Today…

- Last session
  - BigTable Video Lecture and Discussion

- Today’s session
  - Apache Pig, Hive and Zookeeper

- Announcement:
  - Project update is due today
Going beyond MapReduce…

- MapReduce provides a simple abstraction to write distributed programs running on large-scale systems on large amounts of data.

- MapReduce is not suitable for everyone:
  - MapReduce abstraction is low-level and developers need to write custom programs which are hard to maintain and reuse.

- Sometimes user requirements may differ:
  - Interactive processing of large log files.
  - Process big data using SQL syntax rather than Java programs.
  - Warehouse large amounts of data while enabling transactions and queries.
  - Write a custom distributed application but don’t want to manage distributed synchronization and co-ordination.
Unstructured vs. Structured Data

- **Structured Data**
  - Data with a corresponding data model, such as a schema
  - Fits well in relational tables
  - E.g. Data in an RDBMS

- **Unstructured Data**
  - No data model, schema
  - Textual or bit-mapped (pictures, audio, video etc.)
  - E.g. Log Files, E-mails etc.

<table>
<thead>
<tr>
<th>Email ID</th>
<th>First Name</th>
<th>Class</th>
<th>Major</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="mailto:johndoe@cmu.edu">johndoe@cmu.edu</a></td>
<td>“John”</td>
<td>2014</td>
<td>CS</td>
</tr>
<tr>
<td><a href="mailto:janedoe@cmu.edu">janedoe@cmu.edu</a></td>
<td>“Jane”</td>
<td>2013</td>
<td>IS</td>
</tr>
</tbody>
</table>

**Relational Database Table**

Apache Web Server Log

- 123.123.123.123 - - [26/Apr/2000:00:23:48 -0400] "GET /pics/wpaper.gif HTTP/1.0" 200 6248 "http://www.jafsoft.com/asctortf/" "Mozilla/4.05 (Macintosh; I; PPC)"


From: http://www.jafsoft.com/searchengines/log_sample.html
Hadoop Spin-offs

- Pig
- Hive
- Hadoop
- Zookeeper
Why Pig?

- Many ways of dealing with small amounts of data:
  - Unstructured Logs on single machine: awk, sed, grep etc.
  - Structured Data: SQL queries through an RDBMS
- How to process giga/tera/peta-bytes of unstructured data?
  - Web crawls, log files, click streams
  - Converting log files into database entries is tedious
- SQL syntax may not be ideal
  - Strict syntax, not suited for scripting–centric programmers
- MapReduce is tedious!
  - Rigid data flow – Map and Reduce
  - Custom code for common operations such as joins – and difficult!
  - Reuse is difficult
Apache Pig

- Pig latin language
  - High-level language to express operations on data
  - User specifies the operations on the data as a *query execution plan* in *Pig Latin*

- Apache Pig framework
  - Interprets and executes pig latin programs into MapReduce jobs
  - Grunt – a command line interface to pig
  - Pig Pen – debugging environment
Pig Use Cases

- Ad-hoc analysis of unstructured data
  - Web crawls, log files, click streams

- Pig is an excellent ETL tool
  - “Extract, Transform, Load” for pre-processing data before loading it into a data warehouse

- Rapid Prototyping for Analytics
  - You can experiment with large data sets before you write custom applications
Design Goals of Pig Latin

- **Dataflow language**
  - Operations are expressed as a sequence of steps, where each step performs only a single high-level data transformation
  - Unlike SQL where the query should encapsulate most of the operation required

- **Quick start and interoperability**
  - Quickly load flat files and text files, output can also be tailored to user needs
  - Schemas are optional, i.e., fields can be referred to by position ($1, $4 etc.)

- **Fully nested data model**
  - A field can be of any data type, a data type can encapsulate any other data type

- **UDFs as first-class citizens**
  - User defined functions can take in any data type and return any data type
  - Unlike SQL which restricts function parameters and return types
Pig Latin – Data Types

- **Data types**
  - *Atom*: Simple atomic value
  - *Tuple*: A tuple is a sequence of fields, each can be any of the data types
  - *Bag*: A bag is a collection of tuples
  - *Map*: A collection of data items that is associated with a dedicated atom

```
'alice'  ('alice', 'lakers')  { ('alice', 'lakers')  ('alice', ('iPod', 'apple')) }  { 'fan of' → { 'lakers' }, 'age' → 20 }

Atom  Tuple  Bag  Map
```
## Pig Latin – Expressions

\[ t = (\text{‘alice’}, \{ (\text{‘lakers’}, 1) \}, \{ (\text{‘iPod’}, 2) \}, \{ \text{‘age’} \to 20 \}) \]

