Decomposition

Same as in centralized system

1. Normalization
2. Eliminating redundancy
3. Algebraic rewriting

Normalization

Convert from general language to a standard form
E.g., relational algebra
Normalization

Example:

```
select R.A, R.C
from R, S
where R.A = S.A and
  ((R.B = 1 and S.D = 2) or
   (R.C > 3 and S.D = 2))
```

Normalization

Also detect invalid expressions

```
select *
from R
```

✘ R does not have D attribute

where R.D = 3

Eliminating Redundancy

E.g., in conditions

```
(S.A = 1) \land (S.A > 5) \Rightarrow \text{false}
```

```
(S.A < 10) \land (S.A < 5) \Rightarrow S.A < 5
```

Eliminating Redundancy

E.g., sharing common sub-expressions
**Algebraic Rewriting**

E.g., pushing conditions down

\[ \sigma_{\text{cond}} \bowtie R \bowtie \sigma_{\text{cond}1} \bowtie \sigma_{\text{cond}2} \]

**Query Processing**

**Decomposition** ✓

→ One or more algebraic query trees on relations

**Localization**

Replacing relations by corresponding fragments

**Localization**

Steps

1. Start with query
2. Replace relations by fragments
3. Push \( \cup \) up and \( \Pi \), \( \sigma \) down (apply CS 245 rules)
4. Simplify ~ eliminate unnecessary operations

**Localization**

Notation for fragment

\([R : \text{cond}]\)

conditions its tuples satisfy

fragment
Example A

(1) \[ \sigma_{E \leq 3} \]

\[ \left\{ \begin{array}{c}
R_1 : E < 10 \\
R_2 : E \geq 10 \end{array} \right. \]

Example A

(2) \[ \sigma_{E \leq 3} \]

\[ \left\{ \begin{array}{c}
R_1 : E < 10 \\
R_2 : E \geq 10 \end{array} \right. \]

\[ \Rightarrow \emptyset \]

Example A

(3) \[ \sigma_{E \leq 3} \]

\[ \left\{ \begin{array}{c}
R_1 : E < 10 \\
R_2 : E \geq 10 \end{array} \right. \]

\[ \Rightarrow \emptyset \]
**Example A**

(4)

\[ \sigma_{E \geq 3} \]

\[ [R: E < 10] \]

---

**Localization**

**Rule 1**

\[ \sigma_{c1} [R: c2] \implies \sigma_{c1} [R: c1 \land c2] \]

\[ [R: \text{false}] \implies \emptyset \]

---

**Example A**

\[ \sigma_{E=3} [R_2: E \geq 10] \implies \sigma_{E=3} [R_2: E=3 \land E \geq 10] \]

\[ \implies \sigma_{E=3} [R_2: \text{false}] \]

\[ \implies \emptyset \]

---

**Example B**

(1)

\[ A \]

\[ R \quad S \]

\[ A = \text{common attribute} \]
Localization

Rule 2

\[ [R: c_1] \Join_A [S: c_2] \implies [R \Join_A S: c_1 \land c_2 \land R.A = S.A] \]

\[ [R: \text{false}] \implies \emptyset \]

Example B

\[ [R_1: A < 5] \Join_A [S_2: A \geq 5] \]

\[ \implies [R_1 \Join_A S_2: R_1.A < 5 \land S_2.A \geq 5 \land R_1.A = S_2.A] \]

\[ \implies [R_1 \Join_A S_2: \text{false}] \]

\[ \implies \emptyset \]

Example C

(2)

\[ [R_2: A < 10] \]

\[ [R_2: A \geq 10] \]

\[ [S_2: K=R.K \land R.A < 10] \]

\[ [S_2: K=R.K \land R.A \geq 10] \]

\[ [K: \text{derived fragmentation}] \]

Example C

(3)

\[ [R_1] \]

\[ [S_1] \]

\[ [K] \]

\[ [K] \]

\[ [K] \]

\[ [K] \]

\[ [K] \]

\[ [K] \]
Example C

(3)

Example C

(4)

