Performing Data Science with HBase

Aaron Kimball – CTO
Kiyan Ahmadizadeh – MTS
MapReduce and log files

Log files → Batch analysis → Result data set
The way we build apps is changing
The way we build apps is changing
HBase plays a big role

- High performance random access
HBase plays a big role

• High performance random access
• Flexible schema & sparse storage
HBase plays a big role

- High performance random access
- Flexible schema & sparse storage
- Natural mechanism for time series data
  ... organized by user
Batch machine learning
Batch machine learning

```
<table>
<thead>
<tr>
<th>info:name</th>
<th>info:interests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aaron</td>
<td>Technology</td>
</tr>
<tr>
<td>Bob</td>
<td>Sports, Games</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>derived:recs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon Kindle</td>
</tr>
<tr>
<td>Madden 2013</td>
</tr>
</tbody>
</table>
```
WibiData: architecture

GET / PUT API

serve

Data Access Server

Real-time Decision Engine

Workflow

interactive shell

Data Management Tools

store

Schema Mgmt

HBase

analyze

Hadoop
Data science lays the groundwork

- Feature selection requires insight
- Data lies in HBase, not log files
- MapReduce is too cumbersome for exploratory analytics
Data science lays the groundwork

• Feature selection requires insight
• Data lies in HBase, not log files
• MapReduce is too cumbersome for exploratory analytics

• This talk: How do we explore data in HBase?
Why not Hive?

- Need to manually sync column schemas
Why not Hive?

• Need to manually sync column schemas
• No complex type support for HBase + Hive
  – Our use of Avro facilitates complex record types
Why not Hive?

- Need to manually sync column schemas
- No complex type support for HBase + Hive
  - Our use of Avro facilitates complex record types
- No support for time series events in columns
Why not Pig?

• Tuples handle sparse data poorly
Why not Pig?

• Tuples handle sparse data poorly
• No support for time series events in columns
Why not Pig?

• Tuples handle sparse data poorly
• No support for time series events in columns
• UDFs and main pipeline are written in different languages (Java, Pig Latin)
  – (True of Hive too)
Our analysis needs

• Read from HBase
Our analysis needs

• Read from HBase
• Express complex concepts
Our analysis needs

• Read from HBase
• Express complex concepts
• Support deep MapReduce pipelines
Our analysis needs

• Read from HBase
• Express complex concepts
• Support deep MapReduce pipelines
• Be written in concise code
Our analysis needs

- Read from HBase
- Express complex concepts
- Support deep MapReduce pipelines
- Be written in concise code
- Be accessible to data scientists more comfortable with R & python than Java
Our analysis needs

- Concise
- Powerful
- Interactive
Our analysis needs

- Concise: We use Scala
Our analysis needs

- Concise: We use Scala
- Powerful: We use Apache Crunch
Our analysis needs

- Concise: We use Scala
- Powerful: We use Apache Crunch
- Interactive: We built a shell

wibi>
WDL: The WibiData Language
wibi> 2 + 2
res0: Int = 4
WDL: The WibiData Language

```
wibi> 2 + 2
res0: Int = 4
```

```
wibi> :tables
  Table  Description
  ======  =========
  page    Wiki page info
  user    per-user stats
```
Outline

• Analyzing Wikipedia
• Introducing Scala
• An Overview of Crunch
• Extending Crunch to HBase + WibiData
• Demo!
Analyzing Wikipedia

• All revisions of all English pages
• Simulates real system that could be built on top of WibiData
• Allows us to practice real analysis at scale
Per-user information

- Rows keyed by Wikipedia user id or IP address
- Statistics for several metrics on all edits made by each user
Introducing Scala

• Scala language allows declarative statements
Introducing Scala

• Scala language allows declarative statements
• Easier to express transformations over your data in an intuitive way
Introducing Scala

• Scala language allows declarative statements
• Easier to express transformations over your data in an intuitive way
• Integrates with Java and runs on the JVM
Introducing Scala

