SparkSQL and DataFrame

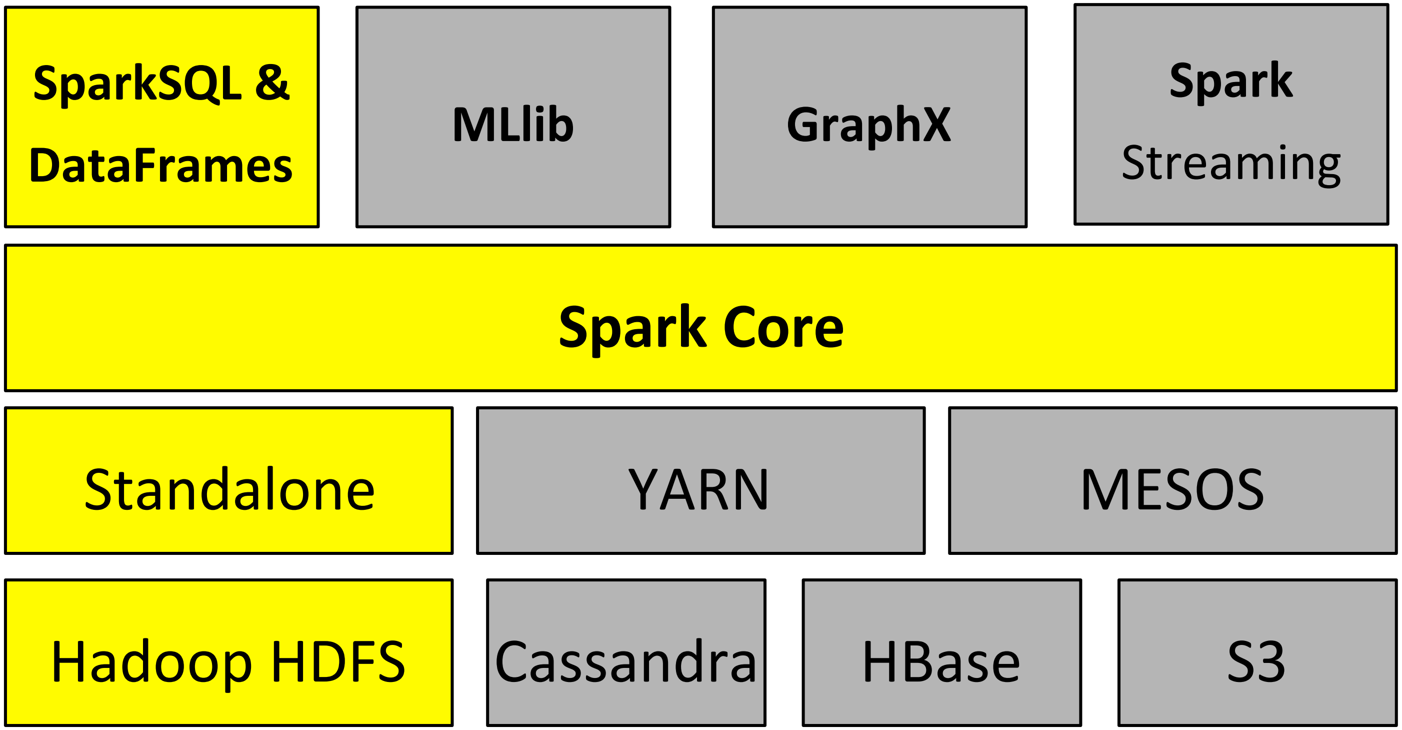
Lecture 5
November 9th, 2017

Jae W. Lee (jaewlee@snu.ac.kr)
Computer Science and Engineering
Seoul National University

***Slide credits**: Prof. Anthony Joseph (BerkeleyX CS105x)*

SparkSQL & DataFrames

- **Special-purpose libraries for a variety of data science tasks**
  - SQL-like query computation by SparkSQL
  - DataFrames: Table



*** Image from https://www.safaribooksonline.com/library/view/data-analytics-with/9781491913734/ch04.html**

Outline

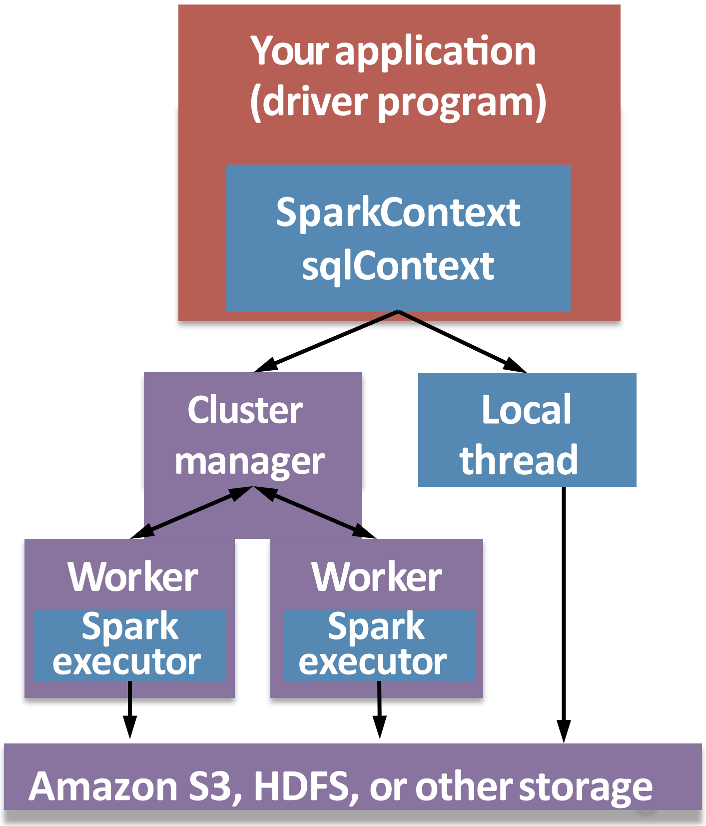
- **DataFrames**
- Spark Transformations and Actions
- Spark Programming Model
- Relational Database and SQL
- SparkSQL

DataFrames

- **Primary abstraction in Spark**
  - *Immutable* (i.e., read-only) once constructed
  - Track lineage information to efficiently recompute lost data
  - Enable operations on collection of elements in parallel
    - "Here's an operation, run it on all of the data"

- **You can construct DataFrames**
  - by *parallelizing* existing Python collections (lists)
  - by *transforming* an existing Spark or pandas DFs
  - from *files* in HDFS or any other storage system

Spark Driver and Workers



- **A Spark program is two programs:**
  - A driver program and a workers program
- **Worker programs run on cluster nodes or in local threads**
- **DataFrames are distributed across workers**

Spark and SQL Contexts

- **A Spark program first creates a `SparkContext` object**
  - `SparkContext` tells Spark how and where to access a cluster
  - pySpark shell automatically creates `SparkContext`
  - iPython and programs must create a new `SparkContext`
- **The program next creates a `sqlContext` object**
- **Use `sqlContext` to create DataFrames**

In the labs, we create the SparkContext and sqlContext for you

DataFrames

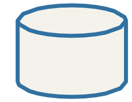
- Each row of a DataFrame is a Row object
- The fields in a Row can be accessed like attributes

