W261 - Week 8
Introduction to SPARK and Data

UC Berkeley - MIDS

Table of Contents

- Housekeeping
- Semester Review
- Final Project
- Data storage and access
    - Vectorized column-based memory layout
    - Wholestage code generation
    - The Catalyst optimizer takes a computational query and converts it into an execution plan. It goes through four transformational phases:
    - 1. Analysis
    - 2. Logical optimization
    - 3. Physical planning
    - 4. Code generation (Project Tungsten)
    - Pushdown predicates
    - Joins
    - Python dataframe API versus SQL API
    - Lab: show-and-tell rather than doing

Housekeeping

MidTerm Course Evaluation (next w

Please fill out the course evals over the next 24-48 h ant for the course and for the program. Thank you!!

https://cc ley.edu/berkeley/

Thank you for complete the midterm evals!

Take 10 minutes to omplete now ue this week)

Semester Schedule

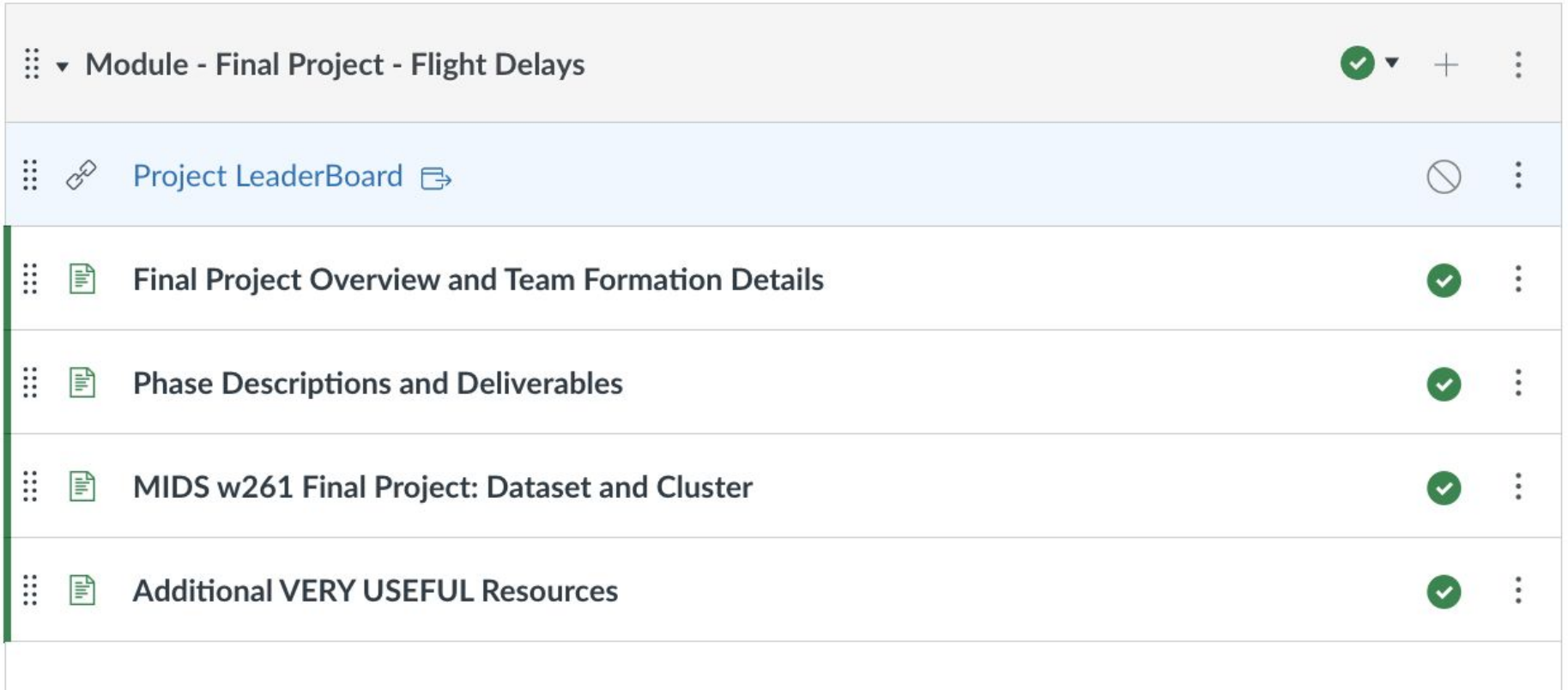
B	C	D	E
Module	**Section**	**i526 Applied Machine Learning**	**Core Assignments: HW/Projects due 11:59PM, Sunday**
1	Intro+Review	Machine Learning at Scale introduction and course overview	
2		Introduction to Hadoop Streaming and Naive Bayes	HW01 Due Sunday midnight at the end of this week
3		Map-Reduce Algorithm Design patterns and map-shuffle-reduce	
4	Intro to Spark	Intro to Spark/Map-Reduce with RDDs (part 1)	HW02 Due
5		Intro to Spark/Map-Reduce with RDDs (part 2)	
6	ML at Scale	Distributed Supervised ML DSML (part 1): Linear Regression	HW03 Due
7		DSML (part 2): Linear Classification, Maximum likelihood	
8		Big Data Systems and Pipelines	HW04 Due
9	Graphs	Graph Algorithms at Scale (part 1): shortest path etc.	FP: Finalize Team members for the final project
10		Graph Algorithms at Scale (part 2): pagerank and variants	FP Phase 1 is due Sunday midnight at the end of this week
11	Trees&Ensembles	Non-gradient-ML: decision trees, random forests, ensembles	FP Phase 2 is due Sunday along with HW5
12	Final Project	Mid-Project review and presentations	FP Phase 3 due
13	Recommenders	Recommender systems, ALS, and Spark ML	FP Phase 4 is due Sunday midnight
14	Final Project	Final Project Presentations	FP Phase 5: Team presentation

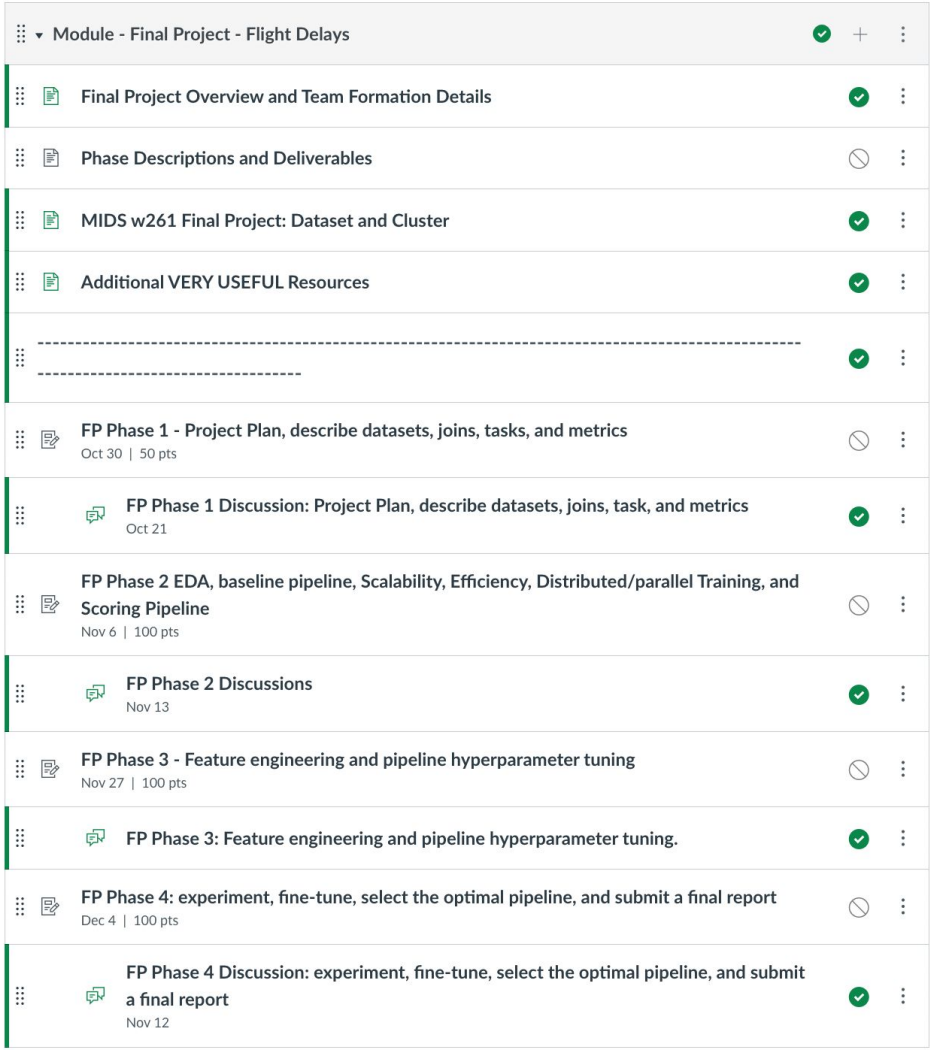
A common way to deserialize JSON is to first create a class with properties and fields that represent one or more of the JSON properties. Then, to deserialize from a string or a file, call the JsonSerializer. Deserialize method.

Final Project

Final Project Schedule (see https://digitalcampus.instructure.com/)

Description		Due Dates
Phase 0: Finalize Teams (teams of four people from same section)		Due Sunday end of Week 9
Phase I - project plan, describe datasets, joins, tasks, and metrics		Due Sunday of Week 10
Phase II - EDA, Baseline Pipeline on all data: Scalability, Efficiency, Distributed/parallel Training, and Scoring Pipeline	Phase II Video Update (2 minutes max)	Due end of Week 11 (Sunday)
Phase III Feature engineering + hyperparameter tuning, + **in-class review**		Due by end of Week 12
Phase IV - Advanced model architectures and loss functions, select an optimal algorithm, fine-tune & Final report write up	Phase IV Video Update (2 minutes max)	Due end of Week 13
Phase V - In-class Presentation		Week 14 live session


Module - Final Project - Flight Delays
Project LeaderBoard
Final Project Overview and Team Formation Details
Phase Descriptions and Deliverables
MIDS w261 Final Project: Dataset and Cluster
Additional VERY USEFUL Resources


Module - Final Project - Flight Delays
Final Project Overview and Team Formation Details
Phase Descriptions and Deliverables
MIDS w261 Final Project: Dataset and Cluster
Additional VERY USEFUL Resources
FP Phase 1 - Project Plan, describe datasets, joins, tasks, and metrics
Oct 30 | 50 pts
FP Phase 1 Discussion: Project Plan, describe datasets, joins, task, and metrics
Oct 21
FP Phase 2 EDA, baseline pipeline, Scalability, Efficiency, Distributed/parallel Training, and Scoring Pipeline
Nov 6 | 100 pts
FP Phase 2 Discussions
Nov 13
FP Phase 3 - Feature engineering and pipeline hyperparameter tuning
Nov 27 | 100 pts
FP Phase 3: Feature engineering and pipeline hyperparameter tuning.
FP Phase 4: experiment, fine-tune, select the optimal pipeline, and submit a final report
Dec 4 | 100 pts
FP Phase 4 Discussion: experiment, fine-tune, select the optimal pipeline, and submit a final report
Nov 12

Review Final Project and timeline: 5 Phases

Teams should have 4 members.

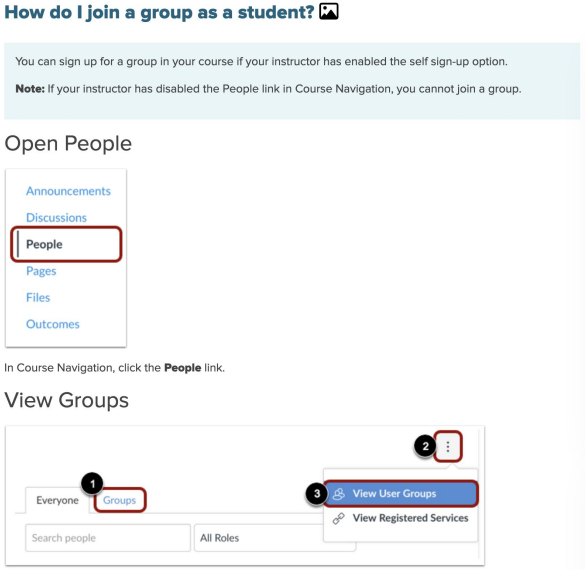
- Create teams on Canvas
- We will circulate instructions on Slack shortly.

- 5 Phases
  - 4 work phases + Phase 5 which consists of the final presentation

Start planning your teams!

You can create teams on Canvas. **We will circulate instructions on Slack shortly. The following is a draft of the instructions:**

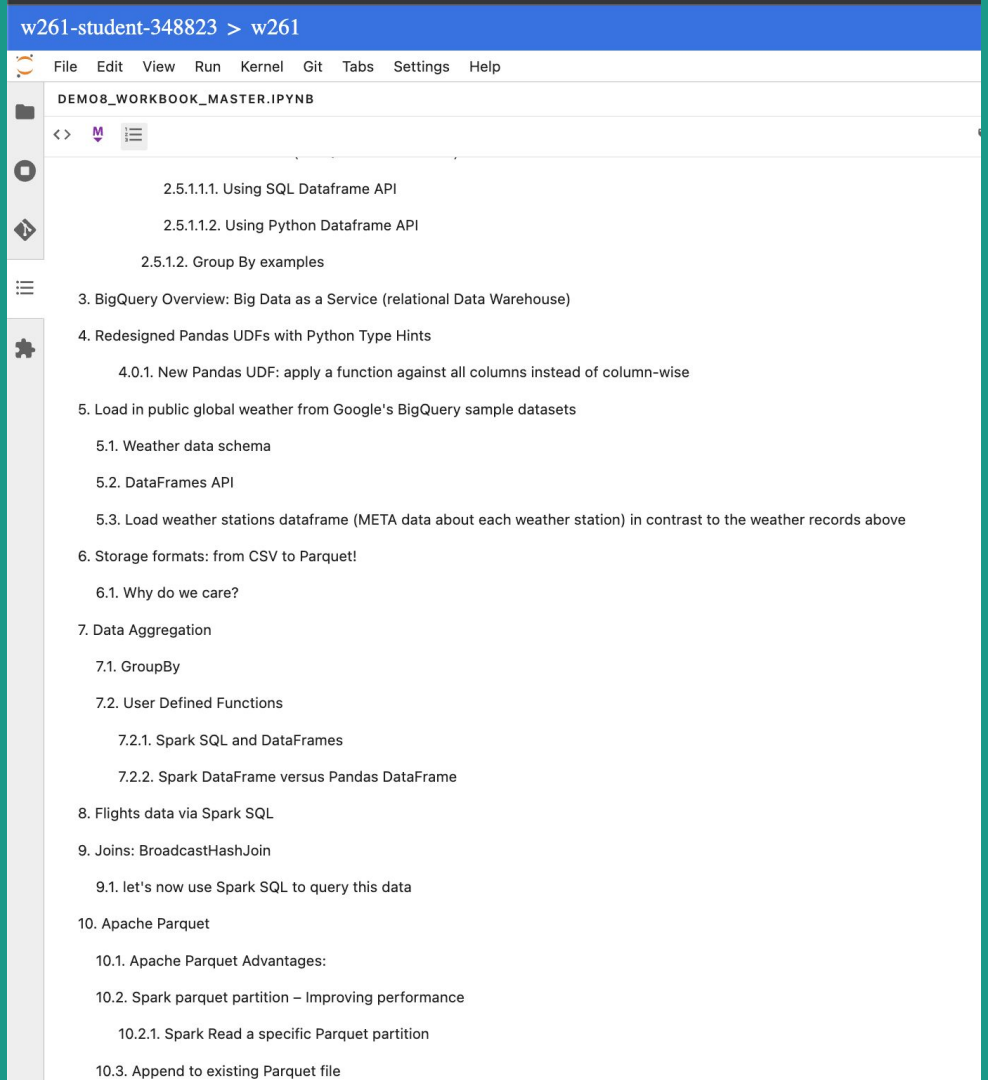
- **Team formation (Teams of four (4) people): due end of week 9 (Sunday)**
  - Please form teams by yourselves (we prefer you do this by yourselves) by the end of week 9. If you don't have a team by **Monday of week 10,** we will randomly assign you to a team. If you need help getting assigned to a group, please let the instructors/TAs know.
  - Once teams are formed, everyone needs to fill in their team details on Canvas under the **"People"** section. You can join a group by adding your name to one of the final project groups in Canvas. Click the **"People"** TAB on the left-hand navigation menu, and select the **"Project Groups"** tab to view available groups. Each group is limited to four students. To learn more about creating groups on Canvas or the process for joining a group, please see here.
  - Please keep the team prefix (such as *Group-02-* as is but feel free to add a team suffix such as *RootFinders* to yield a team name of *Group-02-RootFinders*



