Introduction to Databricks

Martin Oharek, Alisa Benešová

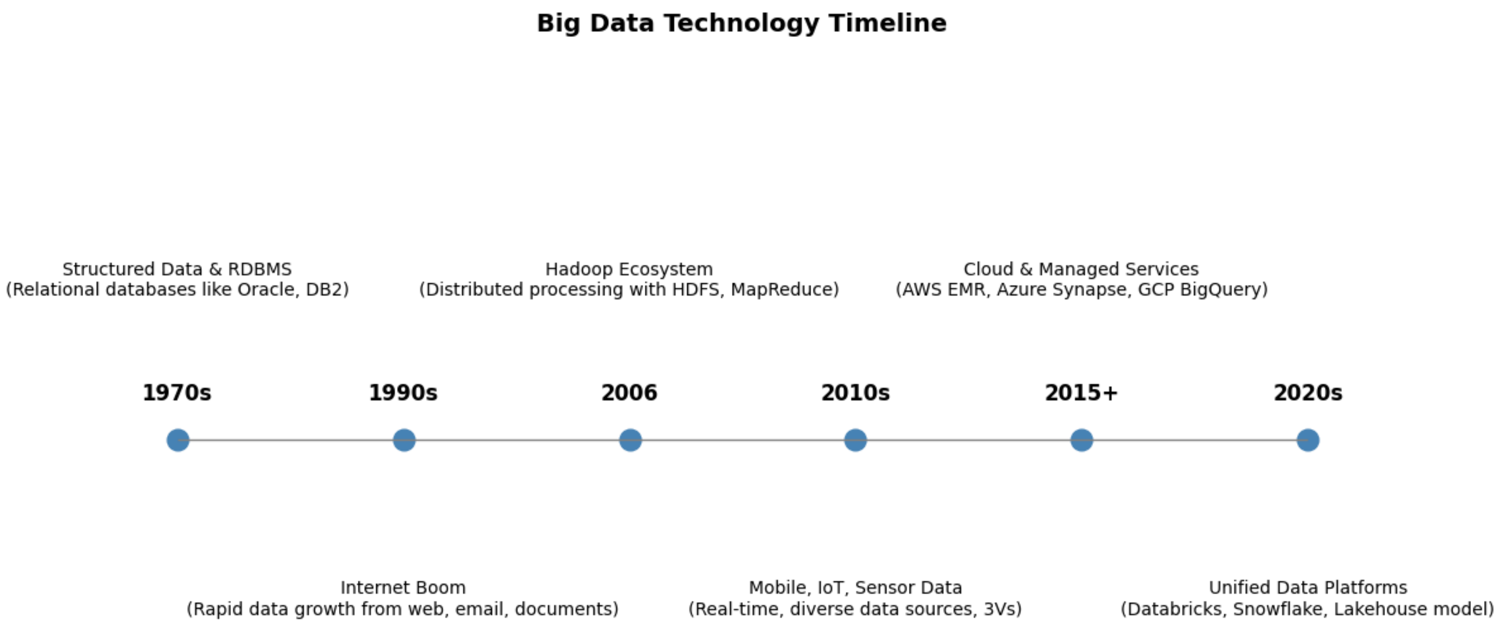
October 8th, 2025

Outline

Outline

1. **History context**
   - Challenges of traditional/legacy solutions
   - What managed platforms (e.g. Databricks) bring to the table and why the companies want it
2. **Databricks intro**
   - Lakehouse concept
   - Databricks architecture
   - Building blocks (cloud integration, Apache Spark, MLFlow, Delta Lake,…)
   - Use cases
3. **Core Databricks features**
   - Compute x Storage, Delta Lake
   - Clusters
   - Databricks workflows
   - Data object types
4. **Real world Databricks use cases**

History context


Big Data Technology Timeline
Structured Data & RDBMS
(Relational databases like Oracle, DB2)
Hadoop Ecosystem
(Distributed processing with HDFS, MapReduce)
Cloud & Managed Services
(AWS EMR, Azure Synapse, GCP BigQuery)
1970s
1990s
2006
2010s
2015+
2020s
Internet Boom
(Rapid data growth from web, email, documents)
Mobile, IoT, Sensor Data
(Real-time, diverse data sources, 3Vs)
Unified Data Platforms
(Databricks, Snowflake, Lakehouse model)

Brief History of Big Data Technologies pt1

PROFINIT

- **1. Structured Data & RDBMS (~1970s)**
  - Traditional relational databases (e.g., Oracle, PostgreSQL) designed for structured, tabular data.
  - Effective for business applications but limited in scalability and flexibility.
- **2. Data Explosion (~1990s–present) + Internet era (~1991):**
  - Rapid growth of user-generated data.
  - **Mobile, IoT, sensors (~2010):** real-time, high-volume data streams.
  - Rise of semi-structured (JSON, XML) and unstructured data (logs, images, video).
- **3. Limits of Traditional Tools**
  - Classic databases couldn't handle the **volume, variety, and velocity** of modern data.
  - Need for distributed processing and scalable storage.
- **4. Hadoop Ecosystem (2006)**
  - Open-source framework for **distributed computing and storage**.
  - Core components: HDFS, MapReduce; later extended by Hive, Pig, HBase, Spark.
  - Enabled cost-effective big data processing on commodity hardware.

Brief History of Big Data Technologies pt2

PROFINIT

- **5. Data Lakes**
  - Centralized storage for structured, semi-structured, and unstructured data.
  - Schema-on-read approach: flexible, scalable, cost-efficient.
- **6. Cloud & Managed Services**
  - Platforms like **AWS (EMR, Kinesis)**, **Azure (Data Factory, Synapse)** enable scalable, on-demand data processing.
  - No infrastructure maintenance, pay-as-you-go model.
- **7. Modern Data Platforms**
  - Unified, cloud-native environments like **Databricks**, **Snowflake**, **BigQuery**.
  - Support batch & real-time processing, machine learning, collaborative workflows.
  - Emphasis on **scalability, simplicity, and integration** across the data lifecycle.
  - Popular Serveless solutions

Legacy technology challenges

PROFINIT

Infrastructure Challenges

- On-premises infrastructure requires full management by the company → Hardware, networking, data centers, monitoring, etc.
- Limited scalability → Scaling up is slow, costly, and often requires upfront investment.
- Over-provisioning is common
  → To handle peak loads, companies must provision more resources than needed most of the time → wasteful and expensive.
- Responsibility for security, availability, reliability, and backups lies entirely with the organization
  → Requires large teams and significant effort.
- Manual updates and patching → Software stack must be constantly maintained, updated, and secured.
- 🔑 Key takeaway:

Many tasks consume time and resources but do not deliver direct business value — they're necessary overhead.

Legacy technology challenges

PROFINIT

Operational Challenges

- Fragmented technology stack
  → Data ingestion, ETL, analytics, dashboarding, and ML often rely on different, disconnected tools.
- Data silos
  → Teams store and manage their own data → limited access across the organization, duplication, and inconsistent “truths.”
- High architectural complexity
  → Increases the cost of development, maintenance, and onboarding.
- Performance issues
  → Non-optimized, fragmented systems can lead to slow queries, failed pipelines, and frustrated users.


I have one problem
I have two problems
I have a lot of problems

Databricks intro

Databricks

- Founded in 2013
- Unified, data analytics platform for building, deploying, sharing, and maintaining enterprise-grade data, analytics, and AI solutions at scale
- Integrated with cloud vendors – AWS, Azure, GCP
- Cloud agnostic
- Databricks Lakehouse platform
- ~ 15% of market share in big-data-analytics domain (https://6sense.com/tech/big-data-analytics/databricks-market-share)
- Databricks account -> Databricks workspaces associated with the account

Figure 1: Magic Quadrant for Data Science and Machine Learning Platforms

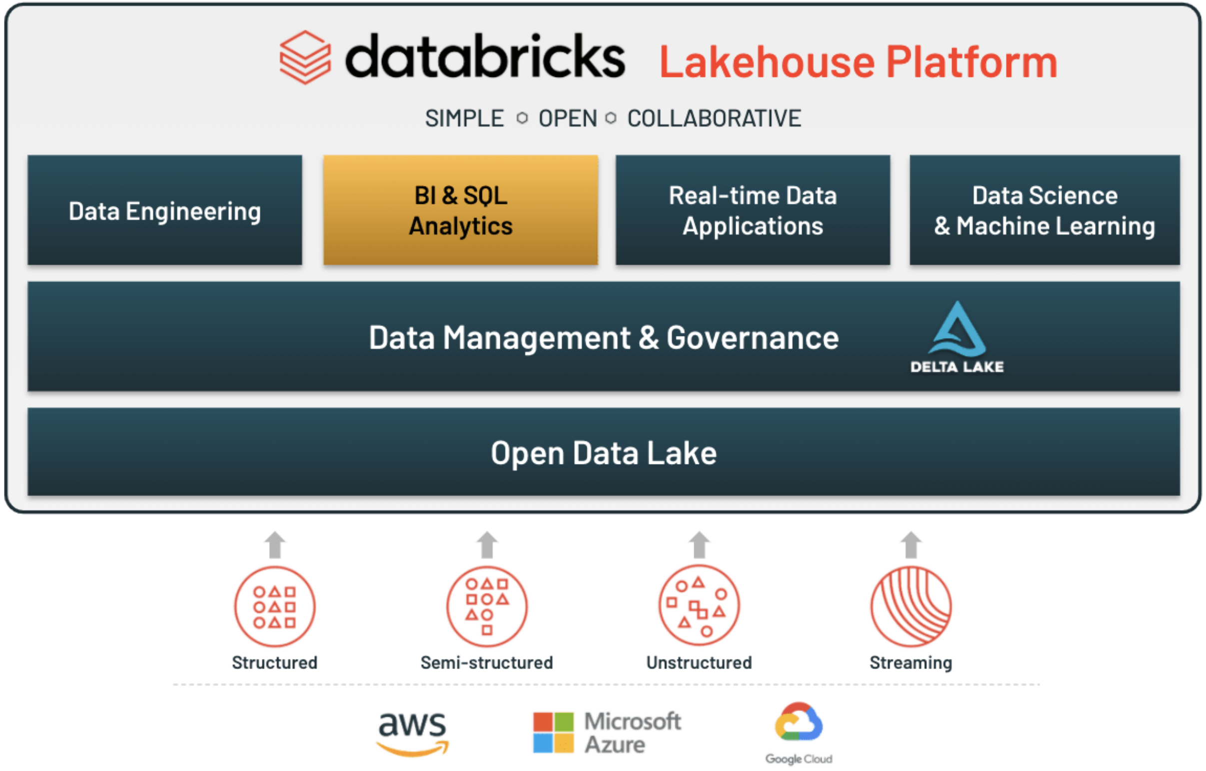
How do Databricks solve the challenges?