<table>
<thead>
<tr>
<th>Expression Type</th>
<th>Example</th>
<th>Value for tuple t</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>‘bob’</td>
<td>Independent of t</td>
</tr>
<tr>
<td>Field by position</td>
<td>$0</td>
<td>‘alice’</td>
</tr>
<tr>
<td>Field by name</td>
<td>f3</td>
<td>[‘age’ → 20]</td>
</tr>
<tr>
<td>Projection</td>
<td>f2,$0</td>
<td>{ (‘lakers’) }</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{ (‘iPod’) }</td>
</tr>
<tr>
<td>Map Lookup</td>
<td>f3#’age’</td>
<td>20</td>
</tr>
<tr>
<td>Function Evaluation</td>
<td>SUM(f2.$1)</td>
<td>1 + 2 = 3</td>
</tr>
<tr>
<td>Conditional Expression</td>
<td>F3#’age’&gt;18?</td>
<td>‘adult’</td>
</tr>
<tr>
<td></td>
<td>‘adult’</td>
<td>‘minor’</td>
</tr>
<tr>
<td>Flattening</td>
<td>FLATTEN(f2)</td>
<td>‘lakers’, 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>‘ipod’, 2</td>
</tr>
</tbody>
</table>
**Pig Latin – Commands / Operators (1)**

- **LOAD** — Specify input data
  
  ```pig
  queries = LOAD 'query_log.txt' USING myLoad()
  AS (userId, querystring, timestamp);
  ```

  - `myLoad()` is a user defined function (UDF)

- **FOREACH** — Per-tuple processing

  ```pig
  expanded_queries = FOREACH queries GENERATE userId, expandQuery(queryString);
  ```
Pig Latin – Commands / Operators (2)

- **FLATTEN** – Remove nested data in tuples

\[
\text{FLATTEN(} \text{expandedQueries}\text{)};
\]

\[
\begin{aligned}
\text{alice, } &\{(\text{lakers rumors})\} \\
\text{bob, } &\{(\text{iPod nano}) \\
\end{aligned}
\rightarrow
\begin{aligned}
\text{alice, lakers rumors} \\
\text{alice, lakers news} \\
\text{bob, iPod nano} \\
\text{bob, iPod shuffle}
\end{aligned}
\]

- **FILTER** – Discarding unwanted data

\[
\text{FILTER expandedQueries BY userId == 'alice'}
\]

\[
\begin{aligned}
\text{alice, lakers rumors} \\
\text{alice, lakers news} \\
\text{bob, iPod nano} \\
\text{bob, iPod shuffle}
\end{aligned}
\rightarrow
\begin{aligned}
\text{alice, lakers rumors} \\
\text{alice, lakers news}
\end{aligned}
\]
Pig Latin – Commands / Operators (3)

- **COGROUP** — Getting related data together
  - grouped_data = COGROUP results BY queryString,
    revenue BY queryString;

GROUP is a special case of COGROUP
**JOIN** – Cross product of two tables

- \( \text{join\_result} = \text{JOIN results BY queryString, revenue BY queryString;} \)

\[
\begin{align*}
\text{results:} & \quad (\text{queryString, url, rank}) \\
& \quad (\text{lakers, nba.com, 1}) \\
& \quad (\text{lakers, espn.com, 2}) \\
& \quad (\text{kings, nhl.com, 1}) \\
& \quad (\text{kings, nba.com, 2})
\end{align*}
\]

\[
\begin{align*}
\text{revenue:} & \quad (\text{queryString, adSlot, amount}) \\
& \quad (\text{lakers, top, 50}) \\
& \quad (\text{lakers, side, 20}) \\
& \quad (\text{kings, top, 30}) \\
& \quad (\text{kings, side, 10})
\end{align*}
\]

\[
\begin{align*}
\text{join\_results:} & \quad (\text{queryString, url, rank, adSlot, revenue}) \\
& \quad (\text{lakers, nba.com, 1, top, 50}) \\
& \quad (\text{lakers, nba.com, 1, side, 20}) \\
& \quad (\text{lakers, espn.com, 2, top, 50}) \\
& \quad (\text{lakers, espn.com, 2, side, 20}) \\
& \quad (\text{kings, nhl.com, 1, top, 30}) \\
& \quad (\text{kings, nhl.com, 1, side, 10}) \\
& \quad (\text{kings, nba.com, 2, top, 30}) \\
& \quad (\text{kings, nba.com, 2, side, 10})
\end{align*}
\]

**JOIN** is the same as **COGROUP + FLATTEN**
Pig Latin – Commands / Operators (5)

- **STORE** — Create output
  
  ```
  final_result = STORE join_results INTO 'myoutput',
  USING myStore();
  ```
Architecture of Pig