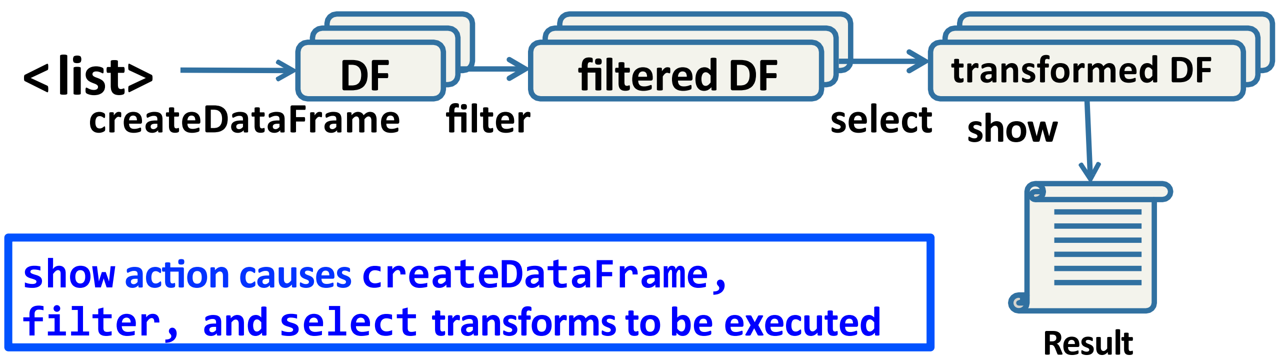
```
>>> row = Row(name="Alice", age=11)
>>> row
Row(age=11, name='Alice')
>>> row['name'], row['age']
('Alice', 11)
>>> row.name, row.age
('Alice', 11)
```

DataFrames

- **Similarities to RDD**
  - Two types of operations: *transformations* and *actions*
  - Transformations are lazy (*not computed immediately*)
  - Transformed DF is executed when action runs on it
  - Persist (cache) DFs in memory or disk

Working with DataFrames

- Create a DataFrame from a data source:  <list>
- Apply *transformations* to a DataFrame: select, filter, ...
- Apply *actions* to a DataFrame: collect, count, ...



`show` action causes `createDataFrame`, `filter`, and `select` transforms to be executed

Creating DataFrames

- Create DataFrames from Python collections (lists)

```
>>> data = [('Alice', 1), ('Bob', 2)]

>>> data

[('Alice', 1), ('Bob', 2)]

>>> df = sqlContext.createDataFrame(data)

[Row(_1=u'Alice', _2=1), Row(_1=u'Bob', _2=2)]

>>> sqlContext.createDataFrame(data, ['name', 'age'])

[Row(name=u'Alice', age=1), Row(name=u'Bob', age=2)]
```

No computation occurs with `sqlContext.createDataFrame()`
- Spark only records how to create the DataFrame

pandas: Python Data Analysis Library

- **Open source data analysis and modeling library**
  - An alternative to using R
- **pandas DataFrame: a table with named columns**
  - The most commonly used pandas object
  - Represented as a Python Dict (column_name → Series)
  - Each pandas Series object represents a column
- **1-D labeled array capable of holding any data type**
  - R has a similar data frame type

Creating DataFrames

- **Easy to create pySpark DataFrames from pandas (and R) DataFrames**

```
# Create a Spark DataFrame from Pandas
>>> spark_df = sqlContext.createDataFrame(pandas_df)
```

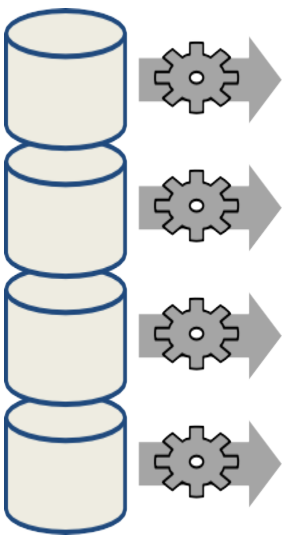
Creating DataFrames

- **From HDFS, text files, JSON files, Apache Parquet, Hypertable, Amazon S3, Apache Hbase, SequenceFiles, any other Hadoop `InputFormat`, and directory or glob wildcard: `/data/201404*`**

```
>>> df = sqlContext.read.text("README.txt")
>>> df.collect()
[Row(value=u'hello'), Row(value=u'this')]
```

Creating a DataFrame from a File

```
distFile = sqlContext.read.text ("...")
```



Loads text file and returns a DataFrame with a single string column named "value"

Each line in text file is a row

Lazy evaluation **means no execution happens now**

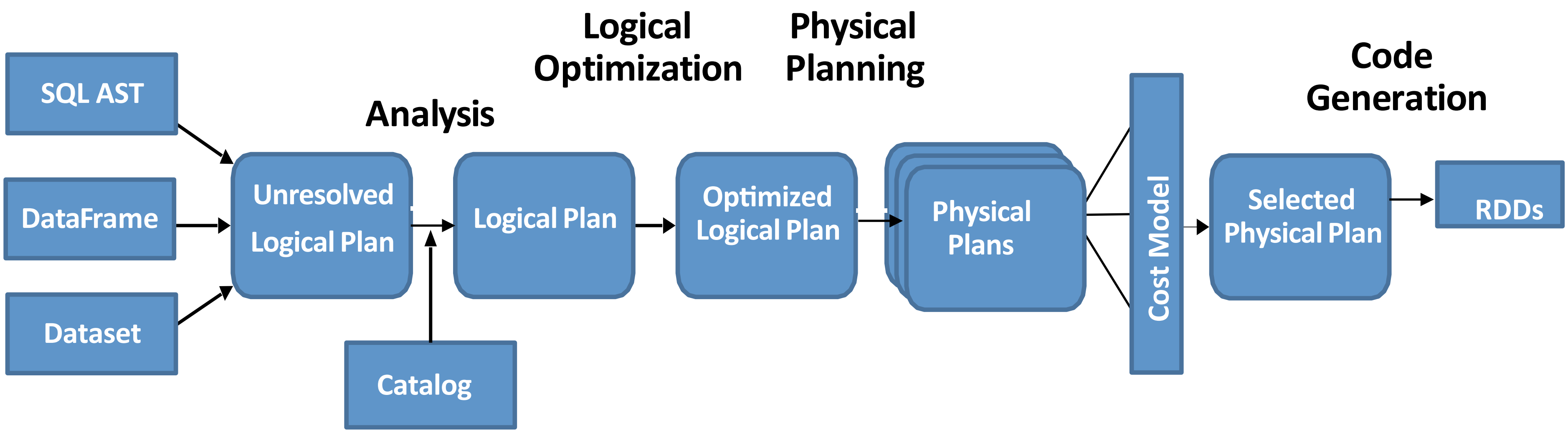
Outline

- DataFrames
- **Spark Transformations and Actions**
- Spark Programming Model
- Relational Database and SQL
- SparkSQL

Spark Transformations

- **Create new `DataFrame` from an existing one**
- **Use *lazy evaluation***
  - Results not computed right away – Spark remembers set of transformations applied to base `DataFrame`
  - Spark uses *Catalyst* to optimize the required calculations
  - Spark recovers from failures and slow workers
- ***Think of this as a recipe for creating result***

Catalyst: Shared Optimization & Execution



- **DataFrames, Datasets, and Spark SQL share the same optimization/execution pipeline**

Column Transformations

The apply method creates a `DataFrame` from one column:

```
>>> ageCol = people.age
```

Column Transformations

The apply method creates a `DataFrame` from one column:

```
>>> ageCol = people.age
```

You can select one or more columns from a `DataFrame`:

```
>>> df.select('*')
```

- `*` selects all the columns

Column Transformations

The apply method creates a `DataFrame` from one column:

```
>>> ageCol = people.age
```

You can select one or more columns from a `DataFrame`:

```
>>> df.select('*')
```

* selects all the columns

```
>>> df.select('name', 'age')
```

* selects the name and age columns

Column Transformations

The apply method creates a `DataFrame` from one column:

```
>>> ageCol = people.age
```

You can select one or more columns from a `DataFrame`:

```
>>> df.select('*')
```

* selects all the columns

```
>>> df.select('name', 'age')
```

* selects the name and age columns

```
>>> df.select(df.name,
              (df.age + 10).alias('age'))
```

* selects the name and age columns,
increments the values in the age column by 10,
and renames (alias) the age + 10 column as age

More Column Transformations

The drop method returns a new `DataFrame` that drops the specified column:

```
>>> df.drop(df.age)
[Row(name=u'Alice'), Row(name=u'Bob')]
```

Review: Python `lambda` Functions

- **Small anonymous functions (not bound to a name)**
  - Example: `lambda a, b: a + b`
    - returns the sum of its two arguments
- **Can use lambda functions wherever function objects are required**
- **Restricted to a single expression**

