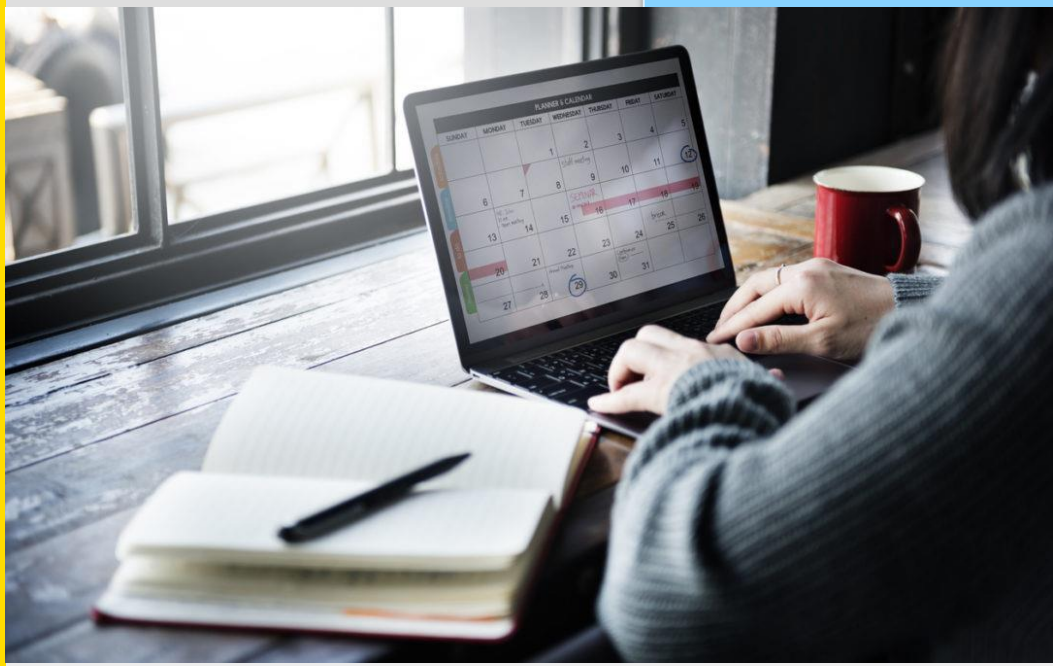
Welcome to ZZEN9313 Big Data Management



- We'll be starting at 8:30 pm
- In preparation for this webinar please check that your microphone is muted.
- We'll be taking questions by chat

This webinar will be recorded and made available for your ongoing reference.

UNSW SYDNEY

Week 6's Tutor: Zixu Zhao and Yufan Sheng

Week 6's QA Sessions:

Tuesday 7－8 pm

Wednesday 7－8 pm

Thursday 7－8 pm

Saturday 7－8 pm

Part 1. Hive

What is Hive?

- A data warehouse system for Hadoop that
  - facilitates easy data summarization
  - supports ad-hoc queries (still batch though…)
  - created by Facebook
- A mechanism to project structure onto this data and query the data using a SQL-like language – HiveQL
  - Interactive-console –or-
  - Execute scripts
  - Kicks off one or more MapReduce jobs in the background
- An ability to use indexes, built-in user-defined functions
- Latest version: 3.1.2, works with Hadoop 3.x.y

Motivation of Hive

- Limitation of MR
  - Have to use M/R model
  - Not Reusable
  - Error prone
  - For complex jobs:
    - Multiple stage of Map/Reduce functions
    - Just like ask developer to write specified physical execution plan in the database
- Hive intuitive
  - Make the unstructured data looks like tables regardless how it really lays out
  - SQL based query can be directly against these tables
  - Generate specified execution plan for this query

Hive Features

- A subset of SQL covering the most common statements
- Agile data types: Array, Map, Struct, and JSON objects
- User Defined Functions and Aggregates
- Regular Expression support
- MapReduce support
- JDBC support
- Partitions and Buckets (for performance optimization)
- Views and Indexes

