Apache Spark Data Intro

Kim Horn

Version 1.75 8 April 2022

Agenda

- Spark Architecture and API History
- Spark's 3 Data Containers:
  - RDD
  - DataFrames
  - DataSets
- Spark SQL
- Big Data Data Formats
- Data Base Tables and Views
- Transformations and Actions
- Checkpoints
- Sharing
- Java RDSs
- Delta Lake - Multi-Hop Vs Lambda Architecture
- Data Mesh
- Stand Alone
- Docker Cluster
- Machine Learning
- EPC Appraoch

GitHub: https://github.com/infomediacode/spark-java-examples-two

What is Apache Spark ?

Apache Spark is a lightning-fast **unified analytics engine** for big data and machine learning. It was originally developed at UC Berkeley in 2009.

JVM based, built in Scala. Python, R and Java supported.

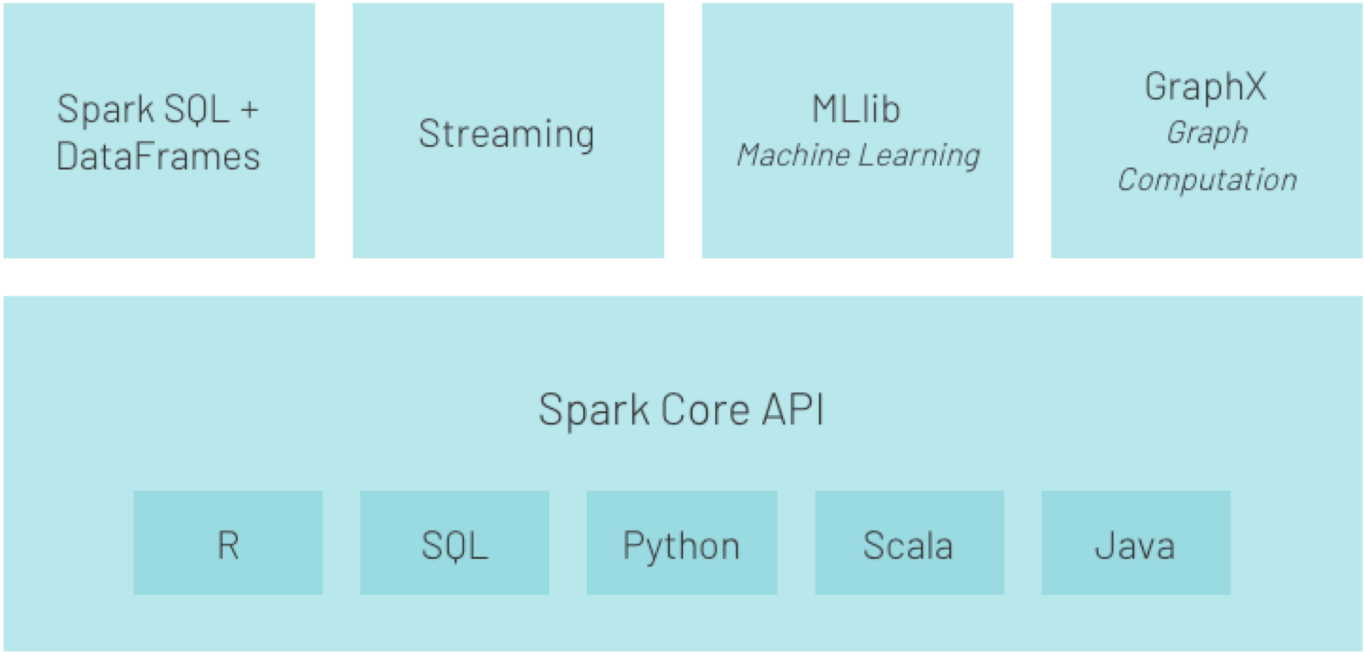
Runs on a cluster or locally. Batch or interactive.

Internet powerhouses such as Netflix, Yahoo, and eBay have deployed Spark at massive scale, collectively processing multiple petabytes of data on clusters of over 8,000 nodes.

Apache Spark is 100% open source, hosted at the vendor-independent Apache Software Foundation.

Databricks, are a company that provides Spark maintenance and technology. Together with the Spark community, Databricks contributes heavily to the Apache Spark project, through both development and community evangelism.

Spark Ecosystem



https://spark.apache.org/docs/latest/index.html
https://spark.apache.org/releases/spark-release-3-2-1.html
https://spark.apache.org/docs/latest/api/java/index.html

Spark 3.2.1 is latest

3 ways of interacting with Spark:

1. ***Local mode***, which is certainly the developers' preferred way, as everything runs on the same computer and does not need any configuration
2. ***Interactive mode***, either directly or via a computer-based notebook, which is probably the preferred way for data scientists and data experimenters
3. ***Cluster mode***, through a resource manager, which deploys your applications in a cluster

Spark Terms

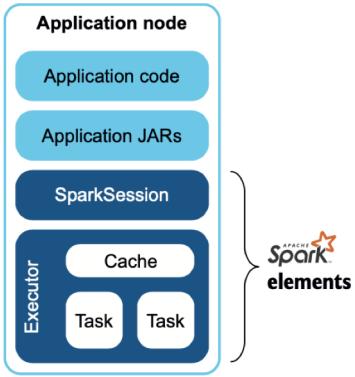
Spark Cluster is a collection of machines or nodes on which Spark is installed. Machines Include:

- **Spark Master (**JVM) that acts as a cluster manager in a Standalone deployment mode to which Spark workers register themselves as part of a quorum. Depending on the deployment mode, it acts as a resource manager and decides where and how many Executors to launch, and on what Spark workers in the cluster.
  - **Cluster Manager:** allocates resources to nodes
- **Spark Worker** (JVM) that **u**pon receiving instructions from Spark Master, launches Executors on the worker on behalf of the Spark Driver. Spark applications, decomposed into units of tasks, are executed on each worker's Executor.
- **Spark Executor (**JVM) container on which Spark runs its tasks. Each worker node launches its own Spark Executor, with a configurable number of cores (or threads). Besides executing Spark tasks, an Executor also stores and caches all data partitions in its memory.
- **Spark Driver** Part of the Spark Application. Instantiates a Spark Session**.** Talks to Cluster. Manager. Once it gets information from the Spark Master of all the workers in the cluster and where they are, the driver program distributes Spark tasks to each worker's Executor. The driver also receives computed results from each Executor's tasks.
  - **Spark Session:** Unified conduit for all operations and data: create JVM parameters, manages data, reads from data sources, cataloguer metadata, and executes SQL queries, etc

etc.

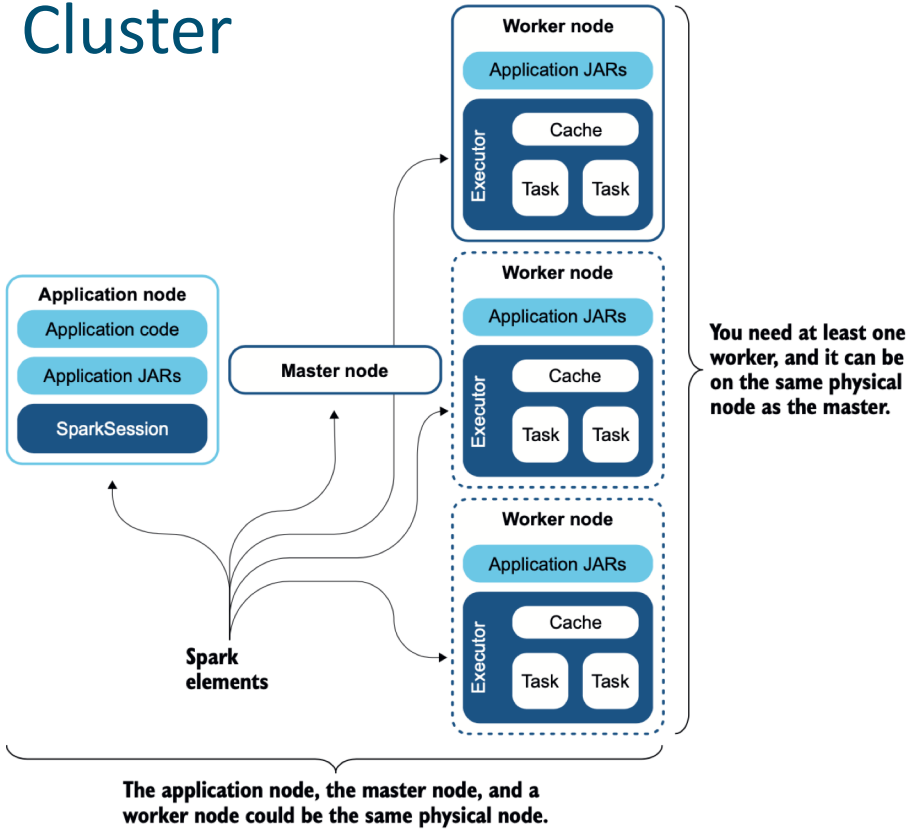
Local VS Cluster Mode

Local

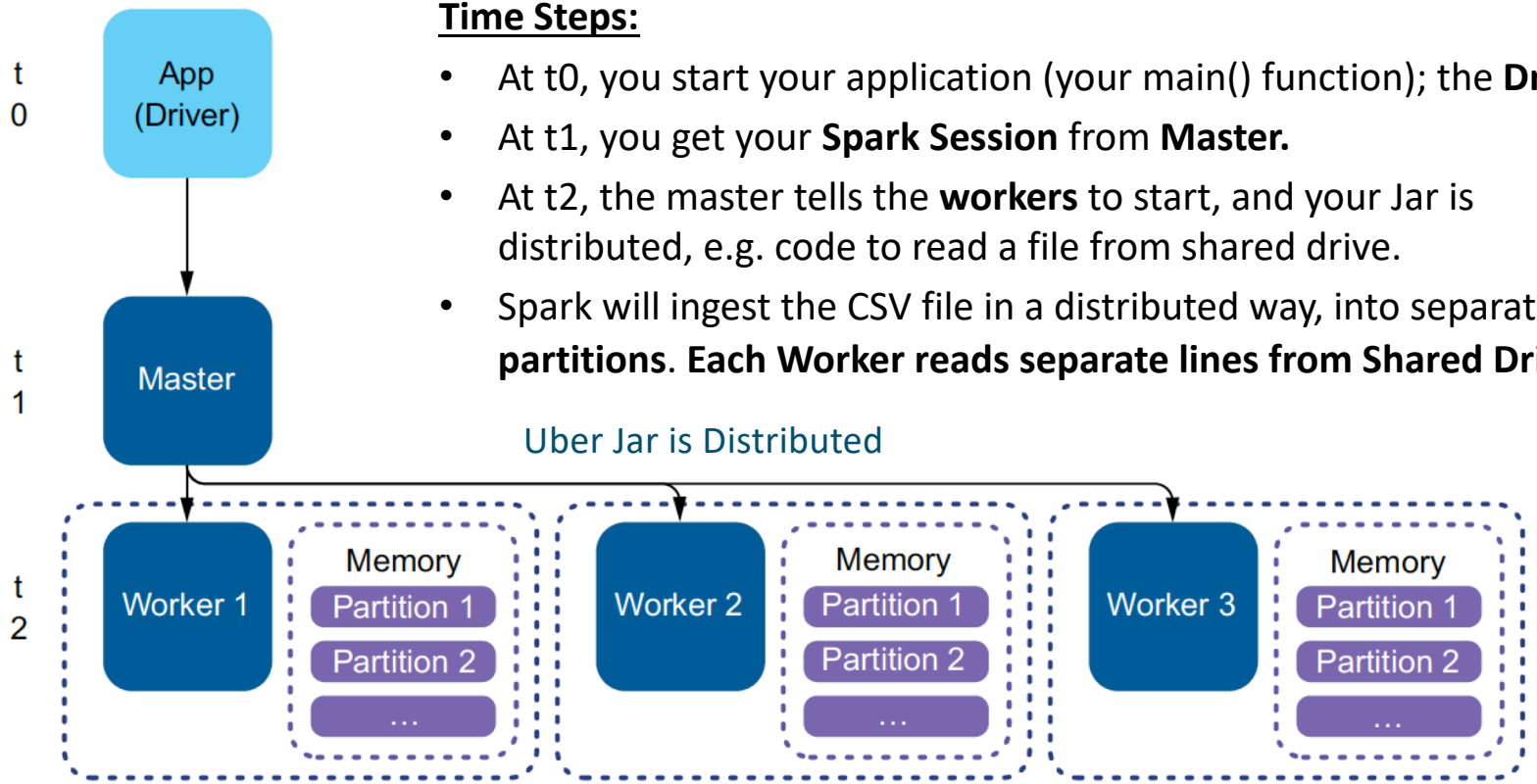


Application is separate to the Spark Elements

Cluster



Spark Architecture – Clustered Workers - Distributed Data



Time Steps:

- At t0, you start your application (your main() function); the **Driver**
- At t1, you get your **Spark Session** from **Master.**
- At t2, the master tells the **workers** to start, and your Jar is distributed, e.g. code to read a file from shared drive.
- Spark will ingest the CSV file in a distributed way, into separate **partitions**. **Each Worker reads separate lines from Shared Drive.**

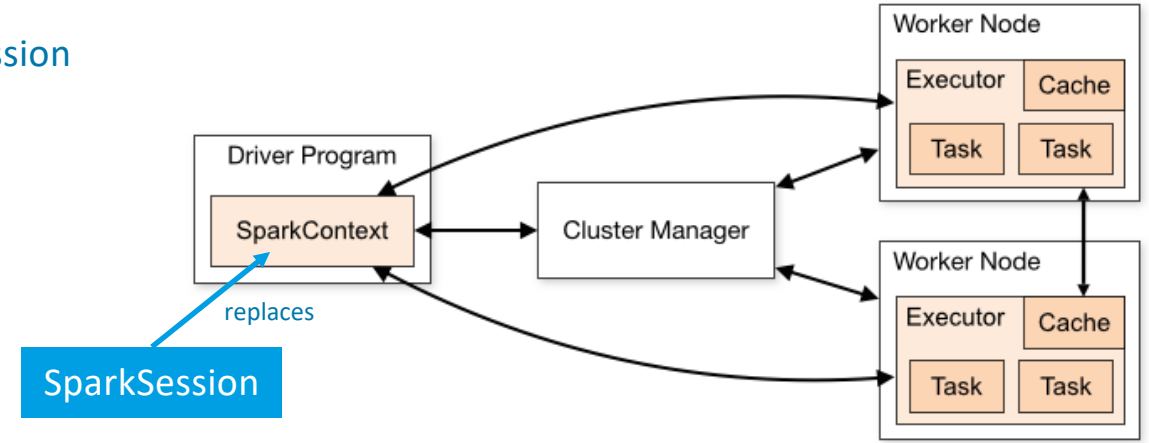
Uber Jar is Distributed

Data is Distributed across partitions – allows parallel processing

Spark Cluster Managers

As of Spark 2.0 SparkSession subsumes:

- SparkContext
- SQLContext
- HiveContext
- StreamingContext
- SparkConf

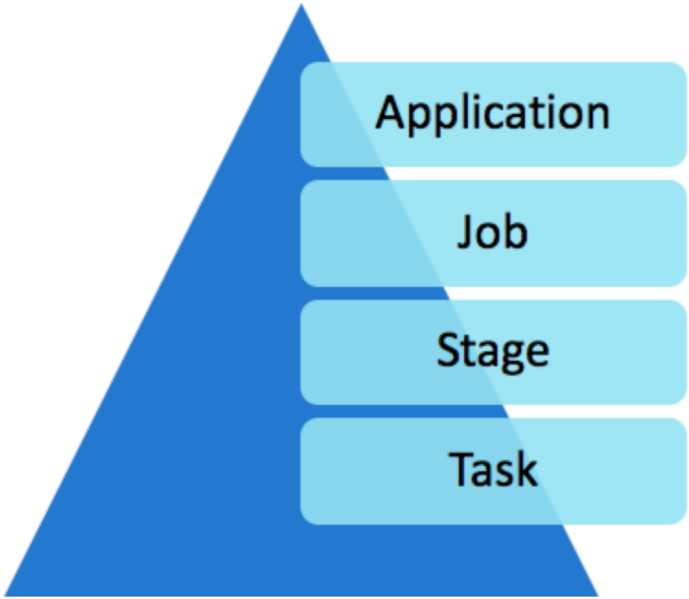


Cluster Manager Types:

- **Local –** has its own manager, but there is no cluster
- **Standalone** – a simple cluster manager included with Spark that makes it easy to set up a cluster.
- **Apache Mesos** – a general cluster manager that can also run Hadoop MapReduce and service applications. (Deprecated)
- **Hadoop YARN** – the resource manager in Hadoop 2.
- **Kubernetes** – an open-source system for automating deployment, scaling, and management of containerized applications.
- **AWS Products:** EMR, Glue – use one of the above managers.

