CSE 544
Parallel Databases

Tuesday, February 17\textsuperscript{th}, 2011
Final Thoughts on Optimization: Parameters!

Chaudhuri “Rethinking the Contract”
Overview of Today’s Lecture

- Parallel databases (Chapter 22.1 – 22.5)
- Map/reduce
- Pig-Latin
  - Some slides from Alan Gates (Yahoo! Research)
Parallel v.s. Distributed Databases

• Parallel database system:
  – Improve performance through parallel implementation
  – Will discuss in class

• Distributed database system:
  – Data is stored across several sites, each site managed by a DBMS capable of running independently
  – Will not discuss in class
Parallel DBMSs

• **Goal**
  – Improve performance by executing multiple operations in parallel

• **Key benefit**
  – Cheaper to scale than relying on a single increasingly more powerful processor

• **Key challenge**
  – Ensure overhead and contention do not kill performance
Performance Metrics for Parallel DBMSs

• **Speedup**
  – More processors \(\Rightarrow\) higher speed
  – Individual queries should run faster
  – Should do more transactions per second (TPS)

• **Scaleup**
  – More processors \(\Rightarrow\) can process more data
  – **Batch scaleup**
    • Same query on larger input data should take the same time
  – **Transaction scaleup**
    • N-times as many TPS on N-times larger database
    • But each transaction typically remains small
Linear v.s. Non-linear Speedup

![Graph showing linear and non-linear speedup vs. number of processors (P)]
Linear v.s. Non-linear Scaleup

Batch Scaleup

\[ \times 1 \quad \times 5 \quad \times 10 \quad \times 15 \]

# processors (=P) AND data size

Dan Suciu -- 544, Winter 2011
Challenges to Linear Speedup and Scaleup

• **Startup cost**
  – Cost of starting an operation on many processors

• **Interference**
  – Contention for resources between processors

• **Skew**
  – Slowest processor becomes the bottleneck
Architectures for Parallel Databases

- Shared memory
- Shared disk
- Shared nothing
Shared Memory

P  P  P

Interconnection Network

Global Shared Memory

D  D  D
Shared Disk

Interconnection Network
Shared Nothing

Interconnection Network

P
M
D

P
M
D

P
M
D

Dan Suciu -- 544, Winter 2011
Shared Nothing

• Most scalable architecture
  – Minimizes interference by minimizing resource sharing
  – Can use commodity hardware

• Also most difficult to program and manage

• Processor = server = node
• P = number of nodes

We will focus on shared nothing
Taxonomy for Parallel Query Evaluation

• **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

We study only intra-operator parallelism: most scalable
Horizontal Data Partitioning

- Relation R split into P chunks $R_0, \ldots, R_{P-1}$, 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 $v_{i-1} < t.A < v_i$
Parallel Selection

Compute $\sigma_{A=v}(R)$, or $\sigma_{v_1<A<v_2}(R)$

• Conventional database:
  – Cost = $B(R)$

• Parallel database with $P$ processors:
  – Cost = $B(R) / P$
Parallel Selection

Different processors do the work:

• Round robin partition: all servers do the work
• Hash partition:
  – One server for $\sigma_{A=v}(R)$,
  – All servers for $\sigma_{v_1<A<v_2}(R)$
• Range partition: one server does the work
Data Partitioning Revisited

What are the pros and cons?

• Round robin
  – Good load balance but always needs to read all the data

• Hash based partitioning
  – Good load balance but works only for equality predicates and full scans

• Range based partitioning
  – Works well for range predicates but can suffer from data skew
Parallel Group By: $\gamma_A, \text{sum}(B)(R)$

Step 1: server $i$ partitions chunk $R_i$ using a hash function $h(t.A): R_{i0}, R_{i1}, \ldots, R_{i,P-1}$

Step 2: server $i$ sends partition $R_{ij}$ to server $j$

Step 3: server $j$ computes $\gamma_A, \text{sum}(B)$ on $R_{0j}, R_{1j}, \ldots, R_{P-1,j}$
Cost of Parallel Group By

Recall conventional cost = 3B(R)

• Step 1: Cost = B(R)/P I/O operations
• Step 2: Cost = (P-1)/P B(R) blocks are sent
  – Network costs << I/O costs
• Step 3: Cost = 2 B(R)/P
  – When can we reduce it to 0?

