Recap on HBase

- Column-Oriented data store
- NoSQL DB
- Data is stored in Tables
  - Tables contain rows
  - Rows made of columns which are grouped in column families
- Data is stored in cells
  - Identified by row - column-family - column
  - Cells’ values are versioned
- Value = Table+RowKey+Family+Column+Timestamp
Recap on HBase

- Internally, a table is made of regions
  - Region - a range of rows stored together
- Region Server - serves one or more regions
  - A region is served by only 1 Region Server
- Master Server - daemon responsible for managing HBase cluster, aka Region Servers

Java API

- Create a Configuration object
  - Similarly to Configuration from HDFS object
  - Adds HBase specific properties
- Construct Htable
  - Provide Configuration object
  - Provide table name
- Perform operations
  - Such as put, get, scan, delete, etc...
- Close HTable instance
  - Flushes all the internal buffers
  - Releases all the resources

Slide based on lecture http://www.coreservlets.com/hadoop-tutorial/
Using Client API

- Create a Configuration object
  Configuration conf = HbaseConfiguration.create();
- Construct HTable
  HTable hTable = new HTable(conf, tableName);
- Perform operations
  hTable.getTableName();
- Close HTable instance
  hTable.close();

Creating HTable instance is not free
- Actually quite costly - scans catalog .META. Table
- Checks that table exists and enabled
- Create once (per thread) and re-use for as long as possible
- HTable is NOT thread safe
  - Create 1 instance per thread
- HTable supports CRUD batch operations
  - Not atomic
  - For performance and convenience
Create/Save Data to HBase

- Construct HTable instance
- Create Put instance
- Add cell values and their coordinates
  - Specify family:column as a coordinate
- Call put on HTable instance
- Close HTable

Put Example

```java
import static org.apache.hadoop.hbase.util.Bytes.*;
public class PutExample {
    public static void main(String[] args) throws IOException {
        Configuration conf = HBaseConfiguration.create();
        HTable hTable = new HTable(conf, "HBaseSamples");
        Put put1 = new Put(toBytes("row1"));
        put1.add(toBytes("test"), toBytes("coll1"), toBytes("val1"));
        put1.add(toBytes("test"), toBytes("col2"), toBytes("val2"));
        hTable.put(put1);
        hTable.close();
    }
}
```
Retrieving Data

- API supports
  - Get a single row by id
  - Get a set of rows by a set of row ids
  - Scan an entire table or a sub set of rows
  - To scan a portion of the table provide start and stop row ids
- Recall that row-ids are ordered by raw byte comparison
- In case of string based ids, the order is alphabetical

Retrieve a Single Row

- Construct HTable instance
- Create Get instance
- Optionally narrow down result
  - Specify family:column coordinate
  - Optionally add filters
- Request and get results
  - Call get on Htable
  - Result instance is returned and will contain the data
- Result instance: single row result of a Get or Scan query.
- Close HTable
Narrow Down Results

• Only retrieve the data that you need
  - If not specified then an entire row is retrieved
  - Important, as HBase allows you to scale to millions of rows
  - Can narrow down by family, column(s), time range and max versions
  - Can provide more than one narrow down criteria
  - Family and column name parameters are in raw bytes

• Narrow down by family
  get.addFamily(family)

• Narrow down by column
  get.addColumn(family, column)

• Narrow down by time range
  get.setTimeRange(minStamp, maxStamp)

• Specify number of versions returned
  get.setMaxVersions(maxVersions)
  - By default set to 1: only returns the latest version

• Can retrieve multiple families and columns
  get.addFamily(family)
  get.addFamily(family1)
  get.addColumn(family2, column1)
  get.addColumn(family2, column2)
  get.setTimeRange(minStamp, maxStamp)
public static void main(String[] args) throws IOException {
    Configuration conf = HBaseConfiguration.create();
    HTable hTable = new HTable(conf, "HBaseSamples");
    Get get = new Get(toBytes("row1"));
    Result result = hTable.get(get);
    print(result);
    get.addColumn(toBytes("test"), toBytes("col2"));
    result = hTable.get(get);
    print(result);
    hTable.close();
}

Scan Rows

- Construct HTable instance
- Create and Initialize Scan
- Retrieve ResultScanner from HTable
- Scan through rows
- Close ResultScanner
- Close HTable
Create and Initialize Scan

- **Scan class is a means to specify what you want to scan**
- **Scan is very similar to Get but allows you to scan through a range of keys**
  - Provide start and stop keys
  - Start key is inclusive while stop key is exclusive
  - If start row id is NOT provided then will scan from the beginning of the table
  - If stop row is NOT provided then will scan to the very end