Spark 1.0, 2.0 and 3.0+

- Vectorized column-based memory layout
- Wholestage code generation
- The Catalyst optimizer takes a computational query and converts it into an execution plan. It goes through four transformational phases:
  - 1. Analysis
  - 2. Logical optimization
  - 3. Physical planning
  - 4. Code generation (Project Tungsten)
- Pushdown predicates
- Joins
- Python dataframe API versus SQL API

Notebook


w261-student-348823 > w261
File Edit View Run Kernel Git Tabs Settings Help
DEMO8_WORKBOOK_MASTER.IPYNB
2.5.1.1.1. Using SQL Dataframe API
2.5.1.1.2. Using Python Dataframe API
2.5.1.2. Group By examples
3. BigQuery Overview: Big Data as a Service (relational Data Warehouse)
4. Redesigned Pandas UDFs with Python Type Hints
4.0.1. New Pandas UDF: apply a function against all columns instead of column-wise
5. Load in public global weather from Google's BigQuery sample datasets
5.1. Weather data schema
5.2. DataFrames API
5.3. Load weather stations dataframe (META data about each weather station) in contrast to the weather records above
6. Storage formats: from CSV to Parquet!
6.1. Why do we care?
7. Data Aggregation
7.1. GroupBy
7.2. User Defined Functions
7.2.1. Spark SQL and DataFrames
7.2.2. Spark DataFrame versus Pandas DataFrame
8. Flights data via Spark SQL
9. Joins: BroadcastHashJoin
9.1. let's now use Spark SQL to query this data
10. Apache Parquet
10.1. Apache Parquet Advantages:
10.2. Spark parquet partition – Improving performance
10.2.1. Spark Read a specific Parquet partition
10.3. Append to existing Parquet file

Wholestage code generation and vectorized column-based memory layout

- What's more, you can seamlessly move between DataFrames or Datasets and RDDs at will using a simple API method call, df.rdd. (Note, however, that this does have a cost and should be avoided unless necessary.) After all, DataFrames and Datasets are built on top of RDDs, and they get decomposed to compact RDD code during wholestage code generation.
- At a programmatic level, Spark SQL allows developers to issue ANSI SQL:2003–compatible queries on structured data with a schema.
-
- Observing how a DataFrame is stored across one executor on a local host, as displayed in Figure 7-4, we can see they all fit in memory (recall that at a low level Data Frames are backed by RDDs).
-
- Spark 2.x introduced the second-generation Tungsten engine, featuring wholestage code generation and vectorized column-based memory layout. Built on ideas and techniques from modern compilers, this new version also capitalized on modern CPU and cache architectures for fast parallel data access with the "single instruction, multiple data" (SIMD) approach.

Catalyst optimizer and Project Tungsten

At the core of the Spark SQL engine are the Catalyst optimizer and Project Tungsten. Together, these support the high-level DataFrame and Dataset APIs and SQL queries.

We'll talk more about Tungsten in Chapter 6 [Learning Spark, Second Edition]; for now, let's take a closer look at the optimizer.

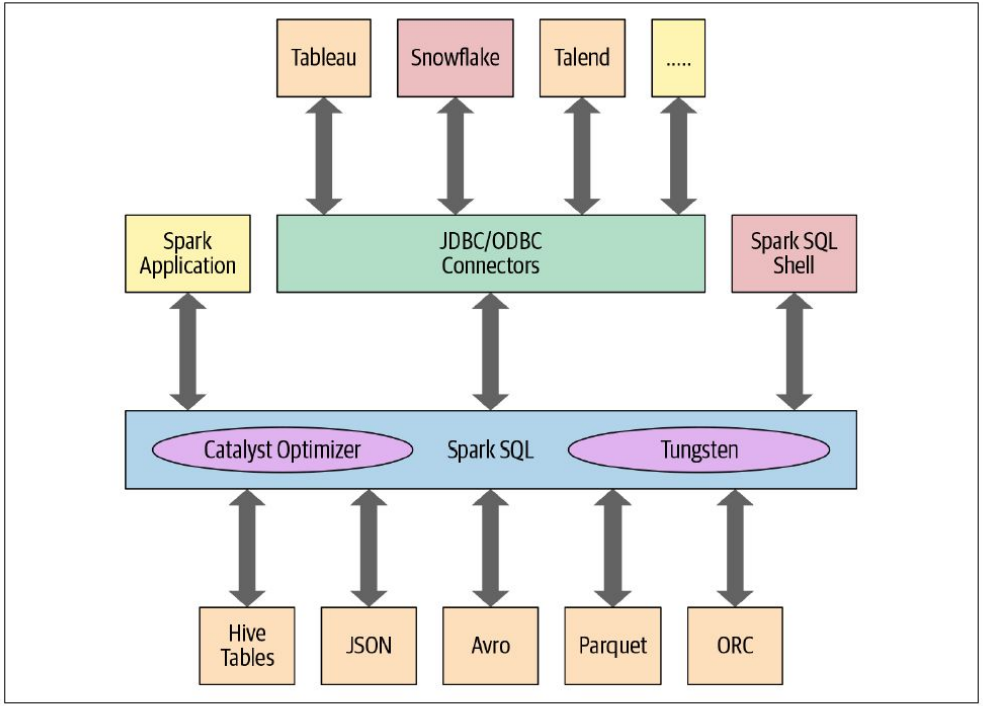


Figure 3-3. Spark SQL and its stack

Src: [Learning Spark, Second Edition]

The Catalyst Optimizer

- The Catalyst optimizer takes a computational query and converts it into an execution plan. It goes through four transformational phases, as shown in Figure 3-4:
  - 1. Analysis
  - 2. Logical optimization
  - 3. Physical planning
  - 4. Code generation (Project Tungsten)

Catalyst optimizer

```
// In Scala
// Users DataFrame read from a Parquet table
val usersDF  = ...
// Events DataFrame read from a Parquet table
val eventsDF = ...
// Join two DataFrames
val joinedDF = users
  .join(events, users("id") === events("uid"))
  .filter(events("date") > "2015-01-01")
```

After going through an initial analysis phase, the query plan is transformed and rearranged by the Catalyst optimizer as shown in Figure 3-5.

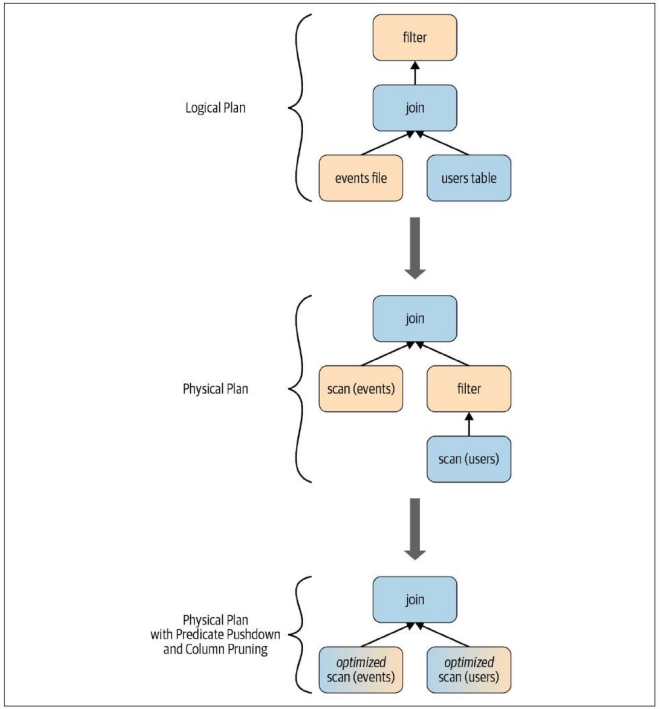


Figure 3-5. An example of a specific query transformation

Vectorized UDFs via pandas_udf

Besides using Spark DataFrame API, users can also develop functions in pure Python using Pandas API but also take advantage of Spark parallel processing.

Bye-bye loops! Hello vectorization

- UDF is an abbreviation of “user defined function” in Spark.
  - Pandas UDF was introduced in Spark 2.3 and continues to be a useful technique for optimizing Spark jobs.
  - The idea of Pandas UDF is to narrow the gap between processing big data using Spark and developing in Python.
  - Pandas UDF is there to apply Python functions directly on Spark DataFrame which allows engineers or scientists to develop in pure Python and still take advantage of Spark’s parallel processing features at the same time.
- Generally, all Spark-native functions applied on Spark DataFrame are vectorized, which takes advantage of Spark’s parallel processing.
  - Although Spark already supports plenty of mainstream functions which cover most of use cases, we might still want to build customized functions to transform data for migration existing scripts or for developers who are not familiar with Spark.

https://medium.com/codex/say-goodbye-to-loops-in-python-and-welcome-vectorization-e4df66615a52

User defined function example

For example, let’s say we need a function to hash columns. Spark support sha or md5 function natively, but UDF allows us to reuse the same hash and salt method on multiple columns. In addition, UDF allows the user to develop more complicated hash functions in pure Python or reuse the same function they have already developed. By converting UDF in Pandas UDF, the Pandas UDF will also process the column parallelly, which provides better performance than a UDF.

Learn more about Pandas API on Spark

Spark native functions are always the best

- Spark native functions will always have the best performance overall.
- However, when we have to develop some transformation function that is not supported by Spark, or it's easier for developers to develop in pure Python, using a Pandas UDF can optimize Spark jobs performance.

Cmd 3

```
1  from pyspark.sql.functions import sha2, col, lit, concat
2  sqlContext.clearCache()
3
4  df_sparkhash = df \
5      .withColumn("hashed",
   sha2(concat(col("target").cast("string"),
   lit(uuid.uuid4().hex)), 512)).cache()
```

Spark native functions: sha2
In 0.28 seconds

▸ df_sparkhash: pyspark.sql.dataframe.DataFrame = [state: string, feature1: double ... 4 more fields]

Command took 0.28 seconds -- by xuminx@nealanalytics.com [redacted]

Cmd 4

```
1  df_sparkhash.count()
```

▸ (2) Spark Jobs

BEST: 11 Seconds

Out[20]: 10000000

Command took 11.11 seconds -- by xuminx@nealanalytics.com [redacted]

Old-fashioned UDF

Spark native functions sha2 : takes 11 seconds
Old fashioned UDF: takes 32 seconds

Slow: row by row

```
1  import hashlib, uuid
2  sqlContext.clearCache()
3
4  @udf('string')
5  def hash_pure_python(c):
6  #   salt = uuid.uuid4().hex
7    hashed_string = hashlib.sha512((str(c) +
   uuid.uuid4().hex).encode("utf-8")).hexdigest()
8    return hashed_string
9
10 df_purepyhash = df.withColumn('hashed',
   hash_pure_python(df.target)).cache()
```

▸ df_purepyhash: pyspark.sql.dataframe.DataFrame = [state: string, feature1: double ... 4 more fields]

Command took 0.23 seconds -- by xuminx@nealanalytics.com a[redacted]

Cmd 7

```
1  df_purepyhash.count()
```

▸ (2) Spark Jobs

Out[23]: 10000000

Command took 31.84 seconds -- by xuminx@nealanalytics.com a[redacted]

[Example adapted from here]

Vectorized UDFs: on the RHS

```
1  import hashlib, uuid
2  sqlContext.clearCache()
3
4  @udf('string')
5  def hash_pure_python(c):
6  #   salt = uuid.uuid4().hex
7    hashed_string = hashlib.sha512((str(c) +
   uuid.uuid4().hex).encode("utf-8")).hexdigest()
8    return hashed_string
9
10 df_purepyhash = df.withColumn('hashed',
   hash_pure_python(df.target)).cache()
```

Spark native functions sha2 : takes 11 seconds
Old fashioned UDF: takes 32 seconds
Vectorized UDFs: takes 24 seconds

Cmd 7

```
1  df_purepyhash.count()
```

▸ (2) Spark Jobs

Out[23]: 10000000

Command took 31.84 seconds -- by xuminx@nealanalytics.com a

```
1  from pyspark.sql.functions import pandas_udf, PandasUDFType
2  import pandas as pd
3  sqlContext.clearCache()
4
5  # Use pandas_udf to define a Pandas UDF
6  @pandas_udf('string')
7  # Input/output are both a pandas.Series of doubles
8
9  def hash_pandas_udf(c: pd.Series) -> pd.Series
10
11 #   hashed_string = hashlib.sha512((c.map(lamb
   str(x).encode('utf-8')))).hexdigest()
12   hashed_string = c.map(lambda x: hashlib.sha512((str(x) +
   uuid.uuid4().hex).encode('utf-8')).hexdigest())
13   return hashed_string
14
15 df_pandasudfhash = df.withColumn('sha',
   hash_pandas_udf(col("target"))).cache()
```

Panda's series map()

▸ df_pandasudfhash: pyspark.sql.dataframe.DataFrame = [state: string, feature1: double ... 4 more fields]

Command took 0.23 seconds -- by xuminx@nealanalytics.com a

Cmd 10

```
1  df_pandasudfhash.count()
```

▸ (2) Spark Jobs

Out[27]: 10000000

Command took 24.39 seconds -- by xuminx@nealanalytics.com a

[Example adapted from here]

User defi

For example, let's sa ; to
reuse the same hash
hash functions in pu F,
the Pandas UDF will

Cmd 3

```
1  from pyspark.sql.functions import sha2, col, lit, concat
2  sqlContext.clearCache()
3
4  df_sparkhash = df \
5      .withColumn("hashed",
   sha2(concat(col("target").cast("string"),
   lit(uuid.uuid4().hex)), 512)).cache()
```

▸ df_sparkhash: pyspark.sql.dataframe.DataFrame = [state: string, feature1: double ... 4 more fields]

Command took 0.28 seconds -- by xuminx@nealanalytics.com

Cmd 4

```
1  df_sparkhash.count()
```

▸ (2) Spark Jobs

Out[20]: 10000000

Command took 11.11 seconds -- by xuminx@nealanalytics.com

[Example adapted from here]

User defined function example

For example,
reuse the sar
hash functio
the Pandas U

Native Spark Function

```
Cmd 3

from pyspark.sql.functions import sha2, col, lit, concat
sqlContext.clearCache()

df_sparkhash = df \
    .withColumn("hashed",
sha2(concat(col("target").cast("string"),
lit(uuid.uuid4().hex)), 512)).cache()

▸ df_sparkhash: pyspark.sql.dataframe.DataFrame = [state: string, feature1: double ... 4 more fields]

Command took 0.28 seconds -- by xuminx@nealanalytics.com at [redacted]

Cmd 4

df_sparkhash.count()

▸ (2) Spark Jobs

Out[20]: 10000000

Command took 11.11 seconds -- by xuminx@nealanalytics.com at [redacted]
```

BEST: 11 Seconds

Spark Native Function:

- **11.11 seconds**
- Always the fastest if functions are supported

UDF

```
import hashlib, uuid
sqlContext.clearCache()

@udf('string')
def hash_pure_python(c):
#   salt = uuid.uuid4().hex
  hashed_string = hashlib.sha512((str(c) +
uuid.uuid4().hex).encode("utf-8")).hexdigest()
  return hashed_string

df_purepyhash = df.withColumn('hashed',
hash_pure_python(df.target)).cache()

▸ df_purepyhash: pyspark.sql.dataframe.DataFrame = [state: string, feature1: double ... 4 more fields]

Command took 0.23 seconds -- by xuminx@nealanalytics.com at [redacted]

Cmd 7

df_purepyhash.count()

▸ (2) Spark Jobs

Out[23]: 10000000

Command took 31.84 seconds -- by xuminx@nealanalytics.com at [redacted]
```

UDF:

- **31.84 seconds**
- Easy to migrate. Much slower.