User Defined Function Transformations

- Transform a DataFrame using a User Defined Function

```
>>> from pyspark.sql.types import IntegerType
>>> slen = udf(lambda s: len(s), IntegerType())
>>> df.select(slen(df.name).alias('slen'))
```

   * Creates a `DataFrame` of [Row(slen=5), Row(slen=3)]

- UDF takes named or lambda function and the return type of the function

Other Useful Transformations

Transformation	Description
`filter(func)`	returns a new `DataFrame` formed by selecting those rows of the source on which *func* returns true
`where(func)`	`where` is an alias for `filter`
`distinct()`	return a new `DataFrame` that contains the distinct rows of the source `DataFrame`
`orderBy(*cols, **kw)`	returns a new `DataFrame` sorted by the specified *column(s)* and in the sort order specified by *kw*
`sort(*cols, **kw)`	Like `orderBy`, `sort` returns a new `DataFrame` sorted by the specified *column(s)* and in the sort order specified by *kw*
`explode(col)`	returns a new row for each element in the given array or map

`func` is a Python named function or `lambda` function

Using Transformations (1)

```
>>> df = sqlContext.createDataFrame(data, ['name', 'age'])
[Row(name=u'Alice', age=1), Row(name=u'Bob', age=2)]
```

Using Transformations (1)

```
>>> df = sqlContext.createDataFrame(data, ['name', 'age'])
[Row(name=u'Alice', age=1), Row(name=u'Bob', age=2)]

>>> from pyspark.sql.types import IntegerType
>>> doubled = udf(lambda s: s * 2, IntegerType())
>>> df2 = df.select(df.name, doubled(df.age).alias('age'))
[Row(name=u'Alice', age=2), Row(name=u'Bob', age=4)]
```

*** selects the name and age columns, applies the UDF to age column and aliases resulting column to age**

Using Transformations (1)

```
>>> df = sqlContext.createDataFrame(data, ['name', 'age'])
[Row(name=u'Alice', age=1), Row(name=u'Bob', age=2)]

>>> from pyspark.sql.types import IntegerType
>>> doubled = udf(lambda s: s * 2, IntegerType())
>>> df2 = df.select(df.name, doubled(df.age).alias('age'))
[Row(name=u'Alice', age=2), Row(name=u'Bob', age=4)]
```

* selects the name and age columns, applies the UDF to age column and aliases resulting column to age

```
>>> df3 = df2.filter(df2.age > 3)
[Row(name=u'Bob', age=4)]
```

* only keeps rows with age column greater than 3

Using Transformations (2)

```
>>> data2 = [('Alice', 1), ('Bob', 2), ('Bob', 2)]
>>> df = sqlContext.createDataFrame(data2, ['name', 'age'])
[Row(name=u'Alice', age=1), Row(name=u'Bob', age=2),
   Row(name=u'Bob', age=2)]
>>> df2 = df.distinct()
[Row(name=u'Alice', age=1), Row(name=u'Bob', age=2)]
```

*** only keeps rows that are distinct**

Using Transformations (2)

```
>>> data2 = [('Alice', 1), ('Bob', 2), ('Bob', 2)]
>>> df = sqlContext.createDataFrame(data2, ['name', 'age'])
[Row(name=u'Alice', age=1), Row(name=u'Bob', age=2),
   Row(name=u'Bob', age=2)]
>>> df2 = df.distinct()
[Row(name=u'Alice', age=1), Row(name=u'Bob', age=2)]
```

* only keeps rows that are distinct

```
>>> df3 = df2.sort("age", ascending=False)
[Row(name=u'Bob', age=2),
Row(name=u'Alice', age=1)]
```

* sort ascending on the age column

Using Transformations (III)

```
>>> data3 = [Row(a=1, intlist=[1,2,3])]
>>> df4 = sqlContext.createDataFrame(data3)
[Row(a=1, intlist=[1,2,3])]
>>> df4.select(explode(df4.intlist).alias("anInt"))
[Row(anInt=1), Row(anInt=2), Row(anInt=3)]
```

* turn each element of the `intlist` column into a `Row`, alias the resulting column to `anInt`, and select only that column

GroupedData Transformations

- `groupBy(*cols)` groups the `DataFrame` using the specified columns, so we can run aggregation on them

GroupedData Function	Description
`agg(*exprs)`	Compute aggregates (avg, max, min, sum, or count) and returns the result as a `DataFrame`
`count()`	counts the number of records for each group
`avg(*args)`	computes average values for numeric columns for each group

Using GroupedData (1)

```
>>> data = [('Alice',1,6), ('Bob',2,8), ('Alice',3,9), ('Bob',4,7)]
>>> df = sqlContext.createDataFrame(data, ['name', 'age', 'grade'])
>>> df1 = df.groupBy(df.name)
>>> df1.agg({"*": "count"}).collect()
[Row(name=u'Alice', count(1)=2), Row(name=u'Bob', count(1)=2)]
```

Using GroupedData (1)

```
>>> data = [('Alice',1,6), ('Bob',2,8), ('Alice',3,9), ('Bob',4,7)]
>>> df = sqlContext.createDataFrame(data, ['name', 'age', 'grade'])
>>> df1 = df.groupBy(df.name)
>>> df1.agg({"*": "count"}).collect()
[Row(name=u'Alice', count(1)=2), Row(name=u'Bob', count(1)=2)]

>>> df.groupBy(df.name).count()
[Row(name=u'Alice', count=2), Row(name=u'Bob', count=2)]
```

Using GroupedData (2)

```
>>> data = [('Alice',1,6), ('Bob',2,8), ('Alice',3,9), ('Bob',4,7)]
>>> df = sqlContext.createDataFrame(data, ['name', 'age', 'grade'])
>>> df.groupBy().avg().collect()
[Row(avg(age)=2.5, avg(grade)=7.5)]
```

Using GroupedData (2)

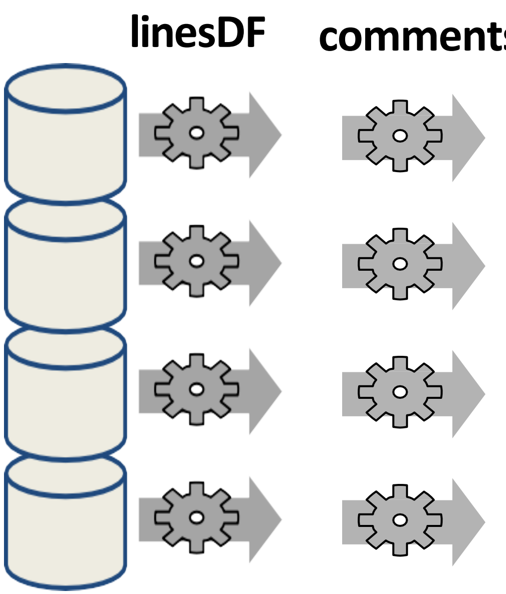
```
>>> data = [('Alice',1,6), ('Bob',2,8), ('Alice',3,9), ('Bob',4,7)]
>>> df = sqlContext.createDataFrame(data, ['name', 'age', 'grade'])
>>> df.groupBy().avg().collect()
[Row(avg(age)=2.5, avg(grade)=7.5)]

>>> df.groupBy('name').avg('age', 'grade').collect()
[Row(name=u'Alice', avg(age)=2.0, avg(grade)=7.5),
   Row(name=u'Bob', avg(age)=3.0, avg(grade)=7.5)]
```

Transforming a DataFrame

```
linesDF = sqlContext.read.text('...')

commentsDF = linesDF.filter(isComment)
```



Lazy evaluation means nothing executes – Spark saves recipe for transforming source

Spark Actions

- Cause Spark to execute recipe to transform source
- Mechanism for getting results out of Spark

