Big Data Tools and Techniques

Chapter 7: Introduction to Spark (Part B)

Chapter Overview

Apache Spark is a lightning-fast cluster computing technology, designed for fast computation. Based on Hadoop MapReduce, it extends the MapReduce model to efficiently use it for more types of computations such as interactive queries and stream processing. The core feature of Spark is its in-memory cluster computing that increases the processing speed of an application.

Spark is designed to cover a wide range of workloads such as batch applications, iterative algorithms, interactive queries and streaming, thereby reducing the management burden of maintaining separate tools.

Chapter Overview (Objectives)

- Learn about Apache Spark's history and development
- Familiarize with Apache Spark as a lightning-fast cluster computing technology
- Appreciate Extract-Transform-Load operations, data analytics and visualization
- Understand the concepts: RDDs, DataFrames
- Know the implementation essentials - Transformations, Actions, pySpark, SparkSQL

Part 2: Spark in Practice

- One machine cannot process or store all the data!
- Solution is to distribute data over cluster of machines and process in memory.
- Provides programming abstraction and parallel runtime to hide complexities of fault-tolerance and slow machines
- PySpark provides an easy-to-use programming abstraction and parallel runtime where Spark dataframe is the key concept.

Background

Spark is for:

Scalable, efficient *analysis* of *Big Data*

Where does big data come from?

It's all happening online – could record every:
» Click
» Ad impression
» Billing event
» Fast Forward, pause,... » Server request
» Transaction
» Network message
» Fault

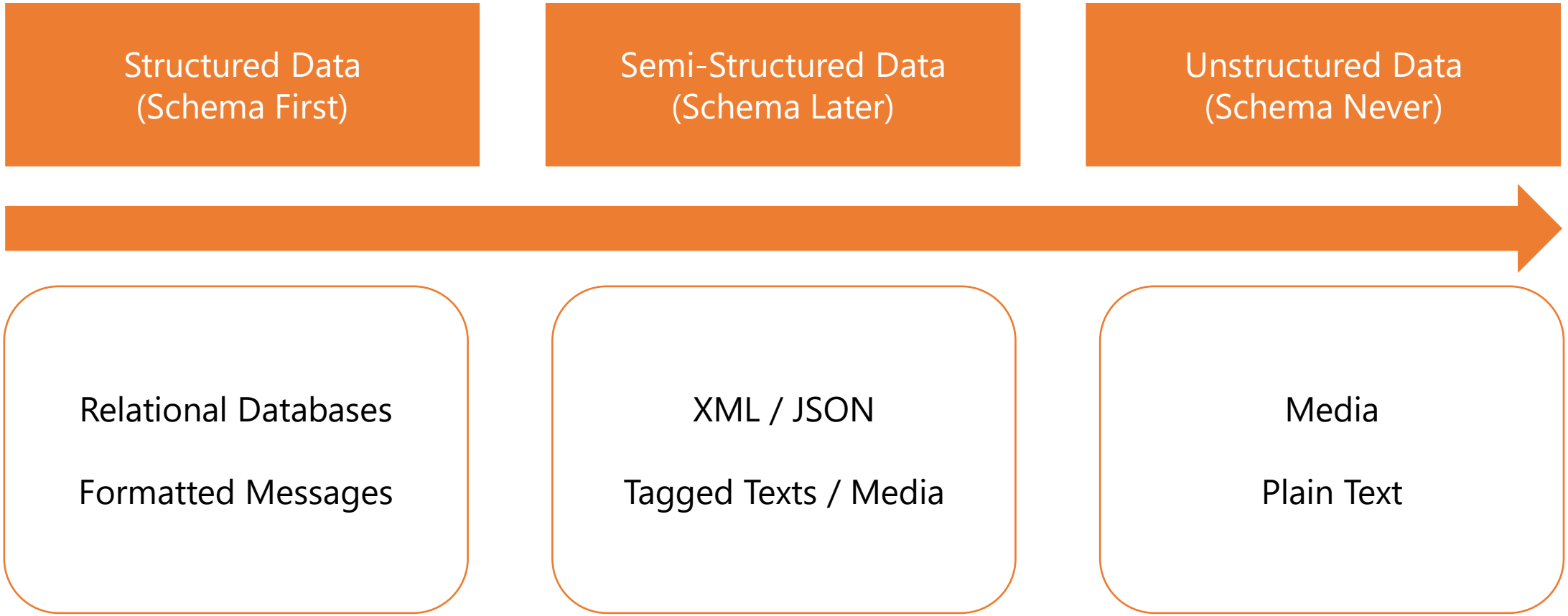
User Generated Content (Web & Mobile)
» Facebook
» Instagram
»Yelp
» TripAdvisor
» Twitter
» YouTube »...

Data Management Concepts - Revisited

- 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

The Structure Spectrum (1/4)



The Structure Spectrum (2/4)

