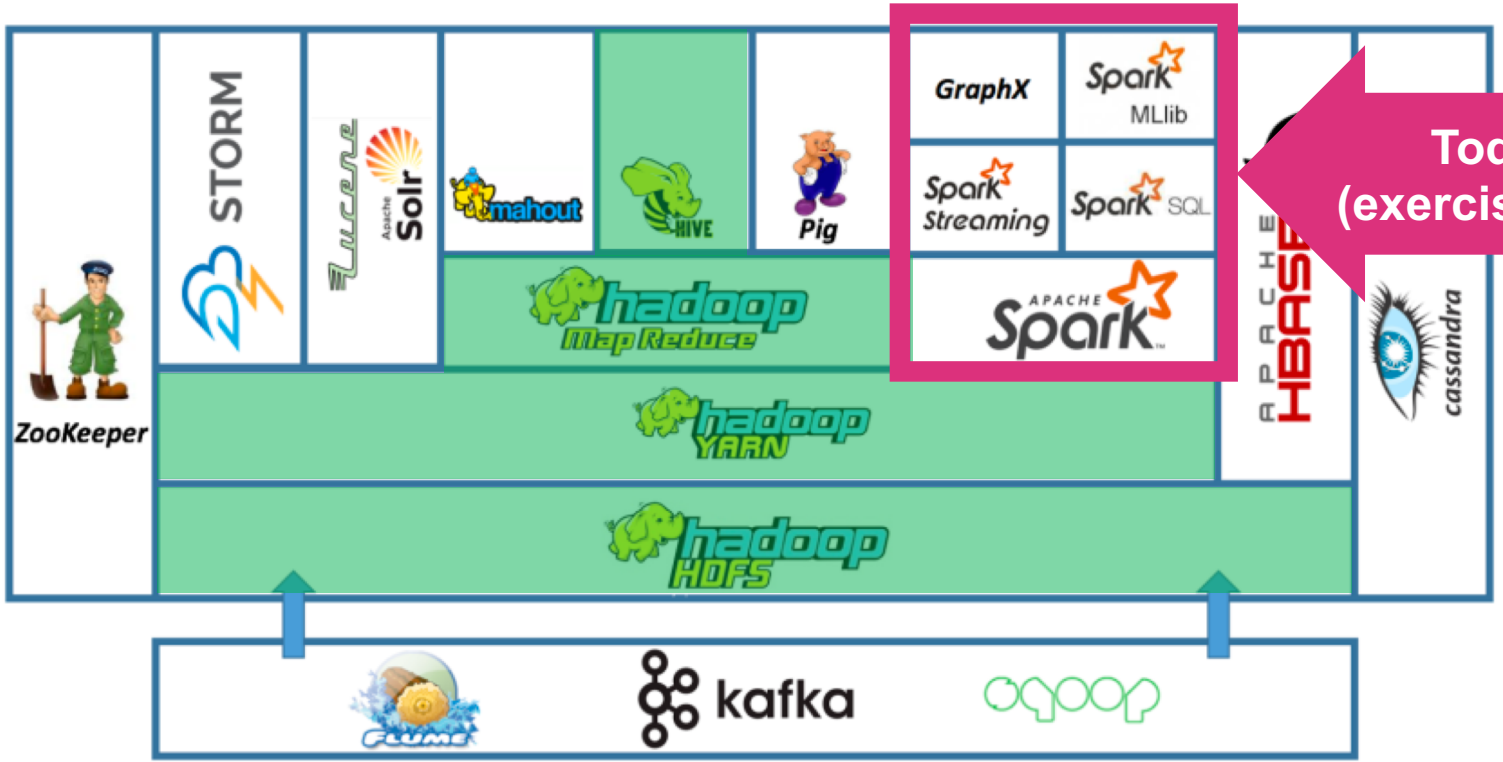
Partitioning – **Partition Lookup**



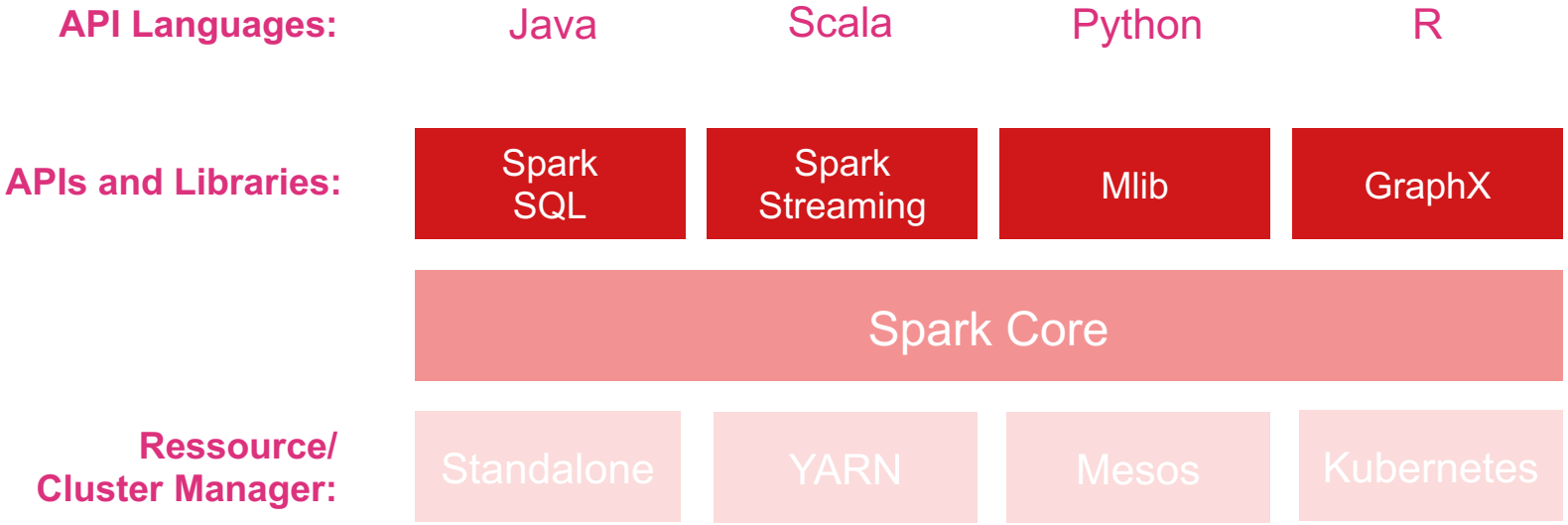
HandsOn – **Spark, Scala, PySpark and Jupyter**