Total = 3B(R) / P + communication costs
Parallel Join: \( R \bowtie_{A=B} S \)

Step 1
- For all servers in \([0,k]\), server \( i \) partitions chunk \( R_i \) using a hash function \( h(t.A): R_{i0}, R_{i1}, \ldots, R_{i,P-1} \)
- For all servers in \([k+1,P]\), server \( j \) partitions chunk \( S_j \) using a hash function \( h(t.A): S_{j0}, S_{j1}, \ldots, R_{j,P-1} \)

Step 2:
- Server \( i \) sends partition \( R_{iu} \) to server \( u \)
- Server \( j \) sends partition \( S_{ju} \) to server \( u \)

Steps 3: Server \( u \) computes the join of \( R_{iu} \) with \( S_{ju} \)
Cost of Parallel Join

- **Step 1:** Cost = \((B(R) + B(S))/P\)

- **Step 2:** 0
  - \((P-1)/P\) \((B(R) + B(S))\) blocks are sent, but we assume network costs to be << disk I/O costs

- **Step 3:**
  - Cost = 0 if small table fits in memory: \(B(S)/P \leq M\)
  - Cost = \(4(B(R)+B(S))/P\) otherwise
Parallel Query Plans

• Same relational operators

• Add special split and merge operators
  – Handle data routing, buffering, and flow control

• Example: exchange operator
  – Inserted between consecutive operators in the query plan
Map Reduce

• Google: paper published 2004
• Free variant: Hadoop

• Map-reduce = high-level programming model and implementation for large-scale parallel data processing
Data Model

Files!

A file = a bag of $(key, \ value)$ pairs

A map-reduce program:

• Input: a bag of $(inputkey, \ value) pairs$
• Output: a bag of $(outputkey, \ value) pairs$
Step 1: the MAP Phase

User provides the MAP-function:

• Input: one \((\text{input key, value})\)
• Output: bag of \((\text{intermediate key, value})\) pairs

System applies the map function in parallel to all \((\text{input key, value})\) pairs in the input file
Step 2: the REDUCE Phase

User provides the REDUCE function:

• **Input:** (intermediate key, bag of values)

• **Output:** bag of output values

System groups all pairs with the same intermediate key, and passes the bag of values to the REDUCE function
Example

- Counting the number of occurrences of each word in a large collection of documents

```java
map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, "1");

reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    int result = 0;
    for each v in values:
        result += ParseInt(v);
    Emit(AsString(result));
```
MAP

(k1, v1) → (i1, w1)
(k2, v2) → (i2, w2)
(k3, v3) → (i3, w3)
...

REDUCE

(i1, w1)
(i2, w2)
(i3, w3)
...

Dan Suciu -- 544, Winter 2011
Map = GROUP BY, Reduce = Aggregate

R(documentKey, word)

SELECT word, sum(1) FROM R GROUP BY word
Implementation

• There is one master node
• Master partitions input file into $M$ splits, by key
• Master assigns workers (=servers) to the $M$ map tasks, keeps track of their progress
• Workers write their output to local disk, partition into $R$ regions
• Master assigns workers to the $R$ reduce tasks
• Reduce workers read regions from the map workers’ local disks
MR Phases
Interesting Implementation Details

• Worker failure:
  – Master pings workers periodically,
  – If down then reassigns its splits to all other workers  \( \rightarrow \) good load balance

• Choice of M and R:
  – Larger is better for load balancing
  – Limitation: master needs \( O(M \times R) \) memory
Interesting Implementation Details

Backup tasks:

- **Straggler** = a machine that takes unusually long time to complete one of the last tasks. Eg:
  - Bad disk forces frequent correctable errors (30MB/s $\rightarrow$ 1MB/s)
  - The cluster scheduler has scheduled other tasks on that machine

- Stragglers are a main reason for slowdown

- **Solution**: pre-emptive backup execution of the last few remaining in-progress tasks
Map-Reduce Summary

• Hides scheduling and parallelization details

• However, very limited queries
  – Difficult to write more complex tasks
  – Need multiple map-reduce operations

• Solution: PIG-Latin !
Following Slides courtesy of: Alan Gates, Yahoo!Research
What is Pig?

