Topics Now (Index Wrap-Up)

• Using Indices in SQL
• Multi-key Index Structures

Reading Chapter 5

• Read
  – 5.1, and 5.3.1 and 5.3.2
Index definition in SQL

• Create index name on rel (attr)
• Create unique index name on rel (attr) → defines candidate key

• Drop INDEX name
Note  CANNOT SPECIFY TYPE OF INDEX
       (e.g. B-tree, Hashing, ...)
OR PARAMETERS
       (e.g. Load Factor, Size of Hash,...)

... at least in SQL...
Note: ATTRIBUTE LIST $\Rightarrow$ MULTIKEY INDEX (next)

e.g., `CREATE INDEX foo ON R(A,B,C)"
Multi-key Index

Motivation: Find records where

DEPT = “Toy” AND SAL > 50k
Strategy I:

- Use one index, say Dept.
- Get all Dept = “Toy” records and check their salary
Strategy II:

- Use 2 Indexes; Manipulate Pointers

Toy $\rightarrow$ [Diagram with pointers] $\leftarrow$ Sal $> 50k$
Strategy III:

- Multiple Key Index

One idea:
Example

Dept Index

<table>
<thead>
<tr>
<th>Art</th>
<th>10k</th>
<th>12k</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sales</td>
<td>15k</td>
<td>15k</td>
</tr>
<tr>
<td>Toy</td>
<td>17k</td>
<td>15k</td>
</tr>
<tr>
<td></td>
<td>21k</td>
<td>19k</td>
</tr>
</tbody>
</table>

Salary Index

<table>
<thead>
<tr>
<th></th>
<th>10k</th>
<th>15k</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>15k</td>
<td>15k</td>
</tr>
<tr>
<td></td>
<td></td>
<td>19k</td>
</tr>
</tbody>
</table>

Example Record

Name=Joe
DEPT=Sales
SAL=15k
For which queries is this index good?

- Find RECs Dept = “Sales” ∧ SAL=20k
- Find RECs Dept = “Sales” ∧ SAL ≥ 20k
- Find RECs Dept = “Sales”
- Find RECs SAL = 20k
The BIG picture....

- Chapters 2 & 3: Storage, records, blocks...
- Chapter 4 & 5: Access Mechanisms
  - Indexes
  - B trees
  - Hashing
  - Multi key
- Chapter 6 & 7: Query Processing
Query Processing

Query in SQL

→

Query Plan in Algebra
Example

Data:
relation R (A, B, C)
relation S (C, D, E)

Query:
SELECT  B, D
FROM  R, S
WHERE  R.A = “c” and  S.E = 2 and  R.C=S.C
SELECT B, D FROM R, S WHERE R.A = "c" and S.E = 2 and R.C=S.C

<table>
<thead>
<tr>
<th>R</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>S</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>x</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>y</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>2</td>
<td>10</td>
<td>30</td>
<td>30</td>
<td>z</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>2</td>
<td>35</td>
<td>40</td>
<td>40</td>
<td>x</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>3</td>
<td>45</td>
<td>50</td>
<td>50</td>
<td>y</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

Answer

<table>
<thead>
<tr>
<th>B</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>x</td>
</tr>
</tbody>
</table>
How do we execute query?

One idea

- Form **Cartesian product** of all tables in FROM-clause
- **Select** tuples that match WHERE-clause
- **Project** columns that occur in SELECT-clause
### Table: R X S

<table>
<thead>
<tr>
<th>R.A</th>
<th>R.B</th>
<th>R.C</th>
<th>S.C</th>
<th>S.D</th>
<th>S.E</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>10</td>
<td>10</td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>a</td>
<td>1</td>
<td>10</td>
<td>20</td>
<td>y</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
<td>10</td>
<td>x</td>
<td>2</td>
</tr>
</tbody>
</table>

Bingo! Got one...
But?

- Performance would be unacceptable!
- We need a better approach to reasoning about queries, their execution orders and their respective costs
Formal Relational Query Languages

- Two mathematical Query Languages form basis for “real” languages (e.g. SQL), and for implementation:
  - *Relational Algebra*: More operational, very useful for representing execution plans.
  - *Relational Calculus*: Lets users describe what they want, rather than how to compute it. (Non-operational, *declarative*.)*
Relational Algebra

- **Tuple**: ordered set of data values
- **Relation**: a set of tuples
- **Algebra**: formal mathematical system consisting of a set of objects and operations on those objects
- **Relational algebra**: Algebra whose objects are relations and operators transform relations into other relations
Relational Algebra - can be used to describe plans...

