What’s the relationship between databases and MapReduce? Can we combine their strengths?

DBMS Overview

- Some material obtained from Ramakrishnan/Gehrke book
- Relational databases have been around since the 1970s
- Parallel DBMS actively researched since 1980s
- Highly successful and ubiquitous
  - Relational technology also found in data warehousing
- Declarative programming
  - Specify WHAT you want, not HOW to get it
  - Optimizer finds efficient query plan
- Data independence
  - Write queries against logical schema
  - Create views to create illusion of different logical schema
- Designed for managing and analyzing large data

Strengths of the Relational Model

- Simple data structure: relations
  - “Flat” table with schema (attribute names and types defining the columns), containing tuples (rows)
  - No nesting or pointers
- Example
  - Students(sid, name, age, GPA)
  - Reservations(sid, bookID, date)
  - Books(bookID, topic, title)

Running Examples

<table>
<thead>
<tr>
<th>SID</th>
<th>Name</th>
<th>Age</th>
<th>GPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Alice</td>
<td>18</td>
<td>3.5</td>
</tr>
<tr>
<td>2</td>
<td>Bob</td>
<td>27</td>
<td>3.4</td>
</tr>
<tr>
<td>3</td>
<td>Carla</td>
<td>20</td>
<td>3.8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BookID</th>
<th>Topic</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>B10</td>
<td>DB</td>
<td>Intro DB</td>
</tr>
<tr>
<td>B11</td>
<td>PL</td>
<td>More PL</td>
</tr>
</tbody>
</table>

More Strengths

- Specially designed query language
- Comparably simple operators that can be composed into complex queries
  - Enables automatic query optimization
- Not Turing-complete, e.g., not designed for complex calculations, but for easy efficient access to large data
- Relational calculus: basis for SQL
- Relational algebra: useful for representing query plans
Relational Algebra

- Basic operations:
  - Selection ($\sigma$): selects a subset of rows
  - Projection ($\pi$): selects a subset of columns
  - Cross-product ($\times$): combines two relations
  - Set-difference ($-$): tuples in one relation but not the other
  - Union ($\cup$): set union
- Additional operations: intersection, join, division, renaming (very useful)
- Algebra is closed, allowing composition
  - Each operation works on relations and returns a relation

Selection

$\sigma_{\text{age}>25}(\text{Students})$

<table>
<thead>
<tr>
<th>SID</th>
<th>Name</th>
<th>Age</th>
<th>GPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Alice</td>
<td>18</td>
<td>3.5</td>
</tr>
<tr>
<td>2</td>
<td>Bob</td>
<td>27</td>
<td>3.4</td>
</tr>
<tr>
<td>3</td>
<td>Carla</td>
<td>20</td>
<td>3.8</td>
</tr>
<tr>
<td>4</td>
<td>Dan</td>
<td>20</td>
<td>3.9</td>
</tr>
</tbody>
</table>

Projection

$\pi_{\text{Name, Age}}(\text{Students})$

<table>
<thead>
<tr>
<th>SID</th>
<th>Name</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Alice</td>
<td>18</td>
</tr>
<tr>
<td>2</td>
<td>Bob</td>
<td>27</td>
</tr>
<tr>
<td>3</td>
<td>Carla</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>Dan</td>
<td>20</td>
</tr>
</tbody>
</table>

Union, Intersection, Set-Difference

- Input relations have to be union-compatible

Cross-Product

- Pair each row from S with each row from Reservations
  - Rename those attributes occurring in both

Join

- Condition-join (aka theta-join)
  - $R \bowtie_{C} S = \sigma_{C}(R \times S)$
- Special cases
  - Equi-join: only equalities in C, no duplication of join columns
  - Natural join: equi-join on all common attributes
Example

- Find names of students who reserved a DB book
  \[
  \pi_{\text{name}}(\sigma_{\text{topic}=\text{DB}}(B) \bowtie R \bowtie S)
  \]

- Which one will be more efficient?
  – A query optimizer can find this automatically.