Example C

Example C

\[ [R_1: A < 10] \bowtie [S_2: K = R.K \land R.A \geq 10] \]

\[ \Rightarrow [R_1 \bowtie S_2: R_1.A < 10 \land S_2.K = R.K \land R.A \geq 10 \land R_1.K = S_2.K] \]

\[ \Rightarrow [R_1 \bowtie S_2: \text{false}] \]

\[ \Rightarrow \emptyset \]

Example D

(1)

\[ \Pi_A \]

\[ R \]

\[ \left\{ \begin{array}{l} R_1 (K, A, B) \\ R_2 (K, C, D) \end{array} \right\} \]

vertical fragmentation
**Rule 3**

Consider the vertical fragmentation

\[ R_i = \Pi_{A_i}(R), A_i \subseteq A \]

Then for any \( B \subseteq A \)

\[ \Pi_B(R) = \Pi_B(\cap \forall R_i | B \cap A_i \neq \emptyset) \]
Query Processing

Decomposition ✔
Localization ✔

Optimization
Overview
Joins and other operations
Inter-operation parallelism
Optimization strategies

Overview

Optimization process
1. Generate query plans
2. Estimate size of intermediate results
3. Estimate cost of plan
4. Pick minimum

Parallel/Distributed Sort

Differences from centralized optimization
New strategies for some operations
- Parallel/distributed sort
- Parallel/distributed join
  - Semi-join
  - Privacy preserving join
- Duplicate elimination
- Aggregation
- Selection
Many ways to assign and schedule processors

Parallel/Distributed Sort

Input
a) Relation \( R \) on single site/disk
b) Relation \( R \) fragmented/partitioned by sort attribute
c) Relation \( R \) fragmented/partitioned by another attribute
Parallel/Distributed Sort

Output
a) Sorted $R$ on single site/disk
b) Sorted $R$ fragments/partitions

Basic Sort

Algorithm
1. Sort each fragment independently
2. Ship results if necessary

Basic Sort

$R(K, ...) \text{ to be sorted on } K$
Horizontally fragmented on $K$ using vector $(k_0, k_1, \ldots, k_n)$
Range-Partition Sort

R(K, ...) to be sorted on K
R located at one or more site/disk, not fragmented on K

Partition Vectors

Problem
Select a good partition vector given fragments

| 7 | 31 | 10 |
| 22 | 8 | 12 |
| 11 | 15 | 4 |
| 14 | 11 | |
| 52 | 32 | |
| 17 | | |

R_a  R_b  R_c

Range-Partition Sort

Algorithm
1. Range partition on K
2. Basic sort

Example approach

Each site sends to coordinator
- MIN sort key
- MAX sort key
- Number of tuples

Coordinator
- Computes vector and distributes to sites
- Decides the number of sites to perform local sorts
Partition Vectors

Sample scenario
Coordinator receives

$S_A$: MIN = 5  MAX = 9  # of tuples = 10
$S_B$: MIN = 7  MAX = 16  # of tuples = 10

Notes 3

Expected tuples assuming we want to sort at 2 sites

$1 = \frac{10}{(16 - 7 + 1)}$
$2 = \frac{10}{(9 - 5 + 1)}$

Sample scenario
Expected tuples

Expected tuples with key < $k_0$ = half of total tuples
$2(k_0 - 5) + (k_0 - 7) = 10$
$k_0 = \frac{(10 + 10 + 7)}{3} = 9$

Notes 3

Variations
Send more info to coordinator:

a) Partition vector for local site

b) Histogram

Notes 3

# of tuples  local vector
Partition Vectors

Variations
Multiple rounds
1. Sites send range and # of tuples to coordinator
2. Coordinator distributes preliminary vector \( V_0 \)
3. Sites send coordinator # of tuples in each \( V_0 \) range
4. Coordinator computes and distributes final vector \( V_f \)

Parallel External Sort-Merge

Same as range-partition sort except does sorting first

Parallel/Distributed Join

Input
Relations \( R, S \)
May or may not be fragmented

Output
\( R \bowtie S \)
Result at one or more sites
**Partitioned Join (Equijoin)**

*Same partition function* $f$ *is used for both* $R$ *and* $S$.

