Announcements

• HW 6 releases tonight
  – Due Nov. 20th
  – Waiting for AWS credit can take up to two days
  – Sign up early:
• Extended office hours Friday to help with first parts of HW 6
  – 11:30 to 5:00pm in CSE 023

Class Overview

• Unit 1: Intro
• Unit 2: Relational Data Models and Query Languages
• Unit 3: Non-relational data
• Unit 4: RDBMS internals and query optimization
  – Unit 5: Parallel query processing
  – Spark and Hadoop
• Unit 6: DBMS usability, conceptual design
• Unit 7: Transactions
• Unit 8: Advanced topics (time permitting)

Why compute in parallel?

• Multi-cores:
  – Most processors have multiple cores
  – This trend will likely increase in the future
• Big data: too large to fit in main memory
  – Distributed query processing on 100x-1000x servers
  – Widely available now using cloud services
  – Recall HW3

Performance Metrics for Parallel DBMSs

Nodes = processors, computers

• Speedup:
  – More nodes, same data ➔ higher speed
• Scaleup:
  – More nodes, more data ➔ same speed

Linear v.s. Non-linear Speedup

# nodes (N)

+1
+5
+10
+15

Speedup
Linear v.s. Non-linear Scaleup

Why Sub-linear Speedup and Scaleup?

- **Startup cost**
  - Cost of starting an operation on many nodes

- **Interference**
  - Contention for resources between nodes

- **Skew**
  - Slowest node becomes the bottleneck

Architectures for Parallel Databases

- Shared memory
- Shared disk
- Shared nothing

Shared Memory

- Nodes share both RAM and disk
- Dozens to hundreds of processors
  - Example: SQL Server runs on a single machine and can leverage many threads to speed up a query
  - Easy to use and program
  - Expensive to scale
    - Last remaining cash cows in the hardware industry

Shared Disk

- All nodes access the same disks
- Found in the largest "single-box" (non-cluster) multiprocessors
  - Example: Oracle
  - No need to worry about shared memory
  - Hard to scale: existing deployments typically have fewer than 10 machines

Shared Nothing

- Cluster of commodity machines on high-speed network
  - Called "clusters" or "blade servers"
  - Each machine has its own memory and disk: lowest contention.
  - Example: Google
  - Easy to maintain and scale
  - Most difficult to administer and tune.

We discuss only Shared Nothing in class
Approaches to Parallel Query Evaluation

- **Inter-query parallelism**
  - Transaction per node
  - Good for transactional workloads

- **Inter-operator parallelism**
  - Operator per node
  - Good for analytical workloads

- **Intra-operator parallelism**
  - Operator on multiple nodes
  - Good for both?

We study only intra-operator parallelism: most scalable

Single Node Query Processing (Review)

- **Selection**: \( \sigma A=123(R) \)
  - Scan file R, select records with A=123

- **Group-by**: \( \gamma A, \text{sum}(B)(R) \)
  - Scan file R, insert into a hash table using A as key
  - When a new key is equal to an existing one, add B to the value

- **Join**: \( R \bowtie S \)
  - Scan file S, insert into a hash table using B as key
  - Scan file R, probe the hash table using B

Distributed Query Processing

- Data is horizontally partitioned on many servers
- Operators may require data reshuffling
- First let’s discuss how to distribute data across multiple nodes / servers

Horizontal Data Partitioning

- **Block Partition**:
  - Partition tuples arbitrarily s.t. \( \text{size}(R_1)=...=\text{size}(R_P) \)

- **Hash partitioned on attribute A**:
  - Tuple \( t \) goes to chunk \( i \), where \( i = h(t.A) \mod P + 1 \)
  - Recall: calling hash fn’s is free in this class

- **Range partitioned on attribute A**:
  - Partition the range of A into \( \infty = v_0 < v_1 < ... < v_P = \infty \)
  - Tuple \( t \) goes to chunk \( i \), if \( v_{i-1} < t.A < v_i \)
Uniform Data v.s. Skewed Data

• Let R(K,A,B,C): which of the following partition methods may result in skewed partitions?
  - Block partition
  - Hash-partition
    - On the key K
    - On the attribute A

Keep this in mind in the next few slides

Parallel Execution of RA Operators: Grouping

Data: R(K,A,B,C)
Query: \( \gamma_A, \text{sum}(C)(R) \)

How to compute group by if:

- R is hash-partitioned on A?
- R is block-partitioned?
- R is hash-partitioned on K?

Speedup and Scaleup

- Consider:
  - Query: \( \gamma_A, \text{sum}(C)(R) \)
  - Runtime: only consider I/O costs
  - If we double the number of nodes P, what is the new running time?
    - Half (each server holds \( \frac{1}{2} \) as many chunks)
  - If we double both P and the size of R, what is the new running time?
    - Same (each server holds the same # of chunks)

But only if the data is without skew!

