Advanced Topics in Data Management

Distributed Query Processing
Horizontal Data Partitioning

**Table**

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>...</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Horizontal Data Partitioning

Table

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>...</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Horizontal Data Partitioning

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>...</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>...</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>...</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>...</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table

R

R1

R2

R3

fragment chunk partition
Horizontal Data Partitioning

• **Block Partition, a.k.a. Round Robin:**
  – 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 \( i = h(t.A) \mod P + 1 \)

• **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 \( v_{i-1} < t.A < v_i \)
Notations

$p = \text{number of servers (nodes) that hold the chunks}$

When a relation $R$ is distributed to $p$ servers, we draw the picture like this:

$R_1 \quad R_2 \quad R_p$

Here $R_1$ is the fragment of $R$ stored on server 1, etc

\[
R = R_1 \cup R_2 \cup \cdots \cup R_p
\]
Uniform Load and Skew

• $|R| = N$ tuples, then $|R_1| + |R_2| + \ldots + |R_p| = N$

• We say the load is uniform when:
  $|R_1| \approx |R_2| \approx \ldots \approx |R_p| \approx N/p$

• Skew means that some load is much larger:
  $\max_i |R_i| >> N/p$

We design algorithms for uniform load, discuss skew later
Parallel Algorithm

- Selection $\sigma$
- Join $\Join$
- Group by $\gamma$
Parallel Selection

Data: \( R(K, A, B, C) \)

Query: \( \sigma_{A=v}(R) \), or \( \sigma_{v_1<A<v_2}(R) \)

- Block partitioned:
- Hash partitioned:
- Range partitioned:
Parallel Selection

Data: \( R(K, A, B, C) \)

Query: \( \sigma_{A=v}(R) \), or \( \sigma_{v1<A<v2}(R) \)

- Block partitioned:
  - All servers need to scan

- Hash partitioned:

- Range partitioned:
Parallel Selection

Data: \( R(K, A, B, C) \)

Query: \( \sigma_{A=v}(R) \), or \( \sigma_{v_1<A<v_2}(R) \)

- Block partitioned:
  - All servers need to scan

- Hash partitioned:
  - Point query: only one server needs to scan
  - Range query: all servers need to scan

- Range partitioned:
Parallel Selection

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

Query: $\sigma_{A=v}(R)$, or $\sigma_{v1<A<v2}(R)$

- Block partitioned:
  - All servers need to scan
- Hash partitioned:
  - Point query: only one server needs to scan
  - Range query: all servers need to scan
- Range partitioned:
  - Only some servers need to scan
Parallel GroupBy

Data: \( R(K, A, B, C) \)

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

Discuss in class how to compute in each case:

- \( R \) is hash-partitioned on \( A \)
- \( R \) is block-partitioned or hash-partitioned on \( K \)
Parallel GroupBy

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

Discuss in class how to compute in each case:

- \( R \) is hash-partitioned on \( A \)
  - Each server \( i \) computes locally \( \gamma_{A, \text{sum}(C)}(R_i) \)
- \( R \) is block-partitioned or hash-partitioned on \( K \)
Parallel GroupBy

Data: \( R(K, A, B, C) \)

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

Discuss in class how to compute in each case:

- \( R \) is hash-partitioned on \( A \)
  - Each server \( i \) computes locally \( \gamma_{A, \text{sum}(C)}(R_i) \)

- \( R \) is block-partitioned or hash-partitioned on \( K \)
  - Need to reshuffle data on \( A \) first (next slide)
  - Then compute locally \( \gamma_{A, \text{sum}(C)}(R_i) \)
Basic Parallel GroupBy

Data: \( R(K, A, B, C) \)

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

- \( R \) is block-partitioned or hash-partitioned on \( K \)
Basic Parallel GroupBy

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 \( A \)

\[ R_1 \quad R_2 \quad \cdots \quad R_p \]
Basic Parallel GroupBy

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 \( A \)