Anatomy of Spark Job

- **Application:** When we submit the Spark code to a cluster it creates a Spark Application.
- **Job:** The Job is the top-level execution for any Spark application. A Job corresponds to an Action in a Spark application.
- **Stage:** Jobs will be divided into stages. The Transformations work in a lazy fashion and will not be executed until an Action is called. Actions might include one or many Transformations and the Transformations define the breakdown of jobs into stages, which corresponds to a shuffle dependency.
- **Task:** Stages will be further divided into various tasks. The task is the most granular unit in Spark applications. Each task represents a local computation on a particular node in the Spark Cluster

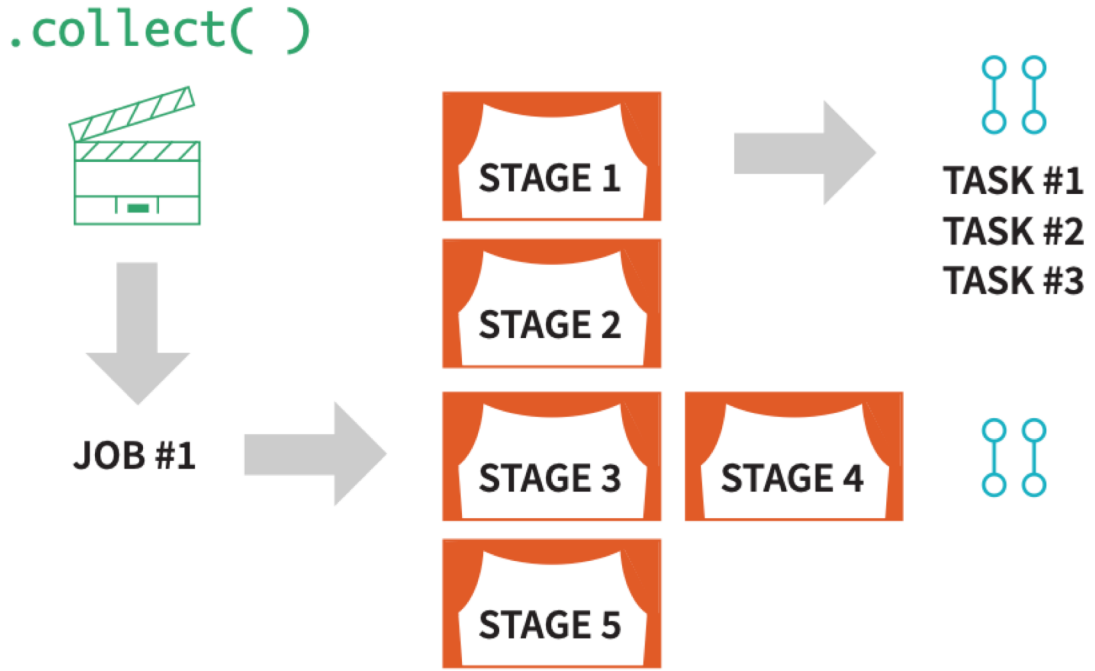


Spark Job and DAGs

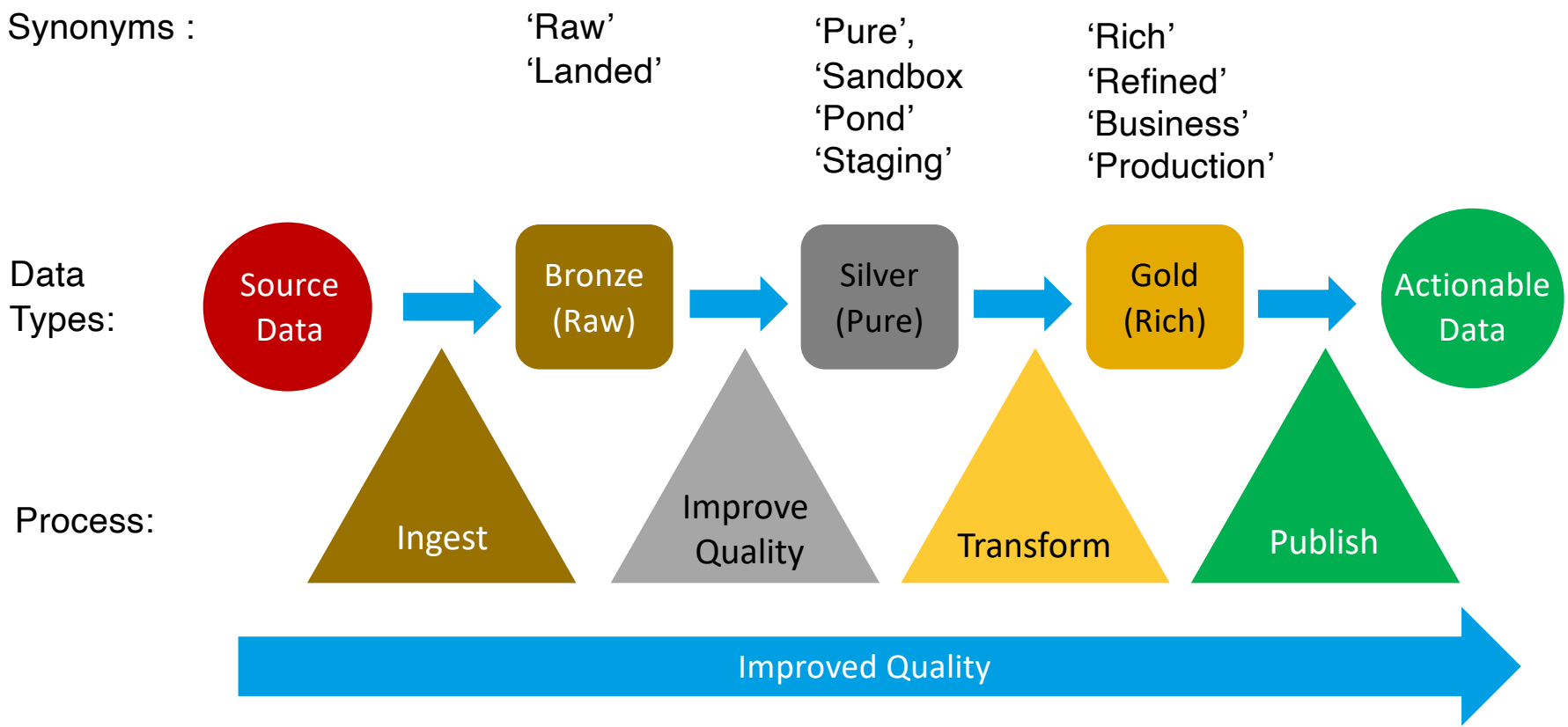
A Spark App in Java is not executed in sequence, or in Java, but just provides a set of instructions for a Job to be executed on Spark.

The Spark API is the point where these instructions are submitted to Spark.

1. A job gets decomposed into multiple stages.
2. The stages are broken down into tasks.
3. The tasks are shipped to workers to be executed.
4. These tasks can then be scheduled and run in parallel.



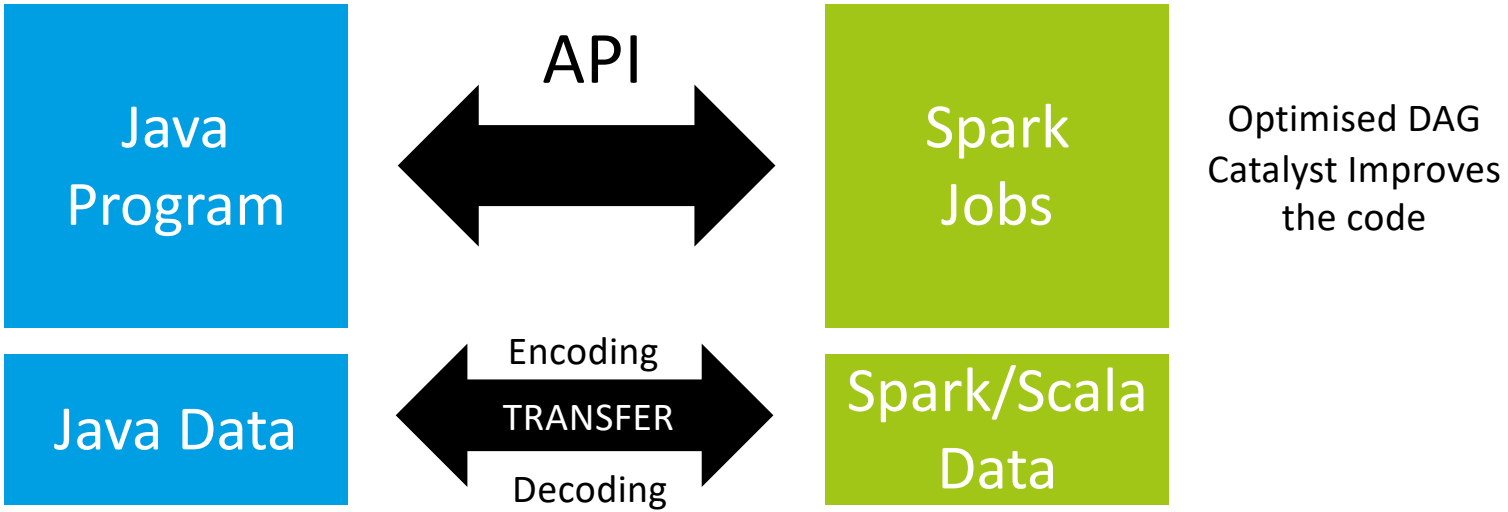
Goal is a Data Pipeline: Multi-Hop Architecture



WARNING !!!!!!

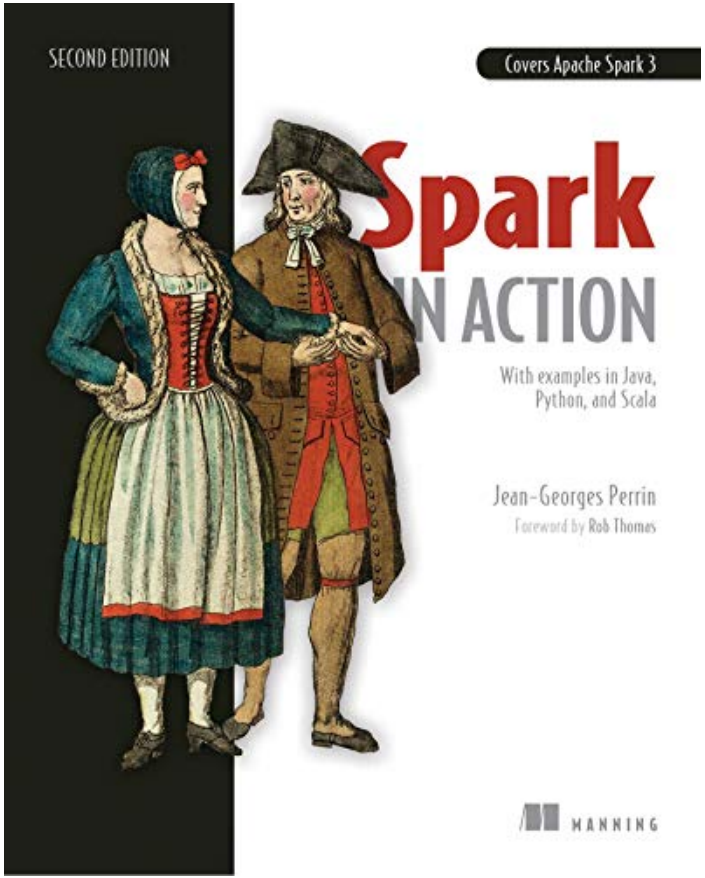
Your Java Code is NOT Spark

Your code is only a way to tell Spark the steps it will execute.
There is a huge penalty in data transfer.



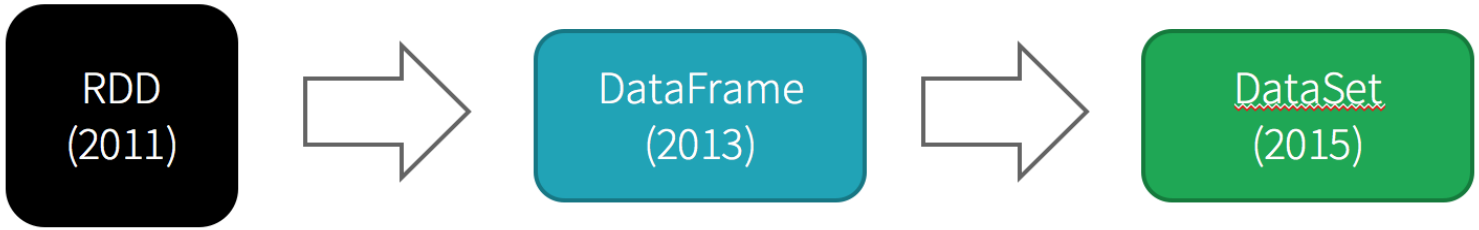
The Best Spark Java Book - only one on Java :-)

Spark in Action, Second Edition, teaches you to create end-to-end analytics applications. In this entirely new book, you'll learn from interesting Java-based examples, including a complete data pipeline for processing NASA satellite data. And you'll discover Java, Python, and Scala code samples hosted on GitHub that you can explore and adapt, plus appendixes that give you a cheat sheet for installing tools and understanding Spark-specific terms.



Spark Data

History of Spark APIs



Distribute collection of JVM objects

Functional Operators (map, filter, etc.)

Distribute collection of Row objects

Expression-based operations and UDFs

Logical plans and optimizer

Fast/efficient internal representations

Internally rows, externally JVM objects

Almost the "Best of both worlds": **type safe + fast**

But slower than DF
Not as good for interactive analysis, especially Python

databricks

RDD - Resilient Distributed Dataset

Been in Spark since the 1.0 release.

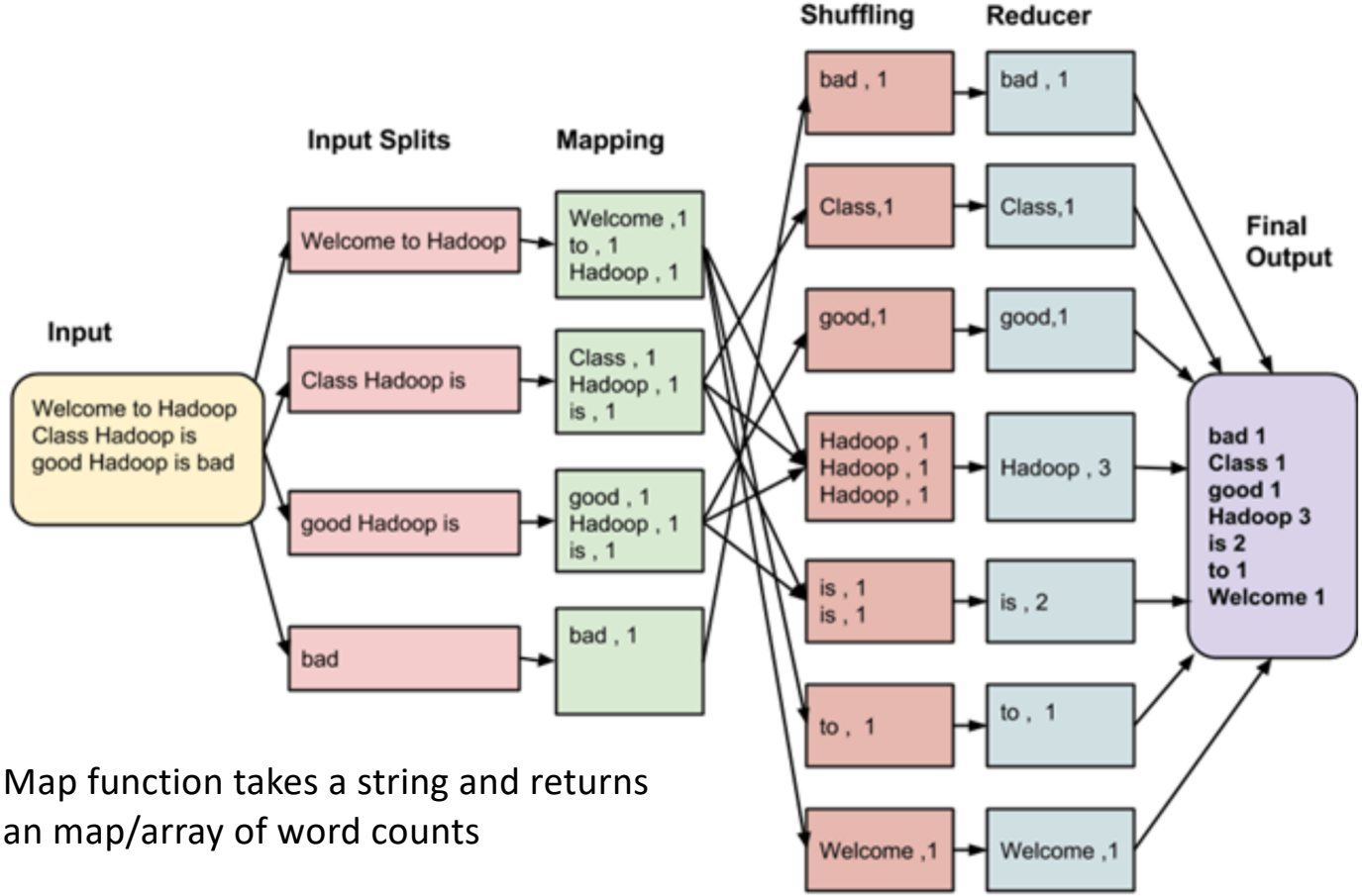
- **Distributed collection:** uses **MapReduce** operations for parallel, distributed algorithms on a cluster.
- **Partitioned**: composed of a collection of records which are partitioned allowing parallelism, no replication.
- **Immutable:** helps to achieve consistency in computations*
- **Fault tolerant:** In a case of we lose some partition of RDD , we can replay the transformations on that partition, rather than replicating across multiple nodes. Removing replication achieves faster computations.
- **Lazy evaluations:** All transformations in Spark are lazy, in that they do not compute their results right away. The transformations are only computed when an action requires a result to be returned to the driver program.
- **Functional transformations:** RDDs support two types of operations:
  - **transformations**, which create a new dataset from an existing one, and
  - **actions**, which return a value to the driver program after running a computation on the dataset.

* e.g. To delete data, don't - create a new data container without fields (transformation).

RDD Limitations

- **No Schemas**
- **No inbuilt optimization engine:** When working with structured data, RDDs cannot take advantages of Spark's advanced optimizers including catalyst optimizer and Tungsten execution engine.
  - Developers need to optimize each RDD based on its attributes.
- **Handling structured data:** Unlike Dataframe and datasets, RDDs don't infer the schema of the ingested data and requires the user to specify it.

MapReduce : Word Count Example



Map function takes a string and returns an map/array of word counts

DataFrames (Tables)

Like an RDD, an immutable distributed collection of data.

- ***Distributed collection of Row Object: data organized into named columns. It is conceptually equivalent to a table in a relational database, with SQL queries but with richer optimizations under the hood.***
- **Has Schema**: when calculating the execution plan, Spark, can use the schema and do substantially better computation optimizations. Can build schema from ingested data.
- **Data Processing:** Processing structured and unstructured data formats (Avro, CSV, elastic search, and Cassandra) and storage systems (HDFS, HIVE tables, MySQL, etc). It can read and write from all these..
- **Optimization using Catalyst optimizer:** It powers both SQL queries and the DataFrame API. Catalyst transformation framework has four phases,
    1. Analyzing a logical plan to resolve references
    2. Logical plan optimization – the DAG, explain() method to see plan
    3. Physical planning
    4. Code generation to compile parts of the query to Java bytecode.
- **Apache Hive Compatibility:** Using Spark SQL, you can run unmodified Hive queries on your existing Hive warehouses. It reuses Hive frontend and MetaStore and gives you full compatibility with existing Hive data, queries, and UDFs.
- **Tungsten project:** Tungsten provides a physical execution backend which explicitly manages memory and dynamically generates bytecode for expression evaluation, and provides cache-aware computation. Massive performance improvement.

DataFrame Limitations

- **Not Typed:** Dataframe API does not support compile time safety which limits you from manipulating data when the structure is not know.
  - It is a Dataset on generic type <Row>
  - Code may be OK during compile time, however, you may get a Runtime exception on execution.
- **Cannot operate on domain Object (lost domain object):**
  - Once you have transformed a domain object into Dataframe, you cannot regenerate it.

DataSet

Starting in Spark 2.0, Dataset takes on two distinct APIs characteristics: a ***strongly-typed*** API and an ***untyped*** AP

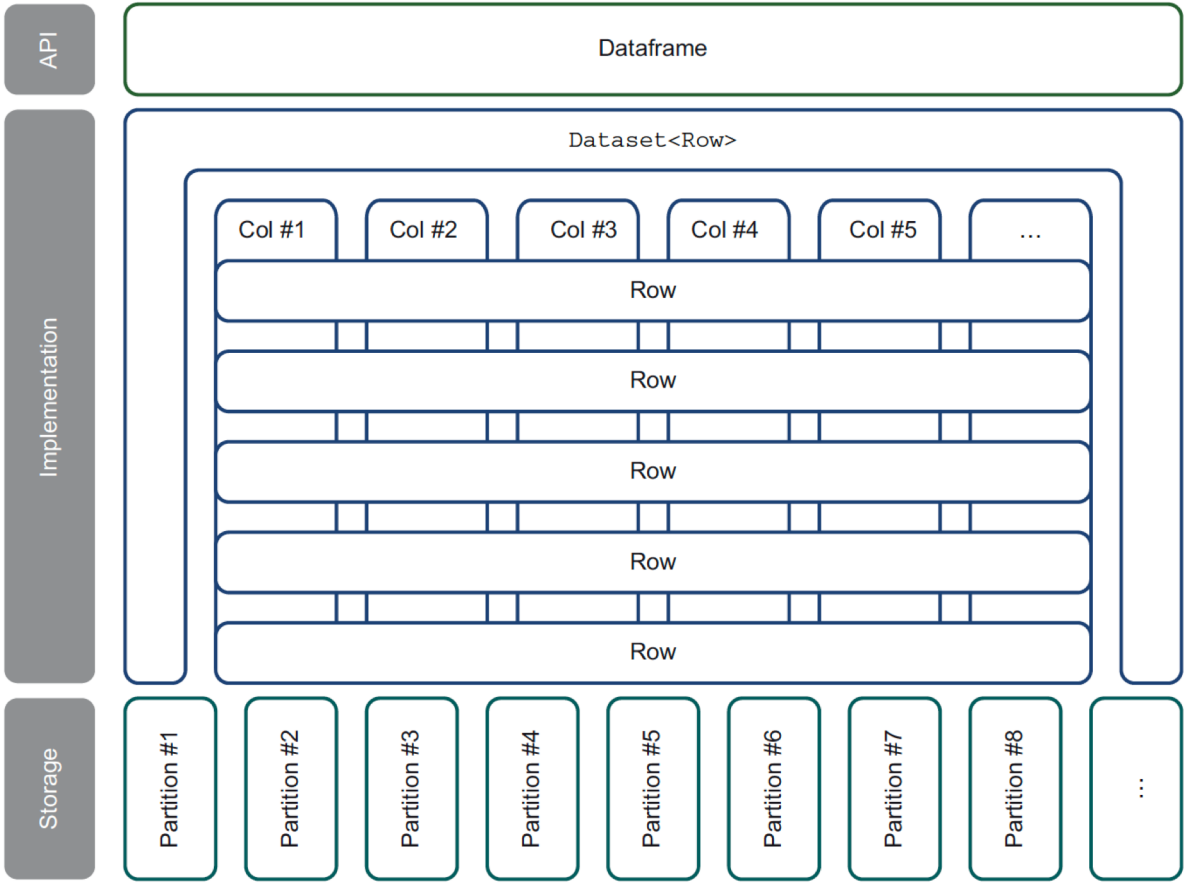
Dataset<Row>, where a *Row* is a generic ***untyped*** JVM object = **DataFrame ***

Dataset<T>, where T is a typed Java Object

- **Provides best of both RDD and DataFrame:**
  - RDD- functional programming, type safety,
  - DataFrame - Relational model (Tables), SQL, Query optimisation, Tungsten execution, sorting and shuffling
- **Encoders:** With the use of Encoders, it is easy to convert any JVM object into a Dataset, allowing users to work with both structured and unstructured data unlike DataFrame.
- **Type Safety:** Datasets API provides compile time safety which was not available in DataFrames.