Hive Type System

- Primitive types
  - Integers: TINYINT, SMALLINT, INT, BIGINT.
  - Boolean: BOOLEAN.
  - Floating point numbers: FLOAT, DOUBLE.
  - Fixed point numbers: DECIMAL
  - String: STRING, CHAR, VARCHAR.
  - Date and time types: TIMESTAMP, DATE
- Complex types
  - Structs: c has type {a INT; b INT}. c.a to access the first field
  - Maps: M['group'].
  - Arrays: ['a', 'b', 'c'], A[1] returns 'b'.
- Example
  - list< map<string, struct< p1:int,p2:int > > >
  - Represents list of associative arrays that map strings to structs that contain two ints

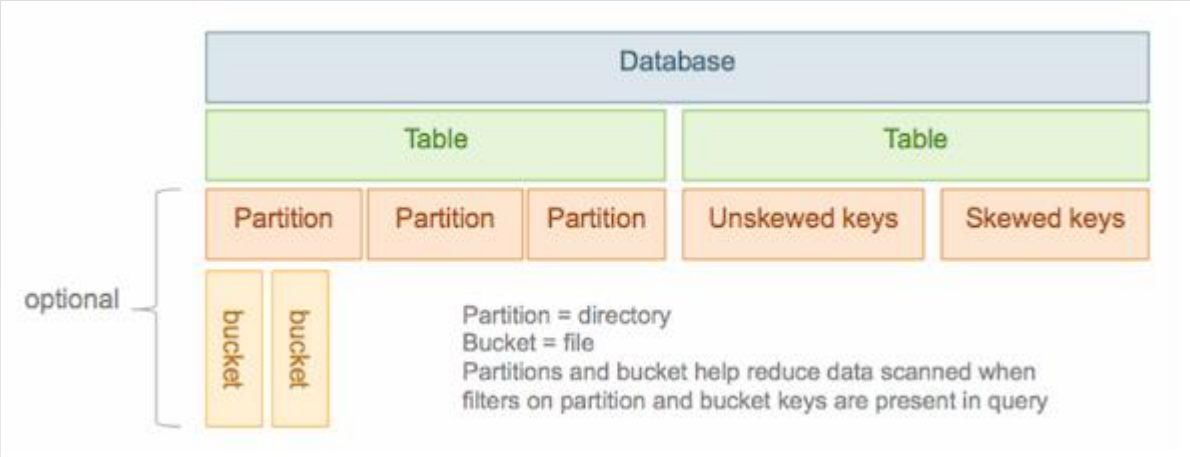
Hive Data Model

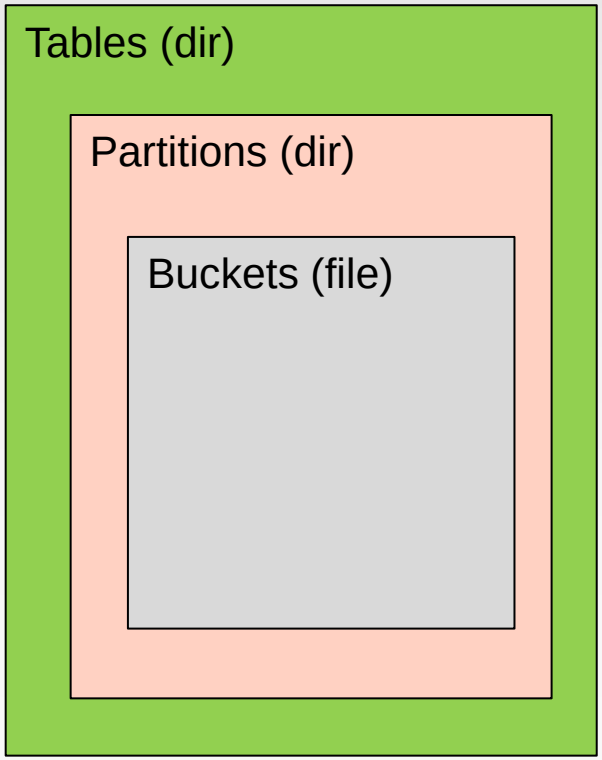
- Databases: Namespaces function to avoid naming conflicts for tables, views, partitions, columns, and so on.
- Tables: Homogeneous units of data which have the same schema.
  - Analogous to tables in relational DBs.
  - Each table has corresponding directory in HDFS.
  - An example table: page_views:
    - timestamp—which is of INT type that corresponds to a UNIX timestamp of when the page was viewed.
    - userid —which is of BIGINT type that identifies the user who viewed the page.
    - page_url—which is of STRING type that captures the location of the page.
    - referer_url—which is of STRING that captures the location of the page from where the user arrived at the current page.
    - IP—which is of STRING type that captures the IP address from where the page request was made.

