cloudera

Importing data from MySQL

Or, “DBInputFormat for fun and profit”

Aaron Kimball
Cloudera Inc.

Feb 18, 2009

Unstructured data is useful

- Take everyone’s favorite example, log parsing:

```
207.181.42.20 - - [07/Feb/2003:11:38:28 -0800] "GET
/archive/2003/02/01/space_sh.shtml HTTP/1.1" 200 11966
"http://www.google.com/search?hl=en&lr=&ie=UTF-8&oe=UTF-
8&q=Space+Shuttle+Columbia+November+2002" "Mozilla/4.0
(compatible; MSIE 6.0; Windows 98; Q312461)"
```

```
ip-address identd authuser [DD/MMM/YYYY:hh:mm:ss TZ]
"request string" status bytes "referrer" "user-agent"
```

Structured data is useful

- Utility of unstructured data improved by structured data

- E.g., IP Geolocation resolves IP addresses to city, state, country
  - ~100 MB of data
  - Available as SQL database dump

Joining data

- Problem: Merge the log records with IP geolocation data
- Too much log data to dump to SQL db, how to bring db to us?
  - Hadoop MapReduce, Hive, Pig… all work from HDFS!

DBInputFormat

- Connects to JDBC interface
- Selects records out of tables, arbitrary queries
- Provides interface to use arbitrary input queries, tables, databases
- Records written to *DBWritable*, provided as value to Mapper

- Constraints:
  - Must be able to totally order results (e.g., by primary key)
  - Must be able to count expected result set size ahead of time

DBWritable

- You define a class to hold a row from the database
  - Must be able to read from JDBC *ResultSet* into fields
  - Must be able to write to JDBC *PreparedStatement*
- Should also implement regular *Writable*

Configuration Example

```
JobConf conf = new JobConf(getConf(), Foo.class);
conf.setInputFormat(DBInputFormat.class);
DBConfiguration.configureDB(conf,
    “com.mysql.jdbc.Driver”,
    “jdbc:mysql://localhost/mydatabase”);
String [] fields = { “my_pkey”, “my_value” };
DBInputFormat.setInput(conf, MyRecord.class, “mytable”,
    null,  “my_pkey”, fields);
// set Mapper, etc., and call JobClient.runJob(conf);
```

DBWritable Example

```
class MyRecord implements Writable, DBWritable {
  long pkey;
  long val;

  public void readFields(DataInput in) throws IOException {
    this.pkey = in.readLong();
    this.val = in.readLong();
  }

  public void readFields(ResultSet resultSet)
      throws SQLException {
    this.pkey = resultSet.getLong(1);
    this.val = resultSet.getLong(2);
  }
}
```

Parallelism and scalability

- Prepares statement of the form:

  ```
  "SELECT … ORDER BY … LIMIT … OFFSET …"
  ```

  for each Mapper
- InputSplit corresponds to OFFSET into query
- (Counting query required ahead of time to determine split count)

- Scalability limited by bandwidth of the database server
  - 100 Mappers/Reducers would easily saturate the pipe from one node
- Could be used once to do a bulk import into HDFS for Hive, etc.

DBOutputFormat

- Define the table and fields to populate with results from MapReduce job
- Individual values emitted by Reducers are bundled into SQL transaction
  - All committed at end of reduce operation (during **`close()`**)
- DBWritable interface provides **`write(PreparedStatement stmt)`**

Flexibility

- Any JDBC database can work (MySQL, Postgres, HSQLdb…)
- Supports quick read-in of existing tables for ad-hoc jobs

- Database sharding currently would need to be handled at db side
  - Future work: support client-side row-level sharding

Conclusions

- Good for ad-hoc queries
- May be useful for bulk loading database into Hive
- Straightforward interface extends existing MapReduce API

- Available in Hadoop 0.19
  - (But HADOOP-2536 can be applied to 0.18.x without much difficulty)

cloudera