A quick Introduction to Spark, Scala, PySpark and Jupyter

The Hadoop Ecosystem

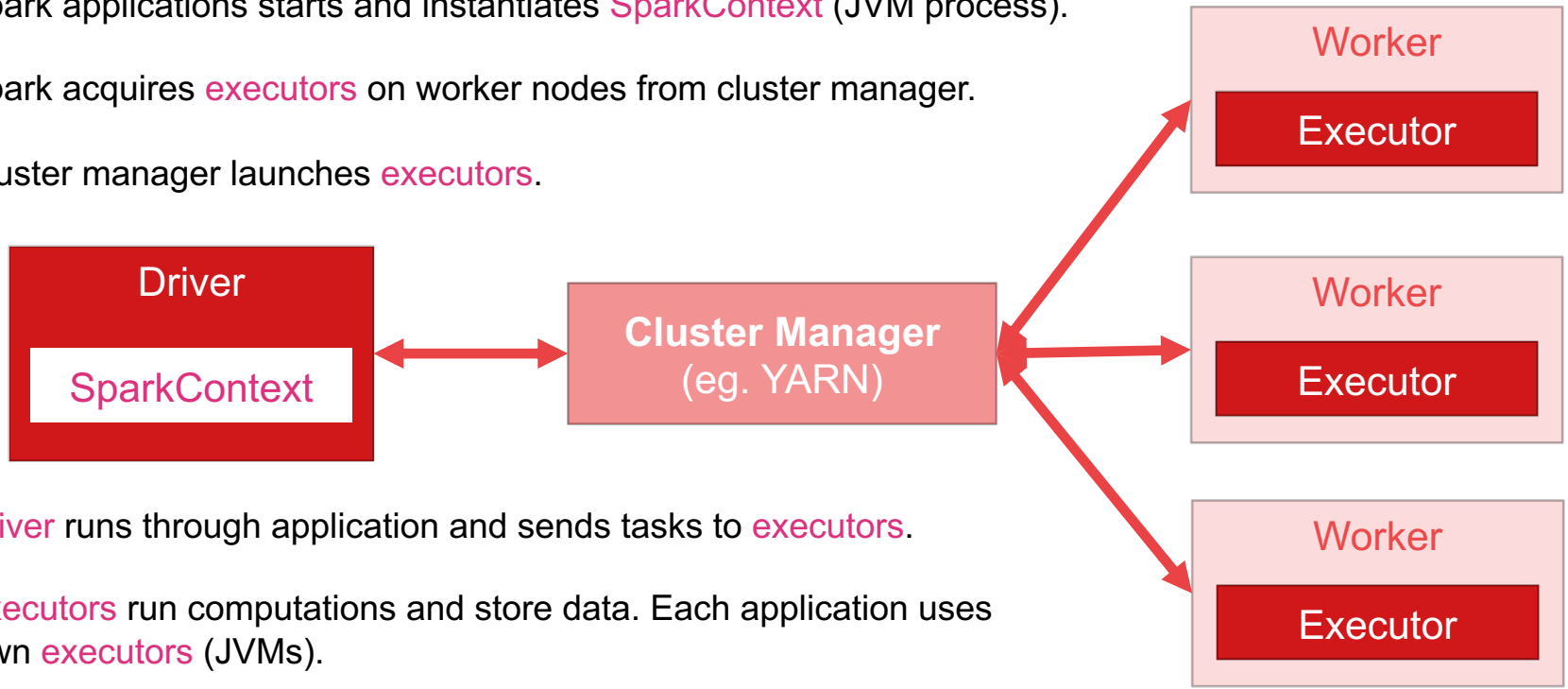


Spark


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, Redis, Riak, Couch-base, 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.



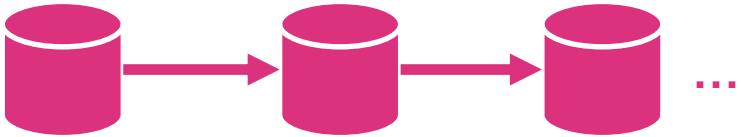
4. Driver runs through application and sends tasks to executors.
5. Executors run computations and store data. Each application uses its own executors (JVMs).
6. If any worker crashes, ist taks will be send to another executor.

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

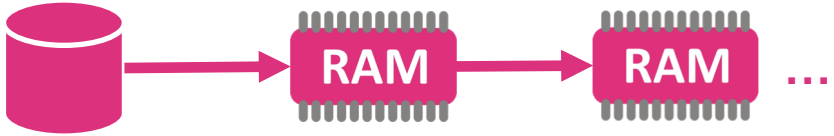
Hadoop MapReduce vs Spark

Hadoop MapReduce



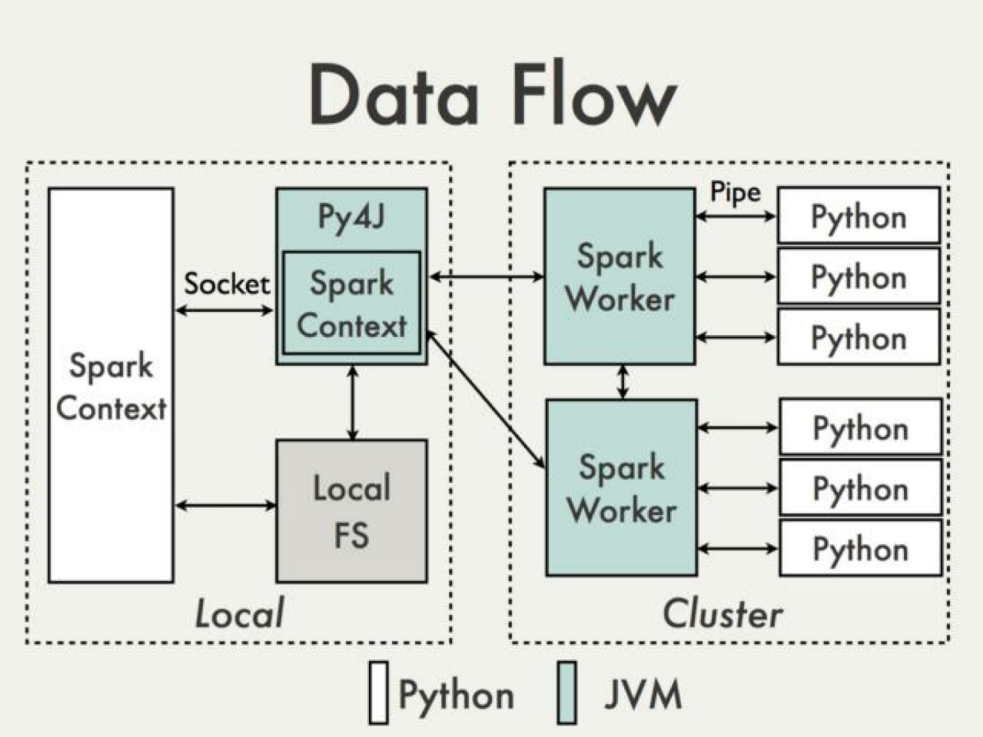
- performs read from HDFS (HDD) before every computation
- performs write on HDFS (HDD) after every computation
- using distributed disk space