$f$ *can be range or hash partitioning.*

Local join can be of any type (use any CS 245 optimization).

Various scheduling options, e.g.,

a) Partition $R$; partition $S$; join

b) Partition $R$; build local hash table for $R$; partition $S$ and join

---

**Partitioned Join (Equijoin)**

We already know why it works.

Goal is to make all $|R_i| + |S_i|$ the same size.

Sometimes we partition just to make this join possible.
Asymmetric Fragment + Replicate Join

Can use any partition function $f$ for $R$ (even round robin)

Can do any join, not just equijoin (e.g., $R \bowtie_{R.A=S.B} S$)

General Fragment + Replicate Join

$f$ partition $\rightarrow$ 3 fragments

$n$ copies of each fragment

S is partitioned in similar fashion
General Fragment + Replicate Join

The asymmetric F+R join is a special case of the general F+R join. Asymmetric F+R may be good if S is small. It also works on non-equijoins.

Semi-Join

Goal: reduce communication traffic

R \bowtie S \Rightarrow (R \bowtie S) \bowtie S \\
\text{or} \\
R \bowtie (S \bowtie R) \\
\text{or} \\
(R \bowtie S) \bowtie (S \bowtie R)
### Semi-Join

**R ∞ S**

<table>
<thead>
<tr>
<th>R</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>A: 2</td>
<td>3</td>
</tr>
<tr>
<td>B: 10</td>
<td>y</td>
</tr>
<tr>
<td>C: 25</td>
<td>z</td>
</tr>
<tr>
<td>D: 30</td>
<td>w</td>
</tr>
</tbody>
</table>

\( \Pi_{A,R} = \{2,10,25,30\} \)

**Transmitted data**

With **semi-join** \( R ∞ (S ∇ R) \): \( T = 4 |A| + 2|A + C| + \text{result} \)

With **join** \( R ∇ S \): \( T = 4 |A + B| + \text{result} \)

**Assume** \( R \) is the smaller relation

Then, in general,

\( (R \bowtie S) \bowtie S \) is better than \( (R \bowtie S) \) if

\[ \text{size}(\Pi_{A,S}) + \text{size}(R \bowtie S) < \text{size}(R) \]

Can do similar comparison for other joins

→ Only taking into account transmission cost
Semi-Join

Trick: encode $\Pi_A S$ (or $\Pi_A R$) as a bit vector

Keys in $S$:

| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |

one bit/possible key

Semi-Join

Useful for three way joins $R \bowtie S \bowtie T$

Semi-Join

Useful for three way joins $R \bowtie S \bowtie T$

Option 1:

$R' \bowtie S' \bowtie T$ where $R' = R \bowtie S$

$S' = S \bowtie T$

Option 2:

$R'' \bowtie S'' \bowtie T$ where $R'' = R \bowtie S''$

$S'' = S \bowtie T$
Semi-Join

Useful for three way joins $R \bowtie S \bowtie T$

Option 1: $R' \bowtie S' \bowtie T$ where $R' = R \bowtie S$
   $S' = S \bowtie T$

Option 2: $R^* \bowtie S^* \bowtie T$ where $R^* = R \bowtie S'$
   $S^* = S \bowtie T$

→ Many options ~ exponential in # of relations

Privacy Preserving Join

Semi-join won’t work

If site 1 sends $\prod_A R$ to site 2, site 2 learns all keys of $R$

Privacy Preserving Join

Fix: send hash keys only

Site 1 hashes each value of $A$ before sending

Site 2 hashes its own $A$ values (same $h$) to see what tuples match
Privacy Preserving Join

What is the problem?

\[ R = \{ h(a), h(a), h(a), h(a) \} \]

site 2 sees it has \( h(a) \), \( h(a) \)

\[ (a, c), (a, c) \]

site 1

Privacy Preserving Join

Our adversary model: honest but curious

Dictionary attack is possible (cheating is internal and can’t be caught)

Sending incorrect keys not possible (cheater could be caught)

