Database Management Systems
CSEP 544

Lecture 5: SQL++
Query Execution and Optimization
Announcements

• Please use the correct tags for your HW / RA!
  – We will start deducting points / not grade them.

• HW4 due today

• HW5 released
  – Please start early!
  – Use “hw5” / “asterixdb” tag to ask questions on Piazza

• Two lectures next week (Tues and Thurs)

• Today:
  – AsterixDB / SQL++ (wrap up)
  – RDBMS implementation and query optimization
A Case Study: AsterixDB
JSON - Overview

• JavaScript Object Notation = lightweight text-based open standard designed for human-readable data interchange. Interfaces in C, C++, Java, Python, Perl, etc.

• The filename extension is .json.

We will emphasize JSon as semi-structured data
JSON Semantics: a Tree!

```json
{
    "person": [
        {
            "name": "Mary",
            "address": {
                "street": "Maple",
                "no": 345,
                "city": "Seattle"
            }
        },
        {
            "name": "John",
            "address": "Thailand",
            "phone": 2345678
        }
    ]
}
```
Mapping Relational Data to JSON

Person

<table>
<thead>
<tr>
<th>name</th>
<th>phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>3634</td>
</tr>
<tr>
<td>Sue</td>
<td>6343</td>
</tr>
<tr>
<td>Dirk</td>
<td>6363</td>
</tr>
</tbody>
</table>

```
{  
  "person":  
  [{  
    "name": "John",  
    "phone": 3634
  },  
  {  
    "name": "Sue",  
    "phone": 6343
  },  
  {  
    "name": "Dirk",  
    "phone": 6383
  }] 
}
```
Asterix Data Model (ADM)

- **Objects:**
  - \{“Name”: “Alice”, “age”: 40\}
  - Fields must be distinct:
    - \{“Name”: “Alice”, “age”: 40, “age”:50\}

- **Arrays:**
  - [1, 3, “Fred”, 2, 9]
  - Note: can be heterogeneous

- **Multisets:**
  - \{{1, 3, “Fred”, 2, 9}\}
Examples

Try these queries:

```
SELECT x.age FROM [{'name': 'Alice', 'age': ['30', '50']}];
```

```
SELECT x.age FROM {{ {'name': 'Alice', 'age': ['30', '50']} }} x;
```

```
-- error
SELECT x.age FROM {'name': 'Alice', 'age': ['30', '50']} x;
```

Can only select from multi-set or array
SQL++ Overview

```
SELECT ... FROM ... WHERE ... [GROUP BY ...]
```
Retrieve Everything

SELECT x.mondial FROM world x;

Answer

```json
{"mondial":
{"country": [country1, country2, ...],
 "continent": [...],
 "organization": [...],
 ...
 ...
}

{"mondial":
{"country": [country1, country2, ...],
 "continent": [...],
 "organization": [...],
 ...
 ...
}
```
Retrieve countries

```
{
  "mondial":
  {
    "country": [country1, country2, ...],
    "continent": [...],
    "organization": [...],
    ...
  }
}
```

Answer

```
{
  "country": [country1, country2, ...],
}
```

```
SELECT x.mondial.country FROM world x;
```
Retrieve countries, one by one

```sql
SELECT y AS country FROM world x, x.mondial.country y;
```

Answer

```
country1
country2
...
```
Heterogeneous Collections

SELECT z.name as province_name, u.name as city_name
FROM world x, x.mondial.country y, y.province z,
(CASE WHEN z.city is missing THEN []
    WHEN is_array(z.city) THEN z.city
    ELSE [z.city] END) u
WHERE y.name='Greece';
Useful Functions

- is_array
- is_boolean
- is_number
- is_object
- is_string
- is_null
- is_missing
- is_unknown = is_null or is_missing
Useful Idioms

• Unnesting
• Nesting
• Group-by / aggregate
• Join
• Multi-value join
Basic Unnesting

- An array: \([a, b, c]\)
- A nested array: \(arr = [[a, b], [], [b, c, d]]\)
- Unnest(arr) = [a, b, b, c, d]

```
SELECT y
FROM arr x, x y
```
Unnesting Specific Field

A nested collection

\[
\text{coll = } \left\{ \begin{array}{l}
\{ \text{A: a1, F: \{B: b1, \{B: b2\}\}, G: \{C: c1\}} \}, \\
\{ \text{A: a2, F: \{B: b3, \{B: b4, \{B: b5\}\}, G: [ ]\}}, \\
\{ \text{A: a3, F: \{B: b6\}, G: \{C: c2, \{C: c3\}\}} \} \\
\end{array} \right. 
\]
Unnesting Specific Field

A nested collection

coll =
 [{A:a1, F:{{B:b1}, {B:b2}}, G:{{C:c1}}},
  {A:a2, F:{{B:b3}, {B:b4}, {B:b5}}, G:[]},
  {A:a3, F:{{B:b6}}, G:{{C:c2},{C:c3}}}],

Unnest_F(coll) =
 [{A:a1, {B:b1}, G:{{C:c1}}},
  {A:a1, {B:b2}, G:{{C:c1}}},
  {A:a2, {B:b3}, G:[]},
  {A:a2, {B:b4}, G:[]},
  {A:a2, {B:b5}, G:[]},
  {A:a3, {B:b6}, G:{{C:c2},{C:c3}}}]
Unnesting Specific Field