\( R_1' \) \( \quad \) \( R_2' \) \( \quad \) \( \ldots \) \( \quad \) \( R_P' \)
Basic Parallel GroupBy

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 \( A \)

![Diagram showing parallel processing with reshuffling on attribute A]
Basic Parallel GroupBy

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 \( A \)
Basic Parallel GroupBy

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 $A$

This is done in one communication step
Reshuffling

• Nodes send data over the network

• Many-many communications possible

• Throughput:
  – Better than disk
  – Worse than main memory
Basic Parallel GroupBy

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 $A$

This is done in one communication step.

Can you think of an optimization?
### GroupBy/Union Commutativity

<table>
<thead>
<tr>
<th>city</th>
<th>...</th>
<th>qant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seattle</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>LA</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>Seattle</td>
<td></td>
<td>30</td>
</tr>
<tr>
<td>NY</td>
<td></td>
<td>40</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>city</th>
<th>...</th>
<th>qant</th>
</tr>
</thead>
<tbody>
<tr>
<td>LA</td>
<td></td>
<td>22</td>
</tr>
<tr>
<td>NY</td>
<td></td>
<td>33</td>
</tr>
<tr>
<td>LA</td>
<td></td>
<td>44</td>
</tr>
<tr>
<td>Austin</td>
<td></td>
<td>55</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>city</th>
<th>...</th>
<th>qant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seattle</td>
<td></td>
<td>66</td>
</tr>
<tr>
<td>LA</td>
<td></td>
<td>77</td>
</tr>
<tr>
<td>NY</td>
<td></td>
<td>88</td>
</tr>
<tr>
<td>LA</td>
<td></td>
<td>99</td>
</tr>
</tbody>
</table>

```sql
SELECT city, sum(quant)
FROM R
GROUP BY city
```
GroupBy/Union Commutativity

<table>
<thead>
<tr>
<th>city</th>
<th>...</th>
<th>qant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seattle</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>LA</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>Seattle</td>
<td></td>
<td>30</td>
</tr>
<tr>
<td>NY</td>
<td></td>
<td>40</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>city</th>
<th>...</th>
<th>qant</th>
</tr>
</thead>
<tbody>
<tr>
<td>LA</td>
<td></td>
<td>22</td>
</tr>
<tr>
<td>NY</td>
<td></td>
<td>33</td>
</tr>
<tr>
<td>LA</td>
<td></td>
<td>44</td>
</tr>
<tr>
<td>Austin</td>
<td></td>
<td>55</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>city</th>
<th>...</th>
<th>qant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seattle</td>
<td></td>
<td>66</td>
</tr>
<tr>
<td>LA</td>
<td></td>
<td>77</td>
</tr>
<tr>
<td>NY</td>
<td></td>
<td>88</td>
</tr>
<tr>
<td>LA</td>
<td></td>
<td>99</td>
</tr>
</tbody>
</table>

\[
\gamma_{\text{city}, \text{sum}(q)}(R_1 \cup R_2 \cup R_3) =
\]

SELECT city, sum(quant)
FROM R
GROUP BY city
## GroupBy/Union Commutativity

<table>
<thead>
<tr>
<th>city</th>
<th>...</th>
<th>qant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seattle</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>LA</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>Seattle</td>
<td></td>
<td>30</td>
</tr>
<tr>
<td>NY</td>
<td></td>
<td>40</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>city</th>
<th>...</th>
<th>qant</th>
</tr>
</thead>
<tbody>
<tr>
<td>LA</td>
<td></td>
<td>22</td>
</tr>
<tr>
<td>NY</td>
<td></td>
<td>33</td>
</tr>
<tr>
<td>LA</td>
<td></td>
<td>44</td>
</tr>
<tr>
<td>Austin</td>
<td></td>
<td>55</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>city</th>
<th>...</th>
<th>qant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seattle</td>
<td></td>
<td>66</td>
</tr>
<tr>
<td>LA</td>
<td></td>
<td>77</td>
</tr>
<tr>
<td>NY</td>
<td></td>
<td>88</td>
</tr>
<tr>
<td>LA</td>
<td></td>
<td>99</td>
</tr>
</tbody>
</table>

