DISTRIBUTED ANALYTIC FRAMEWORKS

6.830 / 6.814 LECTURE 19
TIM KRASKA
RECAP: PARALLEL QUERY PROCESSING

Name 3 ways on how to parallelize a query
Three main ways to parallelize

1. Run multiple queries, each on a different thread
2. Run operators in different threads ("pipeline")
3. Partition data, process each partition in a different processor
Parallel and Distributed Databases:

Data is fragmented and replicated to different nodes (i.e., servers)

Fragmentation/Replication should be transparent for the user

Homogeneous vs. Heterogeneous

Master/Slave vs. P2P

Fragmented vs. Replicated
SCALEOUT

Speedup

#servers

Linear

Sub-linear
Scale-out and down in Cloud Computing = Elasticity

the ability of a system to adapt to changes in load
**Horizontal Partitioning:** Assign tuples of one table to individual partitions

**Vertical Partitioning:** Assigns columns of one table to individual partitions

**Goal:** Provide intra- and inter-query parallelism
HORIZONTAL PARTITIONING: METHODS

(a) Round-Robin

(b) Hashing

(c) Range
You have a data warehouse about sales data. The customer dimension table is mostly accessed using the customer id. What partitioning method would you use for the customer table?

a) **Round-robin**

b) **Hash**

c) **Range**
You have a data warehouse about sales data. The fact table (containing the orders) is mostly filtered by time? What partitioning method would you use?

a) Round-robin

b) Hash

c) Range
HORIZONTAL PARTITIONING: METHODS

Random / Round Robin
- Evenly distributes data (no skew)
- Requires us to repartition for joins
- No pruning

Hash partitioning
- Allows us to perform joins without repartitioning, when tables are partitioned on join attributes
- Only subject to skew when there are many duplicate values
- No pruning

Range partitioning
- Allows us to perform joins without repartitioning, when tables are partitioned on join attributes
- Data can be pruned
- Subject to skew
You have a data warehouse about sales data. The fact table (containing the orders) is mostly filtered by time but highly skewed (e.g., many more orders in December). What partitioning method would you use? Can you think of a way to deal with the skew?

a) Round-robin

b) Hash

c) Range
QUESTION

Can you think of a reason why it might NOT be a good idea to range partition on time?
EXAMPLE: RANGE-PARTITIONING

EXAMPLE: HASH-PARTITIONIERUNG

Partition table using hash-function: \( h(k) = k \% 4 \)

Fact table (orders)

Dimension (product)

\[ h(o\_cid) = o\_cid \% 4 \]

\[ h(c\_cid) = c\_cid \% 4 \]
PARTITIONING FOR STAR SCHEMA

D2: Date

D1: Payment

Lineitem

D3: Region

D4: Product
APPROACH: PARTITIONING AND REPLICATION IN STAR SCHEMA

1. Fact table and largest dimension table are co-partitioned
2. Other smaller dimension tables are replicated
Step 1: Co-partition fact and biggest dimension table

D2: Date

D1: Payment

D3: Region

Lineitem

Lineitem

Lineitem

Lineitem

D4: Product

D4: Product

D4: Product

D4: Product
Step 2: Partitions are allocated to nodes
**EXAMPLE: REPLICATION AND PARTITIONING**

**Step 3:** Replicate non-partitioned dimension tables (D1-D3)
EXAMPLE: QUERY PROCESSING IN PDBMS

SQL-Query

Compilation

Localization

Logical Plan

Logical Plan (distributed)

SELECT SUM(F.price), D3.region
FROM F, D3
WHERE F.d3 = D3.id
AND D3.country='Germany'
GROUP BY D3.region

\[ \gamma_{D3.region, \text{SUM}(F.price)} \]

\[ \sigma_{D3.country='Germany'} \]

\[ F.d3 = D3.id \]

F D3

...
EXAMPLE: QUERY PROCESSING IN PDBMS

SQL-Query

Compilation

Localization

Logical Plan

Node N₁

Naïve Logical Plan (parallel)

Logical Plan (distributed)
EXAMPLE: DISTRIBUTED QUERY PROCESSING (GLOBAL OPTIMIZATION)

Global Optimization: minimize network costs
SELECTIONS IN PDBMS

SELECT * FROM Emp WHERE salary > 1000

\( \sigma \)

(Helga, 2000)
(Hubert, 150)
...

