Lecture 6:
Query Execution and Optimization
Parallel Data processing
Announcements

• HW5 due today

• HW6 released
  – Please start early! You need to apply for credits from Amazon

• Two lectures this week (tonight and Thurs)
  – Query optimization
  – Parallel data processing
  – Conceptual design

• No reading assignment for conceptual design

• OH change this week to Thursday
Query Execution and Optimization
Class overview

• Data models
  – Relational: SQL, RA, and Datalog
  – NoSQL: SQL++
• RDBMS internals
  – Query processing and optimization
  – Physical design
• Parallel query processing
  – Spark and Hadoop
• Conceptual design
  – E/R diagrams
  – Schema normalization
• Transactions
  – Locking and schedules
  – Writing DB applications

Data models
Query Processing
Using DBMS
Query Evaluation Steps Review

1. Parse & Rewrite Query
2. Select Logical Plan
3. Select Physical Plan
4. Query Execution
Implementing Query Operators with the Iterator Interface

interface Operator {

    // initializes operator state
    // and sets parameters
    void open (...);

    // calls next() on its inputs
    // processes an input tuple
    // produces output tuple(s)
    // returns null when done
    Tuple next () ;

    // cleans up (if any)
    void close () ;
}

class Select implements Operator {

    void open (Predicate p, Operator child) {
        this.p = p; this.child = child;
    }

    Tuple next () {
        boolean found = false;
        Tuple r = null;
        while (!found) {
            r = child.next();
            if (r == null) break;
            found = p(r);
        }
        return r;
    }

    void close () { child.close(); }
}

Example “on the fly” selection operator
Implementing Query Operators with the Iterator Interface

```java
interface Operator {

    // initializes operator state
    // and sets parameters
    void open (...);

    // calls next() on its inputs
    // processes an input tuple
    // produces output tuple(s)
    // returns null when done
    Tuple next ();

    // cleans up (if any)
    void close ();
}

Query plan execution

Operator q = parse("SELECT ...");
q = optimize(q);

q.open();
while (true) {
    Tuple t = q.next();
    if (t == null) break;
    else printOnScreen(t);
}
q.close();
```
Suppliers

Supply(sid, pno, quantity)

\begin{align*}
\text{Supplier}(\text{sid, sname, scity, sstate}) \\
\text{Supply}(\text{sid, pno, quantity})
\end{align*}

\textbf{Pipelining}

\( (\text{On the fly}) \)

\( (\text{On the fly}) \sigma_{\text{scity} = 'Seattle' \ and \ sstate = 'WA' \ and \ pno = 2} \)

\( (\text{Nested loop}) \)

\( \pi_{\text{sname}} \)

\( \text{sno} = \text{sno} \)

\textbf{Discuss: open/next/close for nested loop join}

Suppliers (File scan)

Supplies (File scan)
Recall: Physical Data Independence

- Applications are insulated from changes in physical storage details

- SQL and relational algebra facilitate physical data independence
  - Both languages input and output relations
  - Can choose different implementations for operators
Class overview

• Data models
  – Relational: SQL, RA, and Datalog
  – NoSQL: SQL++
• RDBMS internals
  – Query processing and optimization
  – Physical design
• Parallel query processing
  – Spark and Hadoop
• Conceptual design
  – E/R diagrams
  – Schema normalization
• Transactions
  – Locking and schedules
  – Writing DB applications
Hash table example

Index **Student_ID** on **Student.ID**

Data File **Student**

<table>
<thead>
<tr>
<th>ID</th>
<th>fName</th>
<th>lName</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Tom</td>
<td>Hanks</td>
</tr>
<tr>
<td>20</td>
<td>Amy</td>
<td>Hanks</td>
</tr>
</tbody>
</table>

*Index File (in memory)*

*Data file (on disk)*

CSEP 544 - Fall 2017
B+ Tree Index by Example

d = 2

Find the key 40
Basic Index Selection Guidelines

• Consider queries in workload in order of importance

• Consider relations accessed by query
  – No point indexing other relations

• Look at WHERE clause for possible search key

• Try to choose indexes that speed-up multiple queries
Cost of Reading Data From Disk
Cost Parameters