Hive Data Model (Cont')

- Partitions:
  - Each Table can have one or more partition Keys which determines how the data is stored
  - Example:
    - Given the table page_views, we can define two partitions a date_partition of type STRING and country_partition of type STRING
    - All "US" data from "2009-12-23" is a partition of the page_views table
  - Partition columns are virtual columns, they are not part of the data itself but are derived on load
  - It is the user's job to guarantee the relationship between partition name and data content
- Buckets: Data in each partition may in turn be divided into Buckets based on the value of a hash function of some column of the Table
  - Example: the page_views table may be bucketed by userid

Data Model and Storage





/root-path
/table1
/partition1
(2011-11)
/bucket1 (1/3)
/bucket2 (2/3)
/bucket3 (3/3)
/partition2
(2011-12)
/bucket1 (1/3)
/bucket2 (2/3)
/bucket3 (3/3)
/table2
/bucket1 (1/2)
/bucket2 (2/2)

WordCount in Hive

- Create a table in Hive

```
create table doc(
        text string
) row format delimited fields terminated by '\n' stored as textfile;
```

- Load file into table

```
load data local inpath '/home/Words' overwrite into table doc;
```

- Compute word count using select

```
SELECT word, COUNT(*) FROM doc LATERAL VIEW
explode(split(text, ' ')) wTable as word GROUP BY word;
```

explode() Function

- explode() takes in an array (or a map) as an input and outputs the elements of the array (map) as separate rows.
- The following will return a table of words in doc, with a single column word

```
SELECT explode(split(text, ' ')) AS word FROM doc
```

- The following will compute the frequency of each word

```
SELECT word, COUNT(*)
FROM (SELECT explode(SPLIT(text, ' ')) AS word FROM doc) AS
words GROUP BY word;
```

Lateral View

- Lateral view is used in conjunction with user-defined table generating functions such as explode()
- A lateral view first applies the UDTF (User Defined Tabular Function) to each row of base table and then joins resulting output rows to form a virtual table.
- Lateral View Syntax
  - lateralView: LATERAL VIEW udtf(expression) tableAlias AS columnAlias (',' columnAlias)*
  - fromClause: FROM baseTable (lateralView)*
- Compare the two ways:

```
SELECT word, COUNT(*) FROM
(SELECT explode(SPLIT(text, ' ')) AS word FROM doc) AS words
GROUP BY word;
```

```
SELECT word, COUNT(*) FROM
doc LATERAL VIEW explode(split(text, ' ')) wTable as word
GROUP BY word;
```

Hive Operators and User-Defined Functions (UDFs)

- Built-in operators:
  - relational, arithmetic, logical, etc.
- Built-in functions:
  - mathematical, date function, string function, etc.
- Built-in aggregate functions:
  - max, min, count, etc.
- Built-in table-generating functions: transform a single input row to multiple output rows
  - explode(ARRAY): Returns one row for each element from the array.
  - explode(MAP): Returns one row for each key-value pair from the input map with two columns in each row
- Create Custom UDFs
- More details see: https://cwiki.apache.org/confluence/display/Hive/LanguageManual+UDF#LanguageManualUDF-explode

Pros/Cons

- Pros
  - An easy way to process large scale data
  - Support SQL-based queries
  - Provide more user defined interfaces to extend
  - Programmability
  - Efficient execution plans for performance
  - Interoperability with other databases
- Cons
  - No easy way to append data
  - Files in HDFS are immutable

