cloudera

sqoop

Easy, parallel database import/export

Aaron Kimball
Cloudera Inc.

June 8, 2010

Your database

- Holds a lot of really valuable data!
- Many structured tables of several hundred GB
- Provides fast access for OLTP applications
  - Update / delete records
  - Add individual records
  - Complex transactions

- But…

You can only go so far

- Can’t store very large datasets (1 TB+)
- Poor support for complex datatypes / large objects
- Schema evolution is hard
- Analytic queries better suited to a batch-oriented system

Hadoop and MapReduce

- A batch processing system for very large datasets
- Handles complex / unstructured data gracefully
- Can perform deep queries and large ETL tasks in parallel
- Automatic fault tolerance

- … but poor at interactive access

Sqoop is…

… a suite of tools that connect Hadoop

and database systems.

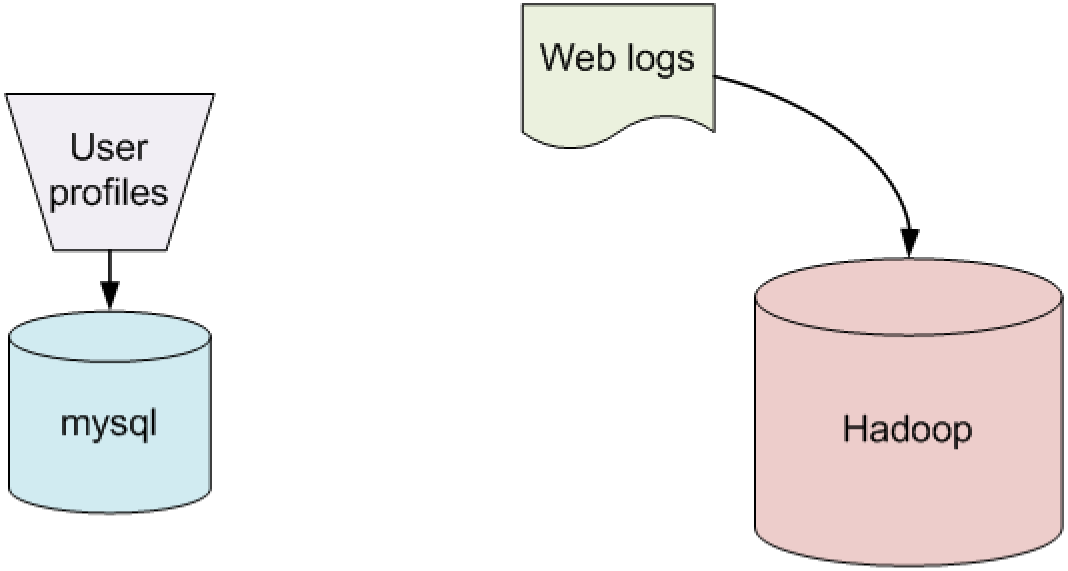
- *Import* tables from databases into HDFS for deep analysis
- *Replicate* database schemas in Hive’s metastore
- *Export* MapReduce results back to a database for presentation to end-users

In this talk…

- How Sqoop works
- Working with imported data
- Parallelism and performance
- Sqoop 1.0 Release