Some Useful Actions

Action	Description
`show(n, truncate)`	prints the first *n* rows of the `DataFrame`
`take(n)`	returns the first *n* rows as a list of `Row`
`collect()`	return all the records as a list of `Row` WARNING: make sure will fit in driver program
`count()`*	returns the number of rows in this `DataFrame`
`describe(*cols)`	Exploratory Data Analysis function that computes statistics (count, mean, stddev, min, max) for numeric columns – if no columns are given, this function computes statistics for all numerical columns

* **`count` for `DataFrames` is an action, while for `GroupedData` it is a transformation**

Getting Data Out of DataFrames (1)

```
>>> df = sqlContext.createDataFrame(data, ['name', 'age'])
>>> df.collect()
[Row(name=u'Alice', age=1), Row(name=u'Bob', age=2)]
```

Getting Data Out of DataFrames (1)

```
>>> df = sqlContext.createDataFrame(data, ['name', 'age'])
>>> df.collect()
[Row(name=u'Alice', age=1), Row(name=u'Bob', age=2)]

>>> df.show()
+-----+---+
| name|age|
+-----+---+
|Alice|  1|
|  Bob|  2|
+-----+---+
```

Getting Data Out of DataFrames (1)

```
>>> df = sqlContext.createDataFrame(data, ['name', 'age'])
>>> df.collect()
[Row(name=u'Alice', age=1), Row(name=u'Bob', age=2)]

>>> df.show()
+-----+---+
| name|age|
+-----+---+
|Alice|  1|
|  Bob|  2|
+-----+---+


>>> df.count()
2
```

Getting Data Out of DataFrames (2)

```
>>> df = sqlContext.createDataFrame(data, ['name', 'age'])
>>> df.take(1)
[Row(name=u'Alice', age=1)]
```

Getting Data Out of DataFrames (2)

```
>>> df = sqlContext.createDataFrame(data, ['name', 'age'])
>>> df.take(1)
[Row(name=u'Alice', age=1)]

>>> df.describe()
+-------+------------------+
|summary|               age|
+-------+------------------+
|  count|                 2|
|   mean|               1.5|
| stddev|0.7071067811865476|
|    min|                 1|
|    max|                 2|
+-------+------------------+
```

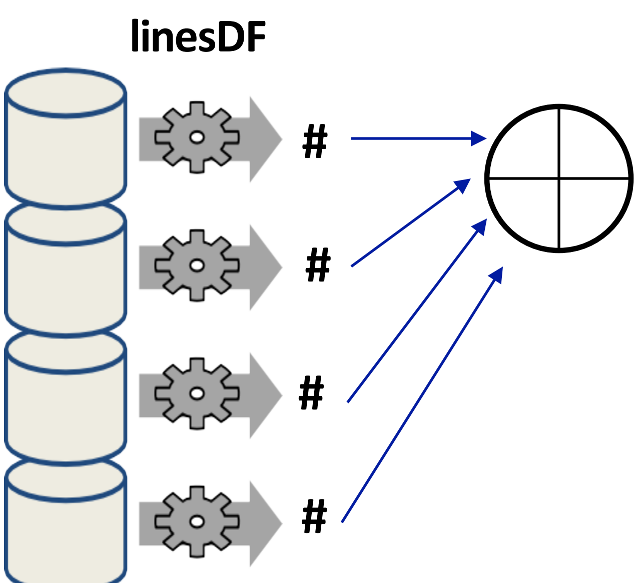
Outline

- DataFrames
- Spark Transformations and Actions
- **Spark Programming Model**
- Relational Database and SQL
- SparkSQL

Spark Programming Model

```
linesDF = sqlContext.read.text('...')

print linesDF.count()
```

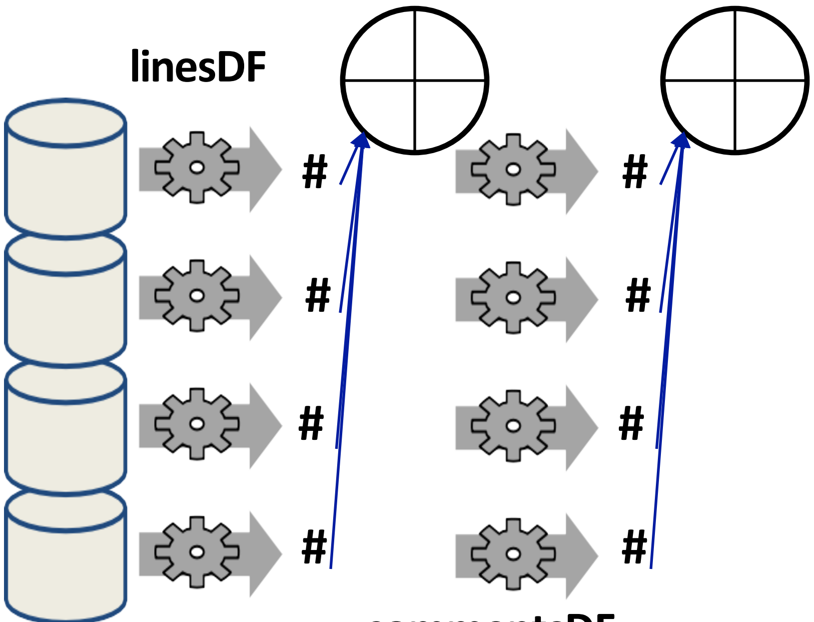


`count()` causes Spark to:

- read data
- sum within partitions
- combine sums in driver

Spark Programming Model

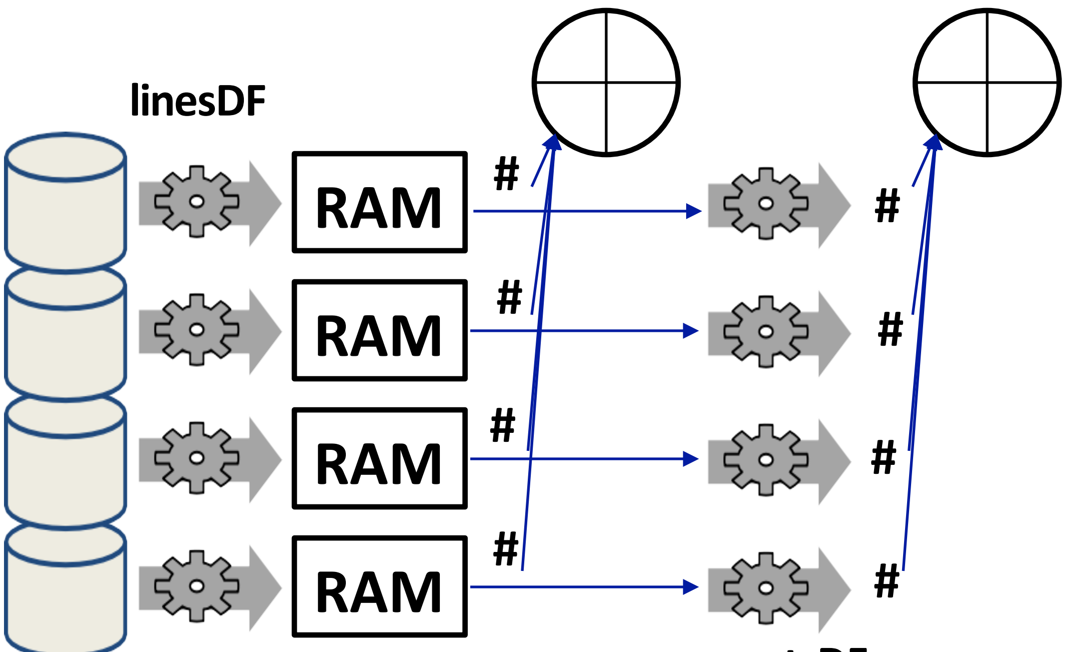
```
linesDF = sqlContext.read.text('...')
commentsDF = linesDF.filter(isComment)
print linesDF.count(), commentsDF.count()
```



Spark recomputes linesDF:
- read data (again)
- sum within partitions
- combine sums in driver

Spark Programming Model

```
linesDF =   sqlContext.read.text('...')
linesDF.cache() # save, don't recompute!
commentsDF =   linesDF.filter(isComment)
print linesDF.count(),commentsDF.count()
```