- Cost = I/O + CPU + Network BW
  - We will focus on I/O in this class

- Parameters:
  - $B(R) = \# \text{ of blocks (i.e., pages) for relation } R$
  - $T(R) = \# \text{ of tuples in relation } R$
  - $V(R, a) = \# \text{ of distinct values of attribute } a$
    - When $a$ is a key, $V(R, a) = T(R)$
    - When $a$ is not a key, $V(R, a)$ can be anything $\leq T(R)$

- Where do these values come from?
  - DBMS collects statistics about data on disk
Selectivity Factors for Conditions

• $A = c$ /* $\sigma_{A=c}(R)$ */
  – Selectivity = $1/V(R,A)$

• $A < c$ /* $\sigma_{A<c}(R)$ */
  – Selectivity = $(c - \min(R, A))/\left(\max(R,A) - \min(R,A)\right)$

• $c_1 < A < c_2$ /* $\sigma_{c_1<A<c_2}(R)$ */
  – Selectivity = $(c_2 - c_1)/\left(\max(R,A) - \min(R,A)\right)$
Cost of Executing Operators (Focus on Joins)
Join Algorithms

- Hash join
- Nested loop join
- Sort-merge join
Hash Join

Hash join: \( R \bowtie S \)

- Scan \( R \), build buckets in main memory
- Then scan \( S \) and join
- Cost: \( B(R) + B(S) \)
- Which relation to build the hash table on?

- One-pass algorithm when \( B(R) \leq M \)
  - \( M = \) number of memory pages available
Nested Loop Joins

• Tuple-based nested loop $R \bowtie S$
• $R$ is the outer relation, $S$ is the inner relation

\[
\begin{align*}
\text{for each tuple } t_1 \text{ in } R & \text{ do} \\
& \quad \text{for each tuple } t_2 \text{ in } S \text{ do} \\
& \quad \quad \text{if } t_1 \text{ and } t_2 \text{ join then output } (t_1, t_2)
\end{align*}
\]

• Cost: $B(R) + T(R)B(S)$
• Multiple-pass since $S$ is read many times
Block-Nested-Loop Refinement

for each group of M-1 pages r in R do
  for each page of tuples s in S do
    for all pairs of tuples t₁ in r, t₂ in s
      if t₁ and t₂ join then output (t₁, t₂)

• Cost: B(R) + B(R)B(S)/(M-1)
Sort-Merge Join

Sort-merge join: \( R \bowtie S \)
- Scan \( R \) and sort in main memory
- Scan \( S \) and sort in main memory
- Merge \( R \) and \( S \)

- Cost: \( B(R) + B(S) \)
- One pass algorithm when \( B(S) + B(R) \leq M \)
- Typically, this is NOT a one pass algorithm
Index Nested Loop Join

\[ R \bowtie S \]

- Assume S has an index on the join attribute
- Iterate over R, for each tuple fetch corresponding tuple(s) from S

- **Cost:**
  - If index on S is clustered:
    \[ B(R) + T(R) \times (B(S) \times \frac{1}{V(S,a)}) \]
  - If index on S is unclustered:
    \[ B(R) + T(R) \times (T(S) \times \frac{1}{V(S,a)}) \]
Cost of Query Plans
Logical Query Plan 1

\[ \sigma_{\text{pno}=2 \land \text{scity}='Seattle' \land \text{sstate}='WA'}(\text{Supplier}) \]

\[ \pi_{\text{sname}}(\text{Supplier}) = 1000 \]

\[ \text{B(Supplier)} = 100 \]

\[ \text{V(Supplier, pno)} = 2500 \]

\[ \text{T(Supplier)} = 10000 \]

\[ \text{V(Supplier, scity)} = 20 \]

\[ \text{V(Supplier, state)} = 10 \]

\[ \text{M} = 11 \]
Logical Query Plan 1

\[\sigma_{\text{pno}=2 \land \text{scity}='Seattle' \land \text{sstate}='WA'}(\text{Supplier})\]

\[\Pi_{\text{sname}}(\text{Supplier})\]

\[\text{SELECT} \ \text{sname} \ \text{FROM} \ \text{Supplier \ x, \ Supply \ y} \ \text{WHERE} \ x.\text{sid} = y.\text{sid} \ \text{and} \ y.\text{pno} = 2 \ \text{and} \ x.\text{scity} = 'Seattle' \ \text{and} \ x.\text{sstate} = 'WA'\]