Approaches to Parallel Query Evaluation

- Inter-query parallelism
  - One query per node
  - Good for transactional (OLTP) workloads

- Inter-operator parallelism
  - Operator per node
  - Good for analytical (OLAP) workloads

- Intra-operator parallelism
  - Operator on multiple nodes
  - Good for both?

We study only intra-operator parallelism: most scalable

Skewed Data

- R(K,A,B,C)
- Informally: we say that the data is skewed if one server holds much more data than the average
- E.g. we hash-partition on A, and some value of A occurs very many times ("Justin Bieber")
- Then the server holding that value will be skewed
Parallel Data Processing in the 20\textsuperscript{th} Century

\section*{Parallel Execution of RA Operators: Partitioned Hash-Join}

- \textbf{Data:} R(\textit{K}_1, \textit{A}, \textit{B}), S(\textit{K}_2, \textit{B}, \textit{C})
- \textbf{Query:} R(\textit{K}_1, \textit{A}, \textit{B}) \bowtie S(\textit{K}_2, \textit{B}, \textit{C})

Initially, both R and S are partitioned on \textit{K}_1 and \textit{K}_2.

Each server computes the join locally.

\section*{Optional Reading}

- Original paper: https://www.usenix.org/legacy/events/osdi04/tech/dean.html
- Rebuttal to a comparison with parallel DBs: http://dl.acm.org/citation.cfm?doid=1629175.1629198
- Chapter 2 (Sections 1, 2, 3 only) of Mining of Massive Datasets, by Rajaraman and Ullman
  http://i.stanford.edu/~ullman/mmds.html

\section*{Parallel Join Illustration}

- \textbf{Data:} R(\textit{K}_1, \textit{A}, \textit{B}), S(\textit{K}_2, \textit{B}, \textit{C})
- \textbf{Query:} R(\textit{K}_1, \textit{A}, \textit{B}) \bowtie S(\textit{K}_2, \textit{B}, \textit{C})

- Reshuffle R on R.\textit{B} and S on S.\textit{B}

- Local Join

- Broadcast Join

- Why would you want to do this?

\section*{Parallel Data Processing @ 2000}

- Google
Motivation

• We learned how to parallelize relational database systems
• While useful, it might incur too much overhead if our query plans consist of simple operations
• MapReduce is a programming model for such computation
• First, let’s study how data is stored in such systems

Distributed File System (DFS)

• For very large files: TBs, PBs
• Each file is partitioned into chunks, typically 64MB
• Each chunk is replicated several times (≥3), on different racks, for fault tolerance
• Implementations:
  – Google’s DFS: GFS, proprietary
  – Hadoop’s DFS: HDFS, open source

MapReduce

• Google: paper published 2004
• Free variant: Hadoop
• MapReduce = high-level programming model and implementation for large-scale parallel data processing

Typical Problems Solved by MR

• Read a lot of data
• Map: extract something you care about from each record
• Shuffle and Sort
• Reduce: aggregate, summarize, filter, transform
• Write the results

Data Model

Files!

A file = a bag of (key, value) pairs
Sounds familiar after HW5?

A MapReduce program:
• Input: a bag of (inputkey, value) pairs
• Output: a bag of (outputkey, value) pairs
  – outputkey is optional

Step 1: the MAP Phase

User provides the MAP-function:
• Input: (input key, value)
• Output: bag of (intermediate key, value)

System applies the map function in parallel to all (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
- Each Document
  - The key = document id (did)
  - The value = set of words (word)

