RESEARCH PAPER:
MAPREDUCE-BASED
DIMENSIONAL ETL MADE EASY

By Xiufeng Liu, Christian Thomsen, Torben Bach Pedersen

Presented by Greg Sherman
CIS 611, Fall 2016
Prof. Sunnie S. Chung
OUTLINE OF DISCUSSION

Abstract

Introduction / Problem Definition

Main Body
  - Architecture of Solution
  - Description of Solution

Implementation / Experiment

Conclusion

References
The paper demonstrates an application of a novel way to perform ETL, via MapReduce. MapReduce is generally not a viable approach to do ETL, and vanilla ETL is normally not parallelizable. However, this paper suggests a novel solution to the problem.

The paper presents a custom MapReduce based ETL system called ETLMR which solves the ETL with MapReduce problem via the creation of a custom framework in Python.

The results are encouraging as linear speedup is seen based on the number of tasks/processes.
This is the second published paper on the ETLMR process. This paper was published in 2012, with the first ETLMR paper published in 2011 by the same authors.

MapReduce was invented by Google and became widely known after the publication of their 2004 paper, “MapReduce: Simplified Data Processing on Large Clusters”

This paper discusses a way to solve the linear issues of ETL so that we can bring parallelization advantages of MapReduce to ETL.

The two papers are unique – there are no other papers that touch on the topic of ETL with MapReduce topic prior to them. There has been a renewed interest in ETL with MapReduce recently, but it has focused on different technologies, mainly Hadoop.
WHAT IS ETLMR?

Combination of two separate concepts:

ETL (Extract, Transform, and Load)
  - An extremely common database pattern, used for loading data from one data source (Extract), change the data into your schema (Transform) and then put the data into your database (Load).

- and -

MR (Map Reduce)
  - A general framework, originally created by Google, enabling fault tolerant parallelization of workloads by breaking workloads into two simple operations – Map and Reduce.
THE PROBLEMS WITH VANILLA ETL

ETL is a (mostly) linear process

It does not scale well

Modern enterprises commonly deal with hundreds of GBs (and sometimes PBs) of data of data they need to extract, transform and load every day.

ETL flow is inherently complex, theoretically making a poor fit for parallelization.
MAPREDUCE — HIGH LEVEL OVERVIEW

From: Google’s Original 2004 Paper:
THE PROBLEM WITH MAP REDUCE

Generic programming model

Lacks support for higher-order processes necessary for ETL, such as support for star schemas, snowflake schemas, slowly changing dimensions, etc.

Because it is low level, getting productive ETL work done using MapReduce takes a long time.

This makes it impractical for ETL processes.
SOLUTION: ETLMR

Consists of a library of ETL-specific functionalities for MapReduce

Offers high-level constructs such as:
- Facts tables
- Dimensions (including Slowly Changing Dimensions)
- Support for Star Schemas and Snowflake Schemas

User creates a simple configuration file and the software does the rest.

Makes Map Reduce easy – user doesn’t have to know details.
Python
- Easy-to-use dynamic programming language

Pygrametl
- ETL Library for Python
- Available from: http://pygrametl.org/

Disco
- Lightweight implementation of MapReduce in Python
- http://discoproject.org/
DISCO VS HADOOP

**Disco (count words):**

```python
from disco.core import Job, result_iterator

def map(line, params):
    for word in line.split():
        yield word, 1

def reduce(iter, params):
    from disco.util import kvgroup
    for word, counts in kvgroup(sorted(iter)):
        yield word, sum(counts)
```

```java
if __name__ == '__main__':
    input = ['http://discoproject.org/media/text/chekhov.txt']
    job = Job().run(input=input, map=map, reduce=reduce)
    for word, count in result_iterator(job.wait()):
        print word, count
```

**Hadoop (count words):**

```
package org.moryq;
import java.io.IOException;
import java.util.*;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.util.*;

public class WordCount {

    public static class Map extends MapReduceBase implements Mapper<LongWritable, Text, Text, IntWritable> {
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();

        public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
            String line = value.toString();
            String tokenizer = new StringTokenizer(line);
            while (tokenizer.hasMoreTokens()) {
                word.set(tokenizer.nextToken());
                output.collect(word, one);
            }
        }
    }

    public static class Reduce extends MapReduceBase implements Reducer<Text, IntWritable, Text, IntWritable> {
        public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
            int sum = 0;
            while (values.hasNext()) {
                sum += values.next().get();
            }
            output.collect(key, new IntWritable(sum));
        }
    }

    public static void main(String[] args) throws Exception {
        JobConf conf = new JobConf(WordCount.class);
        conf.setJobName("wordcount");
        JobConf.set(new JobConfWordCount.class);
        conf.setInputFormat(TextInputFormat.class);
        conf.setOutputFormat(TextOutputFormat.class);
        FileInputFormat.setInputPaths(conf, new Path(args[0]));
        FileOutputFormat.setOutputPath(conf, new Path(args[1]));
        JobClient.runJob(conf);
    }
}
```
PROCESS

