Big Data Analytics
Map-Reduce and Spark

Slides by:
Joseph E. Gonzalez
jegonzal@cs.berkeley.edu

With revisions by:
Josh Hug
hug@cs.Berkeley.edu
From SQL to Big Data (with SQL)

- Last week...
  - Databases
  - (Relational) Database Management Systems
  - SQL: Structured Query Language

- Today
  - More on databases and database design
  - Enterprise data management and the data lake
  - Introduction to distributed data storage and processing
  - Spark
Data in the Organization

A little bit of buzzword bingo!

Operational Data Store

Data Warehouse

ETL (Extract, Transform, Load)

OLAP (Online Analytics Processing)

Star Schema

Snowflake Schema

CUBE

ROLLUP

Drill Down

Schema on Read

Data Lake
Inventory

How we like to think of data in the organization
The reality...

Sales (Asia)  
Inventory  
Sales (US)  
Advertising
Operational Data Stores

- Capture *the now*
- Many different databases across an organization
- Mission critical... be careful!
  - Serving live ongoing business operations
  - Managing inventory
- Different formats (e.g., currency)
  - Different schemas (acquisitions ...)
- Live systems often don’t maintain history

We would like a consolidated, clean, historical snapshot of the data.
Data Warehouse

Collects and organizes historical data from multiple sources

Data is periodically ETLed into the data warehouse:

- **Extracted** from remote sources
- **Transformed** to standard schemas
- **Loaded** into the (typically) relational (SQL) data system
**Extract** $\rightarrow$ **Transform** $\rightarrow$ **Load** (ETL)

**Extract & Load:** provides a snapshot of operational data
- Historical snapshot
- Data in a single system
- Isolates analytics queries (e.g., Deep Learning) from business critical services (e.g., processing user purchase)
- Easy!

**Transform:** clean and prepare data for analytics in a unified representation
- **Difficult** $\rightarrow$ often requires specialized code and tools
- Different schemas, encodings, granularities
Data Warehouse

Collects and organizes historical data from multiple sources

How is data organized in the Data Warehouse?
## Example Sales Data

<table>
<thead>
<tr>
<th>pname</th>
<th>category</th>
<th>price</th>
<th>qty</th>
<th>date</th>
<th>day</th>
<th>city</th>
<th>state</th>
<th>country</th>
</tr>
</thead>
<tbody>
<tr>
<td>Corn</td>
<td>Food</td>
<td>25</td>
<td>25</td>
<td>3/30/16</td>
<td>Wed.</td>
<td>Omaha</td>
<td>NE</td>
<td>USA</td>
</tr>
<tr>
<td>Corn</td>
<td>Food</td>
<td>25</td>
<td>8</td>
<td>3/31/16</td>
<td>Thu.</td>
<td>Omaha</td>
<td>NE</td>
<td>USA</td>
</tr>
<tr>
<td>Corn</td>
<td>Food</td>
<td>25</td>
<td>15</td>
<td>4/1/16</td>
<td>Fri.</td>
<td>Omaha</td>
<td>NE</td>
<td>USA</td>
</tr>
<tr>
<td>Galaxy</td>
<td>Phones</td>
<td>18</td>
<td>30</td>
<td>1/30/16</td>
<td>Wed.</td>
<td>Omaha</td>
<td>NE</td>
<td>USA</td>
</tr>
<tr>
<td>Galaxy</td>
<td>Phones</td>
<td>18</td>
<td>20</td>
<td>3/31/16</td>
<td>Thu.</td>
<td>Omaha</td>
<td>NE</td>
<td>USA</td>
</tr>
<tr>
<td>Galaxy</td>
<td>Phones</td>
<td>18</td>
<td>50</td>
<td>4/1/16</td>
<td>Fri.</td>
<td>Omaha</td>
<td>NE</td>
<td>USA</td>
</tr>
<tr>
<td>Peanuts</td>
<td>Food</td>
<td>2</td>
<td>45</td>
<td>3/31/16</td>
<td>Thu.</td>
<td>Seoul</td>
<td>Korea</td>
<td></td>
</tr>
</tbody>
</table>