Spark



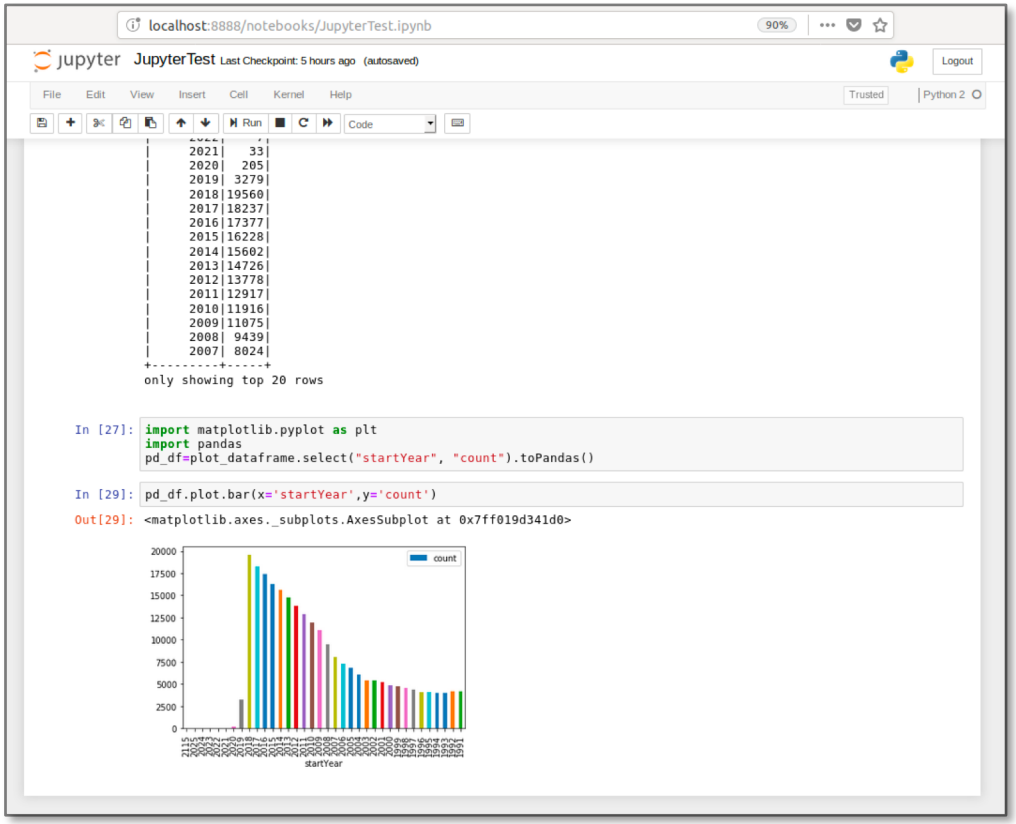
- performs read from RAM before e very computation (except first)
- performs write on RAM after every computation
- using distributed **memory**

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 t he local JVM
- e.g. **sc.textFile()** in Python will call JavaSp arkContext **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

Install and Setup Spark

Download And Install Spark

1. Download Spark 2.3.2 (for Hadoop 2.7++)

```
wget http://mirror.funkfreundelandshut.de/apache/spark/spark-2.3.2/spark-2.3.2-bin-hadoop2.7.tgz
```

2. Extract and Move:

```
tar -xzf http://mirror.funkfreundelandshut.de/apache/spark/spark-2.3.2/spark-2.3.2-bin-hadoop2.7.tgz
```

```
mv spark-2.3.2-bin-hadoop2.7/ spark
```


Configure Spark

1. Setup `spark-env.sh`

```
cp spark/conf/spark-env.sh.template spark/conf/spark-env.sh
```

```
vi spark/conf/spark-env.sh
```

```
export SPARK_MASTER_IP=localhost
export SPARK_WORKER_CORES=1
export SPARK_WORKER_MEMORY=800m
export SPARK_WORKER_INSTANCES=1
export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
SPARK_LOCAL_IP="localhost"
```


Configure Spark

2. Setup `slave`

```
cp spark/conf/slaves.template spark/conf/slaves
```

```
vi spark/conf/slaves
```

```
# A Spark Worker will be started on each of the machines listed below.
localhost
```


Configure Spark

3. Setup Environment Variables

```
vi .bashrc
```

```
export SPARK_HOME=/home/hadoop/spark
export PATH=$SPARK_HOME/bin:$PATH
```

```
source .bashrc
```


Configure Yarn (Yarn Java 8 Bug)

https://issues.apache.org/jira/browse/YARN-4714
https://stackoverflow.com/questions/38988941/running-yarn-with-spark-not-working-with-java-8

1. Setup `yarn-site.xml`

```
vi hadoop/etc/hadoop/yarn-site.xml
```

```
<property>
    <name>yarn.nodemanager.pmem-check-enabled</name>
    <value>false</value>
</property>
<property>
    <name>yarn.nodemanager.vmem-check-enabled</name>
    <value>false</value>
</property>
```