Privacy Preserving Join

What is the problem?

\[ R = \{ h(a), h(a), h(a), h(a) \} \]

site 2 sees it has \( h(a) \), \( h(a) \)

\[ (a, c), (a, c) \]

site 1

Dictionary attack
Site 2 can take all keys \( a_1 \), \( a_2 \), \( a_3 \), ... and checks if \( h(a_1) \), \( h(a_2) \), \( h(a_3) \), ...
matches what site 1 sent

Privacy Preserving Join

One solution

Information Sharing Across Private Databases. Agrawal et al., SIGMOD 2003

→ Use commutative encryption functions

\[ E_i(x) = x \text{ encrypted using a key private to site } i \]

\[ E_i(E_j(x)) = E_j(E_i(x)) \]

Shorthand: \( E_i(x) \) is \( \overline{x} \), \( E_j(x) \) is \( \overline{x} \), \( E_i(E_j(x)) \) is \( \overline{x} \)
Privacy Preserving Join

One solution

CS 347 Notes 3

Computes $(a_1, a_3, a_5, a_7)$, intersects with $(a_1, a_2, a_3, a_4)$

Other Privacy Preserving Operations

Inequality join $R_{R.A > S.A}$

Similarity join $R_{sim(R.A, S.A)}$

Other Parallel Operations

Duplicate elimination
Sort first (in parallel) then eliminate duplicates in result
Partition tuples (range or hash) and eliminate locally

Aggregates
Partition by grouping attributes; compute aggregate locally

Parallel/Distributed Aggregates

\[ \sum_{dept} \text{salary} \]
Parallel/Distributed Aggregates

**R_a**
<table>
<thead>
<tr>
<th>id</th>
<th>dept</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>toy</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>toy</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>sales</td>
<td>15</td>
</tr>
</tbody>
</table>

**R_b**
<table>
<thead>
<tr>
<th>id</th>
<th>dept</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>sales</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>toy</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td>mgmt</td>
<td>15</td>
</tr>
<tr>
<td>7</td>
<td>sales</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>mgmt</td>
<td>30</td>
</tr>
</tbody>
</table>

sum (sal) group by dept

Parallel/Distributed Aggregates

**R_a**
<table>
<thead>
<tr>
<th>id</th>
<th>dept</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>toy</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>toy</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>sales</td>
<td>15</td>
</tr>
</tbody>
</table>

**R_b**
<table>
<thead>
<tr>
<th>id</th>
<th>dept</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>sales</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>toy</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td>mgmt</td>
<td>15</td>
</tr>
<tr>
<td>7</td>
<td>sales</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>mgmt</td>
<td>30</td>
</tr>
</tbody>
</table>

sum (sal) group by dept with less data
Parallel/Distributed Aggregates

Enhancement
Perform aggregate during partition to reduce data transmitted
→ Does not work for all aggregate functions—which ones?

Parallel/Distributed Selection

Straightforward if one can leverage range or hash partitioning

How do indexes work?
Parallel/Distributed Selection
Partition vector can act as the root of a distributed index

Parallel/Distributed Selection
Distributed indexes on a non-partition attributes get complicated

Parallel/Distributed Selection
Indexing schemes
Which one is better in a distributed environment?
How to make updates and expansion efficient?
Where to store the directory and set of participants?
Is global knowledge necessary?

→ If the index is not too big, it may be better to keep it whole and replicate it

Query Processing
Decomposition ✔
Localzation ✔

Optimization
Overview ✔
Joins and other operations ✔
Inter-operation parallelism
Optimization strategies
**Inter-Operation Parallelism**

Pipelined
Independent

**Pipelined Parallelism**

Pipelining cannot be used in all cases

**Independent Parallelism**

1. $\text{temp}_1 \leftarrow R \bowtie S$
2. $\text{temp}_2 \leftarrow T \bowtie V$
3. result $\leftarrow \text{temp}_1 \bowtie \text{temp}_2$
Summary

As we consider query plans for optimization, we must consider various new strategies for

individual operations

scheduling operations