CS435 BIG DATA

PART 2.
DATA STORAGE AND FLOW MANAGEMENT

Instructor: Sangmi Lee Pallickara
Computer Science, Colorado State University
http://www.cs.colostate.edu/~cs435

Today’s topics

• FAQs
• Dynamo
• Pig Latin

FAQs

• Help session for PA3
  • Link is available

• AWS account?
  • Not late! Please create one!

Key Value Stores: Dynamo

(1) Partitioning
(2) High Availability for writes
(3) Handling temporary failures
(4) Recovering from permanent failures
(5) Membership and failure detection

Anti-entropy protocol

• Replica synchronization protocol

• Hinted replica can be lost before they can be returned to the original replica node

• Detect inconsistencies between the replicas faster

• Minimize the amount of transferred data

• Dynamo uses Merkle tree

Merkle tree

• Hash tree where leaves are hashes of the values of individual keys
  • Parent nodes are hashes of their respective children

• Each branch of the tree can be checked independently
  • Without requiring nodes to download the entire tree or dataset

• If the hash values of the root of two trees are equal
  • The values of the leaf nodes in the tree are equal
  • No synchronization needed
Uses of Merkle tree

- Merkle trees can be used to verify any kind of data stored, handled and transferred.
- Used in a peer-to-peer network
- Trusted computing systems
  - Sun’s ZFS (Zeta File System)
- Google’s Wave protocol
- Git
- Cassandra and Dynamo
- BitTorrent protocol

How Merkle tree works

- H5 = Hash value of (H1+H2)
- Top Hash A
- H1 = Hash value of Data 1
- H2 = Hash value of Data 2
- H3 = Hash value of Data 3
- H4 = Hash value of Data 4

How Dynamo uses Merkle tree

- Each node maintains a separate Merkle tree for each key range
- Two nodes exchange the root of the Merkle tree
  - Corresponding to the key ranges that they host in common
- Node performs tree traversal and determines if there is any difference
  - Perform the appropriate synchronization action
- Disadvantage
  - When a new node joins or leaves
    - Tree needs to be recalculated

Ring Membership

- A node outage should not result in re-balancing of the partition assignment or repair of the unreachable replicas
- A node outage is mostly temporary

- Gossip-based protocol
  - Propagates membership changes
  - Maintains an eventually consistent view of membership

- Each node contacts a peer every second
  - Random selection
  - Two nodes reconcile their persisted membership change history
Logical partitioning

- Almost concurrent addition of two new nodes
  - Node A joins the ring
  - Node B joins the ring
- A and B consider themselves members of the ring
  - Yet neither would be immediately aware of each other
    - Logical partitioning

External Discovery

- Seeds
  - Discovered via an external mechanism
  - Known to all nodes
  - Statically configured (or from a configuration service)
- Seed nodes will eventually reconcile their membership with all of the nodes

Communication failure

- Attempts to
  - Communicate with unreachable peers during a get or put operation
  - Transfer partitions and hinted replicas
- Detecting communication failures
  - When there is no response to an initiated communication
- Responding to communication failures
  - Sender will try alternate nodes that map to failed node’s partitions
  - Periodically retry failed node for recovery

Dataflow management over HDFS

: Apache Pig

This material is built based on,


What is Pig?

- An engine for executing data flows in parallel on Hadoop.
  - Includes a language, Pig Latin
    - Express these data flows
- Pig Latin
  - Operators (Join, sort, filter, etc)
  - Customized functions for reading, processing, and writing data
- Apache Open Source project http://pig.apache.org
Pig on Hadoop

- Pig runs on Hadoop
  - Hadoop Distributed File System (HDFS)
  - Hadoop’s processing system, MapReduce

HDFS

- A distributed filesystem
  - Stores files across all of the nodes in a Hadoop cluster
  - Breaks files into large blocks
  - Distributes them across different machines

- Pig
  - Reads input files from HDFS
  - Uses HDFS to store intermediate data between MapReduce jobs
  - Writes its output to HDFS

A Parallel Dataflow Language

- Allows users to describe how data from one or more inputs should be read, processed, and then stored to one or more outputs in parallel
  - Simple linear flows
    - E.g. word count
  - Complex workflows
    - E.g. multiple inputs are joined
  - Uses a Directed Acyclic Graph (DAG)
    - Edges: dataflows
    - Nodes: operators that process the data

Apache Pig

What’s new with Pig?