Configure Yarn

2. Restart HDFS and Yarn

```
stop-all.sh
```

```
start-dfs.sh
```

```
start-yarn.sh
```


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_1539278841328_0001).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.3.2
      /_/

Using Scala version 2.11.8 (OpenJDK 64-Bit Server VM, Java 1.8.0_181)
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```


Start Spark (on Yarn) – Test Install

2. Create Scala Collection `dummy_data` of Numbers 1 – 100:

```
scala> val dummy_data = 1 to 100
dummy_data: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3, 4, 5, 6
, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26,
27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46,
47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66,
67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86,
87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100)
```

3. Create a Scala RDD `dummy_RDD` of previous collection:

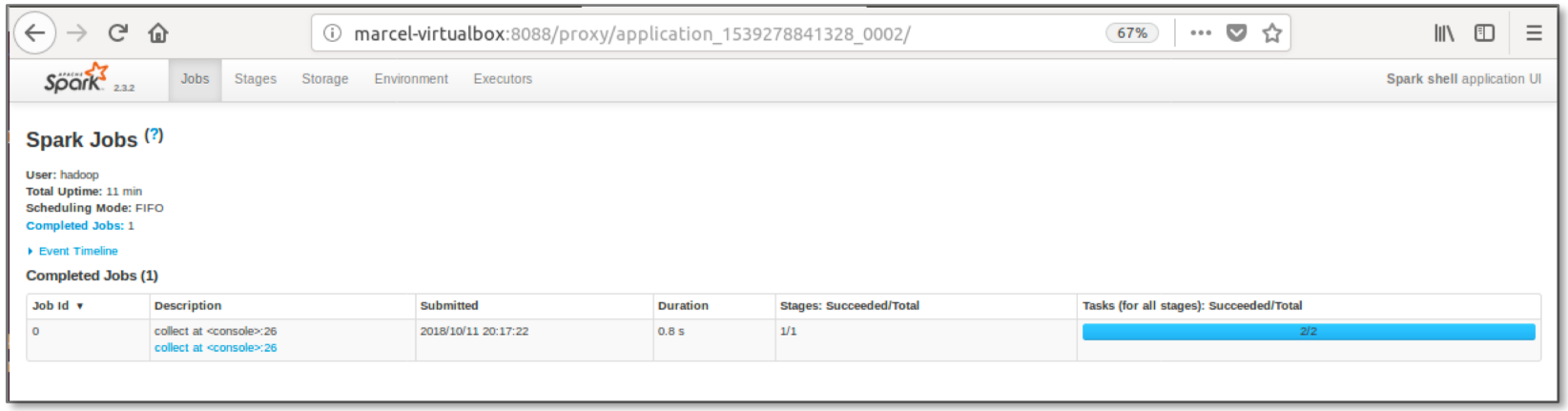
```
scala> val dummy_RDD = sc.parallelize(dummy_data)
dummy_RDD: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallel
ize at <console>:26
```

Start Spark (on Yarn) – Test Install

4. Run simple filter transformation and `collect()` to calculate results:

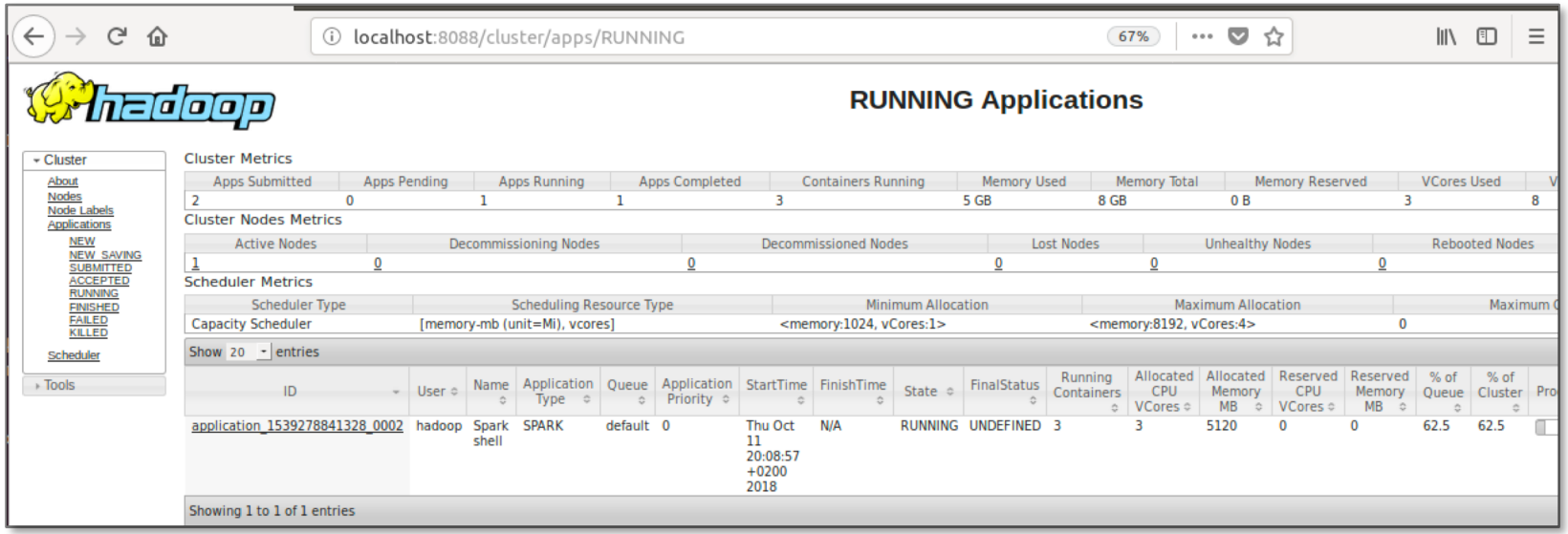
```
scala> dummy_RDD.filter(_ < 10).collect()
res0: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

5. Now you should see a completed stage in Spark Shell UI (*localhost:4040*):



Start Spark (on Yarn) – Test Install

6. As well as the whole time the Spark Shell Container Running on Yarn:



Start Spark (Standalone) – Test Install

1. Start Spark Shell:

```
spark-shell
```

```
Spark context Web UI available at http://localhost:4040
Spark context available as 'sc' (master = local[*], app id = local-1539282394656).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.3.2
      /_/