PROFINIT

- Cloud-based (AWS, Azure, GCP)
  - Infrastructure is managed by the cloud vendor, you just need to provision it.
- Auto-scaling support (alleviate the over-provisioning issue)
- Provide tools for handling all data-related processing demands (batch, streaming, ML, data sharing,…), all unified under single platform
- Software versions, libraries and runtimes are managed by Databricks, also come with handy libraries preinstalled
- On-demand cluster provisioning -> no need to run machines when idle
- Lakehouse concept + centralized data governance solution – supports the „single source of truth“

Databricks

PROFINIT



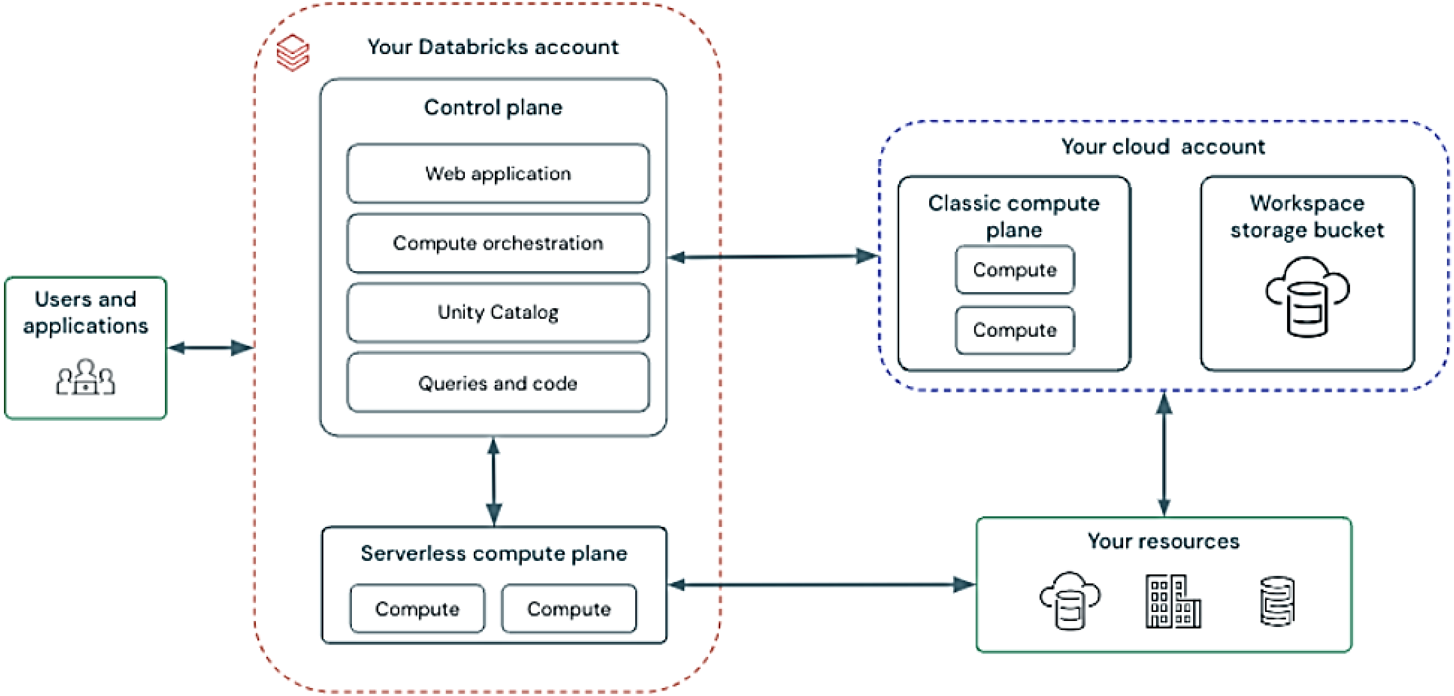
Databricks spaces

PROFINIT

- **Databricks SQL**
  - Compute resources for SQL queries, visualizations and dashboards executed against data sources in the lakehouse
  - SQL warehouse, optimized for processing large-scale data, multi-tenancy
  - Alerting
- **Data Science & Engineering**
  - Notebooks, Apache Spark, Spark Structured Streaming
  - Databricks Jobs
  - ETL – Delta Live Tables
- **Machine learning**
  - AutoML, MLFlow
  - Scalable machine learning - Spark MLLib, HyperOpt, EDA with Spark

Databricks architecture

PROFINIT

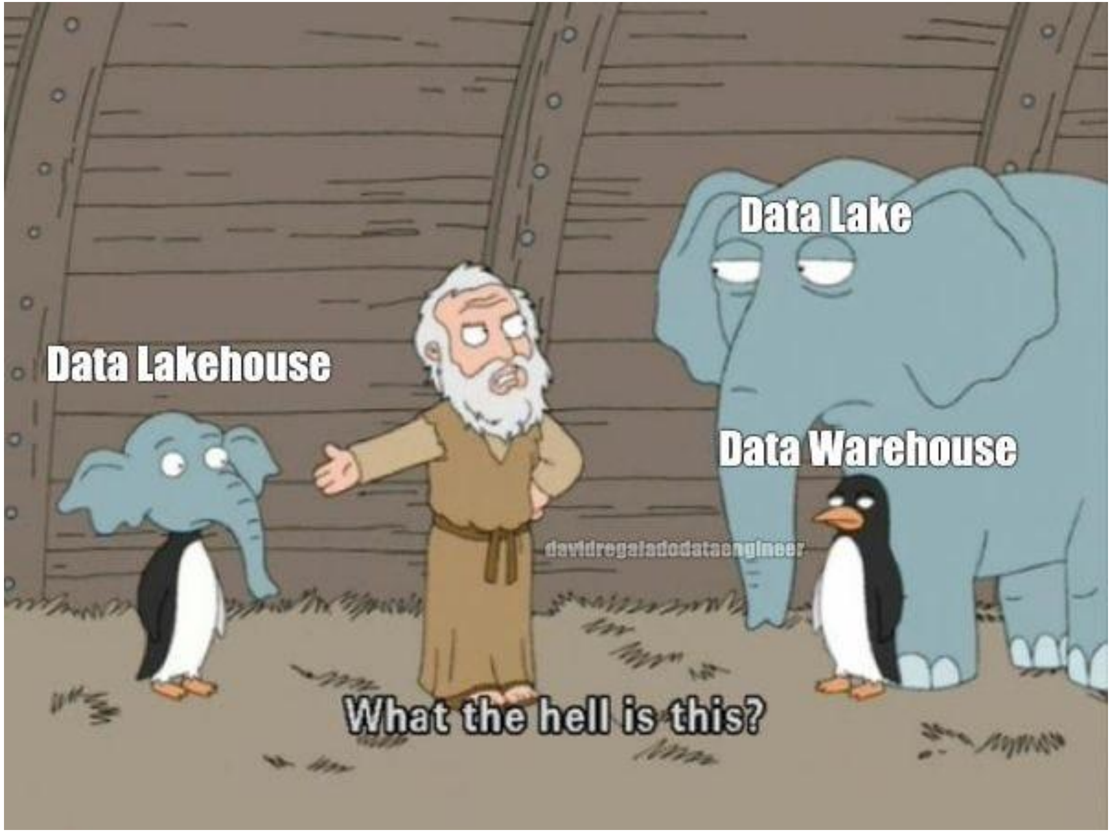


Databricks architecture

PROFINIT

- Control Plane (managed by Databricks)
  - Runs in Databricks' own cloud account (not in customer’s).
  - Manages:Workspace configurationUser permissions & access controlJob scheduling, notebooks, REST API, UI
  - No data processing or storage happens here.
- Classic Data Plane (customer-managed)
  - Runs in the customer's own cloud account (e.g. AWS, Azure).
  - Data is processed and stored within customer's VPC.
  - Used for:Notebooks and JobsClassic / Pro SQL Warehouses
  - Full control, better for regulated environments.
- Serverless Data Plane (Databricks-managed)
  - Compute runs in a shared, managed environment provided by Databricks.
  - Databricks handles provisioning, scaling, and optimization.
  - Used for:Serverless SQL WarehousesModel Serving
  - Fast startup, lower ops overhead.
  - ! Data processed outside customer’s VPC — consider data sensitivity.