\( \sigma \)

(Peter, 20)
(Rhadia, 15000)
...

\( \cup \)
SELECTIONS IN PDBMS

**Approach**

- Each server has a (horizontal) partition of the DB.
- Each server carries out selection (scan + filter) locally.
- Each server sends results to dedicated server.
- Dedicated server carries out union (U) for final result.

**Assessment**

- Scales almost linearly.
- Skew in communicating results may limit scalability.
RECAP: DISTRIBUTED JOIN STRATEGIES

If tables are partitioned properly, just run local joins

• Also, if one table is replicated, no need to join

Otherwise, several options:

• Collect all tables at one node
  • Inferior except in extreme cases, i.e., very small tables
• Re-partition one or both tables
  • Depending on initial partitioning
• Replicate (smaller) table on all nodes
JOIN: REPLICATE

SELECT * FROM R JOIN S ON R.b = S.c
JOIN: REPLICATE

Approach
Ship (all partitions of) Table 2 to all servers
carry out join of $P_i(T1)$ and T2 at each server
compute union of all local joins

Assessment
scales well if there is an efficient broadcast and Table 2 is small
even better if Table T2 is already replicated everywhere
SELECT * FROM A,B WHERE A.a = B.b

Suppose we have hashed A on a, using hash function F to get F(A.a) \( \rightarrow \) 1..n (n = # machines)

Also hash B on b using *same* F
REPARTITION

• Suppose A pre-partitioned on a, but B needs to be repartitioned
• Suppose A pre-partitioned on a, but B needs to be repartitioned

Generalizes to the case of repartitioning both tables
Suppose A pre-partitioned on a, but B needs to be repartitioned.

Each node sends and receives $|B|/n \times (n-1)/n$ bytes.
Replication vs Repartitioning

- Replication requires each node to send smaller table to all other nodes
  - $|T| / n \times n-1$ bytes sent by each node

- When would replication be preferred over repartitioning for joins?
  - If size of smaller table < data sent to repartition one or both tables
  - Should also account for cost of join, since will be higher with replicated table

- E.g., $|B| = 1$ MB, $|A| = 100$ MB, $n = 3$, need to repartition $A$
  - Data to repartition $A$ is $|A|/3 \times 2/3 = 22.22$ MB per node
    - Join $0.33$ MB to $33$ MB
  - Data to broadcast $B$ is $|B| = 1/3 \times 2 = 0.66$ MB
    - Join $1$ MB to $33$ MB
Aggregation

In general, each node will have data for the same groups

So merge will need to combine groups, e.g.:

- $\text{MAX (MAX1, MAX2)}$
- $\text{SUM (SUM1, SUM2)}$

What about average?
- Maintain SUMs and COUNTs, combine in merge step
Generalized Parallel Aggregates

• Express aggregates as 3 functions:
  • INIT – create partial aggregate value
  • MERGE – combine 2 partial aggregates
  • FINAL – compute final aggregate

• E.g., AVG:
  • INIT(tuple) → (SUM=tuple.value, COUNT=1)
  • MERGE (a1, a2) → (SUM=a1.SUM + a2.SUM, COUNT=a1.count+a2.count)
  • FINAL(a) → a.SUM/a.COUNT
What does MERGE do?

- For aggregate queries, receives partial aggregates from each processor, MERGEs and FINALizes them.

- For non-aggregates, just UNIONs results.
WHAT IF WE HAVE TOO MUCH DATA?

Internet Scale :

To analyze large datasets

Using nodes (i.e. servers) in a cluster in parallel
WHAT IS MAPREDUCE?

