Large-Scale User-Centric Data Analysis

Aaron Kimball – CTO
About me

- M.S. from CSE ’08
  - Officially: I studied PL with Dan Grossman
  - Unofficially: Spent most of my time tinkering with Hadoop
- Left to join Cloudera as the first engineer
- Started Odiago in 2010 with Christophe Bisciglia
wibi!data is...

- A large-scale storage, serving, and analysis platform
- For user- or other entity-centric data
wibi!data use cases

• Have a large number of users
• Want to store (large) transaction data as well as derived data (e.g., recommendations)
• Need to serve recommendations interactively
• Require a combination of offline and on-the-fly computation
A typical workflow

- Person
- Globe
- Profile, recommendations
- Clicks, purchases, ad impressions...
- wibi
- Libraries
- Hadoop
- Investigative analytics
- Batch processing
Challenges

• Support real-time retrieval of profile data
• Store a long transactional data history
• Keep related data logically and physically close
• Update data in a timely fashion without wasting computation
This talk...

• Wibi Architecture
• Background: Hadoop & HBase
• Modeling Data: schemas & layouts
• Analysis: producers and gatherers
• Fresheners: on-the-fly recomputation
• Conclusions
• A distributed, fault-tolerant file system (HDFS)
• ... and computation platform (MapReduce)

• Open source (Apache Software Foundation)
• Several commercial vendors, distributions...
Hadoop design principles

• At scale, failure is common
• Shared-nothing architecture
• Data durability through replication
• Bring computation to the data
MapReduce: Map phase

- Mapper converts input (key, value) pairs into intermediate (k, v) pairs
MapReduce: Shuffle phase

- Arranges data so values with the same intermediate key are on the same node
- Performed automatically by the framework
MapReduce: Reduce phase

• Applies an aggregate function to each set of values with a common intermediate key
Difficulties of working with Hadoop

• (key, value)-pairs are a cumbersome format
• Data is stored across a filesystem
  – Hard to discover all data about a user
  – Hard to process multiple data sets together
• Serialization of complex data is user-defined
• Cannot access individual users/records easily
Files: A log-oriented data model

- HDFS-backed records are usually log-oriented

At T1 Alice clicked...
At T2 Bob viewed...
At T3 Alice...
NoSQL storage

- Cousin-project to Hadoop
- 3-d storage: Data organized in rows, columns, and timestamped “cells”
- HBase servers allow fast get/put access to individual rows or cells
- Data physically stored in underlying HDFS
- “Schema free”
HBase data model

- Data in cells, addressed by four “coordinates”
  - Row Id (primary key)
  - Column family
  - Column “qualifier”
  - Timestamp
HBase data model

- Column families are units of physical storage
- Data is stored in sparse files
- Data sorted by row id, qualifier, timestamp
- Cells hold uninterpreted byte arrays
Schema free: not what you want

• HBase may not impose a schema, but your data still has one
• Up to the application to determine how to organize & interpret data
• You still need to pick a serialization system
Schemas = trade-offs

• Different schemas enable efficient storage/retrieval/analysis of different types of data
• Physical organization of data still makes a big difference
  – Especially with respect to read/write patterns
Example: OpenTSDB

• Goal: Capture time-series metrics from a large number of machines
• High write bandwidth from many input nodes
• Reads: aggregates for graph visualizations
• Supports drill-down on pre-specified axes
OpenTSDB
WibiData workloads

• Large number of fat rows (one per user)
• Each row updated relatively few times/day
  – Though updates may involve large records
• Raw data written to one set of columns
• Processed results read from another
  – Often with an interactive latency requirement
• Needs to support complex data types
Serialization with 

• Apache Avro provides flexible serialization
• All data written along with its “writer schema”
• Reader schema may differ from the writer’s

```json
{
    "type": "record",
    "name": "LongList",
    "fields": [
        {
            "name": "value",
            "type": "long"
        },
        {
            "name": "next",
            "type": ["LongList", "null"]
        }
    ]
}
```
Serialization with AVRO

- No code generation required
- Producers and consumers of data can migrate independently
- Data format migrations do not require structural changes to underlying data
WibiData: An extended data model

- Columns or whole families have common Avro schemas for evolvable storage and retrieval

```xml
<column>
  <name>email</name>
  <description>Email address</description>
  <schema>"string"</schema>
</column>
```
WibiData: An extended data model

- Column families are a logical concept
- Data is physically arranged in *locality groups*
- Row ids are hashed for uniform write pressure
WibiData: An extended data model

- Wibi uses 3-d storage
- Data is often sorted by timestamp
Storing Avro-serialized data

• Need to store writer schema with each independently-serialized datum

• Wibi: Store MD5 of schema inside cell
Reading data: the schema table

• MD5 hash of schema is used as the row id in the “schema table”

<table>
<thead>
<tr>
<th>info:schema</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xAB 0x4F 0x56 0x8D</td>
</tr>
<tr>
<td>“string”</td>
</tr>
</tbody>
</table>

• Schema table is small; fully cached by clients
• Write races are ok
Analyzing data: Producers

- Producers create derived column values
- Produce operator works on one row at a time
  - Can be run in MapReduce, or on a one-off basis
Analyzing data: Producers

produce: (ref Row → unit) → Row list → unit

Decouples row mutations from MapReduce execution model, (key, value)-pair data model
Analyzing data: Gatherers

- **Gatherers** aggregate data across all rows
- Always run within MapReduce

```
gather: (Row → ('a, 'b)) → Row list → ('a, 'b) list
```
Example: TF-IDF

- Producers and gatherers can be chained to analyze data for online use
- For example, ranking search results from Wikipedia
- ...Assuming you have a separate text index (e.g., Lucene)
TF-IDF: Clean up inputs

- First producer: remove stop words and stem input
TF-IDF: Global term frequency

- Gatherer: Get word counts across all documents, loads this into a new IDF table
TF-IDF: Per-document term frequency

- Producer: Create a *map-type* column family containing per-document word counts
### TF-IDF: Search ranker

<table>
<thead>
<tr>
<th>word:computer</th>
<th>word:silicon</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

- Ranking program looks up docId’s returned by Lucene, getting TF for each search term
- Calculates relevance score using global IDF table entries for each search term
Interactive access: REST API

- REST API provides interactive access
- Producers can be triggered “on demand” to create fresh recommendations
Conclusions

• Hadoop, HBase, Avro form the core of a large-scale machine learning/analysis platform
• How you set up your schema matters
• Producer/gatherer programming model allows computations over tables to be expressed naturally; works with MapReduce
WibiData & academia

Want to do research on large-scale data? WibiData is free for academic use.

Drop us a line: aaron@odiago.com