Fortune 500 Companies

	A	B	C	D	E	F	G	H	I
1	rank	company	cik	ticker	sic	state_location	state_of_incorporation	revenues	profits
2	1	Wal-Mart Stores	104169	WMT	5331	AR	DE	421849	16389
3	2	Exxon Mobil	34088	XOM	2911	TX	NJ	354674	30460
4	3	Chevron	93410	CVX	2911	CA	DE	196337	19024
5	4	ConocoPhillips	1163165	COP	2911	TX	DE	184966	11358
6	5	Fannie Mae	310522	FNM	6111	DC	DC	153825	-14014
7	6	General Electric	40545	GE	3600	CT	NY	151628	11644
8	7	Berkshire Hathaway	1067983	BRKA	6331	NE	DE	136185	12967
9	8	General Motors	1467858	GM	3711	MI	MI	135592	6172
10	9	Bank of America Corp.	70858	BAC	6021	NC	DE	134194	-2238
11	10	Ford Motor	37996	F	3711	MI	DE	128954	6561
12	11	Hewlett-Packard	47217	HPQ	3570	CA	DE	126033	8761
13	12	AT&T	732717	T	4813	TX	DE	124629	19864
14	13	J.P. Morgan Chase & Co.	19617	JPM	6021	NY	DE	115475	17370
15	14	Citigroup	831001	C	6021	NY	DE	111055	10602
16	15	McKesson	927653	MCK	5122	CA	DE	108702	1263
17	16	Verizon Communications	732712	VZ	4813	NY	DE	106565	2549
18	17	American International Group	5272	AIG	6331	NY	DE	104417	7786
19	18	International Business Machines	51143	IBM	3570	NY	NY	99870	14833
20	19	Cardinal Health	721371	CAH	5122	OH	OH	98601.9	642.2
21	20	Freddie Mac	37785	FMC	2800	PA	DE	98368	-14025

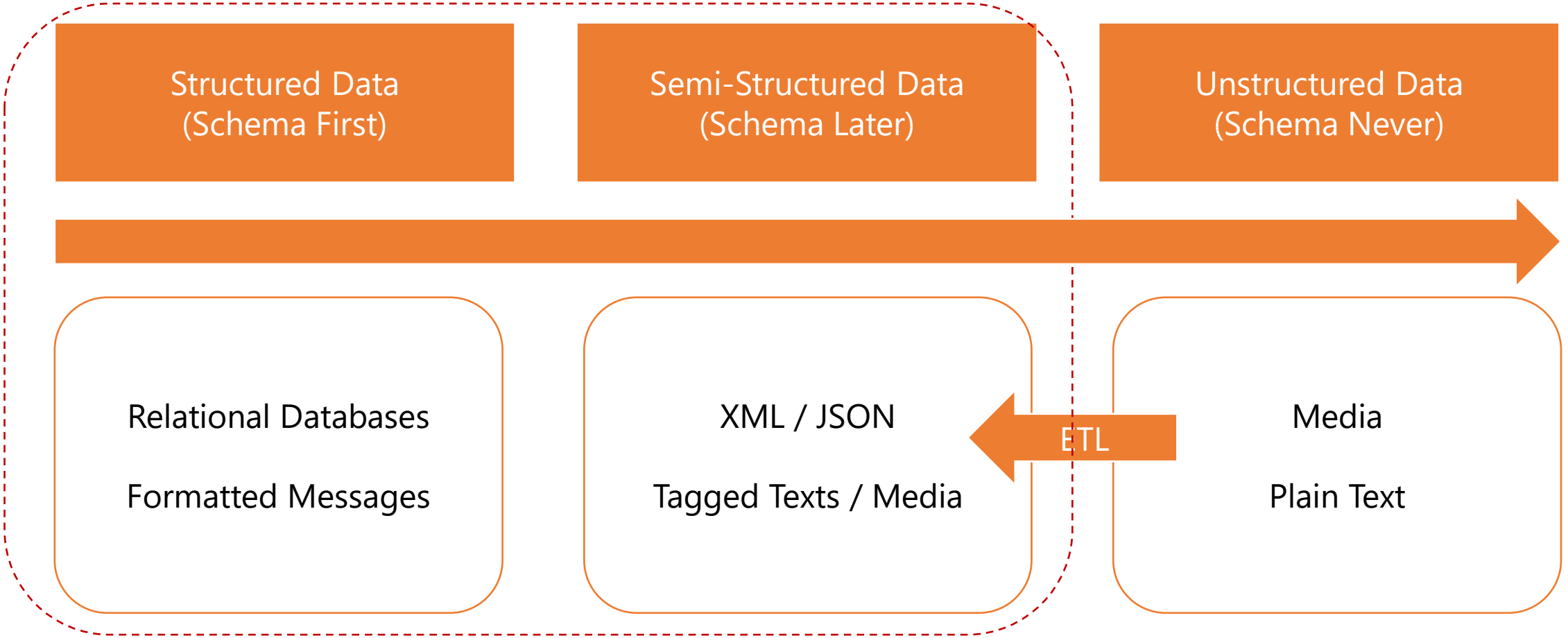
The Structure Spectrum (3/4)

Students Relation (Table)

Students(sid:string, name:string, email:string, age:integer, score: integer)

sid	name	email	age	score
1	Alice	Alice@math	19	70
2	Bob	Bob@math	19	80
3	Candy	Candy@physics	20	85

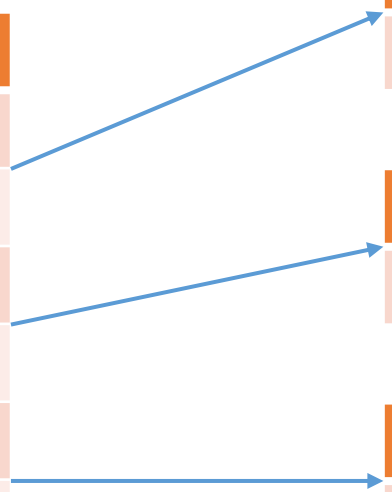
The Structure Spectrum (4/4)