- **Big** table: many *columns* and *rows*
- Substantial redundancy $\rightarrow$ expensive to store and access
- Make mistakes while updating
- Could we organize the data more efficiently?
# Multidimensional Data Model

## Sales Fact Table

<table>
<thead>
<tr>
<th>pid</th>
<th>timeid</th>
<th>locid</th>
<th>sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>25</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>1</td>
<td>30</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>1</td>
<td>50</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>13</td>
<td>2</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>13</td>
<td>3</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2</td>
<td>35</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>2</td>
<td>22</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>2</td>
<td>26</td>
</tr>
</tbody>
</table>

## Locations

<table>
<thead>
<tr>
<th>locid</th>
<th>city</th>
<th>state</th>
<th>country</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Omaha</td>
<td>Nebraska</td>
<td>USA</td>
</tr>
<tr>
<td>2</td>
<td>Seoul</td>
<td></td>
<td>Korea</td>
</tr>
<tr>
<td>5</td>
<td>Richmond</td>
<td>Virginia</td>
<td>USA</td>
</tr>
</tbody>
</table>

## Products

<table>
<thead>
<tr>
<th>pid</th>
<th>pname</th>
<th>category</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>Corn</td>
<td>Food</td>
<td>25</td>
</tr>
<tr>
<td>12</td>
<td>Galaxy</td>
<td>Phones</td>
<td>18</td>
</tr>
<tr>
<td>13</td>
<td>Peanuts</td>
<td>Food</td>
<td>2</td>
</tr>
</tbody>
</table>

## Time

<table>
<thead>
<tr>
<th>timeid</th>
<th>Date</th>
<th>Day</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3/30/16</td>
<td>Wed.</td>
</tr>
<tr>
<td>2</td>
<td>3/31/16</td>
<td>Thu.</td>
</tr>
<tr>
<td>3</td>
<td>4/1/16</td>
<td>Fri.</td>
</tr>
</tbody>
</table>

## Dimension Tables

- **Multidimensional “Cube” of data**
The Star Schema

Products
- pid
- pname
- category
- price

Sales **Fact Table**
- pid
- timeid
- locid
- sales

Time
- timeid
- Date
- Day

Locations
- locid
- city
- state
- country

This looks like a star …
The Star Schema

This looks like a star ...
The Star Schema

This looks like a star ...
The Snowflake Schema

This looks like a snowflake ...

See CS 186 for more!
Online Analytics Processing (OLAP)

Users interact with multidimensional data:

- Constructing ad-hoc and often complex SQL queries
- Using graphical tools that to construct queries
  - e.g. Tableau

Let’s discuss some common types of queries used in OLAP.
Cross Tabulation (Pivot Tables)

- Aggregate data across pairs of dimensions
  - **Pivot Tables**: graphical interface to select dimensions and aggregation function (e.g., SUM, MAX, MEAN)
  - **GROUP BY** queries
- Related to contingency tables and marginalization in stats.
- What about many dimensions?
Cube Operator

- Generalizes cross-tabulation to higher dimensions.

- In SQL:

```sql
SELECT Item, Color, SUM(Quantity) AS QtySum
FROM Furniture
GROUP BY CUBE (Item, Color);
```

<table>
<thead>
<tr>
<th>Item</th>
<th>Color</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Desk</td>
<td>Blue</td>
<td>2</td>
</tr>
<tr>
<td>Desk</td>
<td>Red</td>
<td>3</td>
</tr>
<tr>
<td>Sofa</td>
<td>Blue</td>
<td>4</td>
</tr>
<tr>
<td>Sofa</td>
<td>Red</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Item</th>
<th>Color</th>
<th>QtySum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Desk</td>
<td>Blue</td>
<td>2</td>
</tr>
<tr>
<td>Desk</td>
<td>Red</td>
<td>3</td>
</tr>
<tr>
<td>Desk</td>
<td>*</td>
<td>5</td>
</tr>
<tr>
<td>Sofa</td>
<td>Blue</td>
<td>4</td>
</tr>
<tr>
<td>Sofa</td>
<td>Red</td>
<td>5</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>14</td>
</tr>
<tr>
<td>*</td>
<td>Blue</td>
<td>6</td>
</tr>
<tr>
<td>*</td>
<td>Red</td>
<td>8</td>
</tr>
</tbody>
</table>
Slicing: selecting a value for a dimension