\[\text{T(Supplier)} = 1000 \ \text{B(Supplier)} = 100 \ \text{V(Supplier, scity)} = 20 \ \text{V(Supplier, state)} = 10\]

\[\text{M} = 11\]
Logical Query Plan 1

\[ \sigma_{pno=2 \land scity='Seattle' \land sstate='WA'} \]

\[ \Pi_{sname} \]

\[ T < 1 \]

\[ T = 10000 \]

\[ \text{SELECT } sname \]
\[ \text{FROM } Supplier \times, Supply \times y \]
\[ \text{WHERE } x.\text{sid} = y.\text{sid} \]
\[ \text{and } y.\text{pno} = 2 \]
\[ \text{and } x.\text{scity} = 'Seattle' \]
\[ \text{and } x.\text{sstate} = 'WA' \]
Logical Query Plan 2

```
SELECT sname
FROM Supplier x, Supply y
WHERE x.sid = y.sid
  and y.pno = 2
  and x.scity = 'Seattle'
  and x.sstate = 'WA'
```

```
M = 11
```

Data:
- `T(Supplier) = 10000`
- `B(Supplier) = 100`
- `V(Supplier, scity) = 20`
- `V(Supplier, state) = 10`
- `T(Supply) = 10000`
- `B(Supply) = 100`
- `V(Supply, pno) = 2500`
Logical Query Plan 2

SELECT sname
FROM Supplier x, Supply y
WHERE x.sid = y.sid
   and y.pno = 2
   and x.scity = 'Seattle'
   and x.sstate = 'WA'

T(Supplier) = 10000
B(Supplier) = 100
V(Supplier, scity) = 20
V(Supplier, sstate) = 10

M=11
Logical Query Plan 2

\[
\text{SELECT } sname \\
\text{FROM Supplier x, Supply y} \\
\text{WHERE x.sid = y.sid} \\
\text{and y.pno = 2} \\
\text{and x.scity = 'Seattle'} \\
\text{and x.sstate = 'WA'}
\]

- \( T(Supplier) = 10000 \)
- \( B(Supplier) = 100 \)
- \( V(Supplier, scity) = 20 \)
- \( V(Supplier, state) = 10 \)
- \( M = 11 \)

Very wrong! Why?
Logical Query Plan 2

\[
\begin{align*}
\text{SELECT} & \quad \text{sname} \\
\text{FROM} & \quad \text{Supplier } x, \text{ Supply } y \\
\text{WHERE} & \quad x.\text{sid} = y.\text{sid} \\
& \quad \text{and } y.\text{pno} = 2 \\
& \quad \text{and } x.\text{scity} = \text{'Seattle'} \\
& \quad \text{and } x.\text{sstate} = \text{'WA'}
\end{align*}
\]

\[
T(\text{Supply}) = 10000 \\
B(\text{Supply}) = 100 \\
V(\text{Supply, pno}) = 2500
\]

\[
T(\text{Supplier}) = 1000 \\
B(\text{Supplier}) = 100 \\
V(\text{Supplier, scity}) = 20 \\
V(\text{Supplier, state}) = 10
\]

\[M=11\]
Logical Query Plan 2

\[
\begin{align*}
\text{SELECT } & \quad \text{sname} \\
\text{FROM } & \quad \text{Supplier } x, \text{ Supply } y \\
\text{WHERE } & \quad x.\text{sid} = y.\text{sid} \\
& \quad \text{and } y.\text{pno} = 2 \\
& \quad \text{and } x.\text{scity} = 'Seattle' \\
& \quad \text{and } x.\text{sstate} = 'WA'
\end{align*}
\]

\[
\begin{align*}
T(\text{Supplier}) & = 1000 \\
B(\text{Supplier}) & = 100 \\
V(\text{Supplier}, \text{scity}) & = 20 \\
V(\text{Supplier}, \text{sstate}) & = 10 \\
M & = 11
\end{align*}
\]

\[
\begin{align*}
\text{Supp} & \text{ler}(\text{sid}, \text{sname}, \text{scity}, \text{sstate}) \\
\text{Supply}(\text{sid}, \text{pno}, \text{quantity})
\end{align*}
\]
Physical Plan 1

\[ \Pi_{\text{sname}} \]

\[ \sigma_{\text{pno}=2 \land \text{scity}=\text{Seattle} \land \text{sstate}=\text{WA}} \]

\[ T < 1 \]

\[ T = 10000 \]

\[ \text{sid} = \text{sid} \]