The problem

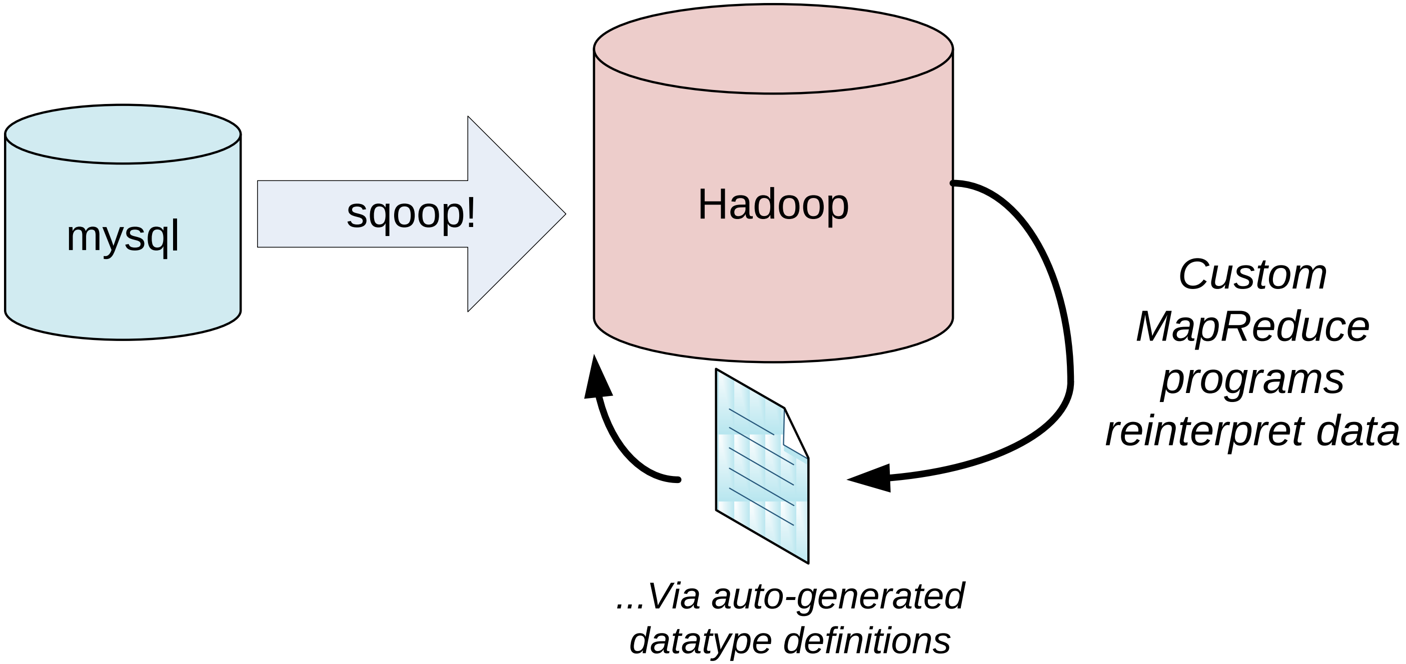
Structured data in traditional databases cannot be easily combined with complex data stored in HDFS



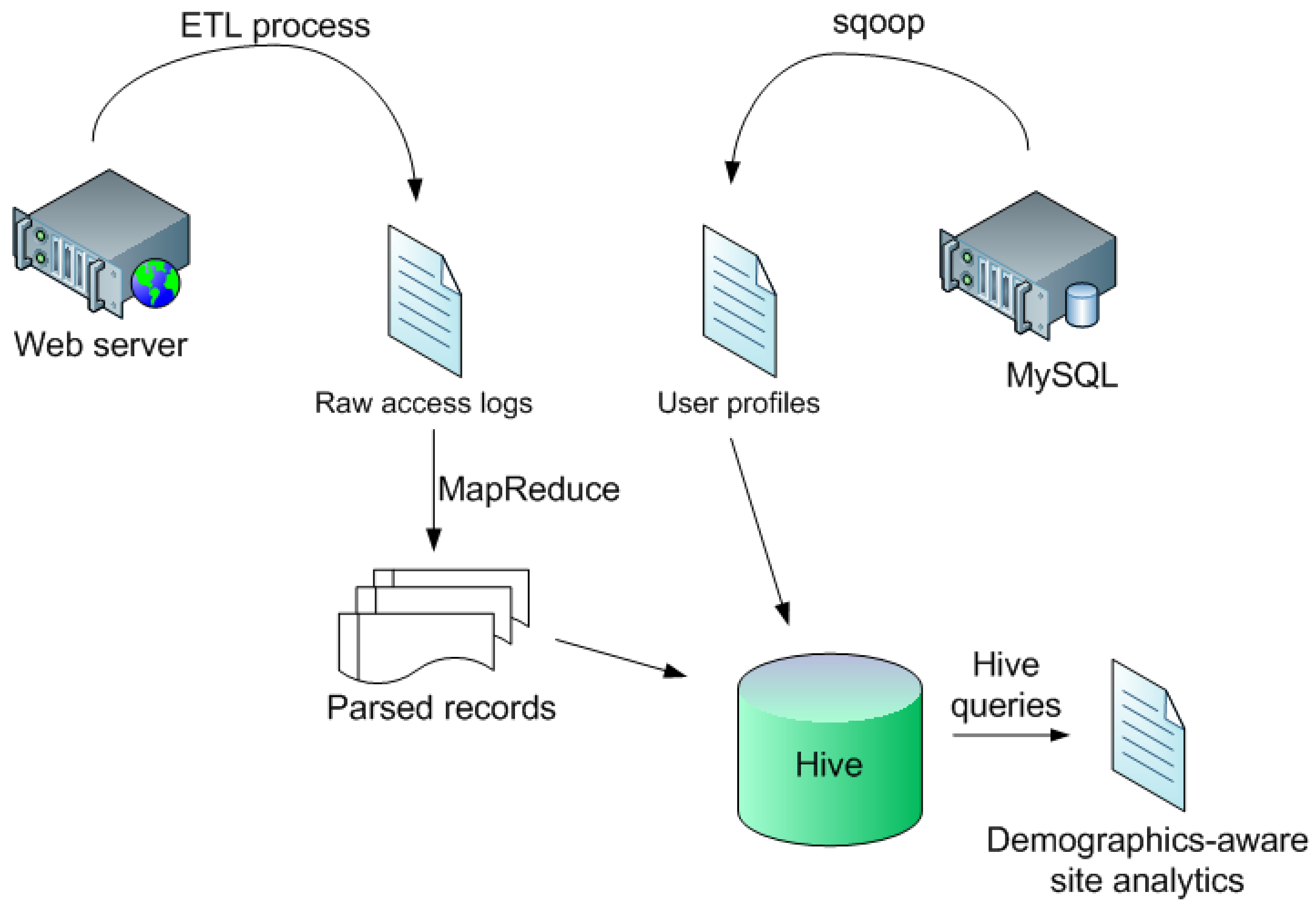
Where's the bridge?