Simple programming model for data-intensive computing on large commodity clusters

Pioneered by Google

• Processes PB’s of data of per day (e.g., process user logs, web crawls, …)

Popularized by Apache Hadoop project (Yahoo)

• Used by Yahoo!, Facebook, Amazon, …

Many other MapReduce-based frameworks today
WHAT IS MAPREDUCE USED FOR?

• **At Google:**
  – Index building for Google Search
  – Analyzing search logs (Google Trends)
  – Article clustering for Google News
  – Statistical language translation

• **At Facebook:**
  – Data mining
  – Ad optimization
  – Machine learning (e.g., spam detection)
EXAMPLE: GOOGLE TRENDS

https://www.google.com/trends/
WHAT ELSE IS MAPREDUCE USED FOR?

In research:

• Topic distribution in Wikipedia (PARC)
• Natural language processing (CMU)
• Climate simulation (UW)
• Bioinformatics (Maryland)
• Particle physics (Nebraska)
• <Your application here>
MAPREDUCE GOALS

• **Scalability to large data volumes:**
  – Scan 100 TB on 1 node @ 50 MB/s = 24 days
  – Scan on 1000-node cluster = 35 minutes

• **Cost-efficiency:**
  – Commodity nodes (cheap, but unreliable)
  – Commodity network (low bandwidth)
  – Automatic fault-tolerance (fewer admins)
  – Easy to use (fewer programmers)
ARCHITECTURE & PROGRAMMING MODEL

MAPREDUCE & HADOOP
HADOOP 1.0: ARCHITECTURE

Open-Source MapReduce System:

- **High-level Programming**
  - Hive ("SQL")
  - Pig (Data-Flow)
  - Others (Mahout, Giraph)

- **Distributed Execution**
  - MapReduce (Programming Model and Runtime System)

- **Distributed Storage**
  - Hadoop Distributed Filesystem (HDFS)
MAIN HADOOP COMPONENTS

**HDFS = Hadoop Distributed File System**

- Single namespace for entire cluster
- Replicates data 3x for fault-tolerance

**MapReduce framework**

- Runs jobs submitted by users
- Manages work distribution & fault-tolerance
- Co-locates work with file system
TYPICAL HADOOP CLUSTER

- 40 nodes/rack, 1000-4000 nodes in cluster
- 1 Gbps bandwidth in rack, 8 Gbps out of rack
- Node specs (Facebook): 8-16 cores, 32 GB RAM, 8×1.5 TB disks, no RAID
TYPICAL HADOOP CLUSTER
CHALLENGES OF COMMODITY CLUSTERS

Cheap nodes fail, especially when you have many

• Mean time between failures for 1 node = 3 years
• MTBF for 1000 nodes = 1 day
• Solution: Build fault tolerance into system

Commodity network = low bandwidth

• Solution: Push computation to the data

Programming in a cluster is hard

• Parallel programming is hard
• Distributed parallel programming is even harder
• Solution: Restricted programming model: Users write data-parallel “map” and “reduce” functions, system handles work distribution and failures
HADOOP DISTRIBUTED FILE SYSTEM

- Files split into **64 MB blocks**
- Blocks **replicated** across several datanodes (often 3)
- **Namenode** stores metadata (file names, locations, etc)
- Optimized for **large files**, sequential reads
- Files are **append-only**
Data type: key-value records

Map function:

\[(K_{in}, V_{in}) \rightarrow \text{list}(K_{inter}, V_{inter})\]

Reduce function:

\[(K_{inter}, \text{list}(V_{inter})) \rightarrow \text{list}(K_{out}, V_{out})\]
def mapper(line):
    foreach word in line.split():
        emit(word, 1)

def reducer(key, values): //values={1,1,1,...}
    emit(key, count(values))
WORD COUNT EXECUTION

Input
- the quick brown fox
- the fox ate the mouse
- how now brown cow

Map
- brown, 2
- fox, 2
- how, 1
- now, 1
- the, 3

Shuffle & Sort
- the, 1
- brown, 1
- fox, 1
- the, 1
- ate, 1
- cow, 1
- mouse, 1
- quick, 1