Query vs. Dataflow language

- SQL: query language
  - Focus on allowing users to form queries
  - Allows user to describe what question they want answered
  - Not how they want it answered!

- Pig
  - What and how
  - Users can define HOW they get the answers.

Dealing with multiple operations-1

- SQL
  - Write separate queries
  - Storing the intermediate data into temporary tables
  - OR, write query with sub-queries

```sql
CREATE TEMP TABLE t1 AS
SELECT customer, sum(purchase) AS total_purchases
FROM transactions
GROUP BY customer;

SELECT customer, total_purchases, zipcode FROM t1,
customer_profile_Colorado
WHERE t1.customer =
customer_profile_Colorado.customer;
```
Dealing with multiple operations-2

A long series of data operations is the main part of the design issue

```sql
--Load the transactions file, group it by customer, and sum their total purchases
txns = load 'transactions' as (customer, purchase);
grouped = group txns by customer;
total = foreach grouped generate group, SUM(txns.purchase) as tp;

--Load the customer_profile file
profile = load 'customer_profile_Colorado' as (customer);

--Join the grouped and summed transactions and customer_profile data
answer = join total by group, profile by customer;

--Write the results to the screen
dump answer;
```

SQL
- Designed for the RDBMS environment
  - Data is normalized
  - Schemas and proper constraints are enforced
- Pig
  - Designed for the Hadoop data-processing environment
  - Schemas are sometimes unknown or inconsistent
  - Data may not be properly constrained
  - Rarely normalized
  - Does not require data to be loaded into table first

How Pig differs from Pure MapReduce

- MapReduce DOES the data-processing.
  - Why is Pig necessary?
- Pig Latin provides all of the standard data-processing operations
  - Join, filter, group by, order by, union, etc.
  - More efficient implementation of complex data operations
    - Re-balancing during the reduce phase

Continued

- Pig can analyze a Pig Latin script and understand the dataflow
  - Early error checking
  - Optimization
- Pig provides type.
  - Error checking before and during runtime

Easy to use

- Find the five pages most visited by users between the ages of 18 and 25

```sql
users = load 'users' as (name, age);
fltrd = filter users by age>=18 and age <=25;
pages = load 'pages' as (user, url);
jnd = join fltrd by name, pages by user;
grpd = group jnd by url;
smmd = foreach grpd generate group, COUNT(jnd) as clicks;
srtd = order smmd by clicks desc;
top5 = limit srdt 5;
Store top5 into 'top5sites';
```

What is Pig Useful for?

- Traditional ETL (extract transform load) data pipeline
- Research on raw data
- Iterative processing
What is Pig NOT useful for?

- If you need to process gigabytes or terabytes of data, Pig is a good choice
  - It expects to read all the records of a file and write all of its output sequentially.

- If you require writing single or small groups of records or look up many different records in random order
  - Pig is NOT a good choice.
    - See NoSQL databases

Pig’s History

- Started out as a research project in Yahoo! Research
- First paper: SIGMOD in 2008
- In Fall 2007
  - Yahoo! Hadoop users started to adopt Pig
  - Pig was open sourced via the Apache Incubator
- In 2008
  - The first Pig release
  - Became a subproject of Apache Hadoop
- In 2009
  - Amazon added Pig as part of its Elastic MapReduce service
  - About 50% of Hadoop jobs in Yahoo! were Pig jobs
- In 2010
  - Became top-level Apache Project

Grunt

- Grunt is Pig’s interactive shell
  - `pig -e local`
- will result in the prompt
  - `grunt>`