Data Lakehouse



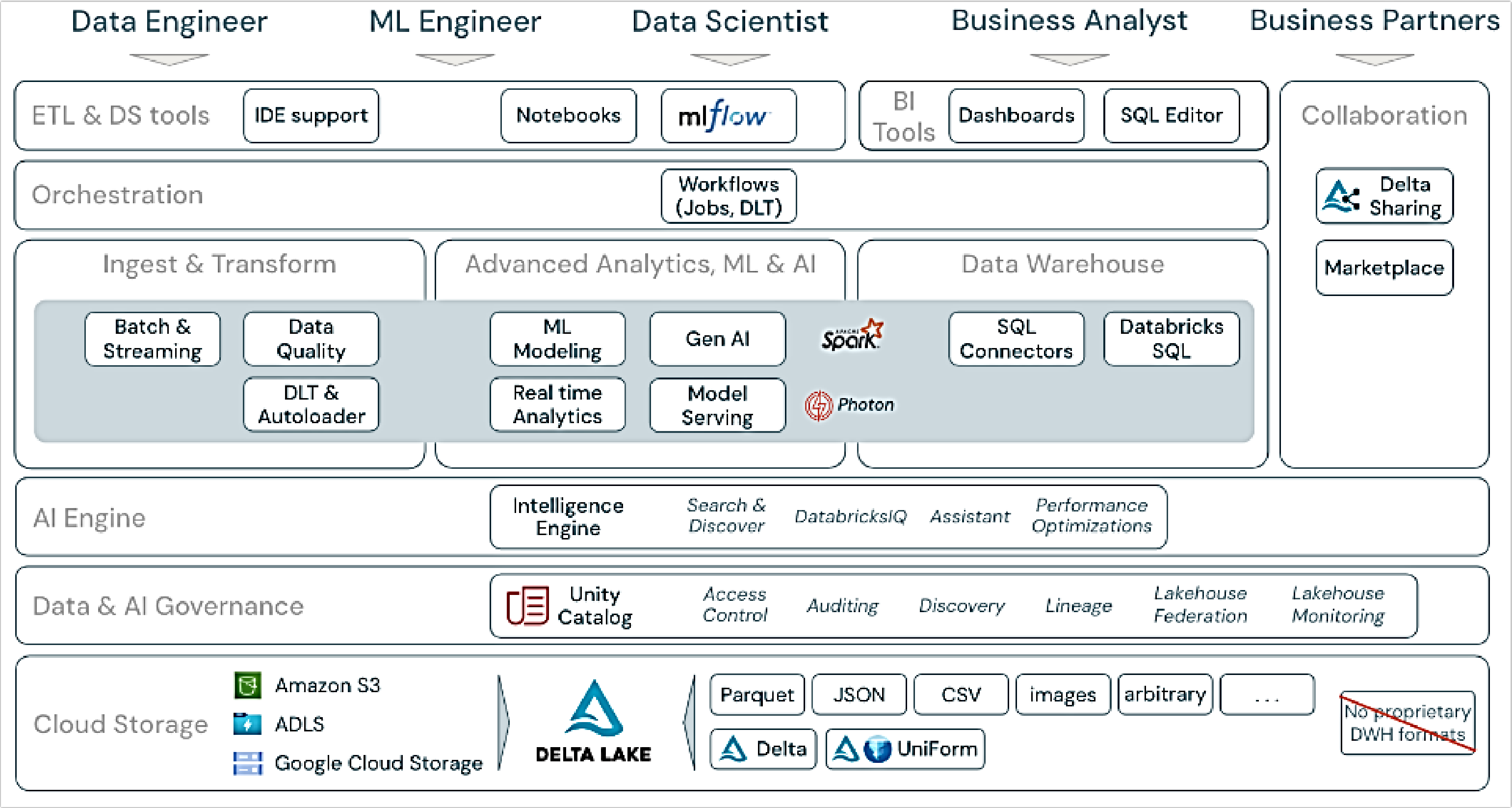
Databricks Lakehouse

PROFINIT

- https://docs.databricks.com/en/lakehouse/index.html
- Combines best elements from
  - Data warehouses
    - ACID transactions, data governance
  - Data lakes
    - Flexibility, cost-efficiency
- Built on top of open source technologies – Parquet, Apache Spark, Delta Lake, MLFlow – prevents vendor-lock
- **Delta tables** (stored with Delta Lake protocol)
  - ACID, Data versioning, ETL, indexing
- **Unity Catalog**
  - Data governance, Data sharing, Data auditing, Data lineage

Databricks Lakehouse

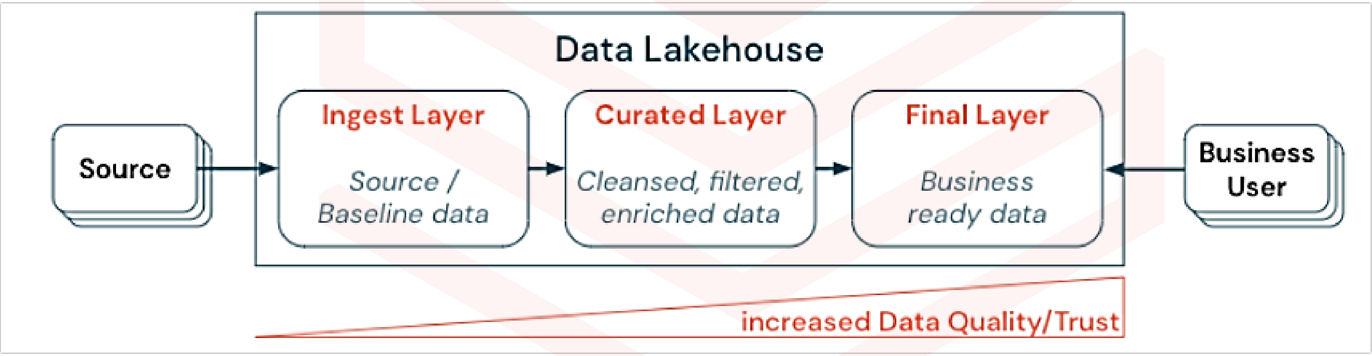
PROFINIT



Data Lakehouse principles

PROFINIT

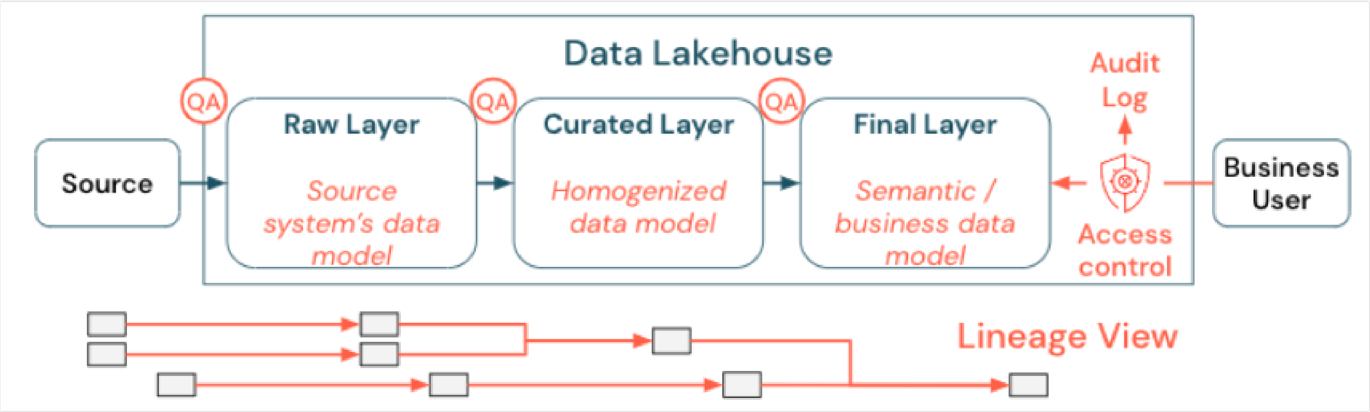
- Multi-hop (medallion) architecture
  - Curate data and offer trusted data-as-products
  - (Landing) → Ingest → Curated → Final
  - (Raw) → Bronze→ Silver→ Gold



Data Lakehouse principles

PROFINIT

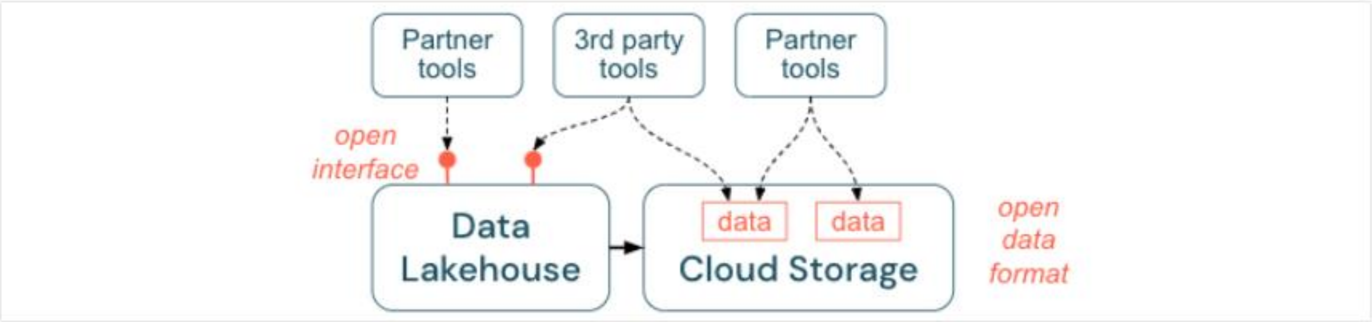
- Adopt an organization-wide data governance strategy



Data Lakehouse principles

〈 PROFINIT 〉

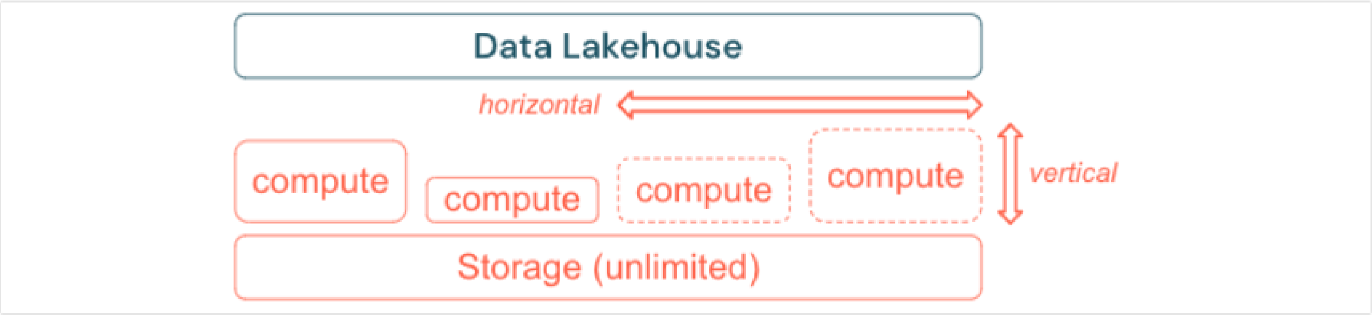
- Encourage open interfaces and open formats