Sqoop = SQL-to-Hadoop

- Easy import of data from many databases to HDFS
- Generates code for use in MapReduce applications
- Integrates with Hive


mysql
sqoop!
Hadoop
Custom MapReduce programs reinterpret data
...Via auto-generated datatype definitions

Example data pipeline



Key features of Sqoop

- JDBC-based implementation
  - *Works with many popular database vendors*
- Auto-generation of tedious user-side code
  - *Write MapReduce applications to work with your data, faster*
- Integration with Hive
  - *Allows you to stay in a SQL-based environment*
- Extensible backend
  - *Database-specific code paths for better performance*

Example input

```
mysql> use corp;
Database changed

mysql> describe employees;
+------------+-------------+------+-----+---------+----------------+
| Field      | Type        | Null | Key | Default | Extra          |
+------------+-------------+------+-----+---------+----------------+
| id         | int(11)     | NO   | PRI | NULL    | auto_increment |
| firstname  | varchar(32) | YES  |     | NULL    |                |
| lastname   | varchar(32) | YES  |     | NULL    |                |
| jobtitle   | varchar(64) | YES  |     | NULL    |                |
| start_date | date        | YES  |     | NULL    |                |
| dept_id    | int(11)     | YES  |     | NULL    |                |
+------------+-------------+------+-----+---------+----------------+
```

Loading into HDFS

```
$ sqoop import \
     --connect jdbc:mysql://db.foo.com/corp \
     --table employees
```

- Imports “employees” table into HDFS directory
  - Data imported as text or SequenceFiles
  - Optionally compress and split data during import
- Generates **`employees.java`** for your use

Example output

```
$ hadoop fs –cat employees/part-00000
0,Aaron,Kimball,engineer,2008-10-01,3
1,John,Doe,manager,2009-01-14,6
```