Block nested loop join

Total cost: 1000

Supply (sid, pno, quantity)

Supplier (sid, sname, scity, sstate)

T(Supply) = 10000
B(Supply) = 100
V(Supply, pno) = 2500

T(Supplier) = 1000
B(Supplier) = 100
V(Supplier, scity) = 20
V(Supplier, state) = 10

M=11
Physical Plan 1

\[ \Pi_{sname} \sigma_{pno=2 \land scity='Seattle' \land sstate='WA'} \]

\[ \text{T = 10000} \]

\[ \text{Block nested loop join} \]

\[ \text{Total cost: } 100 + 100 \times \frac{100}{10} = 1100 \]

\[ \text{T(Supply)} = 10000 \]
\[ \text{B(Supply)} = 100 \]
\[ \text{V(Supply, pno)} = 2500 \]

\[ \text{T(Supplier)} = 1000 \]
\[ \text{B(Supplier)} = 100 \]
\[ \text{V(Supplier, scity)} = 20 \]
\[ \text{V(Supplier, state)} = 10 \]
Physical Plan 2

Suppliers: (sid, sname, scity, sstate)
Supplies: (sid, pno, quantity)

Cost of Supply(pno) = 
Cost of Supplier(scity) = 
Total cost:

T(Supply) = 10000
B(Supply) = 100
V(Supply, pno) = 2500

T(Supplier) = 1000
B(Supplier) = 100
V(Supplier, scity) = 20
V(Supplier, state) = 10

M = 11
Physical Plan 2

\[ \text{Supplier}(\text{sid, sname, scity, sstate}) \]
\[ \text{Supply}(\text{sid, pno, quantity}) \]

\[ T(\text{Supply}) = 10000 \]
\[ B(\text{Supply}) = 100 \]
\[ V(\text{Supply, pno}) = 2500 \]

\[ T(\text{Supplier}) = 1000 \]
\[ B(\text{Supplier}) = 100 \]
\[ V(\text{Supplier, scity}) = 20 \]
\[ V(\text{Supplier, state}) = 10 \]

\[ M = 11 \]

Cost of \( \text{Supply}(\text{pno}) = 4 \)
Cost of \( \text{Supplier}(\text{scity}) = 50 \)
Total cost: 54
Physical Plan 2

```
σ_{pno=2} \rightarrow \text{Supply}
σ_{sstate='WA'} \rightarrow \text{Supplier}
π_{sname} \rightarrow \text{Supplier}
```

Cost of \text{Supply}(pno) = 4
Cost of \text{Supplier}(scity) = 50
Total cost: 54

T(\text{Supply}) = 10000
B(\text{Supply}) = 100
V(\text{Supply}, pno) = 2500

T(\text{Supplier}) = 1000
B(\text{Supplier}) = 100
V(\text{Supplier}, scity) = 20
V(\text{Supplier}, state) = 10

M=11
Physical Plan 3

\[ \pi_{\text{sname}} \]
\[ \sigma_{\text{scity}=\text{Seattle} \land \text{sstate}=\text{WA}} \]
\[ \text{sid} = \text{sid} \]
\[ \sigma_{\text{pno}=2} \]

\begin{align*}
T(\text{Supply}) &= 10000 \\
B(\text{Supply}) &= 100 \\
V(\text{Supply, pno}) &= 2500
\end{align*}

\begin{align*}
T(\text{Supplier}) &= 1000 \\
B(\text{Supplier}) &= 100 \\
V(\text{Supplier, scity}) &= 20 \\
V(\text{Supplier, state}) &= 10
\end{align*}

Cost of \text{Supply(pno)} = 4

Cost of Index join = 4

Total cost: 8

M = 11
Physical Plan 3

\[
\Pi_{\text{sname}} \left( \sigma_{\text{scity}=\text{Seattle} \land \text{sstate}=\text{WA}} \left( \sigma_{\text{pno}=2} \left( \text{Supply} \right) \right) \right)
\]

