Parallel Databases vs. Hadoop

Juliana Freire
New York University

Some slides from J. Lin, C. Olston et al.
The Debate Starts…

MapReduce: A major step backwards

By David DeWitt on January 17, 2008 4:20 PM | Permalink | Comments (44) | TrackBacks (1)

[Note: Although the system attributes this post to a single author, it was written by David J. DeWitt and Michael Stonebraker]

On January 8, a Database Column reader asked for our views on new distributed database research efforts, and we’ll begin here with our views on MapReduce. This is a good time to discuss it, since the recent trade press has been filled with news of the revolution of so-called "cloud computing." This paradigm entails harnessing large numbers of (low-end) processors working in parallel to solve a computing problem. In effect, this suggests constructing a data center by lining up a large number of "jelly beans" rather than utilizing a much smaller number of high-end servers.

For example, IBM and Google have announced plans to make a 1,000 processor cluster available to a few select universities to teach students how to program such clusters using a software tool called MapReduce [1]. Berkeley has gone so far as to plan on teaching their freshman how to program using the MapReduce framework.

As both educators and researchers, we are amazed at the hype that the MapReduce proponents have spread about how it represents a paradigm shift in the development of scalable, data-intensive applications. MapReduce may be a good idea for writing certain types of general-purpose computations, but to the database community, it is:

1. A giant step backward in the programming paradigm for large-scale data intensive applications
2. A sub-optimal implementation, in that it uses brute force instead of indexing
3. Not novel at all -- it represents a specific implementation of well known techniques developed nearly 25 years ago
4. Missing most of the features that are routinely included in current DBMS
5. Incompatible with all of the tools DBMS users have come to depend on

http://homes.cs.washington.edu/~billhowe/mapreduce_a_major_step_backwards.html
The Debate Continues…

- A comparison of approaches to large-scale data analysis. Pavlo et al., SIGMOD 2009
  
  *Parallel DBMS beats MapReduce by a lot!*

- Many were outraged by the comparison

- MapReduce: A Flexible Data Processing Tool. Dean and Ghemawat, CACM 2010
  
  - Pointed out inconsistencies and mistakes in the comparison

- MapReduce and Parallel DBMSs: Friends or Foes? Stonebraker et al., CACM 2010
  
  - Toned down claims…
Why use Database Systems?

- Declarative query languages
- Data independence
- Efficient access through optimization
- Data integrity and security
  - Safeguarding data from failures and malicious access
- Concurrent access
- Reduced application development time
- Uniform data administration
Parallel DBMSs

- Old and mature technology --- late 80’s: Gamma, Grace
- Aim to improve performance by executing operations in parallel
- Benefit: easy and cheap to scale
  - Add more nodes, get higher speed
  - Reduce the time to run queries
  - Higher transaction rates
  - Ability to process more data
- Challenge: minimize overheads and efficiently deal with contention
This partitioning of data is implemented in two basic ways. The main differentiation is whether or not physical data partitioning is used as a foundation—and therefore a static prerequisite—for parallelizing the work. These fundamental approaches are known as shared everything architecture and shared nothing architecture respectively.

In a shared nothing system, the system is divided into individual parallel processing units. Each processing unit has its own processing power (cores) and its own storage component (disks) and its CPU cores are solely responsible for its individual data set on its own disks. The only way to access a specific piece of data is to use the processing unit that owns this subset of data. Such systems are also commonly known as Massively Parallel Processing (MPP) systems. In order to achieve a good workload distribution shared nothing systems have to use a hash algorithm to partition data evenly across all available processing units. The partitioning strategy has to be decided upon initial creation of the system. As a result, shared nothing systems introduce mandatory, fixed parallelism in their systems in order to perform operations that involve table scans; the fixed parallelism completely relies on a fixed static data partitioning at database or object creation time. Most non-Oracle data warehouse systems are shared nothing systems.