- An engine for executing programs on top of Hadoop
- It provides a language, Pig Latin, to specify these programs
- An Apache open source project
  http://hadoop.apache.org/pig/
Map-Reduce

- Computation is moved to the data
- A simple yet powerful programming model
  - Map: every record handled individually
  - Shuffle: records collected by key
  - Reduce: key and iterator of all associated values
- User provides:
  - input and output (usually files)
  - map Java function
  - key to aggregate on
  - reduce Java function
- Opportunities for more control: partitioning, sorting, partial aggregations, etc.
Map Reduce Illustrated
Romeo, Romeo, wherefore art thou Romeo?  What, art thou hurt?
Map Reduce Illustrated

Romeo, Romeo, wherefore art thou Romeo?  What, art thou hurt?

Romeo, 1
Romeo, 1
wherefore, 1
art, 1
thou, 1
Romeo, 1

What, 1
art, 1
thou, 1
hurt, 1
Romeo, Romeo, wherefore art thou Romeo?

What, art thou hurt?

Romeo, 1
Romeo, 1
wherefore, 1
art, 1
thou, 1
Romeo, 1

art, (1, 1)
hurt (1),
thou (1, 1)

What, 1
art, 1
thou, 1
hurt, 1

Romeo, (1, 1, 1)
wherefore, (1)
what, (1)
Romeo, Romeo, wherefore art thou Romeo?

What, art thou hurt?

Romeo, 1
Romeo, 1
wherefore, 1
art, 1
thou, 1
Romeo, 1

What, 1
art, 1
thou, 1
hurt, 1

art, (1, 1)
hurt (1),
thou (1, 1)

Romeo, (1, 1, 1)
wherefore, (1)
what, (1)

art, 2
hurt, 1
thou, 2

Romeo, 3
wherefore, 1
what, 1
Making Parallelism Simple

- Sequential reads = good read speeds
- In large cluster failures are guaranteed; Map Reduce handles retries
- Good fit for batch processing applications that need to touch all your data:
  - data mining
  - model tuning
- Bad fit for applications that need to find one particular record
- Bad fit for applications that need to communicate between processes; oriented around independent units of work
Why use Pig?

Suppose you have user data in one file, website data in another, and you need to find the top 5 most visited sites by users aged 18 - 25.

1. Load Users
2. Filter by age
3. Load Pages
4. Join on name
5. Group on url
6. Count clicks
7. Order by clicks
8. Take top 5
In Map-Reduce

```java
In Map-Reduce

```
In Pig Latin

Users = load 'users' as (name, age);
Fltrd = filter Users by
    age >= 18 and age <= 25;
Pages = load 'pages' as (user, url);
Jnd = join Fltrd by name, Pages by user;
Grpd = group Jnd by url;
Smmd = foreach Grpd generate group,
    COUNT(Jnd) as clicks;
Srted = order Smmd by clicks desc;
Top5 = limit Srted 5;
store Top5 into 'top5sites';

9 lines of code, 15 minutes to write
But can it fly?
Essence of Pig

- Map-Reduce is too low a level to program, SQL too high
- Pig Latin, a language intended to sit between the two:
  - Imperative
  - Provides standard relational transforms (join, sort, etc.)
  - Schemas are optional, used when available, can be defined at runtime
  - User Defined Functions are first class citizens
  - Opportunities for advanced optimizer but optimizations by programmer also possible
How It Works