* Scala has explicit Dataframe type but not java, <Row> is explicit type in all

DataFrame, DataSet, Partitions



Rows are distributed to different partitions, not replicated.

Allows parallel work. For Map-Reduce to work requires independence of 'r**ows'.**

Each Worker machines is allocated partitions. E.g.

- 8 Partitions
- 4 Machines each gets 2 partitions,
- 50% of work in parallel.
- Each partition needs a Thread.

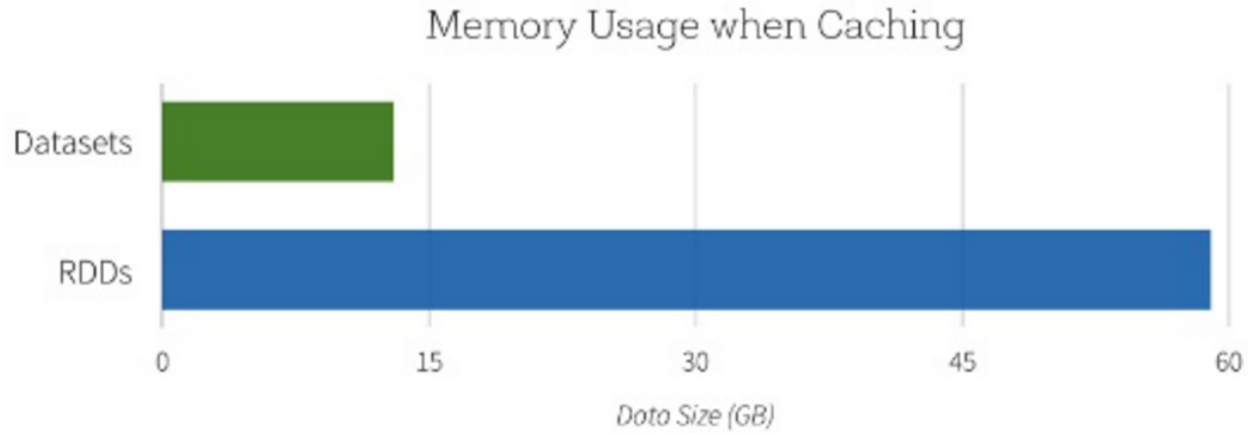
Encoders

An encoder of a particular type encodes either an Java object (of the encoder type) or a data record (in conformance with the **data schema** of the encoder type) into the binary format backed by raw memory and vice-versa.

- Encoders are part of Spark's Tungsten framework. Being backed by the raw memory, updates or querying of relevant information from the encoded binary text is done via **Java Unsafe APIs***.
- Spark provides a **generic Encoder Interface** and a generic Encoder implementing the interface called as ExpressionEncoder . This encoder encodes and decodes (could be understood as serialization and deserialization also) a JVM Object (of type T) via expressions.
- The EncodersFactory provides storage efficient ExpressionEncoders for types, such as Java boxed primitive types (Integer, Long, Double, Short, Float, Byte, etc.), String, Date, Timestamp, Java bean, etc.
  - The factory provides generic Java/Kryo serialization based ExpressionEncoder which can be used for any type so that the Encoders can be created for custom types that are not covered by storage efficient ExpressionEncoder.

*Java APIs to implement operations which are otherwise only available with native C or assembly code.

Space Efficiency



- Spark as a compiler understands your Dataset type JVM object, it maps your type-specific JVM object to Tungsten's internal memory representation using Encoders.
- As a result, Tungsten Encoders can efficiently serialize/deserialize JVM objects as well as generate compact bytecode that can execute at superior speeds.

https://databricks.com/blog/2016/05/23/apache-spark-as-a-compiler-joining-a-billion-rows-per-second-on-a-laptop.html

Encoder Efficiency

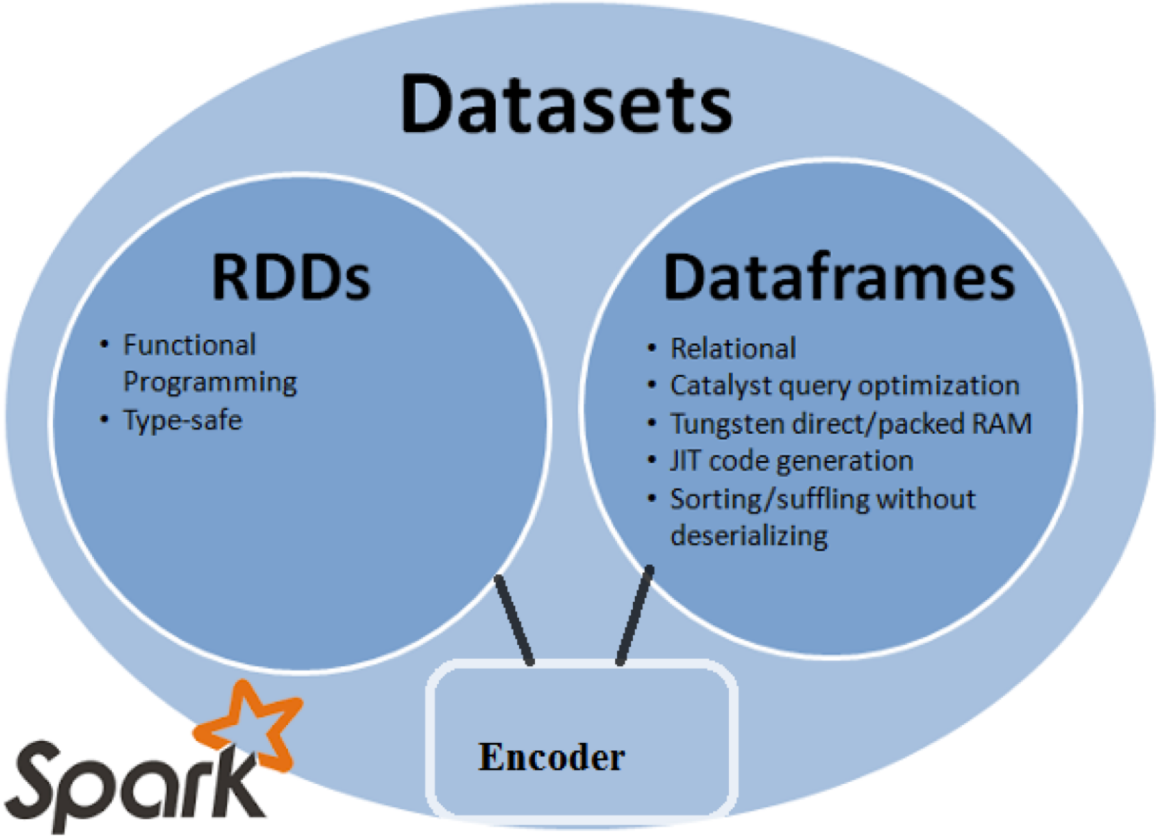
Serialized based ExpressionEncoders:

- The whole object is serialized based on either Java or Kryo serialization and the serialized byte string is **kept as the only single field in the encoded binary forma**t.
- Therefore these lack storage efficiency and one cannot directly query particular fields of the object directly from the encoded binary format.

Java bean based ExpressionEncoders:

- the bean object is mapped to the binary format by just keeping its fields in the binary format thereby providing twin benefits of storage efficiency and faster querying of individual fields.
- Therefore, for Datatsets composed of complex datatypes, one **should always construct datatype as Java bean consisting of fields for which Encoders factory supports** non-serialized based ExpressionEncoders.

Datasets cover Dataframes and RDD



Summary

Feature	RDD	DataFrame	Dataset
Immutable	Yes	Yes	Yes
Fault Tolerant	Yes	Yes	Yes
Type-Safe	Yes	No	Yes
Schema	No	Yes	Yes
Execution Optimization	No	Yes	Yes
Optimizer Engine	N/A	Catalyst Engine	Catalyst Engine
API Level for manipulating distributed collection of data	Low	High	High
language Support	Java, Scala, Pyt	Java, Scala, Python, R	Java, Scala

Type Safety

	SQL	DataFrames	Datasets
Syntax Errors	Runtime	Compile Time	Compile Time
Analysis Errors	Runtime	Runtime	Compile Time

When should I use DataFrames or Datasets?

Use DataFrame or Dataset:

- If you want rich semantics, high-level abstractions, and domain specific APIs,
- If your processing demands high-level expressions, filters, maps, aggregation, averages, sum, SQL queries, columnar access and use of lambda functions on semi-structured data.
- If you want unification and simplification of APIs across Spark Libraries

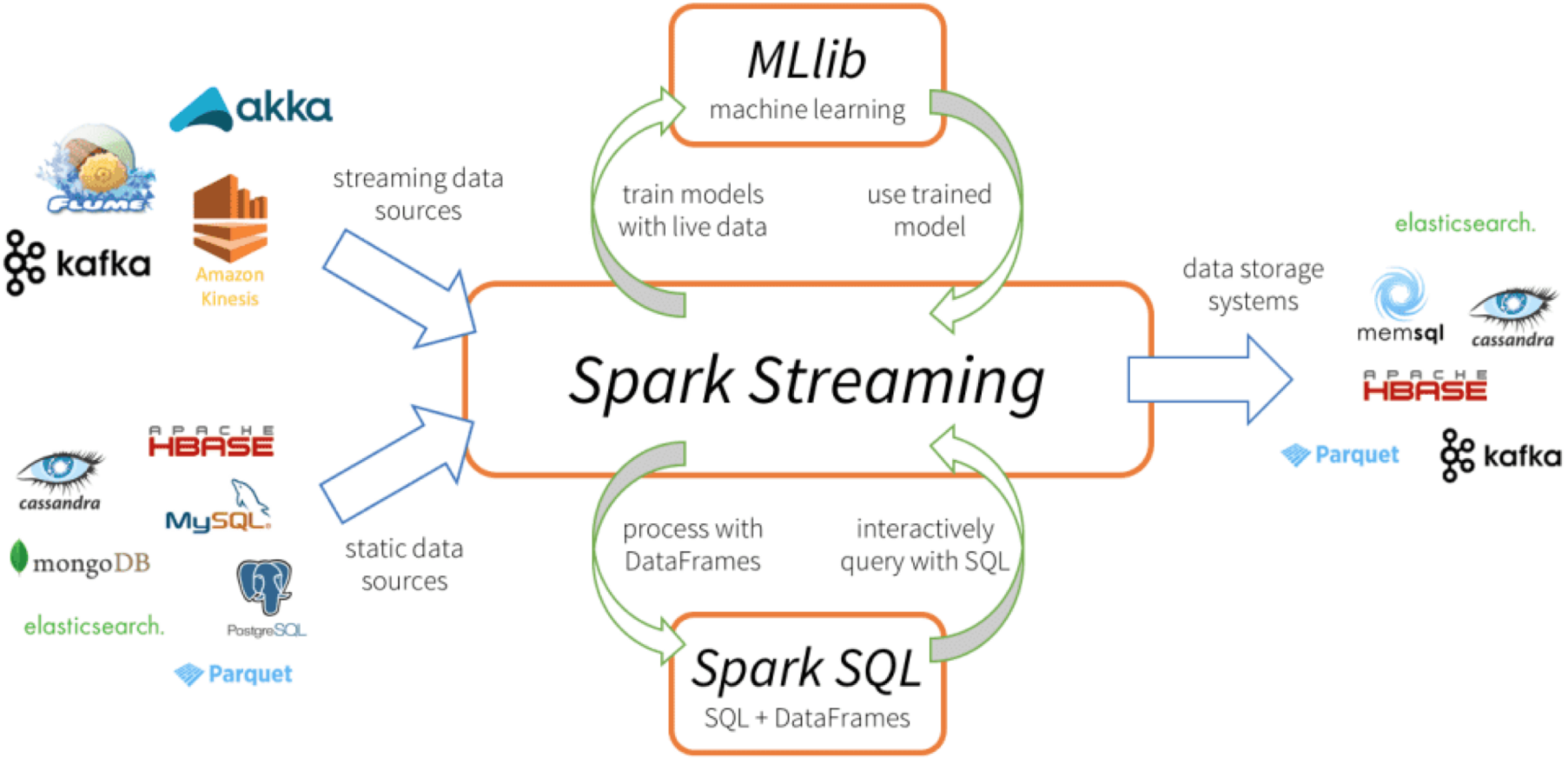
Use Dataset.

- If you want higher degree of type-safety at compile time, want typed JVM objects, take advantage of Catalyst optimization, and benefit from Tungsten's efficient code generation,

Use DataFrames:

- If you are a R user, (R has Dataframes itself)
- If you are a Python user, and resort back to RDDs if you need more control.
- Note that you can always seamlessly interoperate or convert from DataFrame and/or Dataset to an RDD, by simple method call .rdd

Spark SQL + DataFrames, Streaming, ML has its own data types



Examples

Playing Golf Data used across Java Examples

N	OUTLOOK	TEMPERATURE	HUMIDITY	WINDY	PLAY GOLF
1	sunny	85	85	false	Play
2	sunny	80	90	true	Don't Play
3	overcast	83	88	false	Play
4	rain	70	96	false	Play
5	rain	68	80	false	Play
6	rain	65	70	true	Don't Play
7	overcast	64	65	true	Play
8	sunny	72	95	false	Don't Play
9	sunny	69	70	false	Play
10	sunny	85	85	false	Play
11	rain	75	80	false	Play
12	sunny	75	70	true	Play
13	sunny	85	85	false	Play
14	overcast	72	90	true	Play
15	overcast	81	75	false	Play
16	sunny	85	85	false	Don't Play
17	rain	71	96	true	Don't Play

A famous Machine Learning Date Set

Notes – Common Dataset methods we will use

Dataset.**show**(int numRows, int truncate, boolean vertical)

Dataset. **schema**()

Dataset.**printSchema()**

Dataset. **count**()

Dataset. **drop**(String colName)

Dataset.**withColumn**(String colName, **Column** col) - Returns a new Dataset by adding a column or replacing the existing column that has the same name

Column cast(DataType to)

Spark SQL

Dataset API and SQL

Spark SQL supports two "modes" to write structured queries: Dataset API and SQL.

- SQL Mode is used to express structured queries using SQL statements using SparkSession.sqloperator, expr standard function and spark-sql command-line tool.
- Some structured queries can be expressed much easier using Dataset API, but there are some that are only possible in SQL. In other words, you may find mixing Dataset API and SQL modes challenging yet rewarding.
- What is important, and one of the reasons why Spark SQL has been so successful, is that there is no performance difference between the modes. Whatever mode you use to write your structured queries, they all end up as a tree of Catalyst relational data structures.

Spark SQL – Immutability with Views on Datasets

Datasets in Spark are immutable:

- You cannot delete data – no SQL DELETE, or API to delete.
- To remove data either use a filter on your view / create a new data set based on a query:

```
Dataset<Row> newResults =
      spark.sql( "select * FROM TempView where numbParts < 50");
```

```
Dataset<Row> niceDays =
      spark.sql(
            "SELECT * FROM dataview
                  WHERE outlook = 'sunny' and temperature > 70");
```

outlook	temperature	humidity	windy	play
sunny	85	85	FALSE	no
sunny	80	90	TRUE	no
overcast	83	86	FALSE	yes
rainy	70	96	FALSE	yes
rainy	68	80	FALSE	yes
rainy	65	70	TRUE	no
overcast	64	65	TRUE	yes
sunny	72	95	FALSE	no
sunny	69	70	FALSE	yes
rainy	75	80	FALSE	yes
sunny	75	70	TRUE	yes
overcast	72	90	TRUE	yes
overcast	81	75	FALSE	yes
rainy	71	91	TRUE	no

outlook	temperature	humidity	windy	play
sunny	85	85	FALSE	no
sunny	80	90	TRUE	no
sunny	72	95	FALSE	no
sunny	75	70	TRUE	yes

- spark.sql("INSERT INTO dataview values('sunny', '100', '85' ,'FALSE', 'no')");
  - will fail, you cannot change the DataSeta underlying the view
  - You can only insert into a TABLE (not a view)
  - The dataview was read from a file, into a DATASET, can't be overwritten

Spark SQL – Table Statements*

Spark 3.0 introduces two experimental options to comply with the SQL standard: spark.sql.ansi.enabled and spark.sql.storeAssignmentPolicy. When spark.sql.ansi.enabled is set to true, Spark SQL uses an ANSI compliant dialect instead of being Hive compliant.

Data Definition Statements are used to create or modify the structure of database objects in a database, e.g:

- CREATE DATABASE
- CREATE FUNCTION
- CREATE TABLE
- CREATE VIEW
- DROP DATABASE
- DROP FUNCTION
- DROP TABLE
- DROP VIEW
- REPAIR TABLE
- TRUNCATE TABLE
- USE DATABASE

Data Manipulation Statements are used to add, change, or delete data, e.g:

- INSERT INTO
- INSERT OVERWRITE
- INSERT OVERWRITE DIRECTORY
- INSERT OVERWRITE DIRECTORY with Hive format
- LOAD
- NO DELETE !!!!!
- NO UPDATE !!!!!!

Added by DeltaLake Later slides

Data Retrieval

- SELECT Statements
- EXPLAIN

*These are for TABLES, NOT VIEWS !!!!

Data Import ‘Dialects’

A dialect is a small software component, often implemented in a single class, that bridges Apache Spark and the database;

Provided dialects:

- IBM Db2
- Apache Derby
- MySQL
- Microsoft SQL Server Oracle
- PostgreSQL
- Teradata Database
- etc

It is easy to make your own dialect for other DBMS.

Caching Data In Memory

Spark SQL can cache tables using an in-memory columnar format by calling:

spark.catalog.cacheTable("tableName") or
dataFrame.cache().

- Then Spark SQL will scan only required columns and will automatically tune compression to minimize memory usage and GC pressure.