A nested collection

\[
\text{coll = } \left\{ \begin{array}{l}
{A:a1, \{B:b1\}, \{B:b2\}, \{C:c1\}}, \\
{A:a2, \{B:b3\}, \{B:b4\}, \{B:b5\}}, \\
{A:a3, \{B:b6\}, \{C:c2\}, \{C:c3\}}
\end{array} \right\}
\]

\[
\text{Unnest}_F(\text{coll}) =
\left\{ \begin{array}{l}
{A:a1, \{B:b1\}, \{C:c1\}}, \\
{A:a2, \{B:b3\}, \{C:c1\}}, \\
{A:a2, \{B:b4\}}, \\
{A:a2, \{B:b5\}}, \\
{A:a3, \{B:b6\}, \{C:c2\}, \{C:c3\}}
\end{array} \right\}
\]

\[
\text{SELECT } x.A, y.B, x.G \\
\text{FROM coll } x, x.F y
\]

Nested Relational Algebra

SQL++

Refers to relations defined on the left
Unnesting Specific Field

A nested collection

```
coll =

[A:a1, F:{B:b1},{B:b2}, G:{C:c1}],
{A:a2, F:{B:b3},{B:b4},{B:b5}, G:[ ]},
{A:a3, F:{B:b6}, G:{C:c2},{C:c3}}]
```

\( \text{Unnest}_F(\text{coll}) = \)

```
[A:a1, {B:b1}, G:{C:c1}],
{A:a1, {B:b2}, G:{C:c1}],
{A:a2, {B:b3}, G:[]},
{A:a2, {B:b4}, G:[]},
{A:a2, {B:b5}, G:[]},
{A:a3, {B:b6}, G:{C:c2},{C:c3}]]
```

```
SELECT x.A, y.B, x.G
FROM coll x, x.F y
= 
SELECT x.A, y.B, x.G
FROM coll x
UNNEST x.F y
```
Unnesting Specific Field

A nested collection

\[
coll = \\
[A:a1, F:[{B:b1},{B:b2}], G:[{C:c1}]], \\
{A:a2, F:[{B:b3},{B:b4},{B:b5}], G:[ ]}, \\
{A:a3, F:[{B:b6}], G:[{C:c2},{C:c3}]]}
\]

\[
Unnest_F(coll) = \\
[A:a1, {B:b1}, G:[{C:c1}]], \\
{A:a1, {B:b2}, G:[{C:c1}]], \\
{A:a2, {B:b3}, G:[]}, \\
{A:a2, {B:b4}, G:[]}, \\
{A:a2, {B:b5}, G:[]}, \\
{A:a3, {B:b6}, G:[{C:c2},{C:c3}]]}
\]

\[
SELECT x.A, y.B, x.G \\
FROM coll x, x.F y
\]
**Unnesting Specific Field**

A nested collection

\[
\text{coll} = \\
\{\{A:a1, F:\{B:b1, B:b2\}, G:\{C:c1\}\}, \\
{A:a2, F:\{B:b3, B:b4, B:b5\}, G:\[]\}, \\
{A:a3, F:\{B:b6\}, G:\{C:c2, C:c3\}\}}
\]

\[\text{Unnest}_F(\text{coll}) = \\
\{A:a1, B:b1, G:\{C:c1\}\}, \\
{A:a1, B:b2, G:\{C:c1\}\}, \\
{A:a2, B:b3, G:\[]\}, \\
{A:a2, B:b4, G:\[]\}, \\
{A:a2, B:b5, G:\[]\}, \\
{A:a3, B:b6, G:\{C:c2, C:c3\}\}
\]

\[\text{Unnest}_G(\text{coll}) = \\
\{A:a1, F:\{B:b1, B:b2\}, C:c1\}, \\
{A:a3, F:\{B:b6\}, C:c2\}, \\
{A:a3, F:\{B:b6\}, C:c3\}
\]

SELECT x.A, y.B, x.G
FROM coll x, x.F y

SELECT x.A, x.F, z.C
FROM coll x, x.G z

Nested Relational Algebra

SQL++
Nesting (like group-by)

A flat collection

coll =
[{{A:a1, B:b1}, {A:a1, B:b2}, {A:a2, B:b1}}]
Nesting (like group-by)

A flat collection

\[
\text{coll} = \\
\{\{A:a1, B:b1\}, \{A:a1, B:b2\}, \{A:a2, B:b1\}\}
\]

\[
\text{Nest}_A(\text{coll}) = \\
\{\{A:a1, \text{GRP:}\{B:b1, B:b2\}\}\} \\
\{\{A:a2, \text{GRP:}\{B:b2\}\}\}
\]
Nesting (like group-by)

A flat collection

\[ \text{coll} = \{\{A:a1, B:b1\}, \{A:a1, B:b2\}, \{A:a2, B:b1\}\} \]

\[ \text{Nest}_A(\text{coll}) = \{\{A:a1, \text{GRP}:\{B:b1, B:b2\}\}, \{A:a2, \text{GRP}:\{B:b2\}\}\} \]

\[ \text{Nest}_B(\text{coll}) = \{\{B:b1, \text{GRP}:\{A:a1, A:a2\}\}, \{B:b2, \text{GRP}:\{A:a1\}\}\} \]
Nesting (like group-by)