• Scala language allows declarative statements
• Easier to express transformations over your data in an intuitive way
• Integrates with Java and runs on the JVM
• Supports interactive evaluation
Example: Iterating over lists

def printChar(ch: Char): Unit = {
  println(ch)
}

val lst = List('a', 'b', 'c')
lst.foreach(printChar)
... with an anonymous function

```scala
val lst = List('a', 'b', 'c')
lst.foreach( ch => println(ch) )
```

• Anonymous function can be specified as argument to `foreach()` method of a list.
• Lists, sets, etc. are immutable by default
Example: Transforming a list

```scala
val lst = List(1, 4, 7)
val doubled = lst.map(x => x * 2)
```

- `map()` applies a function to each element, yielding a new list. (`doubled` is the list `[2, 8, 14]`)
Example: Filtering

• Apply a boolean function to each element of a list, keep the ones that return true:

```scala
val lst = List(1, 3, 5, 6, 9)
val threes = lst.filter(x => x % 3 == 0)
// ‘threes’ is the list [3, 6, 9]
```
Example: Aggregation

```scala
val lst = List(1, 2, 12, 5)
lst.reduceLeft( (sum, x) => sum + x )
// Evaluates to 20.
```

- `reduceLeft()` aggregates elements left-to-right, in this case by keeping a running sum.
Crunch: MapReduce pipelines

def runWordCount(input, output) = {
    val wordCounts = read(From.textFile(input))
        .flatMap(line =>
            line.toLowerCase.split("\s+)")
        .filter(word => !word.isEmpty())
        .count
    wordCounts.write(To.textFile(output))
}
PCollections: Crunch data sets

• Represent a parallel record-oriented data set
PCollections: Crunch data sets

• Represent a parallel record-oriented data set
• Items in *PCollections* can be lines of text, tuples, or complex data structures
PCollections: Crunch data sets

• Represent a parallel record-oriented data set
• Items in PCollections can be lines of text, tuples, or complex data structures
• Crunch functions (like `flatMap()` and `filter()`) do work on partitions of PCollections in parallel.
PCollections... of WibiRows

• *WibiRow*: Represents a row in a Wibi table
PCollections... of WibiRows

- *WibiRow*: Represents a row in a Wibi table
- Enables access to sparse columns
PCollections... of WibiRows

- *WibiRow*: Represents a row in a Wibi table
- Enables access to sparse columns
- ... as a value: `row(someColumn): Value`
PCollections... of WibiRows

- **WibiRow**: Represents a row in a Wibi table
- Enables access to sparse columns
- ... as a value: `row(someColumn): Value`
- ... As a *timeline* of values to iterate/aggregate: `row.timeline(someColumn): Timeline[Value]`
Introducing: Kiyan Ahmadizadeh
Demo!

I HAVE NO IDEA
WHAT I'M DOING
Demo: Visualizing Distributions

• Suppose you have a metric taken on some population of users.

• Want to visualize what the distribution of the metric among the population looks like.
  – Could inform next analysis steps, feature selection for models, etc.

• *Histograms* can give insight on the shape of a distribution.
  – Choose a set of *bins* for the metric.
  – Count the number of population members whose metric falls into each bin.
Demo: Wikimedia Dataset

• We have a user table containing the average delta for all edits made by a user to pages.

• Edit Delta: The number of characters added or deleted by an edit to a page.