Chunk the import data, put into same-size files

Then Process Dimensions with MapReduce

Next, Process the Facts with MapReduce

Finally do the Bulk Load into Data Warehouse
Can Accelerate these aspects using Map-Reduce

Chunk Data → Process Dimensions → Process Facts → Bulk Load
ETLMR ALGORITHM

Algorithm 1. The ETL process

1: Partition the input data sets;
2: Read the configuration parameters and initialize;
3: Read the input data and relay the data to the map function in the map readers;
4: Process dimension data and load it into dimension stores;
5: Synchronize the dimensions across the clustered computers, if applicable;
6: Prepare fact processing (connect to and cache dimensions);
7: Read the input data for fact processing and perform transformations in mappers;
8: Bulk load fact data into the DW.

Source: https://www.researchgate.net/publication/220802672_ETLMR_A_Highly_Scalable_Dimensional_ETL_Framework_Based_on_MapReduce
ETLMR SCHEMATIC

Figure 1: Parallel ETL using ETLMR

Source: http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=926B3E0491DE0B47B372034EF4F4D389?doi=10.1.1.294.842&rep=rep1&type=pdf
TARGET DATABASE IN DATA WAREHOUSE: PARTIALLY SNOW-FLAKED SCHEMA

Source: http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=926B3E0491DE0B47B372034EF4F4D389?doi=10.1.1.294.842&rep=rep1&type=pdf
DECLARE THE DIMENSIONS WITH MAPREDUCE

• Declarative syntax – no code required.

• Each Dimension has a corresponding definition in the configuration file.

• Can use various classes such as:
  SlowlyChangingDimension
  SnowflakedDimension

```python
# Declared in the configuration file, config.py
from odotables import *

# Declare the dimensions:
testdim=CachedDimension(name='test', key='testid', defaultid='1', attributes=['testname'], lookupatts=['testname'])
datedim = CachedDimension(name='date', key='dateid', attributes=['date', 'day', 'month', 'year', 'week'], lookupatts=['date'])

# Declare the dimension tables of the normalized pagedim.
pagedim = SlowlyChangingDimension(name='page', key='pageid', lookupatts=['url'], attributes=['url', 'size', 'validfrom', 'validto', 'version', 'domainid', 'serverversionid'], versionatts='version', srcdateatts='lastmoddate', fromatts='validfrom', toatts='validto')
topdomaindim=CachedDimension(name='topdomain', key='topdomainid', attributes=['topdomain'], lookupatts=['topdomain'])
domaindim = CachedDimension(name='domain', key='domainid', attributes=['domain', 'topdomainid'], lookupatts=['domain'])
serverdim = CachedDimension(name='server', key='serverid', attributes=['server'], lookupatts=['server'])
serverversiondim=CachedDimension(name='serverversion', key='serverversionid', attributes=['serverversion', 'serverid'], lookupatts=['serverversion'], refdims=[serverdim])

# Define the references in the snowflaked dimension:
pagesdim=SnowflakedDimension(
  (pagedim, (serverversiondim, domaindim)),
  (serverversiondim, serverdim), (domaindim, topdomaindim))
```
DECLARE THE TRANSFORMATIONS

- Transformations are applied to each row.

- References between tables of snowflaked (normalized) dimensions are declared here.