Distributed Memory

Big Data

Word	Index	Count
I	0	1
am	1	1
John	2	1
I	3	1
am	4	1
John	5	1



I	0	1
am	1	1

Partition 1

John	2	1
I	3	1

Partition 2

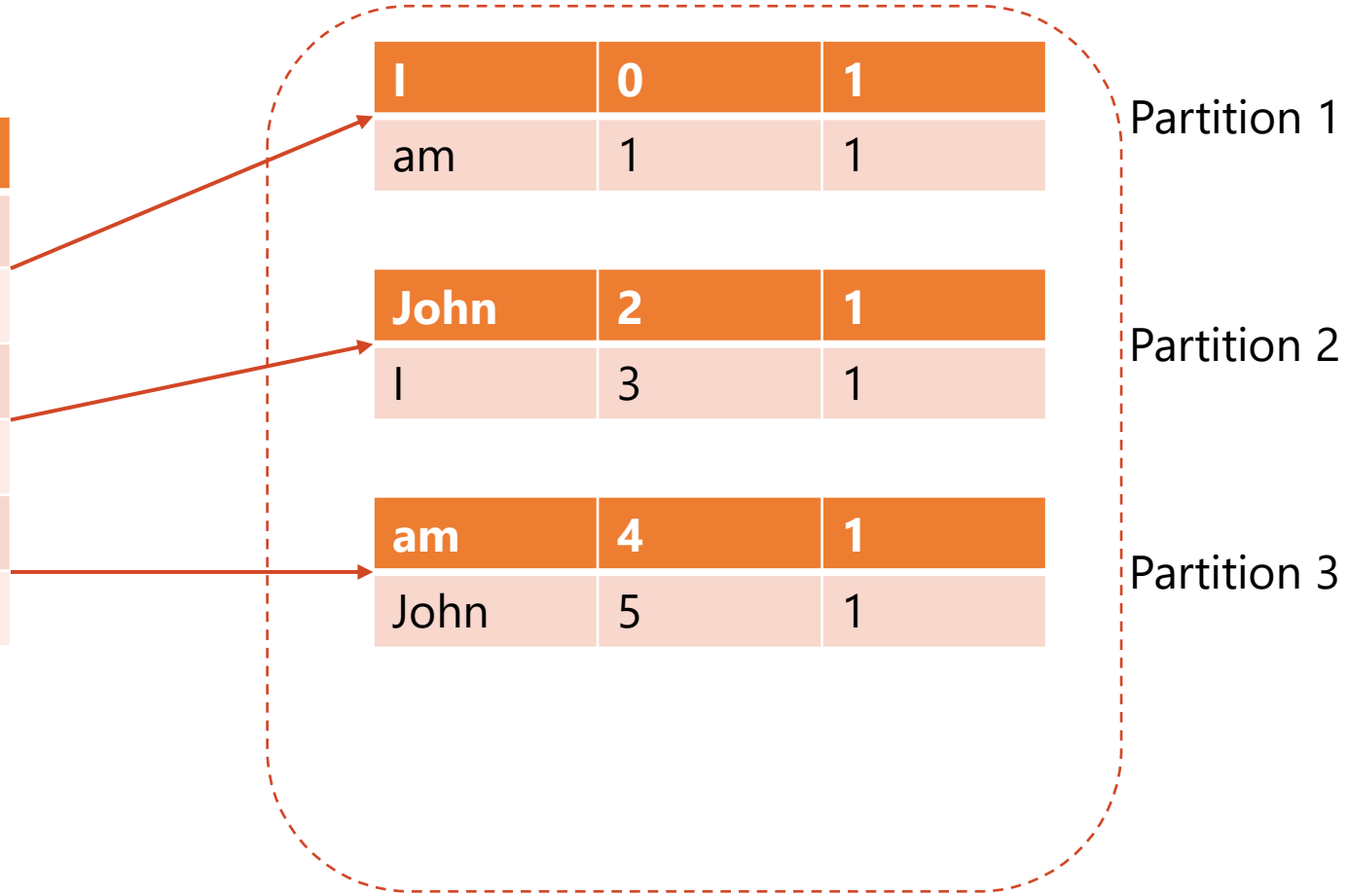
am	4	1
John	5	1

Partition 3

Spark DataFrames

Big Data

Word	Index	Count
I	0	1
am	1	1
John	2	1
I	3	1
am	4	1
John	5	1

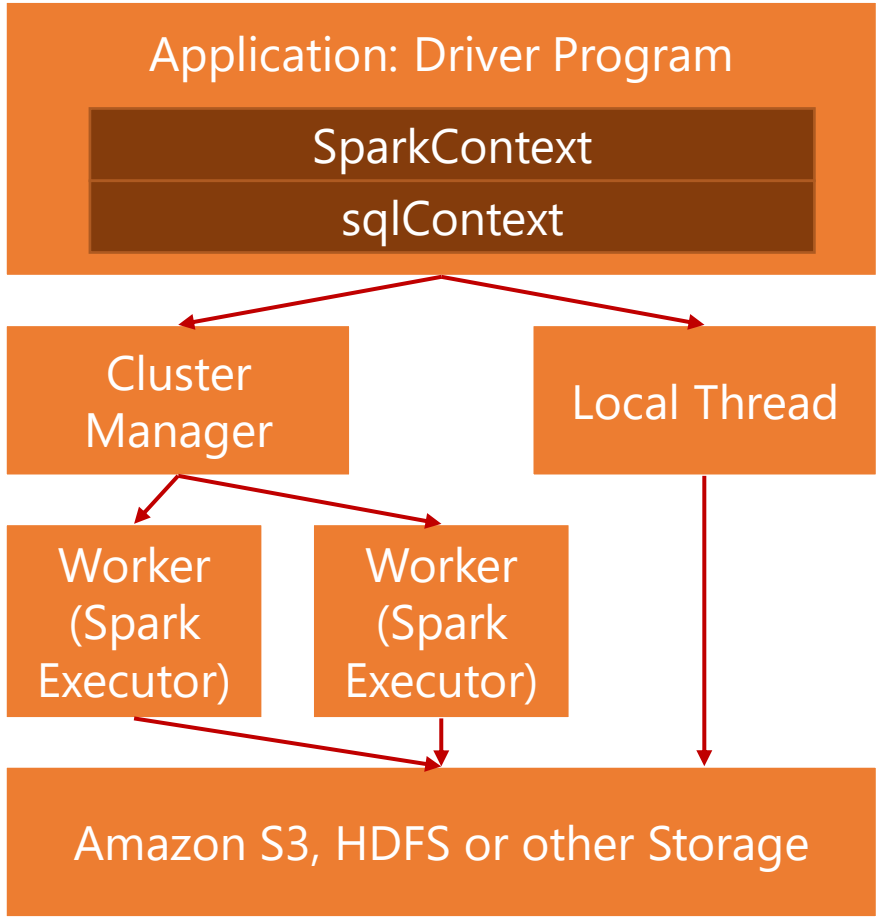


Spark Driver and Workers

A Spark program has two sub-programs:
- A driver program
- A workers program

Worker programs run on cluster nodes or in local threads

DataFrames are distributed across workers



Spark and SQL Context

- A Spark program first creates a SparkContext object
- SparkContext tells Spark how and where to access a cluster
- pySpark shell, Databricks automatically create SparkContext
- iPython and programs must create a new SparkContext

- The program next creates a sqlContext object
- Use sqlContext to create DataFrames

Spark DataFrames

- The primary abstraction in Spark
- Immutable once constructed
- Track lineage information to efficiently recompute lost data
- Enable operations on collection of elements in parallel

- You 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 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=19)
>> row
Row(age=19 name='Alice')
>> row['name'], row['age']
('Alice', 19)
>> row.name, row.age
('Alice', 19)
```

DataFrames

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

Dataframes

- Create a DataFrame from a data source: createDataFrame
- Apply *transformations* to a DataFrame: select, filter
- Apply *actions* to a DataFrame: show, count

Creating DataFrames

- Create DataFrames from Python collections (lists)

```
>> data = [('Bob', 19), ('Candy',20)]
>> data
[('Bob', 19), ('Candy',20)]
>> df = sqlContext.createDataFrame(data)
[Row(_1=u'Bob', _2=19), Row(_1=u'Candy', _2=20)]
>> sqlContext.createDataFrame(data, ['name','age'])
[Row(name=u'Bob', age=19), Row(name='Candy', age=20)]
```

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

Python Data Analysis Library (PANDAS)

- 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
>> 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…

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

Creating DataFrames

- 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

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

Column Transformations

- The dot notation creates a DataFrame from one column:

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