Ex: Plan I

\[ \Pi_{B,D} \sigma_{R.A=\text{"c"} \land S.E=2 \land R.C=S.C} R \times S \]

OR: \[ \Pi_{B,D} [ \sigma_{R.A=\text{"c"} \land S.E=2 \land R.C = S.C} (R \times S)] \]
Example Instances

“Sailors”
and
“Reserves”
relations

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

S1

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>yuppy</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

S2
Relational Algebra

- **Basic operations:**
  - *Selection* (σ) Selects a subset of rows from relation.
  - *Projection* (π) Deletes unwanted columns from relation.
  - *Cross-product* (×) Allows us to combine two relations.
  - *Set-difference* (−) Tuples in reln. 1, but not in reln. 2.
  - *Union* (∪) Tuples in reln. 1 and in reln. 2.

- **Additional operations:**
  - Intersection, *join*, division, renaming: Not essential, but (very!) useful.

- Since each operation returns a relation, operations can be *composed*! (Algebra is “closed”.)
Projection

- Deletes attributes not in projection list.
- Schema of result contains exactly fields in projection list.

- Projection operator has to eliminate duplicates! (Why??)
  - Note: real systems typically don’t do duplicate elimination unless user explicitly asks for it. (Why not?)

<table>
<thead>
<tr>
<th>sname</th>
<th>rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>yuppy</td>
<td>9</td>
</tr>
<tr>
<td>lubber</td>
<td>8</td>
</tr>
<tr>
<td>guppy</td>
<td>5</td>
</tr>
<tr>
<td>rusty</td>
<td>10</td>
</tr>
</tbody>
</table>

\[ \pi_{\text{sname, rating}}(S2) \]

<table>
<thead>
<tr>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>35.0</td>
</tr>
<tr>
<td>55.5</td>
</tr>
</tbody>
</table>

\[ \pi_{\text{age}}(S2) \]
Selection

- Selects rows that satisfy selection condition.
- No duplicates in result! (Why?)
- Schema of result identical to schema of (only) input relation.
- Result relation can be input for another relational algebra operation! (Operator composition.)
Union, Intersection, Set Difference

- Take two input relations, which must be **union-compatible**:
  - Same number of fields.
  - `Corresponding` fields have same type.

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</td>
</tr>
<tr>
<td>28</td>
<td>yuppy</td>
<td>9</td>
<td>35.0</td>
</tr>
</tbody>
</table>

\[ S_1 \cup S_2 \]

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

\[ S_1 - S_2 \]
Cross-Product

- Each row of S1 is paired with each row of R1.
- *Result schema* has one field per field of S1 and R1, with field names `inherited` if possible

<table>
<thead>
<tr>
<th>(sid)</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
<th>(sid)</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

**Conflict:** Both S1 and R1 have a field called *sid*.

**Renaming operator:** \( \rho (C(1 \rightarrow \text{sid}1, 5 \rightarrow \text{sid}2), S1 \times R1) \)
Joins

- **Condition Join**: \( R \bowtie_c S = \sigma_c (R \times S) \)

<table>
<thead>
<tr>
<th>(sid)</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
<th>(sid)</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

\( S1 \bowtie S1.sid < R1.sid \ R1 \)

- **Result schema** same as that of cross-product.
- Fewer tuples than cross-product \( \rightarrow \) more efficient?
- Sometimes called a **theta-join**.
Joins

- **Equi-Join**: A special case of condition join where condition $c$ contains only *equalities*.
  
<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

  \[ S_1 \bowtie_{\text{sid}} R_1 \]

- **Result schema** similar to cross-product, but only one copy of fields for which equality is specified.

- **Natural Join**: Equijoin on *all* common fields.
Division

• Not primitive operator, but useful:

Find sailors who have reserved all boats.