Reduce
- brown, {1,1}
- fox, {1,1}
- ...
**AN OPTIMIZATION: THE COMBINER**

Local reduce function for repeated keys produced by same map
- For associative operations like sum, count, max
- Decreases amount of intermediate data

**Example:** local counting for Word Count:

```python
def combiner(key, values):
    output(key, sum(values))

def reducer(key, values):
    output(key, sum(values))
```
**WORD COUNT WITH COMBINER**

**Input**
- the quick brown fox
- the fox ate the mouse
- how now brown cow

**Output**
- brown, 2
- fox, 2
- how, 1
- now, 1
- the, 3

**Map**
- the, 1
- brown, 1
- fox, 1

**Shuffle & Sort**
- the, 2
- fox, 1

**Reduce**
- the, {2,1}
- ...

**Output**
- at, 1
- cow, 1
- mouse, 1
- quick, 1
You want to implement the dot product $a_1b_1 + a_2b_2 + \ldots + a_nb_n$ of two large vectors $a=[a_1, a_2, \ldots, a_n]$ and $b=[b_1, b_2, \ldots, b_n]$. The input to the mapper is a pair $(a_i,b_i)$.

Which of the following implementations is correct?

**A)**
```python
def mapper(a,b):
    emit(a*b, 1)

def reducer(key, values):
    emit(key, sum(values))
```

**B)**
```python
def mapper(a,b):
    emit(1, a+b)

def reducer(key, values):
    emit(key, sum(values))
```

**C)**
```python
def mapper(a,b):
    emit(1, a*b)

def reducer(key, values):
    emit(key, mult(values))
```

**D)**
```python
def mapper(a,b):
    emit(1, a*b)

def reducer(key, values):
    emit(key, sum(values))
```
MAPREDUCE EXECUTION DETAILS

Mappers preferentially scheduled on same node or same rack as their input block
  • Minimize network use to improve performance

Mappers save outputs to local disk before serving to reducers
  • Allows recovery if a reducer crashes

Reducers save outputs to HDFS
1. If a task crashes:
   - Retry on another node
     - OK for a map because it had no dependencies
     - OK for reduce because map outputs are on disk
   - If the same task repeatedly fails, fail the job or ignore that input block

➢ Note: For the fault tolerance to work, user tasks must be deterministic and side-effect-free
2. If a node crashes:
   - Relaunch its current tasks on other nodes
   - Relaunch also any maps the node previously ran
     - Necessary because their output files were lost along with the crashed node
3. If a task is going slowly (straggler):
   • Launch second copy of task on another node
   • Take the output of whichever copy finishes first, and kill the other one

Critical for performance in large clusters (many possible causes of stragglers)
TAKEAWAYS

By providing a restricted data-parallel programming model, MapReduce can control job execution in useful ways:

• Automatic division of job into tasks
• Placement of computation near data
• Load balancing
• Recovery from failures & stragglers
SAMPLE APPLICATIONS
INVERTED INDEX

Input: (filename, text) records
Output: list of files containing each word

Map:

    foreach word in text.split():
        output(word, filename)

Combine: uniquify filenames for each word

Reduce:

    def reduce(word, filenames):
        output(word, sort(filenames))
INVERTED INDEX EXAMPLE

- **hamlet.txt**
  - to be or not to be
  - be, hamlet.txt
  - or, hamlet.txt
  - not, hamlet.txt

- **12th.txt**
  - be not afraid of greatness
  - afraid, 12th.txt
  - of, 12th.txt
  - greatness, 12th.txt

- **afraid, (12th.txt)**
- **be, (12th.txt, hamlet.txt)**
- **not, (12th.txt, hamlet.txt)**
- **of, (12th.txt)**
- **or, (hamlet.txt)**
- **to, (hamlet.txt)**
IN CLASS TASK

1. Sort all words of all documents alphabetically

2. Return the most popular words of all documents
SORT

Input: (key, value) records
Output: same records, sorted by key

Map: identity function
Reduce: identify function

Trick: Pick partitioning function $p$ such that $k_1 < k_2 \Rightarrow p(k_1) < p(k_2)$
MOST POPULAR WORDS

