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...

Web Server

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
- >4.5 billion msgs/day
- >15 TB click logs/day
- Stores >20 billion photos, and serves 1 million img/sec.
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*/||-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 for 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?
2) Parallel Sorting

• Sort R by R.b while currently partitioned by R.a

• Idea:
  - Scan in parallel, and range-partition as you 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

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

Denote a range partitioning function as \( F \):
3) Partitioned Join

- For **equi-joins**, *partition* the input relations by the join attribute 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.
Dataflow Network for Parallel Join

- Good use of split/merge makes it easier to build parallel versions of sequential join code.
4) Fragment-and-Replicate Join

- Partitioning not possible for non-equijoin conditions
  - E.g., R.A > S.B.
- Use the **fragment and replicate** technique
  - 1) Special case: only one relation is *partitioned*
    - R is partitioned; any partitioning technique can be used.
    - The other relation, S, is *replicated* across all the nodes.
    - Node i then locally computes the join of Ri with all of S using any join technique.
    - Works well when S is small.
  - 2) General case: both R and S are *partitioned*
    - Need to *replicate* all R partitions or all S partitions
    - Depicted on the next slide
Illustrating Fragment-and-Replicate Join

(a) Asymmetric fragment and replicate

(b) Fragment and replicate
5) Parallel Aggregates

• For each aggregate function, need a decomposition:
  – **Distributive**: \( \text{count}(S) = \sum \text{count}(s(i)) \), ditto for \( \text{sum}() \)
  – **Algebraic**: \( \text{avg}(S) = \left( \sum \text{sum}(s(i)) \right) / \sum \text{count}(s(i)) \)
  – **Holistic**: e.g., median, quantiles

• For group-by aggregation:
  – How would you implement parallel group by?
  – How do you add aggregation to each group?
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
MapReduce

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

MapReduce: Simplified Data Processing on Large Clusters.
Jeffrey Dean and Sanjay Ghemawat. OSDI 2004.
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

- 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 from the input.

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

• 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

\text{Clicks}(\text{time}, \text{url}, \text{referral_url}, \text{user_id}, \text{geo_info}...)

\text{map}(\text{String tuple_id, String tuple}):\n\hspace{1cm}\text{EmitIntermediate}(\text{url}, "1");

\text{reduce}(\text{String url, Iterator list_tuples}):\n\hspace{1cm}\text{int result} = 0;
\hspace{1cm}\text{for each t in list_tuples:}\n\hspace{1.5cm}\text{result} += \text{ParseInt(t)};
\hspace{1cm}\text{Emit}(\text{AsString(result)});

\text{Select count(*)}
\text{From Clicks}
\text{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

Grouped

Output

UDF

UDF

k1:v, k1:v, k2:v
k1:v
k3:v, k4:v
k4:v, k5:v
k4:v
k1:v, k3:v

k1:v, v, v, v
k2:v
k3:v, v
k4:v, v, v
k5:v

UDF

k1:v, k1:v, k2:v
k1:v
k3:v, k4:v
k4:v, k5:v
k4:v
k1:v, k3:v
Sessionization in Click Stream Analysis

Input:
- <u1,t1,p7>
- <u2,t2,p2>
- <u3,t3,p4>
- <u2,t4,p1>
- <u2,t5,p6>
- <u3,t6,p9>
- <u1,t7,p2>

Intermediate:
- u1:<t1,p7>
- u2:<t2,p2>
- u3:<t3,p4>
- u2:<t4,p1>
- u2:<t5,p6>
- u3:<t6,p9>
- u1:<t7,p2>

Grouped:
- u1:<t1,p7>,<t7,p2>
- u2:<t2,p2>,<t4,p1>,<t5,p6>
- u3:<t3,p4>,<t6,p9>

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?
Topics

1. Parallel databases (80’s - 90’s)

2. MapReduce (2004 - present)

3. Relational processing on MapReduce
Comparison to Parallel Databases

• Let us consider structured data here.
  – Of course, MapReduce can also handle text and image!

1. A closer look at internal implementation of MapReduce
   – Extract (key, value) using map()
   – Group data by key
   – Then apply reduce() to each group

2. Implementing relational operators using MapReduce
   – Parallel sorting?
   – Parallel Join?
   – Parallel group by-aggregation?

3. MapReduce query plans
Implementing Relational Operators

• Selection: R.a > “abc”
  - ParallelDB: if range partitioned, use a few nodes and indexes
  - MapReduce: scan all nodes, map() only.
  • Can be dominated by start-up cost. No indexes in the original impl.
Implementing Relational Operators

• Most other operators need *repartitioning* data:
  - **ParallelDB**: explicit partitioning function
  - **MapReduce**: more complicated
    (1) *Implicit* partitioning function, fn, controls data shuffling to reducers.
    (Default is *hash partitioning*. Can be changed to *range partitioning.*
    (2) Each reducer uses an additional mechanism to group data by the key.

  □ Consider the task to range partition data and sort data in each range.
  What is the key in the MR programming model?
Join Operators

- **Equijoin**: `R.a = S.a`
  - **ParallelDB**: hybrid hash join.
    - I/O and network costs?
  - **MapReduce**: the programming interface is not natural for joins.
    1. `map()` annotates tuples with ‘r’ and ‘s’,
    2. the system groups all data by the join attribute using sort-merge,
    3. `reduce()` joins ‘r’ and ‘s’ tuples with the same value of the join attribute.
    - *Spark later changed the programming model to make join more natural*
Join Operators

- Non equijoin: $R.a < S.a$
  - **ParallelDB**: fragment-replication
  - **MapReduce**: simulates fragment-replication. If replicate $S$,
    - replicate each $S$ tuple $m$ times in the mapper
    - tweak the partitioning function, $fn$, for shuffling so that these $m$ copies go to different reducers ($fn$ can be customized in Hadoop)
Group By Aggregation

- Scalar aggregate: `count()`, `sum()`
  - **ParallelDB**: partial aggregation + final aggregation
  - **MapReduce**: `map()` is empty; use `combiner()` for partial aggregation; use `reduce()` for final aggregation

- Group by aggregation: \( G_{R.a, \text{aggr}(R.b)} \)
  - **ParallelDB**: unary input version of (hybrid) hash join
  - **MapReduce**: 
    - `map()` simply emits tuples;
    - the system groups data by \( R.a \);
    - `reduce()` computes sum.
    - should use the `combiner()` for partial aggregation earlier.