Cost of Supply(pno) = 4
Cost of Index join = 4
Total cost: 8

Clustered Index join

\[\sigma_{\text{scity}=\text{Seattle} \land \text{sstate}=\text{WA}}\]

Unclustered index lookup

Supply(pno)

\[\sigma_{\text{pno}=2}\]

\text{Supply}

\[\text{Supplier}
\]

T(\text{Supply}) = 10000
B(\text{Supply}) = 100
V(\text{Supply, pno}) = 2500

T(\text{Supplier}) = 1000
B(\text{Supplier}) = 100
V(\text{Supplier, scity}) = 20
V(\text{Supplier, state}) = 10

M = 11
Physical Plan 3

\[ \prod_{\text{name}} \sigma_{\text{city}=\text{Seattle} \land \text{state}=\text{WA}} \]

Cost of \( \text{Supply}(\text{pno}) = 4 \)
Cost of Index join = 4
Total cost: 8
Query Optimizer

```java
lowestCost = ∞;
bestPlan = null;
for (p : physicalPlan(q)) {
    if (cost(p) < lowestCost)
        bestPlan = p;
}
return p;
```

• This never works
• Way too many plans to consider!

• Typical query optimizer:
  • Construct logical plan \( p \)
  • Apply heuristic rules to transform \( p \)
    (e.g., do selection as early as possible)
  • Go through each operator \( \text{op} \) in bottom up manner
  • Choose an implementation for \( \text{op} \) to construct the physical plan
    (why does this not always return the best plan?)
The System R Optimizer
A Case Study
Two Types of Plan Enumeration Algorithms

- Dynamic programming
  - Based on System R (aka Selinger) style optimizer [1979]
  - Limited to joins: *join reordering algorithm*
  - Bottom-up

- Rule-based algorithm *(will not discuss)*
  - Database of rules (=algebraic laws)
  - Usually: dynamic programming
  - Usually: *top-down*
System R Search Space

- Only left-deep plans
  - Enable dynamic programming for enumeration
  - Facilitate tuple pipelining from outer relation
- Consider plans with all “interesting orders”
- Perform cross-products after all other joins (heuristic)
- Only consider nested loop & sort-merge joins
- Consider both file scan and indexes
- Try to evaluate predicates early
Plan Enumeration Algorithm

• Idea: use dynamic programming
• For each subset of \{R_1, \ldots, R_n\}, compute the best plan for that subset
• In increasing order of set cardinality:
  – Step 1: for \{R_1\}, \{R_2\}, \ldots, \{R_n\}
  – Step 2: for \{R_1,R_2\}, \{R_1,R_3\}, \ldots, \{R_{n-1}, R_n\}
  – ...
  – Step n: for \{R_1, \ldots, R_n\}
• It is a bottom-up strategy
• A subset of \{R_1, \ldots, R_n\} is also called a subquery
Dynamic Programming Algo.

• For each subquery $Q \subseteq \{R_1, \ldots, R_n\}$ compute the following:
  – Size($Q$)
  – A best plan for $Q$: Plan($Q$)
  – The cost of that plan: Cost($Q$)
Dynamic Programming Algo.

• **Step 1:** Enumerate all single-relation plans
  
  – Consider selections on attributes of relation
  – Consider all possible access paths
  – Consider attributes that are not needed

  – Compute cost for each plan

  – Keep cheapest plan per “interesting” output order
Dynamic Programming Algo.

• **Step 2:** Generate all two-relation plans
  - For each each single-relation plan from step 1
  - Consider that plan as outer relation
  - Consider every other relation as inner relation
  - Compute cost for each plan
  - Keep cheapest plan per “interesting” output order
Dynamic Programming Algo.