A flat collection

\[
\text{coll} = \\
\{\{A:a1, B:b1\}, \{A:a1, B:b2\}, \{A:a2, B:b1\}\}
\]

\[
\text{Nest}_A(\text{coll}) = \\
\{\{A:a1, \text{GRP}:[\{B:b1\}, \{B:b2\}]\}, \{A:a2, \text{GRP}:[\{B:b2\}]\}\}
\]

\[
\text{Nest}_B(\text{coll}) = \\
\{\{B:b1, \text{GRP}:[\{A:a1\}, \{A:a2\}]\}, \{B:b2, \text{GRP}:[\{A:a1\}]\}\}
\]

SELECT DISTINCT \(x.A\), 
(SELECT \(y.B\) FROM \(coll\ y\) WHERE \(x.A = y.A\)) as \(\text{GRP}\) FROM \(coll\ x\)
Nesting (like group-by)

A flat collection

\[
\text{coll} = \{\{A:a1, B:b1\}, \{A:a1, B:b2\}, \{A:a2, B:b1\}\}
\]

\[\text{Nest}_A(\text{coll}) = \{\{A:a1, \text{GRP}:[\{B:b1\}, \{B:b2\}]\}, \{A:a2, \text{GRP}:[\{B:b2\}]\}\}\]

\[\text{Nest}_B(\text{coll}) = \{\{B:b1, \text{GRP}:[\{A:a1\}, \{A:a2\}]\}, \{B:b2, \text{GRP}:[\{A:a1\}]\}\}\]

\[
\text{SELECT DISTINCT} \ x.A, \quad (\text{SELECT} \ y.B \ \text{FROM} \ \text{coll} \ y \ \text{WHERE} \ x.A = y.A) \ \text{as} \ \text{GRP}
\]

\text{FROM} \ \text{coll} \ x

\[
\text{SELECT DISTINCT} \ x.A, \ g \ \text{as} \ \text{GRP}
\]

\text{FROM} \ \text{coll} \ x

\text{LET} \ g = (\text{SELECT} \ y.B \ \text{FROM} \ \text{coll} \ y \ \text{WHERE} \ x.A = y.A)
Group-by / Aggregate

A nested collection

coll =
[[{A:a1, F: [{B:b1}, {B:b2}], G: [{C:c1}]}],
 {A:a2, F: [{B:b3}, {B:b4}, {B:b5}], G: [ ]},
 {A:a3, F: [{B:b6}], G: [{C:c2}, {C:c3}]}]

Count the number of elements in the F collection
Group-by / Aggregate

A nested collection

```
coll = 
[{A:a1, F:[{B:b1},{B:b2}], G:[{C:c1}]},
 {A:a2, F:[{B:b3},{B:b4},{B:b5}], G:[ ]},
 {A:a3, F:[{B:b6}], G:[{C:c2},{C:c3}]}]
```

Count the number of elements in the F collection

```
SELECT x.A, COLL_COUNT(x.F) as cnt
FROM coll x
```
A nested collection

coll =
\[
\{\{A:a1, F:\{\{B:b1\}, \{B:b2\}\}, G:\{\{C:c1\}\}\}, \\
\{A:a2, F:\{\{B:b3\}, \{B:b4\}, \{B:b5\}\}, G:\[]\}, \\
\{A:a3, F:\{\{B:b6\}\}, G:\{\{C:c2\}, \{C:c3\}\}\}\}
\]

Count the number of elements in the F collection

\[
SELECT x.A, COLL\_COUNT(x.F) \text{ as } cnt \\
FROM coll x
\]

\[
SELECT x.A, COUNT(*) \text{ as } cnt \\
FROM coll x, x.F y \\
GROUP BY x.A
\]

These are NOT equivalent! (Why?)
## Group-by / Aggregate

<table>
<thead>
<tr>
<th>Function</th>
<th>NULL</th>
<th>MISSING</th>
<th>Empty Collection</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLL_COUNT</td>
<td>counted</td>
<td>counted</td>
<td>0</td>
</tr>
<tr>
<td>COLL_SUM</td>
<td>returns NULL</td>
<td>returns NULL</td>
<td>returns NULL</td>
</tr>
<tr>
<td>COLL_MAX</td>
<td>returns NULL</td>
<td>returns NULL</td>
<td>returns NULL</td>
</tr>
<tr>
<td>COLL_MIN</td>
<td>returns NULL</td>
<td>returns NULL</td>
<td>returns NULL</td>
</tr>
<tr>
<td>COLL_AVG</td>
<td>returns NULL</td>
<td>returns NULL</td>
<td>returns NULL</td>
</tr>
<tr>
<td>ARRAY_COUNT</td>
<td>not counted</td>
<td>not counted</td>
<td>0</td>
</tr>
<tr>
<td>ARRAY_SUM</td>
<td>ignores NULL</td>
<td>ignores NULL</td>
<td>returns NULL</td>
</tr>
<tr>
<td>ARRAY_MAX</td>
<td>ignores NULL</td>
<td>ignores NULL</td>
<td>returns NULL</td>
</tr>
<tr>
<td>ARRAY_MIN</td>
<td>ignores NULL</td>
<td>ignores NULL</td>
<td>returns NULL</td>
</tr>
<tr>
<td>ARRAY_AVG</td>
<td>ignores NULL</td>
<td>ignores NULL</td>
<td>returns NULL</td>
</tr>
</tbody>
</table>