Dicing: selecting a range of values in multiple dimension
- **Rollup:** Aggregating along a dimension
- **Drill-Down:** de-aggregating along a dimension

OLAP Queries

Similar to CUBE
Reporting and Business Intelligence (BI)

- Use high-level tools to interact with their data:
  - Automatically generate SQL queries
  - Queries can get big!

- Common!
Data Warehouse

Collects and organizes historical data from multiple sources

So far...
- Star Schemas
- Data cubes
- OLAP Queries
Data Warehouse

Collects and organizes historical data from multiple sources

- How do we deal with semi-structured and unstructured data?
- Do we really want to force a schema on load?
Data Warehouse

Collects and organizes historical data from multiple sources

- How do we deal with semi-structured and unstructured data?
- Do we really want to force a schema on load?

<table>
<thead>
<tr>
<th>iid</th>
<th>date_taken</th>
<th>is_cat</th>
<th>is_grumpy</th>
<th>image_data</th>
</tr>
</thead>
<tbody>
<tr>
<td>45123</td>
<td>01-22-2016</td>
<td>1</td>
<td>1</td>
<td><img src="image1.png" alt="Cat" /></td>
</tr>
<tr>
<td>47234</td>
<td>06-17-2017</td>
<td>0</td>
<td>1</td>
<td><img src="image2.png" alt="Cat" /></td>
</tr>
<tr>
<td>57182</td>
<td>03-15-2009</td>
<td>0</td>
<td>0</td>
<td><img src="image3.png" alt="Construction" /></td>
</tr>
<tr>
<td>23847</td>
<td>05-18-2018</td>
<td>0</td>
<td>0</td>
<td><img src="image4.png" alt="Pumpkin" /></td>
</tr>
</tbody>
</table>

Unclear what a good schema for this image data might look like. Something like above will work, but it is inflexible!
Data Lake*

Store a copy of all the data
- in one place
- in its original “natural” form

Enable data consumers to choose how to transform and use data.
- Schema on Read

What could go wrong?

*Still being defined…[Buzzword Disclaimer]
The Dark Side of Data Lakes

- Cultural shift: Curate → Save Everything!
  - Noise begins to dominate signal

- Limited data governance and planning
  - **Example:** hdfs://important/joseph_big_file3.csv_with_json
  - **What** does it contain?
  - **When** and **who** created it?

- No cleaning and verification → lots of dirty data

- New tools are more complex and old tools no longer work

Enter the data scientist
A Brighter Future for Data Lakes

Enter the data scientist

- Data scientists bring new skills
  - Distributed data processing and cleaning
  - Machine learning, computer vision, and statistical sampling

- Technologies are improving
  - SQL over large files
  - Self describing file formats (e.g. Parquet) & catalog managers

- Organizations are evolving
  - Tracking data usage and file permissions
  - New job title: data engineers
How do we **store** and **compute** on large unstructured datasets

- Requirements:
  - Handle very **large files** spanning **multiple computers**
  - Use **cheap** commodity devices that **fail frequently**
  - **Distributed data processing** quickly and **easily**

- Solutions:
  - **Distributed file systems** → spread data over multiple machines
    - Assume machine **failure** is common → **redundancy**
  - **Distributed computing** → load and process files on multiple machines concurrently
    - Assume machine **failure** is common → **redundancy**
    - **Functional programming** computational pattern → **parallelism**
Distributed File Systems
Storing very large files
Fault Tolerant Distributed File Systems

How do we store and access very large files across cheap commodity devices?
Fault Tolerant Distributed File Systems

[Big File]

[Machine 1]

[Machine 2]

[Machine 3]

[Machine 4]

[Ghemawat et al., SOSP’03]
Fault Tolerant Distributed File Systems

- Split the file into smaller parts.
- How?
  - Ideally at record boundaries
  - What if records are big?

