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

1) Programming Model

- Borrows from functional programming
- Users implement an interface of **only two** functions:
  
  - \( \text{map} \ (\text{in\_key}, \text{in\_value}) \rightarrow \)  
    \[ \text{list(} \text{out\_key}, \text{intermediate\_value} \) \]

  - \( \text{reduce} \ (\text{out\_key}, \text{list(intermediate\_value)}) \rightarrow \)  
    \[ \text{list(} \text{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

Clicks(time, url, referral_url, user_id, geo_info...)

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 completed.
Data Parallelism in MapReduce

Input

UDF

Intermediate

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

System code

Group by Key

Grouped

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

UDF

Output
**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>

**Group by Key**
- **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
  - u1:<t1,p7>
  - u1:<t7,p2>
  - u2:<t2,p2>,<t4,p1>,<t5,p6>
  - u3:<t3,p4>,<t6,p9>
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?
Optimization 2: Hashing for Group-By

- Consider stored data first. Use independent hash functions $h_1$ (in map task), $h_2, h_3, \ldots$ (in reduce task)
MapReduce Hash (Hybrid-hash)

- Use independent hash functions $h_1$ (in map task), $h_2, h_3, \ldots$ (in reduce task) to avoid data skew

- If a bucket does not fit in memory, **recursively** partition it with $h_4, h_5, \ldots$
3) Yarn: a New Resource Manager
Yarn: a New Resource Manager

- **ResourceManager** has Scheduler and ApplicationsManager.

- **Scheduler:**
  - Scheduling based the resource requirements of the applications, using the notion of a resource **container** which incorporates elements such as memory, cpu, disk, network etc.
  - No monitoring or tracking of status for the application.
  - No guarantees about restarting failed tasks either due to application failure or hardware failures.
  - A pluggable policy plug-in for partitioning the cluster resources among the various queues, applications etc. E.g., current Map-Reduce schedulers such as the CapacityScheduler and the FairScheduler.

- **ApplicationsManager:**
  - accepts job-submissions, negotiates the first container for executing the application specific ApplicationMaster, and provides the service for restarting the ApplicationMaster container on failure.
Yarn: a New Resource Manager

• Per-application **ApplicationMaster**:  
  • negotiates appropriate resource **containers** from the Scheduler, tracks their status and monitoring for progress.

• **NodeManager**:  
  • per-machine framework agent responsible for containers, monitoring their resource usage (cpu, memory, disk, network), and reporting the same to the ResourceManager/Scheduler.
4) Fault Tolerance

• If an analytical task runs with \( n \) machines, and if every machine has probability \( p \) to fail in the lifetime of the task
• Then the probability that the task sees any failure in lifetime is

\[
1 - (1 - p)^n \longrightarrow 1
\]
Fault Tolerance

• Fine-grained fault tolerance: `materialize` map output onto local disk before the map task completes

• Master detects worker failures
  – Re-executes completed & in-progress `map()` tasks
  – Re-executes in-progress `reduce()` tasks

• Master notices particular input key/values cause crashes in `map()`, and skips those values on re-execution.
  – Effect: Can work around bugs in third-party libraries!
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: parallel (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, \( \text{fn} \), for shuffling so that these \( m \) copies go to different reducers (\( \text{fn} \) 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.
    - can use the `combiner()` for partial aggregation earlier.
MapReduce Query Plan: Q1

- How many rounds of map reduce jobs?
- In each round, what is the key, map(), reduce()? 

SELECT DISTINCT U.city 
FROM Users U, Clicks C 
WHERE U.uid=C.uid 
AND C.url LIKE '%%google%%';

Hints:
- Draw the relational plan
- Determine # MR jobs: every time the partitioning scheme changes, start a new MR job
- For each MapReduce job, specify the key, map(), reduce()
MapReduce Plan for Q1

- How many rounds of mapreduce jobs?
- In each round, what is in map(), what is in reduce()?

```sql
SELECT DISTINCT U.city
FROM Users U, Clicks C
WHERE U.uid=C.uid
AND C.url LIKE '%google%';
```

**Round 1:**
- **Key**: uid
- **Map**: (1) selection, (2) create ‘u’, ‘c’ tuples with labels
- **Reduce**: (1) join tuples within each group, (2) emit cities

**Round 2:**
- **Key**: city
- **Map**: emit
- **Reduce**: emit a tuple in each group