**Lesson:** Read the *#$@## manual!!**
Joint

Two flat collection

\[
coll1 = \{\{A:a1, B:b1\}, \{A:a1, B:b2\}, \{A:a2, B:b1\}\}
coll2 = \{\{B:b1,C:c1\}, \{B:b1,C:c2\}, \{B:b3,C:c3\}\}
\]

\[
\begin{align*}
\text{SELECT } & x.A, x.B, y.C \\
\text{FROM } & coll1 x, coll2 y \\
\text{WHERE } & x.B = y.B
\end{align*}
\]
Behind the Scences

Query Processing on NFNF data:

• Option 1: give up on query plans, use standard java/python-like execution

• Option 2: represent the data as a collection of flat tables, convert SQL++ to a standard relational query plan
Flattening SQL++ Queries

A nested collection

coll =
{{A:a1, F:[{B:b1},{B:b2}], G:[{C:c1}]},
  {A:a2, F:[{B:b3},{B:b4},{B:b5}], G:[ ]},
  {A:a1, F:[{B:b6}], G:[{C:c2},{C:c3}]}}
Flattening SQL++ Queries

A nested collection

coll =

[{A:a1, F: [B:b1, B:b2], G: [C:c1]},
{A:a2, F: [B:b3, B:b4, B:b5], G: []},
{A:a1, F: [B:b6], G: [C:c2, C:c3]}]
Flattening SQL++ Queries

A nested collection

coll =

\[
\begin{align*}
\{ & \{ A: a1, F: \{ B: b1, b2 \}, G: \{ C: c1 \} \}, \\
& \{ A: a2, F: \{ B: b3, b4, b5 \} \}, \\
& \{ A: a1, F: \{ B: b6 \}, G: \{ C: c2, c3 \} \} \}
\end{align*}
\]

Flat Representation

coll:

<table>
<thead>
<tr>
<th>id</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a1</td>
</tr>
<tr>
<td>2</td>
<td>a2</td>
</tr>
<tr>
<td>3</td>
<td>a1</td>
</tr>
</tbody>
</table>

F

<table>
<thead>
<tr>
<th>parent</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>b1</td>
</tr>
<tr>
<td>1</td>
<td>b2</td>
</tr>
<tr>
<td>2</td>
<td>b3</td>
</tr>
<tr>
<td>2</td>
<td>b4</td>
</tr>
<tr>
<td>2</td>
<td>b5</td>
</tr>
<tr>
<td>3</td>
<td>b6</td>
</tr>
</tbody>
</table>

g

<table>
<thead>
<tr>
<th>parent</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>c1</td>
</tr>
<tr>
<td>3</td>
<td>c2</td>
</tr>
<tr>
<td>3</td>
<td>c3</td>
</tr>
</tbody>
</table>

SELECT x.A, y.B
FROM coll x, x.F y
WHERE x.A = 'a1'
### Flattening SQL++ Queries

#### A nested collection

```
coll = 
[{A:a1, F:[{B:b1},{B:b2}], G:[]},
 {A:a2, F:[{B:b3},{B:b4},{B:b5}], G:[]},
 {A:a1, F:[{B:b6}], G:[]}]
```

#### Flat Representation

<table>
<thead>
<tr>
<th>id</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a1</td>
<td>b1</td>
<td>c1</td>
</tr>
<tr>
<td>2</td>
<td>a2</td>
<td>b2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>a1</td>
<td>b6</td>
<td></td>
</tr>
</tbody>
</table>

#### SQL++

```sql
SELECT x.A, y.B
FROM coll x, x.F y
WHERE x.A = 'a1'
```

#### SQL

```sql
SELECT x.A, y.B
FROM coll x, F y
WHERE x.id = y.parent and x.A = 'a1'
```
Flattening SQL++ Queries

A nested collection

```
coll = [{A:a1, F:{B:b1}, B:b2}, {A:a2, F:{B:b3}, B:b4}, {A:a1, F:{B:b6}}]
```

Flat Representation

<table>
<thead>
<tr>
<th></th>
<th>id</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>a2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>a1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>parent</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>b1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>b2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>b3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>b4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>b5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>parent</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>c1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>c2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>c3</td>
<td></td>
</tr>
</tbody>
</table>

SQL++

```sql
SELECT x.A, y.B
FROM coll x, x.F y
WHERE x.A = ‘a1’
```

SQL

```sql
SELECT x.A, y.B
FROM coll x, F y
WHERE x.A = ‘a1’
```

```sql
SELECT x.A, y.B
FROM coll x, x.F y, x.G z
WHERE y.B = z.C
```

```sql
SELECT x.A, y.B
FROM coll x, F y, G z
WHERE y.B = z.C
WHERE z.C = ‘a1’
```
Flattening SQL++ Queries