• **Step 3**: Generate all three-relation plans
  - For each each two-relation plan from step 2
  - Consider that plan as outer relation
  - Consider every other relation as inner relation
  - Compute cost for each plan
  - Keep cheapest plan per “interesting” output order

• **Steps 4 through n**: repeat until plan contains all the relations in the query
Query Optimizer Summary

• Input: A logical query plan
• Output: A good physical query plan
• Basic query optimization algorithm
  – Enumerate alternative plans (logical and physical)
  – Compute estimated cost of each plan
  – Choose plan with lowest cost

• This is called cost-based optimization
Parallel Data Processing
Class overview

• Data models
  – Relational: SQL, RA, and Datalog
  – NoSQL: SQL++

• RDMBS internals
  – Query processing and optimization
  – Physical design

• Parallel query processing
  – Spark and Hadoop

• Conceptual design
  – E/R diagrams
  – Schema normalization

• Transactions
  – Locking and schedules
  – Writing DB applications
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 and HW6
Performance Metrics for Parallel DBMSs

Nodes = processors, computers

- **Speedup:**
  - More nodes, same data \(\rightarrow\) higher speed

- **Scaleup:**
  - More nodes, more data \(\rightarrow\) same speed
Linear v.s. Non-linear Speedup

Speedup

# nodes (=P)

×1  ×5  ×10  ×15

Ideal
Linear v.s. Non-linear Scaleup

Batch Scaleup

# nodes (=P) AND data size

×1  ×5  ×10  ×15

Ideal
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
- check your HW3 query plans

- 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

Because all machines today have many cores and many disks, shared-nothing systems typically run many "nodes" on a single physical machine.

- Easy to maintain and scale
- Most difficult to administer and tune.

We discuss only Shared Nothing in class
Parallel Data Processing @ 1990
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)

Given relations R(A,B) and S(B, C), no indexes:

• **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

<table>
<thead>
<tr>
<th>K</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Data:

Servers:

1  2  ...  P
Horizontal Data Partitioning

Data:

<table>
<thead>
<tr>
<th>K</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Servers:

1

<table>
<thead>
<tr>
<th>K</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

2

<table>
<thead>
<tr>
<th>K</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

... 

P

<table>
<thead>
<tr>
<th>K</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Which tuples go to what server?
Horizontal Data Partitioning

• **Block Partition:**
  – Partition tuples arbitrarily s.t. \( \text{size}(R_1) \approx \ldots \approx \text{size}(R_P) \)

• **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 \( -\infty = v_0 < v_1 < \ldots < v_P = \infty \)
  – 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
  - Uniform

  • Hash-partition
    – On the key K
    – On the attribute A
      – Uniform
      – May be skewed

Assuming good hash function

E.g. when all records have the same value of the attribute A, then all records end up in the same partition

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 $K$?
Parallel Execution of RA Operators: Grouping

**Data:** $R(K, A, B, C)$

**Query:** $\gamma_{A, \text{sum}(C)}(R)$

- $R$ is block-partitioned or hash-partitioned on $K$

Reshuffle $R$ on attribute $A$

Run grouping on reshuffled partitions
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)
Parallel Execution of RA Operators: Partitioned Hash-Join

- **Data**: \( R(K_1, A, B), S(K_2, B, C) \)
- **Query**: \( R(K_1, A, B) \bowtie S(K_2, B, C) \)
  - Initially, both \( R \) and \( S \) are partitioned on \( K_1 \) and \( K_2 \)

- Reshuffle \( R \) on \( R.B \) and \( S \) on \( S.B \)
- Each server computes the join locally
Data: R(K1, A, B), S(K2, B, C)
Query: R(K1, A, B) \bowtie S(K2, B, C)

Parallel Join Illustration

Partition

Shuffle on B

Local Join
**Data:** $R(A, B), S(C, D)$

**Query:** $R(A, B) \bowtie_{B=C} S(C, D)$

### Broadcast Join

Reshuffle $R$ on $R.B$

Broadcast $S$

Why would you want to do this?
A Challenge

• Have P number of servers (say P=27 or P=1000)

• How do we compute this Datalog query in one step?