Input: (filename, text) records
Output: the 100 words occurring in most files

Two-stage solution:

- **MapReduce Job 1:**
  Create inverted index, giving (word, list(file)) records

- **MapReduce Job 2:**
  Map each (word, list(file)) to (count, word)
  Sort these records by count as in sort job
HADOOP APIS
INTRODUCTION TO HADOOP

Download from hadoop.apache.org

To install locally, unzip and set JAVA_HOME

Docs: hadoop.apache.org/common/docs/current

Three ways to write jobs:

• Java API
• Hadoop Streaming (for Python, Perl, etc.): Uses std.in and out
• Pipes API (C++)
public static class MapClass extends MapReduceBase
    implements Mapper<LongWritable, Text, Text, IntWritable> {

    private final static IntWritable ONE = new IntWritable(1);

    public void map(LongWritable key, Text value,
                    OutputCollector<Text, IntWritable> output,
                    Reporter reporter) throws IOException {
        String line = value.toString();
        StringTokenizer itr = new StringTokenizer(line);
        while (itr.hasMoreTokens()) {
            output.collect(new Text(itr.nextToken()), ONE);
        }
    }
}
public static class Reduce extends MapReduceBase
    implements Reducer<Text, IntWritable, Text, IntWritable> {

    public void reduce(Text key, Iterator<IntWritable> values,
            OutputCollector<Text, IntWritable> output,
            Reporter reporter) throws IOException {

        int sum = 0;
        while (values.hasNext()) {
            sum += values.next().get();
        }
        output.collect(key, new IntWritable(sum));
    }
}
public static void main(String[] args) throws Exception {
    JobConf conf = new JobConf(WordCount.class);
    conf.setJobName("wordcount");

    conf.setMapperClass(MapClass.class);
    conf.setCombinerClass(Reduce.class);
    conf.setReducerClass(Reduce.class);

    FileInputFormat.setInputPaths(conf, args[0]);
    FileOutputFormat.setOutputPath(conf, new Path(args[1]));

    conf.setOutputKeyClass(Text.class); // out keys are words (strings)
    conf.setOutputValueClass(IntWritable.class); // values are counts

    JobClient.runJob(conf);
}
HADOOP STREAMING

Mapper.py:

```python
import sys

for line in sys.stdin:
    for word in line.split():
        print(word.lower() + "\t" + 1)
```

Reducer.py:

```python
import sys

counts = {}

for line in sys.stdin:
    word, count = line.split("\t")
    dict[word] = dict.get(word, 0) + int(count)

for word, count in counts:
    print(word.lower() + "\t" + 1)
```
PIG & HIVE
MOTIVATION

MapReduce is powerful: many algorithms can be expressed as a series of MR jobs

But it’s fairly low-level: must think about keys, values, partitioning, etc.

Can we capture common “job patterns”? 
PIG

Started at Yahoo! Research

Features:

• Expresses sequences of MapReduce jobs
• Data model: nested “bags” of items
• Provides relational (SQL) operators (JOIN, GROUP BY, etc)
• Easy to plug in Java functions
Suppose you have user data in one file, website data in another, and you need to find the top 5 most visited pages by users aged 18-25.
IN MAPREDUCE

Example from http://wiki.apache.org/pig-data/attachments/PigTalksPapers/attachments/ApacheConEurope09.ppt
IN PIG LATIN

Users = load ‘users’ as (name, age);
Filtered = filter Users by
           age >= 18 and age <= 25;
Pages = load ‘pages’ as (user, url);
Joined = join Filtered by name, Pages by user;
Grouped = group Joined by url;
Summed = foreach Grouped generate group,
         count(Joined) as clicks;
Sorted = order Summed by clicks desc;
Top5 = limit Sorted 5;

store Top5 into ‘top5sites’;

Example from http://wiki.apache.org/pig-data/attachments/PigTalksPapers/attachments/ApacheConEurope09.ppt
Notice how naturally the components of the job translate into Pig Latin.