Part 2. Spark DataFrame

A Brief Review of RDD

- The RDD is the most basic abstraction in Spark. There are three vital characteristics associated with an RDD:
  - Dependencies (lineage)
    - When necessary to reproduce results, Spark can recreate an RDD from the dependencies and replicate operations on it. This characteristic gives RDDs resiliency.
  - Partitions (with some locality information)
    - Partitions provide Spark the ability to split the work to parallelize computation on partitions across executors
    - Reading from HDFS—Spark will use locality information to send work to executors close to the data
  - Compute function: Partition => Iterator[T]
    - An RDD has a compute function that produces an Iterator[T] for the data that will be stored in the RDD.

Compute Average Values for Each Key

- Assume that we want to aggregate all the ages for each name, group by name, and then compute the average age for each name

```
dataRDD = sc.parallelize([("Brooke", 20), ("Denny", 31), ("Jules", 30),
("TD", 35), ("Brooke", 25)])
```

```
agesRDD = (dataRDD.map(lambda x: (x[0], (x[1], 1)))
```

```
.reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1]))
```

```
.map(lambda x: (x[0], x[1][0]/x[1][1])))
```

```
>>> agesRDD.collect()
[('Brooke', 22.5), ('Denny', 31.0), ('Jules', 30.0), ('TD', 35.0)]
```

Problems of RDD Computation Model

- The compute function (or computation) is opaque to Spark
  - Whether you are performing a join, filter, select, or aggregation, Spark only sees it as a lambda expression

```
dataRDD.map(lambda x: (x[0], (x[1], 1)))
```

- Spark has no way to optimize the expression, because it's unable to inspect the computation or expression in the function.
- Spark has no knowledge of the specific data type in RDD
  - To Spark it's an opaque object; it has no idea if you are accessing a column of a certain type within an object

Spark's Structured APIs

- Spark 2.x introduced a few key schemes for structuring Spark,
- This specificity is further narrowed through the use of a set of common operators in a DSL (domain specific language), including the Dataset APIs and DataFrame APIs
  - These operators let you tell Spark what you wish to compute with your data
  - It can construct an efficient query plan for execution.
- Structure yields a number of benefits, including better performance and space efficiency across Spark components

Spark's Structured APIs

- E.g, for the average age problem, using the DataFrame APIs:

```
from pyspark.sql.functions import avg

data_df = spark.createDataFrame([("Brooke", 20),
("Denny", 31), ("Jules", 30), ("TD", 35), ("Brooke", 25)],
["name", "age"])
avg_df = data_df.groupBy("name").agg(avg("age")).show()
```

```
+------+--------+
|  name|avg(age)|
+------+--------+
|Brooke|    22.5|
| Jules|    30.0|
|    TD|    35.0|
| Denny|    31.0|
+------+--------+
```

- Spark now knows exactly what we wish to do: group people by their names, aggregate their ages, and then compute the average age of all people with the same name.
- Spark can inspect or parse this query and understand our intention, and thus it can optimize or arrange the operations for efficient execution.

Datasets and DataFrames

- A *Dataset* is a distributed collection of data
  - provides the benefits of RDDs (e.g., strong typing) with the benefits of Spark SQL's optimized execution engine
  - A Dataset can be constructed from JVM objects and then manipulated using functional transformations (map, flatMap, etc.)

- A *DataFrame* is a *Dataset* organized into named columns
  - It is conceptually equivalent to a table in a relational database or a data frame in R/Python, but with richer optimizations
  - An abstraction for selecting, filtering, aggregating and plotting structured data
  - A DataFrame can be represented by a Dataset of Rows