Some HDFS commands in Grunt

- Grunt acts as a shell for HDFS

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cat filename</code></td>
<td>Print the content of a file to stdout</td>
</tr>
<tr>
<td><code>copyFromLocal localfile hdfsfile</code></td>
<td>Copy a file from your local disk to HDFS</td>
</tr>
<tr>
<td><code>copyToLocal hdfsfile localfile</code></td>
<td>Copy a file from HDFS to your local disk</td>
</tr>
<tr>
<td><code>rmdir filename</code></td>
<td>Remove files recursively. This is equivalent to <code>rm -r</code> in Unix</td>
</tr>
</tbody>
</table>
## Controlling Pig from Grunt

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kill jobid jobid</td>
<td>Kill the MapReduce job associated with jobid</td>
</tr>
<tr>
<td>exec param param_name=param_value[] param_file filename[] script</td>
<td>Execute the Pig Latin Script. Aliases defined in script are not imported into Grunt. Good for testing.</td>
</tr>
<tr>
<td>run param param_name=param_value[] param_file filename[] script</td>
<td>Execute the Pig Latin script in the current Grunt shell. All aliases referenced in script are available to Grunt. Shell history is available.</td>
</tr>
</tbody>
</table>

## Data types: scalar types

- **Pig’s Data Types**
  - Scalar types
  - Complex types

- **Scalar types**
  - **Int**
    - An integer
    - Ints are represented in interfaces by java.lang.Integer
    - 4 byte signed integer
  - **Long**
    - A long integer
    - Longs are represented in interfaces by java.lang.Long
    - 8 byte signed integer

- **Double**
  - A double-precision floating-point number
  - Doubles are represented in interfaces by java.lang.Double
  - 8 bytes to store their value

- **Chararray**
  - A string or character array
  - Java.lang.String

- **Bytearray**
  - A blob or array of bytes
  - DataByteArray that wraps a Java byte[]

- **Float**
  - A floating-point number
  - Floats are represented in interfaces by java.lang.Float
  - 4 bytes to store their value

## Data types: complex types

- **Tuple**
  - A fixed-length, ordered collection of Pig data elements
  - Divided into fields
  - (‘bob’, 55) describes a tuple with two fields
  - (‘John’, 18, 4.0F) describes a tuple with three fields

- **Bag**
  - An unordered collection of tuples
  - Not possible to reference tuples by position

- A bag can have:
  - Duplicate of tuples
  - Tuples with different number of fields
  - Tuples with different data types
Data types: complex types

- **Map**
  - Set of key-value pairs
  - Key must be a **chararray** to data element mapping, where that element can be any Pig type
  - `{name#bob, age#55}` creates map with two keys, “name”, and “age”.
  - The first value (‘age’) is a chararray and the second value (55) is an integer.
  - The key should be unique

Nulls

- Data of any type can be null
- Null in Pig means
  - The data is unknown
  - Missing data, error, etc
- Pig does not have a notion of constraints on the data
  - Null can happen always

Schemas

- Pig supports **schema**
  - If a schema for the data is available
    - Up-front error checking
    - Optimization
  - If no schema is available
    - Pig will still process the data

If a schema is available..

- Your program should tell Pig what it is when you load the data:

  ```pig
  Dividends = load 'NYSE_dividends' as (exchange:chararray, symbol:chararray, date:chararray, dividend:float);
  ```

If no schema is available

- Use a dollar sign + position

  ```pig
  daily = load 'NYSE_daily';
  calc = foreach daily generate $7/1000, $3 * 100.0, SUBSTRING($0,0,1), $6 - $3;
  ```

- Pig makes a safe guess
  - $7/1000
  - Guess: $7 (eighth field) is numeric type
  - $3*100.0
  - Guess: $3 is numeric type
  - Substring()
  - Guess: $0 is chararray
  - $6 - $3
  - Guess: $6 and $3 are numeric type