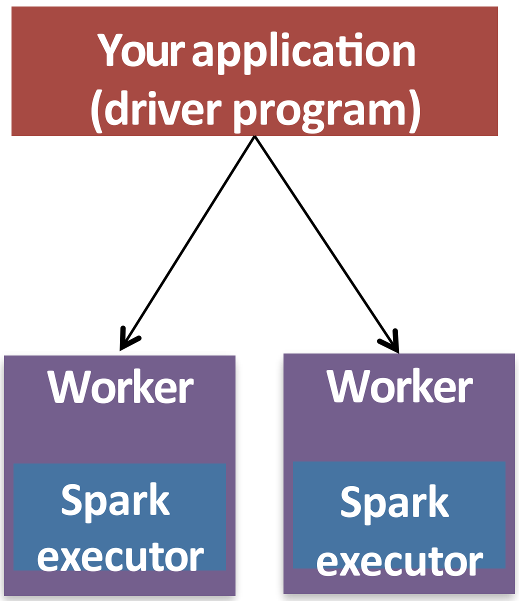


Spark Program Lifecycle

- Create DataFrames from external data or createDataFrame from a collection in driver program
- Lazily transform them into new DataFrames
- `cache()` some DataFrames for reuse
- Perform actions to execute parallel computation and produce results

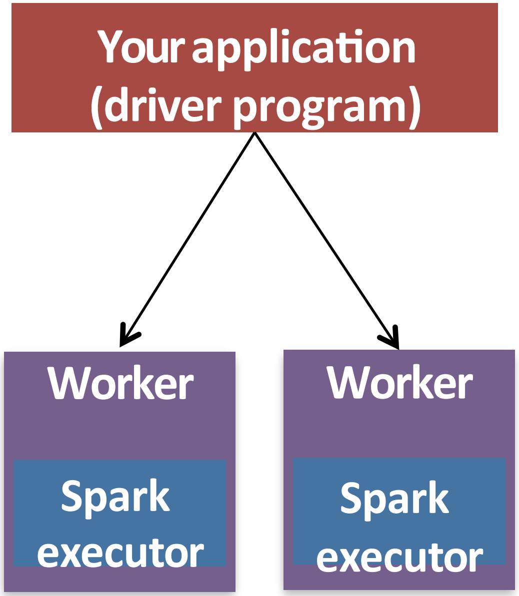
Local or Distributed?

- **Where does code run?**
  - Locally, in the driver
  - Distributed at the executors
  - Both at the driver and the executors
- **Very important question:**
  - Executors run in parallel
  - Executors have much more memory



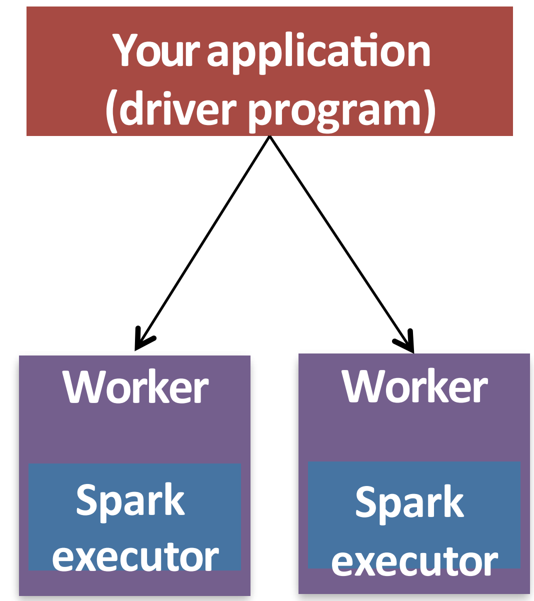
Where Code Runs

- **Most Python code runs in driver**
  - Except for code passed to transformations
- **Transformations run at executors**
- **Actions run at executors and driver**



Examples

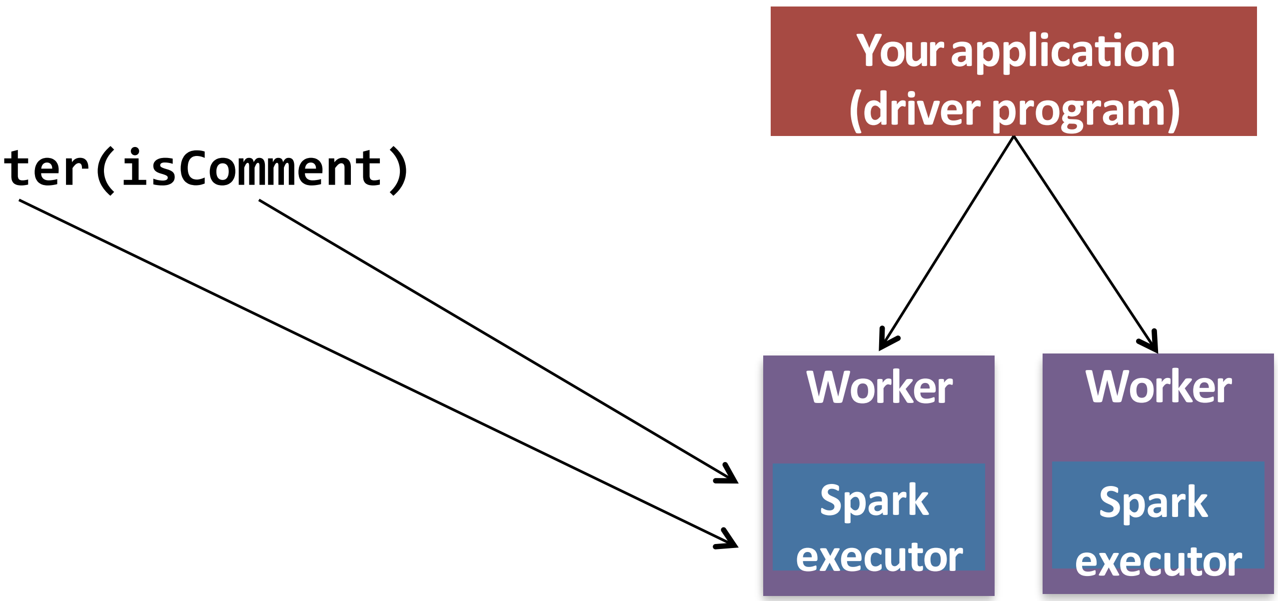
```
>>> a = a + 1
```



Examples

```
>>> a = a + 1

>>> linesDF.filter(isComment)
```

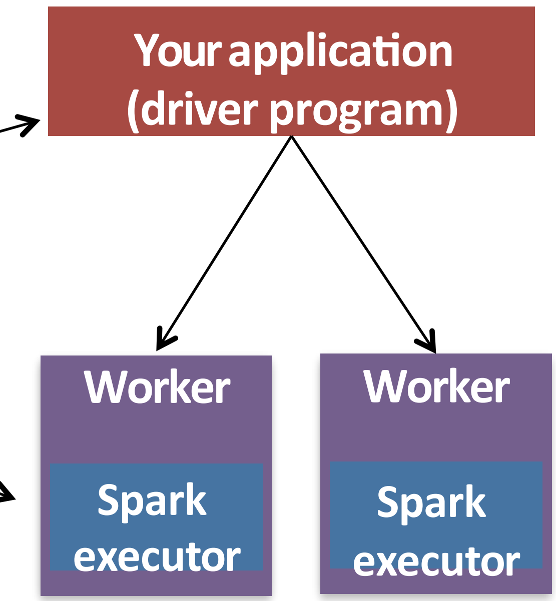


Examples

```
>>> a = a + 1

>>> linesDF.filter(isComment)

>>> commentsDF.count()
```



How Not to Write Code

- Let's say you want to combine two DataFrames: aDF, bDF
- You remember that `df.collect()` returns a list of Row, and in Python you can combine two lists with +
- A naïve implementation would be:

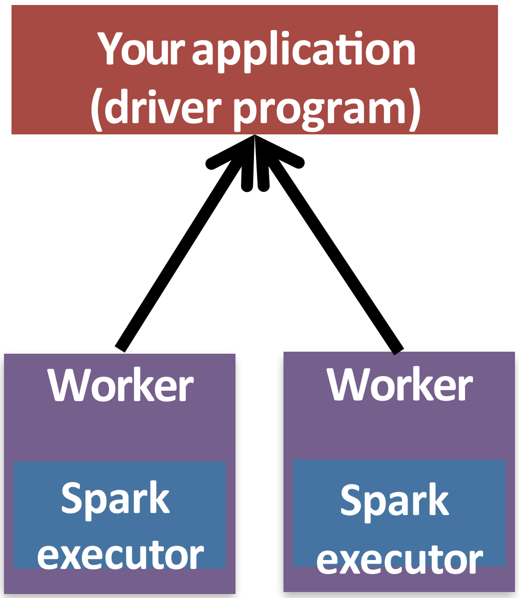
```
>>> a = aDF.collect()
>>> b = bDF.collect()
>>> cDF = sqlContext.createDataFrame(a + b)
```

- Where does this code run?