Oracle Database relies on a shared everything architecture. This architecture does not require any pre-defined data partitioning to enable parallelism; however by using Oracle Partitioning, Oracle Database can deliver the exact same parallel processing capabilities as a shared nothing system. It does so however without the restrictions of the fixed parallel access encompassed in the data
Linear vs. Non-Linear Speedup

Speedup

# processors (=P)
Achieving Linear Speedup: Challenges

- Start-up cost: starting an operation on many processors
- Contention for resources between processors
- Data transfer
- Slowest processor becomes the bottleneck
- Data skew $\rightarrow$ load balancing
- Shared-nothing:
  - Most scalable architecture
  - Minimizes resource sharing across processors
  - Can use commodity hardware
  - Hard to program and manage

Any feeling of déjà vu?
What to Parallelize?

- Inter-query parallelism
  - Each query runs on one processor

- Inter-operator parallelism
  - A query runs on multiple processors
  - An operator runs on one processor

- Intra-operator parallelism
  - An operator runs on multiple processors
Partitioning Data

Figure 3 - Automatic hash-based data distribution

From: Greenplum Database Whitepaper
Partitioning a Relation across Disks

• If a relation contains only a few tuples which will fit into a single disk block, then assign the relation to a single disk.

• Large relations are preferably partitioned across all the available disks.

• The distribution of tuples to disks may be skewed — that is, some disks have many tuples, while others may have fewer tuples.

• Ideal: partitioning should be balanced.
Horizontal Data Partitioning

- Relation R split into P chunks $R_0, ..., R_{P-1}$, stored at the P nodes
- *Round robin*: tuple $T_i$ to chunk $(i \mod P)$
- *Hash-based partitioning on attribute A*:
  - Tuple $t$ to chunk $h(t.A) \mod P$
- *Range-based partitioning on attribute A*:
  - Tuple $t$ to chunk $i$ if $v_{i-1} < t.A < v_i$
  - E.g., with a partitioning vector [5,11], a tuple with partitioning attribute value of 2 will go to disk 0, a tuple with value 8 will go to disk 1, while a tuple with value 20 will go to disk 2.

Partitioning in Oracle:
http://docs.oracle.com/cd/B10501_01/server.920/a96524/c12parti.htm

Partitioning in DB2:
Range Partitioning in DBMSs

CREATE TABLE sales_range
(salesman_id NUMBER(5),
salesman_name VARCHAR2(30),
sales_amount NUMBER(10),
sales_date DATE)
PARTITION BY RANGE(sales_date)
(
PARTITION sales_jan2000 VALUES LESS THAN(TO_DATE('02/01/2000', 'DD/MM/YYYY')),
PARTITION sales_feb2000 VALUES LESS THAN(TO_DATE('03/01/2000', 'DD/MM/YYYY')),
PARTITION sales_mar2000 VALUES LESS THAN(TO_DATE('04/01/2000', 'DD/MM/YYYY')),
PARTITION sales_apr2000 VALUES LESS THAN(TO_DATE('05/01/2000', 'DD/MM/YYYY'))
);

CREATE TABLE sales_hash
(salesman_id NUMBER(5),
salesman_name VARCHAR2(30),
sales_amount NUMBER(10),
week_no NUMBER(2))
PARTITION BY HASH(salesman_id)
PARTITIONS 4
STORE IN (data1, data2, data3, data4);
Parallel Selection

- Compute
  
  \[
  \text{SELECT * FROM } R
  \]
  
  \[
  \text{WHERE } R.A < v_1 \text{ AND } R.A > v_2
  \]

  \[
  \text{SELECT * FROM } R
  \]
  
  \[
  \text{WHERE } R.A = v
  \]

- What is the cost of these operations on a conventional database?
  
  Cost = B(R)

- What is the cost of these operations on a parallel database with P processors?
Parallel Selection: Round Robin

\[ \text{tuple } T_i \text{ to chunk } (i \mod P) \]

(1) \( \text{SELECT * FROM R} \)
(2) \( \text{SELECT * FROM R} \)
WHERE \( R.A = v \)

(3) \( \text{SELECT * FROM R} \)
WHERE \( R.A < v_1 \text{ AND } R.A > v_2 \)