DataFrame API

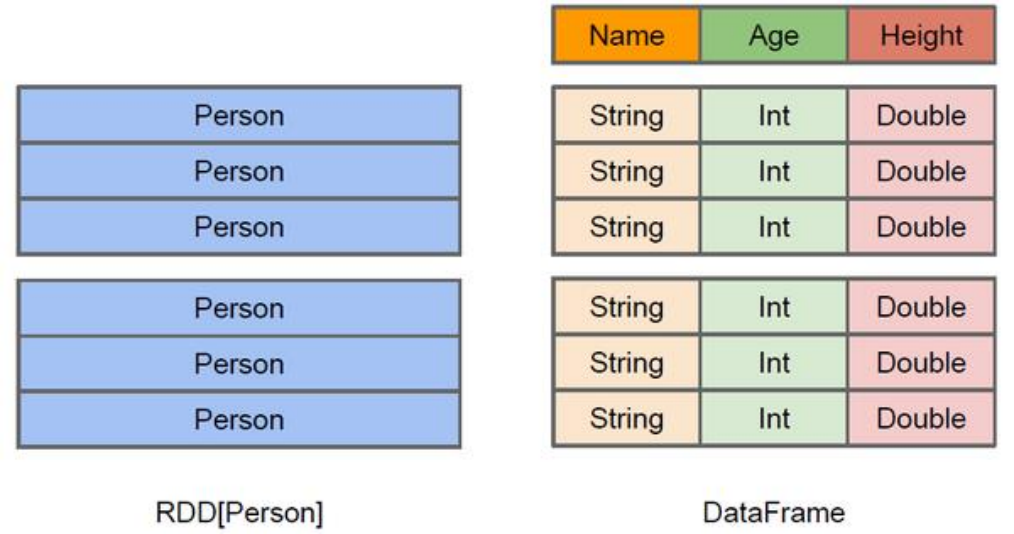
- Spark DataFrames are like distributed in-memory tables with named columns and schemas, where each column has a specific data type.
- When data is visualized as a structured table, it's not only easy to digest but also easy to work with

Id (Int)	First (String)	Last (String)	Url (String)	Published (Date)	Hits (Int)	Campaigns (List[Strings])
1	Jules	Damji	https://tinyurl.1	1/4/2016	4535	[twitter, LinkedIn]
2	Brooke	Wenig	https://tinyurl.2	5/5/2018	8908	[twitter, LinkedIn]
3	Denny	Lee	https://tinyurl.3	6/7/2019	7659	[web, twitter, FB, LinkedIn]
4	Tathagata	Das	https://tinyurl.4	5/12/2018	10568	[twitter, FB]

The table-like format of a DataFrame

Difference between DataFrame and RDD

- DataFrame more like a traditional database of two-dimensional form, in addition to data, but also to grasp the structural information of the data, that is, schema



  - RDD[Person] although with Person for type parameters, but the Spark framework itself does not understand internal structure of Person class
  - DataFrame has provided a detailed structural information, making Spark SQL can clearly know what columns are included in the dataset, and what is the name and type of each column. Thus, Spark SQL query optimizer can target optimization

Spark's Basic Data Types

- Spark supports basic internal data types, which can be declared in your Spark application or defined in your schema

Data type	Value assigned in Python	API to instantiate
ByteType	int	DataTypes.ByteType
ShortType	int	DataTypes.ShortType
IntegerType	int	DataTypes.IntegerType
LongType	int	DataTypes.LongType
FloatType	float	DataTypes.FloatType
DoubleType	float	DataTypes.DoubleType
StringType	str	DataTypes.StringType
BooleanType	bool	DataTypes.BooleanType
DecimalType	decimal.Decimal	DecimalType

Spark's Structured and Complex Data Types

- For complex data analytics, you'll need Spark to handle complex data types, such as maps, arrays, structs, dates, timestamps, fields, etc.