- You can call spark.catalog.uncacheTable("tableName") or dataFrame.unpersist() to remove the table from memory.

Data Formats

Comparison of the Big 3 Big Data Formats

	Avro	Parquet	ORC
Schema Evolution Support			
Compression			
Splitability			
Most Compatible Platforms	Kafka, Druid	Impala, Arrow Drill, Spark	Hive, Presto
Row or Column	Row	Column	Column
Read or Write	Write	Read	Read

Parquet

Apache Parquet is a COLUMNAR format that is supported by many other data processing systems. Where as for comparison CSV is a ROW based format. Spark SQL provides support for both reading and writing Parquet files that automatically preserves the schema of the original data. When reading Parquet files, all columns are automatically converted to be nullable for compatibility reasons.

- Parquet is an open source file format available to any project in the Hadoop ecosystem. It is designed for efficient as well as performant flat columnar storage format of data compared to row based formats like CSV or TSV files.
- Parquet provides considerable data compression advantages, which reduces cost considerably when stored on S3, over CSV formats.
- Parquet uses the record shredding and assembly algorithm which is superior to simple flattening of nested namespaces. Parquet is optimized to work with complex data in bulk and features different ways for efficient data compression and encoding types.
- Like Protocol Buffer, Avro, and Thrift, Parquet also supports schema evolution. Users can start with a simple schema, and gradually add more columns to the schema as needed. In this way, users may end up with multiple Parquet files with different but mutually compatible schemas. The Parquet data source is now able to automatically detect this case and merge schemas of all these files.
- Parquet files are in a compressed directory, with multiple binary files. Conceptually each column of data is in a separate file, allowing fast loading compared to CSV that needs to load a whole row of data, with all columns, to get one value.

LIBSVMformat for use with Spark MLib

- *LIBSVM* is a compact text format for encoding data (usually representing training data sets)
- Widely used in Machine Learning (MLib) to represent sparse feature vectors
- A file in *LIBSVM* format is shaped as a matrix in which each line is a space-delimited record that represents a labelled sparse feature (attribute / property) vector
- The layout is as follows:

  class_label index1:value1 index2:value2 ...

  - where the numeric indices represent features; values are separated from indices via a colon (':')
  - MLib expects you to start class labelling from 0 (Classes are Integers not categories in English)
  - Feature indices are one-based in ascending order (1,2,3, etc.); where a feature is not present in the data record, it is omitted from the record
  - Allows the columns to be in any order, in a row, as they are specified by Index Number, not position.
  - It is a sparse format so 0 (Zero) values do not need to be stored.
- Big issue when compared to other standard ML formats is that Symbolic data is not allowed (has to be numeric) which defeats the purpose of Machine Learning in many cases, e.g. algorithms designed for Symbolic Explanation and processing. This also adds complexities to the code, e.g. having to specify the number of categories (now represented as numbers not Symbols/Enums).

Reading JSON – only if you have too

Spark SQL can automatically infer the schema of a JSON dataset and load it as a Dataset[Row]. This conversion can be done using SparkSession.read.json() on either a Dataset[String], or a JSON file.

2 Formats supported:

1. JSONL
2. MultiLine JSON - set the ”multiline” option to true.

Using JSONL format may be inconvenient but JSON is simply not designed to be processed in parallel in distributed systems:

- It provides no schema and without making some very specific assumptions about its formatting and shape it is almost impossible to correctly identify top level documents.
- Arguably this is the worst possible format to imagine to use in systems like Apache Spark. It is also quite tricky and typically impractical to write valid JSON in distributed systems.

JSONL (Lines) Format

JSONL is a convenient format for storing structured data that may be processed one record at a time, as the JSON records are contained in one row, with “\n” as record separator.

A CSV like JSONL representation, (or a table like representation) Each line is a JSON list.

["Name", "Session", "Score", "Completed"]
["Gilbert", "2013", 24, true]
["Alexa", "2013", 29, true]
["May", "2012B", 14, false]
["Deloise", "2012A", 19, true]

A Nested JSONL structure, of winning hands, using Key:Value pairs:

{"name": "Gilbert", "wins": [["straight", "7 "], ["one pair", "10 "]]}
{"name": "Alexa", "wins": [["two pair", "4 "], ["two pair", "9 "]]}
{"name": "May", "wins": []}
{"name": "Deloise", "wins": [["three of a kind", "5 "]]}

To view nested JSONL as normal JSON:
grep pair winning_hands.jsonl | jq .

Explode Function to (Flatten / De-normalise) JSON

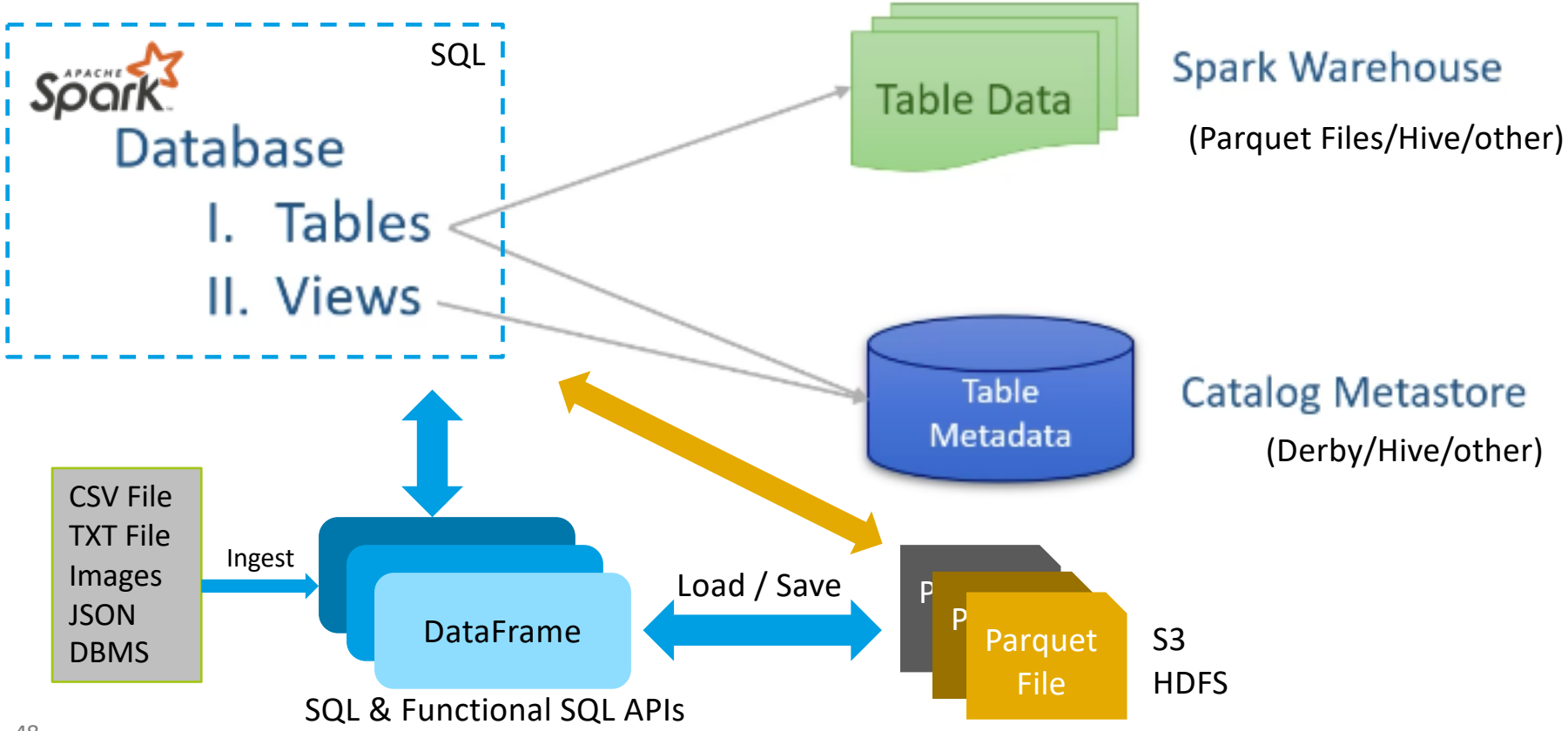
```
[
  {
  "shipmentId": 458922,
  "date": "2019-10-05",
  "supplier": {
    "name": "Manning Publications",
    "city": "Shelter Island",
    "state": "New York",
    "country": "USA"
  },
  "customer": {
    "name": "Michael Perrin",
    "city": "Chapel Hill",
    "state": "North Carolina",
    "country": "USA"
  },
  "books": [
    {
      "id" : "001",
      "title": "A very good book ",
      "qty": 2
    },
    {
      "id" : "002",
      "title": "Another nice book",
      "qty": 25
    },
    {
      "id" : "003",
      "title": "Another Nice Book V2",
      "qty": 1
    }
  ]
}
,
{
  "shipmentId": 12345,
  "date": "2019-10-10",
  "supplier": {
```

ID	Shipment ID	Date	Supplier Name	Supplier City	Supplier State	Supplier Country
1	450922	2019/10/05	Manning	Shelter	New York	USA
2	450922	2019/10/05	Manning	Shelter	New York	USA
3	450922	2019/10/05	Manning	Shelter	New York	USA
4	12345	…	…	….	….	…

Customer Name	Customer City	Customer State	Customer Country	Book ID	Book Title	Book Qnty
Michael Perrin	Chapel Hill	North Carolina	USA	001	A very Good Book	2
Michael Perrin	Chapel Hill	North Carolina	USA	002	Another Nice Book	25
Michael Perrin	Chapel Hill	North Carolina	USA	003	Another Nice Book V2	1
…..	…..	….	….	….	….	…

Databases, Tables and Views

Spark Database: Tables and Views

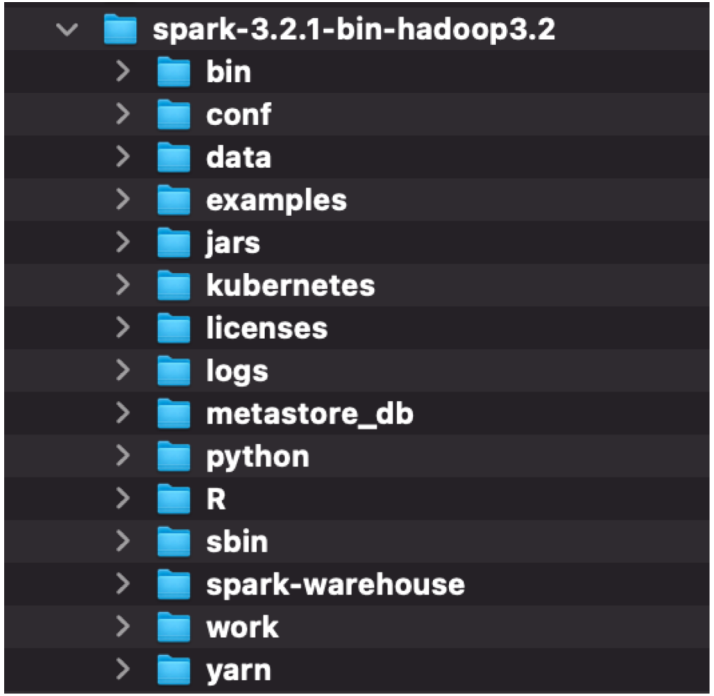


metastore_db VS Spark-warehouse

To persist tables you need both:

1. **HIVE Metastore** – a catalog database created by Spark to hold the Table metadata, where and what data is persisted:
   - You do not need a HIVE implementation
   - By default it is an 'in-memory' DB only.
   - When POM dependency included, and HIVE configured, Spark by default uses a local Derby DB in the **metastore_db** directory.
   - You can point to another external data base implementation, e.g. a Apach HIVE implementation

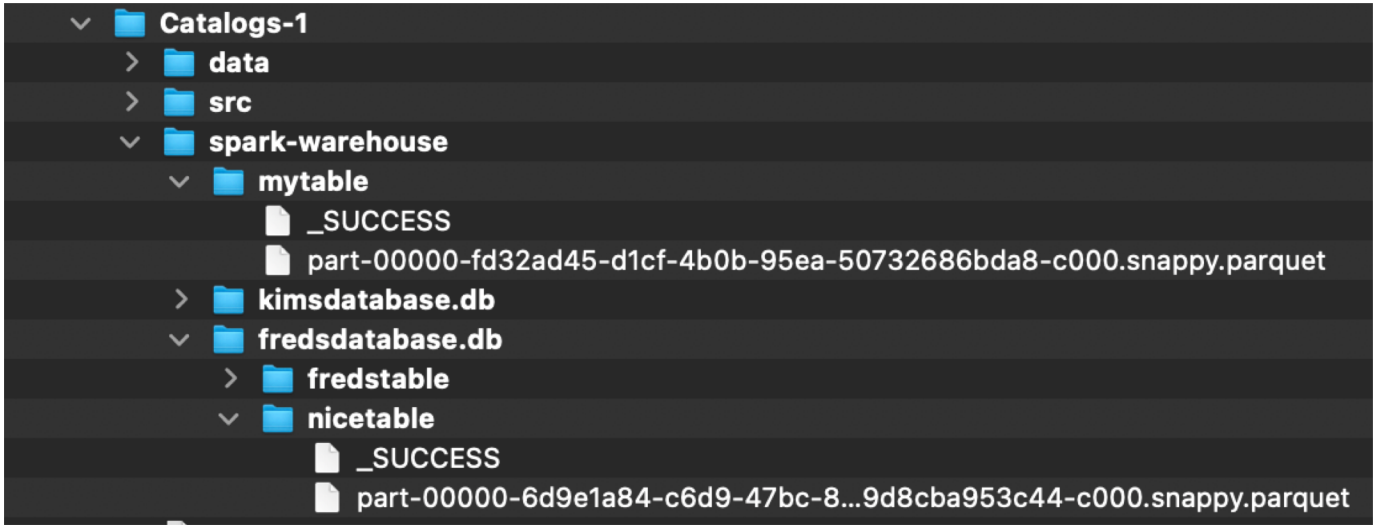
2. **Spark_warehouse** – directory created by Spark to hold tables and global views data.
   - in the **spark_warehouse** directory



Local Mode Spark Warehouse

When not configured by the hive-site.xml, the context automatically creates a DB in the directory configured by **spark.sql.warehouse.dir.**

- defaults to the directory **spark-warehouse** in the current directory of the Spark application
- Global views, and Created Databases are stored here



metastore_db VS Spark-warehouse

By default when Hive is not included in the build, the Metastore is in memory only.

- As a result, it will only be present while a driver is running, e.g. in your local app.
- If you have two sessions in you Spark app then you can share data via this metastore, as they share driver memory
- However, when the app stops, you no longer have access to the in memory metastore.
- To have a persistence metastore:
  - **Add HIVE to POM in a local app (not needed in cluster, as it uses cluster for Hive) :**

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-hive_2.12</artifactId>
    <version>3.1.2</version>
</dependency>
```

  - **Initialise SparkSession, add:**

```
.enableHiveSupport()
```

SQL works on Views and Tables

When using SparkSQL you need to create a View on the data, these can be Local or Global across the session in the Spark application.

- **Local Views** are temporary and are session scoped. They will disappear when the session ends.
- **Global Temporary Views** exist across sessions in the application, until the Spark application terminates. You may have multiple sessions in your application.

Tables can be managed or Unmanaged:

- **Managed (or Internal) Tables:** for these tables, Spark manages both the data and the metadata. In particular, data is usually saved in the Spark SQL warehouse directory - that is the *default* for managed tables - whereas metadata is saved in a meta-store of relational entities (including *databases, tables, temporary views*) and can be accessed through an interface known as the **"catalog".**
- **Unmanaged (or External) Tables:** for these tables, Spark only manages the metadata, but requires you to specify the exact location where you wish to save the table or, alternatively, the source directory from which data will be pulled to create a table. These tables exist after a Spark Table delete (not so for above).

The combinations of these are discussed in next slides:

Spark documentation Hive Tables

When working with Hive, one must instantiate **SparkSession** with Hive support, including connectivity to a persistent Hive metastore, support for Hive serdes, and Hive user-defined functions. Users who do not have an existing Hive deployment can still enable Hive support. When not configured by the **hive-site.xml**, the context automatically creates **metastore_db** in the current directory and creates a directory configured by **spark.sql.warehouse.dir**, which defaults to the directory **spark-warehouse** in the current directory that the Spark application is started. Note that the **hive.metastore.warehouse.dir** property in **hive-site.xml** is deprecated since Spark 2.0.0. Instead, use **spark.sql.warehouse.dir** to specify the default location of database in warehouse. You may need to grant write privilege to the user who starts the Spark application.

Example Table - SQL CREATE DB and TABLE

```
spark.sql("CREATE database IF NOT EXISTS studentdatabase");
spark.sql("USE studentdatabase");
spark.sql("CREATE TABLE IF NOT EXISTS students (name VARCHAR(64), address VARCHAR(64)) USING PARQUET
PARTITIONED BY (student_id INT)");
spark.sql("INSERT INTO students VALUES('Amy Smith', '123 Park Ave, San Jose', 111111)");
```

CREATE TABLE USING <DATA SOURCE>

A DATA SOURCE can be:

- CSV
- JSON
- Parquet
- JDBC
- Hive (an Apache data warehouse)
- etc

Tables

1. Global Managed Table

A managed table is a Spark SQL table for which Spark manages both the data and the metadata. A global managed table is available across all clusters. Creates a Hive Metastore (Hive actual is not required).

- The data is persistent. When you drop the table *both* data and metadata gets dropped.

```
dataframe.write.saveAsTable("my_table")
```

2. Global Unmanaged/External Table

Spark manages the metadata, while you control the data location. As soon as you add ‘path’ option in dataframe writer it will be treated as global external/unmanaged table. When you drop table only metadata gets dropped. A global unmanaged/external table is available across all clusters.

```
dataframe.write.option('path', "<your-storage-path>").saveAsTable("my_table")
```

3. Local Table = Temporary Table = Temporary View

Spark session scoped. A local table is not accessible from other clusters and is not registered in the metastore.

```
dataframe.createOrReplaceTempView()
```

Tables Continued

4. Global Temporary View

Spark application scoped, global temporary views are tied to a system preserved temporary database global_temp. This view can be shared across different spark sessions (or if using databricks notebooks, then shared across notebooks).

