Parallel Data Processing

Prof. Yanlei Diao

Slides Courtesy of R. Ramakrishnan and J. Gehrke
Data Analytics for A Social Network

User profiles:
Background, pics, postings, friends...

Click Streams:
many billion rows/day
many TB/day

Data Loading:
High Volume + Transformation

Data Processing Backend

Quick lookups and updates:
Update your own profile, read friends’ profiles, write msgs,…

Analysis Queries:
Ad targeting, fraud detection, resource provisioning…
Some (Old) Numbers about Facebook

- 500 million active users
- 9.5% Internet traffic
- >30,000 servers
- Initial software: PHP + MySQL cluster + Memcached
- One of the largest MySQL cluster
- Stores >20 billion photos, and serves 1 million img/sec.
- >4.5 billion msgs/day
- >15 TB click logs/day
Google AlphaGo
Three Forms of Parallelism

Data Parallelism

Pipeline Parallelism

Model Parallelism
Topics

1. Parallel databases (80’s - 90’s)

2. MapReduce (2004 - present)

3. Relational processing on MapReduce
Parallel Databases 101

- Rise of parallel databases: late 80’s
- Architecture: shared-nothing systems
  - A number of nodes connected by fast Ethernet switches
  - Inter-machine messages are the only way for communication
  - But used special-purpose hardware (costly, slow to evolve)
  - Small scale (hence did not focus on fault tolerance)
- Typical systems
  - Gamma: U. of Wisconsin Madison
  - TeraData: Wal-Mart’s 7.5TB sales data in hundreds of machines
  - Tandem
  - IBM / DB2
  - Informix…
Some Parallel (||) Terminology

- **Speed-Up**
  - Holds the problem size, grows the system
  - Reports serial time/parallel time
  - Ideally, linear

- **Scale-Up**
  - Grows both the system and the problem, reports running time
  - Ideally, constant
Some Parallel (||) Terminology

- **Speed-Up**
  - Holds the problem size, grows the system
  - Reports serial time/||-time
  - Ideally, linear

- **Utilization**
  - Speed-up / degree of ||-ism
  - Ideally, constant
Data Partitioning Schemes

Partitioning a table:

- **Range**
  - sequential scan
  - associative search
  - sorting
  - may have data skew

- **Hash**
  - sequential scan
  - equality search
  - equijoins if matching the hash attribute
  - range search, operations that do not match the hash attr.
  - can have data skew

- **Round Robin**
  - sequential scan
  - useless for other query operations
1) Parallel Scans & Index Lookups

- Scan in parallel, and merge.
- Selection may not require all sites given range or hash partitioning.
  - Want to restrict selection to a few nodes, or restrict “small” queries to a few nodes.
  - Indexes can be built at each partition.
  - What happens during data inserts and lookups?
    - Use the partitioning function to find relevant node(s)
    - Use the local index at each node to find matching records
2) Parallel Sorting

- Sort R by \( R.b \) while currently partitioned by \( R.a \)
  - Goal: evenly distribute the workload to \( n \) machines

- Idea:
  - Scan in parallel, and range-partition on \( R.b \) as we go.
  - As tuples come into each node, begin “local” sorting
  - Resulting data is sorted, and range-partitioned.
  - Problem: skew!
  - Solution: “sample” the data at start to determine partition boundaries.
Parallel Sort by R.b

Denote a range partitioning function as \( F \):

1. Run sampling to build an equi-depth histogram and set \( F \) for \( R.b \).

2. Partition all data by \( F \) on \( R.b \).

3. Local sort.

Data partitioned by \( R.a \).
3) Partitioned Join

• For equi-joins, *partition* the input relations by the join attribute \((R.b, S.b)\) on all nodes, and compute the join locally.

• Can use either *range partitioning* or *hash partitioning*, on the join attribute
  
  - \(R\) and \(S\) each are partitioned into \(n\) partitions, denoted as \(R_0, R_1, ..., R_{n-1}\) and \(S_0, S_1, ..., S_{n-1}\).
  
  - Partitions \(R_i\) and \(S_i\) are sent to node \(i\).
  
  - Each node locally computes the join using any method.
Topics

1. Parallel databases (80’s - 90’s)

2. MapReduce (2004 - present)

3. Relational processing on MapReduce
Motivation: Large Scale Data Processing

• Want to process lots of data, *unstructured or structured*
• Want to parallelize across *hundreds/thousands* of commodity computers
  - New definition of cluster computing: *large numbers of low-end processors working in parallel to solve a computing problem.*
  - Parallel DB: *a small number of high-end servers.*
• Want to make this easy for programmers
MapReduce

1) Clean abstraction for programmers
2) Automatic parallelization & distribution
3) Resource management
4) Fault-tolerance

1) Programming Model

• Borrows from functional programming
• Users implement an interface of only two functions:

  - map (in_key, in_value) ->
    list(out_key, intermediate_value)

  - reduce (out_key, list(intermediate_value) ->
    list(out_value)
Map function

- Input: a key-value pair. E.g.,
  - A line out of files (filename, line),
  - A row of a database (row_id, row),
  - A document (doc_name, document)

- map() produces one or more intermediate values along with an output key for each such value.

- map() is stateless: one input leaves no state that would affect the processing of the next input.
Reduce function

- After the map phase is over, all the intermediate values for a given output key are collected into a list.
- reduce() combines those intermediate values into one or more final values for that same output key.
- reduce() can be stateful: it operates on all the intermediate values of a certain key.
Example: Count Word Occurrences

map(String input_key, String input_value):
    // input_key: document name
    // input_value: document contents
    for each word w in input_value:
        EmitIntermediate(w, "1");

reduce(String output_key, Iterator intermediate_values):
    // output_key: a word
    // output_values: a list of counts
    int result = 0;
    for each v in intermediate_values:
        result += ParseInt(v);
    Emit(AsString(result));

How do we implement this using a relational DBMS? ETL (extract-transform-load) for customized data loading, then Group By.
Click Stream Analysis: Page Frequencies

Clicks\(\text{(time, url, referral\_url, user\_id, geo\_info...)}\)

```java
map(String tuple\_id, String tuple):
    EmitIntermediate(url, "1");

reduce(String url, Iterator list\_tuples):
    int result = 0;
    for each t in list\_tuples:
        result += ParseInt(t);
    Emit(AsString(result));
```

Select count(*)
From Clicks
Group By url;
2) Automatic Parallelism

- The map() function is stateless, so many instances can run in parallel on different splits (chunks) of input data.

- The reduce() function is stateful, but works on an output key at a time, so many copies can run in parallel on different keys (groups).

- **Barrier**: reduce phase can’t start until map phase is completely finished.
Data Parallelism in MapReduce

Input

Intermediate

System code

Group by Key

Grouped

UDF

Output
Sessionization in Click Stream Analysis

Input

Intermediate

Grouped

Physical partitioning

Logical partitioning: group (key, value) by key
Logical Partitioning in Hadoop

- Logical partitioning is based on group-by
- Common MapReduce implementations use parallel sort-merge
Optimization 1: Incremental Computation

- “Combiner” functions can be applied earlier, e.g., right after map() finishes on the same machine.
- Causes a mini-reduce phase to occur before the real reduce phase, to save bandwidth.
- Common examples: word frequency, url frequency.
- Also called partial aggregation.

Under which conditions is it sound to use a combiner?
When can we use combiner()?

- When reduce() is:
  - Commutative and Associative

- Any other viable condition?