PandasUDF

```
from pyspark.sql.functions import pandas_udf, PandasUDFType
import pandas as pd
sqlContext.clearCache()

# Use pandas_udf to define a Pandas UDF
@pandas_udf('string')
# Input/output are both a pandas.Series of doubles

def hash_pandas_udf(c: pd.Series) -> pd.Series:

#   hashed_string = hashlib.sha512((c.map(lambda x:
str(x).encode('utf-8')))).hexdigest()
  hashed_string = c.map(lambda x: hashlib.sha512((str(x) +
uuid.uuid4().hex).encode('utf-8')).hexdigest())
  return hashed_string

df_pandasudfhash = df.withColumn('sha',
hash_pandas_udf(col("target"))).cache()

▸ df_pandasudfhash: pyspark.sql.dataframe.DataFrame = [state: string, feature1: double ... 4 more fields]

Command took 0.23 seconds -- by xuminx@nealanalytics.com at [redacted]

Cmd 10

df_pandasudfhash.count()

▸ (2) Spark Jobs

Out[27]: 10000000

Command took 24.39 seconds -- by xuminx@nealanalytics.com at [redacted]
```

PandasUDF:

- **24.39 seconds**
- Faster than UDF

Spark native functions will always have the best performance overall. However, when we have to develop some transformation function that is not supported by Spark, or it's easier for developers to develop in pure Python, using a Pandas UDF can optimize Spark jobs performance.

UDFs

UDFs can be implemented in Python, Scala, Java and R, and UDAFs in Scala and Java. When using UDFs with PySpark, data serialization costs must be factored in, and the two strategies discussed above to address this should be considered.

UDFs, UDAFs, and Datasets all provide ways to intermix arbitrary code with Spark SQL.

We recommend you write your UDFs in Java or Scala-the small amount of time it atkes to write the function in Java or Scala will always yield significant speed ups. And you can still use the function from within python!

(p.113 - Spark, The Definitive Guide)

Scala UDF in python example:

- https://medium.com/wbaa/using-scala-udfs-in-pyspark-b70033dd69b9
- https://github.com/johnmuller87/spark-udf

Pandas UDFs: low-overhead, high-performance UDFs entirely in Python

Over the past few years, Python has become the default language for data scientists.

- Packages such as pandas, numpy, statsmodel, and scikit-learn have gained great adoption and become the mainstream toolkits. At the same time, Apache Spark has become the de facto standard in processing big data. To enable data scientists to leverage the value of big data, Spark added a Python API in version 0.7, with support for user-defined functions. These user-defined functions operate one-row-at-a-time, and thus suffer from high serialization and invocation overhead. As a result, many data pipelines define UDFs in Java and Scala and then invoke them from Python.

Pandas UDFs built on top of Apache Arrow bring you the best of both worlds—the ability to define low-overhead, high-performance UDFs entirely in Python.

User-Defined Functions

While Apache Spark has a plethora of built-in functions, the flexibility of Spark allows for data engineers and data scientists to define their own functions too. These are known as user-defined functions (UDFs).

Evaluation order and null checking in Spark SQL

Spark SQL (this includes SQL, the DataFrame API, and the Dataset API) does not guarantee the order of evaluation of subexpressions. For example, the following query does not guarantee that the `s is NOT NULL` clause is executed prior to the `strlen(s) > 1` clause:

```
spark.sql("SELECT s FROM test1 WHERE s IS NOT NULL AND strlen(s) > 1")
```

Therefore, to perform proper `null` checking, it is recommended that you do the following:

1. Make the UDF itself `null`-aware and do `null` checking inside the UDF.
2. Use `IF` or `CASE WHEN` expressions to do the `null` check and invoke the UDF in a conditional branch.

Following up to my Scaling Python for Data Science using Spark post where I mentioned Spark 2.3 introducing Vectorized UDFs, I'm using the same Data (from NYC yellow cabs) with this code:

```
from pyspark.sql import functions as F
from pyspark.sql.functions import pandas_udf, PandasUDFType
from pyspark.sql.types import *
import pandas as pd

df = spark.read\
  .option("header", "true")\
  .option("inferSchema", "true")\
  .csv("yellow_tripdata_2017-06.csv")

def timestamp_to_epoch(t):
 return t.dt.strftime("%s").apply(str) # <-- pandas.Series calls

f_timestamp_copy = pandas_udf(timestamp_to_epoch, returnType=StringType())
df = df.withColumn("timestamp_copy", f_timestamp_copy(F.col("tpep_pickup_datetime")))
df.select('timestamp_copy').distinct().count()

# s = pd.Series({'ds': pd.Timestamp('2018-03-03 04:31:19')})
# timestamp_to_epoch(s)
## ds 1520080279
## dtype: object
```

Pandas Scalar (the default; as opposed to grouped map) UDFs operate on pandas.Series objects for both input and output, hence the .dt call chain as opposed to directly calling strftime on a python datetime object. The entire functionality is dependent on using PyArrow (>= 0.8.0).

UDFs: row by row to vectorized execution

You can now use Spark SQL to execute either of these `cubed()` functions:

```
// In Scala/Python
// Query the cubed UDF
spark.sql("SELECT id, cubed(id) AS id_cubed FROM udf_test").show()
```

```
+---+--------+
| id|id_cubed|
+---+--------+
|  1|       1|
|  2|       8|
|  3|      27|
|  4|      64|
|  5|     125|
|  6|     216|
|  7|     343|
|  8|     512|
+---+--------+
```

Evaluation order and null checking in Spark SQL

Spark SQL (this includes SQL, the DataFrame API, and the Dataset API) does not guarantee the order of evaluation of subexpressions. For example, the following query does not guarantee that the `s is NOT NULL` clause is executed prior to the `strlen(s) > 1` clause:

```
spark.sql("SELECT s FROM test1 WHERE s IS NOT NULL AND strlen(s) > 1")
```

Therefore, to perform proper `null` checking, it is recommended that you do the following:

1. Make the UDF itself `null`-aware and do `null` checking inside the UDF.
2. Use `IF` or `CASE WHEN` expressions to do the `null` check and invoke the UDF in a conditional branch.

Speeding up and distributing PySpark UDFs with Pandas UDFs

One of the previous prevailing issues with using PySpark UDFs was that they had slower performance than Scala UDFs. This was because the PySpark UDFs required data movement between the JVM and Python, which was quite expensive. To resolve this problem, Pandas UDFs (also known as vectorized UDFs) were introduced as part of Apache Spark 2.3. A Pandas UDF uses Apache Arrow to transfer data and Pandas to work with the data. You define a Pandas UDF using the keyword `pandas_udf` as the decorator, or to wrap the function itself. Once the data is in Apache Arrow format, there is no longer the need to serialize/pickle the data as it is already in a format consumable by the Python process. Instead of operating on individual inputs row by row, you are operating on a Pandas Series or DataFrame (i.e., vectorized execution).

From Apache Spark 3.0 with Python 3.6 and above, Pandas UDFs were split into two API categories: Pandas UDFs and Pandas Function APIs.

Pandas UDFs

With Apache Spark 3.0, Pandas UDFs infer the Pandas UDF type from Python type hints in Pandas UDFs such as `pandas.Series`, `pandas.DataFrame`, `Tuple`, and `Iterator`. Previously you needed to manually define and specify each Pandas UDF type. Currently, the supported cases of Python type hints in Pandas UDFs are Series to Series, Iterator of Series to Iterator of Series, Iterator of Multiple Series to Iterator of Series, and Series to Scalar (a single value).

Pandas Function APIs

Pandas Function APIs allow you to directly apply a local Python function to a PySpark DataFrame where both the input and output are Pandas instances. For Spark 3.0, the supported Pandas Function APIs are grouped map, map, cogrouped map.

For more information, refer to "Redesigned Pandas UDFs with Python Type Hints" on page 354 in Chapter 12.

The following is an example of a scalar Pandas UDF for Spark 3.0:[2]

```
# In Python
# Import pandas
import pandas as pd

# Import various pyspark SQL functions including pandas_udf
from pyspark.sql.functions import col, pandas_udf
from pyspark.sql.types import LongType

# Declare the cubed function
def cubed(a: pd.Series) -> pd.Series:
    return a * a * a

# Create the pandas UDF for the cubed function
cubed_udf = pandas_udf(cubed, returnType=LongType())
```

The preceding code snippet declares a function called `cubed()` that performs a *cubed* operation. This is a regular Pandas function with the additional `cubed_udf = pandas_udf()` call to create our Pandas UDF.

2 Note there are slight differences when working with Pandas UDFs between Spark 2.3, 2.4, and 3.0.

Let's start with a simple Pandas Series (as defined for `x`) and then apply the local function `cubed()` for the cubed calculation:

```
# Create a Pandas Series
x = pd.Series([1, 2, 3])

# The function for a pandas_udf executed with local Pandas data
print(cubed(x))
```

The output is as follows:

```
0     1
1     8
2    27
dtype: int64
```

Now let's switch to a Spark DataFrame. We can execute this function as a Spark vectorized UDF as follows:

```
# Create a Spark DataFrame, 'spark' is an existing SparkSession
df = spark.range(1, 4)

# Execute function as a Spark vectorized UDF
df.select("id", cubed_udf(col("id"))).show()
```

Here's the output:

```
+---+---------+
| id|cubed(id)|
+---+---------+
|  1|        1|
|  2|        8|
|  3|       27|
+---+---------+
```

As opposed to a local function, using a vectorized UDF will result in the execution of Spark jobs; the previous local function is a Pandas function executed only on the Spark driver. This becomes more apparent when viewing the Spark UI for one of the stages of this `pandas_udf` function (Figure 5-1).

For a deeper dive into Pandas UDFs, refer to pandas user-defined functions documentation.

Using row-at-a-time UDFs:

```
from pyspark.sql.functions import udf

# Use udf to define a row-at-a-time udf
@udf('double')
# Input/output are both a single double value
def plus_one(v):
      return v + 1

df.withColumn('v2', plus_one(df.v))
```

In the row-at-a-time version, the user-defined function takes a double “v” and returns the result of “v + 1” as a double.

Using Pandas UDFs:

```
from pyspark.sql.functions import pandas_udf, PandasUDFType

# Use pandas_udf to define a Pandas UDF
@pandas_udf('double', PandasUDFType.SCALAR)
# Input/output are both a pandas.Series of doubles

def pandas_plus_one(v):
    return v + 1

df.withColumn('v2', pandas_plus_one(df.v))
```

In the Pandas version, the user-defined function takes a `pandas.Series` “v” and returns the result of “v + 1” as a `pandas.Series`.
Because “v + 1” is vectorized on `pandas.Series`, the Pandas version is much faster than the row-at-a-time version.

https://databricks.com/blog/2017/10/30/introducing-vectorized-udfs-for-pyspark.html

Grouped Map Pandas UDFs:

This example shows a simple use of grouped map Pandas UDFs: subtracting mean from each value in the group.

we subtract mean of v from each value of v for each group

```
@pandas_udf(df.schema, PandasUDFType.GROUPED_MAP)
# Input/output are both a pandas.DataFrame
def subtract_mean(pdf):
    return pdf.assign(v=pdf.v - pdf.v.mean())

df.groupby('id').apply(subtract_mean)
```

Debug your UDFs locally. See here for details.