\[
\gamma_{\text{city}, \text{sum}(q)}(\mathcal{R}_1 \cup \mathcal{R}_2 \cup \mathcal{R}_3) = \\
= \gamma_{\text{city}, \text{sum}(q)}(\gamma_{\text{city}, \text{sum}(q)}(\mathcal{R}_1) \cup \gamma_{\text{city}, \text{sum}(q)}(\mathcal{R}_2) \cup \gamma_{\text{city}, \text{sum}(q)}(\mathcal{R}_3))
\]

SELECT city, sum(quant)  
FROM R  
GROUP BY city
Basic Parallel GroupBy

Data: $R(K, A, B, C)$
Query: $\gamma_{A,\text{sum}(C)}(R)$
Basic Parallel GroupBy

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

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

Step 0: [Optimization] each server $i$ computes local group-by:

$$T_i = \gamma_{A, \text{sum}(C)}(R_i)$$
Basic Parallel GroupBy

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

Step 0: [Optimization] each server \( i \) computes local group-by:
\[
T_i = \gamma_{A,\text{sum}(C)}(R_i)
\]

Step 1: partitions tuples in \( T_i \) using hash function \( h(A) \):
\[
T_{i,1}, T_{i,2}, \ldots, T_{i,p}
\]
then send fragment \( T_{i,j} \) to server \( j \)
Basic Parallel GroupBy

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

Step 0: [Optimization] each server $i$ computes local group-by:
$$T_i = \gamma_{A, \text{sum}(C)}(R_i)$$

Step 1: partitions tuples in $T_i$ using hash function $h(A)$:
$$T_{i,1}, T_{i,2}, \ldots, T_{i,p}$$
then send fragment $T_{i,j}$ to server $j$

Step 2: receive fragments, union them, then group-by
$$R'_j = T_{1,j} \cup \ldots \cup T_{p,j}$$
Answer$_j = \gamma_{A, \text{sum}(C)}(R'_j)$
Pushing Aggregates Past Union

Which other rules can we push past union?

• Sum?
• Count?
• Avg?
• Max?
• Median?
Pushing Aggregates Past Union

Which other rules can we push past union?

• Sum?
• Count?
• Avg?
• Max?
• Median?

<table>
<thead>
<tr>
<th>Distributive</th>
<th>Algebraic</th>
<th>Holistic</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{sum}(a_1+a_2+\ldots+a_9) = \text{sum}(\text{sum}(a_1+a_2+a_3) + \text{sum}(a_4+a_5+a_6) + \text{sum}(a_7+a_8+a_9))$</td>
<td>$\text{avg}(B) = \frac{\text{sum}(B)}{\text{count}(B)}$</td>
<td>$\text{median}(B)$</td>
</tr>
</tbody>
</table>
Example Query with Group By

```sql
SELECT a, sum(b) as sb
FROM R WHERE c > 0
GROUP BY a
```
Example Query with Group By

```
SELECT a, sum(b) as sb
FROM R WHERE c > 0
GROUP BY a
```
Example Query with Group By

```
SELECT a, sum(b) as sb
FROM R WHERE c > 0
GROUP BY a
```

\[
\gamma a, \sum(b) \rightarrow sb
\]

\[
\sigma_{c>0}
\]

\[
R
\]
SELECT a, sum(b) as sb FROM R WHERE c > 0 GROUP BY a
SELECT a, sum(b) as sb FROM R WHERE c > 0 GROUP BY a
SELECT a, sum(b) as sb  FROM R  WHERE c > 0 GROUP BY a
SELECT a, sum(b) as sb  FROM R  WHERE c > 0 GROUP BY a
SELECT a, sum(b) as sb FROM R WHERE c > 0 GROUP BY a
SELECT a, sum(b) as sb FROM R WHERE c > 0 GROUP BY a
Speedup and Scaleup

Consider the query $\gamma_{A,\text{sum}(C)}(R)$
Assume the local runtime for group-by is linear $O(|R|)$

If we double number of nodes $P$, what is the runtime?