Data Lakehouse principles

PROFINIT

- Build to scale and optimize for performance and cost



Databricks core features

Databricks core features

PROFINIT

- **Decoupled compute from storage**
  - Storage provided by cloud object storage (e.g. AWS S3) or external locations
  - Compute provided by compute clusters
    - Clusters also have their own disk attached
- **Storage layer powered by Delta Lake**
  - **Data versioning, historization**
  - **Indexing, optimization**
  - **ACID transactions**
  - **Optimized for structured streaming**
- **Databricks workflows (jobs)**
  - Running non-interactive workloads
  - On schedule, on demand
  - Notifications

Databricks clusters

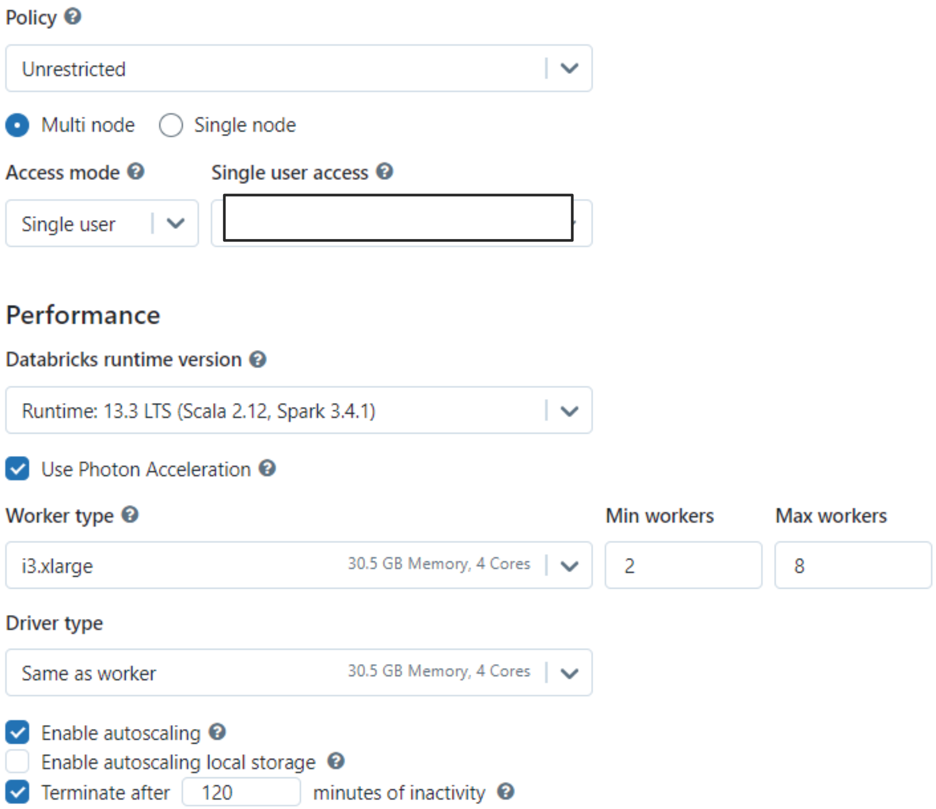
PROFINIT

- Computation resources for data engineering, data science and analytics workloads
- Created on classic data plane = your AWS account
- Running Spark
- **All-purpose clusters**
  - For interactive workloads, usually used with notebooks
  - Can be shared accross multiple users
- **Job clusters**
  - For non-interactive workloads, automated jobs
  - Is terminated when job is finished
- Controlled with UI, CLI, or REST API
- Pools
  - Keep warm instances as idle to reduce start and scale-up times, ! costs

Databricks clusters

Databricks clusters

PROFINIT



- 1 driver node, 0-n worker nodes
- Autoscaling
  - Add or remove instances from the cluster based on the workload
- Init script for custom initializations
- Arbitrary Spark configurations
- Policy, access mode
- Databricks runtime
  - Scala, Spark preinstalled
- Autotermination
- Tags (Metadata)
- Arbitrary log destinations

Delta Lake

- Default data storage format
- Data stored as Parquet files
- ACID transactions
  - Secured by transaction log, tracks all changes made to the table
- Data are versioned
  - Keep data files for every version (w.r. to retention period)
  - Time travel

Databricks workflows

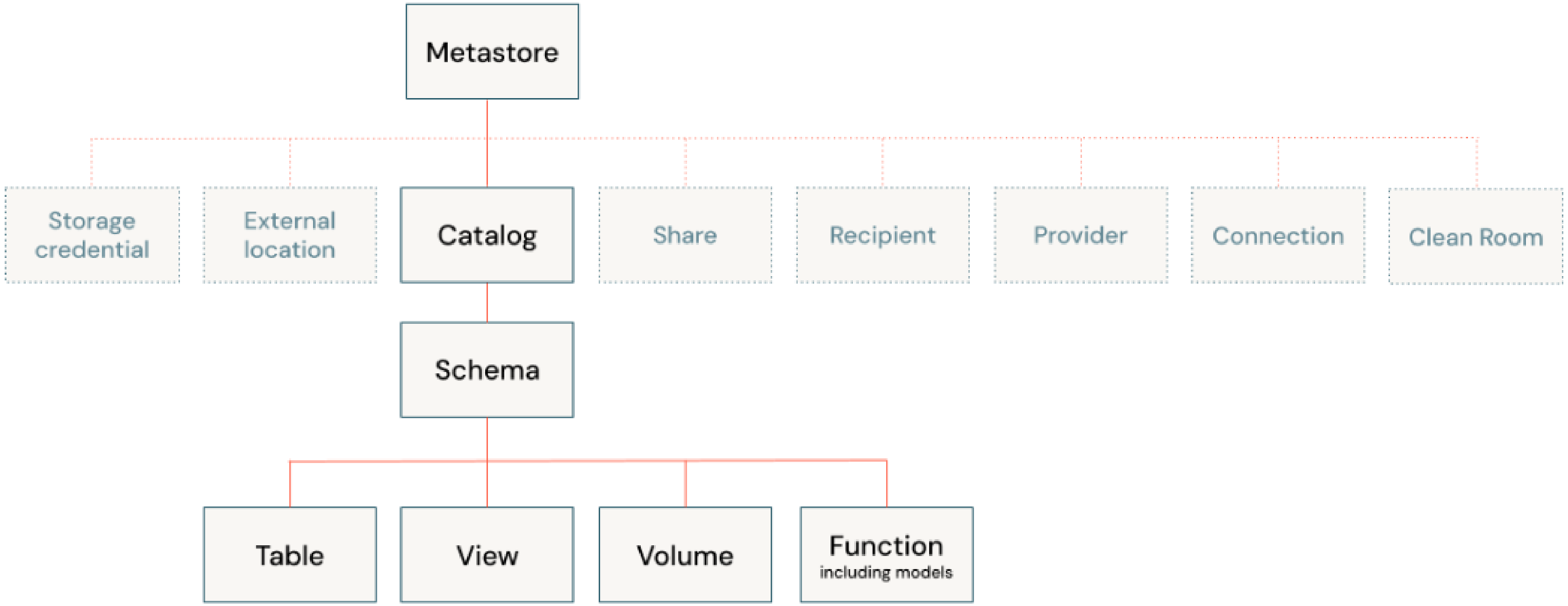
PROFINIT

- Using job clusters
  - Job clusters are terminated immediately after job is finished
- Consisting of tasks
  - Python script
  - Spark submit
  - Notebook
  - JAR, Python wheel
  - SQL – Query, Dashboard, Alert
  - Job
- Compute can be shared or different cluster can be selected for different tasks
- Run on demand/schedule/trigger
- Databricks native alternative to open source orchestration tools (AirFlow, Dagster, etc.)
- Can show nice DAG (graphical view)

Data objects

PROFINIT

- Kept and organized in cloud object storage (AWS S3, Azure Blob Storage,…)



Data objects

PROFINIT

- **Metastore**
  - Contains metadata of data objects
  - Configured with root storage in cloud object storage (e.g. S3 bucket in AWS)
  - Can be assigned to multiple workspaces
  - One workspace may have only a single metastore
- **Catalog**
  - The highest abstraction in DBX Lakehouse relational model
  - Collection of schemas (databases)
  - Default catalog is *hive_metastore*
- **Schema**
  - LOCATION on cloud object storage
  - Collection of tables, views and functions

Data objects

PROFINIT

- **Table**
  - Collection of structured data
  - Default storage provider – Delta Lake (https://delta.io/)
    - ACID transcations
    - Optimized performance (OPTIMIZE, Z-ORDER,…)
    - Driven by parquet
  - Managed table
    - In the same location as database
    - Metadata and data is managed by Databricks
    - DROP = delete data and metadata

```
CREATE TABLE table_name AS SELECT * FROM another_table
```

  - Unmanaged table, EXTERNAL
    - Only metadata is managed by Databricks
    - DROP = data is preserved

```
CREATE TABLE table_name
(field_name1 INT, field_name2 STRING)
LOCATION '/path/to/empty/directory'
```

Data objects

PROFINIT

- **View**
  - Query text is registered to the metastore (database)
  - No actual data is written

```sql
CREATE VIEW main.default.experienced_employee
  (id COMMENT 'Unique identification number', Name)
  COMMENT 'View for experienced employees'
AS SELECT id, name
   FROM all_employee
   WHERE working_years > 5;
```

- **Temporary view**
  - Limited scope and persistence
  - Not registered to metastore
  - Scopes:
    - Notebooks and jobs
    - Databricks SQL – query level
    - Global temporary views – cluster level

```sql
CREATE TEMPORARY VIEW subscribed_movies
  AS SELECT mo.member_id, mb.full_name, mo.movie_title
       FROM movies AS mo
       INNER JOIN members AS mb
          ON mo.member_id = mb.id;
```