In this example, we subtract mean of v from each value of v for each group. The grouping semantics is defined by the "groupby" function, i.e, each input `pandas.DataFrame` to the user-defined function has the same "id" value. The input and output schema of this user-defined function are the same, so we pass "df.schema" to the decorator `pandas_udf` for specifying the schema.

https://databricks.com/blog/2017/10/30/introducing-vectorized-udfs-for-pyspark.html

Storage formats: A Quick review

8: Data Systems and Pipelines

46m Total Video Time

Course Content

8.1 Weekly Introduction 8

8.2 Types of Data

8.3 Evolution of Data-Processing Systems

8.4 Analytical Data Systems

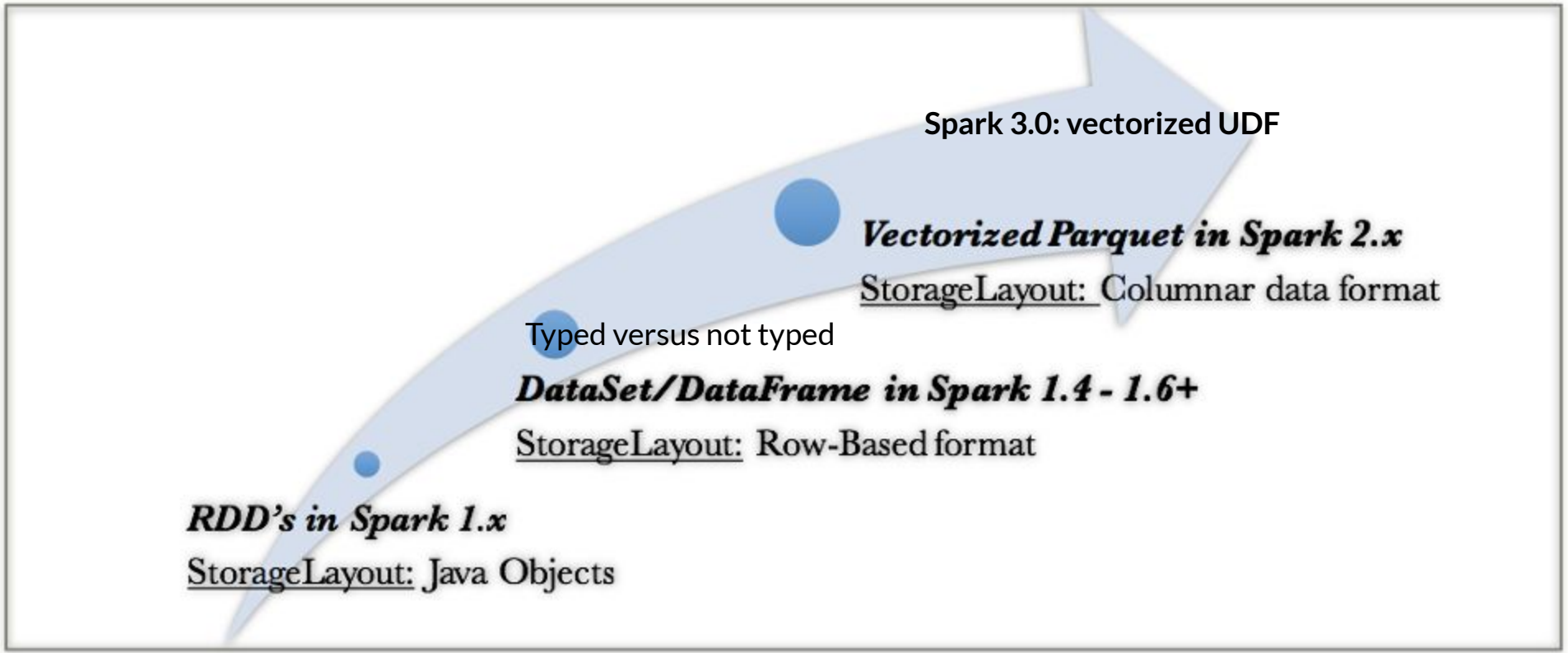
8.5 Data Persistence and Storage

8.6 Working With Data: SQL and MapReduce

8.7 Summary

Data Persistence & Storage

Evolution of data layout storage-format's in Spark:



OLTP / OLAP Workflows

RealTime

Online Transaction Processing (OLTP)

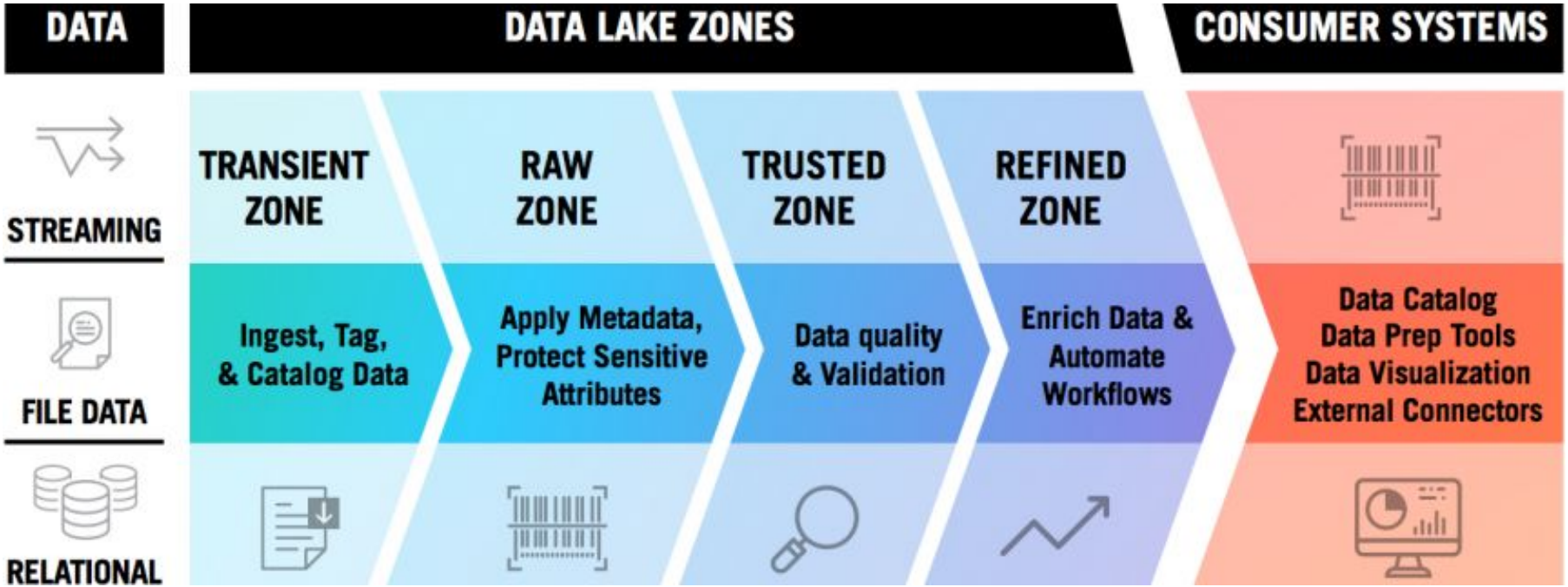
- Larger numbers of short queries / transactions
- More processing than analysis focused
- Geared towards record (row) based processing than column based
- More frequent data updates / deletes (transactions)

Online Analytical Processing (OLAP)

- More analysis than processing focused
- Geared towards column-based data analytics processing
- Less frequent transactions
- More analytic complexity per query

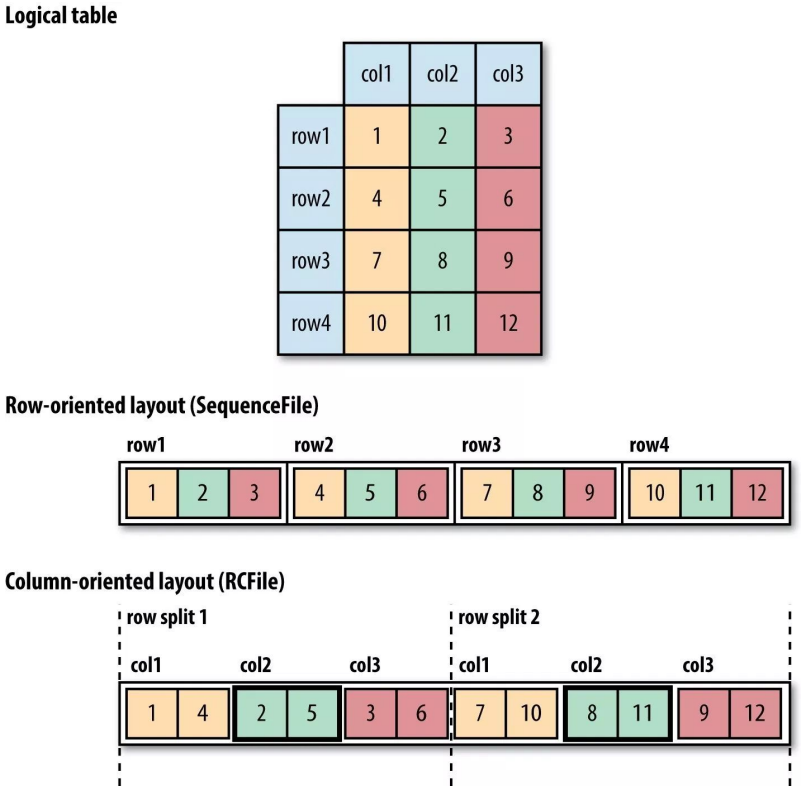
Insight: Data access patterns should inform the selection of file formats

Data Lakes - Governance Models



On-disk/In-memory Row vs Column Oriented data Formats

- In ACID systems we want atomicity and act on individual rows in a transaction
  - ACID is an acronym that stands for atomicity, consistency, isolation, and durability. Together, these ACID properties ensure that a set of database operations (grouped together in a transaction) leave the database in a valid state even in the event of unexpected errors.
  - This is good for transactional workloads
- What if we want to do aggregates?
  - Column oriented data allows us to aggregate on columns easily
- Choose based on your use case
- Orientation is based on serialization in HDFS/Spark while orientation in databases is dependent on the architecture



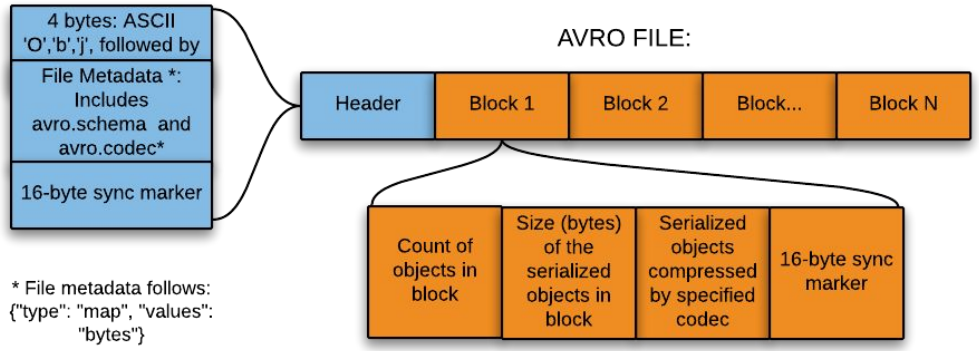
CSV

```
Generic,Storage Type,Hierarchy Type,Attributes Header,comment,bso data storage,aso data storage,
    two pass calculation,aso dimension formula,consolidation type,uda,parent,alias:Default,alias:English
Customers,SPARSE,STORED,,,LABELONLY,STOREDATA,N,,,UDA,,alias:Default,alias:English
NoCustomer,SPARSE,Disabled,,,StoreData,StoreData,N,,+,,,No Customer,No Customer
AllCustomers,SPARSE,Disabled,,,StoreData,StoreData,N,,+,,,All Customers,All Customers
Big Box,SPARSE,,,,StoreData,StoreData,N,,+,,AllCustomers,,
BB100,SPARSE,,,,StoreData,StoreData,N,,+,,Big Box,Q Mart,Q Mart
BB200,SPARSE,,,,StoreData,StoreData,N,,+,,Big Box,Bike Depot,Bike Depot
BB300,SPARSE,,,,StoreData,StoreData,N,,+,,Big Box,Mountain Adventures,Mountain Adventures
Specialty Retailers,SPARSE,,,,StoreData,StoreData,N,,+,,AllCustomers,,
SR100,SPARSE,,,,StoreData,StoreData,N,,+,,Specialty Retailers,Bobs Bikes,Bobs Bikes
SR200,SPARSE,,,,StoreData,StoreData,N,,+,,Specialty Retailers,Rose Town Bikes,Rose Town Bikes
SR300,SPARSE,,,,StoreData,StoreData,N,,+,,Specialty Retailers,The Cyclery,The Cyclery
Webstore,SPARSE,,,,StoreData,StoreData,N,,+,,AllCustomers,,
```

- Text files are human readable and easily parsable
- Text files are slow to read and write
- Data size is relatively bulky and not as efficient to query
- No metadata is stored in the text files so we need to know the structure of the fields
- Text files are not splittable after compression
- Limited support for schema evolution: new fields can only be appended at the end of the records and existing fields can never be removed

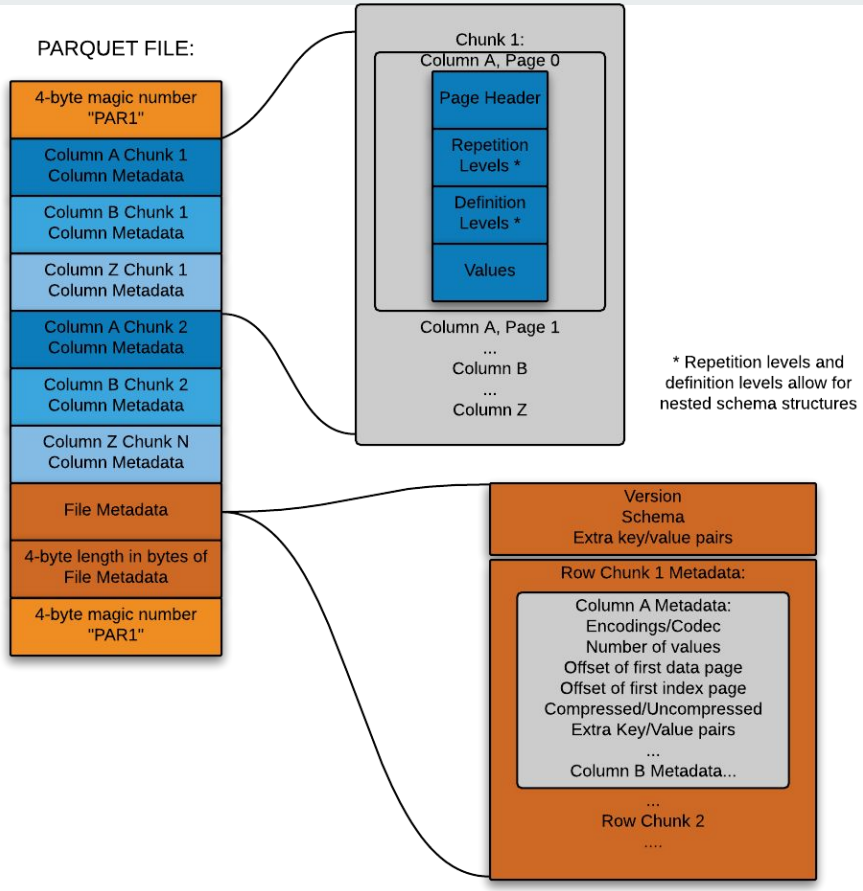
Avro: Av-roW

- Row-based
- Interoperability: can serialize into Avro/Binary or Avro/Json
- Compact binary form
- Extensible schema language
- Untagged data
- Dynamic typing
- Supports block compression
- Splittable
- Best compatibility for evolving data schemas



Parquet

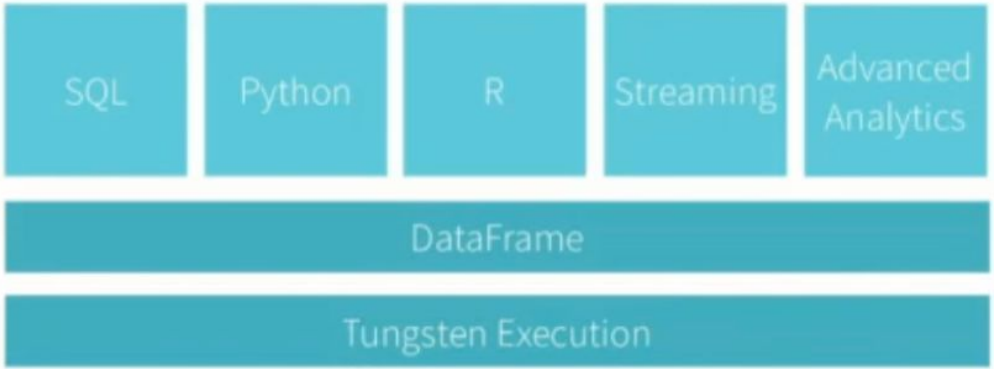
- Column-oriented
- Efficient in terms of disk I/O and memory
- Efficiently encoding of nested structures and sparsely populated data
- Provides extensible support for per-column encodings
- Provides extensibility of storing multiple types of data in column data
- Offers better write performance by storing metadata at the end of the file
- Records in columns are homogeneous so it's easier to apply encoding schemes
- Non-appendable due to metadata



Spark Format Showdown		File Format			
		CSV	JSON	Avro	Parquet
Attribute	Columnar	No	No		Yes
	Compressable	Yes	Yes		Yes
	Splittable	Yes*	Yes**		Yes
	Human Readable	Yes	Yes		No
	Nestable	No	Yes		Yes
	Complex Data Structures	No	Yes		Yes
	Default Schema: Named columns	Manual	Automatic (full read)		Automatic (instant)
	Default Schema: Data Types	Manual (full read)	Automatic (full read)		Automatic (instant)

Goals of Project Tungsten

Substantially improve the memory and CPU efficiency of Spark backend execution and push performance closer to the limits of modern hardware.