```
dataframe.createOrReplaceGlobalTempView("my_global_view")
```

can be accessed as,

```
spark.read.table("global_temp.my_global_view")
```

5. Global Permanent View

Persist a dataframe as permanent view. The view definition is recorded in the underlying metastore. You can only create permanent view on global managed table or global unmanaged table. Not allowed to create a permanent view on top of any temporary views or dataframe. Note: Permanent views are only available in SQL API — not available in dataframe API

```
spark.sql("CREATE VIEW permanent_view AS SELECT * FROM table")
```

There *isn't* a function like dataframe.createOrReplacePermanentView()

3 ways to create a Table

```
df.write.mode("overwrite").saveAsTable("newTable_1")
```

```
spark.sql("CREATE TABLE IF NOT EXISTS newTable_2 AS SELECT * FROM Local_View"
```

```
spark.sql("CREATE TABLE newTable_3 (
order_id INT,
order_date DATE,
item_type STRING,
sales_channel STRING,
units_sold FLOAT,
unit_price FLOAT,
total_revenue FLOAT ");

spark.sql("INSERT INTO newTable_3
SELECT * FROM Local_View");
```

Note: Local_View is on DataFrame DF

Transformations and Actions

2 Types of Spark Operations

Transformations:

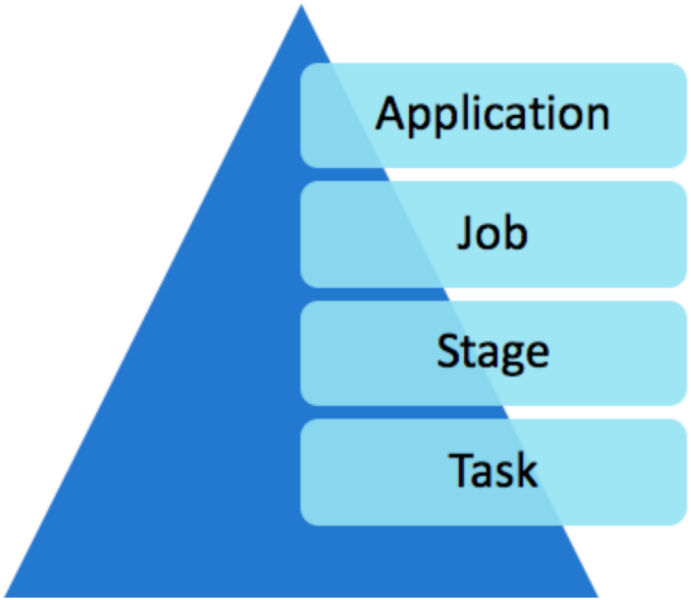
- These operations work in a lazy fashion.
- When you apply a transformation on a Dataset, or RDD it will not be evaluated immediately but will only be stored in a DAG (Directed Acyclic Graph) and will be evaluated at some later point of time after an action is executed.
- Some common transformations are map, filter, flatMap, groupByKey, reduceByKey..

Actions:

- These operations will be executed immediately.
- While Transformations return another Dataset, or RDD, Actions return language native data structures.
- Some common actions are first, last, count, collect.

Anatomy of Spark Job

- **Application:** When we submit the Spark code to a cluster it creates a Spark Application.
- **Job:** The Job is the top-level execution for any Spark application. A Job corresponds to an Action in a Spark application.
- **Stage:** Jobs will be divided into stages. ***The Transformations work in a lazy fashion and will not be executed until an Action is called***. Actions might include one or many Transformations and the Transformations define the breakdown of jobs into stages, which corresponds to a shuffle dependency.
- **Task:** Stages will be further divided into various tasks. The task is the most granular unit in Spark applications. Each task represents a local computation on a particular node in the Spark Cluster



RDD Operations

 = easy  = medium

Essential Core & Intermediate Spark Operations

TRANSFORMATIONS

General

- map
- filter
- flatMap
- mapPartitions
- mapPartitionsWithIndex
- groupBy
- sortBy

Math / Statistical

- sample
- randomSplit

Set Theory / Relational

- union
- intersection
- subtract
- distinct
- cartesian
- zip

Data Structure / I/O

- keyBy
- zipWithIndex
- zipWithUniqueID
- zipPartitions
- coalesce
- repartition
- repartitionAndSortWithinPartitions
- pipe

ACTIONS

- reduce
- collect
- aggregate
- fold
- first
- take
- forEach
- top
- treeAggregate
- treeReduce
- forEachPartition
- collectAsMap

- count
- takeSample
- max
- min
- sum
- histogram
- mean
- variance
- stdev
- sampleVariance
- countApprox
- countApproxDistinct

- takeOrdered

- saveAsTextFile
- saveAsSequenceFile
- saveAsObjectFile
- saveAsHadoopDataset
- saveAsHadoopFile
- saveAsNewAPIHadoopDataset
- saveAsNewAPIHadoopFile

Transformation Functions

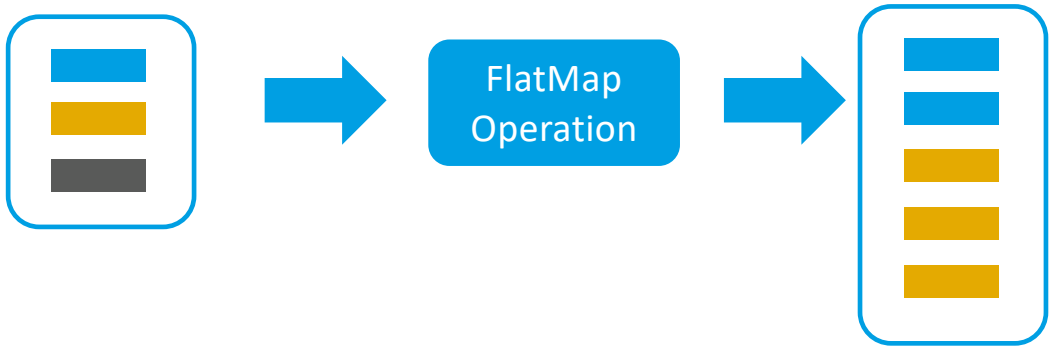
Function	Description
map (func, encoding)	Returns a new distributed dataset formed by passing each element of the source through a function func with an encoder function
filter (func)	Returns a new dataset formed by selecting those elements of the source on which func returns true. *Sparks most used function*
flatMap (func, encoding)	Similar to map, but each input item can be mapped to 0 or more output items.
mapPartitions (func, encoding)	Similar to map, but runs separately on each partition (block) of the RDD, so func must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.
sample (withReplacement, fraction, seed)	Samples a fraction of the data, with or without replacement, using a given random number generator seed.
union (otherDataset)	Returns a new dataset that contains the union of the elements in the source dataset and the argument. The number and type of columns must be matching.
intersection (otherDataset)	Returns a new RDD that contains the intersection of elements in the source dataset and the argument.

Transformation Functions

Function	Description
distinct ()	Returns a new dataset that contains the distinct elements of the source dataset.
groupByKey (func, encoding)	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs.
reduceByKey (func, [numPartitions])	When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs, where the values for each key are aggregated using the given reduce function func, which must be of type (V, V) => V.
sortByKey ([ascending], [numPartitions])	When called on a dataset of (K, V) pairs, where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the Boolean ascending argument.
join (otherDataset, [numPartitions])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W))pairs with all pairs of elements for each key. Joins can be inner, cross, outer (full, full_outer), left_outer (left), right_outer (right), left_semi, or left_anti.

Map and FlatMap

Spark Map function takes one element as input process it according to custom code (specified by the developer) and returns one element at a time.



A FlatMap function takes one element as input process it according to custom code (specified by the developer) and returns 0 or more elements at a time. First applies the function to all elements, and then flattens the results.

Dataset Map

Dataset<U> map(MapFunction<T, U>, Encoder<U>)

- Go through every record of the dataset
- Do something in the call() method of the MapFunction class
- Return a typed dataset

If we have a Weather domain object and want to apply a function across Rows:

Dataset<Weather> map(MapFunction<Row, Weather>, Encoder.bean<Weather.class>)

Action Functions – Trigger Transformations

Function	Description
count()	Returns the number of elements in a dataset
take(n)	Returns an array with the first n elements of the dataset to the driver program.
first()	Same as take(1)
takeSample(withReplacement, num, [seed])	Returns an array with a random sample of num elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.
collect()	Returns all the elements of the dataset as an array to the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
reduce(func)	Aggregates the elements of the dataset by using a function *func* (which takes two arguments and returns one), also known as a *reduce operation*. The function should be commutative and associative so that it can be computed correctly in parallel. The pairs on the same machine with the same key are combined.

Array is Row[]

Dangerous if large data sets, may crash driver.

Action Functions – Trigger Transformations

Function	Description
foreach (func)	Runs a function func on each element of the dataset. This is usually done for side effects such as updating an accumulator or interacting with external storage systems.
saveAsTextFile (path)	Writes the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem,
saveAsObjectFile (path)	Writes the elements of the dataset in a simple format by using Java serialization, which can then be loaded using SparkContext.objectFile()

Can use Lambdas or the Functional versions

https://spark.apache.org/docs/latest/api/java/index.html?org/apache/spark/api/java/function/package-summary.html

ForeachFunction<T>	Base interface for a function used in Dataset's foreach function.
ForeachPartitionFunction<T>	Base interface for a function used in Dataset's foreachPartition function.
Function<T1,R>	Base interface for functions whose return types do not create special RDDs.
Function0<R>	A zero-argument function that returns an R.
Function2<T1,T2,R>	A two-argument function that takes arguments of type T1 and T2 and returns an R.
Function3<T1,T2,T3,R>	A three-argument function that takes arguments of type T1, T2 and T3 and returns an R.
Function4<T1,T2,T3,T4,R>	A four-argument function that takes arguments of type T1, T2, T3 and T4 and returns an R.
MapFunction<T,U>	Base interface for a map function used in Dataset's map function.
MapGroupsFunction<K,V,R>	Base interface for a map function used in GroupedDataset's mapGroup function.
MapPartitionsFunction<T,U>	Base interface for function used in Dataset's mapPartitions.
PairFlatMapFunction<T,K,V>	A function that returns zero or more key-value pair records from each input record.
PairFunction<T,K,V>	A function that returns key-value pairs (Tuple2<K, V>), and can be used to construct PairRDDs.
ReduceFunction<T>	Base interface for function used in Dataset's reduce.

Checkpoints

Checkpointing, Cache, Persist, Explain

Apache Spark offers two distinct techniques for increasing performance:

- Caching, through cache() or persist(), which can save your data and the data lineage.
- Checkpointing, through checkpoint(), to save your data, without the lineage.

Data lineage is represented by the directed acyclic graph (DAG)

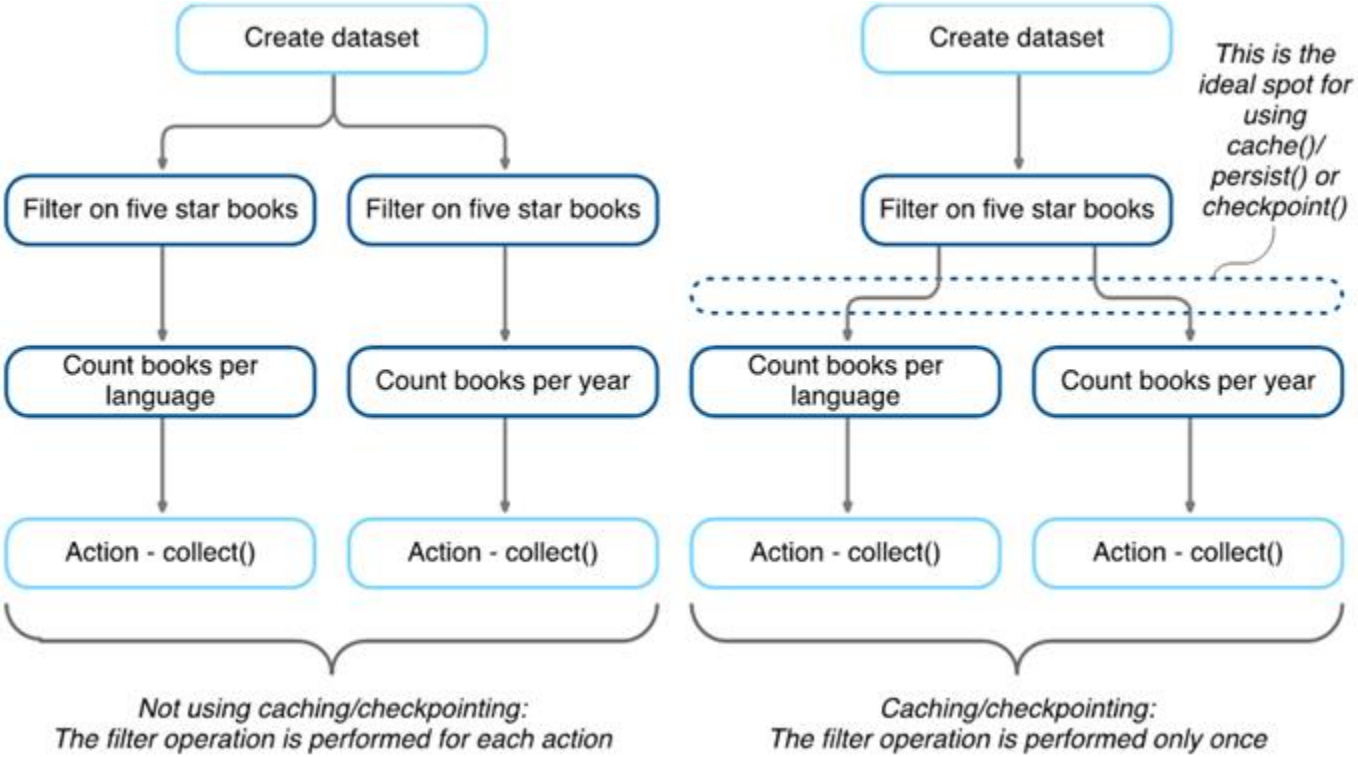
The DAG is not directly accessible to the user, except through the use of the dataframe's explain() method.

4 Types of Checkpointing:

	Eager	**Lazy**
Reliable	checkpoint	checkpoint(eager = false)
Local	localCheckpoint	localCheckpoint(eager = false)

- **Eager checkpointing** is the default checkpointing and happens immediately when requested.
- **Lazy checkpointing** does not and will only happen when an action is executed.
- **Local checkpointing** uses executor storage to write checkpoint files to and due to the executor lifecycle is considered unreliable.
- **Reliable checkpointing** uses a reliable data storage like Hadoop HDFS.

CheckPointing



Spark Session and Context

Earlier versions of Spark the SparkContext (JavaSparkContext for Java) is an entry point to Spark programming with RDD and to connect to Spark Cluster.

The **SparkContext** is used by the Driver Process of the Spark Application in order to establish a communication with the cluster and the resource managers in order to coordinate and execute jobs.

SQLContext is entry point of **SparkSQL** which can be received from sparkContext.

- Since Spark 2.0 **SparkSession** has been introduced and became an entry point to start programming with RDDS, DataFrame and Dataset. **It replaces the SQLContext**
- You can get the **SparkContext** from the **SparkSession**
- **SparkContext** is a Scala implementation entry point and **JavaSparkContext** is a java wrapper of SparkContext.

So in most cases today you only need SparkSession – except for checkpointDir.

You need to set a Checkpoint Directory

```
// Creates a session on a local master
  SparkSession spark = SparkSession.builder()
      .appName("CSV to Dataset")
      .master("local[*]")
      .getOrCreate();

  // set the checkpoint directory, get context from session
  SparkContext sparkContext = spark.sparkContext();
  sparkContext.setCheckpointDir("/tmp");
```

Explain()

Old Options:

- df1.explain(); // Show Physical Plan
- df1.explain(true); // Show Physical, Optimized Logical Plan, Analyzed Logical Plan, Parsed Logical Plan

New Options:

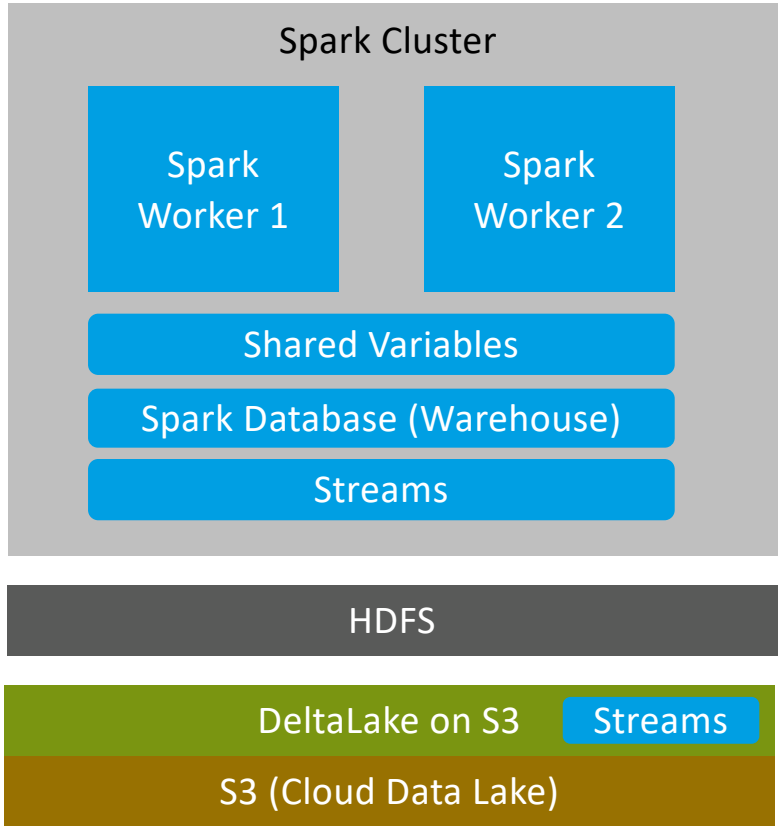
- df1.explain(“simple”); // as above
- df1.explain("extended"); // as above with (true)
- df1.explain("codegen"); // Shows the generated Code
- df1.explain("cost"); // Info stats, e.g. Union, Statistics(sizeInBytes=956.0 B)
- df1.explain("formatted"); // All the plans nicely formatted.

Formatted: Code reads CSV twice, adds two columns XX, YY then Union.

```
== Physical Plan ==
Union (5)
:- * Project (2)
:  +- Scan csv  (1)
+- * Project (4)
   +- Scan csv  (3)

(1) Scan csv
Output [5]: [outlook#16, temperature#17, humidity#18, windy#19, play#20]
Batched: false
Location: InMemoryFileIndex [file:/Users/kimhorn/Documents/JWork/SparkWork/spark-java-examples-two/Functional/CheckPointExplain-6/data/weather.csv]
ReadSchema: struct<outlook:string,temperature:string,humidity:string,windy:string,play:string>

(2) Project [codegen id : 1]
Output [7]: [outlook#16, temperature#17, humidity#18, windy#19, play#20, temperature#17 AS XX#109, humidity#18 AS YY#116]
Input [5]: [outlook#16, temperature#17, humidity#18, windy#19, play#20]

(3) Scan csv
Output [5]: [outlook#42, temperature#43, humidity#44, windy#45, play#46]
Batched: false
Location: InMemoryFileIndex [file:/Users/kimhorn/Documents/JWork/SparkWork/spark-java-examples-two/Functional/CheckPointExplain-6/data/weather.csv]
ReadSchema: struct<outlook:string,temperature:string,humidity:string,windy:string,play:string>

(4) Project [codegen id : 2]
Output [7]: [outlook#42, temperature#43, humidity#44, windy#45, play#46, temperature#43 AS XX#124, humidity#44 AS YY#131]
Input [5]: [outlook#42, temperature#43, humidity#44, windy#45, play#46]

(5) Union
```

Sharing

Ways to share and communicate across Spark Cluster and Session



Used to keep workers, parallel jobs in sync within cluster

Used to share data tables within cluster, or sessions

Pass data from job to job concurrently

Transient, not that reliable (unless on dedicated storage, e.g. EBS)

- old, not recommended these days, more inhouse - not cloud

Transactional, Upserts, Schema mngt, Unifies streams, caching Meta data, concurrency, performance (also on HDFS)

Sharable, Managed, Durable, Reliable, Cheaper

Cluster Wide Variables

Shared variables that can be used in parallel operations.

- By default, when Spark runs a function in parallel as a set of tasks on different nodes, it ships a copy of each variable used in the function to each task.
- Sometimes, a variable needs to be shared across tasks, or between tasks and the driver program.
- Spark supports two types of shared variables:
  1. *broadcast variables*, which can be used to cache a value in memory on all nodes, and
  2. *accumulators*, which are variables that are only "added" to, such as counters and sums.