• \( Q(x, y, z) :\neg R(x, y), S(y, z), T(z, x) \)
A Challenge

- Have P number of servers (say P=27 or P=1000)
- How do we compute this Datalog query in one step? $Q(x,y,z) = R(x,y), S(y,z), T(z,x)$
- Organize the P servers into a cube with side $P^{1/3}$
  - Thus, each server is uniquely identified by $(i,j,k)$, $i,j,k \leq P^{1/3}$
HyperCube Join

• Have P number of servers (say P=27 or P=1000)
• How do we compute this Datalog query in one step?
  \[ Q(x,y,z) = R(x,y), S(y,z), T(z,x) \]
• Organize the P servers into a cube with side \( P^{\frac{1}{3}} \)
  – Thus, each server is uniquely identified by \((i,j,k)\), \(i,j,k \leq P^{\frac{1}{3}}\)
• Step 1:
  1, 2
  – Each server sends \( R(x,y) \) to all servers \((h(x), h(y), *)\)
  – Each server sends \( S(y,z) \) to all servers \((*, h(y), h(z))\)
  – Each server sends \( T(x,z) \) to all servers \((h(x), (*), h(z))\)
HyperCube Join

- Have P number of servers (say P=27 or P=1000)
- How do we compute this Datalog query in one step?
  \[ Q(x,y,z) = R(x,y), S(y,z), T(z,x) \]
- Organize the P servers into a cube with side \( P^{1/3} \)
  - Thus, each server is uniquely identified by \((i,j,k)\), \(i,j,k \leq P^{1/3}\)
- Step 1:
  - Each server sends \( R(x,y) \) to all servers \((h(x), h(y), \ast)\)
  - Each server sends \( S(y,z) \) to all servers \((\ast, h(y), h(z))\)
  - Each server sends \( T(x,z) \) to all servers \((h(x), \ast, h(z))\)
- Final output:
  - Each server \((i,j,k)\) computes the query \( R(x,y), S(y,z), T(z,x) \) locally
HyperCube Join

• Have P number of servers (say P=27 or P=1000)
• How do we compute this Datalog query in one step? $Q(x,y,z) = R(x,y), S(y,z), T(z,x)$
• Organize the P servers into a cube with side $P^{\frac{1}{3}}$
  – Thus, each server is uniquely identified by $(i,j,k)$, $i,j,k \leq P^{\frac{1}{3}}$
• Step 1:
  – Each server sends $R(x,y)$ to all servers $(h(x), h(y), *)$
  – Each server sends $S(y,z)$ to all servers $(*, h(y), h(z))$
  – Each server sends $T(x,z)$ to all servers $(h(x), *, h(z))$
• Final output:
  – Each server $(i,j,k)$ computes the query $R(x,y), S(y,z), T(z,x)$ locally
• Analysis: each tuple $R(x,y)$ is replicated at most $P^{\frac{1}{3}}$ times
\[ Q(x,y,z) = R(x,y), S(y,z), T(z,x) \]

<table>
<thead>
<tr>
<th>R1</th>
<th>S1</th>
<th>T1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>z</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>x</td>
<td>y</td>
<td>z</td>
</tr>
<tr>
<td>y</td>
<td>z</td>
<td>x</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

**Partition**

**Shuffle**

**Local Join**

**Hypercube join**

<table>
<thead>
<tr>
<th>R2</th>
<th>S2</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R2’</th>
<th>S2’</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R3</th>
<th>S3</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>z</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>z</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R3’</th>
<th>S3’</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>z</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**P1:** \( (1, 2, 7) \)

**P2:** \( (1, 2, 3) \)

**P3:** \( (3, 2, 3) \)
\[ Q(x, y, z) = R(x, y), S(y, z), T(z, x) \]

**Hypercube join**

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>S1</th>
<th>T1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td>y z</td>
<td>z x</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>1 1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>4</td>
<td>3 3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>R2</th>
<th>S2</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td>y z</td>
<td>z x</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>2</td>
<td>3 9</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>2</td>
<td>3 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>R3</th>
<th>S3</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td>y z</td>
<td>z x</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>6</td>
<td>7 1</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>6</td>
<td>3 1</td>
</tr>
</tbody>
</table>

**Partition**

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>S1</th>
<th>T1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td>y z</td>
<td>z x</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>1 1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>4</td>
<td>3 3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>R2</th>
<th>S2</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td>y z</td>
<td>z x</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>2</td>
<td>3 9</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>2</td>
<td>3 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>R3</th>
<th>S3</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td>y z</td>
<td>z x</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>6</td>
<td>7 1</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>6</td>
<td>3 1</td>
</tr>
</tbody>
</table>

**Shuffle**

What if

h(x): h(1) = h(3)?
The query \( Q(x,y,z) = R(x,y), S(y,z), T(z,x) \) is illustrated with a hypercube join. The partition and shuffle steps are shown with values distributed across partitions and then joined. The local join is applied to the results of the partition and shuffle steps.