Script
A = load
B = filter
C = group
D = foreach

Parser
Logical Plan ≈ relational algebra
Semantic Checks
Logical Optimizer
Logical Plan
Logical to Physical Translator
Physical Plan = physical operators to be executed
MapReduce Launcher
Map-Reduce Plan
Jar to hadoop

Map-Reduce Plan = physical operators broken into Map, Combine, and Reduce stages

Physical To MR Translator
Physical Plan
Cool Things We’ve Added In the Last Year

- Multiquery – Ability to combine multiple group bys into a single MR job (0.3)
- Merge join – If data is already sorted on join key, do join via merge in map phase (0.4)
- Skew join – Hash join for data with skew in join key. Allows splitting of key across multiple reducers to handle skew. (0.4)
- Zebra – Contrib project that provides columnar storage of data (0.4)
- Rework of Load and Store functions to make them much easier to write (0.7, branched but not released)
- Owl, a metadata service for the grid (committed, will be released in 0.8).
Fragment Replicate Join

Aka “Broadcast Join”
Fragment Replicate Join

Users = load 'users' as (name, age);
Pages = load 'pages' as (user, url);
Jnd = join Pages by user, Users by name using "replicated";
Fragment Replicate Join

Users = load 'users' as (name, age);
Pages = load 'pages' as (user, url);
Jnd = join Pages by user, Users by name using "replicated";

Aka "Broadcast Join"
Fragment Replicate Join

Users = load ‘users’ as (name, age);
Pages = load ‘pages’ as (user, url);
Jnd = join Pages by user, Users by name using “replicated”;

Aka “Broadcast Join”
Users = \texttt{load} ‘users’ as (name, age);
Pages = \texttt{load} ‘pages’ as (user, url);
Jnd = \texttt{join} Pages by user, Users by name using “replicated”;
Hash Join

Pages

Users
Hash Join

Users = load 'users' as (name, age);
Pages = load 'pages' as (user, url);
Jnd = join Users by name, Pages by user;
Hash Join

Users = \texttt{load} 'users' as (name, age);
Pages = \texttt{load} 'pages' as (user, url);
Jnd = \texttt{join} Users \texttt{by} name, Pages \texttt{by} user;
Hash Join

Users = \texttt{load} \ 'users' \ \texttt{as} \ (name, age);
Pages = \texttt{load} \ 'pages' \ \texttt{as} \ (user, url);
Jnd = \texttt{join} \ Users \ \texttt{by} \ name, \ Pages \ \texttt{by} \ user;
Hash Join

Users = load 'users' as (name, age);
Pages = load 'pages' as (user, url);
Jnd = join Users by name, Pages by user;

Map 1
User block n
(1, user)

Map 2
Page block m
(2, name)
Hash Join

Users = load 'users' as (name, age);
Pages = load 'pages' as (user, url);
Jnd = join Users by name, Pages by user;

Map 1
- User block n
  - (1, user)
  - (2, fred)
  - (2, fred)

Map 2
- Page block m
  - (1, jane)
  - (2, jane)
  - (2, jane)

Reducer 1
- (1, fred)
- (2, fred)

Reducer 2
- (1, jane)
- (2, jane)
Skew Join

Pages

Users
Skew Join

Users = load 'users' as (name, age);
Pages = load 'pages' as (user, url);
Jnd = join Pages by user, Users by name using "skewed";
Skew Join

Users = load 'users' as (name, age);
Pages = load 'pages' as (user, url);
Jnd = join Pages by user, Users by name using "skewed";
Skew Join

Users = load 'users' as (name, age);
Pages = load 'pages' as (user, url);
Jnd = join Pages by user, Users by name using "skewed";

Map 1
Pages block n

Map 2
Users block m
Skew Join

Users = load 'users' as (name, age);
Pages = load 'pages' as (user, url);
Jnd = join Pages by user, Users by name using "skewed";
Skew Join