Relational Calculus

- Basis of SQL query language
- Algebra was not declarative, but calculus is
  – Many different algebra “implementations” possible for a calculus expression
  - Tuple relational calculus (TRC)
    – Variables range over tuples
  - Domain relational calculus (DRC)
    – Variables ranges over attribute values
  - Calculus expressions are called formulas
    – Answer tuple = assignment of constants to variables that make the formula evaluate to true

Domain Relational Calculus

- Query: \{\langle x_1, x_2, \ldots, x_n \rangle \ | \ p(x_1, x_2, \ldots, x_n)\}
- Answer = all tuples \(\langle x_1, x_2, \ldots, x_n \rangle\) that make formula \(p(x_1, x_2, \ldots, x_n)\) true
- All variables \(x_1, x_2, \ldots, x_n\) must be free, i.e., not bound by quantifier, in formula \(p(\ldots)\)
- No other variable in \(p(\ldots)\) is allowed to be free

DRC Formulas

- Atomic formula (op is one of \(<, >, =, \neq, \leq, \geq\) )
  – \(x_1, x_2, \ldots, x_n \in \text{Relation}, or \)
  – \(x \ op y, or \)
  – \(x \ op \text{const} \)
- Formula \((p, q \text{ are formulas})\)
  – Atomic formula, or
  – \(\neg p, p \land q, p \lor q, or \)
  – \(\exists x (p(x))\) or \(\forall x (p(x))\) where variable \(x\) is free in \(p(x)\)
  – Quantifiers \(\exists x\) and \(\forall x\) bind variable \(x\); free variables are not bound

Example

- All students with GPA above 3.6
  – \(\{S, N, A, G\} \in \text{Students} \land G > 3.6\)
- First condition: domain variables \(S, N, A, G\) have to be attributes of the same student tuple
- Second condition: GPA selection

<table>
<thead>
<tr>
<th>SID</th>
<th>Name</th>
<th>Age</th>
<th>GPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Alice</td>
<td>18</td>
<td>3.5</td>
</tr>
<tr>
<td>2</td>
<td>Bob</td>
<td>27</td>
<td>3.4</td>
</tr>
<tr>
<td>3</td>
<td>Carla</td>
<td>20</td>
<td>3.8</td>
</tr>
<tr>
<td>4</td>
<td>Dan</td>
<td>20</td>
<td>3.9</td>
</tr>
</tbody>
</table>

Example

- Students with GPA above 3.6 who reserved book B10
  – \(\{S, N, A, G\} \in \text{Students} \land G > 3.6\)
    \land \exists S_2, B, D(\langle S_2, B, D \rangle \in \text{Reservations} \land S = S_2\)
    \land B = B10)
- \(\exists S_2, B, D\) is a shorthand for \(\exists S_2 (\exists B (\exists D (\ldots)))\)
- Exists clause is used to find a tuple in Reservations that joins with the student tuple under consideration

<table>
<thead>
<tr>
<th>SID</th>
<th>Name</th>
<th>Age</th>
<th>GPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Alice</td>
<td>18</td>
<td>3.5</td>
</tr>
<tr>
<td>2</td>
<td>Bob</td>
<td>27</td>
<td>3.4</td>
</tr>
<tr>
<td>3</td>
<td>Carla</td>
<td>20</td>
<td>3.8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SID</th>
<th>BookID</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>B10</td>
<td>01/12/12</td>
</tr>
<tr>
<td>3</td>
<td>B11</td>
<td>01/14/12</td>
</tr>
</tbody>
</table>
Safe Queries and Expressive Power

- Possible to write calculus queries with infinite number of answers—called unsafe queries
  \[ \{(S, N, A, G) \mid \neg (S, N, A, G) \in \text{Students}\} \]
- Safe query: returns same result, no matter the attribute domains
- Every relational algebra query can be expressed as a safe query in DRC/TRC, and vice versa
- Relational completeness: query language, e.g., SQL, can express every relational algebra query

Example

```sql
SELECT S.age, S.age+1 AS age1, 2*S.age AS age2
FROM Students S
WHERE S.name LIKE 'J_%E'
```