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

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

‘*’ implies selection of all columns.

```
>>> df.select(’name’, ‘score’)
```

```
>>> df.select(df.name, (df.score+5).alias(‘updatedscore’) )
```

Column Transformations

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

```
>> df.drop(df.age)
```

User Defined Functions

- Transform a DataFrame using a User Defined Function

```
>> from pyspark.sql.types import IntegerType
>> stringlength = udf(lambda string: len(string), IntegerType())
>> df.select(stringlength(df.name).alias('namelength'))
[Row(namelength=3), Row(namelength=5)]
```

- 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

```
>> df = sqlContext.createDataFrame(data, [‘name’, ’age’])
[Row(name=u’Bob’, age=19), Row(name=u’Candy’, age=20)]

>> from pyspark.sql.types import IntegerType
>> moderate = udf(lambda x: x + 10, IntegerType())
>> df2 = df.select(df.name, moderate(df.score).alias(‘updatedscore’))
[Row(name=u’Bob’, updatedscore=90), Row(name=‘Candy’, updatedscore=95)]

>> df3 = df2.filter(df2.updatedscore > 90)
[Row(name=u’Candy’, updatedscore=95)]
```

Using Transformations

```
>> data2 = [('Alice', 19),('Candy', 20), ('Candy', 20)]
>> df  = sqlContext.createDataFrame(data2, ['name', 'age'])
[Row(name=u'Alice', age=19), Row(name=u'Candy', age=20),
Row(name=u'Candy', age=20)]
>> df2 = df.distinct()
[Row(name=u'Alice', age=19), Row(name=u'Candy', age=20)]

>> df3 = df2.sort("age", ascending=False)
[Row(name=u'Candy', age=20), Row(name=u'Alice', age=19)]
```

Using Transformations

```
>> data3 = [Row(name='Alice', scorelist=[60, 70, 80])]
>> df4 = sqlContext.createDataFrame(data3)
[Row(name=u'Alice', scorelist=[60, 70,80])]
>> df4.select(explode(df4.scorelist).alias('score'))
[Row(score=60), Row(score=70), Row(score=80)]
```

Groupby Transformations

- groupBy(**cols*) groups the DataFrame using the specified columns, so as to 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

Groupby Transformations

```
>> data = [('Alice', 19, 70), ('Bob', 19, 80), ('Bob', 19, 90)]
>> df = sqlContext.createDataFrame(data, ['name', 'age', 'score'])
>> df1 = df.groupBy(df.name)
>> df1.agg({"*":"count"}).collect()
[Row(name=u'Alice', count(1)=1), Row(name=u'Bob', count(1)=2)]

>> df.groupBy(df.name).count()
[Row(name=u'Alice', count=1), Row(name=u'Bob', count=2)]
```

Groupby Transformations