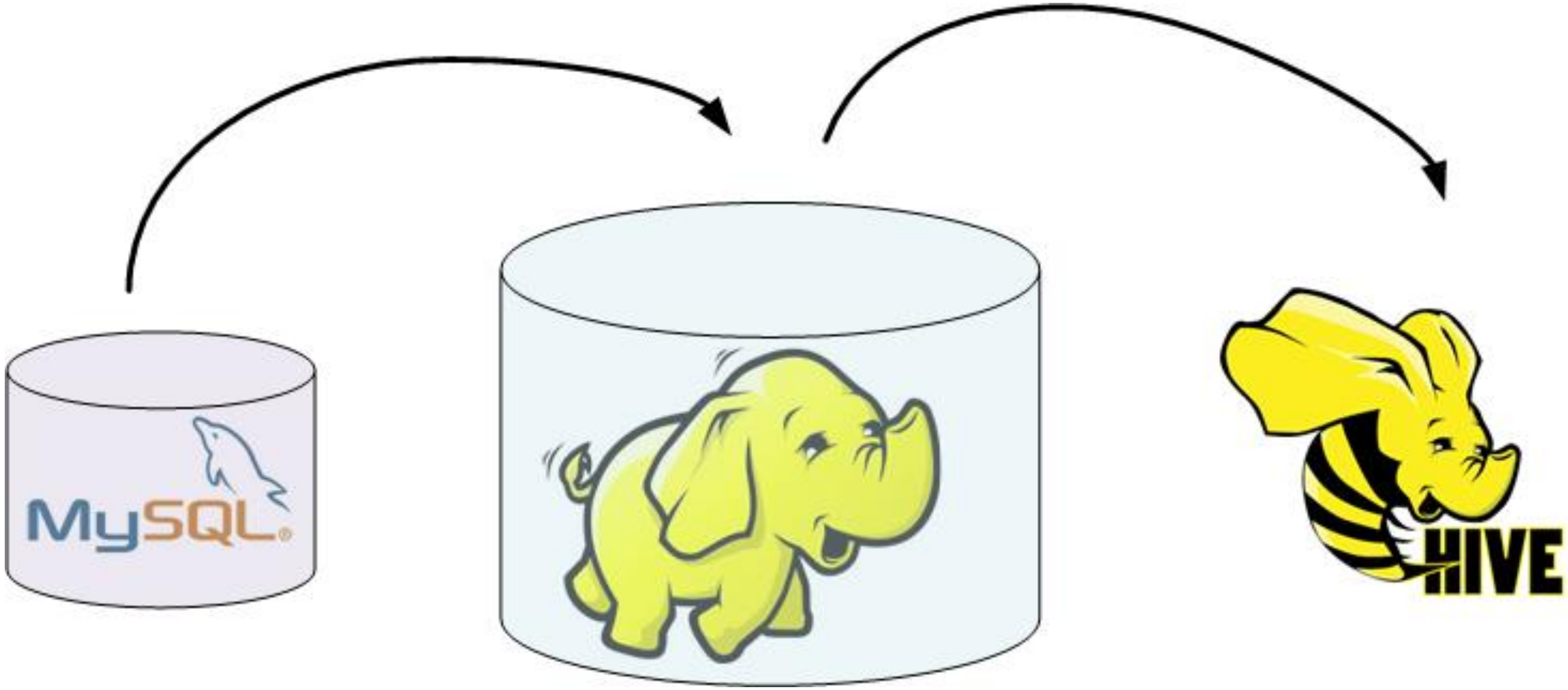
- Files can be used as input to MapReduce processing

Auto-generated class

```
public class employees {
  public Integer get_id();
  public String get_firstname();
  public String get_lastname();
  public String get_jobtitle();
  public java.sql.Date get_start_date();
  public Integer get_dept_id();
  // parse() methods that understand text
  // and serialization methods for Hadoop
}
```

Hive integration

Imports table definition into Hive after data is imported to HDFS



Export back to a database

```
mysql> CREATE TABLE ads_results (
    id INT NOT NULL PRIMARY KEY,
    page VARCHAR(256),
    score DOUBLE);

$ sqoop export \
     --connect jdbc:mysql://db.foo.com/corp \
     --table ads_results  --export-dir results
```

- Exports “results” dir into “ads_results” table

Additional options

- Multiple data representations supported
  - TextFile – ubiquitous; easy import into Hive
  - SequenceFile – for binary data; better compression support, higher performance
- Supports local and remote Hadoop clusters, databases
- Can select a subset of columns, specify a **`WHERE`** clause
- Controls for delimiters and quote characters:
  - **`--fields-terminated-by`**, **`--lines-terminated-by`**, **`--optionally-enclosed-by`**, etc.
  - Also supports delimiter conversion (**`--input-fields-terminated-by`**, etc.)

Under the hood…

- JDBC
  - Allows Java applications to submit SQL queries to databases
  - Provides metadata about databases (column names, types, etc.)
- Hadoop
  - Allows input from arbitrary sources via different *InputFormats*
  - Provides multiple JDBC-based InputFormats to read from databases
  - Can write to arbitrary sinks via *OutputFormats* – Sqoop includes a high-performance database export OutputFormat

InputFormat woes

- DBInputFormat allows database records to be used as mapper inputs
- The trouble with using DBInputFormat directly is:
  - Connecting an entire Hadoop cluster to a database is a performance nightmare
  - Databases have lower read bandwidth than HDFS; for repeated analyses, much better to make a copy in HDFS first
  - Users must write a class that describes a record from each table they want to import or work with (a “DBWritable”)