- Grunt (CLI)
- PigPen
- Pig Driver
- Hadoop Cluster

Logical Plan
- Query Parser
- Semantic Checking
- Logical Optimizer

Physical Plan

Logical to Physical Translator

MapReduce Plan

Physical to MapReduce Plan Translator

Execution on Hadoop

Carnegie Mellon Qatar
Interpretation of a Pig Program

- The Pig interpreter parses each command and builds a **logical plan** for each bag created by the user.
- The logical plan is converted to a **physical plan**
- Pig then creates an **execution plan** of the physical plan with maps and reduces
- Execution starts only after output is requested—**lazy compilation**
Hadoop Spin-offs

- Pig
- Hive
- Hadoop
- Zookeeper
Motivation for Hive

- Organizations that have been using SQL-based RDBMS for storage
  - Oracle, MSSQL, MySQL etc.

- The RDBMS has grown beyond what one server can handle
  - Storage can be expanded to a limit
  - Processing of Queries is limited by the computational power of a single server

- Traditional business analysts with SQL experience
  - May not be proficient at writing Java programs for MapReduce
  - Require SQL interface to run queries on TBs of data
Apache Hive

- Hive is a data warehouse infrastructure built on top of Hadoop that can compile SQL-style queries into MapReduce jobs and run these jobs on a Hadoop cluster
  - MapReduce for execution
  - HDFS for storage

- Key principles of Hive’s design:
  - SQL Syntax familiar to data analysts
  - Data that does not fit traditional RDBMS systems
  - To process terabytes and petabytes of data
  - Scalability and Performance
Hive Use Cases

- Large-scale data processing with SQL-style syntax:
  - Predictive Modeling & Hypothesis Testing
  - Customer Facing Business Intelligence
  - Document Indexing
  - Text Mining & Data Analysis
Hive Components

- **HiveQL**
  - Subset of SQL with extensions for loading and storing

- **Hive Services**
  - The Hive Driver – compiler, executor engine
  - Web Interface to Hive
  - Hive Hadoop Interface to the JobTracker and NameNode

- **Hive Client Connectors**
  - For existing Thrift, JDBC and ODBC applications
Hive Data Model

- **Tables**
  - Similar to Tables in RDBMS
  - Each Table is a unique directory in HDFS

- **Partitions**
  - Partitions determine the distribution of data within a table.
  - Each partition is a sub-directory of the main directory in HDFS

- **Buckets**
  - Partitions can be further divided into buckets.
  - Each bucket is stored as a file in the directory
HiveQL Commands

- **Data Definition Language**
  - Used to describe, view and alter tables.
  - For E.g. `CREATE TABLE` and `DROP TABLE` commands with extensions to define file formats, partitioning and bucketing information.

- **Data Manipulation Language**
  - Used to load data from external tables and insert rows using the `LOAD` and `INSERT` commands.

- **Query Statements**
  - `SELECT` 
  - `JOIN` 
  - `UNION` 
  - etc.
User-Defined Functions in Hive

- Four Types
  - User Defined Functions (UDF)
    - Perform tasks such as Substr, Trim etc. on data elements
  - User Defined Aggregation Functions (UDAF)
    - Performed on Columns
    - Sum, Average, Max, Min… etc.
  - User Defined Table-Generating Functions (UDTF)
    - Outputs a new table
    - Explode is an example – similar to FLATTEN() in Pig.
  - Custom MapReduce scripts
    - The MR scripts must read rows from standard output
    - Write rows to standard input.
Architecture of Hive

- Data Analyst / SQL Programmer
- Driver (Compiler, Optimizer, Executor)
- Metastore
- HDFS Client
- JobClient
- Hive Thrift Client
- Hive JDBC Client
- Hive ODBC Client
- CLI
- Hive Server
- Hive Web Interface
- Traditional DB
- Hadoop Cluster
- Compute and Storage Back-ends
Compilation of Hive Programs

Parser
Parses the query string into a parse tree representation

Semantic Analyzer
Retrieves the schema and verifies the validity of the query. Transforms the query into an internal representation

Logical Plan Generator
Converts the internal query representation into a logical execution plan

Optimizer
Multiple passes over the logical plan and rewrites it. Combines Multiple joins, reduces the number of MR jobs, etc.

Physical Plan Generator
Logical plan is converted into a physical plan, which is a DAG of Map-Reduce jobs.

Execution in Hadoop
Hadoop Spin-offs

Pig  Hive

Hadoop  Zookeeper
Why ZooKeeper?