• A has 2 fields x and y; B has only field y:
  - \( A/B = \{ \langle x \rangle | \exists \langle x, y \rangle \in A \ \forall \langle y \rangle \in B \} \)
  - i.e., \( A/B \) contains all \( x \) tuples (sailors) such that for every \( y \) tuple (boat) in \( B \), there is an \( xy \) tuple in \( A \).
Examples of Division $A/B$

- **A**:
  - $s1$: p1, p2, p3, p4
  - $s2$: p1, p2
  - $s3$: p2
  - $s4$: p2, p4

- **A/B1**:
  - $pno$
    - p2

- **B1**:
  - $sno$
    - s1
    - s2

- **B2**:
  - $pno$
    - p2
    - p4

- **B3**:
  - $sno$
    - s1
    - s2
    - s3
    - s4
Expressing A/B Using Basic Operators

• Division is useful shorthand.
• Idea: For $A/B$, compute all $x$ values that are not `disqualified’ by some $y$ value in $B$.
  - $x$ value is disqualified if by attaching $y$ value from $B$, we obtain an $xy$ tuple that is not in $A$.

Disqualified $x$ values: $\pi_x (\pi_x(A) \times B) - A$

$A/B$: $\pi_x(A)$ — all disqualified tuples
Find names of sailors who’ve reserved boat #103

Solution 1: \( \pi_{\text{sname}}((\sigma_{\text{bid}=103} \text{Reserves}) \bowtie \text{Sailors}) \)

- Solution 2: \( \rho (\text{Temp1, } \sigma_{\text{bid}=103} \text{Reserves}) \)
  \( \rho (\text{Temp2, Temp1 } \bowtie \text{Sailors}) \)
  \( \pi_{\text{sname}}(\text{Temp2}) \)

- Solution 3: \( \pi_{\text{sname}}(\sigma_{\text{bid}=103}(\text{Reserves } \bowtie \text{Sailors})) \)
Find names of sailors who’ve reserved a red boat

- Information about boat color only available in Boats; so need an extra join:

\[
\pi_{sname}(\sigma_{\text{color}=\text{red}}(\text{Boats} \bowtie \text{Reserves}\bowtie \text{Sailors}))
\]

- A more efficient solution:

\[
\pi_{sname}(\pi_{sid}(\pi_{bid}(\sigma_{\text{color}=\text{red}}(\text{Boats} \bowtie \text{Res})\bowtie \text{Sailors})))
\]

A query optimizer can find this, given the first solution!
Find sailors who’ve reserved a red or a green boat

• Can identify all red or green boats, then find sailors who’ve reserved one of these boats:

\[
\rho \left( \sigma_{\text{color} = 'red' \lor \text{color} = 'green'} (\text{Tempboats}) \right) \\
\pi_{\text{sname}} (\text{Tempboats} \bowtie \text{Reserves} \bowtie \text{Sailors})
\]

• Can also define Tempboats using union! (How?)

• What happens if \( \lor \) is replaced by \( \land \) in this query?
Find sailors who’ve reserved a red and a green boat

- Must identify sailors who’ve reserved red boats, sailors who’ve reserved green boats, then find the intersection (sid is a key for Sailors):

\[
\rho (\text{Tempred}, \pi_{sid}((\sigma_{\text{color} = 'red'} \text{Boats}) \bowtie \text{Reserves}))
\]

\[
\rho (\text{Tempgreen}, \pi_{sid}((\sigma_{\text{color} = 'green'} \text{Boats}) \bowtie \text{Reserves}))
\]

\[
\pi_{sname}((\text{Tempred} \cap \text{Tempgreen}) \bowtie \text{Sailors})
\]
Find names of sailors who’ve reserved all boats

- Uses division; schemas of input relations to / must be carefully chosen:
  \[
  \rho (\text{Tempsids}, (\pi_{\text{sid,bid}} \text{Reserves}) / (\pi_{\text{bid}} \text{Boats}))
  \]
  \[
  \pi_{\text{sname}} (\text{Tempsids} \bowtie \text{Sailors})
  \]

- To find sailors who’ve reserved all ‘Interlake’ boats:
  \[
  \ldots / \pi_{\text{bid}} (\sigma_{\text{bname} = 'Interlake'} \text{Boats})
  \]
Relational Algebra - can be used to describe plans...

Ex: Plan I

\[ \Pi_{B,D} \sigma_{R.A = "c" \land S.E = 2 \land R.C = S.C} (R \times S) \]

OR: \[ \Pi_{B,D} [ \sigma_{R.A = "c" \land S.E = 2 \land R.C = S.C} (R \times S) ] \]