```
Broadcast<int[]> broadcastVar = sc.broadcast(new int[] {1, 2, 3});
broadcastVar.value();
// returns [1, 2, 3]

LongAccumulator accum = jsc.sc().longAccumulator();
sc.parallelize(Arrays.asList(1, 2, 3, 4)).foreach(x -> accum.add(x));
accum.value();
// returns 10
```

Java RDDs

Spark Context / Java Context – Specific JavaRDDs

```
SparkSession spark = SparkSession.builder()
        .appName("WordCount DSS")
        .master("local")
        .getOrCreate();
SparkContext sc = spark.sparkContext();
JavaSparkContext jsc = JavaSparkContext.fromSparkContext(sc);
```

Have to use these old contexts:

- **SparkContext** represents the connection to a Spark cluster, and can be used to create RDDs, accumulators and broadcast variables on that cluster.
- **JavaSparkContext** is a Java-friendly version of SparkContext that returns JavaRDDs and works with Java collections instead of Scala ones.

Class JavaRDD<T>

Class JavaPairRDD<K,V>

Delta Lake

Spark PARQUET has limitations for complex Pipelines

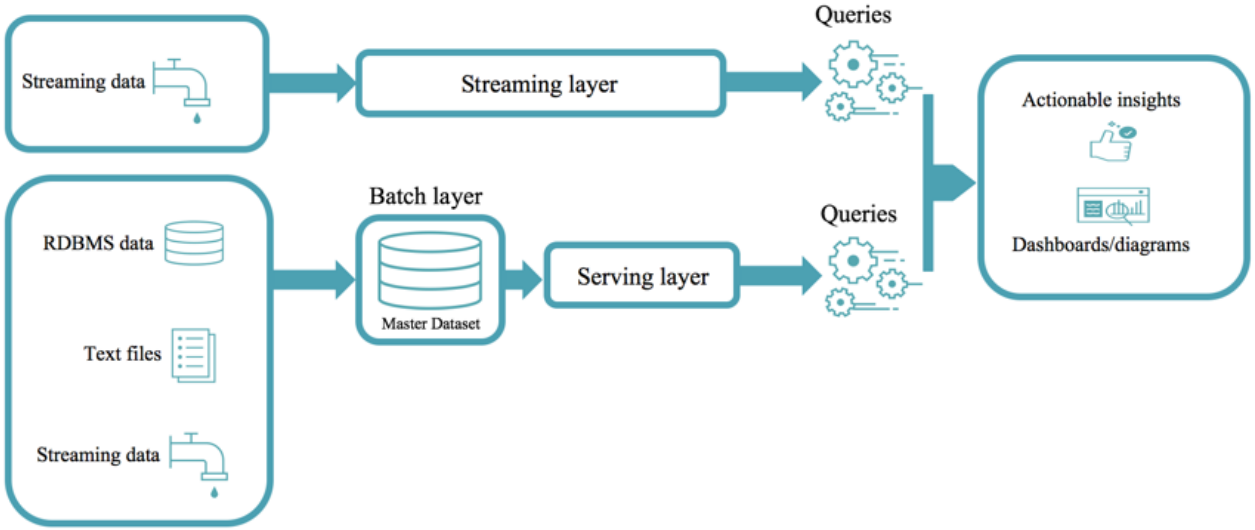
- Can’t do Updates and Deletes on Parquet based tables
- Merges with Parquet are possible, but a lot of code.
- Can’t read consistent data and write at same time from different applications (SparkSessions).
- Can’t read incrementally from large table with good throughput
- Can’t rollback with bad write, or overwrite (dangerous)
- Can’t replay historical data
- Can’t add late arriving data easily
- Can’t stream data in

Delta Lake fixes these issues.

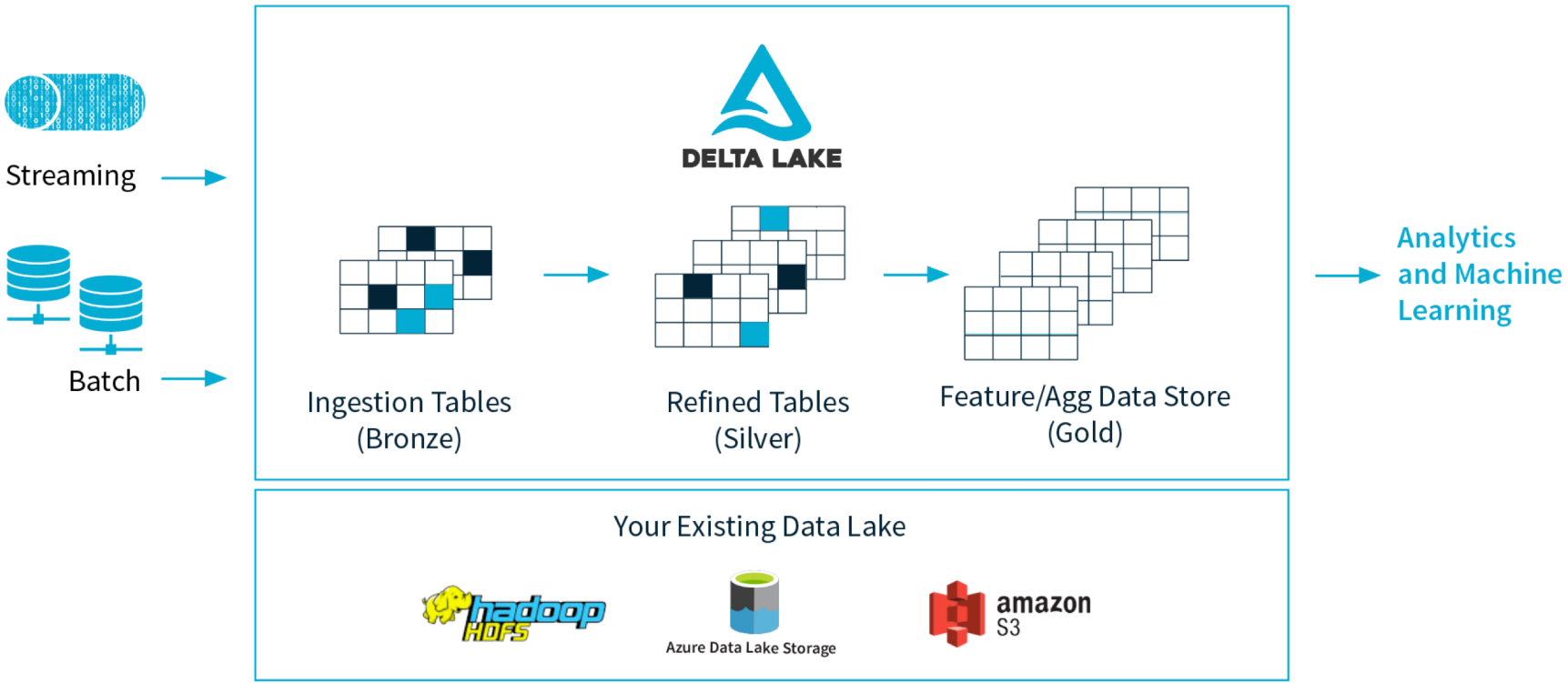
Lambda Architecture

With large data sets, it can take a long time to run the sort of queries that clients need. The **lambda architecture** addresses this problem by combining both batch and real-time processing methods. It features an append only immutable data source that serves as system of record. Timestamped events are appended to existing events (nothing is overwritten). Data is implicitly ordered by time of arrival.

It uses two pipelines, one batch and one streaming, hence the name *lambda* architecture. Streaming data is the changes after the batch 'Master Data' was built. The Master is Iteratively re-built.



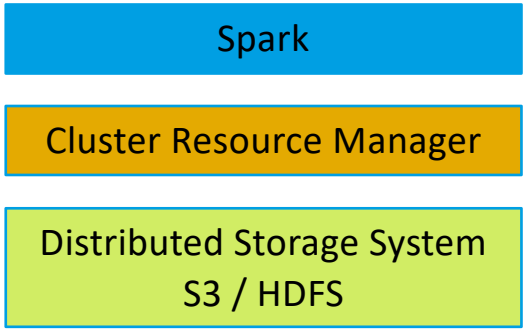
Delta Lake – Stream and Batch at the same time

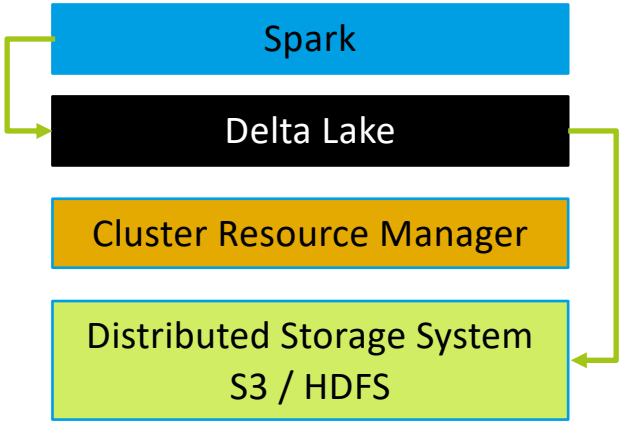


Delta Lake an Open Source Storage Layer

Delta Lake is a transactional layer (Spark library) applied on top of S3 data lake storage to get trustworthy data.

- It is an open-source project launched by Databricks, that also has a commercially supported version.
- It uses standard Parquet files for storage but adds many features for their management.
- Delta Lake can be used minimally to improve Spark Pipelines or in a larger scope as a Lakehouse.





Delta Lake

Rather than write intermediate processed data to Parquet files can use Open Source Delta Lake.

It Provides:

- **100% compatibility with Spark** – Tables, Catalogue API consistent
- **ACID transactions on Spark:** Serializable isolation levels ensure that readers never see inconsistent data.
- **Scalable metadata handling:** Leverages Spark distributed processing power to handle all the metadata for petabyte-scale tables with billions of files at ease.
- **Streaming and batch unification:** A table in Delta Lake is a batch table as well as a streaming source and sink. Streaming data ingest, batch historic backfill, interactive queries all just work out of the box.
- **Schema enforcement:** Automatically handles schema variations to prevent insertion of bad records during ingestion.
- **Time travel:** Data versioning enables rollbacks, full historical audit trails, and reproducible machine learning experiments.
- **Upserts and deletes:** Supports merge, update and delete operations to enable complex use cases like change-data-capture, slowly-changing-dimension (SCD) operations, streaming upserts, and so on.
- Rewrite data within a partition without registering changes in an application streaming from that table.

Performance Improvements

Delta boasts query performance of 10 to 100 times faster than with Apache Spark on Parquet.

- **Data Indexing** – Delta creates and maintains indexes on the tables.
- **Data Skipping** – Delta maintains file statistics on the data subset so that only relevant portions of the data is read in a query. (ZOrdering)
- **Compaction** – Delta manages file sizes of the underlying Parquet files for the most efficient use.
- **Data Caching** – Delta automatically caches highly accessed data to improve run times for commonly run queries.

Multi-Hop Architecture

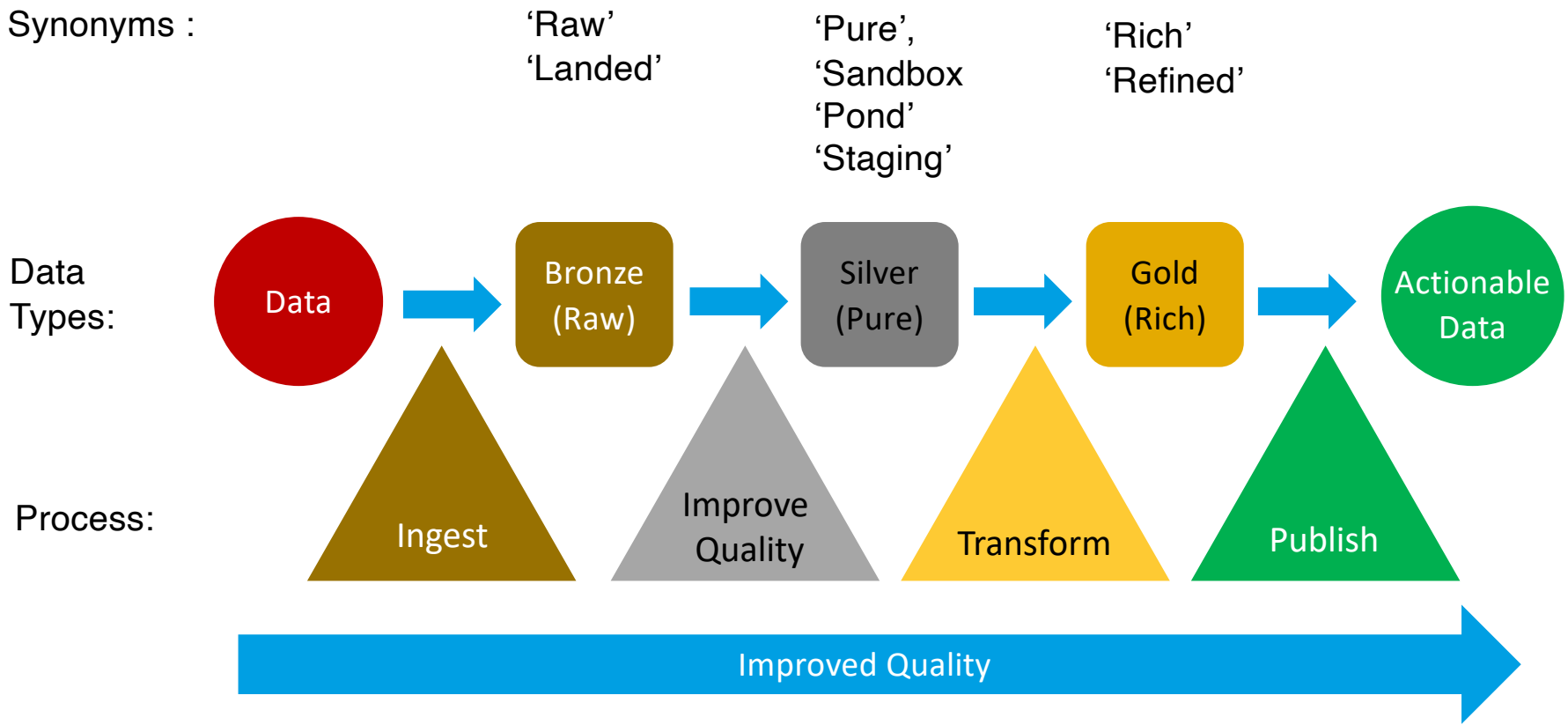
A common architecture that uses ‘Tables’ that correspond to different quality levels in the data engineering pipeline, progressively adding structure to the data. It is an improvement on the Lambda Architecture, see later slide.

It relies of 3 stages of processing that can combine batch and streaming workflows:

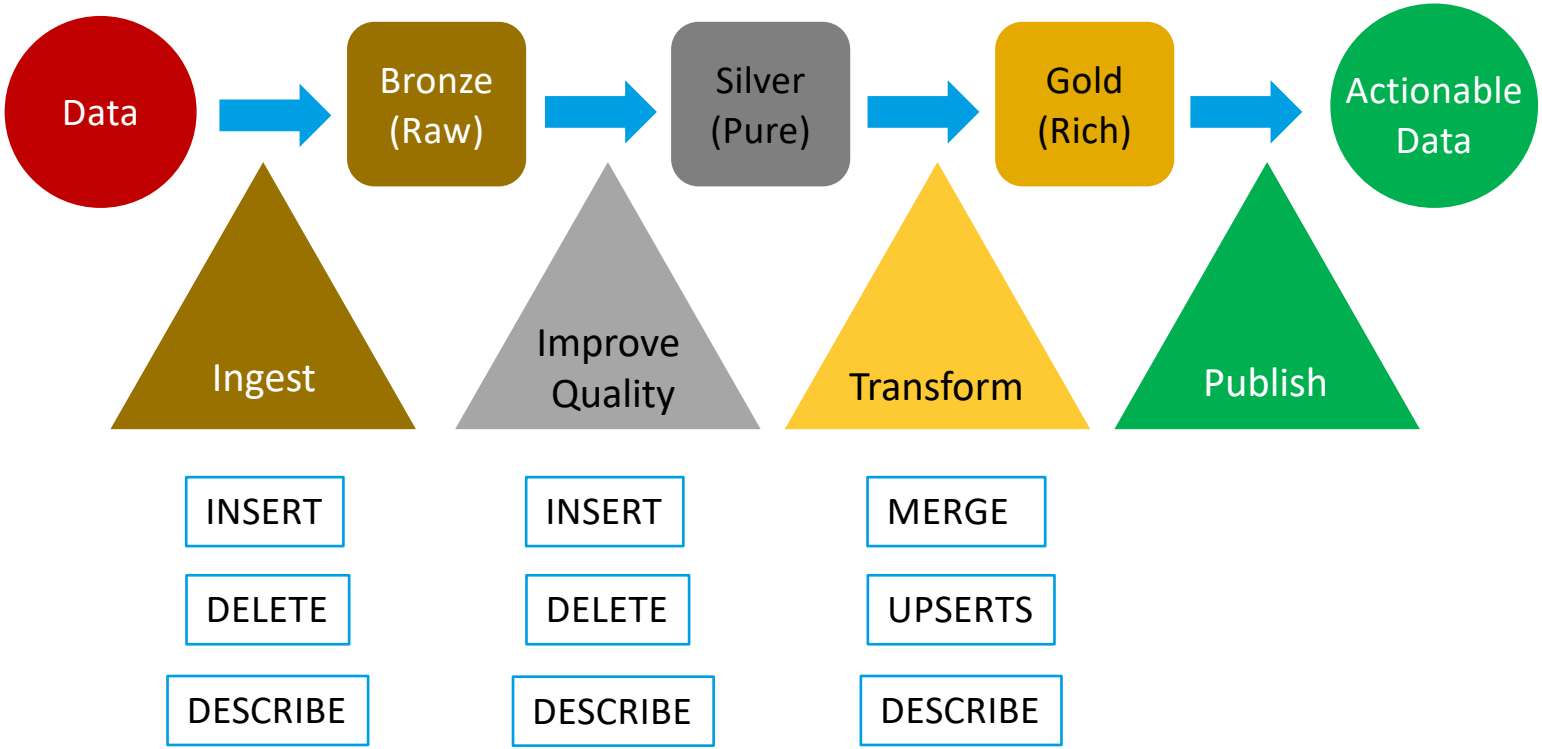
- **“Bronze” tables** - data ingestion, the raw data, the **“single source of truth”,** the data lake. *Often called ‘Landed’ data, or ‘Raw’ data.*
- **“Silver” tables** – cleaned, filtered, transformation and feature engineering performed. The data may be modelled into canonical formats. Has gone through operational processes to ensure data quality, security, metadata and other business defined SLAs are met. *Often called ‘Pure’, ‘Sandbox’ or ‘Pond’ Data*
- **“Gold” tables** - provide business level aggregates often used for reporting and dashboarding. You can join it with other datasets, apply custom functions, perform aggregations, implement machine learning, and more. The goal of this step is to get *rich data*, the fruit of your analytics work. It is finalised data ready for consumption, e.g. for machine learning training or prediction. *Often called ‘Production’ or ‘Refined’ Data*

The inherent Table base, is a property of Delta Lake. Means that the properties Delta Lake adds to Spark is critical.

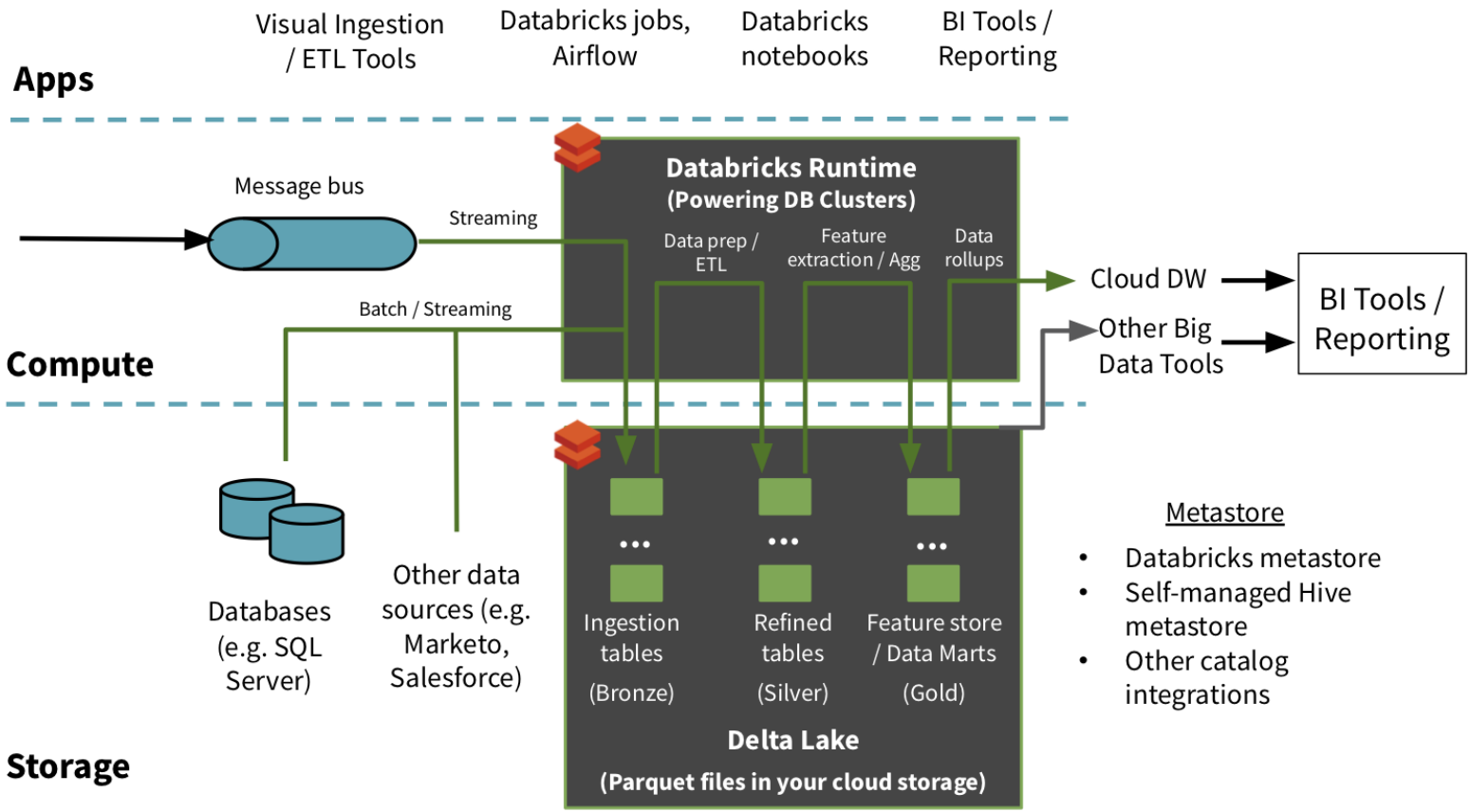
Multi-Hop - Data Types



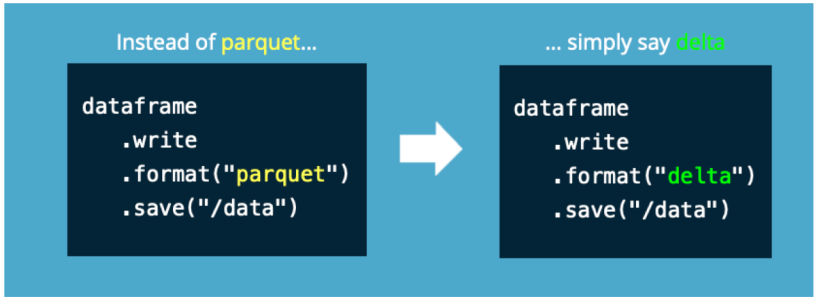
DeltaLake Operations

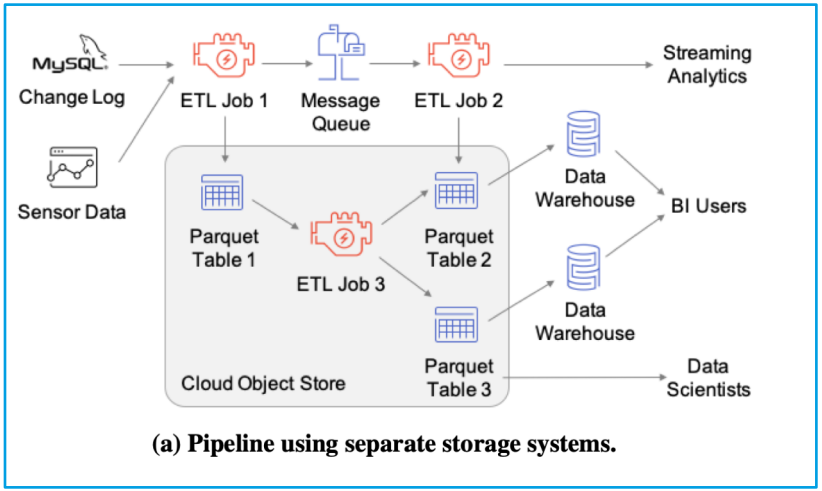


Delta Lake Architecture - Improved Pipeline

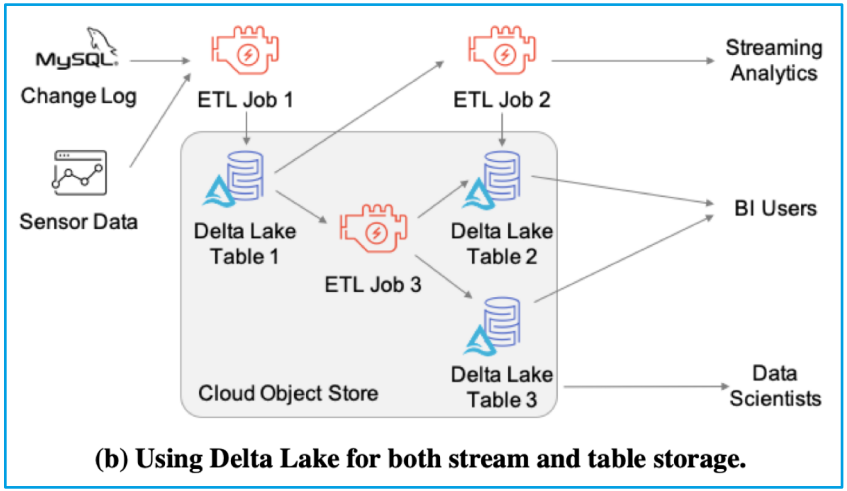


Simple Integration with ‘delta’ format





(a) Pipeline using separate storage systems.



(b) Using Delta Lake for both stream and table storage.

show TABLE extended from default.deltaweather:

A Table based on existing Parquet file, so its Not Managed.

database	tableName	isTemporary	information
default	deltaweather	false	Database: default Table: deltaweather Created Time: Mon Mar 21 11:05:11 AEDT 2022 Last Access: UNKNOWN Created By: Spark 3.1.2 Type: EXTERNAL Provider: delta Location: file:/Users/kimhorn/Documents/JWork/SparkWork/spark-java-examples-two/DeltaLake/data/tmp/weather2 Partition Provider: Catalog

show TABLE extended from bronze.deltaweather2:

A Table created in a Managed DB.

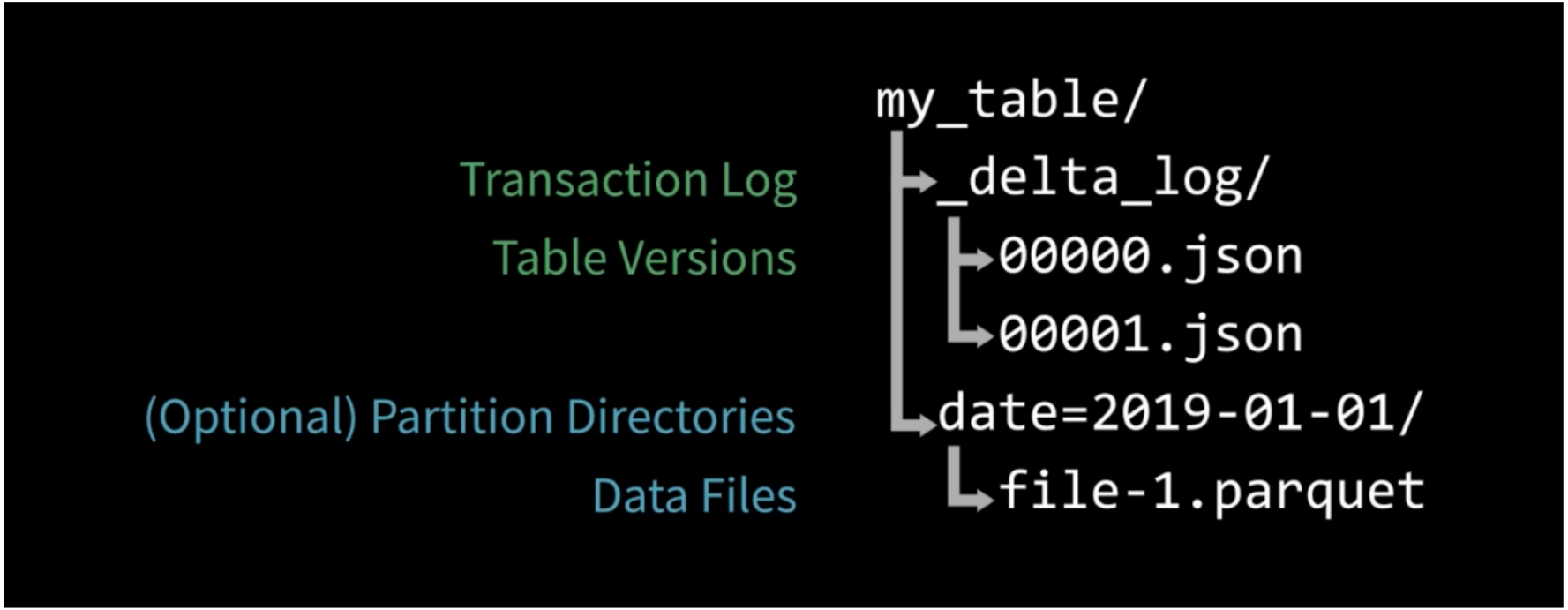
database	tableName	isTemporary	information
default	deltaweather2	false	Database: bronze Table: deltaweather2 Created Time: Mon Mar 21 11:05:12 AEDT 2022 Last Access: UNKNOWN Created By: Spark 3.1.2 Type: MANAGED Provider: delta Location: file:/Users/kimhorn/Documents/JWork/SparkWork/spark-java-examples-two/DeltaLake/DeltaLakeMetaData-13/spark-warehouse/bronze.db/deltaweather2 Partition Provider: Catalog

Delta Table Type (io.delta.tables)