- Writing distributed applications is hard
  - Need to deal with synchronization, concurrency, naming, consensus, configuration etc.
  - Well known algorithms exist for each of these problems
  - But programmers have to re-implement them for each distributed application they write.

- Master-slave architecture is popular for distributed applications
  - But how do you deal with master failures?
  - Single master can quickly become the performance bottleneck for many distributed applications.
What is Apache ZooKeeper?

- ZooKeeper is a distributed co-ordination service for large-scale distributed systems.

- ZooKeeper allows application developers to build the following systems for their distributed application:
  - Naming
  - Configuration
  - Synchronization
  - Organization
  - Heartbeat systems
  - Democracy / Leader election
ZooKeeper Architecture
Client Interactions with Zookeeper

- Clients must have the list of all the zookeeper servers in the ensemble
  - Clients will attempt to connect to the next server in the ensemble if one fails

- Once a client connects to a server, it creates a new *session*
  - The application can set the session timeout value
  - Session is kept alive through the heartbeat mechanism.
  - Failure events are automatically handled and watch events are delivered to the client on reconnection.
Zookeeper Data Model

- Similar to a filesystem
  - Hierarchical layout to denote a membership list.
- Each node is known as a *znode*
  - Znodes can be *ephemeral* or *persistent*
  - An ephemeral znode exists as long as the session of the client who created it.
  - Ephemeral znodes cannot have children.
  - *Sequential* znodes are persistent and have a sequence number attached.
  - For e.g. if a second goat znode is declared under /zoo, it will be /zoo/goat2 etc.

- Znodes can store data and have an associated ACL
  - Size limit of 1 MB per znode
  - Sanity check as its more than enough to store configuration/state information
## ZooKeeper API

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>create</td>
<td>Creates a znode</td>
</tr>
<tr>
<td>delete</td>
<td>Deletes a znode (znode should not have any children)</td>
</tr>
<tr>
<td>exists</td>
<td>Tests if a znode exists and retrieves its metadata</td>
</tr>
<tr>
<td>getACL, setACL</td>
<td>Gets/sets ACL for a znode</td>
</tr>
<tr>
<td>getChildren</td>
<td>Gets a list of children for a znode</td>
</tr>
<tr>
<td>getData, setData</td>
<td>Gets and sets data for a znode</td>
</tr>
<tr>
<td>sync</td>
<td>Synchronizes a client’s view of a znode with ZooKeeper</td>
</tr>
</tbody>
</table>
Reads, Writes and Watches

- Reads can be collected from any server.
- Write requests are always forwarded to the leader which commits the write to a majority of servers \textit{atomically}.

A \textit{watch} can be optionally set on a znode after a read operation to monitor if it has been deleted or changed.
- A watch is triggered when there is an update to a specific znode and it can be used to notify clients that have read the znode.
Zookeeper Protocol : Zab

- **Zab** ensures zookeeper can keep its promises to clients. It is a two-phase protocol
- **Phase 1: Leader Election**
  - All the members of the ensemble elect a distinguished member, called the leader and other members are designated as followers.
  - The election is declared complete when a majority (quorum) of followers have synchronized the state with the leader.
- **Phase 2: Atomic Broadcast**
  - Write requests are always forwarded to the leader.
  - The update is broadcast to all the followers.
  - The leader then commits the update when a majority of followers have persisted the change.
  - The writes thus happen atomically in accordance with a two-phase commit (2PC) protocol.
Zookeeper guarantees…

- That every modification to the znode tree is *replicated to a majority* of the ensemble.
- That *fault tolerance* is achieved:
  - As long as a majority of the nodes in the ensemble are active.
  - Ensembles are typically configured to be an odd number.
- That every update is *sequentially consistent*.
- That all updates to the znode state are *atomic*.
- That every client sees only a *single system image*.
- That updates are *durable* and persist, in spite of server failures.
- That client’s view is *timely* and is not out-of-date.
Creating Higher-level Constructs with Zookeeper

- **Barrier**
  - Creating a barrier for distributed clients is easy.
  - Designate a barrier node, and clients check if it exists.

  ```
  Client ➔ /b
  exists() ➔ true
  Wait for barrier znode deletion watch event
  Client ➔ /b
  exists() ➔ false
  Proceed
  ```

- **Queue**
  - `create()` sequential znodes under a parent to designate queue items.
  - Queue can be processed using a `getchildren()` call on the `/q` item. A watch can notify client of new items on the queue.

  ```
  Client ➔ /q
  create(/q/i-)
  /q ➔ /q/i-1
  /q ➔ /q/i-2
  . . . .
  /q ➔ /q/i-n
  ```
Next Class

Virtualization