[Ghemawat et al., SOSP'03]
Fault Tolerant Distributed File Systems

[Ghemawat et al., SOSP’03]
Fault Tolerant Distributed File Systems

[Ghemawat et al., SOSP'03]
Fault Tolerant Distributed File Systems

[Ghemawat et al., SOSP'03]
Fault Tolerant Distributed File Systems

Big File

Machine 1
- B
- C
- D

Machine 2
- A
- B
- C

Machine 3
- A
- C
- D

Machine 4
- A
- B
- D

[Ghemawat et al., SOSP’03]
Fault Tolerant Distributed File Systems

- Split large files over multiple machines
  - Easily support massive files spanning machines

- Read parts of file in parallel
  - Fast reads of large files

- Often built using cheap commodity storage devices

Cheap commodity storage devices will fail!
Fault Tolerant Distributed File Systems

Failure Event

Big File

Machine 1
-B-C-D-

Machine 2
-A-B-C-

Machine 3
-A-C-D-

Machine 4
-A-B-D-

[Ghemawat et al., SOSP’03]
Fault Tolerant Distributed File Systems

**Failure Event**

Machine 1

Machine 2

Machine 3

Machine 4

Big File

[Ghemawat et al., SOSP'03]
Fault Tolerant Distributed File Systems

Failure Event

[Big File]

Machine 2
A  B
C

Machine 4
A  B  D

[Ghemawat et al., SOSP’03]
Fault Tolerant Distributed File Systems

Failure Event

[Ghemawat et al., SOSP'03]
Map-Reduce
Distributed Aggregation
Computing across very large files
Interacting With the Data

Good for smaller datasets
- Faster more natural interaction
- Lots of tools!

Good for bigger datasets and compute intensive tasks

$\sum = \bigoplus_{r \in \text{Data}} f_\theta(r)$

Query: $f_\theta$

Response: $\sum$

Compute Locally

Cluster Compute
How would you compute the number of occurrences of each word in all the books using a team of people?
Simple Solution
Simple Solution

1) Divide Books Across Individuals
Simple Solution

1) Divide Books Across Individuals

2) Compute Counts Locally

<table>
<thead>
<tr>
<th>Word</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apple</td>
<td>2</td>
</tr>
<tr>
<td>Bird</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Word</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apple</td>
<td>0</td>
</tr>
<tr>
<td>Bird</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Simple Solution

1) Divide Books Across Individuals

2) Compute Counts Locally

<table>
<thead>
<tr>
<th>Word</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apple</td>
<td>2</td>
</tr>
<tr>
<td>Bird</td>
<td>7</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

3) Aggregate Tables

<table>
<thead>
<tr>
<th>Word</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apple</td>
<td>2</td>
</tr>
<tr>
<td>Bird</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Word</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apple</td>
<td>0</td>
</tr>
<tr>
<td>Bird</td>
<td>8</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
The Map Reduce Abstraction

**Example: Word-Count**

**Map**(book):

```
for (word in set(book)):
    emit (word, book.count(word))
```

**Reduce**(word, counts) {

```
sum = 0
for count in counts:
    sum += count
emit (word, SUM(counts))
```

[Dean & Ghemawat, OSDI'04]
The Map Reduce Abstraction (Simpler)

Example: Word-Count

Map(book):
for (word in book):
emit (word, 1)

Reduce(word, counts) {
    sum = 0
    for count in counts:
        sum += count
    emit (word, SUM(counts))
}
The Map Reduce Abstraction (General)

Example: Word-Count

**Map** (record, f):
for (key in record):
emit (key, f(key))

**Reduce** (key, values, f) {
agg = f(values[0], values[1])
for value in values[2:]:
agg = f(agg, value)
emit (word, agg)
}

Map: Deterministic
Reduce: Commutative and Associative

[Dean & Ghemawat, OSDI'04]
Key properties of Map And Reduce

- **Deterministic Map**: allows for re-execution on failure
  - If some computation is lost we can always re-compute
  - Issues with samples?