```
SparkSession spark = SparkSession.builder()
    .config("spark.sql.extensions", "io.delta.sql.DeltaSparkSessionExtension")
    .config("spark.sql.catalog.spark_catalog", "org.apache.spark.sql.delta.catalog.DeltaCatalog")
    .appName("CSV to DeltaLake")
    .master("local")
    .getOrCreate();

DeltaTable deltaTable = DeltaTable.forPath("../data/tmp/weather");
deltaTable.toDF().show();

deltaTable.update( // predicate using Spark SQL functions
    functions.col("windy").equalTo("TRUE"), new HashMap<String, Column>() {
      {
        put("windy", functions.lit("VERY TRUE"));
      }
    });

deltaTable.toDF().show();
```

Main Methods: create, update, delete, merge, columnBuilder, forPath, toDF, history, replace, vacuum, convertToDelta

Delta On Disk



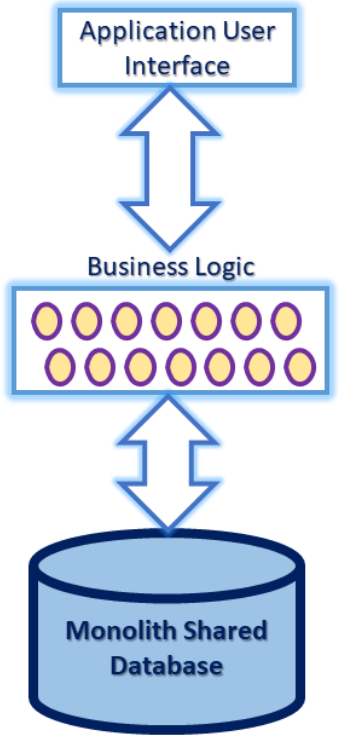
Storage with parquet files with each transaction change in a delta log json file.

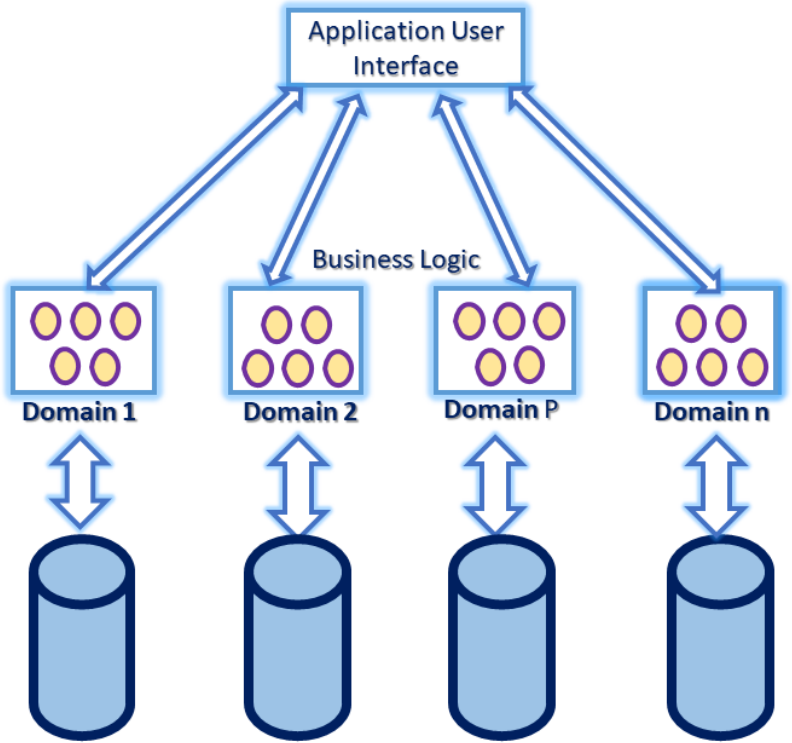
Delta Lake Versions

Delta Lake version	Apache Spark version
1.1.x, (1.1.0)	3.2.x
1.0.x (1.0.1, 1.0.0)	3.1.x
0.7.x and 0.8.x	3.0.x

Data Mesh

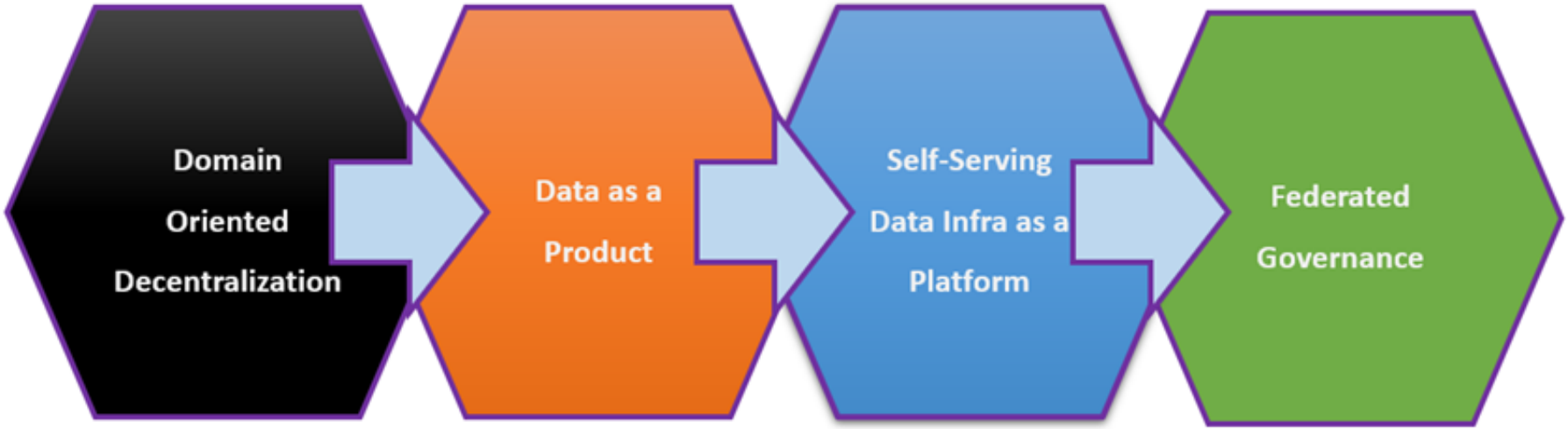
Database for each Domain





Different Database for different domain

Data Mesh Principles



Principles

1. **Domain Oriented Approach:** With a data mesh, a mass of data is broken down into domain-specific streams, which can be transformed to create a joint aggregate view of the business domain. These data streams are owned by independent teams or users, who are, in turn, attached to business experts who analyze the data for insights.
2. **Data as a Product:** The concerned team sees the data as their product and is responsible for processing the data to be used as a ready-to-use unit.
3. **Self-serve data infrastructure as a platform:** Further, an automated platform (Infra as a platform) allows the product team to control the lifecycle of data from the hands of a developer at the source to connect the data product and then run the semantic queries at mesh level. The platform provides storage, pipeline, data catalog, and access control to the domains. It reduces any chance of duplication in effort and hence increases the efficiency of data treatment.
4. **Federated Governance:** In the end, the implementation of data mesh architecture is defined by federated governance with standards and interoperability as primary architectural guidelines. It emphasizes that each domain should be discoverable, addressable, self-describing, secure, trustworthy, and interoperable.

Why Data Mesh ?

The data mesh architecture deals with the data more reliably and processes that information in real-time. Additionally, it can overcome multiple challenges of monolith data architecture. A few of them are listed below:

- It brings ownership and hence the accountability and responsibility of sharing that data as a product.
- It improves quality as data is treated at the source and is good to be used as an independent unit.
- With the data cleaned at the source, the responsibilities are equally shared and hence support organizational scaling.
- It increases efficiency with the incorporation of data infrastructure as a platform.
- It can handle a vast set of data without any difficulty, as they are domain-wise segregated for their respective domain experts to work on them.
- The data can now be directly incorporated into data analytics tools, making it more feasible in operational values.

Differences to Monolith

Moving to the data mesh architecture from monolith require a mindset shift, which is more apparent from the below-listed analogy (shift from monolith mindset to the data mesh mindset):

- Centralized ownership to decentralized ownership
- Pipelines as a priority to domain data as a first-class concern
- Data as a by-product to data as a product
- A siloed data engineering team to cross-functional domain-data teams
- A centralized data lake/warehouse to an ecosystem of data products

Spark Stand Alone

Starting Spark Stand Alone

Download Spark TAR from: https://spark.apache.org/downloads.html

e.g. **spark-3.2.1-bin-hadoop3.2.tgz**
Un-tar into directory

Start a Master :
>> ./sbin/start-master.sh

Logs will tell you the URL, e.g.
22/02/09 14:14:54 INFO Master: Starting Spark master at spark://Kims-Work-Mac.local:7077
22/02/09 14:14:54 INFO Master: Running Spark version 3.2.1
22/02/09 14:14:54 INFO Utils: Successfully started service 'MasterUI' on port 8080.

Web Browser: http://localhost:8080/
Look on app to see the Spark URL, also mentioned in log above.

Find the URL

Spark Master at spark://Kims-MacBook-Pro.local:7077

URL: spark://Kims-MacBook-Pro.local:7077
Alive Workers: 0
Cores in use: 0 Total, 0 Used
Memory in use: 0.0 B Total, 0.0 B Used
Resources in use:
Applications: 0 Running, 0 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers (0)

Worker Id	Address	State	Cores	Memory	Resources

Running Applications (0)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration

Completed Applications (0)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration

Start a Worker with the URL

On the current machine - the machines to be workers in a cluster:

```
>> ./sbin/start-worker.sh  spark://Kims-Work-Mac.local:7077 --cores 2 --memory 4g
```

Spark only allows you to start **1 Worker** on a local machine now (starting > 1 is now deprecated).

- Can't have a Mac cluster unless use docker (see next section).
- Start-slave is deprecated.