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

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

```
MAP

<table>
<thead>
<tr>
<th>did1, v1</th>
<th>did2, v2</th>
<th>did3, v3</th>
<th>. . .</th>
</tr>
</thead>
</table>

REDUCE

Shuffle

Mappers write file to local disk
Reducers read the files (=reshuffling); if the server fails, the reduce task is restarted on another server.

Fault Tolerance

- If one server fails once every year...
  ... then a job with 10,000 servers will fail in less than one hour

- MapReduce handles fault tolerance by writing intermediate files to disk:
  - Mappers write file to local disk
  - Reducers read the files (=reshuffling); if the server fails, the reduce task is restarted on another server

Workers

- A worker is a process that executes one task at a time
- Typically there is one worker per processor, hence 4 or 8 per node
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

Interesting Implementation Details

Backup tasks:
• Straggler = a machine that takes unusually long time to complete one of the last tasks. E.g.:
  – 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

Straggler Example

Using MapReduce in Practice:
Implementing RA Operators in MR

Relational Operators in MapReduce

Given relations \( R(A,B) \) and \( S(B,C) \) compute:

• Selection: \( \sigma_{A=123}(R) \)
• Group-by: \( \gamma_{A,sum(B)}(R) \)
• Join: \( R \bowtie S \)
Selection \( \sigma_{A=123}(R) \)

map(Tuple t):
- if \( t.A = 123 \):
  - EmitIntermediate(\( t.A, t \));

reduce(String A, Iterator values):
- No need for reduce.

But need system hacking in Hadoop to remove reduce from MapReduce

Group By \( \gamma_{A, \text{sum}(B)}(R) \)

map(Tuple t):
- EmitIntermediate(\( t.A, t.B \));

reduce(String A, Iterator values):
- for each \( v \) in values:
  - Emit(v);

\[ (23, [t_1]) \]
\[ (42, [t_4]) \]
\[ (123, [t_2, t_3]) \]

Join

Two simple parallel join algorithms:
- Partitioned hash-join (we saw it, will recap)
- Broadcast join

Partitioned Hash-Join

Initially, both \( R \) and \( S \) are horizontally partitioned

\( R(A,B) \bowtie_{B=C} S(C,D) \)

map(Tuple t):
- case \( t.\text{relationName} \) of
  - 'R': EmitIntermediate(\( t.B, ('R', t) \));
  - 'S': EmitIntermediate(\( t.C, ('S', t) \));

reduce(String k, Iterator values):
- for each \( v \) in values:
  - R.insert(\( v \))
  - S.insert(\( v \));

for \( v1 \) in R, for \( v2 \) in S
  Emit(\( v1,v2 \));

Broadcast Join

\( R(A,B) \bowtie_{B=C} S(C,D) \)

R1, R2, R3, R'1, R'2, R'3
S1, S2, S3

Broadcast S.
Broadcast Join

\[ R(A,B) \bowtie_{B=C} S(C,D) \]

**map(String value):**
- readFromNetwork(S); /* over the network */
- hashTable = new HashTable()
  - for each w in S:
    - hashTable.insert(w.C, w)
- for each v in value:
  - for each w in hashTable.find(v.B)
  - Emit(v,w); /* empty: map-side only */

**reduce(…):**

HW6

- HW6 will ask you to write SQL queries and MapReduce tasks using Spark
- You will get to “implement” SQL using MapReduce tasks
  - Can you beat Spark’s implementation?

Spark

A Case Study of the MapReduce Programming Paradigm

Issues with MapReduce

- Difficult to write more complex queries
- Need multiple MapReduce jobs: dramatically slows down because it writes all results to disk

Spark

- Open source system from UC Berkeley
- Distributed processing over HDFS
- Differences from MapReduce:
  - Multiple steps, including iterations
  - Stores intermediate results in main memory
  - Closer to relational algebra (familiar to you)
- Details:
  [http://spark.apache.org/examples.html](http://spark.apache.org/examples.html)
Spark

• Spark supports interfaces in Java, Scala, and Python
  – Scala: extension of Java with functions/closures
• We will illustrate use the Spark Java interface in this class
• Spark also supports a SQL interface (SparkSQL), and compiles SQL to its native Java interface

Resilient Distributed Datasets

• RDD = Resilient Distributed Datasets
  – A distributed, immutable relation, together with its lineage
  – Lineage = expression that says how that relation was computed = a relational algebra plan
• Spark stores intermediate results as RDD
• If a server crashes, its RDD in main memory is lost. However, the driver (=master node) knows the lineage, and will simply recompute the lost partition of the RDD

Programming in Spark

• A Spark program consists of:
  – Transformations (map, reduce, join...). Lazy
  – Actions (count, reduce, save...). Eager

• Eager: operators are executed immediately
• Lazy: operators are not executed immediately
  – A operator tree is constructed in memory instead
  – Similar to a relational algebra tree

The RDD Interface

Collections in Spark

• RDD<T> = an RDD collection of type T
  – Partitioned, recoverable (through lineage), not nested
• Seq<T> = a sequence
  – Local to a server, may be nested

Example

Given a large log file hdfs://logfile.log retrieve all lines that:
• Start with “ERROR”
• Contain the string “sqlite”

```java
s = SparkSession.builder().getOrCreate();
lines = s.read().textFile("hdfs://logfile.log");
errors = lines.filter(l -> l.startsWith("ERROR"));
sqlerrors = errors.filter(l -> l.contains("sqlite"));
sqlerrors.collect();
```
Example
Given a large log file `hdfs://logfile.log`
retrieve all lines that:
• Start with “ERROR”
• Contain the string “sqlite”