How Not to Write Code

```
>>> a = aDF.collect()
>>> b = bDF.collect()
```

* all distributed data for a and b is sent to driver

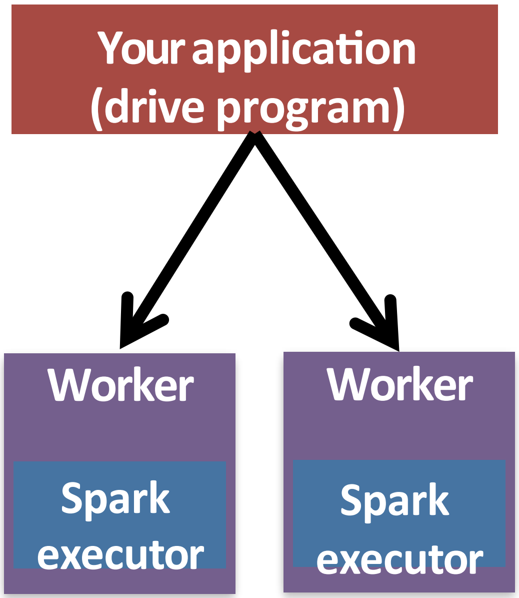


- **What if a and/or b is very large?**
  - Driver could run out of memory: Out Of Memory error (OOM)
  - Also, takes a long time to send the data to the driver

How Not to Write Code

```
>>> cDF = sqlContext.createDataFrame(a + b)
```

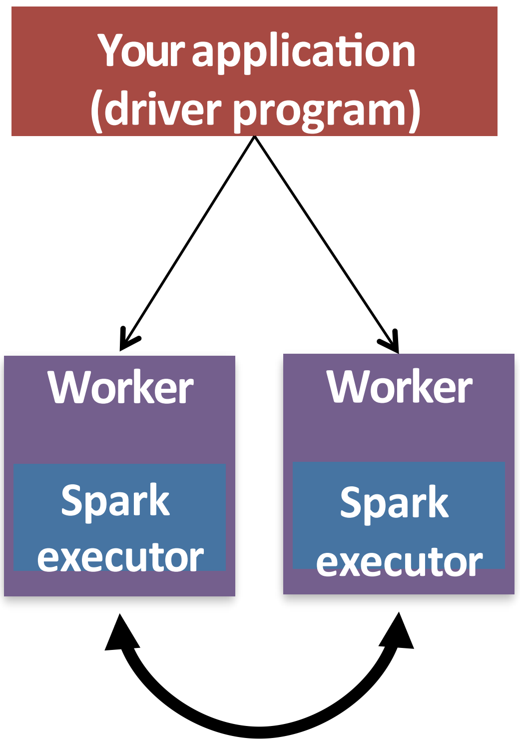
* all data for cDF is sent to the executors



- **What if the list a + b is very large?**
  - Driver could run out of memory: Out Of Memory error (OOM)
  - Also, takes a long time to send the data to executors

How Not to Write Code: The Best Way

```
>>> cDF = aDF.unionAll(bDF)
```



- **Use the `DataFrame` reference API**
  - `unionAll()`
    “Return a new `DataFrame` containing union of rows in this frame and another frame”
- **Runs completely at executors:**
  - Very scalable and efficient

Some Programming Best Practices

- **Use Spark Transformations and Actions wherever possible**
  - Search `DataFrame` reference API
- **Never use `collect()` in production, instead use `take(n)`**
- **`cache()` DataFrames that you reuse a lot**

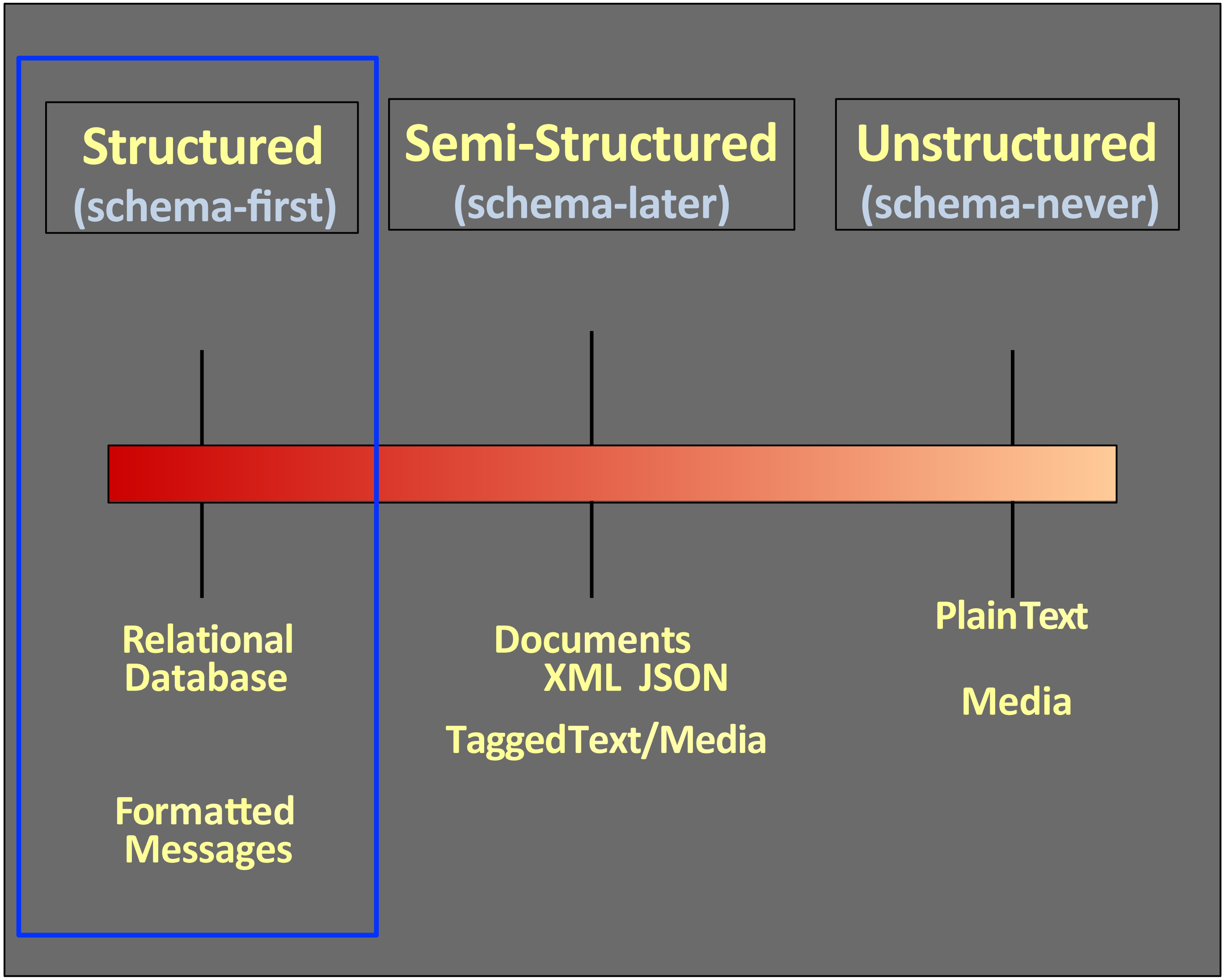
Outline

- DataFrames
- Spark Transformations and Actions
- Spark Programming Model
- **Relational Database and SQL**
- **SparkSQL**