- Can use arithmetic expressions in attribute-list and also in WHERE clause
- AS gives name to a result attribute
  - Could also use "as": age1 = S.age+1
- LIKE matches strings
  - "_*" matches any single character
  - "%" matches 0 or more arbitrary characters

Students Who Reserved DB or PL Book

```sql
SELECT S.SID
FROM Students S, Reservations R, Books B
AND (B.topic = 'DB' OR B.topic = 'PL')
```

- \( \{(S, N, A, G) \mid \{(S, N, A, G) \in \text{Students} \land G > 36
\land \exists B_1, D_1 (S = S_2
\land \exists B_2, O_1 (B_2 = B_2
\land O_1 = 'DB' \lor O_1 = 'PL'))\}\} \)
- What if we want those who reserved a DB and a PL book?
  - AND instead of OR would not work
  - Need to use INTERSECT
  - Careful: intersection needs to be on unique students, i.e., SID
    not just S.name

Nested Query with Correlation

```sql
SELECT S.name
FROM Students S
WHERE EXISTS (SELECT *
FROM Reservations R
WHERE R.bookID = 'B10' AND S.SID = R.SID)
```

- EXISTS tests is set empty
- If sub-query depends on outside attributes, have to re-compute for every value of them
- UNIQUE (instead of EXISTS): checks if there is at most one reservation for the student
  - Choice of attribute-list in sub-query affects UNIQUE

Aggregate Operators

```sql
SELECT COUNT(*), COUNT( [DISTINCT] A )
SELECT SUM( [DISTINCT] A )
SELECT AVG( [DISTINCT] A )
SELECT MAX(A), MIN(A)
```

- COUNT(*), COUNT( [DISTINCT] A )
- SUM( [DISTINCT] A )
- AVG( [DISTINCT] A )
- MAX(A), MIN(A)
GROUP BY and HAVING

SELECT [DISTINCT] attribute-list
FROM relation-list
WHERE condition
GROUP BY grouping-list
HAVING group-condition

- Attribute-list contains attribute names and terms with aggregate operations
  - Attributes in attribute-list must appear in grouping-list
  - Reason: single attribute value per group!

Example

- Among all students with GPA > 3.4, find the lowest GPA for each age group with at least 2 students

SELECT S.age, MIN(S.GPA) AS MinGPA
FROM Students S
WHERE S.GPA > 3.4
GROUP BY S.age
HAVING COUNT(*) >= 2

Evaluation

What Is a Query Plan?

- DAG of relational operators and their implementations
  - Use index vs. scan entire relation
  - Data partitioning for divide-and-conquer strategy
- Pull interface: output “pulls” next tuple from upstream operators
  - Enables pipelining, avoids buffering

The System R Optimizer

- Most widely used optimizer style
- Transforms SQL query to initial plan (actually multiple query blocks)
- Considers alternative plans by leveraging relational algebra equivalences
- For each plan, combined CPU and I/O cost is estimated
  - Challenge: estimate size of intermediate results
- Search space too large, hence optimizer relies on heuristics to enumerate candidate plans

Plans Involving Joins

- Avoid Cartesian products
- Joins can be executed in any order
  - Exponential number of query plans
- Optimizer only considers left-deep plans
  - Allows pipelining of intermediate results
  - No need to materialize temporary relations
- Still many possible join orders...
Simple Example

Relational algebra tree

\[ \sigma_{GPA>3.7 \land date=1/1/12}(R \bowtie S) \]

Cost: 1000+1000-500 for join plus zero I/O for on-the-fly
Total: ~500,000 I/O

Alternative Plan: Push Selections

- Scan R, write Temp1: 1000+10
  - Assume 100 reservation days
- Scan S, write Temp2: 500+250
  - Assume 50% have GPA>3.7
- Join: 2*2*10 + 2*2*250 + 10+250
  - Total: 3060 I/O

Could lower cost further by also pushing projections and using other join implementation