```
>> data = [('Alice', 19, 70), ('Bob', 19, 80), ('Bob', 19, 90)]
>> df = sqlContext.createDataFrame(data, ['name', 'age', 'score'])
>> df.groupBy().avg().collect()
[Row(avg(age)=19, avg(score)=80)]

>> df.groupBy('name').avg('age','score').collect()
[Row(name=u'Alice', avg(age)=19, avg(score)=70),
Row(name=u'Bob', avg(age)=19, avg(score)=85)]
```

Transforming a Dataframe

```
>> codeDF = sqlContext.read.text(‘…’)
>> comments = codeDF.filter(isComment)
```

Spark Actions

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

Action	Description
show(n)	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

Note: count for DataFrames is an action, but for GroupedData it is a transformation.

Showing Data

```
>> df.groupBy('name').avg('age', 'score').collect()
[Row(name=u'Alice', avg(age)=19, avg(score)=70),
Row(name=u'Bob', avg(age)=19, avg(score)=85)]
>> df.show()
>> df.count()
```

Showing Data

```
>> data = [('Alice', 70), ('Bob', 80), ('Candy', 90)]
>> df = sqlContext.createDataFrame(data, ['name','score'])
>> df.take(1)
>> df.describe()
```

Caching DataFrames

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

codeDF.cache()
```

- *Note: save, minimize on recompute!*

```
commentsDF = codeDF.filter(isComment)

print codeDF.count(), commentsDF.count()
```