Key Data Management Concepts

- **A data model is a collection of concepts for describing data**
- **A schema is a description of a particular collection of data, using a given data model**
- **A relational data model is the most used data model**
  - Relation, a table with rows and columns
  - Every relation has a schema defining fields in columns

The Structure Spectrum



Next topic

Relational Database: Definitions

- ***Relational database:*** **a set of *relations***
- **Two parts to a *Relation*:**
  - *Schema*: specifies name of relation, plus each column's name and type

```
Students(sid: string, name: string, email: string,
         age: integer, gpa: real)
```

  - *Instance*: the actual data at a given time
    - # rows = *cardinality*
    - # fields = *degree*

What is a Database?

- **A large organized collection of data**
  - Transactions used to modify data

- **Models real world, e.g., enterprise**
  - Entities
    - e.g., teams, games
  - Relationships
    - e.g., A plays against B in The World Cup

Large Databases

- US Internal Revenue Service: 150Terabytes
- Australian Bureau of Stats: 250 Terabytes
- AT&T call records: 312Terabytes
- eBay database: 1.4 Petabytes
- Yahoo click data: 2 Petabytes
- *What matters for these databases?*

Large Databases

- US Internal Revenue Service: 150Terabytes
- Australian Bureau of Stats: 250 Terabytes
- AT&T call records: 312Terabytes
- eBay database: 1.4 Petabytes
- Yahoo click data: 2 Petabytes
- *What matters for these databases?*

Accuracy, Consistency, Durability, Rich queries

Fast, Rich queries

Accuracy, Consistency, Durability

Availability
Timeliness

Example: Instance of Students Re

Attribute names

`Students(`*`sid`*`:string, `*`name`*`:string, `*`login`*`:string, `*`age`*`:integer, `*`gpa`*`:real)`

Table name

sid	name	login	age	gpa
[illegible]66	Jones	jones@eecs	18	3.4
53688	Smith	smith@statistics	18	3.2
53650	Smith	smith@math	19	3.8

Tuples or rows

- **Cardinality = 3 (rows)**
- **Degree = 5 (columns)**
- **All rows (tuples) are distinct.**

SQL – A language for Relational DBs

- **SQL = Structured Query Language**
- **Supported by Spark DataFrames (SparkSQL)**
- **Some of the functionality SQL provides:**
  - Create, modify, delete relations
  - Add, modify, remove tuples
  - ***Specify queries to find tuples matching criteria***

Queries in SQL

- Single-table queries are straightforward
- To find all 18 year old students, we can write:

```
SELECT *
  FROM Students S
 WHERE S.age=18
```

- To find just names and logins:

```
SELECT S.name, S.login
  FROM Students S
 WHERE S.age=18
```

Querying Multiple Relations

- Can specify a *join* over two tables as follows:

```
SELECT S.name, E.cid
 FROM Students S, Enrolled E
 WHERE S.sid=E.sid
```

Enrolled

E

E.sid	E.cid	E.grade
53831	Physics203	A
53650	Topology112	A
53341	History105	B

Students

S

S.sid	S.name	S.login	S.age	S.gpa
53341	Jones	jones@cs	18	3.4
53831	Smith	smith@ee	18	3.2

- First, combine the two tables, S and E

Querying Multiple Relations

- **Cross Join: Cartesian product of two tables (E x S)**

Enrolled

E

E.sid	E.cid	E.grade
53831	Physics203	A
53650	Topology112	A
53341	History105	B

Students

S

S.sid	S.name	S.login	S.age	S.gpa
53341	Jones	jones@cs	18	3.4
53831	Smith	smith@ee	18	3.2

Querying Multiple Relations

- **Cross Join: Cartesian product of two tables (E x S) (cont'd)**

Enrolled

E

E.sid	E.cid	E.grade
53831	Physics203	A
53650	Topology112	A
53341	History105	B

Students

S

S.sid	S.name	S.login	S.age	S.gpa
53341	Jones	jones@cs	18	3.4
53831	Smith	smith@ee	18	3.2

E.sid	E.cid	E.grade	S.sid	S.name	S.login	S.age	S.gpa
53831	Physics203	A	53341	Jones	jones@cs	18	3.4
53650	Topology112	A	53341	Jones	jones@cs	18	3.4
53341	History105	B	53341	Jones	jones@cs	18	3.4
53831	Physics203	A	53831	Smith	smith@ee	18	3.2
53650	Topology112	A	53831	Smith	smith@ee	18	3.2
53341	History105	B	53831	Smith	smith@ee	18	3.2

Querying Multiple Relations

- **Where clause: Choose matching rows using Where clause:**

```
SELECT S.name, E.cid
  FROM Students S, Enrolled E
  WHERE S.sid=E.sid
```

E.sid	E.cid	E.grade	S.sid	S.name	S.login	S.age	S.gpa
53831	Physics203	A	53341	Jones	jones@cs	18	3.4
53650	Topology112	A	53341	Jones	jones@cs	18	3.4
53341	History105	B	53341	Jones	jones@cs	18	3.4
53831	Physics203	A	53831	Smith	smith@ee	18	3.2
53650	Topology112	A	53831	Smith	smith@ee	18	3.2
53341	History105	B	53831	Smith	smith@ee	18	3.2

Querying Multiple Relations

- **Select clause: Filter columns using Select clause:**

```
SELECT S.name, E.cid
 FROM Students S, Enrolled E
 WHERE S.sid=E.sid
```

E.sid	E.cid	E.grade	S.sid	S.name	S.login	S.age	S.gpa
53831	Physics203	A	53341	Jones	jones@cs	18	3.4
53650	Topology112	A	53341	Jones	jones@cs	18	3.4
53341	History105	B	53341	Jones	jones@cs	18	3.4
53831	Physics203	A	53831	Smith	smith@ee	18	3.2
53650	Topology112	A	53831	Smith	smith@ee	18	3.2
53341	History105	B	53831	Smith	smith@ee	18	3.2

Querying Multiple Relations

- **Results**

```
SELECT S.name, E.cid
 FROM Students S, Enrolled E
 WHERE S.sid=E.sid
```

Enrolled

E

E.sid	E.cid	E.grade
53831	Physics203	A
53650	Topology112	A
53341	History105	B

Students

S

S.sid	S.name	S.login	S.age	S.gpa
53341	Jones	jones@cs	18	3.4
53831	Smith	smith@ee	18	3.2

Result =

S.name	E.cid
Jones	History105
Smith	Physics203

Explicit SQL Joins

```
SELECT S.name, E.classid
 FROM Students S INNER JOIN  Enrolled  E  ON  S.sid=E.sid
```

S

S.name	S.sid
Jones	11111
Smith	22222
Brown	33333

E

E.sid	E.classid
11111	History105
11111	DataScience194
22222	French150
44444	English10

Result

S.name	E.classid
Jones	History105
Jones	DataScience194
Smith	French150

Equivalent SQL Join Notations

- Explicit Join notation (preferred):

```
SELECT S.name, E.classid
 FROM Students S INNER JOIN Enrolled E ON S.sid=E.sid
```

```
SELECT S.name, E.classid
 FROM Students S JOIN Enrolled E ON S.sid=E.sid
```

- Implicit join notation (deprecated):

```
SELECT S.name, E.cid
 FROM Students S, Enrolled E  WHERE
 S.sid=E.sid
```

SQL Types of Joins

```
SELECT S.name, E.classid
 FROM Students S INNER JOIN  Enrolled  E ON S.sid=E.sid
```