- The dimensions are processed from the leaves towards the roots.

```python
# Defined in config.py
dims={
pagedim: {'srcfields': ('url', 'serverversion', 'domain', 'size', 'lastmoddate'),
             'rowhandlers': (UDF_extractdomain, UDF_extractserverver)},
                 domaindim: {'srcfields': ('url',),
                      'rowhandlers': (UDF_extractdomain,),
                         topdomaindim: {'srcfields': ('uri',),
                                        'rowhandlers': (UDF_extracttopdomain,),
                                           serverversiondim: {'srcfields': ('serverversion',),
                                                              'rowhandlers': (UDF_extractserverver,),
                                                       serverdim: {'srcfields': ('serverversion',),
                                                                  'rowhandlers': (UDF_extractserver,),
                                                       datedim: {'srcfields': ('downloaddate',),
                                                                  'rowhandlers': (UDF_explodedate,),
                                           testdim: {'srcfields': ('test',), 'rowhandlers': ()}
                          }
                     }
                }
```
DECLARE THE OUTPUT FACT TABLE

- For the example, only one Fact table is used.
- Multiple Fact tables can be declared, they will be processed in parallel.
- Data files are assigned to the Map/reduce in a round-robin fashion.
- User-defined transformation is applied to the rows from the data file, keys are looked up and output is piped into destination buffer.
- When destination buffer is full, data is loaded into Data Warehouse via the Bulk Loader.

```python
# In config.py
# Declare the fact table (here we support bulk loading):
testresultsfact = BulkFactTable(name='testresultsfact',
    keyrefs=['pageid','testid','dateid'],
    measures=['errors'],
    bulkloader=UDF_pgcopy, bulksize=5000000)

# Set the referenced dimensions and the transformations to apply to facts:
facts = {testresultsfact:
    {'refdims':(pagedim, datedim, testdim),
    'rowhandlers':(UDF_convertStrToInt,)})
```
We need to perform user defined transformations and a projection for each of the dimensions in our input.

One way to do it is to separate out individual dimensions and give each dimension to a single individual reducer for final processing.

Straightforward, but problematic because if the dimension size is large, the reducer might run out of memory, or the results are unbalanced if some dimensions are of different sizes from other dimensions.
Solution: Split each dimension into equal-sized chunks, and give each reducer these chunks of data to process.

Map output is partitioned in a round-robin fashion.

Each reducer processes the output of all dimensions.

This step requires an internal DBMS for bookkeeping, and results in a duplication of keys because keys created as part of processing each dimension chunk may be duplicated.
Chunking of dimensions can create duplication of keys since the reducers are independent of other reducers that are processing the same dimension.

Solution is a process called “Post-fixing”, where the duplicate keys are reconciled after the reducers have completed their work.

Requires that uniqueness constraint on the dimension key must be disabled before data processing for this to work.
RESULTS / SCALABILITY

80 GB data set used with 13,918,502 rows in page dimension.

Map/Reduce tasks were scaled from 4 to 20 to get details about scalability.

Result: ETLMR achieved nearly a linear speedup in processing:

<table>
<thead>
<tr>
<th>no. of tasks</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (min)</td>
<td>260.95</td>
<td>135.65</td>
<td>91.39</td>
<td>70.73</td>
<td>55.22</td>
</tr>
</tbody>
</table>
RESULTS (CONTINUED)

Speedup is near linear with the number of processors/tasks:

Figure 8: Speedup with increasing tasks, 80GB

Figure 9: Processing time when scaling up data size
NOTES ON THE PROCESS (PAST TO PRESENT)

This reviewed paper was published in 2012, and another paper by the same authors was published in 2011.

Because MapReduce itself is fairly young (2004) there is not a lot of other research out there on ETL/MapReduce.

The authors of the paper haven’t published anything more recent since 2012, but others have taken up their ideas and ran with them.

Other companies have stepped into the ring offering MapReduce and ETL: example CASK, Intel and Apache Sqoop. Both are using Hadoop for their solution.
THE PRESENT: CASK
THE PRESENT: INTEL

Intel published a white paper in 2013 that outlines using Hadoop with ETL:

THE PRESENT: APACHE SQOOP

Apache Sqoop is a framework similar to the framework outlined in the paper, but it uses Apache Hadoop as the underlying foundation.

Instead of having to describe the database, the Sqoop tool automatically introspects the database, saving an important step.

Source:
http://hortonworks.com/apache/sqoop/#section_2
SOURCES

MapReduce Based ETL (2012) – The main article:

A Highly Scalable Dimensional ETL Framework Based on Map Reduce (2011):
https://www.researchgate.net/publication/220802672_ETLMR_A_Highly_Scalable_Dimensio
nal_ETL_Framework_Based_on_MapReduce

MapReduce: Simplified Data Processing on Large Clusters (2004):
http://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-
osdi04.pdf

Intel: ETL Big Data With Hadoop (2013):