Spark Program Flow

- 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

Spark Processes

- **Where does code run?**

- Locally, in the driver

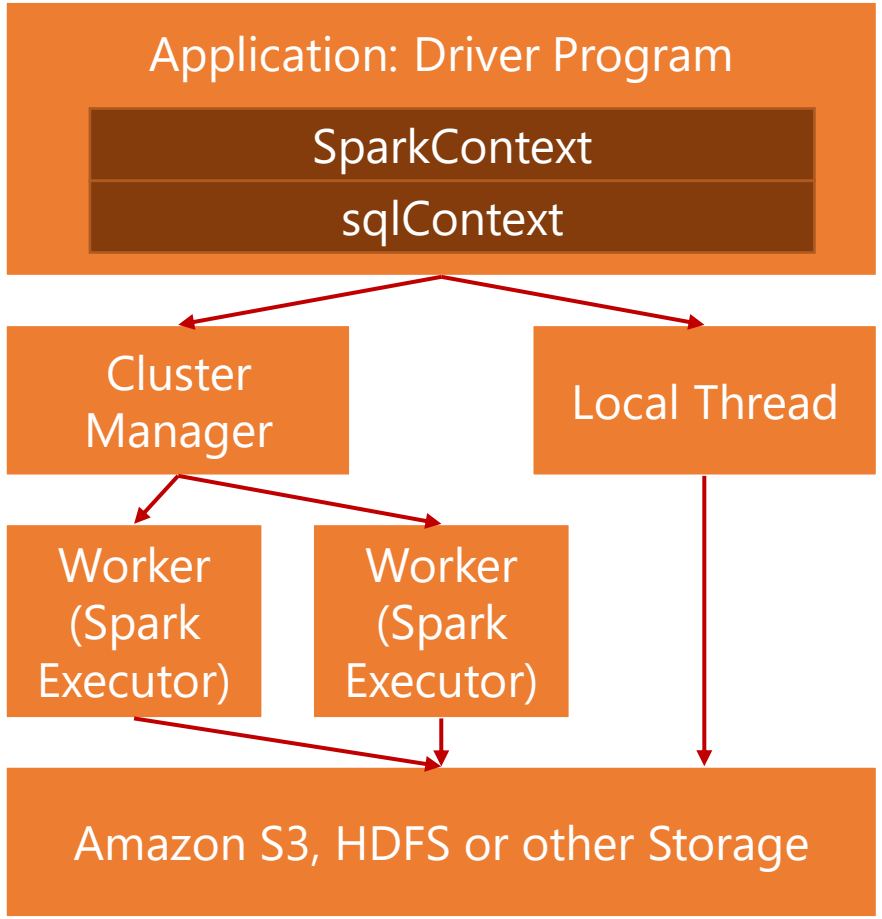
- Distributed at the executors

- Both at the driver and the executors

- **Very important to note:**

- Executors run in parallel

- Executors have much more memory

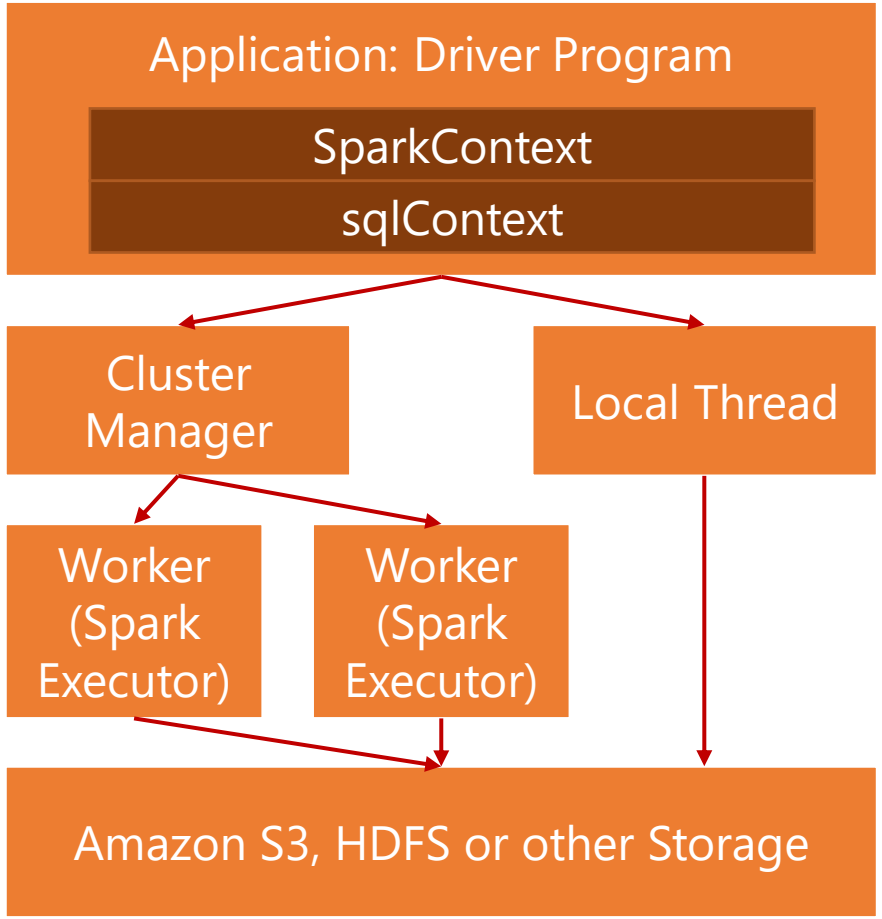


Spark Processes

- Most Python code runs in driver
- Except for code passed to transformations