- **Commutative Reduce**: allows for re-order of operations
  - \( \text{Reduce}(A,B) = \text{Reduce}(B,A) \)
  - Example (addition): \( A + B = B + A \)

- **Associative Reduce**: allows for regrouping of operations
  - \( \text{Reduce}(\text{Reduce}(A,B), C) = \text{Reduce}(A, \text{Reduce}(B,C)) \)
  - Example (max): \( \max(\max(A,B), C) = \max(A, \max(B,C)) \)
    - Warning: Floating point operations (e.g. addition) are not guaranteed associative.
Question

- Suppose our reduction function computes \( a \cdot b + 1 \).
- Suppose we have 3 values associated with the key ‘cat’. What is the result of the reduction operation?

![Diagram showing the reduction process for the key 'cat' with values 3, 2, and 1, resulting in a question mark as the output.]
Executing Map Reduce

Map

Record

Key Value

Key Value

...
Executing Map Reduce

Distributing the Map Function
Executing Map Reduce

Distributing the Map Function
Executing Map Reduce

The map function applied to a local part of the big file.

Run in Parallel.

Output is cached for fast recovery on node failure.
Executing Map Reduce

Reduce function can be run on many machines ...
Run in Parallel

Machine 1

Map

Reduce

apple 1

Machine 2

Map

Reduce

the 2

Machine 3

Map

Reduce

cat 1

Machine 4

Map

Reduce

big 1

dog 1
Reduce
Machine 2
A
B
C
Machine 3
A
C
D
Machine 4
A
B
D
Map
Reduce
apple 1
Reduce
the 2
1
Reduce
cat 1
3
1
Reduce
big 1
1
Reduce
dog 1
1
Output File
apple 1
the 3
cat 5
big 2
dog 2
If part of the file or any intermediate computation is lost we can simply **recompute it** without recomputing everything.
Map Reduce Technologies
Hadoop

- First open-source map-reduce software
  - Managed by Apache foundation

- Based on Google’s
  - Google File System
  - MapReduce

- Companies formed around Hadoop:
  - Cloudera
  - Hortonworks
  - MapR
Hadoop

- Very active open source ecosystem
- Several key technologies
  - **HDFS**: Hadoop File System
  - **MapReduce**: map-reduce compute framework
  - **YARN**: Yet another resource negotiator
  - **Hive**: SQL queries over MapReduce
  - ...

- Downside: Tedious to use!
  - Joey: Word count example from before is 100s of lines of Java code.
In-Memory Dataflow System
Developed at the UC Berkeley AMP Lab


What Is **Spark**

- Parallel execution engine for big data processing
- **General**: efficient support for multiple workloads
- **Easy** to use: 2-5x less code than Hadoop MR
  - High level API's in Python, Java, and Scala
- **Fast**: up to 100x faster than Hadoop MR
  - Can exploit in-memory when available
  - Low overhead scheduling, optimized engine
Spark Programming Abstraction

- Write programs in terms of transformations on distributed datasets

- Resilient Distributed Datasets (RDDs)
  - Distributed collections of objects that can stored in memory or on disk
  - Built via parallel transformations (map, filter, …)
  - Automatically rebuilt on failure
RDD: Resilient Distributed Datasets

- Collections of objects partitioned & distributed across a cluster
  - Stored in RAM or on Disk
  - Resilient to failures

- Operations
  - Transformations
  - Actions
Operations on RDDs

- Transformations $f(RDD) \Rightarrow RDD$
  - Lazy (not computed immediately)
  - E.g., "map", "filter", "groupBy"

- Actions:
  - Triggers computation
  - E.g. "count", "collect", "saveAsTextFile"
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns.
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```python
lines = spark.textFile("hdfs://file.txt")
```
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns.

