Big Data Analytics
Map-Reduce and Spark

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

With revisions by:
Josh Hug & John DeNero
Data in the Organization

A little bit of buzzword bingo!

Data Warehouse
Operational Data Store
ETL (Extract, Transform, Load)
Schema on Read
Snowflake Schema
Star Schema
OLAP (Online Analytics Processing)
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 → Transform → 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** → often requires specialized code and tools
- Different schemas, encodings, granularities
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 → 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 1</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>
The Star Schema

- **Products**
  - pid, pname, category, price

- **Time**
  - timeid, Date, Day

- **Locations**
  - locid, city, state, country

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

← 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!
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
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?

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?

Machine 1

Machine 2

Machine 3

Machine 4

[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

Machine 2
- A
- B

Machine 3
- A

Machine 4
- A
- B

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

Big File

Machine 1
B
C

Machine 2
A
B
C

Machine 3
A
C

Machine 4
A
B

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

[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

Machine 3
- A
- C

Machine 4
- A
- B

[Ghemawat et al., SOSP’03]
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

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

Failure Event

[Ghemawat et al., SOSP'03]
Distributed Computing

Spark Programming Abstraction

- Write programs in terms of transformations on distributed datasets

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

Slide provided by M. Zaharia
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

```
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.

```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

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

```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()
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()
```
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
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.

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.
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.