Data type	Value assigned in Python	API to instantiate
BinaryType	bytearray	BinaryType()
TimestampType	datetime.datetime	TimestampType()
DateType	datetime.date	DateType()
ArrayType	List, tuple, or array	ArrayType(dataType, [nullable])
MapType	dict	MapType(keyType, valueType, [nullable])
StructType	List or tuple	StructType([fields])
StructField	A value type corresponding to the type of this field	StructField(name, dataType, [nullable])

Create DataFrames

- You can convert a Python list to a DataFrame directly

```
// Given a list of pairs including names and ages
data = [("Brooke", 20), ("Denny", 31), ("Jules", 30),("TD", 35),
("Brooke", 25)]

// Create DataFrame' from a list
dataDF = spark.createDataFrame(data)
```

- You can also convert an RDD into a DataFrame

```
data = sc.parallelize([("Brooke", 20), ("Denny", 31), ("Jules",
30),("TD", 35), ("Brooke", 25)])

// Create DataFrame' from 'RDD'
dataDF = spark.createDataFrame(data)
```

Create DataFrames

- Using the above method, we can get the DataFrame as below:

```
scala> dataDF.show()
+------+---+
|    _1| _2|
+------+---+
|Brooke| 20|
| Denny| 31|
| Jules| 30|
|    TD| 35|
|Brooke| 25|
+------+---+
```

- We can see that the schema is not defined, and the columns have no meaningful names. To define the names for columns, we can use the the toDF() method

```
dataDF = spark.createDataFrame(data).toDF("name", "age")
```

```
scala> dataDF.show()
+------+---+
|  name|age|
+------+---+
|Brooke| 20|
| Denny| 31|
| Jules| 30|
|    TD| 35|
|Brooke| 25|
+------+---+
```

Schemas in Spark

- A schema in Spark defines the column names and associated data types for a DataFrame
- Defining a schema up front offers three benefits
  - You relieve Spark from the onus of inferring data types.
  - You prevent Spark from creating a separate job just to read a large portion of your file to ascertain the schema, which for a large data file can be expensive and time-consuming.
  - You can detect errors early if data doesn't match the schema.
- Define a DataFrame programmatically with three named columns, author, title, and pages

```
from pyspark.sql.types import *
schema = StructType([StructField("author", StringType(), False),
StructField("title", StringType(), False),
StructField("pages", IntegerType(), False)])
```

- Defining the same schema using DDL is much simpler:

```
schema = "author STRING, title STRING, pages INT"
```

Create DataFrames with Schema

- We can use spark.createDataFrame(data, schema) to create DataFrame, after the schema is defined for the data.
  - The first argument is the data, either a Python object or RDD
  - The second argument is the schema, either a string or *StructType*

```
from pyspark.sql.types import *
// Create the schema
schema = StructType([StructField("name", StringType(), False),
StructField("age", IntegerType(), False)])
// or
// schema = "name STRING, age INT"

// Given a list of pairs including names and ages
data = [("Brooke", 20), ("Denny", 31), ("Jules", 30),("TD", 35),
("Brooke", 25)]
// or
// data = sc.parallelize([("Brooke", 20), ("Denny", 31),
("Jules", 30),("TD", 35), ("Brooke", 25)])

// Create DataFrame' from a list and the schema
dataDF = spark.createDataFrame(data, schema)
```

Columns

- Each column describe a type of field
- We can list all the columns by their names, and we can perform operations on their values using relational or computational expressions
  - List all the columns

```
>>> dataDF.columns
['name', 'age']
```

  - Access a particular column with col and it returns a Column type

```
>>> dataDF["name"]
Column<'name'>
```

  - We can also use logical or mathematical expressions on columns

```
>>> dataDF.select(dataDF["age"]*2).show()
+---------+
|(age * 2)|
+---------+
|       40|
|       62|
|       60|
|       70|
|       50|
+---------+
```