Note the focus on “execution” not “optimizer”: relatively easy to pick broadcast hash join that is 1000X faster than Cartesian join, but hard to optimize broadcast join to be an order of magnitude faster.

Parquet format confers some considerable performance advantages

- *Spark introduced support for columnar data using a new technique called Vectorization in spark 2.x to better leverage the advancements of Modern CPU's and hardware like loop-unrolling, SIMD (Single instruction, multiple data) etc.*
- *Parquet supports specifying different compression schemes for each column, as well as column-specific encoding schemes such as run-length encoding, dictionary encoding, and delta encoding.*
- Leads to Zero serialiation cost when reading from disk (vectorized parquet reader)
- Improved compatibility with other ML frameworks

https://spoddutur.github.io/spark-notes/deep_dive_into_storage_formats.html

Parquet file format confers some considerable performance advantages

- *Specifically, it supports most of the common database types (int, double, string, etc.), along with arrays and records, including nested types.*
- *Significantly, it is a columnar file format, meaning that values for a particular column from many records are stored contiguously on disk (see Figure 9-2). This physical data layout allows for far more efficient data encoding/compression, and significantly reduces query times by minimizing the amount of data that must be read/deserialized*
- *Parquet supports specifying different compression schemes for each column, as well as column-specific encoding schemes such as run-length encoding, dictionary encoding, and delta encoding.*
-

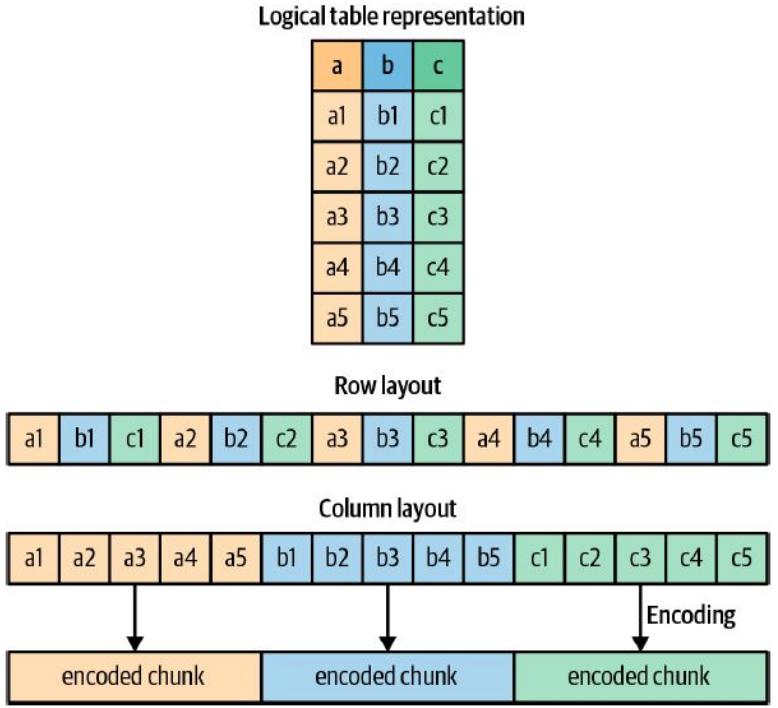


Figure 9-2. Differences between a row-major and column-major data layout

Src: Chapter 9: Advanced Analytics with PySpark Book

Click to add titl

Click to add text

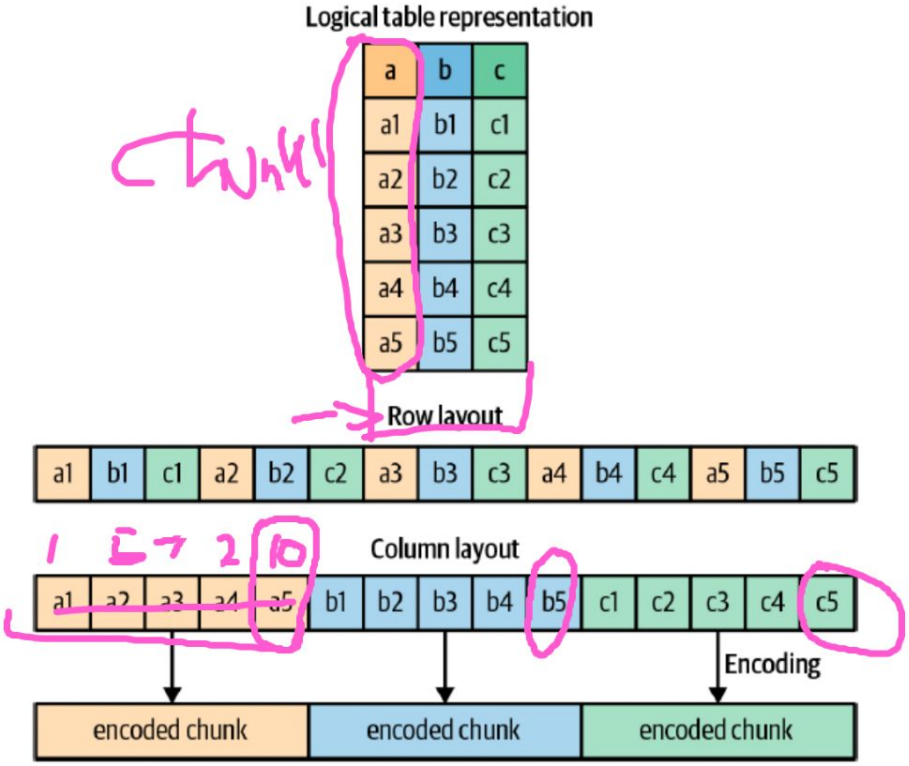


Figure 9-2. Differences between a row-major and column-major data layout

Src: Chapter 9: Advanced Analytics with PySpark Book

Wholestage code generation

- Vectorized column-based memory layout
- **Wholestage code generation**
  - **The Catalyst optimizer takes a computational query and converts it into an execution plan. It goes through four transformational phases:**
    - **1. Analysis**
    - **2. Logical optimization**
    - **3. Physical planning**
    - **4. Code generation (Project Tungsten)**

Wholestage code generation and vectorized column-based memory layout

- What's more, you can seamlessly move between DataFrames or Datasets and RDDs at will using a simple API method call, df.rdd. (Note, however, that this does have a cost and should be avoided unless necessary.) After all, DataFrames and Datasets are built on top of RDDs, and they get decomposed to compact RDD code during wholestage code generation.
- At a programmatic level, Spark SQL allows developers to issue ANSI SQL:2003–compatible queries on structured data with a schema.
-
- Observing how a DataFrame is stored across one executor on a local host, as displayed in Figure 7-4, we can see they all fit in memory (recall that at a low level Data Frames are backed by RDDs).
-
- Spark 2.x introduced the second-generation Tungsten engine, featuring wholestage code generation and vectorized column-based memory layout. Built on ideas and techniques from modern compilers, this new version also capitalized on modern CPU and cache architectures for fast parallel data access with the "single instruction, multiple data" (SIMD) approach.

Catalyst optimizer and Project Tungsten

At the core of the Spark SQL engine are the Catalyst optimizer and Project Tungsten. Together, these support the high-level DataFrame and Dataset APIs and SQL queries.

We'll talk more about Tungsten in Chapter 6 [Learning Spark, Second Edition]; for now, let's take a closer look at the optimizer.

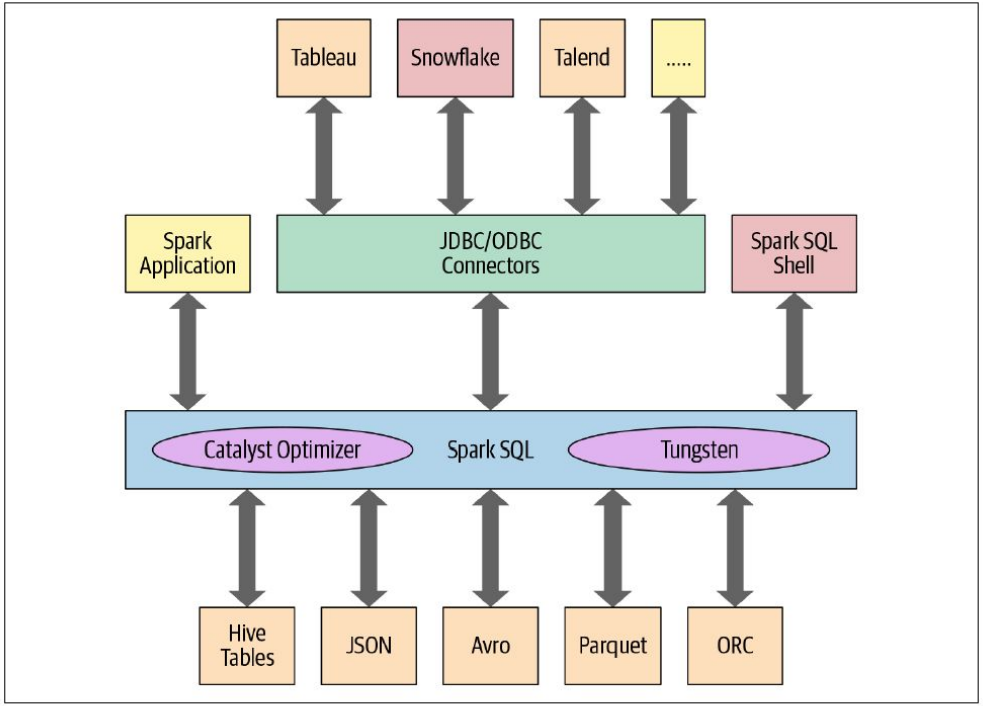


Figure 3-3. Spark SQL and its stack

Src: [Learning Spark, Second Edition]

The Catalyst Optimizer

- The Catalyst optimizer takes a computational query and converts it into an execution plan. It goes through four transformational phases, as shown in Figure 3-4:
  - 1. Analysis
  - 2. Logical optimization
  - 3. Physical planning
  - 4. Code generation (Project Tungsten)

Catalyst optimizer

```
// In Scala
// Users DataFrame read from a Parquet table
val usersDF  = ...
// Events DataFrame read from a Parquet table
val eventsDF = ...
// Join two DataFrames
val joinedDF = users
  .join(events, users("id") === events("uid"))
  .filter(events("date") > "2015-01-01")
```

After going through an initial analysis phase, the query plan is transformed and rearranged by the Catalyst optimizer as shown in Figure 3-5.

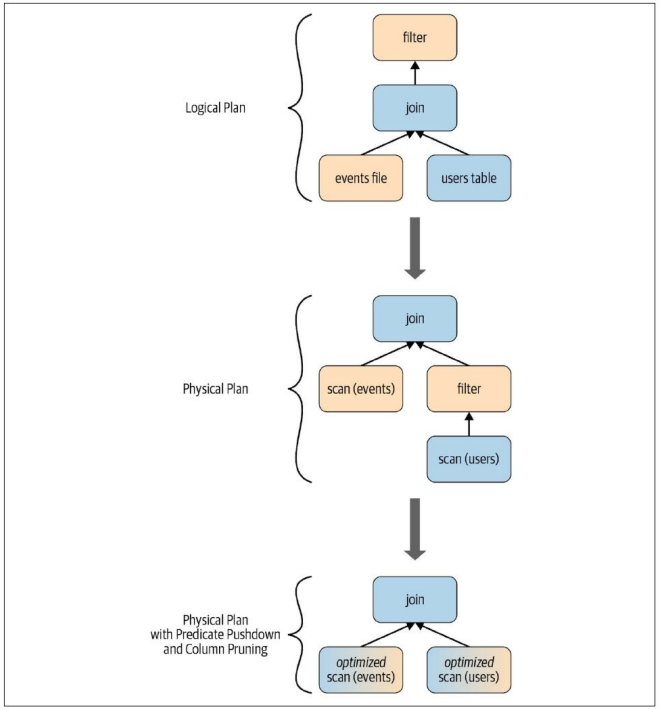
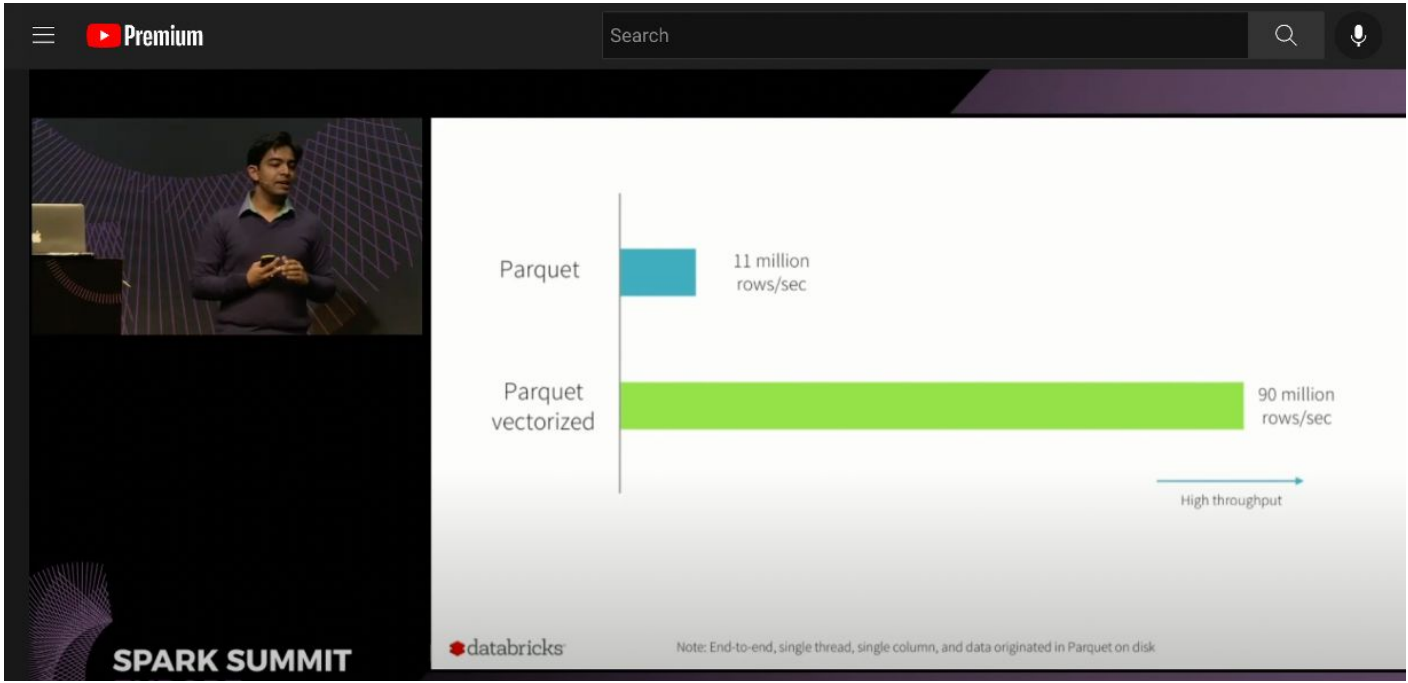