• Want to visualize the distribution of average deltas among users.
Demo!
Code: Accessing Data

```scala
val stats = 
    ColumnFamily[Stats]("edit_metric_stats")

val userTable = read(
    From.wibi(WibiTableAddress.of("user"), stats))
```
val stats = ColumnFamily[Stats]("edit_metric_stats")

val userTable = read(
    From.wibi(WibiTableAddress.of("user"), stats))

Will act as a handle for accessing the column family.
Code: Accessing Data

```scala
val stats =
    ColumnFamily[Stats]("edit_metric_stats")

val userTable = read(
    From.wibi(WibiTableAddress.of("user"), stats))
```

Type annotation tells WDL what kind of data to read out of the family.
val stats = 
ColumnFamily[Stats]("edit_metric_stats")

val userTable = read(
  From.wibi(WibiTableAddress.of("user"), stats))

userTable is a PCollection[WibiRow] obtained by reading the column family “edit_metric_stats” from the Wibi table “user.”
def getBin(bins: Range, value: Double): Int = {
    bins.reduceLeft ( (choice, bin) =>
        if (value < bin) choice else bin)
}

def inRange(bins: Range, value: Double): Boolean =
    range.start <= value && value <= range.end
def getBin(bins: Range, value: Double): Int = {
  bins.reduceLeft ({choice, bin} =>
    if (value < bin) choice else bin
  )
}

def inRange(bins: Range, value: Double): Boolean =
  range.start <= value && value <= range.end
Code: Filtering

```scala
val filtered = userTable.filter { row =>
  // Keep editors who have edit_metric_stats:delta defined
  !row(stats).isEmpty && row(stats).get.contains("delta")
}
```
val filtered = userTable.filter { row =>
  // Keep editors who have edit_metric_stats:delta defined
  !row(stats).isEmpty && row(stats).get.contains("delta")
}

Boolean predicate on elements in the PCollection
val filtered =  userTable.filter { row =>
  // Keep editors who have edit_metric_stats:delta defined
  !row.stats.isEmpty && row(stats).get.contains("delta")
}

filtered is a PCollection of rows that have the column edit_metric_stats:delta
val filtered = userTable.filter { row =>
    // Keep editors who have edit_metric_stats:delta defined
    !row(stats).isEmpty && row(stats).get.contains("delta")
}

Use stats variable we declared earlier to access the column family.

val stats = ColumnFamily[Stats]("edit_metric_stats")
val binCounts = filtered.map { row =>
  // Bucket mean deltas for histogram
  getBin(bins, abs(row(stats).get("delta")).getMean())
}.count()

binCounts.write(To.textFile("output_dir"))
val binCounts = filtered.map { row =>
  // Bucket mean deltas for histogram
  getBin(bins, abs(row(stats).get("delta")).getMean))
}.count()

binCounts.write(To.textFile("output_dir"))

Map each editor to the bin their mean delta falls into.
Code: Binning

```scala
val binCounts = filtered.map { row =>
  // Bucket mean deltas for histogram
  getBin(bins, abs(row(stats).get("delta")).getMean))
}.count()

binCounts.write(To.textFile("output_dir"))
```

Count how many times each bin occurs in the resulting collection.
Code: Binning

```scala
val binCounts = filtered.map { row =>
  // Bucket mean deltas for histogram
  getBin(bins, abs(row(stats).get("delta")).getMean))
}.count()

binCounts.write(To.textFile("output_dir"))
```

binCounts contains the number of editors that fall in each bin.
Code: Binning

```scala
val binCounts = filtered.map { row =>
  // Bucket mean deltas for histogram
  getBin(bins, abs(row(stats).get("delta")).getMean))
}.count()

binCounts.write(To.textFile("output_dir"))
```

Writes the result to HDFS.
Code: Visualization

```python
Histogram.plot(binCounts,
    bins,
    "Histogram of Editors by Mean Delta",
    "Mean Delta",
    "Number of Editors",
    "delta_mean_hist.html")
```
Analysis Results: 1% of Data
Analysis Results: Full Data Set

Histogram of Editors by Mean Delta

Number of Editors

Mean Delta
Conclusions

Concise  Powerful  Interactive
Conclusions

Concise  Powerful  Interactive

+  +  +

[Images of a yellow elephant, a database icon, and the Scala logo]
Conclusions

Concise + Powerful + Interactive = Scalable analysis of sparse data
Aaron Kimball – aaron@wibidata.com
Kiyan Ahmadizadeh – kiyan@wibidata.com