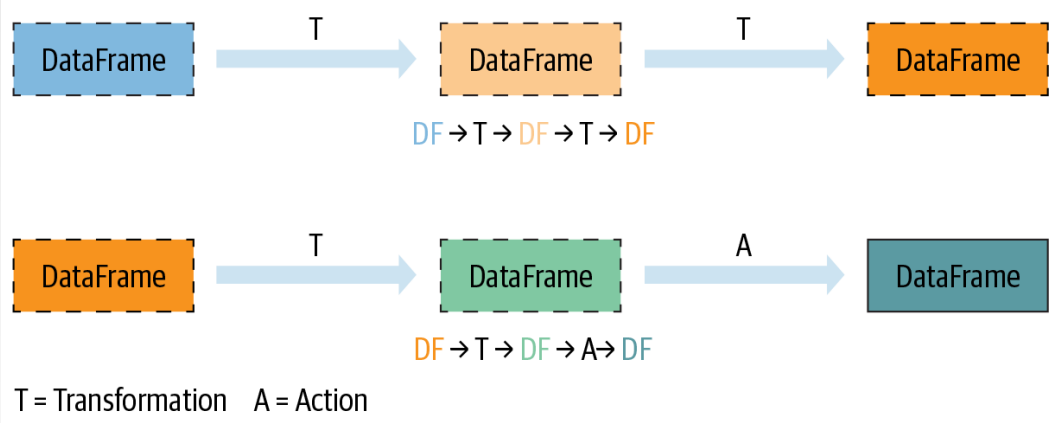
Rows

- A row in Spark is a generic Row object, containing one or more columns
- Row is an object in Spark and an ordered collection of fields, we can access its fields by an index starting at 0
- Row objects can be used to create DataFrames

```
>>> rows = [Row("Matei Zaharia", "CA"), Row("Reynold Xin", "CA")]
>>> authors_df = spark.createDataFrame(rows, ["Authors", "State"])
>>> authors_df.show()
+-------------+-----+
|      Authors|State|
+-------------+-----+
|Matei Zaharia|   CA|
|  Reynold Xin|   CA|
+-------------+-----+
```

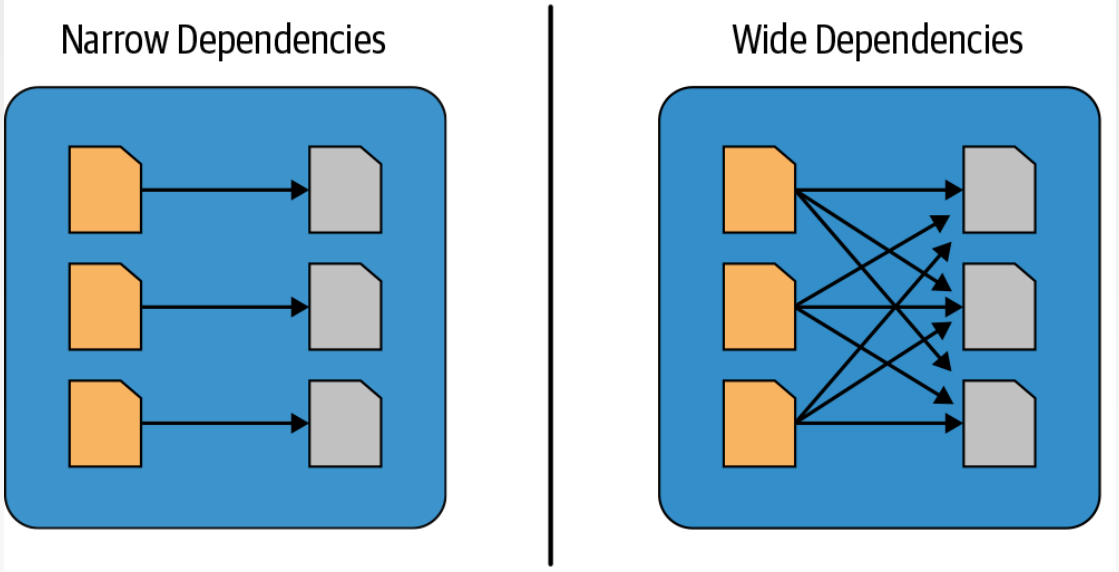
Transformations, Actions, and Lazy Evaluation

- Spark DataFrame operations can also be classified into two types: transformations and actions.
  - All transformations are evaluated lazily - their results are not computed immediately, but they are recorded or remembered as a lineage
  - An action triggers the lazy evaluation of all the recorded transformations