Data objects

PROFINIT

- **User-defined function**
  - Associate user-defined logic with a database
  - In SQL or Python/Scala/Java
    - Code in Python can have a negative impact on performance
      - Outside of JVM – data serialization
      - Databricks have code optimizers for SQL, not Python
  - Usually not good for production workloads (instead use native Apache Spark methods if possible)

```sql
CREATE FUNCTION convert_f_to_c(unit STRING, temp DOUBLE)
RETURNS DOUBLE
RETURN CASE
  WHEN unit = "F" THEN (temp - 32) * (5/9)
  ELSE temp
END;

SELECT convert_f_to_c(unit, temp) AS c_temp
FROM tv_temp;
```

```python
def convertFtoC(unitCol, tempCol):
  from pyspark.sql.functions import when
  return when(unitCol == "F", (tempCol - 32) * (5/9)).otherwise(tempCol)

from pyspark.sql.functions import col

df_query = df.select(convertFtoC(col("unit"), col("temp"))).toDF("c_temp")
display(df_query)
```

Data objects

PROFINIT

- **Volume**
  - Represents logical volume of storage in cloud object storage location
  - Accessing, storing, governing and organizing files
  - Add governance over also to non-tabular datasets
  - Only in Unity Catalog
  - **Managed**

```
CREATE VOLUME myManagedVolume
  COMMENT 'This is my example managed volume';
```

```
SELECT * FROM csv.`dbfs:/Volumes/mycatalog/myschema/mymanagedvolume/sample.csv`
```

  - **External**

```
CREATE EXTERNAL VOLUME IF NOT EXISTS myCatalog.mySchema.myExternalVolume
      COMMENT 'This is my example external volume'
      LOCATION 's3://my-bucket/my-location/my-path'
```

```
SELECT * FROM csv.`/Volumes/mycatalog/myschema/myexternalvolume/sample.csv`
```

Advanced Databricks features – to be continued

PROFINIT

- Machine learning tooling
  - MLFlow, Scalable ML with Spark, AutoML, Model serving
- Delta Live Tables
  - ETL tool
  - Declarative definitions
  - A lot of „self optimization and maintanance“
  - Development or production modes
- Photon
  - New generation data processing engine
  - Written in C++
  - Compatible with Apache Spark APIs
- SQL warehouses
- Lakehouse federation
- LakehouseIQ

Real-world Databricks use cases

PROFINIT

- **Gucci**
  - **Use case: media budget allocation to maximize ROI** https://www.youtube.com/watch?v=mq3IxO_toDA
  - MLOps
  - Trying to adopt community-recommended best practices
  - Speed-up time to market
  - Benefit from managed ML services – distributed hyperparameter tuning with HyperOpt and Spark, MLFlow, AutoML (kick-off stage)
- **CDQ**
  - **Use case: migrate custom reporting ETL pipeline to Databricks**
  - Get scalable solution with usage of Delta Live Tables
  - Exploit Lakehouse architecture
  - Performance boost
- **Shell**
  - **Use case: Databricks as key tool in Shell.ai platform** https://www.databricks.com/customers/shell
  - Democratize data access in organization, supported cross-team collaboration, develop over 100 AI models

Q&A

Apache Spark - basics

Martin Oharek, Alisa Benešová

October 7th, 2025

Outline

Outline

1. Spark overview
2. How Spark works
3. Spark Dataframes
4. Spark architecture
5. Spark configuration
6. Spark vs Databricks

Spark overview

The What, Why and When of Apache Spark

PROFINIT

- **What:**
  - Unified engine for big data and machine learning
  - Distributed data processing engine -> up to petabytes of data up to thousands of physical or virtual machines
  - Open Source with over 1000 contributors from 250+ organizations
  - Founded by people who founded Databricks
- **Why:**
  - High speed data querying, analysis, and transformation with large data sets.
  - Great for iterative algorithms (using a sequence of estimations based on the previous estimate).
  - Supports multiple languages (Java, Scala, R, Python)
  - Free of charge
- **When:**
  - When you're using functional programming (output of functions only depends on their arguments, not global states)
  - Performing ETL or SQL batch jobs with large data sets
  - Processing streaming
  - Machine Learning tasks

Spark - facts

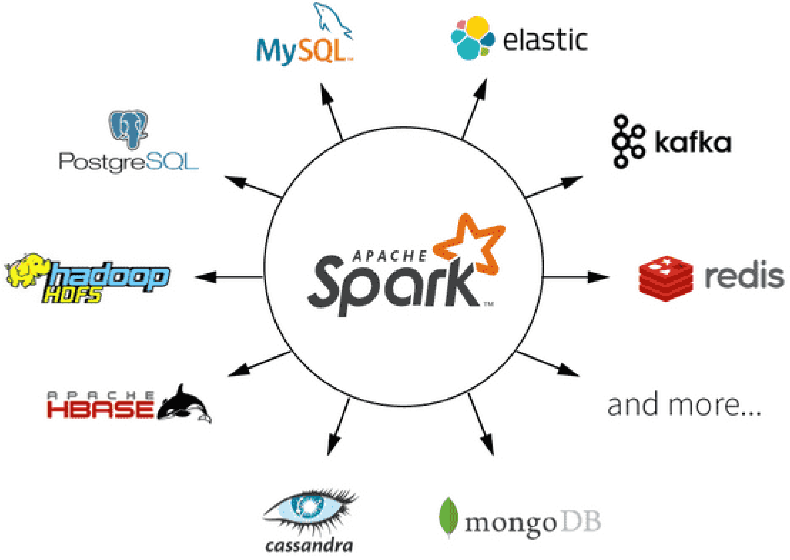
PROFINIT

- In-memory Map-Reduce engine
- Written in Scala
- Fault-tolerant
- Connected with all major big data technologies
- Runs „Everywhere“

Google Cloud

aws

Microsoft Azure



Apache Spark Evolution

- **Spark 1.x – 2014 :**
  - Spark CORE - Fault-tolerant in memory computation engine
  - Spark RDD (Resilient Distributed Dataset) API
  - API for Streaming and Mlib
  - Spark SQL
- **Spark 2.x - 2016:**
  - Speedups the computation 5 to 20 times.
  - API for structured Streaming
  - API for graph data processing
  - SQL 2003 support
  - Datasets API over RDD
- **Spark 3.x - 2020:**
  - adaptive query execution, dynamic partition pruning and other optimizations
  - Significant improvements in pandas APIs, including Python type hints and additional pandas UDFs
  - Up to 40x speedup for calling R user-defined functions
  - SQL ANSI supports

When does Spark work best?

- Enable collaboration between data engineers, data scientists, BI analysts, and more
- Support both batch and streaming processing workflows
- Common Use Cases:
  - Client scoring: Risk scoring, fraud detection
  - ETL and batch SQL jobs: Data processing and aggregation
  - Streaming data triggers: Real-time event response (e.g., alerts, notifications)
  - Machine Learning: Model training and inference at scale
  - Graph algorithms: Social networks, recommendations, fraud network detection

When Spark is not so good / appropriate?

PROFINIT

- Low-latency, real-time applications
  - Spark (even with Structured Streaming) has higher latencies (hundreds of ms+)
  - Not suitable for use cases needing sub-second responses (e.g., real-time bidding, chat apps)
- Small or simple datasetsSpark introduces overhead due to distributed execution
  - For small data or lightweight ETL, pandas, SQL, or dbt may be more efficient
- Highly interactive use (BI dashboards)
- Complex stateful stream processing
  - Spark Structured Streaming supports state, but with limits
  - Kafka Streams, Flink, or other stream-first engines handle complex event time and state bette
- Very tight resource constraints
  - Spark is memory-intensive; not ideal for constrained environments (e.g. edge devices, IoT)

How to work with Spark?

PROFINIT

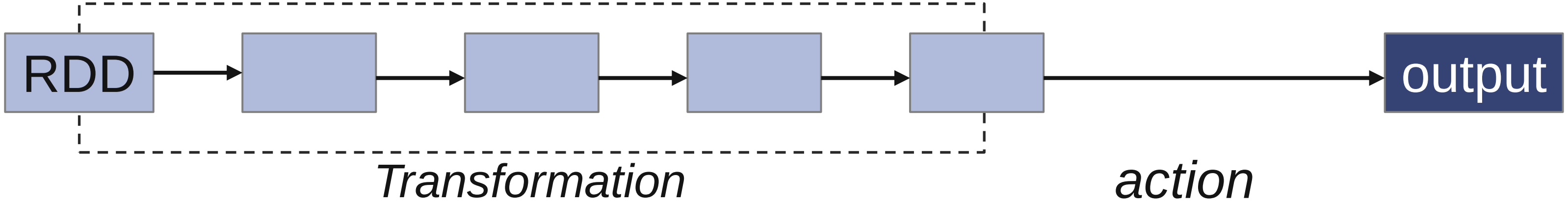
- Interactively
  - Command line (shell for both Python and Scala)
  - **Databricks notebook**
  - Zeppelin/Jupyter notebook
  - From IDE (Pycharm, IntelliJ, …)

- Batch / application
  - compiled .jar file
  - *.py file

- Learning path:
  - http://spark.apache.org
  - https://www.databricks.com/spark/getting-started-with-apache-spark