- Best suited for sequential scan of entire relation on each query. Why?
- All disks have almost an equal number of tuples; retrieval work is thus well balanced between disks
- Range queries are difficult to process --- tuples are scattered across all disks
CREATE TABLE employees (  
id INT NOT NULL,  
fname VARCHAR(30),  
lname VARCHAR(30),  
hired DATE NOT NULL DEFAULT '1970-01-01',  
separated DATE NOT NULL DEFAULT '9999-12-31',  
job_code INT,  
store_id INT  
)
PARTITION BY HASH(store_id)
PARTITIONS 4;

Parallel Selection: Hash Partitioning

(1) SELECT * FROM R
(2) SELECT * FROM R
   WHERE R.A = v
(3) SELECT * FROM R
   WHERE R.A < v1 AND R.A > v2

- Good for sequential access
- Assuming hash function is good, and partitioning attributes form a key, tuples will be equally distributed between disks
- Retrieval work is then well balanced between disks
- Good for point queries on partitioning attribute
- Can lookup single disk, leaving others available for answering other queries.
- Index on partitioning attribute can be local to disk, making lookup and update more efficient
- No clustering, so difficult to answer range queries
CREATE TABLE employees (  
id INT NOT NULL,  
fname VARCHAR(30),  
lname VARCHAR(30),  
hired DATE NOT NULL DEFAULT '1970-01-01',  
job_code INT NOT NULL,  
store_id INT NOT NULL
)
PARTITION BY RANGE (store_id) (  
PARTITION p0 VALUES LESS THAN (6),  
PARTITION p1 VALUES LESS THAN (11),  
PARTITION p2 VALUES LESS THAN (16),  
PARTITION p3 VALUES LESS THAN (21)
);
Parallel Selection: Range Partitioning

1. SELECT * FROM R
2. SELECT * FROM R
   WHERE R.A = v
3. SELECT * FROM R
   WHERE R.A < v1 AND R.A > v2

- Provides data clustering by partitioning attribute value
- Good for sequential access
- Good for point queries on partitioning attribute: only one disk needs to be accessed.
- For range queries on partitioning attribute, one to a few disks may need to be accessed.
- Remaining disks are available for other queries
- Caveat: badly chosen partition vector may assign too many tuples to some partitions and too few to others
Parallel Join

- The join operation requires pairs of tuples to be tested to see if they satisfy the join condition, and if they do, the pair is added to the join output.
- Parallel join algorithms attempt to split the pairs to be tested over several processors. Each processor then computes part of the join locally.
- In a final step, the results from each processor can be collected together to produce the final result.

How would you implement a join in MapReduce?
Partitioned Join

- For equi-joins and natural joins, it is possible to *partition* the two input relations across the processors, and compute the join locally at each processor.
- Let \( r \) and \( s \) be the input relations, and we want to compute \( r \bowtie_{r.A=s.B} s \).
- \( r \) and \( s \) each are partitioned into \( n \) partitions, denoted \( r_0, r_1, ..., r_{n-1} \) and \( s_0, s_1, ..., s_{n-1} \).
- Can use either *range partitioning* or *hash partitioning*.
- \( r \) and \( s \) must be partitioned on their join attributes \( r.A \) and \( s.B \), *using the same range-partitioning vector or hash function*.
- Partitions \( r_i \) and \( s_i \) are sent to processor \( P_i \).
- Each processor \( P_i \) locally computes \( r_i \bowtie_{r_i.A=s_i.B} s_i \). Any of the standard join methods can be used.
Pipelined Parallelism

• Consider a join of four relations
  • \( r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4 \)
• Set up a pipeline that computes the three joins in parallel
  • Let P1 be assigned the computation of \( \text{temp1} = r_1 \bowtie r_2 \)
  • And P2 be assigned the computation of \( \text{temp2} = \text{temp1} \bowtie r_3 \)
  • And P3 be assigned the computation of \( \text{temp2} \bowtie r_4 \)
• Each of these operations can execute in parallel, sending result tuples it computes to the next operation even as it is computing further results
  • Provided a *pipelineable* join evaluation algorithm (e.g., indexed nested loops join) is used
Can we implement pipelined joins in MapReduce?
Factors Limiting Utility of Pipeline Parallelism