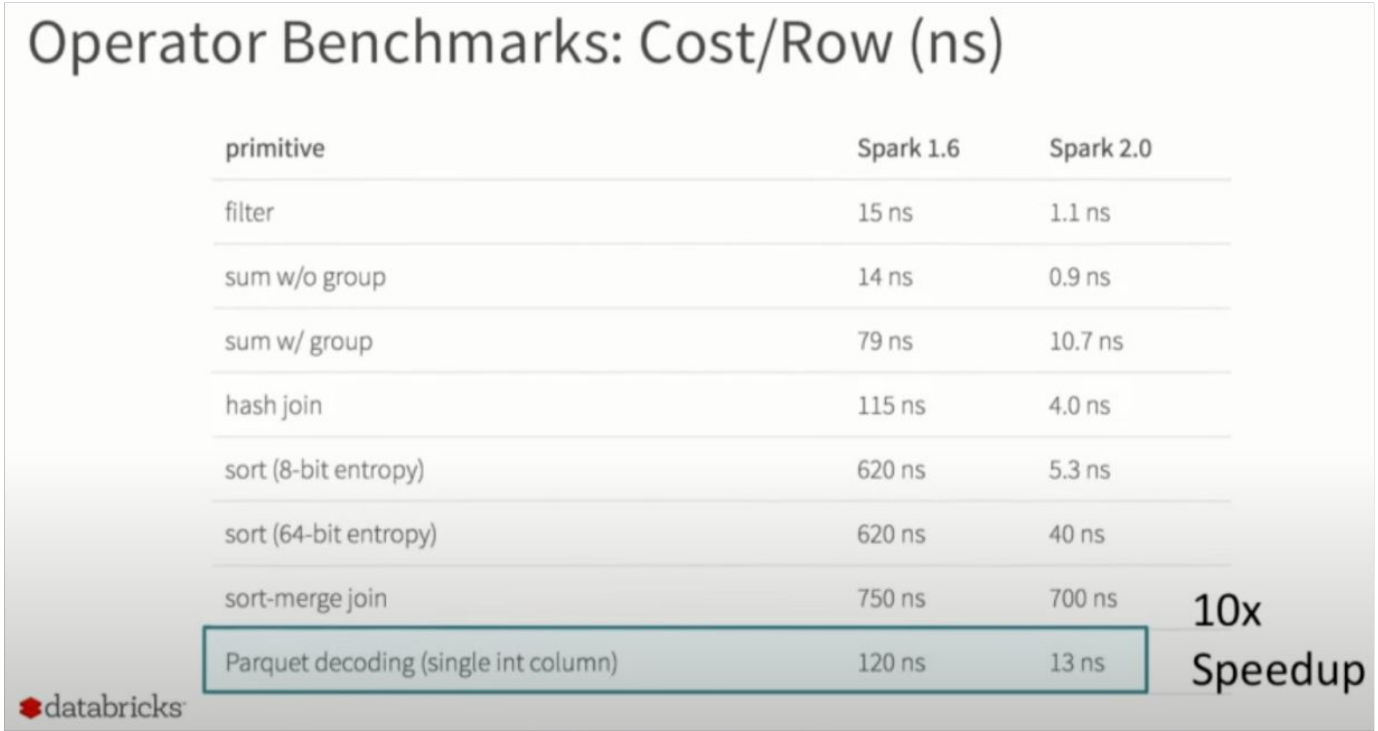
Figure 3-5. An example of a specific query transformation

Read columns from Parquet to Spark (in a vectorized manner) using whole-stage code generation/vectorization



https://www.youtube.com/watch?v=RlbBPrWJEEM

Speedups due to Parquet reader



predicate pushdown

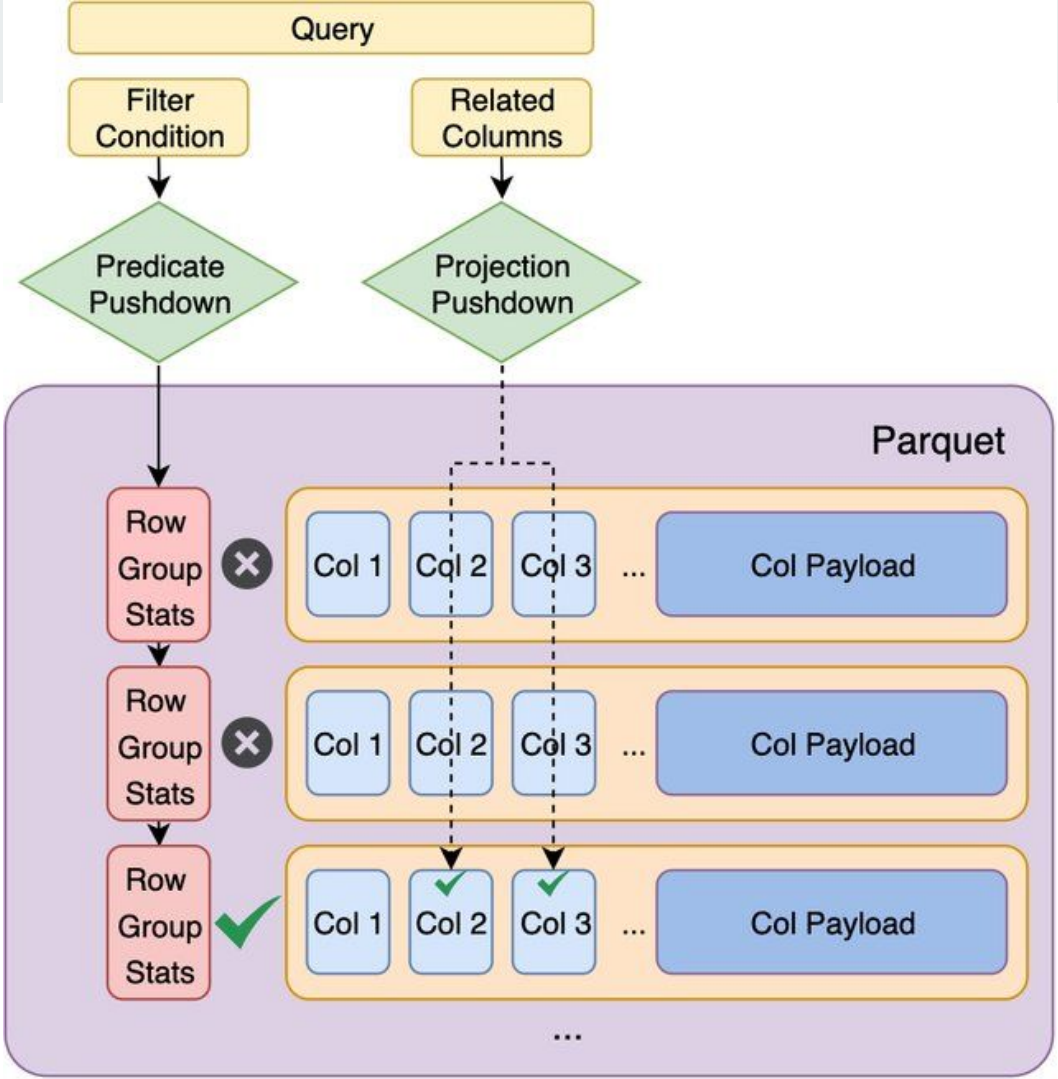
- *Another useful feature of Parquet for increasing performance is predicate pushdown.*
- *A predicate is some expression or function that evaluates to true or false based on the data record (or equivalently, the expressions in a SQL WHERE clause).*
- *In our earlier CFTR query, Spark had to deserialize/materialize each AlignmentRecord before deciding whether or not it passed the predicate. This leads to a significant amount of wasted I/O and CPU time.*
- *The Parquet reader implementations allow us to provide a predicate class that only deserializes the necessary columns for making the decision, before materializing the full record.*

Predicate Pushdown efficiency

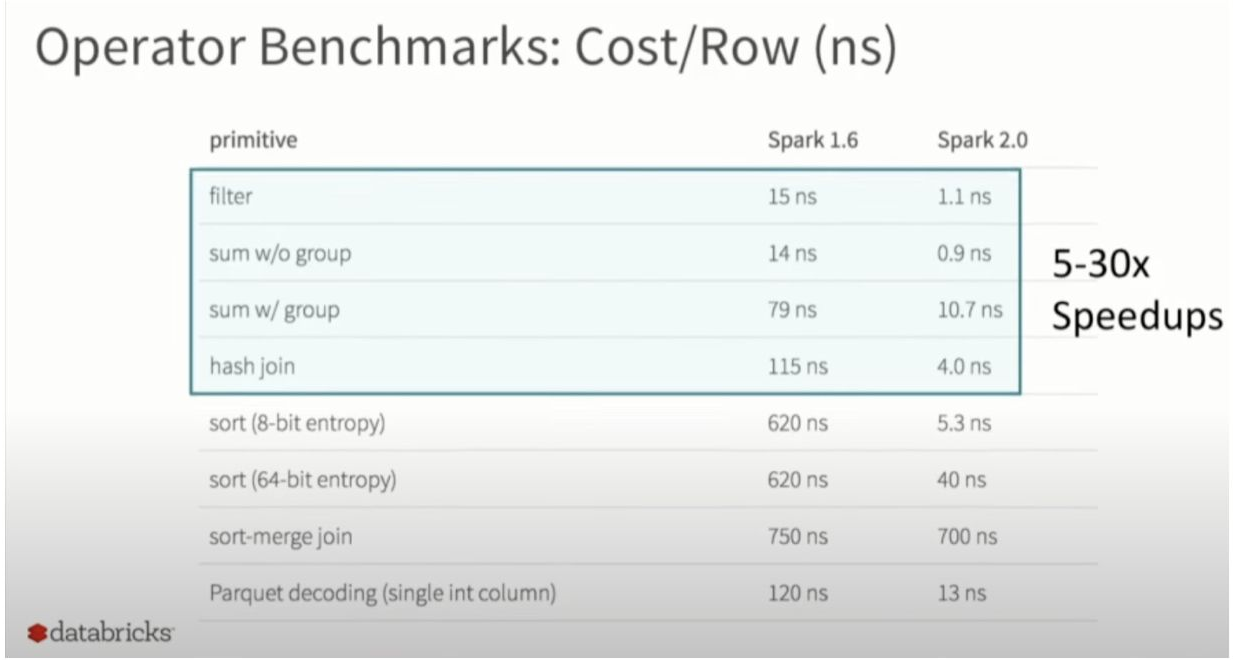
A predicate is a statement or mathematical assertion that contains variables, sometimes referred to as predicate variables, and may be true or false depending on those variables' value or values.

Predicate Pushdown

- Predicate push down is another feature of Spark and Parquet that can improve query performance by reducing the amount of data read from Parquet files. Predicate push down works by evaluating filtering predicates in the query against metadata stored in the Parquet files.
- Parquet can optionally store statistics (in particular the minimum and maximum value for a column chunk) in the relevant metadata section of its files and can use that information to take decisions, for example, to skip reading chunks of data if the provided filter predicate value in the query is outside the range of values stored for a given column.
    - This is a simplified explanation, there are many more details and exceptions that it does not catch, but it should give you a gist of what is happening under the hood.
- Predicate Pushdown efficiency:
    - Parquet is very useful in query optimization.
    - Suppose, if you have to apply some filter logic to the huge amount of data, then Spark Catalyst optimizer takes advantage of file format and push down lot of calculation to file format.
        - Parquet as a part of metadata information it will store some stats such as min, max, avg value of the partitions, which makes the query to fetch data from metadata itself.


Query
Filter Condition
Related Columns
Predicate Pushdown
Projection Pushdown
Parquet
Row Group Stats
Col 1
Col 2
Col 3
...
Col Payload
Row Group Stats
Col 1
Col 2
Col 3
...
Col Payload
Row Group Stats
Col 1
Col 2
Col 3
...
Col Payload
...

Speedups due to Wholestage code generator



Predicate Pushdown in Parquet

- When executing queries in the most generic and basic manner, filtering happens very late in the process. Moving filtering to an earlier phase of query execution provides significant performance gains by eliminating non-matches earlier, and therefore saving the cost of processing them at a later stage. This group of optimizations is collectively known as predicate pushdown.
  - When filtering query results, a consumer of the parquet-mr API (for example, Hive or Spark) can fetch all records from the API and then evaluate each record against the predicates of the filtering condition. However, this requires assembling all records in memory, even non-matching ones.
  - With predicate pushdown, these conditions are passed to the parquet-mr library instead, which evaluates the predicates on a lower level and discards non-matching records without assembling them first.
- For example,
  - when evaluating the record **{title: "The Starry Night", width: 92, height: 74}** against the condition **height > 80**, it is not necessary to assemble the whole record, because it can be discarded solely based on its height attribute.
  - However, while the condition height > width does not match the record either, predicate pushdown cannot be used in this case, because we need multiple fields of the same record to evaluate the predicate.
- Additionally, predicate pushdown also allows discarding whole row groups that cannot contain any matches based on their min/max statistics. For example, if the statistics of a row group include {min: 62, max: 78} for the height column and the filtering condition is height > 80, then none of the records in that row group can match, thus the whole row group can be discarded.

Aggregate push down

Aggregate push down

Aggregate functions are often used in SQL to compute a single result from a set of input values. The most commonly used aggregate functions are AVG, COUNT, MAX, MIN and SUM.