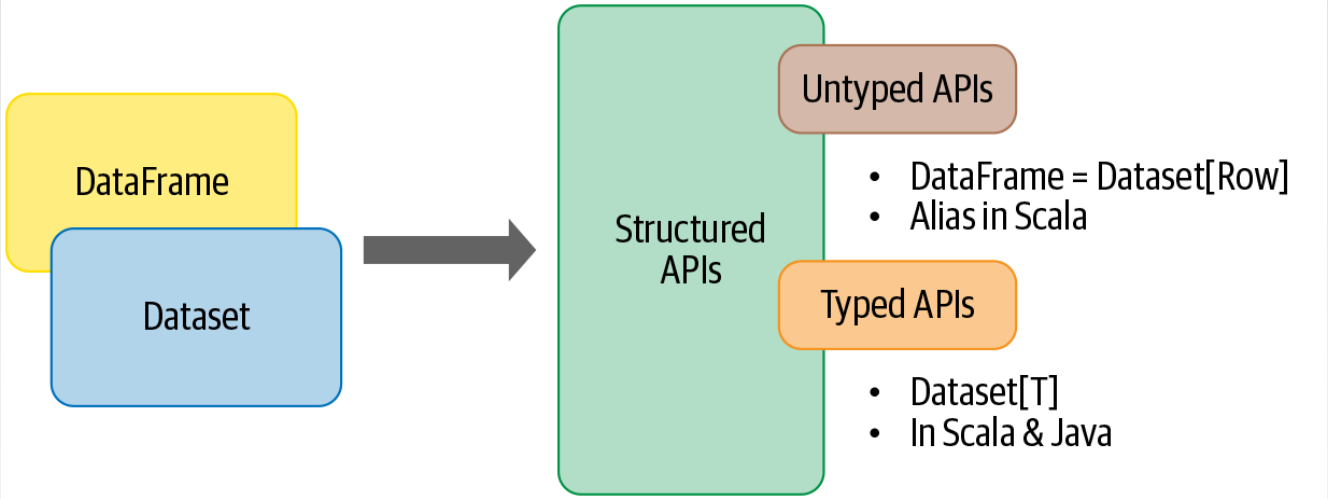
Narrow and Wide Transformations

- Transformations can be classified as having either narrow dependencies or wide dependencies
  - Any transformation where a single output partition can be computed from a single input partition is a narrow transformation, like filter()
  - Any transformation where data from other partitions is read in, combined, and written to disk is a wide transformation, like groupBy()



DataSet

- Spark 2.0 unified the DataFrame and Dataset APIs as Structured APIs with similar interfaces
- Datasets take on two characteristics: typed and untyped APIs



- Conceptually, you can think of a DataFrame in Scala as an alias for Dataset[Row]

Part 3: Spark SQL

Spark SQL Overview

- Part of the core distribution since Spark 1.0, Transform RDDs using SQL in early versions (April 2014)
- Tightly integrated way to work with structured data (tables with rows/columns)
- Data source integration: Hive, Parquet, JSON, and more

- Spark SQL is **not** about SQL.
  - Aims to Create and Run Spark Programs Faster:

Running SQL Queries Programmatically

- The sql function on a SparkSession enables applications to run SQL queries programmatically and returns the result as a DataFrame.

```
// Register the DataFrame as a SQL temporary view
df.createOrReplaceTempView("people")

sqlDF = spark.sql("SELECT * FROM people")
sqlDF.show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

Global Temporary View

- Temporary views in Spark SQL are session-scoped and will disappear if the session that creates it terminates
- Global temporary view: a temporary view that is shared among all sessions and keep alive until the Spark application terminates
- Global temporary view is tied to a system preserved database global_temp, and we must use the qualified name to refer it, e.g. SELECT * FROM global_temp.view1

Global Temporary View Example

```
// Register the DataFrame as a global temporary view
df.createGlobalTempView("people")

// Global temporary view is tied to a system preserved database `global_temp`
spark.sql("SELECT * FROM global_temp.people").show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+

// Global temporary view is cross-session
spark.newSession().sql("SELECT * FROM global_temp.people").show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

Find full example code at
https://github.com/apache/spark/blob/master/examples/src/main/python/sql/basic.py

Error Detection of Structured APIs

- If you want errors caught during compilation rather than at runtime, choose the appropriate API