If we double both $P$ and size of $R$, what is the runtime?
Speedup and Scaleup

Consider the query $\gamma_{A,\text{sum}(C)}(R)$
Assume the local runtime for group-by is linear $O(|R|)$

If we double number of nodes $P$, what is the runtime?
• Half (chunk sizes become $\frac{1}{2}$)

If we double both $P$ and size of $R$, what is the runtime?
• Same (chunk sizes remain the same)
Consider the query $\gamma_{A, \text{sum}(C)}(R)$
Assume the local runtime for group-by is linear $O(|R|)$

If we double number of nodes $P$, what is the runtime?
• Half (chunk sizes become $\frac{1}{2}$)

If we double both $P$ and size of $R$, what is the runtime?
• Same (chunk sizes remain the same)

But only if the data is without skew!
Parallel/Distributed Join

Three “algorithms”:

• Hash-partitioned

• Broadcast

• Combined: “skew-join” or other names
Hash-Partitioned Join, a.k.a. Distributed Join
Hash Join: \( R \bowtie_{A=B} S \)

Data: \( R(A, C), S(B, D) \)

Query: \( R \bowtie_{A=B} S \)

Initially, \( R \) and \( S \) are block partitioned.
Notice: they may be stored in DFS (recall MapReduce)
Some servers hold \( R \)-chunks, some hold \( S \)-chunks, some hold both
Hash Join: $R \bowtie_{A=B} S$

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

Reshuffle $R$ on $R.A$ and $S$ on $S.B$

Initially, $R$ and $S$ are block partitioned. Notice: they may be stored in DFS (recall MapReduce)
Some servers hold $R$-chunks, some hold $S$-chunks, some hold both
Hash Join: $R \bowtie_{A=B} S$

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

Query: $R \bowtie_{A=B} S$

Initially, $R$ and $S$ are block partitioned.

Notice: they may be stored in DFS (recall MapReduce)

Some servers hold $R$-chunks, some hold $S$-chunks, some hold both

Reshuffle $R$ on $R.A$ and $S$ on $S.B$
Hash Join: \( R \bowtie_{A=B} S \)

Data: \( R(A, C), S(B, D) \)

Query: \( R \bowtie_{A=B} S \)

Each server computes the join locally.

Reshuffle R on R.A and S on S.B

Initially, R and S are block partitioned.
Notice: they may be stored in DFS (recall MapReduce).
Some servers hold R-chunks, some hold S-chunks, some hold both.
Hash Join: \( R \bowtie_{A=B} S \)

- **Step 1**
  - Every server holding any chunk of \( R \) partitions its chunk using a hash function \( h(t.A) \)
  - Every server holding any chunk of \( S \) partitions its chunk using a hash function \( h(t.B) \)

- **Step 2:**
  - Each server computes the join of its local fragment of \( R \) with its local fragment of \( S \)
Broadcast Join, a.k.a. Small Join
Broadcast Join

• When joining R and S
• If $|R| >> |S|$
  – Leave R where it is
  – Replicate entire S relation across R-nodes

• Also called a small join or a broadcast join
Query: \( R \bowtie S \)

Broadcast Join

\( R_1 \quad R_2 \quad R_P \quad S \quad \ldots \)
Query: \( R \bowtie S \)
Query: \( R \bowtie S \)

Broadcast Join

Same place...