Using Scala version 2.11.8 (OpenJDK 64-Bit Server VM, Java 1.8.0_181)
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```


Start Spark (Standalone) – Test Install

2. Run previous example:

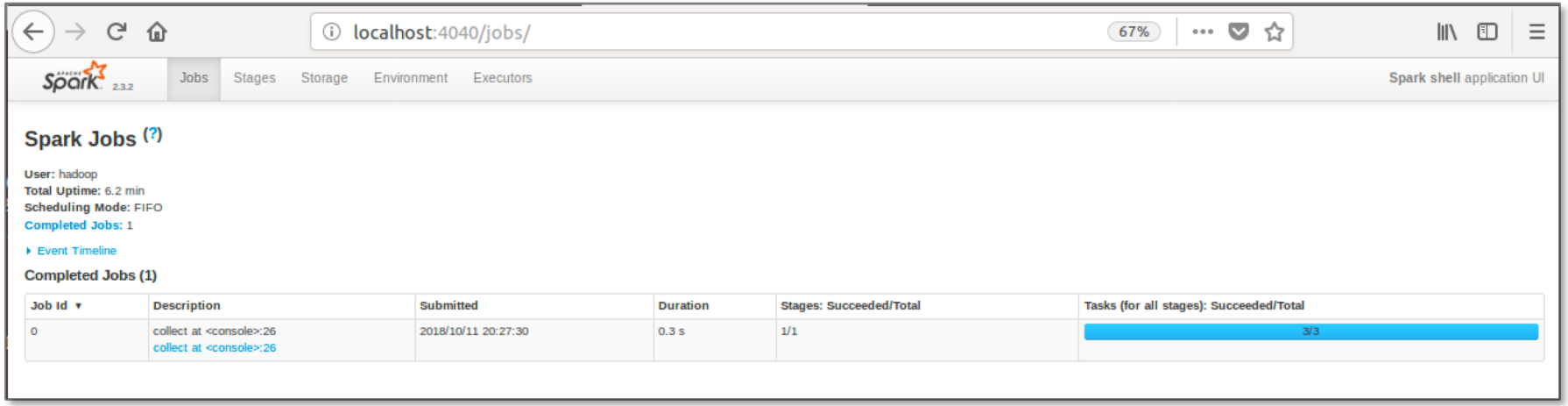
```
scala> val dummy_data = 1 to 100
dummy_data: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1
1, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34,
35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58,
59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82,
83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100)
```

```
scala> val dummy_RDD = sc.parallelize(dummy_data)
dummy_RDD: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>
:26
```

```
scala> dummy_RDD.filter(_ < 10).collect()
res0: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

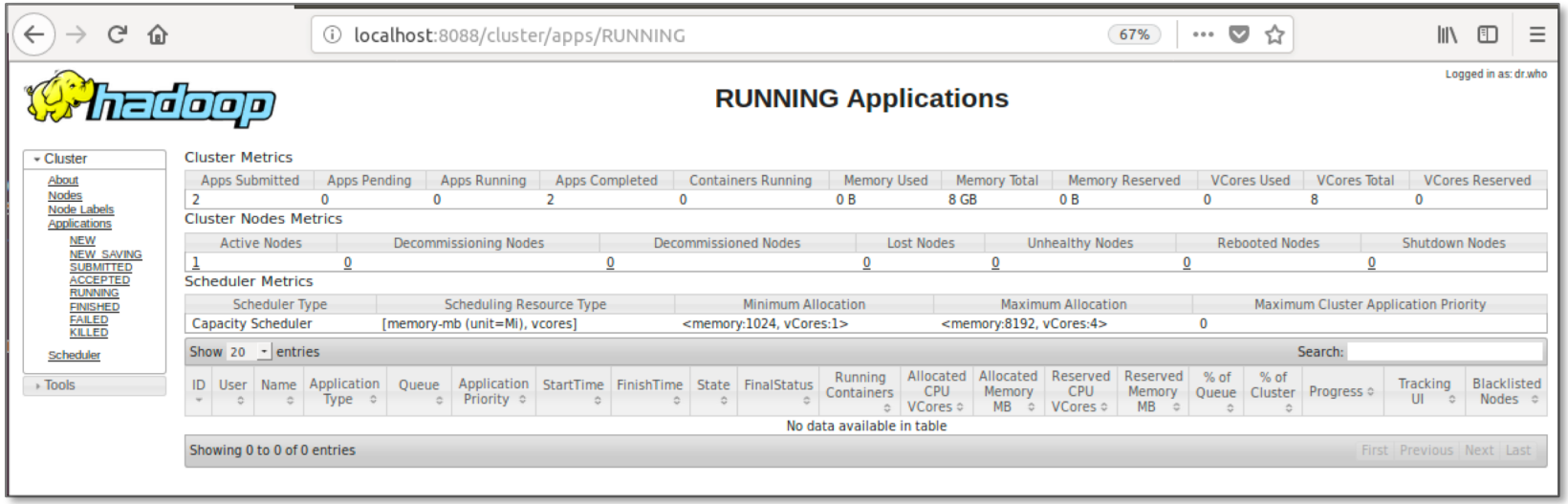

Start Spark (Standalone) – Test Install

3. Now you should see a completed stage in Spark Shell UI (*localhost:4040*):



Start Spark (Standalone) – Test Install

4. And see, nothing Running On Yarn:



Exercises Preparation II

Install and Setup PySpark

Start and Use PySpark (on Yarn) – Test Install

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

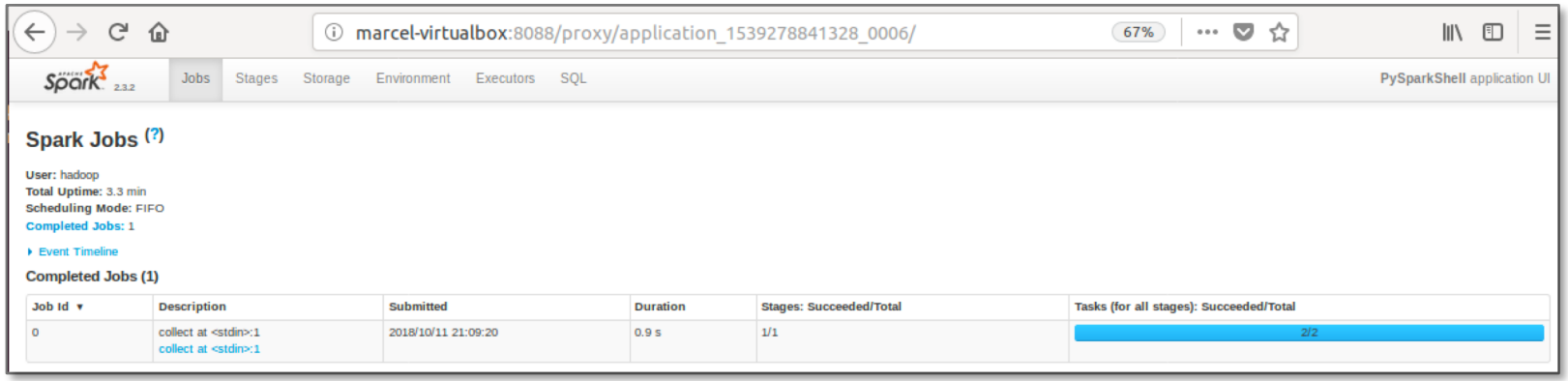
```
pyspark --master yarn
```