---

**Construction options**

- `new Scan()` - will scan through the entire table
- `new Scan(startRow)` - begin scan at the provided row, scan to the end of the table
- `new Scan(startRow, stopRow)` - begin scan at the provided startRow, stop scan when a row id is equal to or greater than to the provided stopRow
- `new Scan(startRow, filter)` - begin scan at the provided row, scan to the end of the table, apply the provided filter

---
Create and Initialize Scan

- Once Scan is constructed you can further narrow down (very similar to Get)
  - scan.addFamily(family)
  - scan.addColumn(family, column)
  - scan.setTimeRange(minStamp, maxStamp)
  - scan.setMaxVersions(maxVersions)
  - scan.setFilter(filter)

- For example:
  Scan scan = new Scan(toBytes(startRow), toBytes(stopRow));
  scan.addColumn(toBytes("metrics"), toBytes("counter"));
  scan.addFamily(toBytes("info"));
  ResultScanner scanner = hTable.getScanner(scan);
  for ( Result result : scanner){
    // do stuff with result
  }

Using HBase in MapReduce jobs (III)

```java
public static void main(String[] args) throws Exception {
  Configuration conf = HBaseConfiguration.create();
  Job job = new Job(conf, "AirPollution");
  job.setJarByClass(AirPollution.class);

  Scan scan = new Scan();
  scan.addFamily(Bytes.toBytes("location"));
  scan.addColumn(Bytes.toBytes("data"), Bytes.toBytes("value"));

  FilterList li = newFilterList(FilterList.Operator.MUST_PASS_ALL);
  SingleColumnValueFilter filter = new SingleColumnValueFilter(
      Bytes.toBytes("location"), Bytes.toBytes("region"),
      CompareOp.EQUAL, Bytes.toBytes("12"));
  li.addFilter(filter);
  scan.setFilter(li);
}
```
Using HBase in MapReduce job

- **TableInputFormat**
  - Converts data in HTable to format consumable to MapReduce
  - Split: Rows in one HBase Region (provided Scan may narrow down the result)
  - Record: Row, returned columns are controlled by a provided scan
  - Key: ImmutableBytesWritable
  - Value: Result (HBase class)

- **TableOutputFormat**
  - Saves data into HTable
  - Reducer output key is ignored
  - Reducer output value must be HBase’s Put or Delete objects

Using HBase in MapReduce job

- Mapper class needs to extend TableMapper
- Reducer class needs to extend TableReducer
static class Mapper extends TableMapper<ImmutableBytesWritable, DoubleWritable> {
    public void map(ImmutableBytesWritable row, Result values, Context context) throws IOException {
        byte[] results = values.getValue(...);

        ImmutableBytesWritable userKey = new 
            ImmutableBytesWritable(key-name);
        context.write(userKey, new DoubleWritable (Bytes.toDouble 
            (results)));
    }
}

public static class Reducer extends TableReducer <ImmutableBytesWritable, DoubleWritable, ImmutableBytesWritable> {
    public void reduce(ImmutableBytesWritable key, Iterable<DoubleWritable> values, Context context) 
        throws IOException, InterruptedException {
        Put put = new Put(key.get());

        put.add(Bytes.toBytes("data"), Bytes.toBytes("average"), 
            Bytes.toBytes(sum / count));
        context.write(key, put);
    }
}
public static void main(String[] args) throws Exception {
    Configuration conf = HBaseConfiguration.create();
    Job job = new Job(conf, "AverageGINByCountryCalculator");
    job.setJarByClass(AverageGINByCountryCalculator.class);

    Scan scan = new Scan();
    scan.addFamily("ByCountry").getBytes());
    scan.setCaching(500); // 1 is the default in Scan, which
    // will be bad for MapReduce jobs
    scan.setCacheBlocks(false); // don't set to true for MR
    // jobs
    TableMapReduceUtil.initTableMapperJob(
            "HDI", // input table
            scan, // scan instance
            Mapper.class, // mapper class
            ImmutableBytesWritable.class, // mapper output key
            DoubleWritable.class, // mapper output value
            job);

    TableMapReduceUtil.initTableReducerJob(
            "HDIResult", // output table
            Reducer.class, // reducer class
            job);
    System.exit(job.waitForCompletion(true) ? 0 : 1);
}
Bulk Loading

- Importing large amounts of data tricky
  - Simple Java code could be used to enter data row by row
  - Very inefficient
- Can use a MapReduce job to load data
  - Text input format class
  - HFileOutputFormat class: writes out data in HBase's internal storage format
  - After data has been prepared using HFileOutputFormat, it is loaded into the cluster using the completebulkload tool.
    - command line tool provided by hadoop
    - iterates through the prepared output files and determines for each the region the file belongs to.
    - Contacts appropriate Region Server and moves Hfiles it into its storage directory