A nested collection

```
coll = 
[{A:a1, F:{B:b1},{B:b2}}, G:{C:c1}],
{A:a2, F:{B:b3},{B:b4},{B:b5}}, G:[ ]},
{A:a1, F:{B:b6}}, G:{C:c2},{C:c3}]}
```

Flat Representation

```
coll:  F               G
      id  A  parent  B     parent  C
     1   a1  1      b1      1      c1
     2   a2  1      b2      3      c2
     3   a1  2      b3
        2      b4
        2      b5
        3      b6
```

SQL++

```
SELECT x.A, y.B
FROM coll x, x.F y
WHERE x.A = 'a1'
```

SQL

```
SELECT x.A, y.B
FROM coll x, F y
WHERE x.id = y.parent and x.A = 'a1'
```

```
SELECT x.A, y.B
FROM coll x, F y, G z
WHERE y.B = z.C
```

```
SELECT x.A, y.B
FROM coll x, F y, G z
WHERE x.id = y.parent and x.id = z.parent
and y.B = z.C
```
Conclusion

- Semistructured data best suited for *data exchange*
- For quick, ad-hoc data analysis, use a native query language: SQL++, or AQL, or XQuery
  - Modern, advanced query processors like AsterixDB / SQL++ can process semistructured data as efficiently as RDBMS
- For long term data analysis: spend the time and effort to normalize it, then store in a RDBMS
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
Query Evaluation Steps Review

1. Parse & Rewrite Query
2. Select Logical Plan
3. Select Physical Plan
4. Query Execution

Disk

Logical plan (RA)
Physical plan
Logical vs Physical Plans

• Logical plans:
  – Created by the parser from the input SQL text
  – Expressed as a relational algebra tree
  – Each SQL query has many possible logical plans

• Physical plans:
  – Goal is to choose an efficient implementation for each operator in the RA tree
  – Each logical plan has many possible physical plans
SELECT sname 
FROM Supplier x, Supply y 
WHERE x.sid = y.sid 
  and y.pno = 2 
  and x.scity = 'Seattle' 
  and x.sstate = 'WA'

Relational algebra expression is also called the "logical query plan"
A physical query plan is a logical query plan annotated with physical implementation details.

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

Physical Query Plan 1

(On the fly) \(\pi_{sname}\)

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

(Nested loop)

Supplier (File scan)

Supply (File scan)
Physical Query Plan 2

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

Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)
Physical Query Plan 3

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

Different but equivalent logical query plan; different physical plan

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

• For each SQL query... many logical plans

• For each logical plan... many physical plans

• Next: we will discuss physical operators; how exactly are query executed?
Query Execution
Implementing Query Operators with the Iterator Interface

Each operator implements three methods:

- `open()`
- `next()`
- `close()`
Implementing Query Operators with the Iterator Interface

Example “on the fly” selection operator

```
interface Operator {

void open (...);
// calls next () on its inputs
// processes an input tuple
// produces output tuple(s)
Tuple next ();
// cleans up (if any)
void close ();
}
```
Implementing Query Operators with the Iterator Interface

Example “on the fly” selection operator

```java
interface Operator {

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

}
```
Implementing Query Operators with the Iterator Interface

Example “on the fly” selection operator

```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 ();

    void close ();
}
```

Implementing Query Operators with the Iterator Interface

Example “on the fly” selection operator

```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 ();
}
```

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

Example “on the fly” selection operator

```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 ();
}

class Select implements Operator {...

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

    Tuple next () {
        boolean found = false;
        while (!found) {
            Tuple in = child.next ();
            if (in == EOF) return EOF;
            found = p(in);
        }
        return in;
    }

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

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

Example “on the fly” selection operator

```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 ();
}

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);
        }
    }
}
```
Implementing Query Operators with the Iterator Interface

Example “on the fly” selection operator

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 ();
}

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 ();
}
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 ();
}

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