How Spark works

Logical point of view

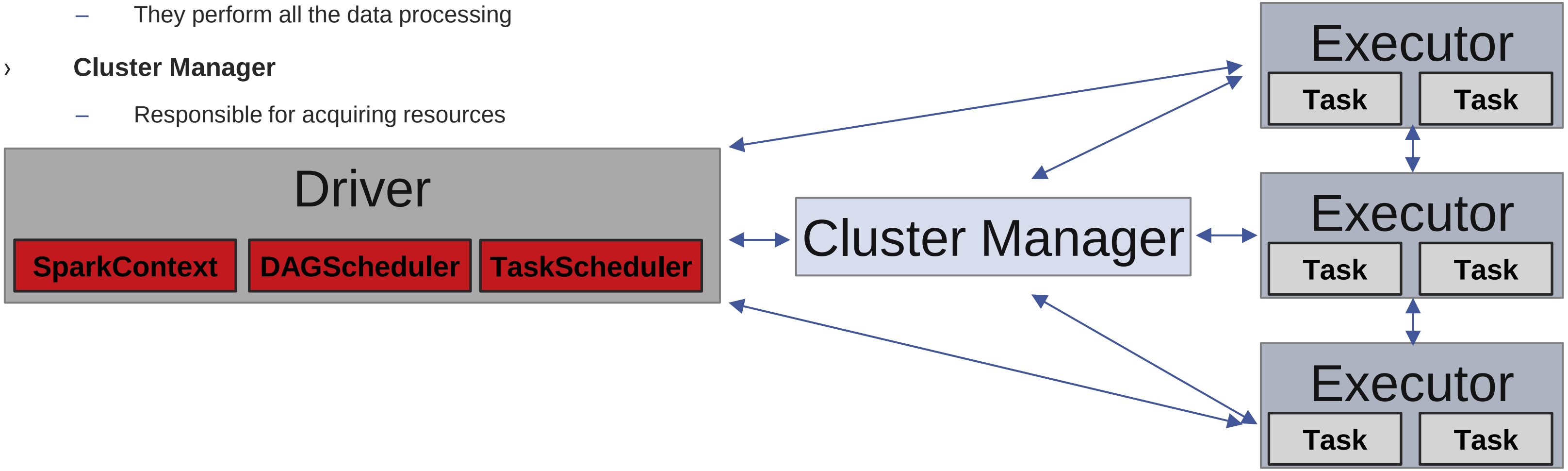


- **RDD:**
  - resilient distributed dataset - the abstractions of Spark. It is used to handle distributed collection of data elements (e.g.: rows in text file, data matrix, set of binary data) across all the nodes in a cluster.
  - is immutable
- **Transformation:**
  - are planned and optimized, but not evaluated
  - planned as DAG – Direct acyclic graph
- **Action – lazy evaluation:**
  - action is a trigger that started the whole process

Components of Spark Architecture

PROFINIT

- **Driver**
  - It is a **master node**.
  - Translates user code into a specified job.
  - Schedules the job execution and negotiates with the cluster manager.
  - Stores the metadata about all RDDs as well as their partitions.
  - The key component is a SparkContext, others are DAG Scheduler, Task scheduler, backend scheduler and block manager.
- **Executors - workers**
  - They are distributed agents those are responsible for the execution of tasks
  - They perform all the data processing
- **Cluster Manager**
  - Responsible for acquiring resources



Example – word count

- Task: **count number of words in document**
- Source: text file splitted to lines
- Approach:
  - Load file from disk
  - Transformation of lines: line ⇨ split to words ⇨ split to items (word, 1)
  - Group items with the same word and sum up ones
- Result of transformation: RDD with items (word, frequency)

Example – word count

PROFINIT

Transformation:

```
lines = sc.textFile("bible.txt")
words = lines.flatMap(lambda line: line.split(" "))
items = words.map(lambda word: (word, 1))
counts = items.reduceByKey(lambda a, b: a + b)
```

Action:

```
counts.take(5)
```

Spark Dataframes

Spark SQL and DataFrames (DataSets)

PROFINIT

- New from spark 2.x ⇨ Enhances the classical RDD approach
- Data structure **DataFrame** = „RDD with columns“
  - like database relation table
  - with metadata (field names, types)
  - works with columns –> SQL syntax can be used
- RDD

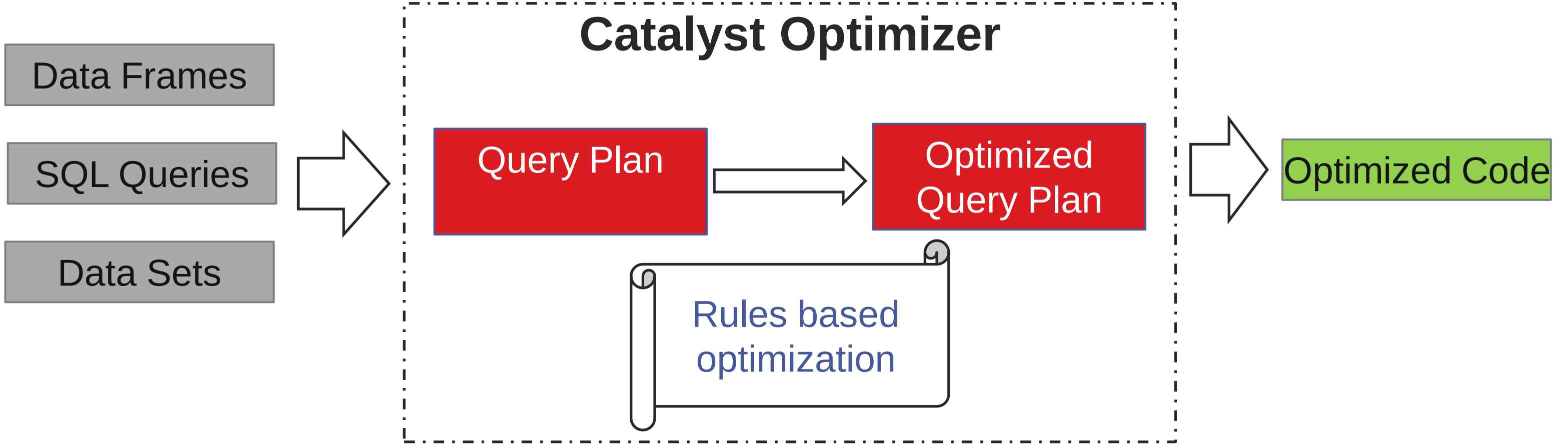
1;Andrea;35;64.3;Praha
2;Martin;43;87.1;Ostrava
3;Simona;18;57.8;Brno

- Dataframe

id	name	age	weight	city
1	Andrea	35	64.3	Praha
2	Martin	42	87.1	Ostrava
3	Simona	18	57.8	Brno

WHY use DataFrames

- Advantage over Spark RDD:
  - Dataframe API - shorter and easier code
  - Columns and Types
  - SQL languague can be used
  - Simplified work with databases
  - **Catalyst Optimizer can be applied ⇨ is faster**



How to get a DataFrame?

PROFINIT

- transformation from existing RDD
  - if convertable
  - `sqlContext.createDataFrame(RDD, schema)`
- direct input of file
  - schema may be defined (Parquet, ORC) or inferred (CSV)
  - `sqlContext.read.format(format).load(path)`
- Hive query
  - `sqlContext.sql(sql_query)`

How to work with a DataFrame?

PROFINIT

1. registration of temporary table + SQL querying
   - `DF.registerTempTable("table")`
   - `sqlContext.sql("select * from table")`
2. SPARK API
   - `DF.operations`; select, filter, join, groupBy, sort...
3. Convert to RDD -> RDD operation (map, flatMap, …) and then convert back -> Dataframe

Example – word count with Dataframes

- Transformation
- *df_final = (*
- *df.withColumn("word", explode(split(col("lines"), ' ')))*
- *.groupBy("word")*
- *.count()*
- *)*

```
df_final = (
    df.withColumn("word", explode(split(col("lines"), ' ')))
    .groupBy("word")
    .count()
)
```

- Action

```
df_final.show()
```

Example – word count with Spark SQL

PROFINIT

- Transformation
- `df.registerTempTable("temp_df")`
- *`df_final = (`*
- *`sqlContext.sql("`*
- *`    SELECT word, count(*) FROM`*
- *`    (SELECT explode(split(Description, ' ')) AS word FROM temp_df)`*
- *`    GROUP BY word`*
- *`")`*