- Pipeline parallelism is useful since it avoids writing intermediate results to disk.
- Cannot pipeline operators which do not produce output until all inputs have been accessed (e.g., blocking operations such as aggregate and sort).
MapReduce: A Step Backwards
Dewitt and Stonebraker Views

- We are amazed at the hype that the MapReduce proponents have spread about how it represents a paradigm shift in the development of scalable, data-intensive applications.
- 1. A giant step backward in the programming paradigm for large-scale data intensive applications.
- 2. A sub-optimal implementation, in that it uses brute force instead of indexing.
- 3. Not novel at all -- it represents a specific implementation of well known techniques developed nearly 25 years ago.
- 4. Missing most of the features that are routinely included in current DBMS.
- 5. Incompatible with all of the tools DBMS users have come to depend on.

http://homes.cs.washington.edu/~billhowe/mapreduce_a_major_step_backwards.html
The database community has learned the following three lessons over the past 40 years:

- Schemas are good.
- Separation of the schema from the application is good.
- High-level access languages are good.

MapReduce has learned none of these lessons:

- MapReduce is a poor implementation.
- MapReduce advocates have overlooked the issue of skew.

Skew is a huge impediment to achieving successful scale-up in parallel query systems.

When there is wide variance in the distribution of records with the same key, lead some reduce instances to take much longer to run than others → execution time for the computation is the running time of the slowest reduce instance.
I/O bottleneck: $N$ map instances produces $M$ output files

If $N$ is 1,000 and $M$ is 500, the map phase produces 500,000 local files. When the reduce phase starts, each of the 500 reduce instances needs to read its 1,000 input files and must use a protocol like FTP to "pull" each of its input files from the nodes on which the map instances were run.

In contrast, Parallel Databases do not materialize their split files.

...
Case for Parallel Databases

Pavlo et al., SIGMOD 2009
MapReduce vs. Parallel Databases

- [Pavlo et al., SIGMOD 2009] compared the performance of Hadoop against Vertica and a Parallel DBMS
- Why use MapReduce when parallel databases are so efficient and effective?
- Point of view from the perspective of database researchers
- Compare the different approaches and perform an experimental evaluation
Architectural Elements: ParDB vs. MR

- Schema support:
  - Relational paradigm: rigid structure of rows and columns
  - Flexible structure, but need to write parsers and challenging to share results

- Indexing
  - B-trees to speed up access
  - No built-in indexes --- programmers must code indexes

- Programming model
  - Declarative, high-level language
  - Imperative, write programs

- Data distribution
  - Use knowledge of data distribution to automatically optimize queries
  - Programmer must optimize the access
Architectural Elements: ParDB vs. MR

- Execution strategy and fault tolerance:
  - Pipeline operators (push), failures dealt with at the transaction level
  - Write intermediate files (pull), provide fault tolerance
# Architectural Elements

<table>
<thead>
<tr>
<th></th>
<th>Parallel DBMS</th>
<th>MapReduce</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Schema Support</strong></td>
<td>✓</td>
<td>Not out of the box</td>
</tr>
<tr>
<td><strong>Indexing</strong></td>
<td>✓</td>
<td>Not out of the box</td>
</tr>
<tr>
<td><strong>Programming Model</strong></td>
<td>Declarative (SQL)</td>
<td>Imperative (C/C++, Java, ...) Extensions through Pig and Hive</td>
</tr>
<tr>
<td><strong>Optimizations</strong></td>
<td>✓</td>
<td>Not out of the box</td>
</tr>
<tr>
<td><strong>Flexibility</strong></td>
<td>Not out of the box</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Fault Tolerance</strong></td>
<td>Coarse grained techniques</td>
<td>✓</td>
</tr>
</tbody>
</table>

[Pavlo et al., SIGMOD 2009, Stonebraker et al., CACM 2010, …]
Experimental Evaluation

• 5 tasks --- including task from original MapReduce paper
• Compared: Hadoop, DBMS-X, Vertica
• 100-node cluster
• Test speedup using clusters of size 1, 10, 25, 50, 100 nodes
  • Fix the size of data in each node to 535MB (match MapReduce paper)
  • Evenly divide data among the different nodes
• We will look at the Grep task --- see paper for details on the other tasks
• Hadoop outperforms both Vertica and DBMS-X

**Figure 1:** Load Times – Grep Task Data Set (535MB/node)

**Figure 2:** Load Times – Grep Task Data Set (1TB/cluster)
Grep Task

- Scan through a data set of 100-byte records looking for a three-character pattern. Each record consists of a unique key in the first 10 bytes, followed by a 90-byte random value.