```
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();
```
Supplier\(\langle \text{sid, sname, scity, sstate} \rangle\)
Supply\(\langle \text{sid, pno, quantity} \rangle\)

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

\(\text{Nested loop}\)

\(\text{Suppliers } (\text{File scan})\)
\(\text{Supplies } (\text{File scan})\)

Discuss: open/next/close for nested loop join
Suppliers

\( \text{Suppliers} = \text{Suppliers} \cap \text{scity} = \text{Seattle} \) and \( \text{sstate} = \text{WA} \) and \( pno = 2 \)

\( \Pi_{\text{sname}} \)

\( \text{open()} \)

\( \text{Suppliers} \)

\( \text{Supplies} \)

(On the fly)

(Nested loop)

(On the fly)

Discuss: open/next/close for nested loop join

Suppliers (File scan)

Supplies (File scan)

CSEP 544 - Fall 2017 74
Suppliers (sid, sname, scity, sstate)
Supply(sid, pno, quantity)

Pipelining

(On the fly) 

(On the fly) 

(Nested loop) 

σ_{scity='Seattle' and sstate='WA' and pno=2}

sno = sno

π_{sname}

open() 

open()

open()

Discuss: open/next/close for nested loop join

Suppliers (File scan)

Supplies (File scan)
Suppliers
Supply(sid, pno, quantity)

(Pipelining)

\[ \text{Suppliers} \]
\[ \text{Supply} \]

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

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

\[ \text{open()} \]

\[ \text{sno} = \text{sno} \]

\[ \text{Suppliers (File scan)} \]
\[ \text{Supplies (File scan)} \]

Discuss: open/next/close for nested loop join
Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

(On the fly) Supplier(sname)

(On the fly) Supply(pno)

(Nested loop) snos = sno σ scity = 'Seattle' and sstate = 'WA' and pno = 2

Discuss: open/next/close for nested loop join
Pipelining

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

(On the fly)

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

(On the fly)

\( \sigma_{\text{scity} = \text{Seattle} \text{ and sstate} = \text{WA} \text{ and pno} = 2} \)

(Nested loop)

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

Suppliers (File scan)

\( \text{open()} \)

Supplies (File scan)

\( \text{open()} \)

\( \text{open()} \)

Discuss: open/next/close for nested loop join
Suppliers

\( \text{Supplier}(\text{sid}, \text{sname}, \text{scity}, \text{sstate}) \)

Supply\((\text{sid, pno, quantity})\)

**Pipelining**

\( \text{next()} \)

\( \prod_{\text{sname}} \)

(On the fly)

\( \sigma_{\text{scity}=\text{Seattle} \text{ and sstate}=\text{WA} \text{ and pno}=2} \)

(Nested loop)

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

Suppliers

(File scan)

Supplies

(File scan)

Discuss: open/next/close for nested loop join
Suppliers

Supplies

\( \text{Suppliers}(\text{sid}, \text{sname}, \text{scity}, \text{sstate}) \)

\( \text{Supply}(\text{sid}, \text{pno}, \text{quantity}) \)

**Pipelining**

(On the fly)

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

(On the fly)

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

(Nested loop)

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

Suppliers

(File scan)

(On the fly)

next()

(On the fly)

next()

(next())

(next())

Next()

Discuss: open/next/close for nested loop join
Suppliers

(On the fly)

(On the fly)

(Suppliers

(File scan)

(Supplies

(File scan)

\(n_{\text{name}}\))

next()

next()

next()

next()

Discuss: open/next/close for nested loop join
Suppliers

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

Supplies

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

\( \text{Suppliers} \) (File scan)

\( \text{Supplies} \) (File scan)

\( \text{Supplier}(\text{sid, sname, scity, sstate}) \)

\( \text{Supply}(\text{sid, pno, quantity}) \)

Discuss: open/next/close for nested loop join

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

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

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

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

Pipelining

(On the fly)

(On the fly)

(Nested loop)

\(\Pi_{\text{sname}}\)

\(\sigma_{\text{scity} = \text{Seattle} \text{ and sstate} = \text{WA} \text{ and } pno=2}\)

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

next()

next()

next()

next()

next()

(On the fly)

(On the fly)

(next)

Discuss: open/next/close for nested loop join

Suppliers
(File scan)

Supplies
(File scan)
Pipelining

Suppliers

Supply(sid, pno, quantity)

(Nested loop)

(On the fly)

Discuss: open/next/close for nested loop join
Suppliers
(Suppliers)
(File scan)

Supply(sno = sno)

σ scity = 'Seattle' and sstate = 'WA' and pno = 2

π sname

(Hash Join)

(On the fly)

Discuss hash-join in class
Suppliers

\[ \sigma_{\text{scity} = 'Seattle' \text{ and } \text{sstate} = 'WA' \text{ and } \text{pno} = 2}^{\text{sno} = \text{sno}} \]

Tuples from here are pipelined

Suppliers

(File scan)

\[ \pi_{\text{sname}} \]

Supplies

(File scan)

Discuss hash-join in class

Pipelining

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

\[ \text{Suppliers} \]

(On the fly)

\[ \text{(File scan)} \]

\[ \text{Suppliers} \]

(On the fly)

\[ \text{(File scan)} \]

\[ \text{Supplies} \]

(Hash Join)

\[ \text{Supplies} \]

(On the fly)

\[ \text{(File scan)} \]

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

\[ \pi_{\text{sname}} \]

\[ \text{sno = sno} \]

Discuss hash-join in class

Tuples from here are “blocked”

Tuples from here are pipelined
Blocked Execution

\[ \text{Suppliers} = \sigma_{\text{scity}= 'Seattle' \text{ and } \text{sstate}= 'WA' \text{ and } \text{pno}=2} \left( \text{Supplier} \right) \]

\[ \text{Supplies} = \pi_{\text{sname}} \left( \text{Supply} \left( \text{sno} = \text{sno} \right) \right) \]

(On the fly)

Discuss merge-join in class

(Merge Join)

Suppliers (File scan)

Supplies (File scan)
Blocked Execution

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

(On the fly)

\[
\sigma_{\text{scity}=\text{‘Seattle’} \text{ and sstate}=\text{‘WA’} \text{ and } \text{pno}=2}
\]

(On the fly)

(Merge Join)

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

\[
\text{sno} = \text{sno}
\]

Blocked

Suppliers

(File scan)

Blocked

Supplies

(File scan)

Discuss merge-join in class
Pipelined Execution