S

S.name	S.sid
Jones	11111
Smith	22222
Brown	33333

E

E.sid	E.classid
11111	History105
11111	DataScience194
22222	French150
44444	English10

Result

S.name	E.classid
Jones	History105
Jones	DataScience194
Smith	French150

Unmatched keys

The type of join controls how unmatched keys are handled

SQL Joins: Left Outer Join

```
SELECT S.name, E.classid
 FROM Students S LEFT OUTER  JOIN  Enrolled  E  ON  S.sid=E.sid
```

S

S.name	S.sid
Jones	11111
Smith	22222
Brown	33333

E

E.sid	E.classid
11111	History105
11111	DataScience194
22222	French150
44444	English10

Result

S.name	E.classid
Jones	History105
Jones	DataScience194
Smith	French150
Brown	<NULL>

Unmatched keys

SQL Joins: Right Outer Join

```
SELECT S.name, E.classid
 FROM Students S LEFT OUTER  JOIN Enrolled  E ON S.sid=E.sid
```

S

S.name	S.sid
Jones	11111
Smith	22222
Brown	33333

E

E.sid	E.classid
11111	History105
11111	DataScience194
22222	French150
44444	English10

Result

S.name	E.classid
Jones	History105
Jones	DataScience194
Smith	French150
<NULL>	English10

Unmatched keys

Running an SQL query on Spark

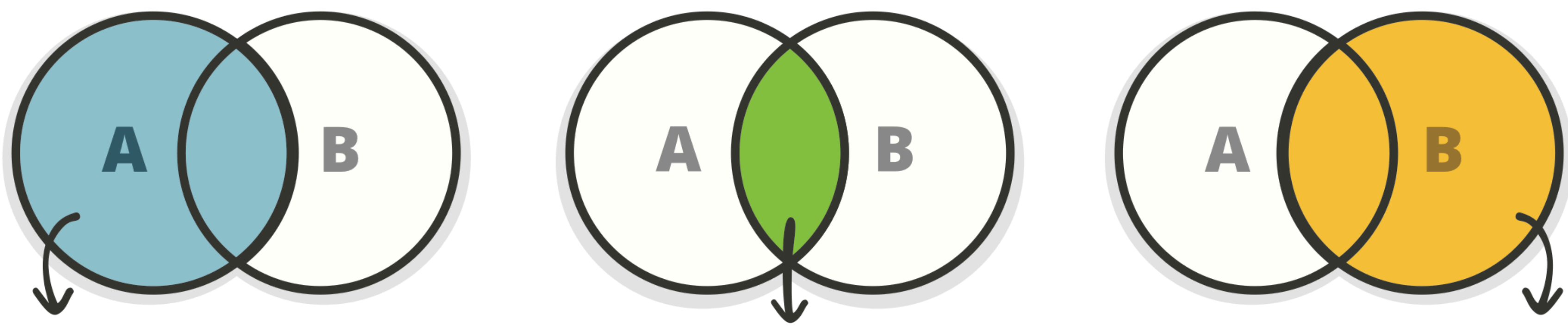
- **`SparkSession.sql(`*`sqlQuery`*`)`**
  - Returns a DataFrame representing the result of the given query.

```
>>> df.createOrReplaceTempView("table1")
>>> df2 = spark.sql("SELECT field1 AS f1, field2 as f2 from table1")
>>> df2.collect()
[Row(f1=1, f2=u'row1'), Row(f1=2, f2=u'row2'), Row(f1=3, f2=u'row3')]
```

- **`createOrReplaceTempView(`*`name`*`)`**
  - Creates or replaces a local temporary view with this DataFrame
  - The lifetime of this temporary table is tied to the `SparkSession` that was used to create this `DataFrame`.

Spark Joins

- SparkSQL and Spark DataFrames support joins
- `join(other, on, how)`:
  - other – right side of the join
  - on –join column name, list of column (names), or join expression
  - how – inner, outer, left_outer, right_outer, ...



LEFT OUTER JOIN - *all rows from table A, even if they do not exist in table B*

INNER JOIN - *fetch the results that exist in both tables*

RIGHT OUTER JOIN - *all rows from table B, even if they do not exist in table A*

Source: https://zeroturnaround.com/rebellabs/sql-cheat-sheet/

Spark Join Examples (1)

- **Inner Join – `X.join(Y, cols)`**
  - Return DF of rows with matching cols in both X and Y

```
>>> df = sqlContext.createDataFrame(data, ['name', 'age'])
[Row(name=u'Alice', age=1), Row(name=u'Bob', age=2)]
>>> df2 = sqlContext.createDataFrame(data2,['name', 'height'])
[Row(name=u'Chris', height=80), Row(name=u'Bob', height=85)]

>>> df.join(df2, 'name')
[Row(name=u'Bob', age=2, height=85)]
```

Spark Join Examples (2)

- **Inner Join – `X.join(Y, cols)`**
  - Return DF of rows with matching cols in both X and Y

```
>>> df = sqlContext.createDataFrame(data, ['name', 'age'])
[Row(name=u'Alice', age=1), Row(name=u'Bob', age=2)]
>>> df2 = sqlContext.createDataFrame(data2,['name', 'height'])
[Row(name=u'Chris', height=80), Row(name=u'Bob', height=85)]

>>> df.join(df2, 'name').select(df.name, df2.height)
[Row(name=u'Bob', height=85)]
```

Spark Join Examples (3)

- **Outer Join – `X.join(Y, cols,'outer')`**
  - Return DF of rows with matching cols in either X and Y

```
>>> df = sqlContext.createDataFrame(data, ['name', 'age'])
[Row(name=u'Alice', age=1), Row(name=u'Bob', age=2)]
>>> df2 = sqlContext.createDataFrame(data2,['name', 'height'])
[Row(name=u'Chris', height=80), Row(name=u'Bob', height=85)]

>>> df.join(df2, 'name', 'outer')
[Row(name=u'Chris', age=None, height=80),
Row(name=u'Alice', age=1, height=None),
Row(name=u'Bob', age=2, height=85)]
```

Spark Join Examples (4)

- **Outer Join – `X.join(Y, cols,'outer')`**
  - Return DF of rows with matching cols in either X and Y

```
>>> df = sqlContext.createDataFrame(data, ['name', 'age'])
[Row(name=u'Alice', age=1), Row(name=u'Bob', age=2)]
>>> df2 = sqlContext.createDataFrame(data2,['name', 'height'])
[Row(name=u'Chris', height=80), Row(name=u'Bob', height=85)]

>>> df.join(df2, 'name', 'outer').select('name', 'height')
[Row(name=u'Chris', height=80),
 Row(name=u'Alice', height=None),
 Row(name=u'Bob', height=85)]
```

Spark Join Examples (5)

- **Left Outer Join – `X.join(Y, cols, 'left_outer')`**
  - Return DF of rows with matching cols in X

```
>>> df = sqlContext.createDataFrame(data, ['name', 'age'])
[Row(name=u'Alice', age=1), Row(name=u'Bob', age=2)]
>>> df2 = sqlContext.createDataFrame(data2,['name', 'height'])
[Row(name=u'Chris', height=80), Row(name=u'Bob', height=85)]

>>> df.join(df2, 'name', 'left_outer')
[Row(name=u'Alice', age=1, height=None),
Row(name=u'Bob', age=2, height=85)]
```

Spark Join Examples (6)

- **Right Outer Join – `X.join(Y, cols, 'right_outer')`**
  - Return DF of rows with matching cols in Y

```
>>> df = sqlContext.createDataFrame(data, ['name', 'age'])
[Row(name=u'Alice', age=1), Row(name=u'Bob', age=2)]
>>> df2 = sqlContext.createDataFrame(data2,['name', 'height'])
[Row(name=u'Chris', height=80), Row(name=u'Bob', height=85)]

>>> df.join(df2, 'name', 'right_outer')
[Row(name=u'Chris', age=None, height=80),
Row(name=u'Bob', age=2, height=85)]
```