Another Alternative: With Indexes

- Index on R:
  - 100K/100=1000 tuples on 1000/100=10 pages
- Join: for each R-tuple, index lookup on S:
  - 1000*1.2
- Total: 1210 I/O

Parallel Databases

- Same SQL query, just replace the optimizer
  - Take data location and network cost into account
  - Optimize for latency or total cost
- Add new operators
  - Exchange: behaves like an iterator, but receives input via inter-process communication rather than iterator procedure calls
  - Split and Merge: create and join parallel dataflows
- Add new operator implementations
  - Semi-join to reduce network communication cost

Distributed Query Optimization

- Start: calculus query on global relations
- Transform into algebraic query on global relations
- Perform data localization, using fragment schema, to generate algebraic query on fragments
- Perform global optimization to create distributed query execution plan
- Run on local sites in parallel

Pipeline Parallelism

- Computation of one operator proceeds in parallel with another
- Model: output pulls from last operators, which pulls from its inputs and so on
Limited Benefits of Pipeline Parallelism

- Relational pipelines are usually not very long
  - Ten or longer is rare
- Some operators are blocking and cannot be pipelined
  - Aggregates, sorting
- Execution cost of one operator might be much larger than the others
  - Limits speedup obtained by pipelining

Partitioned Parallelism

- Query performs batch-style computation on many input tuples

Data Partitioning

- Round-robin
  - Simple, but not helpful for associative access
- Hash partitioning
  - Assign tuples to partition using hash function
  - Good for associative access (equality-based)
  - Not good for range queries
- Range partitioning
  - Partition data into continuous ranges
  - Good for range queries, parallel sort
  - Risks data skew (uneven partitions) and execution skew (uneven access pattern)

Distributed Transactions?

- Transactions were crucial for the success of database systems
- Enable concurrent processing of multiple queries, but programmers could write them as if they executed in isolation

Transactions

- Transaction = user program, consisting of a sequence of DB reads and writes
- Let users write programs under the illusion that there is no concurrent access
- DBMS automatically takes care of scheduling
  - Interleaves transactions, but ensures result is identical to isolated execution
- Give programmer a simple mechanism for declaring all-or-nothing execution of a block of statements

ACID Properties

- Atomicity: Either all or none of the transaction’s actions are executed
  - Even when a crash occurs mid-way
- Consistency: Transaction run by itself must preserve consistency of the database
  - User’s responsibility
- Isolation: Transaction semantics do not depend on other concurrently executed transactions
- Durability: Effects of successfully committed transactions should persist, even when crashes occur
Example

- Two bank accounts, A and B, owned by same user
- User transfers $100 from B to A
- Bank computes 1% interest on both
- Assume start state is A=500, B=500
- Correct serial executions: total interest = $10
  - T1, T2: A=606, B=404
  - T2, T1: A=605, B=405

Interleaving Scenarios

- First: ok, equivalent to T1, T2 order
- Second: not ok
  - A=606, B=505
- Abstract view shows the conflict

Scheduling Transactions

- Serial schedule: Schedule that does not interleave the actions of different transactions
  - Easy for programmer, easy to achieve consistency
  - Bad for performance
- Equivalent schedules: For any database state, the effect (on the objects in the database) of executing the first schedule is identical to the effect of executing the second schedule
- Serializable schedule: A schedule that is equivalent to some serial execution of the transactions
  - Retains advantages of serial schedule, but addresses performance issue
- Note: If each transaction preserves consistency, every serializable schedule preserves consistency

Anomalies: WR

- Reading uncommitted data (WR-conflict, "dirty read")
  - T2 reads value A written by T1 before T1 completed its changes
  - If T1 later aborts, T2 worked with invalid data
- Example: T1 deposits check to A, T2 credits interest to A

Problems With Dirty Reads

- Dirty read can result in unrecoverable schedule
  - T2 worked with invalid data and hence has to be aborted as well
  - But T2 already committed...
- Recoverable schedule: cannot allow T2 to commit until T1 has committed
  - Can still lead to cascading aborts