```
Python 2.7.15rc1 (default, Apr 15 2018, 21:51:34)
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 2.3.2
      /_/

Using Python version 2.7.15rc1 (default, Apr 15 2018 21:51:34)
SparkSession available as 'spark'.
>>> dummy_data = range(1, 100)
>>> dummy_rdd = sc.parallelize(dummy_data)
>>> print(dummy_rdd.filter(lambda x: x<10).collect())
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
```

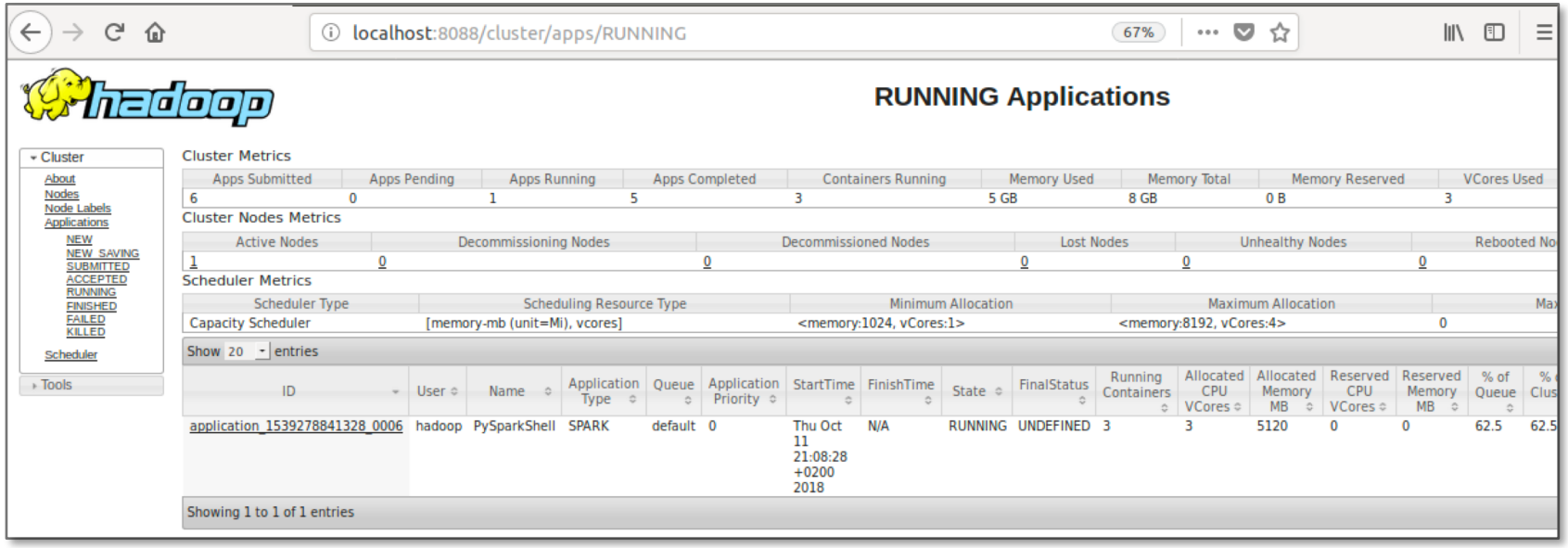
Start and Use PySpark (on Yarn) – Test Install

2. Now you should see a completed stage in **PySpark** UI (*localhost:4040*):



Start and Use PySpark (on Yarn) – Test Install

3. As well as the related **PySpark** Container Running on Yarn:



Exercises Preparation III

Install and Setup Jupyter (on PySpark)

Install and Setup Jupyter

1. Install Python

```
sudo apt-get install python
```

2. Install `pip` for Python

```
sudo apt-get install python-pip
```

3. Install Jupyter

```
pip2 install jupyter
```


Install and Setup Jupyter

4. As Jupyter is installed under `~/.local/bin/jupyter` add path to **$PATH**:

```
vi .bashrc
```

```
export PATH=$PATH:~/.local/bin
```

```
source .bashrc
```

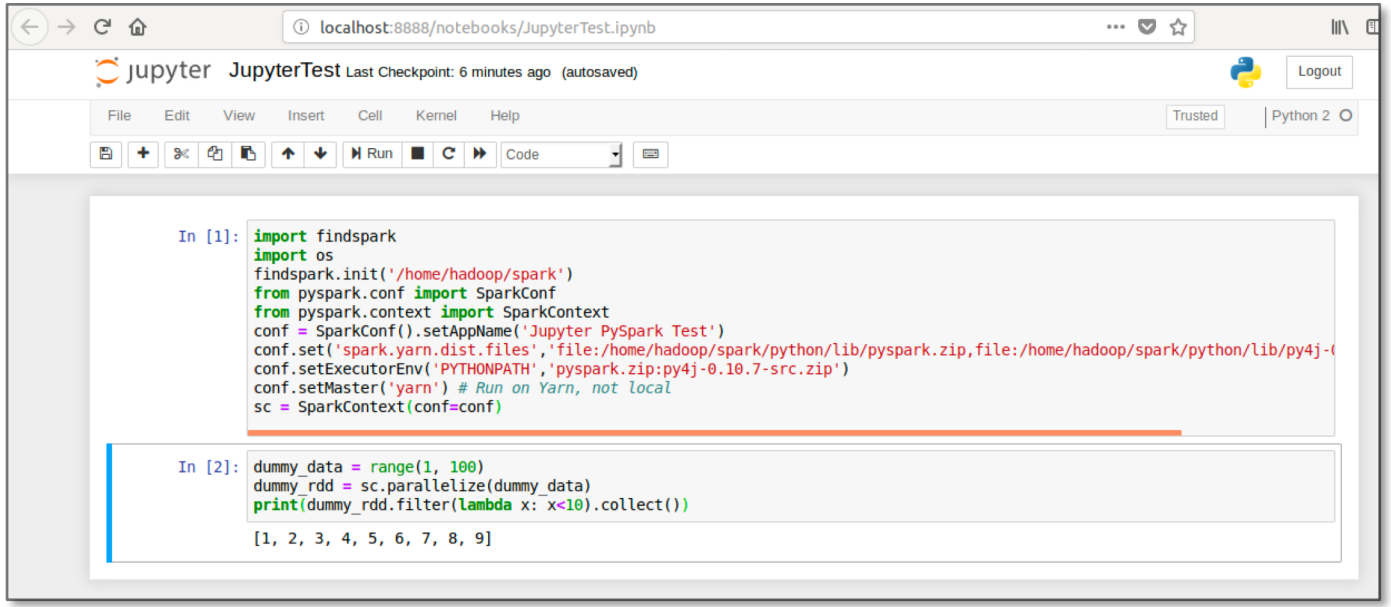
5. Start Jupyter Notebook

```
jupyter notebook
```


Start and Use Jupyter (on PySpark on Yarn)

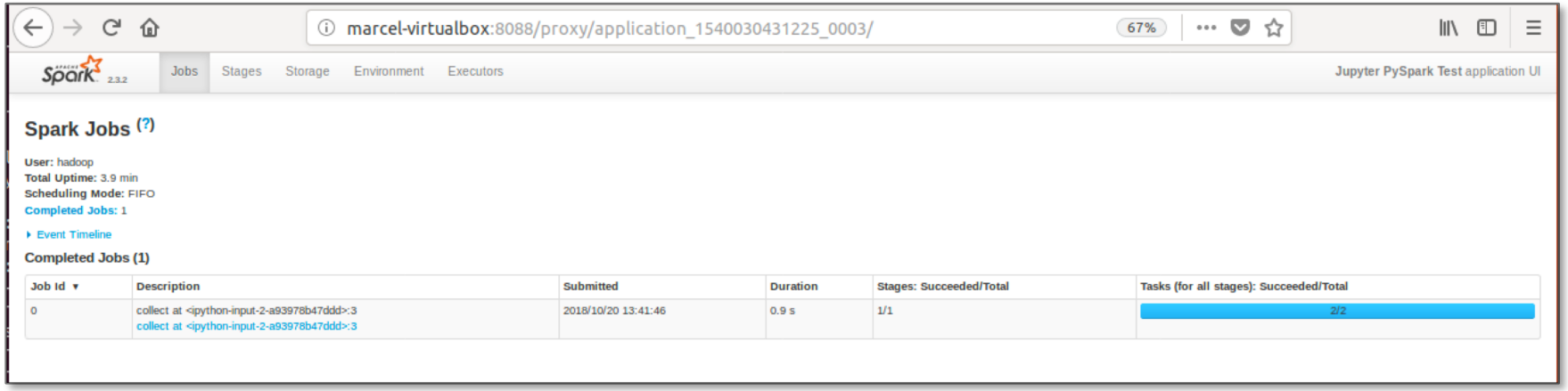
6. Open Jupyter Notebook and execute previous example as Python code:

```
jupyter notebook
```



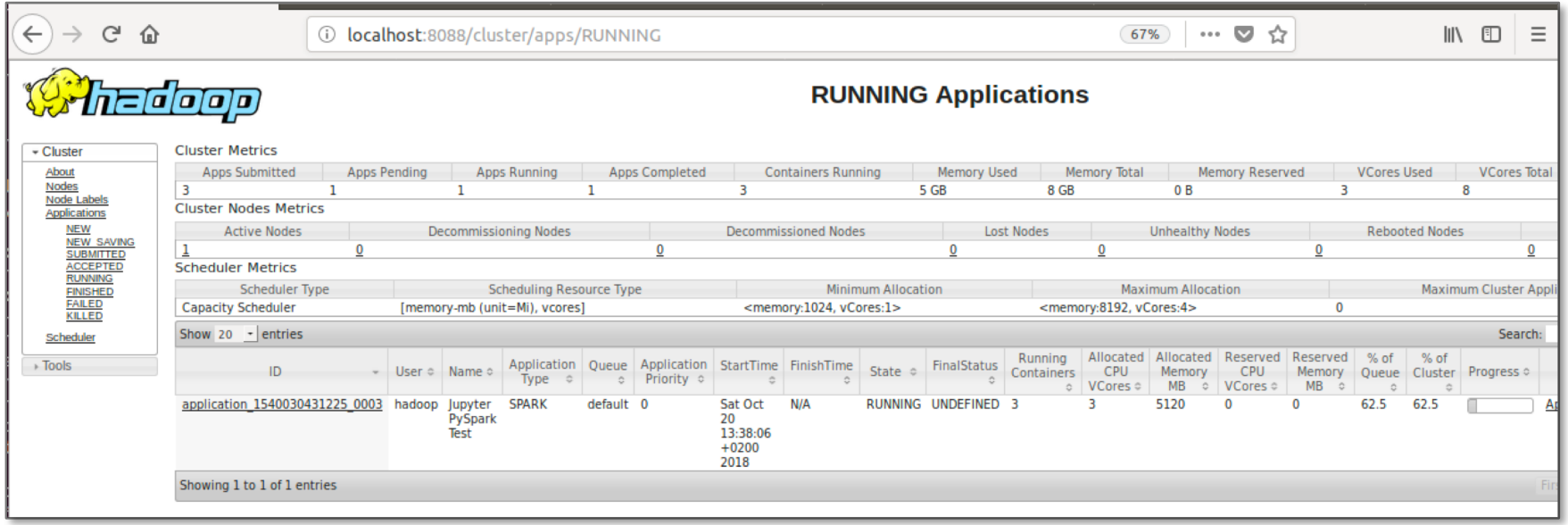
Use Jupyter (on PySpark on Yarn)

7. Now you should see a completed stage in **PySpark** UI (*localhost:4040*):



Use Jupyter (on PySpark on Yarn)

8. As well as the related **PySpark** Container Running on Yarn:



Use Jupyter (on PySpark on Yarn)

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