DBWritable example

```
class MyRecord implements Writable, DBWritable {
  long msg_id;
  String msg;
  public void readFields(ResultSet resultSet)
      throws SQLException {
    this.msg_id = resultSet.getLong(1);
    this.msg = resultSet.getString(2);
  }
  public void readFields(DataInput in) throws
      IOException {
    this.msg_id = in.readLong();
    this.msg = in.readUTF();
  }
}
```

DBWritable example

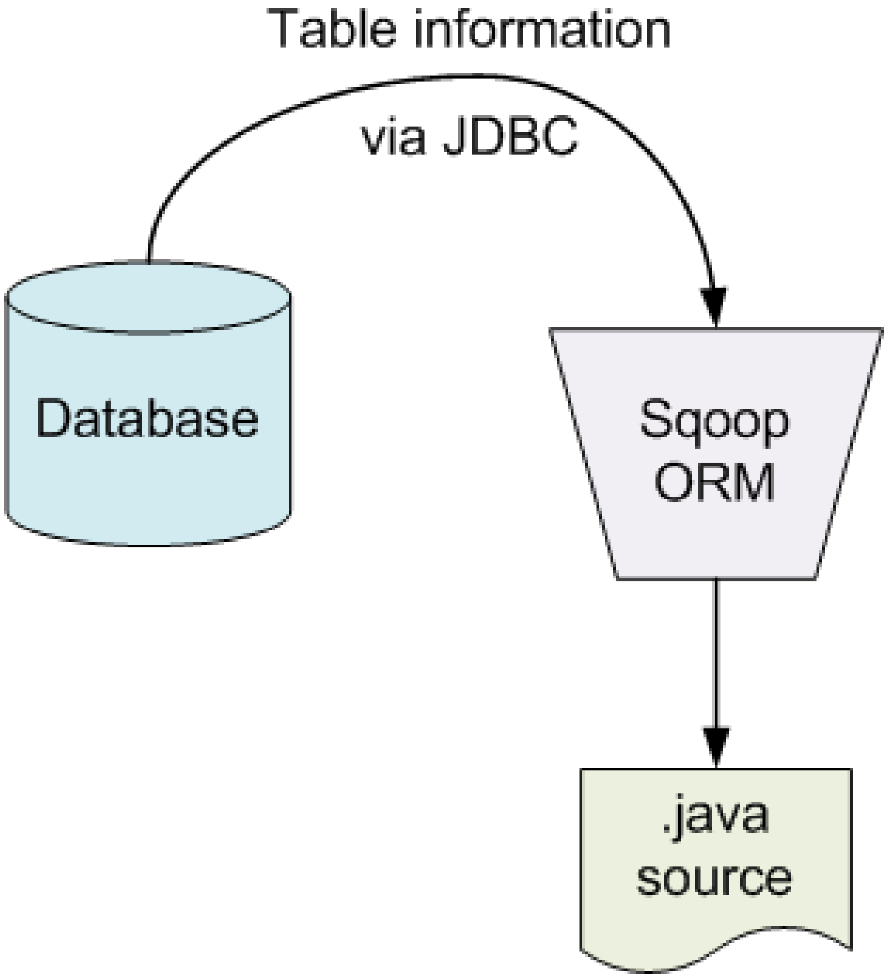
```
class MyRecord implements Writable, DBWritable {
  long msg_id;
  String msg;
  public void readFields(ResultSet resultSet)
      throws SQLException {
    this.msg_id = resultSet.getLong(1);
    this.msg = resultSet.getString(2);
  }
  public void readFields(DataInput in) throws
      IOException {
    this.msg_id = in.readLong();
    this.msg = in.readUTF();
  }
}
```

A direct type mapping

JDBC Type	Java Type
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT	double
DOUBLE	double
BINARY	byte[]
VARBINARY	byte[]
LONGVARBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

http://java.sun.com/j2se/1.3/docs/guide/jdbc/getstart/mapping.html

Class auto-generation



Working with Sqoop

- Basic workflow:
  - Import initial table with Sqoop
  - Use auto-generated table class in MapReduce analyses
  - … Or write Hive queries over imported tables
  - Perform periodic re-imports to ingest new data
  - Use Sqoop to export results back to databases for online access
- Table classes can parse records from delimited files in HDFS