Users = load 'users' as (name, age);
Pages = load 'pages' as (user, url);
Jnd = join Pages by user, Users by name using "skewed";
Skew Join

Users = load ‘users’ as (name, age);
Pages = load ‘pages’ as (user, url);
Jnd = join Pages by user, Users by name using “skewed”;

Map 1

Pages

Users

Map 2

Pages block n

Users block m

Reducer 1

Reducer 2

(1, user)

(2, name)

(1, fred, p1)

(1, fred, p2)

(2, fred)

(1, fred, p3)

(1, fred, p4)

(2, fred)
Merge Join

Pages
aaron
zach

Users
aaron
zach
Merge Join

Users = load ‘users’ as (name, age);
Pages = load ‘pages’ as (user, url);
Jnd = join Pages by user, Users by name using “merge”;

<table>
<thead>
<tr>
<th>Pages</th>
<th>Users</th>
</tr>
</thead>
<tbody>
<tr>
<td>aaron</td>
<td>aaron</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>zach</td>
<td>zach</td>
</tr>
</tbody>
</table>
Merge Join

Users = load 'users' as (name, age);
Pages = load 'pages' as (user, url);
Jnd = join Pages by user, Users by name using "merge";

<table>
<thead>
<tr>
<th>Pages</th>
<th>Users</th>
</tr>
</thead>
<tbody>
<tr>
<td>aaron</td>
<td>aaron</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>zach</td>
<td>zach</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Merge Join

Users = load 'users' as (name, age);
Pages = load 'pages' as (user, url);
Jnd = join Pages by user, Users by name using "merge";

Map 1

Pages
aaron...

Users
aaron...

Map 2

Pages
amr...

Users
aaron...

Pages
amy...

Users
amy...
Multi-store script

A = load ‘users’ as (name, age, gender, city, state);
B = filter A by name is not null;
C1 = group B by age, gender;
D1 = foreach C1 generate group, COUNT(B);
store D into ‘bydemo’;
C2 = group B by state;
D2 = foreach C2 generate group, COUNT(B);
store D2 into ‘bystate’;
Multi-Store Map-Reduce Plan

map
  --filter--
  split
    local rearrange
    local rearrange
  reduce
    demux
      package
      foreach
      package
      foreach
What are people doing with Pig

- At Yahoo ~70% of Hadoop jobs are Pig jobs
- Being used at Twitter, LinkedIn, and other companies
- Available as part of Amazon EMR web service and Cloudera Hadoop distribution
- What users use Pig for:
  - Search infrastructure
  - Ad relevance
  - Model training
  - User intent analysis
  - Web log processing
  - Image processing
  - Incremental processing of large data sets
What We’re Working on this Year

- Optimizer rewrite
- Integrating Pig with metadata
- Usability – our current error messages might as well be written in actual Latin
- Automated usage info collection
- UDFs in python
Research Opportunities

- Cost based optimization – how does current RDBMS technology carry over to MR world?
- Memory Usage – given that data processing is very memory intensive and Java offers poor control of memory usage, how can Pig be written to use memory well?
- Automated Hadoop Tuning – Can Pig figure out how to configure Hadoop to best run a particular script?
- Indices, materialized views, etc. – How do these traditional RDBMS tools fit into the MR world?
- Human time queries – Analysts want access to the petabytes of data available via Hadoop, but they don’t want to wait hours for their jobs to finish; can Pig find a way to answer analysts question in under 60 seconds?
- Map-Reduce-Reduce – Can MR be made more efficient for multiple MR jobs?
- How should Pig integrate with workflow systems?
- See more: [http://wiki.apache.org/pig/PigJournal](http://wiki.apache.org/pig/PigJournal)
Learn More

• Visit our website:  http://hadoop.apache.org/pig/
• On line tutorials
• A couple of Hadoop books are available that include chapters on Pig, search at your favorite bookstore
• Join the mailing lists:
  – pig-user@hadoop.apache.org for user questions
  – pig-dev@hadoop.apache.com for developer issues
• Contribute your work, over 50 people have so far