SELECT * FROM Data WHERE field LIKE ‘%XYZ%’
Grep Task: Analysis

- Fig 4: Little data is processed in each node --- start-up costs for Hadoop dominate
  “that takes 10–25 seconds before all Map tasks have been started and are running at full speed across the nodes in the cluster”

Figure 4: Grep Task Results – 535MB/node Data Set
Grep Task: Analysis

- Fig 5: Hadoop’s start-up costs are amortized --- more data processed in each node
- Vertica’s superior performance is due to aggressive compression

Figure 5: Grep Task Results – 1TB/cluster Data Set
Discussion

• Installation, configuration and use:
  • Hadoop is easy and free
  • DBMS-X is very hard --- lots of tuning required; and very expensive
• Task start-up is an issue with Hadoop
• Compression is helpful and supported by DBMS-X and Vertica
• Loading is much faster on Hadoop --- 20x faster than DBMS-X
• If data will be processed only a few times, it might not be worth it to use a parallel DBMS
Case for MapReduce

By Dean and Ghemawat, CACM 2010
MapReduce vs. Parallel Databases

- [Dean and Ghemawat, CACM 2010] criticize the comparison by Pavlo et al.
- Point of view from the creators of MapReduce
- Discuss misconceptions in Pavlo et al.
  - MapReduce cannot use indices
  - Inputs and outputs are always simple files in a file system
  - Require inefficient data formats
  - MapReduce provides storage independence and fine-grained fault tolerance
  - Supports complex transformations
Heterogeneous Systems

- Production environments use a plethora of storage systems: files, RDBMS, Bigtable, column stores
- MapReduce can be extended to support different storage backends — it can be used to combine data from different sources
- Parallel databases require all data to be loaded
- Would you use a ParDB to load Web pages retrieved by a crawler and build an inverted index?
An Aside: BigTable

- BigTable is a distributed storage system for managing structured data.
- Designed to scale to a very large size
  - Petabytes of data across thousands of servers
- Used for many Google projects
  - Web indexing, Personalized Search, Google Earth, Google Analytics, Google Finance, …
- Flexible, high-performance solution for Google’s products

[Chang et al., OSDI 2006]
BigTable: Motivation

- Lots of (semi-)structured data at Google
  - URLs:
    - Contents, crawl metadata, links, anchors, pagerank, …
  - Per-user data:
    - User preference settings, recent queries/search results, …
  - Geographic locations:
    - Physical entities (shops, restaurants, etc.), roads, satellite image data, user annotations, …
- Scale is large
  - Billions of URLs, many versions/page (~20K/version)
  - Hundreds of millions of users, thousands or q/sec
  - 100TB+ of satellite image data
Why not just use commercial DB?

- Scale is too large for most commercial databases
- Even if it weren’t, cost would be very high
  - Building internally means system can be applied across many projects for low incremental cost
- Low-level storage optimizations help performance significantly
  - Much harder to do when running on top of a database layer
Goals

• Want asynchronous processes to be continuously updating different pieces of data
  • Want access to most current data at any time

• Need to support:
  • Very high read/write rates (millions of ops per second)
  • Efficient scans over all or interesting subsets of data
  • Efficient joins of large one-to-one and one-to-many datasets

• Often want to examine data changes over time
  • E.g., Contents of a web page over multiple crawls
BigTable

- Distributed multi-level map
- Fault-tolerant, persistent
- Scalable
  - Thousands of servers
  - Terabytes of in-memory data
  - Petabyte of disk-based data
  - Millions of reads/writes per second, efficient scans
- Self-managing
  - Servers can be added/removed dynamically
  - Servers adjust to load imbalance
Data model: a big map