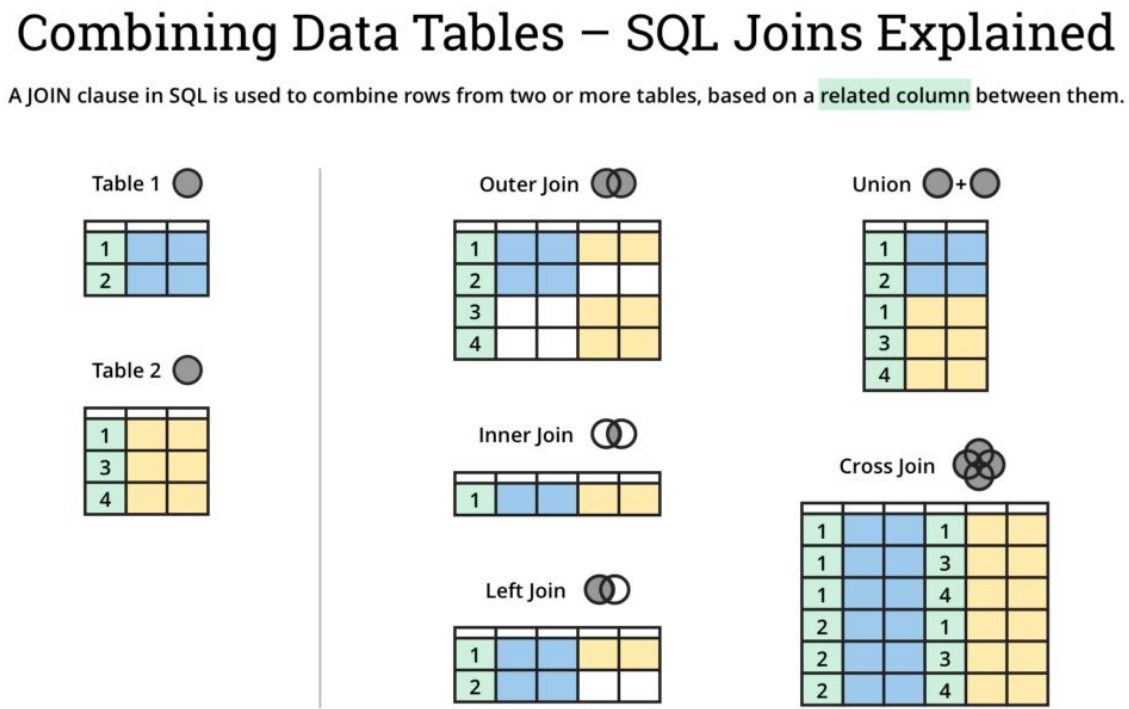
If the aggregates in the SQL statement are supported by data source with the same exact semantics as Spark, these aggregates can be pushed down to the data source level to improve performance.

Joins → Mapside hash joins

SQL - Joins

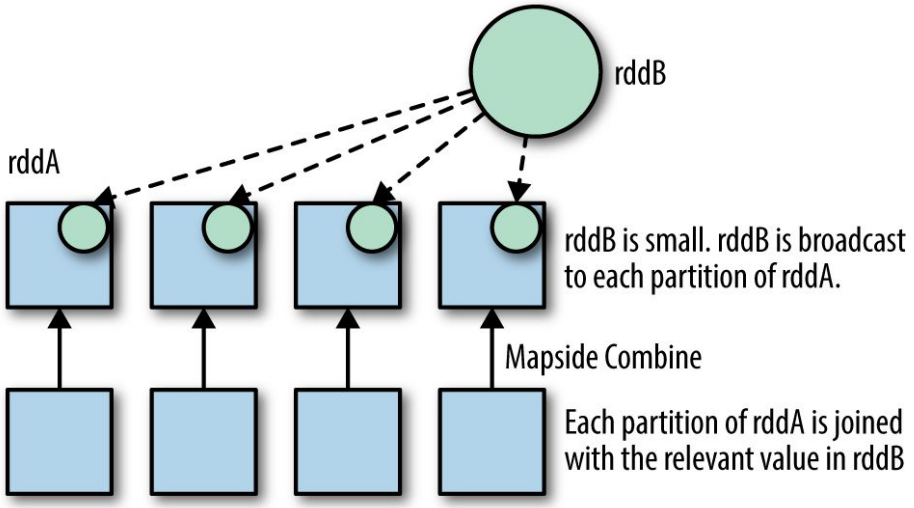
- Sometimes we need to combine data from multiple tables
- We do this with a join or union
- Let's talk through the diagram

3NF: denormalizing



MapReduce - Map-Side (Broadcast) Join

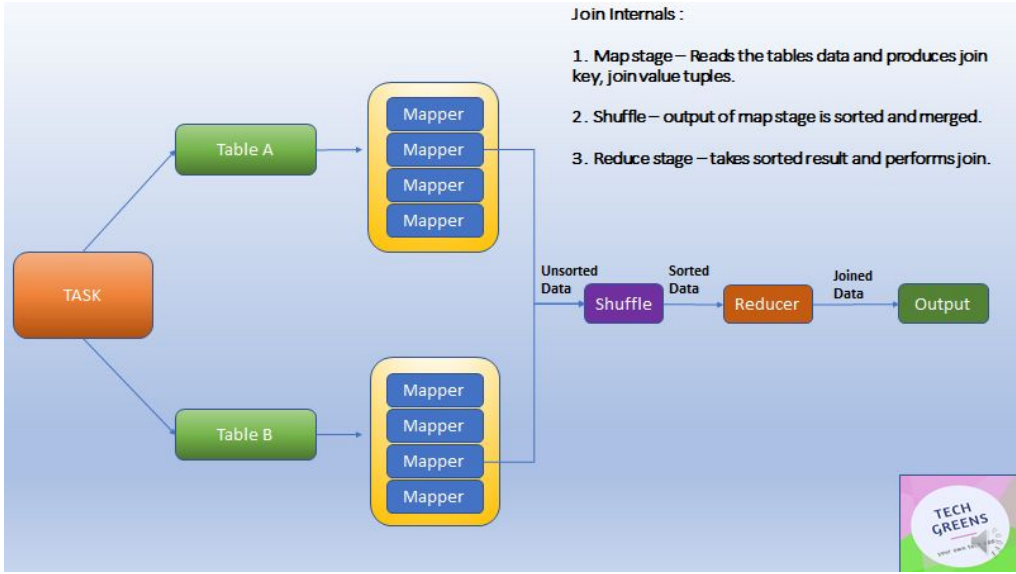
- The reduce side join had a shuffle and we know that's expensive now
- What if our user data was small?
- We could share the User data with every executor
    - In Spark this is a broadcast
- This removes the shuffle for the reduce phase and makes the joins significantly more efficient
- We make join keys and do the lookup in our shared User data in one mapping phase



Joins in MapReduce (reduce side join)

MapReduce - Reduce-Side Join (so slow!)

- Sometimes we encounter data that needs to be joined together such as
  - Clickstream
  - User demographics
- Both of these might be rather large datasets for large companies
- Clickstream data is effectively unbounded and could be trillions of rows
- User demographics for some companies might be in the billions (Netflix, Google, Facebook), but be smaller for other companies (startups)
- How do we join these datasets together efficiently?



Dataframes (read/write/format)
Repartition versus partition

wk8 Demo - Advanced Spark - Pipelines and Optimizations with DataFrames

`MIDS w261: Machine Learning at Scale | UC Berkeley School of Information | Spring 2019`

So far we've been using Spark's low level APIs. In particular, we've been using the RDD (Resilient Distiributed Datasets) API to implement Machine Learning algorithms from scratch. This week we're going to take a look at how Spark is used in a production setting. We'll look at DataFrames, SQL, and UDFs (User Defined Functions). As discussed previously, we still need to understand the internals of Spark and MapReduce in general to write efficient and scalable code.

In class today we'll get some practice working with larger data sets in Spark. We'll start with an introduction to efficiently storing data and approach a large dataset for analysis. After that we'll discuss a ranking problem which was covered in Chapter 6 of the High Performance Spark book and how we can apply that to our problem. We'll follow up with a discussion on things that could be done to make this more effiicent.

- ... **describe** differences between data serialization formats.
- ... **choose** a data serialization format based on use case.
- ... **change** submission arguements for a `SparkSession`.
- ... **set** custom configuration for a `SparkSession`.
- ... **describe** and **create** a data pipeline for analysis.
- ... **use** a user defined function (UDF).
- ... **understand** feature engineering and aggregations in Spark.

`Additional Resources:` Writing performant code in Spark requires a lot of thought. Holden's High Performance Spark book covers this topic very well. In addition, Spark - The Definitive Guide, by Bill Chambers and Matei Zaharia, provides some recent developments.

Notebook Set-Up

```
# imports
import re
import os
import numpy as np
import pandas as pd
```

Load CSV file into a dataframe; read a parquet file into a DF

```
input_df=spark.read.csv("input.csv", header=True)
input_df.show()
```

```
+-------+-------+--------------+-----------+
|ROLL_NO|SUBJECT|MARKS_OBTAINED|TOTAL_MARKS|
+-------+-------+--------------+-----------+
|   1001|English|            84|        100|
|   1001|Physics|            65|        100|
|   1001|  Maths|            45|        100|
|   1001|Science|            25|        100|
|   1001|History|            32|        100|
+-------+-------+--------------+-----------+
```

```
#Read the parquet file format
read_parquet=spark.read.parquet('out_parq\part*.parquet')
read_parquet.show()
```

Output of the above snippet will be the data in tabled structure as shown below.

Out[]:

```
#Read the parquet file format
read_paruet=spark.read.parquet('out_parq\part*.parquet')
read_paruet.show()
```

```
+-------+-------+--------------+-----------+
|ROLL_NO|SUBJECT|MARKS_OBTAINED|TOTAL_MARKS|
+-------+-------+--------------+-----------+
|   1001|English|            84|        100|
|   1001|Physics|            65|        100|
|   1001|  Maths|            45|        100|
|   1001|Science|            25|        100|
|   1001|History|            32|        100|
+-------+-------+--------------+-----------+
```

Save the dataframe as a parquet file

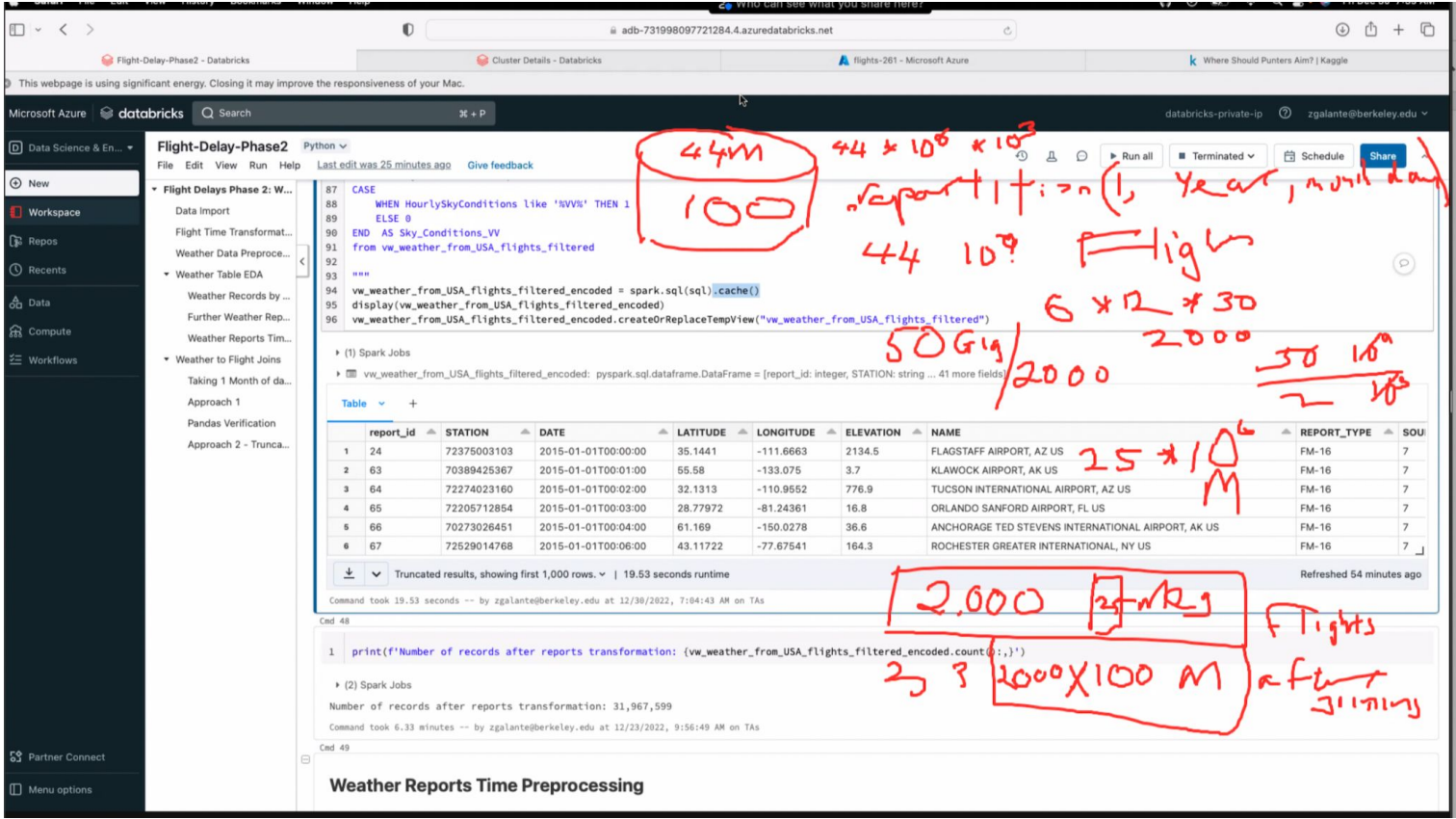
```
#Save as parquet file
input_df.coalesce(1).write.format('parquet') \
                          .mode('overwrite') \
                          .save('out_parq')
```

Coalesce(1) in our snippet will combine all the partition and result with one single partitioned file written to the target location as parquet file. From the below figure, one can notice that the file format of data stored is in Parquet.

Note: Parquet File format unlike CSV or textfile format can't be read directly using hadoop commands. To read the data in Parquet, we need to create hive table on top of data or else use spark command to read the data.

Out[]:

Repartition versus partition



Repartition versus partition

https://mungingdata.com/apache-spark/partitionby/

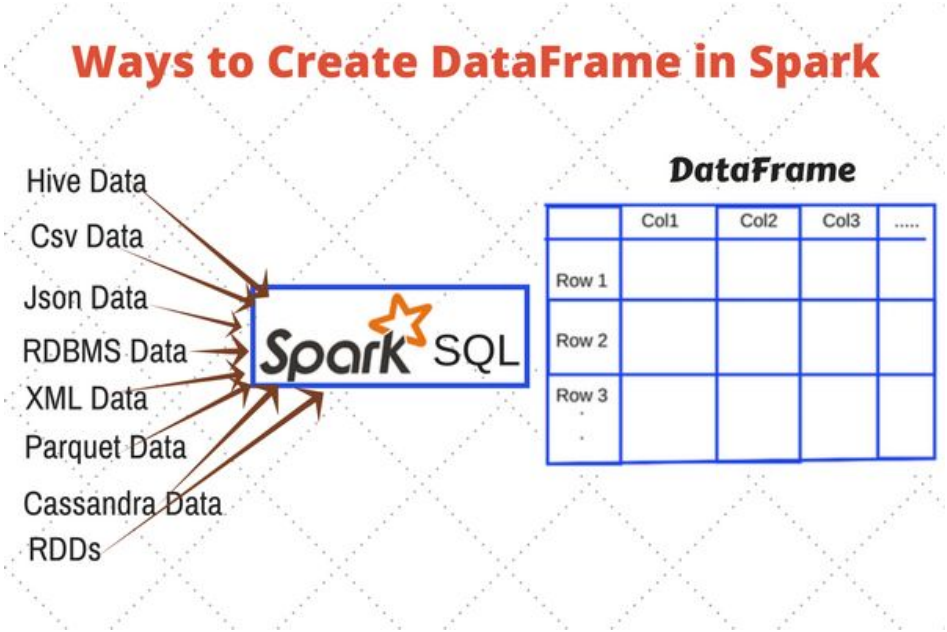
https://sparkbyexamples.com/pyspark/pyspark-repartition-vs-partitionby/?amp=1

You can create in-memory partitions before disk partitions like below.

df=df1.**repartition**(col("year"),col("month"),col("day"))

df.coalesce(2).write.**partitionBy**('year', 'month', 'day').parquet("location",mode='append')

How to create a DataFrame ?