- Action

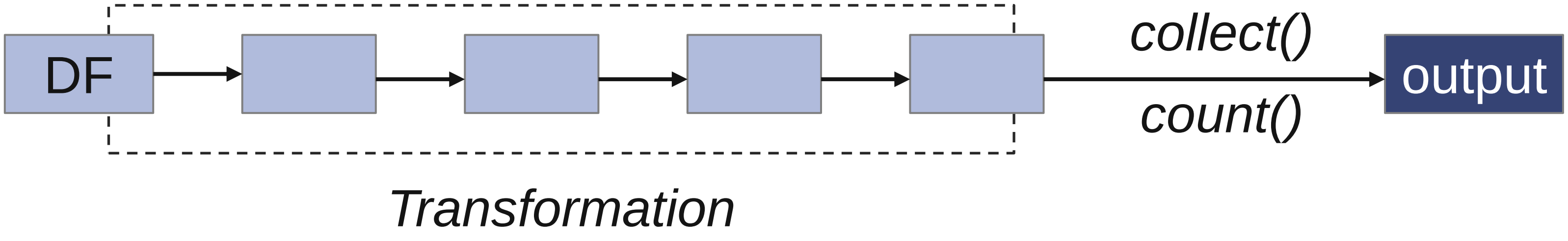
  `df_final.show()`

Spark Actions

Spark Actions

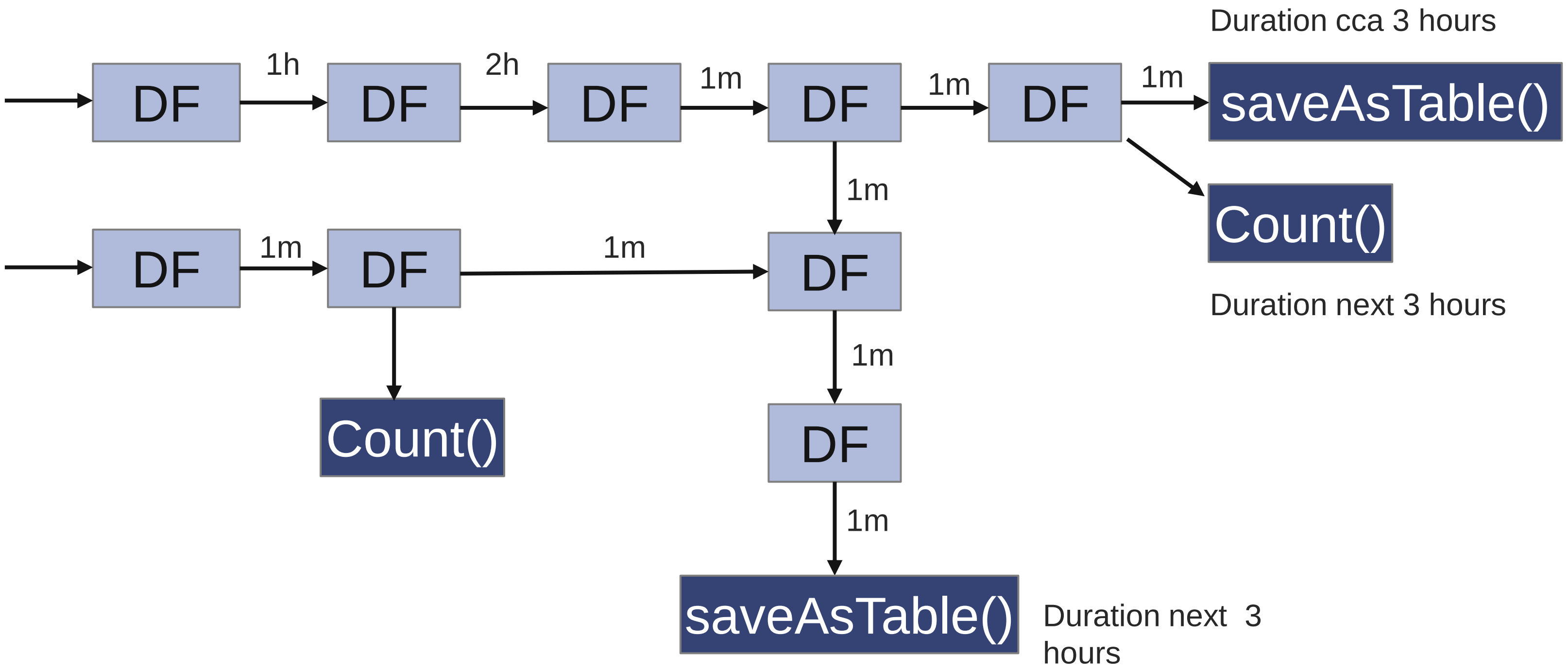
PROFINIT

RDD	Dataframe	Description
take	take,show	Show first n rows
count	count	Count of rows
collect	collect	Show rdd/dataframe as list of rows
saveAsTextFile	saveAsTable, write	Save file/create table
…	…	



- Every action starts all steps of transformation from the beginning!

Spark Actions



Spark Caching

- **Methods:**
  - `persist()` (several options)
  - `cache()` (use persist with MEMORY_ONLY option)
  - `unpersist()` (release persisted data)

- **Persist options:**
  - MEMORY_ONLY – Default –> deserialized JVM memory
  - MEMORY_AND_DISK –> excessed partitions into disk.
  - MEMORY_ONLY_SER -> serialized JVM memory
  - MEMORY_AND_DISK_SER -> etc.

- Persist is not an action!

Spark Caching

❬ PROFINIT ❭

- Different from (proprietary) **Databricks Disk Cache – optimized caching on SSDs**

Feature	disk cache	Apache Spark cache
Stored as	Local files on a worker node.	In-memory blocks, but it depends on storage level.
Applied to	Any Parquet table stored on S3, ABFS, and other file systems.	Any DataFrame or RDD.
Triggered	Automatically, on the first read (if cache is enabled).	Manually, requires code changes.
Evaluated	Lazily.	Lazily.
Force cache	`CACHE SELECT` command	`.cache` + any action to materialize the cache and `.persist`.
Availability	Can be enabled or disabled with configuration flags, enabled by default on certain node types.	Always available.
Evicted	Automatically in LRU fashion or on any file change, manually when restarting a cluster.	Automatically in LRU fashion, manually with `unpersist`.

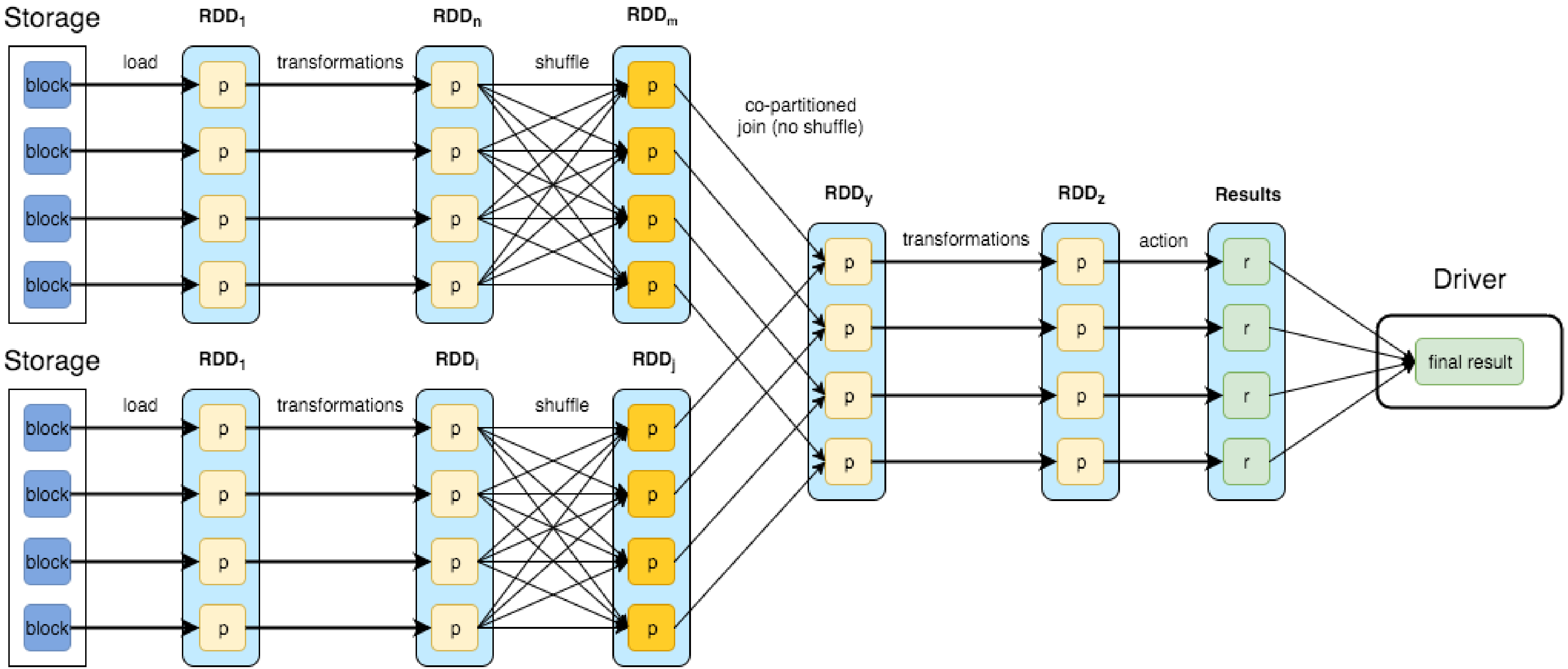
- Cache consistency:
  - Databricks disk caching – changes are automatically detected and cache is updated
  - Spark caching – cache must be manually invalidated and refreshed

Spark data partitions

PROFINIT

- **Partition**
  - part of data managed in one task
  - default partition = 1 HDFS block = 1 task = 1 core
  - partition is ideally managed on the node where is stored – data locality!
  - More partitions ⇨ more tasks ⇨ higher parallelization
    - ⇨ smaller data ⇨ lower efficiency ⇨ higher overhead
- **Default for Joins:**
  - The default number of partitions to use when shuffling data for joins or aggregations.
  - `spark.sql.shuffle.partitions = 200`
- **How to change number of partition?**
  - in load: `sc.textFile(file, count_of_partitions)`
  - In the code (before/after specific transformation/action):
    - `coalesce (count_of_partitions)`
    - `repartition(count_of_partitions)`
    - `partitionBy (count_of_partitions)`

Data locality & shuffling



Start and configuration

PROFINIT

- `pyspark | spark-shell | spark-submit --*param value*`
- **Useful parameters:**
- --name -> name of the application
- *--class -> The entry point for your application*
- *--master -> The master URL for the cluster (local, Yarn, Mesos, ..)*
- *--deploy_mode -> where the driver will be deployed (client/cluster)*
- --driver-memory -> memory for driver
- --num-executors -> count of executors
- --executor-cores -> count of cores for executor
- --executor-memory -> memory for *executor*
- *NOTE:* **Spark is deployed in Databricks clusters by default and Spark Context (Spark session) is initialized, you don‘t need to care about running Spark on your own**

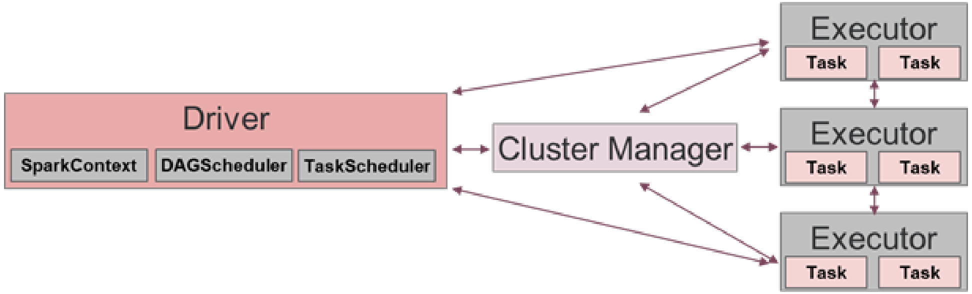
Deploy mode

Deploy mode (execution mode)

PROFINIT

- **Deploy mode**
- Determines where the resources used by Spark application are physically located
- **Deploy mode types:**
  - Local mode
  - Client mode
  - Cluster mode

- **Differences:**
- Where the driver runs – client or cluster ?
- Where the executors run - client or cluster ?
- What is cluster manager – spark CM or 3rdParty (yarn, messos, ..)