- Tuples generated by an operator are immediately sent to the parent

- Benefits:
  - No operator synchronization issues
  - No need to buffer tuples between operators
  - Saves cost of writing intermediate data to disk
  - Saves cost of reading intermediate data from disk

- This approach is used whenever possible
Query Execution Bottom Line

• SQL query transformed into **physical plan**
  – **Access path selection** for each relation
    • Scan the relation or use an index (next lecture)
  – **Implementation choice** for each operator
    • Nested loop join, hash join, etc.
  – **Scheduling decisions** for operators
    • Pipelined execution or intermediate materialization

• Pipelined execution of physical plan
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
Query Performance

• My database application is too slow… why?
• One of the queries is very slow… why?

• To understand performance, we need to understand:
  – How is data organized on disk
  – How to estimate query costs

  – In this course we will focus on disk-based DBMSs
Data Storage

- DBMSs store data in **files**
- Most common organization is row-wise storage
- On disk, a file is split into blocks
- Each block contains a set of tuples

In the example, we have 4 blocks with 2 tuples each

<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>
<tr>
<td>50</td>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>200</td>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>220</td>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>420</td>
<td>…</td>
<td>…</td>
</tr>
<tr>
<td>800</td>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>
Data File Types

The data file can be one of:

- **Heap file**
  - Unsorted

- **Sequential file**
  - Sorted according to some attribute(s) called *key*

<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>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The data file can be one of:

- **Heap file**
  - Unsorted

- **Sequential file**
  - Sorted according to some attribute(s) called *key*

Note: *key* here means something different from primary key: it just means that we order the file according to that attribute. In our example we ordered by **ID**. Might as well order by **fName**, if that seems a better idea for the applications running on our database.
Index

• An **additional** file, that allows fast access to records in the data file given a search key
Index

• An **additional** file, that allows fast access to records in the data file given a search key

• The index contains (key, value) pairs:
  – The key = an attribute value (e.g., student ID or name)
  – The value = a pointer to the record
Index

- An **additional** file, that allows fast access to records in the data file given a search key.
- The index contains (key, value) pairs:
  - The key = an attribute value (e.g., student ID or name)
  - The value = a pointer to the record
- Could have many indexes for one table

Key = means here search key
This Is Not A Key

Different keys:

• **Primary key** – uniquely identifies a tuple
• **Key of the sequential file** – how the data file is sorted, if at all
• **Index key** – how the index is organized
Example 1: Index on ID

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>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Example 2: Index on fName

Index `Student_fName` on `Student.fName`

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>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

CSEP 544 - Fall 2017
Index Organization

We need a way to represent indexes after loading into memory so that they can be used

Several ways to do this:

• Hash table

• B+ trees – most popular
  – They are search trees, but they are not binary instead have higher fanout
  – Will discuss them briefly next

• Specialized indexes: bit maps, R-trees, inverted index
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

\[ 40 \leq 80 \]

\[ 20 < 40 \leq 60 \]

\[ 30 < 40 \leq 40 \]

\[ 80 \]
Clustered vs Unclustered Index entries

Data Records

CLUSTERED

B+ Tree

Index entries
(Index File)
(Data file)

UNCLUSTERED

Data Records

B+ Tree

Every table can have only one clustered and many unclustered indexes.

Why?
Index Classification

- **Clustered/unclustered**
  - Clustered = records close in index are close in data
    - Option 1: Data inside data file is sorted on disk
    - Option 2: Store data directly inside the index (no separate files)
  - Unclustered = records close in index may be far in data
Index Classification

- **Clustered/unclustered**
  - Clustered = records close in index are close in data
    - Option 1: Data inside data file is sorted on disk
    - Option 2: Store data directly inside the index (no separate files)
  - Unclustered = records close in index may be far in data

- **Primary/secondary**
  - Meaning 1:
    - Primary = is over attributes that include the primary key
    - Secondary = otherwise
  - Meaning 2: means the same as clustered/unclustered
Index Classification

• **Clustered/unclustered**
  – Clustered = records close in index are close in data
    • Option 1: Data inside data file is sorted on disk
    • Option 2: Store data directly inside the index (no separate files)
  – Unclustered = records close in index may be far in data

• **Primary/secondary**
  – Meaning 1:
    • Primary = is over attributes that include the primary key
    • Secondary = otherwise
  – Meaning 2: means the same as clustered/unclustered

• **Organization** B+ tree or Hash table
Scanning a Data File

- Disks are mechanical devices!
  - Technology from the 60s; density much higher now
- Read only at the rotation speed!
- Consequence:
  Sequential scan is MUCH FASTER than random reads
  - Good: read blocks 1, 2, 3, 4, 5, ...
  - Bad: read blocks 2342, 11, 321, 9, ...
- Rule of thumb:
  - Random reading 1-2% of the file ≈ sequential scanning the entire file; this is decreasing over time (because of increased density of disks)
- Solid state (SSD): $$$ expensive; put indexes, other “hot” data there, not enough room for everything (NO LONGER TRUE)
Assume the database has indexes on these attributes:

- `index_takes_courseID` = index on Takes.courseID
- `index_student_ID` = index on Student.ID