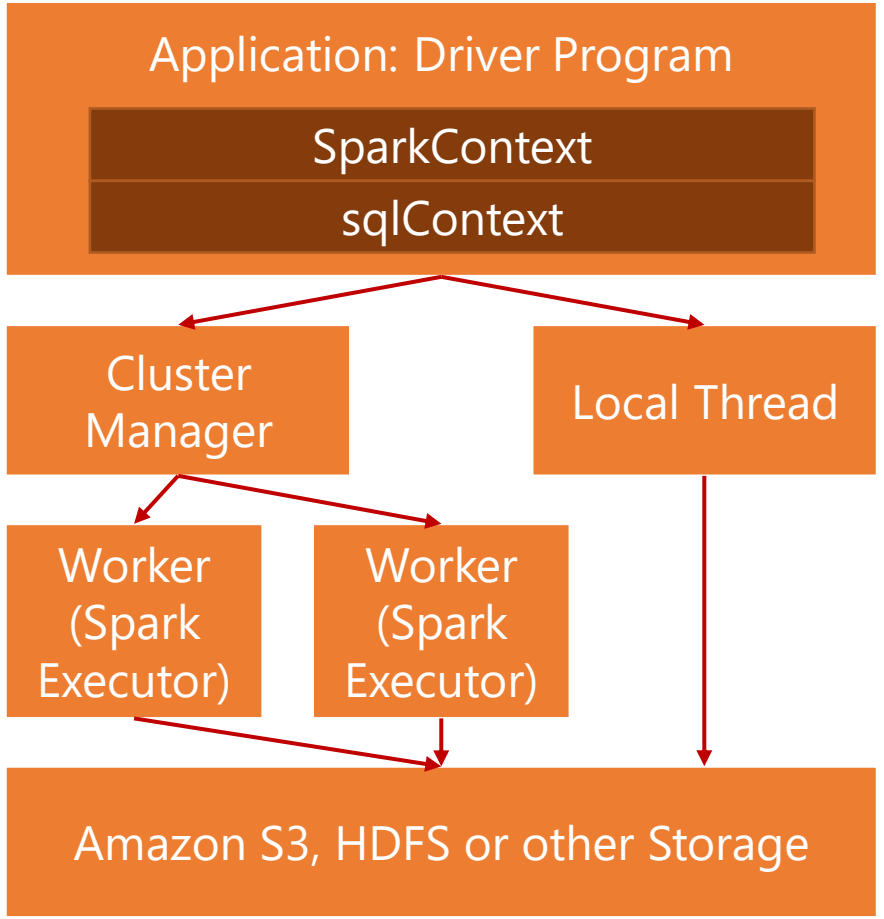
- **Transformations** run at executors

- **Actions** run at executors and driver



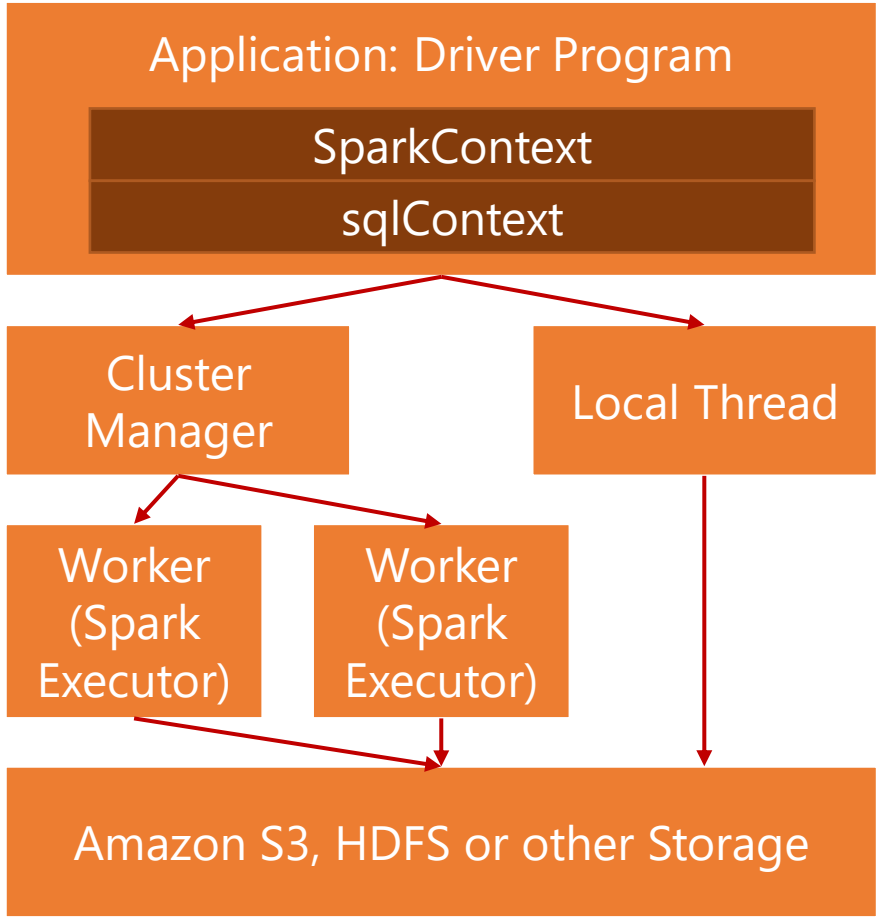
Spark Processes

- `x = x + 10`
- `codeDF.filter(isComment)`
- `commentsDF.count()`



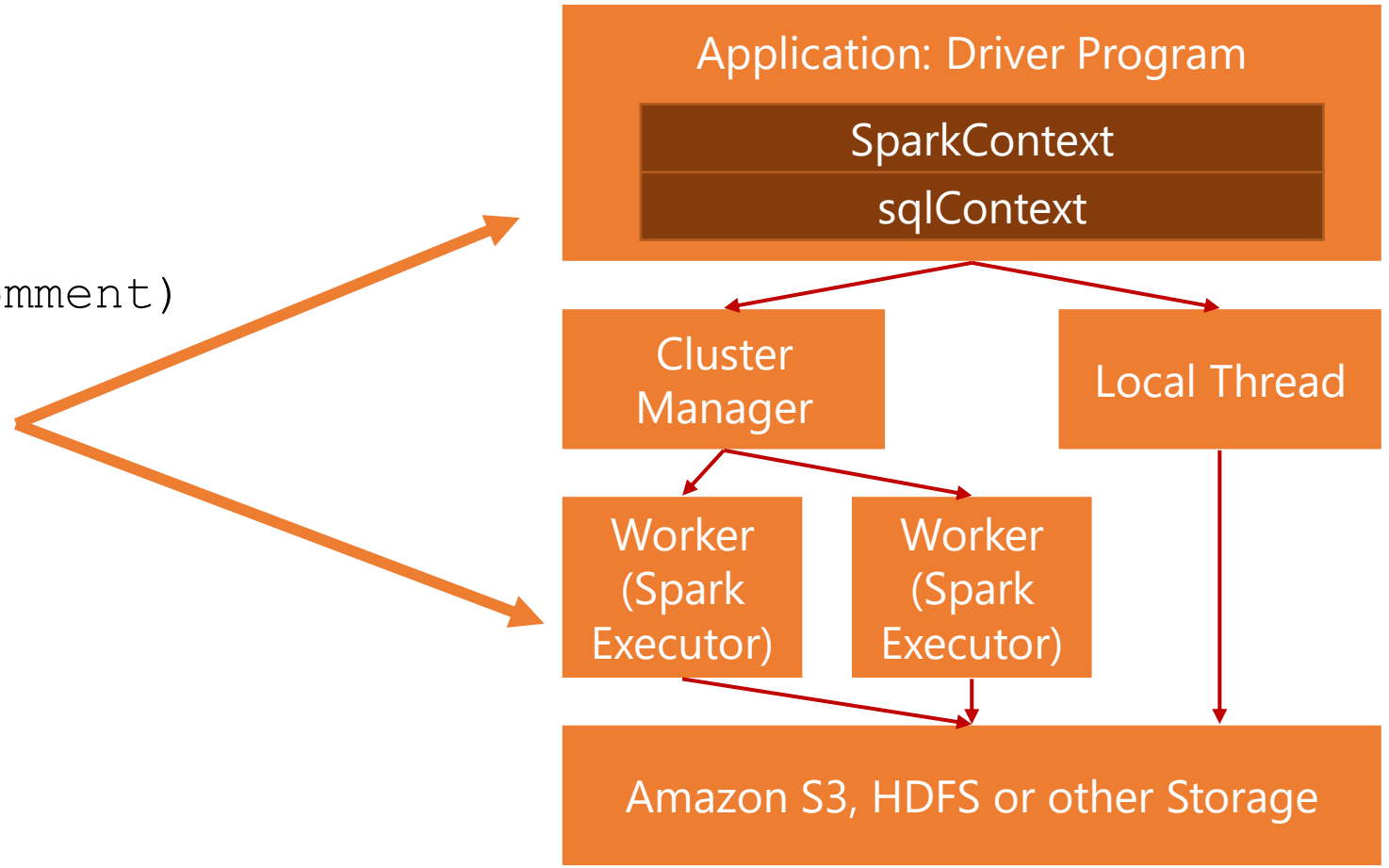
Spark Processes

- `x = x + 10`
- `codeDF.filter(isComment)`
- `commentsDF.count()`



Spark Processes

- `x = x + 10`
- `codeDF.filter(isComment)`
- `commentsDF.count()`



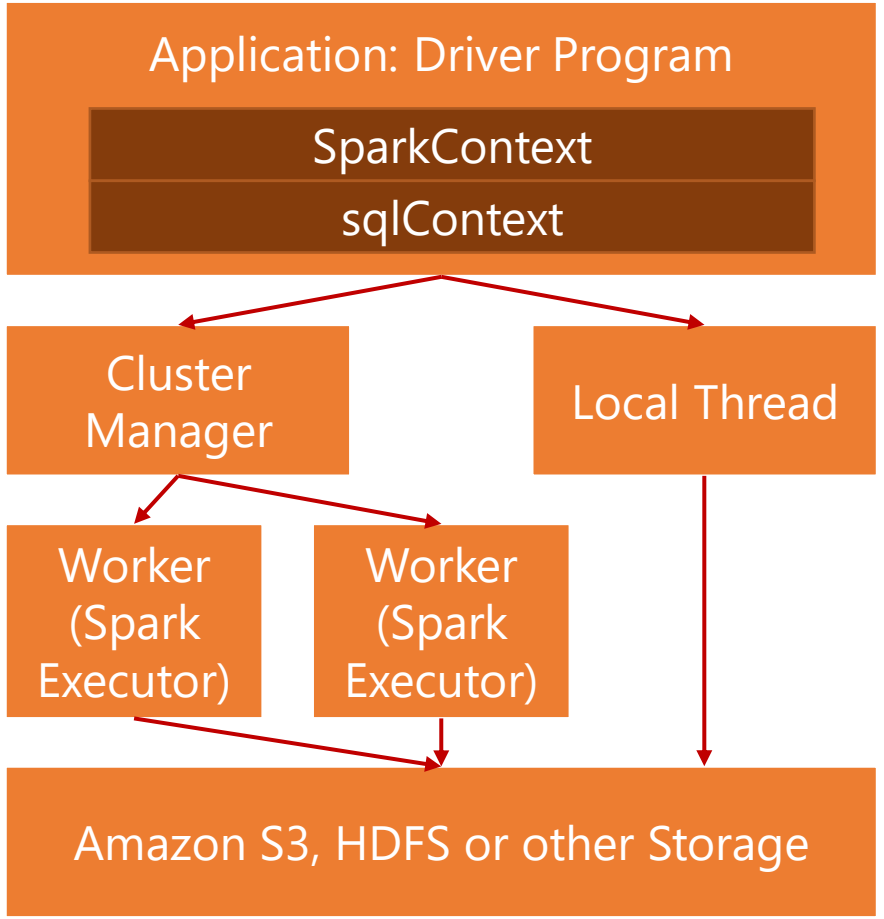
Spark Processes

```