Deploy mode: Local mode

PROFINIT

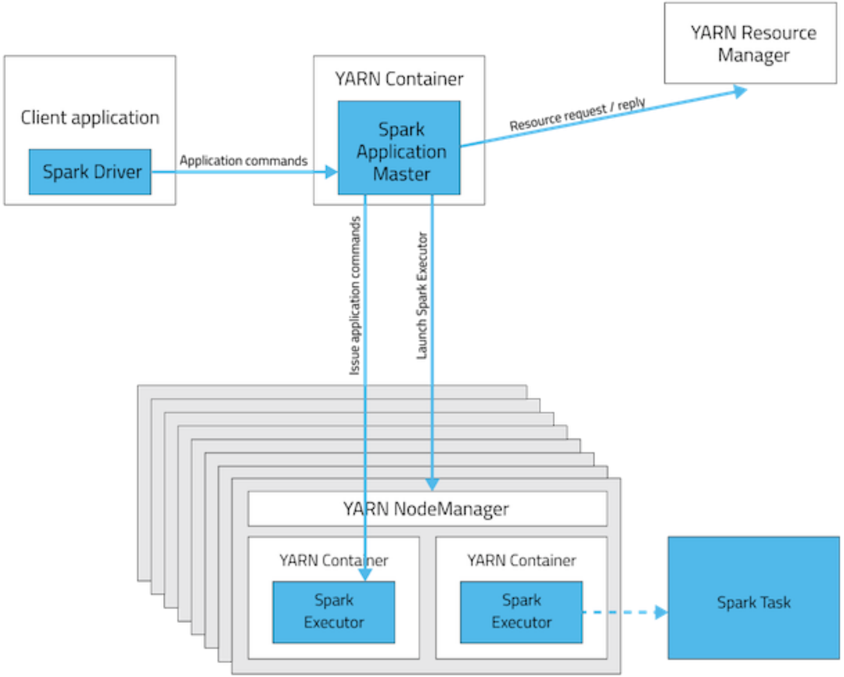
- Properties:
  - The entire application is run on a single machine (paralelism through threads)
  - The Spark **driver runs on the client machine**
  - The **Executor** processes **run on the client machine**
  - **Spark CM** is used
  - Used on Databricks single node clusters
- Purpose:
  - Development
  - Debugging
  - Testing

Deploy mode: Client mode

PROFINIT

- Properties
  - The **Spark driver runs on the client machine** that submitted the application (usually an edge node)
  - The **executor** processes run on **cluster**
  - Cluster manager is used
  - On Databricks multi-node clusters in interactive environment (e.g. Notebook)
- Purpose
  - Spark-shell (interactive sessions)
  - Easy debugging
    - Input and output attached
- Can overload the edge node

Deploy mode: Client mode (example for YARN)

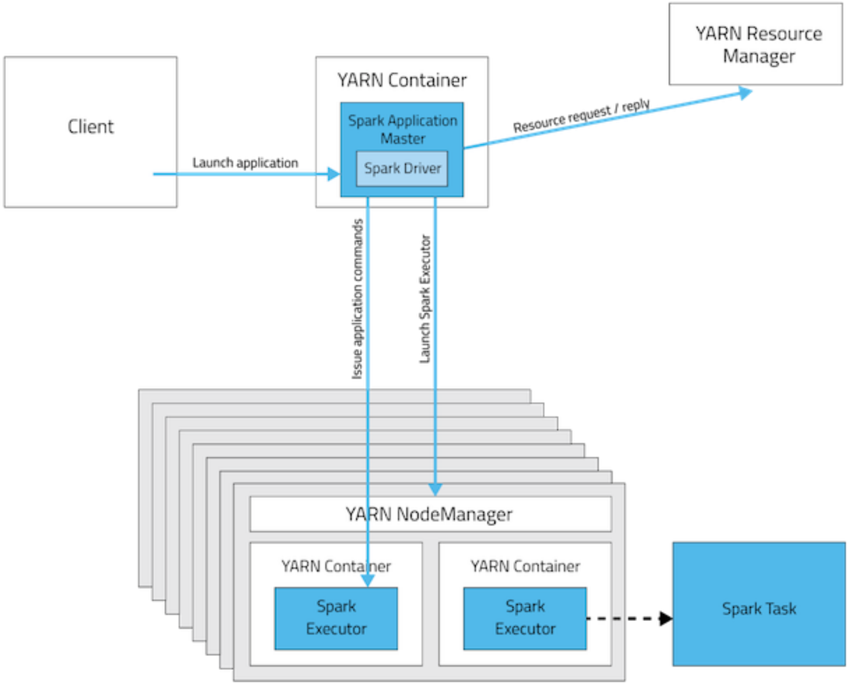


Deploy mode: Cluster mode

- Properties
  - The **Spark driver runs on a worker node** inside the cluster
  - The **executor** processes run on **cluster**
  - The cluster manager maintans the executor processes
  - **Databricks job clusters**
- Purpose
  - The best deploy mode for stable applications
  - Better resource utilization than in client mode
  - More difficult debugging

Deploy mode: Cluster mode (example for YARN)

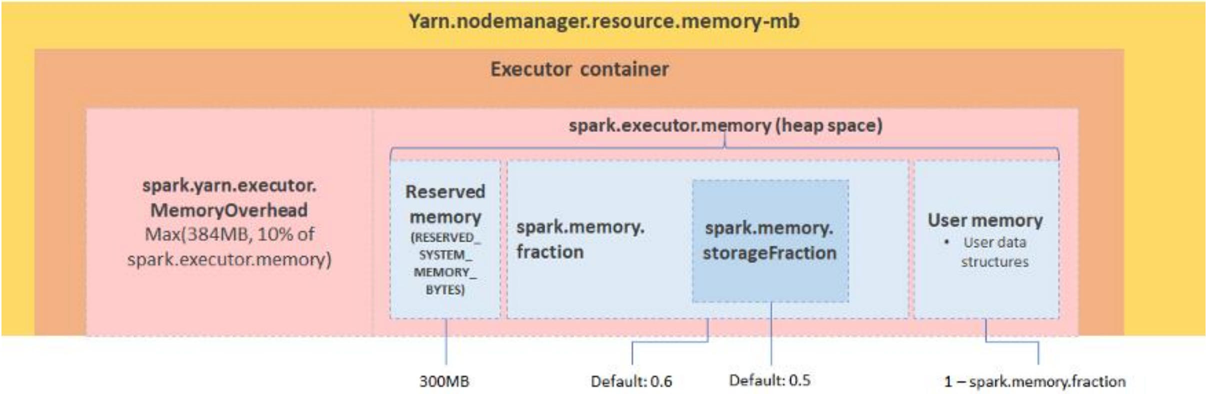
PROFINIT



Spark configuration

Spark executor memory

PROFINIT



- **Reserved** - the memory is reserved for the system and is used to store Spark's internal object. The size is hardcoded.
- **User Memory** - It's used for storing your data structures and data needed for RDD conversion operations, such as lineage.
- **Unified memory:**
  - **Execution memory** - It's mainly used to store temporary data in the calculation process of Shuffle, Join, Sort, Aggregation, etc.
  - **Storage Memory** - It's mainly used to store Spark cache data, such as RDD cache, Unroll data, and so on.
  - *Size* of an Execution and Storage memory can by *dynamically changed* by the **Dynamic occupancy mechanism** process.
- **Memory overhead** - Off heap (no GC). Call stacks, shared libraries, constants defined in Code, the code itself, ….

Spark executor memory example

PROFINIT

- Spark.executor.memory = 4 GB

  - **Memory overhead** = 10% of executor memory, max 384 MB
  - **Reserved memory** = 300 MB
  - **User Memory** = (Java Heap — Reserved Memory) * (1.0 — spark.memory.fraction) = (3640-300)* (1-0,6)= 1336 MB
  - **Storage Memory** = (Java Heap — Reserved Memory) * spark.memory.fraction * spark.memory.storageFraction = (3640-300)*(0,6*0,5)= 1002 MB
  - **Execution Memory** = (Java Heap — Reserved Memory) * spark.memory.fraction * (1.0 — spark.memory.storageFraction) = (3640-300)*(0,6*0,5)=1002 MB

Resources configuration: Spark Driver

Considerations:

- Client or cluster mode
- With the client mode – beware of overloading Edge node
- Size of the result returned by executor (collect action)

Resources configuration: Spark Executor

PROFINIT

Considerations

- Resources available in the cluster, sizing of cluster nodes
- Few large executors or many small executors?
  - **Small executors**
    - Higher parallelization but more shuffling
    - One partition - one executor, risk of spilling the data to disk
    - Total overhead grows (Reserved memory)
  - **Large executors**
    - Lower parallelization
    - Issue with resources allocation
    - Might be wasteful
    - GC overhead

Resources configuration: Recommendations

PROFINIT

- Allows Spark dynamically change the number of executors based on the workload
- Number of cores – deside based on the load. Usually 2 - 4 cores/executor
- Driver memory – keep default
- Executor memory – ((data size) *1,5)/0,6) / number of executors (max 16G)
  - For start use `spark.executor.memory = 2G.`
- Number of executors - number of task / executor > 100
  - For start use `spark.dynamicAllocation.maxExecutors` < 10.
- For long running processes set `spark.sql.ui.retainedExecutions <= 100 (default 1000)`

Spark vs Databricks

Spark vs Databricks

- Databricks
  - Tool/platform built **on top of Apache Spark**
  - Add other functionality, e.g. Notebooks, production jobs and workflows, etc.
  - Databricks runtime
    - Built on Apache Spark and optimized for performance
      - Photon engine
      - Disk caching, dynamic file pruning, predictive I/O, cost-based optimizers, etc.
      - Auto-scaling compute
      - Pre-installed Java, Scala, Python and R libraries
      - …
  - Apache Spark is running on Databricks clusters
    - Can set spark configuration on cluster level and change some configurations during runtime
    - Managed Delta Lake
    - You **cannot** set spark configuration for managed compute (SQL warehouse)

Q&A

Thanks for attention!

Profinit EU, s.r.o.
Tychonova 2, 160 00 Praha 6

Tel.: + 420 224 316 016, web: www.profinit.eu

LINKEDIN
linkedin.com/company/profinit

TWITTER
@profinit_EU

FACEBOOK
facebook.com/Profinit.EU

YOUTUBE
Profinit EU, s.r.o.

PROFINIT

Thanks for attention!

Profinit EU, s.r.o.
Tychonova 2, 160 00 Praha 6

Tel.: + 420 224 316 016, web: www.profinit.eu

LINKEDIN
linkedin.com/company/profinit

TWITTER
@profinit_EU

FACEBOOK
facebook.com/Profinit.EU

YOUTUBE
Profinit EU, s.r.o.