Example from http://wiki.apache.org/pig-data/attachments/PigTalksPapers/attachments/ApacheConEurope09.ppt
HIVE

Developed at Facebook

Used for most Facebook jobs

SQL-like interface built on Hadoop

- Maintains table schemas
- SQL-like query language (which can also call Hadoop Streaming scripts)
- Supports table partitioning, complex data types, sampling, some query optimization
MapReduce’s data-parallel programming model hides complexity of distribution and fault tolerance

Principal philosophies:

- *Make it scale*, so you can throw hardware at problems
- *Make it cheap*, saving hardware, programmer and administration costs (but necessitating fault tolerance)

Hive and Pig further simplify programming

MapReduce is not suitable for all problems, but when it works, it may save you a lot of time
OTHER SYSTEMS

More general execution engines

• **Dryad** (Microsoft): general task DAG
• **Pregel** (Google): in-memory iterative graph algs.
• **Spark** (Berkeley): general in-memory computing

Language-integrated interfaces

• Run computations directly from host language
• **DryadLINQ** (MS), **FlumeJava** (Google), **Spark**
MapReduce simplified “big data” analysis on large, unreliable clusters

But as soon as organizations started using it widely, users wanted more:

• More complex, multi-stage applications
• More interactive queries
• More low-latency online processing
SPARK MOTIVATION

Complex jobs, interactive queries and online processing all need one thing that MR lacks:

Efficient primitives for data sharing

Iterative job

Interactive mining

Stream processing
EXAMPLES

Input

HDFS read → iter. 1 → HDFS read → iter. 2 → ...

HDFS write

query 1 → result 1
query 2 → result 2
query 3 → result 3

...
**EXAMPLES**

**Problem:** in MR, only way to share data across jobs is stable storage (e.g. file system) -> slow!
GOAL: IN-MEMORY DATA SHARING

10-100× faster than network and disk
SOLUTION: RESILIENT DISTRIBUTED DATASETS (RDDs)

Partitioned collections of records that can be stored in memory across the cluster

Manipulated through a diverse set of transformations (map, filter, join, etc)

Fault recovery without costly replication

- Remember the series of transformations that built an RDD (its lineage) to recompute lost data
EXAMPLE: LOG MINING

Load error messages from a log into memory, then interactively search for various patterns

```scala
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split(\t')(2))
messages.cache()

messages.filter(_.contains("foo")).count
messages.filter(_.contains("bar")).count
...

Result: scaled to 1 TB data in 5-7 sec (vs 170 sec for on-disk data)
```
Fault Tolerance

```
lines
  filter(_.startsWith("ERROR"))
  errors
    filter(_.contains("HDFS"))
    HDFS errors
      map(_.split("\t")(3))
      time fields
```
Scheduling Stages

Stage 1
- A
- B: groupBy

Stage 2
- C: map
- D
- E: union

Stage 3
- F: join
- G

Not Cached RDD
Cached RDD
EXAMPLE: LINEAR REGRESSION

Find best line separating two sets of points

random initial line

target
val data = spark.textFile(...).map(readPoint).cache()

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
    val gradient = data.map(p =>
        (1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y * p.x
    ).reduce(_ + _)
    w -= gradient
}

println("Final w: " + w)
LINEAR REGRESSION PERFORMANCE

127 s / iteration

first iteration 174 s
further iterations 6 s
OTHER PROJECTS

Hive on Spark (SparkSQL): SQL engine

Spark Streaming: incremental processing with in-memory state

MLLib: Machine learning library

GraphX: Graph processing on top of Spark
OTHER RESOURCES

Hadoop: http://hadoop.apache.org/common
Pig: http://hadoop.apache.org/pig
Hive: http://hadoop.apache.org/hive
Spark: http://spark-project.org

Hadoop video tutorials: www.cloudera.com/hadoop-training

Amazon Elastic MapReduce: http://aws.amazon.com/elasticmapreduce/