```
>> ./sbin/start-worker.sh spark://Kims-Work-Mac.local:7077 --cores 2 --memory 4g
>> org.apache.spark.deploy.worker.Worker running as process 75279.  Stop it first.
```

To stop the worker (don't do it now):

```
>> ./sbin/stop-worker.sh spark://Kims-Work-Mac.local:7077
```

To stop the master (don't do it now):

```
>> ./sbin/stop-master.sh
```

Worker is Started

Spark Master at spark://Kims-MacBook-Pro.local:7077

URL: spark://Kims-MacBook-Pro.local:7077
Alive Workers: 1
Cores in use: 2 Total, 0 Used
Memory in use: 4.0 GiB Total, 0.0 B Used
Resources in use:
Applications: 0 Running, 0 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers (1)

Worker Id	Address	State	Cores	Memory	Resources
worker-20220209191654-192.168.0.16-49343	192.168.0.16:49343	ALIVE	2 (0 Used)	4.0 GiB (0.0 B Used)	

Running Applications (0)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration

Completed Applications (0)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration

Click on Worker Link

Spark Worker at 192.168.0.16:49343

ID: worker-20220209191654-192.168.0.16-49343
Master URL: spark://Kims-MacBook-Pro.local:7077
Cores: 2 (0 Used)
Memory: 4.0 GiB (0.0 B Used)
Resources:

Back to Master

Running Executors (0)

ExecutorID	State	Cores	Memory	Resources	Job Details	Logs

We can now submit a job to master:

./bin/spark-submit --class com.thing.SimpleCluster --master "spark://Kims-Work-Mac.local:7077" /Users/kimhorn/Documents/JWork/SparkWork/spark-java-examples-two/Basic/SimpleCluster-Fat-Jar-9/target/spark-simplecluster.jar

See Completed Application Below

Spark Master at spark://Kims-MacBook-Pro.local:7077

URL: spark://Kims-MacBook-Pro.local:7077
Alive Workers: 1
Cores in use: 2 Total, 0 Used
Memory in use: 4.0 GiB Total, 0.0 B Used
Resources in use:
Applications: 0 Running, 1 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers (1)

Worker Id	Address	State	Cores	Memory	Resources
worker-20220209191654-192.168.0.16-49343	192.168.0.16:49343	ALIVE	2 (0 Used)	4.0 GiB (0.0 B Used)	

Running Applications (0)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration

Completed Applications (1)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
app-20220209194315-0000	Simple Cluster Example	2	1024.0 MiB		2022/02/09 19:43:15	kim	FINISHED	6 s

Completed Applications

Spark Worker at 192.168.0.16:49343

ID: worker-20220209191654-192.168.0.16-49343
Master URL: spark://Kims-MacBook-Pro.local:7077
Cores: 2 (0 Used)
Memory: 4.0 GiB (0.0 B Used)
Resources:

Back to Master

Running Executors (0)

ExecutorID	State	Cores	Memory	Resources	Job Details	Logs

Finished Executors (1)

ExecutorID	State	Cores	Memory	Resources	Job Details	Logs
0	KILLED	2	1024.0 MiB		**ID:** app-20220209194315-0000 **Name:** Simple Cluster Example **User:** kim	stdout stderr

Interactive Spark

>> ./bin/spark-shell (for Scala, or ./bin/pyspark for python)

....

....

To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).

22/02/09 19:47:17 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable

Spark context Web UI available at http://192.168.0.16:4040

Spark context available as 'sc' (master = local[*], app id = local-1644396438758).

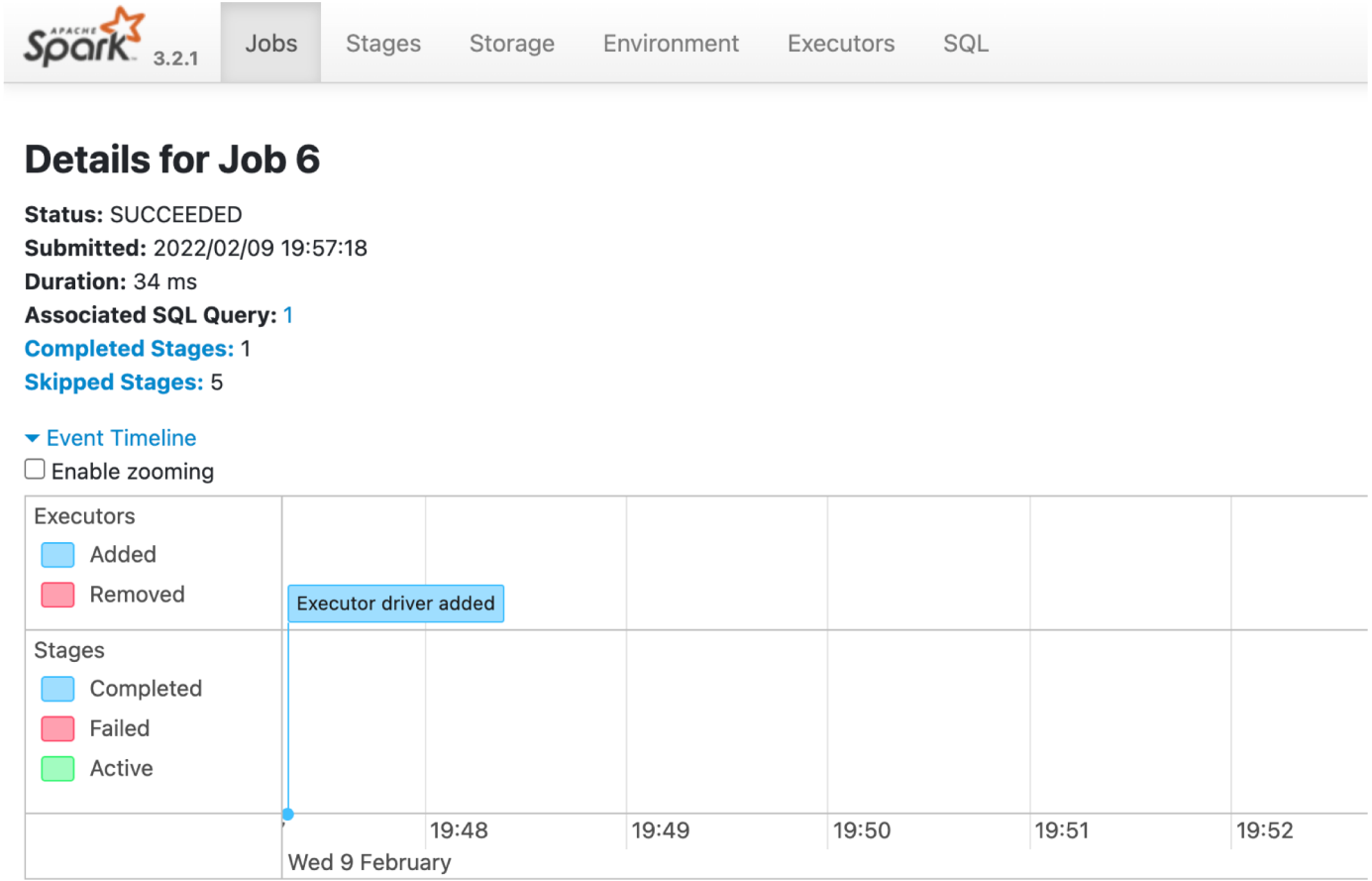
Spark session available as 'spark'.

Now Open Browser: http://192.168.0.16:4040

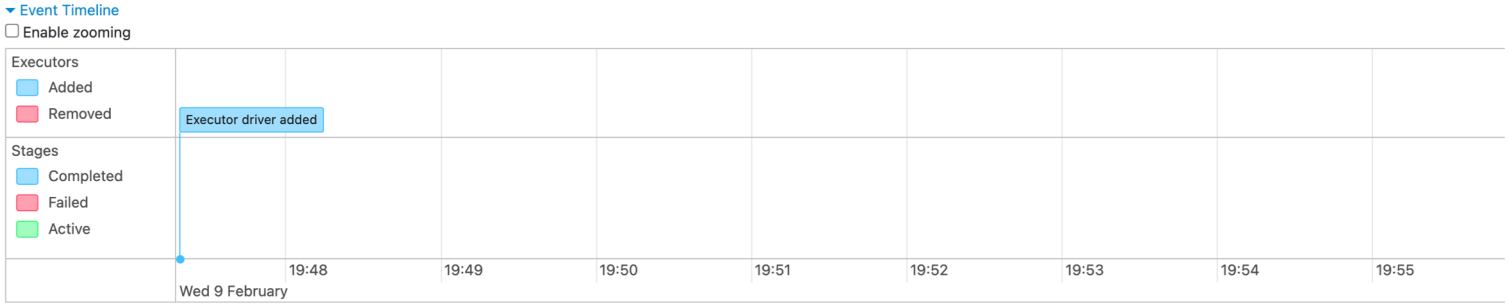
Spark Scala Interface

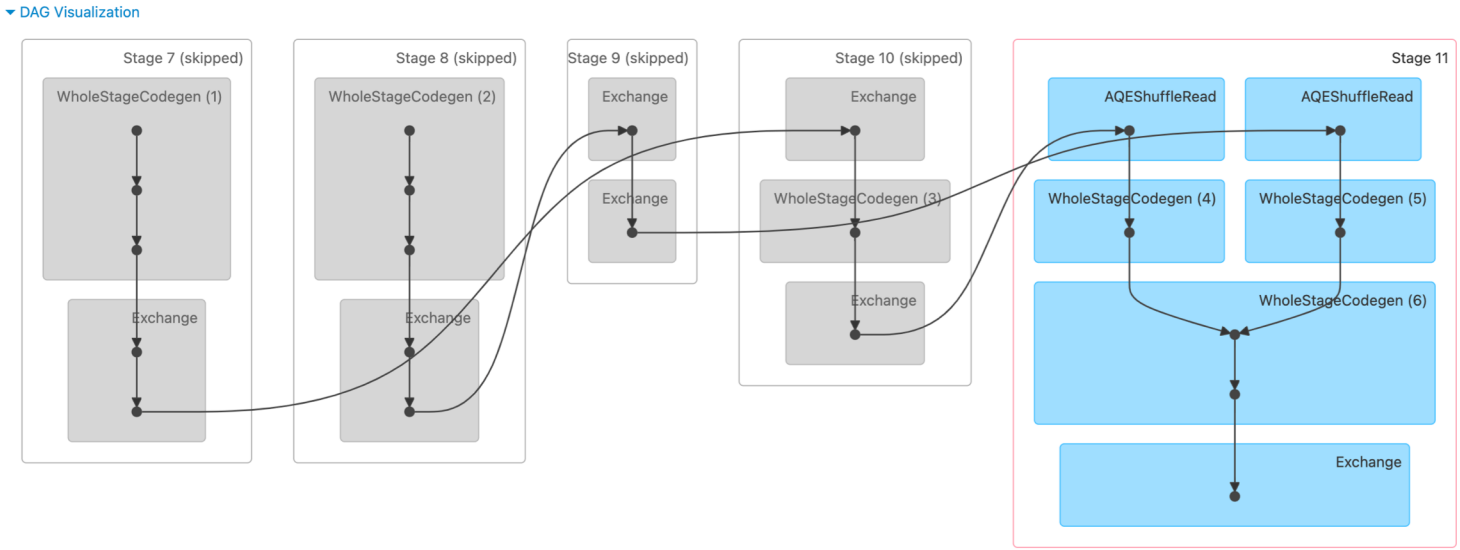
```
scala> val simpleNumbers = spark.range(1, 1000000)
simpleNumbers: org.apache.spark.sql.Dataset[Long] = [id: bigint]

scala> val times5 = simpleNumbers.selectExpr("id * 5 as id")
times5: org.apache.spark.sql.DataFrame = [id: bigint]

scala> times5.show()
+---+
| id|
+---+
|  5|
| 10|
| 15|
.....
|100|
+---+
```

The Web UI


Spark 3.2.1
Jobs
Stages
Storage
Environment
Executors
SQL
Details for Job 6
Status: SUCCEEDED
Submitted: 2022/02/09 19:57:18
Duration: 34 ms
Associated SQL Query: 1
Completed Stages: 1
Skipped Stages: 5
Event Timeline
Enable zooming
Executors
Added
Removed
Executor driver added
Stages
Completed
Failed
Active
19:48
19:49
19:50
19:51
19:52
Wed 9 February

DAG Viewer


Event Timeline
Enable zooming
Executors
Added
Removed
Executor driver added
Stages
Completed
Failed
Active
19:48
19:49
19:50
19:51
19:52
19:53
19:54
19:55
Wed 9 February


DAG Visualization
Stage 7 (skipped)
WholeStageCodegen (1)
Exchange
Stage 8 (skipped)
WholeStageCodegen (2)
Exchange
Stage 9 (skipped)
Exchange
Exchange
Stage 10 (skipped)
Exchange
WholeStageCodegen (3)
Exchange
Stage 11
AQEShuffleRead
AQEShuffleRead
WholeStageCodegen (4)
WholeStageCodegen (5)
WholeStageCodegen (6)
Exchange

PySpark ./bin/pyspark

```
>>> from pyspark.sql import SparkSession
>>> spark = SparkSession.builder.getOrCreate()
>>> from datetime import datetime, date
>>> from pyspark.sql import Row
>>> df = spark.createDataFrame([
...     Row(a=1, b=2., c='string1', d=date(2000, 1, 1), e=datetime(2000, 1, 1, 12, 0)),
...     Row(a=2, b=3., c='string2', d=date(2000, 2, 1), e=datetime(2000, 1, 2, 12, 0)),
...     Row(a=4, b=5., c='string3', d=date(2000, 3, 1), e=datetime(2000, 1, 3, 12, 0))
... ])
>>> df
DataFrame[a: bigint, b: double, c: string, d: date, e: timestamp]
>>> df.show()
+---+---+-------+----------+-------------------+
|  a|  b|      c|         d|                  e|
+---+---+-------+----------+-------------------+
|  1|2.0|string1|2000-01-01|2000-01-01 12:00:00|
|  2|3.0|string2|2000-02-01|2000-01-02 12:00:00|
|  4|5.0|string3|2000-03-01|2000-01-03 12:00:00|
+---+---+-------+----------+-------------------+
```

Spark Docker Cluster

Laptop Cluster using Docker-Compose

To solve the issue of not being able to start 2 workers on a local Spark, a docker cluster can be used for testing.
See the “spark-docker-cluster” Git project.

The docker-compose starts 3 containers,
each with Spark 3.2.1 :

- spark-master: Port 9000
- spark-worker-one: Port 9001
- spark-worker-two: Port 9002

You can submit a Jar from the laptop with exec,
or from a CLI on the master container.

There are two mounts one for JARS, and one for shared data

- /opt/spark-apps -Used to make available your app's jars on all workers & master
- /opt/spark-data -Used to make available your app's data on all workers & master

Spark Master

Spark Master at spark://81b53d27d1c1:7077

URL: spark://81b53d27d1c1:7077
Alive Workers: 2
Cores in use: 2 Total, 0 Used
Memory in use: 2.0 GiB Total, 0.0 B Used
Resources in use:
Applications: 0 Running, 1 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers (2)

Worker Id	Address	State	Cores	Memory	Resources
worker-20220214235402-172.27.0.3-7000	172.27.0.3:7000	ALIVE	1 (0 Used)	1024.0 MiB (0.0 B Used)	
worker-20220214235403-172.27.0.4-7000	172.27.0.4:7000	ALIVE	1 (0 Used)	1024.0 MiB (0.0 B Used)	

Running Applications (0)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration

Completed Applications (1)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
app-20220215001328-0000	Simple Cluster Example	2	1024.0 MiB		2022/02/15 00:13:28	root	FINISHED	21 s

Machine Learning

Spark Machine Learning

Includes:

- **ML Algorithms:** common learning algorithms such as classification, regression, clustering, and collaborative filtering
- **Featurization**: feature extraction, transformation, dimensionality reduction, and selection
- **Pipelines:** tools for constructing, evaluating, and tuning ML Pipelines
- **Persistence:** saving and load algorithms, models, and Pipelines
- **Utilities:** linear algebra, statistics, data handling, etc.
- Spark 3.1.2 includes updates that deprecate the RDD based approaches, and now supports DataFrames/Sets. The library is still the MLlib library.
- The DataFrame-based API for MLlib provides a uniform API across ML algorithms and across multiple languages.

ML Process

1. Spark ML only accepts numeric data, so all categories have to be converted (transformed) to numeric, using a ***StringIndexer*.** You don't need to use dataset map.
2. For most models the data (not in *LIBSVM format)* has to be formatted into a single column called "features", using a ***VectorAssembler***.
3. Need to deal with null values, e.g. remove rows, replace with mean or mode, etc.
4. ***Fit*** and ***Transform*** methods.
    1. A feature ***transformer*** might take a DataFrame, read a column (e.g., text), map it into a new column (e.g., feature vectors), and output a new DataFrame with the mapped column appended.
    2. An ***Estimator*** abstracts the concept of a learning algorithm or any algorithm that ***fits*** or trains on data. Technically, an Estimator implements a method ***fit***(), which accepts a DataFrame and produces a ***Model***, which is a ***Transformer***. Have to set the Label (target-name) and Features columns:

        .setLabelCol("target-name")

        .setFeaturesCol("features")
    3. A learning model ***transformer*** might take a DataFrame, read the column containing feature vectors, and using the ***Model*** predict the label for each feature vector, and output a new DataFrame with predicted labels appended as a column.

        .setLabelCol("target-Name")

        .setPredictionCol("prediction");

The Famous “Iris Data”

Setosa

Vericolor

Virginica

Spark ML is based on Pipelines

A pipeline consists of a series of transformer and estimator stages that typically prepare a DataFrame for modelling and then train a predictive model. For example, a pipeline with seven stages:

1. A **StringIndexer estimator** that converts string values to indexes for categorical features
2. A **VectorAssembler** that combines categorical features into a single vector
3. A **VectorIndexer** that creates indexes for a vector of categorical features
4. A **VectorAssembler** that creates a vector of continuous numeric features
5. A **MinMaxScaler** that normalizes continuous numeric features
6. A **VectorAssembler** that creates a vector of categorical and continuous features
7. A **DecisionTreeClassifier** that trains a classification model.

Pipeline

The pipeline can be re-used on different data sets.

Run the Pipeline to train a model

Run the pipeline as an Estimator on the training data to train (fit) a model.

```
piplineModel = pipeline.fit(train)
```

Run the Pipeline to test a model

Generate label predictions. Transform the test data with all of the stages and the trained model in the pipeline to generate label predictions.

```
prediction = piplineModel.transform(test)
```