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>S1</th>
<th>T1</th>
<th></th>
<th>R2</th>
<th>S2</th>
<th>T2</th>
<th></th>
<th>R3</th>
<th>S3</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x</td>
<td>y</td>
<td></td>
<td></td>
<td>x</td>
<td>y</td>
<td></td>
<td></td>
<td>x</td>
<td>y</td>
<td></td>
</tr>
<tr>
<td>P1</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td>4</td>
<td>7</td>
<td>1</td>
<td></td>
<td>8</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
<td>4</td>
<td>9</td>
<td>3</td>
<td></td>
<td>9</td>
<td>6</td>
<td>3</td>
</tr>
</tbody>
</table>

The partitions are labeled P1, P2, and P3. The local join results in P1: (1, 2, 7), P2: (1, 2, 3), and P3: (3, 2, 3). The question asks: What if \( h(x): h(1) = h(3) \)?
Putting it Together: Example Parallel Query Plan

Find all orders from today, along with the items ordered

SELECT *
FROM Order o, Line i
WHERE o.item = i.item
AND o.date = today()
Example Parallel Query Plan
Example Parallel Query Plan

Order(oid, item, date), Line(item, ...)

Node 1
- scan Item i
- hash h(i.item)

Node 2
- scan Item i
- hash h(i.item)

Node 3
- scan Item i
- hash h(i.item)

Join
- o.item = i.item
- date = today()
Example Parallel Query Plan

Node 1
- join: o.item = i.item
  - contains all orders and all lines where hash(item) = 1

Node 2
- join: o.item = i.item
  - contains all orders and all lines where hash(item) = 2

Node 3
- join: o.item = i.item
  - contains all orders and all lines where hash(item) = 3
The MapReduce Programming Paradigm
Parallel Data Processing @ 2000
Optional Reading

- Original paper: [https://www.usenix.org/legacy/events/osdi04/tech/dean.html](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](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](http://i.stanford.edu/~ullman/mmds.html)
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

Paradigm stays the same, change map and reduce functions for different problems

slide source: Jeff Dean
Data Model

Files!

A file = a bag of \((\text{key}, \text{value})\) pairs

A MapReduce program:
• Input: a bag of \((\text{inputkey}, \text{value})\) pairs
• Output: a bag of \((\text{outputkey}, \text{value})\) pairs
Step 1: the **MAP** Phase

User provides the **MAP**-function:
- Input: \((\text{input key, value})\)
- Output: bag of \((\text{intermediate key, value})\)

System applies the map function in parallel to all \((\text{input key, value})\) pairs in the input file
Step 2: the REDUCE Phase

User provides the REDUCE function:
• Input: \((\text{intermediate key}, \text{bag of values})\)
• Output: bag of output \((\text{values})\)

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

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += parseInt(v);
  Emit(AsString(result));
Jobs v.s. Tasks

• A MapReduce Job
  – One single “query”, e.g. count the words in all docs
  – More complex queries may consists of multiple jobs

• A Map Task, or a Reduce Task
  – A group of instantiations of the map-, or reduce-function, which are scheduled on a single worker
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
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
MAP Tasks

REDUCE Tasks

Shuffle

CSEP 544 - Fall 2017
MapReduce Execution Details

Reduce

Task

Reduce (Shuffle)

Intermediate data goes to local disk: $M \times R$ files (why?)

Output to disk, replicated in cluster

Map

Task

Data not necessarily local

File system: GFS or HDFS

CSEP 544 - Fall 2017
MapReduce Phases

Map Task:
1. Split
2. Record Reader
3. Map
4. Combine

Reduce Task:
1. Copy
2. Sort
3. Reduce

Local storage

HDFS
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

Worker failure:

• Master pings workers periodically,

• If down then reassigns the task to another worker
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 → 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

Worker 1

Worker 2

Worker 3

Backup execution

Straggler

Killed

Killed

CSEP 544 - Fall 2017