Keep R in place

Broadcast S

\( R_1 \)  \( R_2 \)  \( R_P \)  \( S \)
Query: $R \bowtie S$

Broadcast Join

- $R_1, S$
- $R_2, S$
- $R_P, S$
- $S$

 Same place...

Keep R in place

Broadcast S

Broadcast S
Discussion

• Hash-join:
  – Both relations are partitioned (good)
  – May have skew (bad)
Discussion

• Hash-join:
  – Both relations are partitioned (good)
  – May have skew (bad)

• Broadcast join
  – One relation must be broadcast (bad)
  – No worry about skew (good)
Discussion

• Hash-join:
  – Both relations are partitioned (good)
  – May have skew (bad)

• Broadcast join
  – One relation must be broadcast (bad)
  – No worry about skew (good)

• Skew join (has other names):
  – Combine both (next)
Skew-Join

Key / foreign-key join: $R(A,B) \bowtie S(B, C)$:

- **Step 1:** fix some large threshold $T$:
  - A value $b$ is called *heavy-hitter* if there are $> T$ tuples with $R.B = b$
  - Let $H = \{b_1, b_2, \ldots\}$ the set of heavy hitters
  - Note that $H$ is small: $H < |R| / T$

- **Step 2:** partitioned join on light hitters

- **Step 3:** broadcast join on heavy hitters
Example Query Execution

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()
```
Order(oid, item, date), Line(item, …)

Query Execution

Node 1

Node 2

Node 3

Node 1

Node 2

Node 3
Query Execution

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

Node 1

hash
h(i.item)
scan Item i

Node 2

hash
h(i.item)
scan Item i

Node 3

hash
h(i.item)
scan Item i

join o.item = i.item
date = today()

Line i

Order o
Query Execution

Node 1

Node 2

Node 3

contains all orders and all lines where hash(item) = 3
contains all orders and all lines where hash(item) = 2
contains all orders and all lines where hash(item) = 1

Order(oid, item, date), Line(item, ...)
Example 2

SELECT *
FROM R, S, T
WHERE R.b = S.c AND S.d = T.e AND (R.a - T.f) > 100
… WHERE R.b = S.c AND S.d = T.e AND (R.a - T.f) > 100
... WHERE R.b = S.c AND S.d = T.e AND (R.a - T.f) > 100
... WHERE R.b = S.c AND S.d = T.e AND (R.a - T.f) > 100
... WHERE R.b = S.c AND S.d = T.e AND (R.a - T.f) > 100

σ_{R.a - T.f > 100}

Machine 1
1/3 of R, S, T

Machine 2
1/3 of R, S, T

Machine 3
1/3 of R, S, T
... WHERE R.b = S.c AND S.d = T.e AND (R.a - T.f) > 100

\[ \sigma_{R.a - T.f > 100} \]

Broadcasting S and T

Machine 1

1/3 of R, S, T

Machine 2

1/3 of R, S, T

Machine 3

1/3 of R, S, T
Skew
Skew

• Skew means that one server runs much longer than the other servers

• Reasons:
  – Computation skew
  – Data skew
Computation Skew

- All workers receive the same amount of input data, but some need to run much longer than others
- E.g. perform some image processing whose runtimes depends on the image
- Solution: use virtual servers
Virtual Servers

Main idea:

• If we send the data uniformly to the P servers, and one of them is stuck with the complicated image, then we have skew

• Solution: pretend we have many “virtual” servers. (Next slide.)
Virtual Servers

Large number $P_v$ of “virtual servers”

• Design algorithm for $P_v$ virtual servers

• Scale down to $P << P_v$ physical servers, by simulating them round-robin

E.g. MapReduce: $P=$workers, $P_v=$map tasks
Data Skew

• We fail to distribute the data uniformly to the servers
• Question: why can this happen?
Data Skew

• We fail to distribute the data uniformly to the servers
• Question: why can this happen?
• Answer:
  – Range partition may have many more tuples in one bucket than another
  – Hash partition may suffer from heavy hitters