- `<Row, Column, Timestamp> → string`
  - triple for key - lookup, insert, and delete API
- Arbitrary “columns” on a row-by-row basis
  - Column-oriented physical store- rows are sparse!
- Does not support a relational model
  - No table-wide integrity constraints
  - No multirow transactions: read/write row is atomic
Indices

- Techniques used by DBMSs can also be applied to MapReduce
- For example, HadoopDB gives Hadoop access to multiple single-node DBMS servers (e.g., PostgreSQL or MySQL) deployed across the cluster.
  - It pushes as much as possible data processing into the database engine by issuing SQL queries (usually most of the Map/Combine phase logic is expressible in SQL).
- Indexing can also be obtained through appropriate partitioning of the data, e.g., *range partitioning*.
  - Log files are partitioned based on date ranges.
Complex Functions

• MapReduce was designed for complex tasks that manipulate diverse data:
  • Extract links from Web pages and aggregating them by target document
  • Generate inverted index files to support efficient search queries
  • Process all road segments in the world and rendering map images
• These data do not fit well in the relational paradigm
  • Remember: SQL is not Turing-complete!
• RDMS supports UDF, but these have limitations
  • Buggy in DBMS-X and missing in Vertica
Structured Data and Schemas

• Schemas are helpful to share data
• Google’s MapReduce implementation supports the Protocol Buffer format
• A high-level language is used to describe the input and output types
  • Compiler-generated code hides the details of encoding/decoding data
  • Use optimized binary representation --- compact and faster to encode/decode; huge performance gains – 80x for example in paper!
Protocol Buffer format

Quick Example

You write a .proto file like this:

```proto
message Person {
  required int32 id = 1;
  required string name = 2;
  optional string email = 3;
}
```

Then you compile it with protoc, the protocol buffer compiler, to produce code in C++, Java, or Python.

Then, if you are using C++, you use that code like this:

```cpp
Person person;
person.set_id(123);
person.set_name("Bob");
person.set_email("bob@example.com");

fstream out("person.pb", ios::out | ios::binary | ios::trunc);
person.SerializeToOstream(&out);
out.close();
```

Or like this:

```cpp
Person person;
fstream in("person.pb", ios::in | ios::binary);
if (!person.ParseFromIstream(&in)) {
  cerr << "Failed to parse person.pb." << endl;
  exit(1);
}

cout << "ID: " << person.id() << endl;
cout << "name: " << person.name() << endl;
if (person.has_email()) {
  cout << "e-mail: " << person.email() << endl;
}
```

http://code.google.com/p/protobuf/
Hadoop Alternatives

- Avro: http://avro.apache.org/docs/current
- Thrift: http://thrift.apache.org/
Fault Tolerance

- Pull model is necessary to provide fault tolerance
- It may lead to the creation of many small files
- Use implementation tricks to mitigate these costs
  - Keep this in mind when writing your MapReduce programs!
Conclusions

It doesn’t make much sense to compare MapReduce and Parallel DBMS: they were designed for different purposes!

You can do anything in MapReduce
it may not be easy, but it is possible

MapReduce is free, Parallel DB are expensive

Growing ecosystem around MapReduce is making it more similar to PDBMSs
Conclusions (cont.)

- There is a lot of ongoing work on adding DB features to the Cloud environment
- Spark: support streaming
  https://spark.apache.org/
- Shark: large-scale data warehouse system for Spark
- SQL API https://amplab.cs.berkeley.edu/software/
- Now https://spark.apache.org/sql/
- HadoopDB: hybrid of DBMS and MapReduce technologies that targets analytical workloads
- ElasTraS: An elastic, scalable, and self-managing transactional database for the cloud
- Twister: enhanced runtime that supports iterative MapReduce computations efficiently
Conclusions (cont.)

- Apache Hbase: random, real-time read/write access to Big Data
- Apache Accumulo (https://accumulo.apache.org): distributed key/value store – similar to BigTable
- High-level languages
  - Pig and Hive
- Hive (https://hive.apache.org): data warehouse that supports querying and managing large data residing in distributed storage
Questions