>>x = dfX.collect()
>>y = dfY.collect()
>>dfZ = sqlContext.createDataFrame(x+y)
```

Where do these statements run?

What if the list x + y is too big?

- Driver may run out of memory
- Plus it takes a long while to send the data to executors

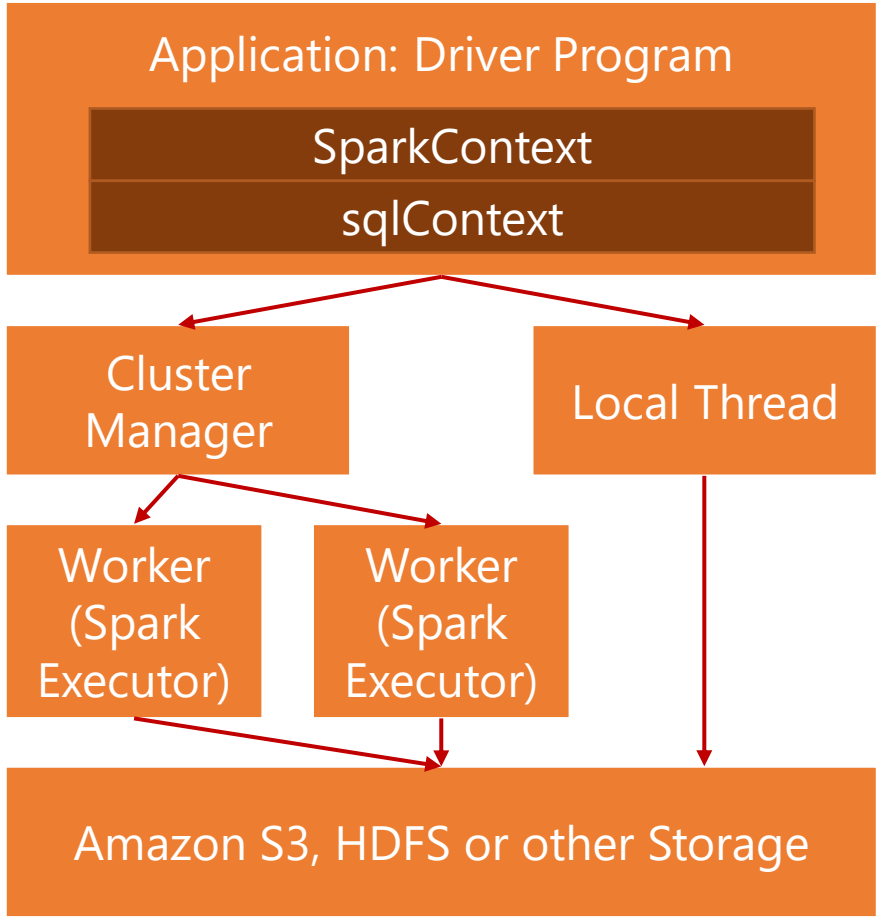


Spark Processes

Better to use

```
>> dfZ = dfX.union(dfY)
```

- union(): “Return a new DataFrame containing union of rows across two dataframes”
- Runs at executors.



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