```java
s = SparkSession.builder()...getOrCreate();
lines = s.read().textFile("hdfs://logfile.log");
errors = lines.filter(l -> l.startsWith("ERROR"));
sqlerrors = errors.filter(l -> l.contains("sqlite"));
sqlerrors.collect();
```

**Example**
Given a large log file `hdfs://logfile.log`
retrieve all lines that:
• Start with “ERROR”
• Contain the string “sqlite”

```java
s = SparkSession.builder()...getOrCreate();
lines = s.read().textFile("hdfs://logfile.log");
errors = lines.filter(l -> l.startsWith("ERROR"));
sqlerrors = errors.filter(l -> l.contains("sqlite"));
sqlerrors.collect();
```

**Example**
Recall: anonymous functions
(lambda expressions) starting in Java 8

```java
errors = lines.filter(l -> l.startsWith("ERROR"));
```

is the same as:

```java
class FilterFn implements Function<Row, Boolean>{
    Boolean call (Row r) {
        return l.startsWith("ERROR");
    }
}
errors = lines.filter(new FilterFn());
```

**Example**
Given a large log file `hdfs://logfile.log`
retrieve all lines that:
• Start with “ERROR”
• Contain the string “sqlite”

```java
s = SparkSession.builder()...getOrCreate();
lines = s.read().textFile("hdfs://logfile.log");
errors = lines.filter(l -> l.startsWith("ERROR"));
sqlerrors = errors.filter(l -> l.contains("sqlite"));
sqlerrors.collect();
```

**MapsReduce Again…**
Steps in Spark resemble MapReduce:
• `col.filter(p)` applies in parallel the predicate `p` to all elements `x` of the partitioned collection, and returns collection with those `x` where `p(x) = true`
• `col.map(f)` applies in parallel the function `f` to all elements `x` of the partitioned collection, and returns a new partitioned collection

**Persistence**
If any server fails before the end, then Spark must restart
If any server fails before the end, then Spark must restart

Spark can recompute the result from errors

A Spark/Scala program consists of:
- Transformations (map, reduce, join...). Lazy
- Actions (count, reduce, save...). Eager

RDD<T> = an RDD collection of type T
Persistence
RDD:

Example

Recap: Programming in Spark
Transformations:
- `map(f : T -> U): RDD<T> -> RDD<U>`
- `flatMap(f: T -> Seq(U)): RDD<T> -> RDD<U>`
- `filter(f: T -> Bool): RDD<T> -> RDD<T>`
- `groupByKey(): RDD<(K,V)> -> RDD<(K,Seq[V])>`
- `reduceByKey(F:(V,V) -> V): RDD<(K,V)> -> RDD<(K,V)>`
- `union(): (RDD<T>,RDD<T>) -> RDD<T>`
- `join(): (RDD<(K,V)>,RDD<(K,W)>) -> RDD<(K,(V,W))>`
- `cogroup(): (RDD<(K,V)>,RDD<(K,W)>) -> RDD<(K,((Seq<V>,Seq<W>))>`
- `crossProduct(): (RDD<T>,RDD<U>) -> RDD<(T,U)>`

Actions:
- `count(): RDD<T> -> Long`
- `collect(): RDD<T> -> Seq<T>`
- `reduce(f:(T,T) -> T): RDD<T> -> T`
- `save(path: String): Outputs RDD to a storage system e.g., HDFS`

DataFrames
- Like RDD, also an immutable distributed collection of data
- Organized into named columns rather than individual objects
  - Just like a relation
  - Elements are untyped objects called Row's
- Similar API as RDDs with additional methods
  - `people = spark.read().textFile(…);`
  - `ageCol = people.col("age");`
  - `ageCol.plus(10); // creates a new DataFrame`

Datasets
- Similar to DataFrames, except that elements must be typed objects
  - E.g.: `Dataset<People>` rather than `Dataset<Row>`
  - Can detect errors during compilation time
  - DataFrames are aliased as `Dataset<Row>` (as of Spark 2.0)
  - You will use both Datasets and RDD APIs in HW6

Datasets API: Sample Methods
- Functional API
  - `agg(Column expr, Column… exprs)`
  - `groupBy(String col, String… cols)`
  - `reduceBy(Column… cols)`
  - `orderBy(Column… sortExprs)`
  - `select(Column… cols)`
- "SQL" API
  - `SparkSession.sql("select * from R");`
- Look familiar?

Conclusions
- Parallel databases
  - Predefined relational operators
  - Optimization
  - Transactions
- MapReduce
  - User-defined map and reduce functions
  - Must implement/optimize manually relational ops
  - No updates/transactions
- Spark
  - Predefined relational operators
  - Must optimize manually
  - No updates/transactions