Anomalies: RW

- Unrepeatable read (RW conflict)
- T1 sees different values of A, even though it did not change it
- Example: online book store
  - Only one copy left; both T1 and T2 try to order it
  - One will get an error message
  - Could not have happened with serial execution
### Anomalies: WW

**T1:** \( W(A), W(B), C \)  
**T2:** \( W(A), W(B), C \)

- Overwriting uncommitted data (WW conflict)
- T1’s B and T2’s A persist, which would not happen with serial execution
- Example: two employees with same salary
  - T1 sets both salaries to $4000, T2 to $4500
  - Above schedule results in \( A=4500, B=4000 \)

### Preventing Anomalies Through Locking

- Block problematic concurrent actions, but allow non-conflicting ones
  - Many transactions can read the same data concurrently
  - If T1 affects accounts A and B, while T2 works on X and Y, they can even perform updates concurrently
- Lock the right DB objects using appropriate locks to allow maximum concurrency without suffering from anomalies

### Locking Basics

- Before being able to read an object, transaction needs to acquire a **shared lock** (S-lock) on it
- Before being able to modify an object, transaction needs to acquire an **exclusive lock** (X-lock) on it
- Multiple transactions can hold a shared lock on the same object
- At most one transaction can hold an exclusive lock on an object

### Two-Phase Locking

- Phase 1: acquire locks
- Phase 2: release locks (cannot acquire new locks any more)
- Ensures serializable schedule, but does not necessarily prevent dirty reads
- **Strict 2PL:** all locks are released only when the transaction is completed
  - Prevents all anomalies shown earlier
- Problem: deadlocks

### Deadlocks

- Ex: T1, T2 both want to read and write objects A and B  
  - T1 acquires X-lock on A; T2 acquires X-lock on B  
  - T1 wants to update B: waits for T2 to release its lock on B  
  - T2 wants to read A: waits for T1 to release its lock on A  
  - Strict 2PL does not allow either to release its locks before the transaction completed. **Deadlock!**
- DBMS can detect this
  - Cycle in waits-for graph (nodes = transactions, edges = objects they are waiting for)
  - Breaks deadlock by aborting one of the involved transactions
    - Which one to choose? Work performed is lost.

### Aborting a Transaction

- All of T1’s actions have to be undone
  - If another txn T2 has read an object last written by T1, T2 must be aborted as well!
  - Strict 2PL avoids such **cascading aborts** by releasing a transaction’s locks only at commit time
- In order to undo the actions of an aborted transaction, the DBMS maintains a log in which every write is recorded
  - This mechanism is also used to recover from system crashes: all active txns at the time of the crash are aborted when the system comes back up
The Phantom Problem

• Assume initially the youngest student is 20 years old
• T1 contains twice: SELECT MIN(age) FROM Students
• T2 inserts a new student with age 18
• Consider the following schedule:
  – T1 runs query, T2 inserts new student, T1 runs query again
  – T1 sees two different results, i.e., an unrepeatable read
• Would Strict 2PL prevent this?
  – Assume T1 acquires S-lock on each existing Student tuple
  – T2 inserts a new tuple, which is not locked by T1
  – T2 releases its X-lock on the new student before T1 reads Students again
• What went wrong?

What Should We Lock?

• T1 cannot lock a tuple that T2 will insert
  …but T1 could lock the entire Students table
  – Now T2 cannot insert anything until T1 completed
• What if T1 computed a slightly different query:
  – SELECT MIN(age) FROM Students WHERE GPA > 3.5
• Now locking the entire Students table seems excessive, because inserting a new student with GPA ≤ 3.5 would not create a problem
  – T1 could lock the predicate [GPA > 3.5] on Students
• General challenge: DBMS needs to choose appropriate granularity for locking

Performance Of Locking

• Locks force transactions to wait
• Abort and restart due to deadlock wastes the work done by the aborted transaction
  – In practice, deadlocks are rare, e.g., due to lock downgrades approach
    • Request X-lock initially, then downgrade to S-lock when it becomes clear that only read access was needed
• More concurrent transactions => more lock contention
  – Allowing more concurrent transactions initially increases throughput, but at some point leads to thrashing
  – Solution: limit max number of concurrent transactions
  – Minimize lock contention by reducing time locks are held and by avoiding hotspots (objects frequently accessed)