Processing records in MapReduce

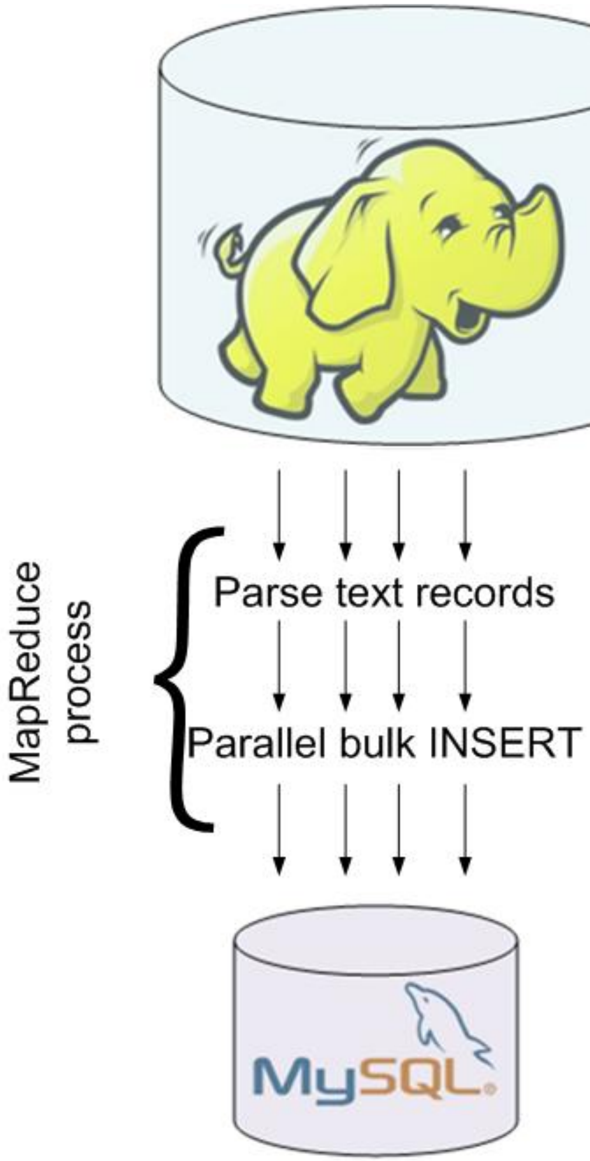
```
void map(LongWritable k, Text v, Context c) {
  MyRecord r = new MyRecord();
  r.parse(v); // auto-generated text parser
  process(r.get_msg()); // your logic here
  ...
}
```

Import parallelism

- Sqoop uses indexed columns to divide a table into ranges
  - Based on min/max values of the primary key
  - Allows databases to use index range scans
  - Several worker tasks import a subset of the table each
- MapReduce is used to manage worker tasks
  - Provides fault tolerance
  - Workers write to separate HDFS nodes; wide write bandwidth

Parallel exports

- Results from MapReduce processing stored in delimited text files
- Sqoop can parse these files, and insert the results in a database table



Direct-mode imports and exports

- MySQL provides **`mysqldump`** for high-performance table output
  - Sqoop special-cases **`jdbc:mysql://`** for faster loading
  - With MapReduce, think “distributed **`mk-parallel-dump`**”

- Similar mechanism used for PostgreSQL
- Avoids JDBC overhead

- On the other side…
  - **`mysqlimport`** provides really fast Sqoop *exports*
  - Writers stream data into **`mysqlimport`** via named FIFOs

Recent Developments

- April 2010: Sqoop moves to github
- May 2010: Preparing for 1.0 release
  - Higher-performance pipelined export process
  - Improved support for storage of large data (CLOBs, BLOBs)
  - Refactored API, improved documentation
  - Better platform compatibility:
    - Will work with to-be-released Apache Hadoop 0.21
    - Will work with Cloudera's Distribution for Hadoop 3
- June 2010: Planned 1.0.0 release (in time for Hadoop Summit)
- Plenty of bugs to fix and features to add – see me if you want to help!

Conclusions

- Most database import/export tasks are “turning the crank”
- Sqoop can automate a lot of this
  - Allows more efficient use of existing data sources in concert with new, complex data
- Available as part of Cloudera’s Distribution for Hadoop

The pitch: www.cloudera.com/sqoop

The code: github.com/cloudera/sqoop

aaron@cloudera.com

cloudera