3. Create DataFrame

- Let's Create a DataFrame by reading a CSV file. You can find the dataset explained in this article at Github zipcodes.csv file

```
df=spark.read.option("header",True) \
        .csv("/tmp/resources/simple-zipcodes.csv")
df.printSchema()

#Display below schema
root
 |-- RecordNumber: string (nullable = true)
 |-- Country: string (nullable = true)
 |-- City: string (nullable = true)
 |-- Zipcode: string (nullable = true)
 |-- state: string (nullable = true)
```

From above DataFrame, I will be using `state` as a partition key for our examples below.

https://sparkbyexamples.com/pyspark/pyspark-partitionby-example/#google_vignette

```
#partitionBy()
df.write.option("header",True) \
        .partitionBy("state") \
        .mode("overwrite") \
        .csv("/tmp/zipcodes-state")
```

On our DataFrame, we have a total of 6 different states hence, it creates 6 directories as shown below. The name of the sub-directory would be the partition column and its value (partition column=value).

Note: While writing the data as partitions, PySpark eliminates the partition column on the data file and adds partition column & value to the folder name, hence it saves some space on storage.To validate this, open any partition file in a text editor and check.

```
$ ls -lrt zipcodes-state
total 24
drwxr-xr-x 1 prabha 197121 0 Mar  4 21:18 'state=AL'/
drwxr-xr-x 1 prabha 197121 0 Mar  4 21:18 'state=AZ'/
drwxr-xr-x 1 prabha 197121 0 Mar  4 21:18 'state=FL'/
drwxr-xr-x 1 prabha 197121 0 Mar  4 21:18 'state=NC'/
drwxr-xr-x 1 prabha 197121 0 Mar  4 21:18 'state=PR'/
drwxr-xr-x 1 prabha 197121 0 Mar  4 21:18 'state=TX'/
-rw-r--r-- 1 prabha 197121 0 Mar  4 21:18  _SUCCESS
```

partitionBy("state") example output

On each directory, you may see one or more part files (since our dataset is small, all records for each `state` are kept in a single part file). You can change this behavior by `repartition()` the data in memory first. Specify the number of partitions (part files) you would want for each `state` as an argument to the repartition() method.

Enables PREDICTATE Pushdown

```
#partitionBy() multiple columns
df.write.option("header",True) \
        .partitionBy("state","city") \
        .mode("overwrite") \
        .csv("/tmp/zipcodes-state")
```

It creates a folder hierarchy for each partition; we have mentioned the first partition as `state` followed by `city` hence, it creates a `city` folder inside the `state` folder (one folder for each `city` in a `state`).

```
$ ls -lrt zipcodes-state/state=AL
total 12
drwxr-xr-x 1 prabha 197121 0 Mar  4 22:16 'city=SPRING%20GARDEN'/
drwxr-xr-x 1 prabha 197121 0 Mar  4 22:16 'city=SPRINGVILLE'/
drwxr-xr-x 1 prabha 197121 0 Mar  4 22:16 'city=SPRUCE%20PINE'/
```

partitonBy(“state”,”city”) multiple columns

```
#Use repartition() and partitionBy() together
dfRepart.repartition(2)
        .write.option("header",True) \
        .partitionBy("state") \
        .mode("overwrite") \
        .csv("c:/tmp/zipcodes-state-more")
```

If you look at the folder, you should see only 2 part files for each `state`. Dataset has 6 unique states and 2 memory partitions for each state, hence the above code creates a maximum total of 6 x 2 = 12 part files.

```
$ ls -lrt zipcodes-state-more/state=AL
total 2
-rw-r--r-- 1 prabha 197121 65 Mar  5 18:12 part-00001-40b295a0-ea4a-4c4c-b740-aab6231612d3.c000.csv
-rw-r--r-- 1 prabha 197121 91 Mar  5 18:12 part-00000-40b295a0-ea4a-4c4c-b740-aab6231612d3.c000.csv
```

Note: Since total zipcodes for each US state differ in large, California and Texas have many whereas Delaware has very few, hence it creates a Data Skew (Total rows per each part file differs in large).

Data Skew – Control Number of Records per Partition File

Use option `maxRecordsPerFile` if you want to control the number of records for each partition. This is particularly helpful when your data is skewed (Having some partitions with very low records and other partitions with high number of records).

```
#partitionBy() control number of partitions
df.write.option("header",True) \
        .option("maxRecordsPerFile", 2) \
        .partitionBy("state") \
        .mode("overwrite") \
        .csv("/tmp/zipcodes-state")
```

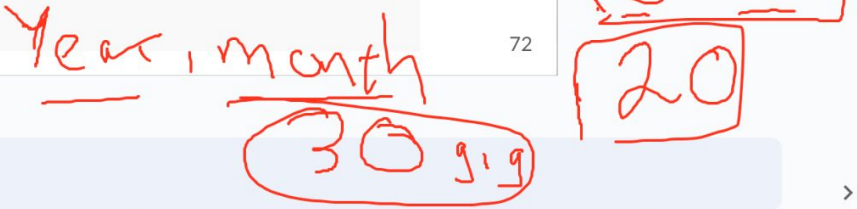
The above example creates multiple part files for each `state` and each part file contains just 2 records.

Data Skew – Control Number of Records per Partition File

Use option `maxRecordsPerFile` if you want to control the number of records for each partition. This is particularly helpful when your data is skewed (Having some partitions with very low records and other partitions with high number of records).


```
#partitionBy() control number of partitions
df.write.option("header",True) \
        .option("maxRecordsPerFile", 2) \
        .partitionBy("state") \
        .mode("overwrite") \
        .csv("/tmp/zipcodes-state")
```

The above example creates multiple part files for each `state` and each part file contains just 2 records.



72

es

8. Read a Specific Partition

Reads are much faster on partitioned data. This code snippet retrieves the data from a specific partition `"state=AL and city=SPRINGVILLE"`. Here, It just reads the data from that specific folder instead of scanning a whole file (when not partitioned).

```
dfSinglePart=spark.read.option("header",True) \
            .csv("c:/tmp/zipcodes-state/state=AL/city=SPRINGVILLE")
dfSinglePart.printSchema()
dfSinglePart.show()

#Displays
root
 |-- RecordNumber: string (nullable = true)
 |-- Country: string (nullable = true)
 |-- Zipcode: string (nullable = true)

+------------+-------+-------+
|RecordNumber|Country|Zipcode|
+------------+-------+-------+
|       54355|     US|  35146|
+------------+-------+-------+
```

While reading specific Partition data into DataFrame, it does not keep the partitions columns on DataFrame hence, you `printSchema()` and DataFrame is missing `state` and `city` columns.

9. PySpark SQL – Read Partition Data

Pushdown predicate in action

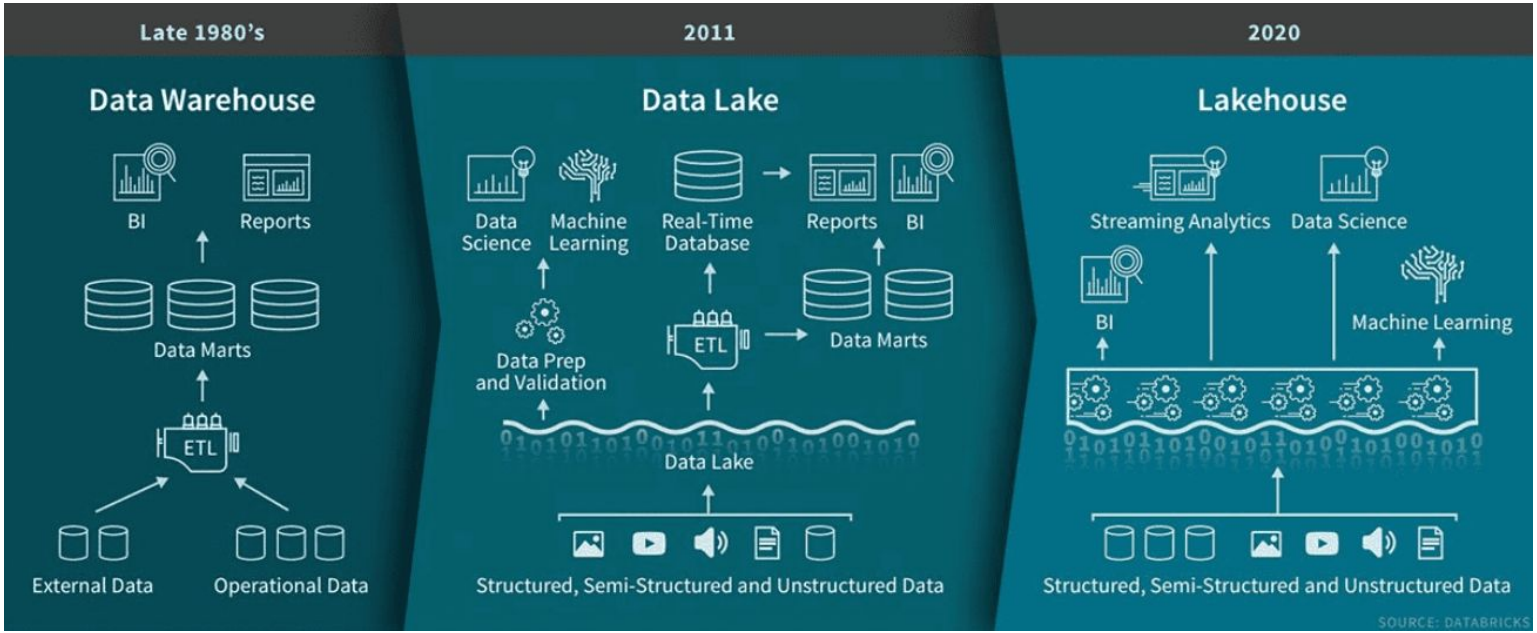
This is an example of how to write a Spark DataFrame by preserving the partition columns on DataFrame.

```
parqDF = spark.read.option("header",True) \
                  .csv("/tmp/zipcodes-state")
parqDF.createOrReplaceTempView("ZIPCODE")
spark.sql("select * from ZIPCODE  where state='AL' and city = 'SPRINGVIL
    .show()

#Display
+------------+-------+-------+-----+-----------+
|RecordNumber|Country|Zipcode|state|       city|
+------------+-------+-------+-----+-----------+
|       54355|     US|  35146|   AL|SPRINGVILLE|
+------------+-------+-------+-----+-----------+
```

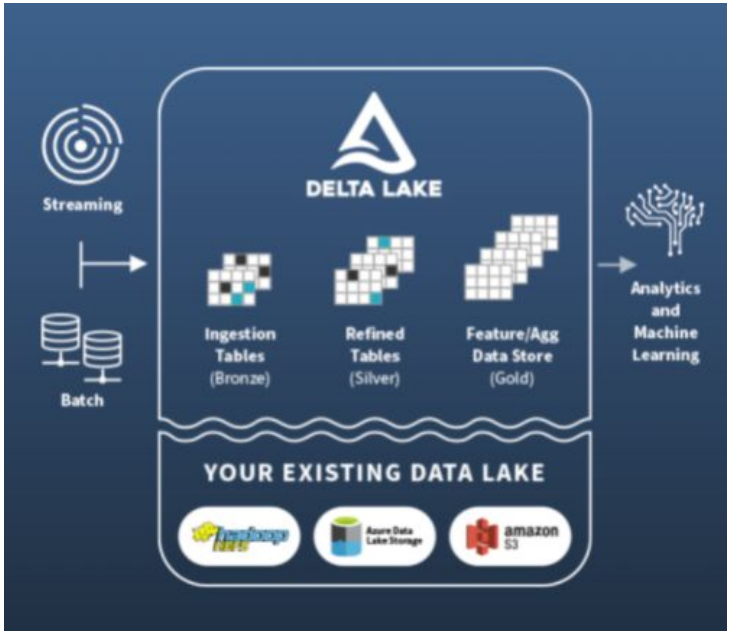
The execution of this query is also significantly faster than the query without partition. It filters the data first on `state` and then applies filters on the `city` column without scanning the entire dataset.

From BI to AI: Data Lakehouse



What is Delta Lake?

- ACID transactions
- Scalable metadata handling
- Unifies streaming and batch data processing

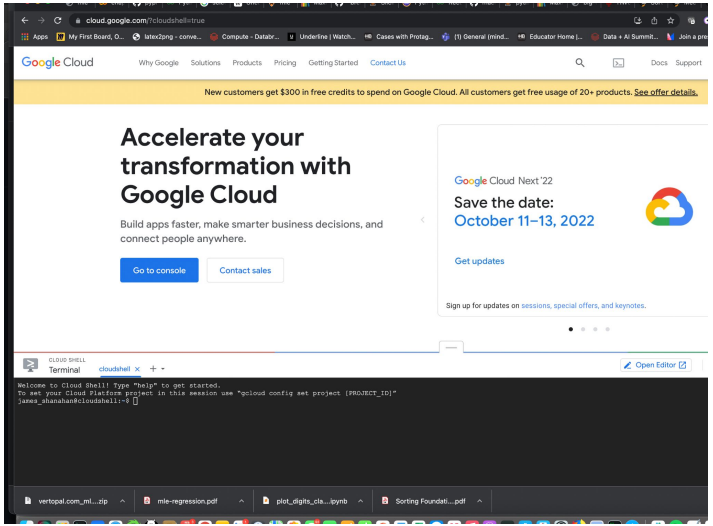


What's new in Spark 3.0

- Adaptive Query Execution
  - Enable it with: `spark.sql.adaptive.enabled=true`
- Dynamic Partition Pruning (DPP)
  - Avoid partition scanning based on the query results of the other query fragments
- Join Hints
- Improved Pandas UDFs
  - Type Hints
  - Iterators
  - Pandas Function API (mapInPandas, applyInPandas, etc)

DataProc: start new cluster with BigQuery

```
REGION=us-central1
GOOGLE_CLOUD_PROJECT =w261
# CREATE DATAPROC CLUSTER
gcloud dataproc clusters create w261 \
 --enable-component-gateway \
 --region ${REGION} \
 --subnet default \
 --no-address \
 --single-node \
 --master-machine-type n1-standard-4 \
 --master-boot-disk-size 100 \
 --image-version 2.0-debian10 \
 --optional-components JUPYTER \
 --project $GOOGLE_CLOUD_PROJECT \
 --properties spark:spark.jars="gs://spark-lib/bigquery/spark-bigquery-latest_2.12.jar" \
 --properties spark:spark.jars.packages="org.apache.spark:spark-avro_2.12:3.1.2" \
 --max-idle 3h \
 --async
```



https://cloud.google.com/