Distributed Transactions

• Transactions take longer to access remote objects
  – Need to hold locks longer
  – Greater probability for waiting and deadlocks
• What if the network partitions?
  – Transaction cannot acquire/release some locks
• Even without partitions, the problem is hard
  – Need to coordinate commit between multiple nodes
  – What happens if some participating node crashes?
• Standard protocol: 2PC (2-phase commit)

2PC Basics

• Commit-request phase
  – Coordinator asks all participants to prepare for commit
  – Participants vote YES or NO to commit request
• Commit phase
  – Based on participants’ votes, coordinator decides to commit (if all voted YES) or abort
  – Coordinator notifies participants about decision
  – Participants apply corresponding action (commit or abort) locally

2PC Problems

• 2PC = blocking protocol
  – Nodes cannot make a decision without hearing from coordinator, e.g., might hold on to locks forever if coordinator is down and they answered YES to first request
• Expensive for many-worker transactions
• Some issues were addressed by later 2PC modifications, but the basic problems remain
NoSQL to the Rescue?

- **Examples:** MongoDB, CouchDB, HBase, Google’s BigTable, Amazon’s Dynamo
- **Many driven by performance challenges**
  - Inherent tradeoff: consistency, availability, and tolerance to network partitions (Eric Brewer, UC Berkeley)
  - Maintaining consistent state across 100s of machines requires expensive agreement
  - Failures reduce availability, unless consistency is weakened
- **Solutions:** weaker consistency guarantees, limited functionality, or tailored solution for specific workload

Bigtable


HBase

- Open-source implementation of BigTable
- Part of the Hadoop ecosystem
- Supports fast “random” reads and writes in Big Data
- Scales by adding more nodes
  - Scales to billions of rows, millions of columns
- Does not support SQL
- No transactions, but row-level atomicity
  - Explicit row locks can be set by client application

Data Model

- Data stored in tables
- Tables consist of rows and columns
  - Each table cell is versioned, e.g., html content of www.neu.edu on different dates
  - Cell content = un-interpreted array of bytes
- Row keys (=primary key) are byte arrays
  - Can use any serializable type
- Table is stored sorted by row key: byte-ordered
  - Choose key wisely according to query workload

HBase Characteristics

- All data accesses by row key or scanning
- No real indexes
  - Fast loading of data
  - No secondary indexes, but some projects exist for adding them
- No support for joins
  - Store wide de-normalized table
- When adding a node, some regions will be moved to it automatically
Column Families

- Columns are grouped into column families
  - E.g., `temperature` family contains columns `temperature:air` and `temperature:dew_point`
- Column families are stable
  - Specified as part of schema definition
- Individual columns can be added or removed easily
- All column family members are stored and managed together

Data Storage

- Table automatically partitioned into regions
  - Range of row keys
- Region managed by RegionServer, stored in HDFS or S3 etc.
- Small table → single partition → single-node database until it splits

Accessing HBase

- Clients connect to ZooKeeper to find Master
- Learn about RegionServer holding the requested data from Master
- Contact RegionServer for the actual data directly
  - Client caches region information it has learned for future accesses
- Write-ahead logging to HDFS ensures durability even if RegionServer crashes
  - Simplified DBMS-style redo of committed writes