```python
lines = spark.textFile("hdfs://file.txt")
```
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```python
lines = spark.textFile("hdfs://file.txt")
errors = lines.filter(lambda s: s.startswith("ERROR"))
```
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile("hdfs://file.txt")
errors = lines.filter(lambda s: s.startswith("ERROR"))
```
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```python
lines = spark.textFile("hdfs://file.txt")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
```
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```python
lines = spark.textFile("hdfs://file.txt")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
```
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```python
lines = spark.textFile("hdfs://file.txt")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
```
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```python
lines = spark.textFile("hdfs://file.txt")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
```
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile("hdfs://file.txt")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
```
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns.

```python
lines = spark.textFile("hdfs://file.txt")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
```
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns.

```python
lines = spark.textFile("hdfs://file.txt")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
```
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```python
lines = spark.textFile("hdfs://file.txt")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t") [2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
```
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile("hdfs://file.txt")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
```
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```python
lines = spark.textFile("hdfs://file.txt")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t") [2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
```
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```python
lines = spark.textFile("hdfs://file.txt")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
```
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```python
lines = spark.textFile("hdfs://file.txt")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
```

Cache your data ➔ Faster Results

*Full-text search of Wikipedia*
- 60GB on 20 EC2 machines
- 0.5 sec from mem vs. 20s for on-disk
Example: Counting Words

```python
moby_dick = spark.textFile(“hdfs://books/mobydick.txt”)  
lines = moby_dick.flatMap(lambda line: line.split(“ ”))  
counts = lines.map(lambda word: (word, 1))  
    .reduceByKey(lambda word: (word, 1))

counts.toDF().toPandas()
```

- The `flatMap` and `map` calls produce transformed RDDs.
  - Computation is lazy! Nothing happens until we get to our action.

- The `reduceByKey` calls kicks off the actual computing.

Much simpler than Hadoop map reduce code!
Abstraction: *Dataflow Operators*

<table>
<thead>
<tr>
<th>map</th>
<th>reduce</th>
<th>sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>filter</td>
<td>count</td>
<td>take</td>
</tr>
<tr>
<td>groupBy</td>
<td>fold</td>
<td>first</td>
</tr>
<tr>
<td>sort</td>
<td>reduceByKey</td>
<td>partitionBy</td>
</tr>
<tr>
<td>union</td>
<td>cogroup</td>
<td>mapWith</td>
</tr>
<tr>
<td>join</td>
<td>cross</td>
<td>pipe</td>
</tr>
<tr>
<td>leftOuterJoin</td>
<td>zip</td>
<td>save</td>
</tr>
<tr>
<td>rightOuterJoin</td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
Abstraction: Dataflow Operators

map          reduce          sample
filter       count           take
groupBy      fold            first
sort         reduceByKey    partitionBy
union        groupByKey     mapWith
cross        cogroup        pipe
zip          leftOuterJoin  save
            rightOuterJoin  ...

Spark Demo
ETL is used to bring data from operational data stores into a data warehouse.
- Many ways to organize tabular data warehouse, e.g. star and snowflake schemas.

Online Analytics Processing (OLAP) techniques let us analyze data in data warehouse.
- Examples: Pivot table, CUBE, slice, dice, rollup, drill down.

Unstructured data is hard to store in a tabular format in a way that is amenable to standard techniques, e.g. finding pictures of cats.
- Resulting new paradigm: The Data Lake.
Summary (2/2)

- Data Lake is enabled by two key ideas:
  - Distributed file storage.
  - Distributed computation.

- Distributed file storage involves replication of data.
  - Better speed and reliability, but more costly.

- Distributed computation made easier by map reduce.
  - Hadoop: Open-source implementation of distributed file storage and computation.
  - Spark: Typically faster and easier to use than Hadoop.