```
In [3]: from pyspark.sql import SparkSession
        spark = SparkSession(sc)

In [4]: imdb_ratings_dataframe = spark.read.format('com.databricks.spark.csv').options(delimiter = '\t',header ='true',nullValue
        ='null',inferSchema='true').load('hdfs:///user/hadoop/imdb_raw/title_ratings/2018/12/7/title.ratings.tsv')

In [5]: imdb_ratings_dataframe.show(5) # show first 5 lines of tsv file (now a spark dataframe)

        +---------+-------------+--------+
        |   tconst|averageRating|numVotes|
        +---------+-------------+--------+
        |tt0000001|          5.8|    1418|
        |tt0000002|          6.4|     167|
        |tt0000003|          6.6|    1013|
        |tt0000004|          6.4|     100|
        |tt0000005|          6.2|    1712|
        +---------+-------------+--------+
        only showing top 5 rows


In [6]: imdb_ratings_dataframe.printSchema() # show schema of tsv file (now dataframe)

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

https://github.com/marcelmittelstaedt/BigData/tree/master/exercises/04_spark_pyspark_jupyter/JupyterTest.html

Use Jupyter (on PySpark on Yarn)

9. Basic PySpark: Operations on DataFrames:

```
In [8]: imdb_ratings_dataframe.count() # show number of rows within dataframe

Out[8]: 872954

In [9]: from pyspark.sql.functions import col, avg
        imdb_ratings_dataframe.agg(avg(col("averageRating"))).show() # calculate average movie rating

        +------------------+
        |avg(averageRating)|
        +------------------+
        | 6.929355269579343|
        +------------------+

In [11]: imdb_ratings_dataframe.filter(col('averageRating') > 9.5).show(5) # filter movie ratings > 9.5 and show first 5

         +---------+-------------+--------+
         |   tconst|averageRating|numVotes|
         +---------+-------------+--------+
         |tt0015927|          9.7|       6|
         |tt0041069|          9.7|       6|
         |tt0050536|          9.7|       7|
         |tt0053560|          9.6|       5|
         |tt0055416|          9.7|      31|
         +---------+-------------+--------+
         only showing top 5 rows

In [16]: imdb_ratings_dataframe.write.format("csv").save("hdfs:///user/hadoop/imdb_ratings")
         # saved on HDFS as /user/hadoop/imdb_ratings/part-00000-ce6e8310-64c0-4fb9-b597-d66a92f5309b-c000.csv
```

https://github.com/marcelmittelstaedt/BigData/tree/master/exercises/04_spark_pyspark_jupyter/JupyterTest.html

Use Jupyter (on PySpark on Yarn)

9. Basic PySpark: Join DataFrames:

```
In [16]: imdb_ratings_dataframe.write.format("csv").save("hdfs:///user/hadoop/imdb_ratings")
         # saved on HDFS as /user/hadoop/imdb_ratings/part-00000-ce6e8310-64c0-4fb9-b597-d66a92f5309b-c000.csv

In [17]: imdb_movies_dataframe = spark.read.format('com.databricks.spark.csv').options(delimiter = '\t',header ='true',nullValue =
         'null',inferSchema='true').load('hdfs:///user/hadoop/imdb_raw/title_basics/2018/12/7/title.basics.tsv')

In [18]: all_imdb_dataframe = imdb_ratings_dataframe.join(imdb_movies_dataframe, imdb_ratings_dataframe.tconst == imdb_movies_data
         frame.tconst)

In [19]: all_imdb_dataframe.select("primaryTitle", "startYear", "averageRating", "numVotes").show(5)
+--------------------+---------+-------------+--------+
|        primaryTitle|startYear|averageRating|numVotes|
+--------------------+---------+-------------+--------+
|The Puppet's Nigh...|     1908|          6.5|     126|
|The Lighthouse Ke...|     1911|          7.1|       8|
|    The Sands of Dee|     1912|          6.9|      51|
|Zigomar contre Ni...|     1912|          6.8|      10|
|His Favorite Pastime|     1914|          5.1|     814|
+--------------------+---------+-------------+--------+
only showing top 5 rows
```

https://github.com/marcelmittelstaedt/BigData/tree/master/exercises/04_spark_pyspark_jupyter/JupyterTest.html

Use Jupyter (on PySpark on Yarn)

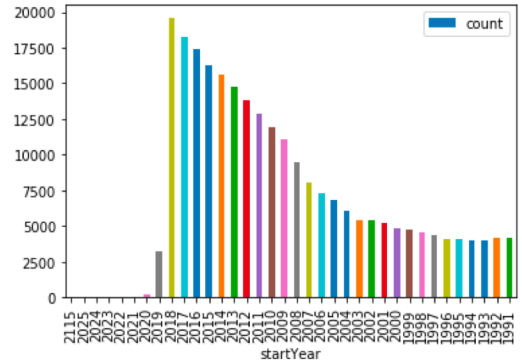
9. Basic Python: Plot results:

```
In [25]: plot_dataframe = imdb_movies_dataframe.filter(col('startYear') != '\N').filter(col('startYear') > 1990).filter(col('titleType') == 'movie').groupBy('startYear').count().sort(col("startYear").desc())

In [27]: import matplotlib.pyplot as plt
         import pandas
         pd_df=plot_dataframe.select("startYear", "count").toPandas()

In [29]: pd_df.plot.bar(x='startYear',y='count')

Out[29]: <matplotlib.axes._subplots.AxesSubplot at 0x7ff019d341d0>
```



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 **movies** are within the IMDB dataset?

    b) Who is the **oldest** 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 2000
    - have an average rating better than 8
    - have been voted more than 100.000 times

    save 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)