HBase Clients

- Java program, e.g., from MapReduce
- Avro
- REST
- Thrift

Source for following examples: Hadoop: The Definitive Guide, by Tom White

```java
public class ExampleClient {
    public static void main(String[] args) throws IOException {
        Configuration config = HBaseConfiguration.create();
        // Create table
        HBaseAdmin admin = new HBaseAdmin(config);
        HTableDescriptor htd = new HTableDescriptor("test");
        HColumnDescriptor hcd = new HColumnDescriptor("data");
        htd.addFamily(hcd);
        admin.createTable(htd);
        byte[] tablename = htd.getName();
        HTableDescriptor[] tables = admin.listTables();
        if (tables.length != 1 && Bytes.equals(tablename, tables[0].getName())) {
            throw new IOException("Failed create of table");
        }

        // Add new row named row1 to the table
        byte[] row1 = Bytes.toBytes("row1");
        Put p1 = new Put(row1);
        byte[] databytes = Bytes.toBytes("data");
        p1.add(databytes, Bytes.toBytes("1"), Bytes.toBytes("value1"));
        table.put(p1);

        // Drop the table
        admin.disableTable(tablename);
        admin.deleteTable(tablename);
    }
}
```
HBase and MapReduce

- Use `org.apache.hadoop.hbase.mapreduce`
- `TableInputFormat` makes sure each map task receives a single region
- `TableOutputFormat` allows reduce to write to an HBase table
- Look at the weather example in the Tom White book

Hive

- Initially developed by Facebook
- SQL-style data analysis on top of MapReduce
  - Write query in `HiveQL`
  - Automatically translated to plain MapReduce
- Integrates well with SQL tools, e.g., ODBC and JDBC
- Examples from Hadoop: The Definitive Guide, by Tom White

Hive vs. Relational DBMS

- Schema verified at query time
  - DBMS: schema enforced at data load time
- No updates or deletes, but insert is possible
- Rudimentary, but expanding, indexing support
  - Compact index: HDFS block numbers for each value
  - Bitmap index: compressed sets of rows where some value appears

Hive vs. Relational DBMS (cont.)

- Table- and partition-level locking
  - Locks managed by ZooKeeper
- Complex types ARRAY, MAP, and STRUCT
- Ongoing integration with HBase
Hive Storage

- Can use local file system, HDFS, S3 and so on
- Managed table: Hive copies files into its warehouse directory
- External table: location outside warehouse directory
- Row-oriented data layout: row 1, row 2, row 3
- Column-oriented layout: col 1 of some rows, col 2 of some rows, col 1 of next rows, col 2 of next rows

```
CREATE TABLE records (year STRING, temperature INT, quality INT)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ',';
LOAD DATA INPATH 'input/ncdc/micro-tab/sample.txt'
OVERWRITE INTO TABLE records;
```

Partitions

- Tables divided into partitions
  - Based on partition column(s), e.g., date
  - Partition attribute values determine directory structure
- Attributes used for partitioning do not appear in table schema any more

```
CREATE TABLE logs (ts BIGINT, line STRING)
PARTITIONED BY (dt STRING, country STRING);
```

Buckets

- Re-orders data in table
  - “Groups by” some attribute
  - Can sort within each group
- Very useful for equi-join
  - Hash-join style implementation

```
CREATE TABLE bucketed_users (id INT, name STRING)
CLUSTERED BY (id) SORTED BY (id ASC) INTO 4 BUCKETS;
```

Hive Query

- Plain SQL
- Executed as MapReduce job

```
SELECT year, MAX(temperature)
FROM records
WHERE temperature != 9999
AND (quality = 0 OR quality = 1 OR quality = 4 OR quality = 5 OR quality = 9)
GROUP BY year;
```

Joins in Hive

- Inner join, outer join, semi join
- Hive uses rule-based optimizer
- Cost-based optimizer might be added in the future

```
SELECT sales.*, things.*
FROM sales JOIN things ON (sales.id = things.id);
```

```
SELECT sales.*, things.*
FROM sales LEFT OUTER JOIN things ON (sales.id = things.id);
```

```
SELECT * FROM things LEFT SEMI JOIN sales ON (sales.id = things.id);
Same as: SELECT * FROM things WHERE things.id IN (SELECT id FROM sales);
```

Advanced Features

- Subqueries
  - Limited compared to DBMS: only in FROM clause
- Views
  - Defined by HiveQL query
- User-defined functions: written in Java
- User-defined aggregate functions
  - Init() to reset internal state
  - Iterate() to update state
  - TerminatePartial() to get partial result
  - Merge() to combine partial results
  - Terminate() to generate final result