```sql
SELECT *
FROM Student x, Takes y
WHERE x.ID = y.studentID AND y.courseID > 300
```

```
for y in Takes
    if courseID > 300 then
        for x in Student
            if x.ID = y.studentID
                output *
```

```
for y in index_Takes_courseID where y.courseID > 300
    for x in Student where x.ID = y.studentID
    output *
```
SELECT *
FROM Student x, Takes y
WHERE x.ID=y.studentID AND y.courseID > 300

for y in Takes
  if courseID > 300 then
    for x in Student
      if x.ID=y.studentID
        output *

Assume the database has indexes on these attributes:
- \texttt{index\_takes\_courseID} = index on Takes.courseID
- \texttt{index\_student\_ID} = index on Student.ID

for y in \texttt{index\_Takes\_courseID} where y.courseID > 300
  for x in Student where x.ID = y.studentID
  output *
Example

Assume the database has indexes on these attributes:
- `index_takes_courseID = index on Takes.courseID`
- `index_student_ID = index on Student.ID`

```sql
SELECT *
FROM Student x, Takes y
WHERE x.ID = y.studentID AND y.courseID > 300
```

```
for y in Takes
  if courseID > 300 then
    for x in Student
      if x.ID = y.studentID
        output *
```

```sql
for y in index_Takes_courseID where y.courseID > 300
  for x in Student where x.ID = y.studentID
    output *
```
Creating Indexes in SQL

CREATE TABLE V(M int, N varchar(20), P int);

CREATE INDEX V1 ON V(N)

CREATE INDEX V2 ON V(P, M)

CREATE INDEX V3 ON V(M, N)

CREATE UNIQUE INDEX V4 ON V(N)

CREATE CLUSTERED INDEX V5 ON V(N)
Which Indexes?

- How many indexes **could** we create?
- Which indexes **should** we create?
Which Indexes?

• How many indexes **could** we create?

• Which indexes **should** we create?

<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>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

*In general this is a very hard problem*
Which Indexes?

• The *index selection problem*
  – Given a table, and a “workload” (big Java application with lots of SQL queries), decide which indexes to create (and which ones NOT to create!)

• Who does index selection:
  – The database administrator DBA
  – Semi-automatically, using a database administration tool
Which Indexes?

• The *index selection problem*
  – Given a table, and a “workload” (big Java application with lots of SQL queries), decide which indexes to create (and which ones NOT to create!)

• Who does index selection:
  – The database administrator DBA
  – Semi-automatically, using a database administration tool
Index Selection: Which Search Key

• Make some attribute $K$ a search key if the WHERE clause contains:
  – An exact match on $K$
  – A range predicate on $K$
  – A join on $K$
The Index Selection Problem 1

V(M, N, P);

Your workload is this

100000 queries:

SELECT * 
FROM V 
WHERE N=?

100 queries:

SELECT * 
FROM V 
WHERE P=?
The Index Selection Problem 1

V(M, N, P);

Your workload is this

100000 queries:

100 queries:

SELECT *
FROM V
WHERE N=?

SELECT *
FROM V
WHERE P=?

What indexes?
The Index Selection Problem 1

V(M, N, P);

Your workload is this

100000 queries:

SELECT *
FROM V
WHERE N=?

100 queries:

SELECT *
FROM V
WHERE P=?

A: V(N) and V(P) (hash tables or B-trees)
The Index Selection Problem 2

V(M, N, P);

Your workload is this

100000 queries:
SELECT * FROM V WHERE N>? and N<?

100 queries:
SELECT * FROM V WHERE P=?

100000 queries:
INSERT INTO V VALUES (?, ?, ?)

What indexes?
The Index Selection Problem 2

```
V(M, N, P);
```

Your workload is this

100000 queries:
```
SELECT * FROM V WHERE N>? and N<?
```

100 queries:
```
SELECT * FROM V WHERE P=?
```

100000 queries:
```
INSERT INTO V VALUES (?, ?, ?)
```

A: definitely V(N) (must B-tree); unsure about V(P)
Two typical kinds of queries

- Point queries
  - What data structure should be used for index?

- Range queries
  - What data structure should be used for index?
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
To Cluster or Not

- Range queries benefit mostly from clustering
- Covering indexes do \textit{not} need to be clustered: they work equally well unclustered
SELECT * FROM R WHERE R.K>? and R.K<?
SELECT * 
FROM R 
WHERE R.K>? and R.K<?
SELECT * 
FROM R 
WHERE R.K>? and R.K<?
SELECT *  
FROM R  
WHERE R.K>? and R.K<?
Choosing Index is Not Enough

• To estimate the cost of a query plan, we still need to consider other factors:
  – How each operator is implemented
  – The cost of each operator
  – Let’s start with the basics
Cost of Reading Data From Disk
Cost Parameters

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

- Parameters:
  - \( B(R) \) = # of blocks (i.e., pages) for relation R
  - \( T(R) \) = # of tuples in relation R
  - \( V(R, a) \) = # 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

- \( A = c \)  
  
  \[ /* \sigma_{A=c}(R) */ \]
  
  \[ \text{Selectivity} = 1/V(R,A) \]

- \( A < c \)
  
  \[ /* \sigma_{A<c}(R) */ \]
  
  \[ \text{Selectivity} = (c - \min(R, A))/(\max(R,A) - \min(R,A)) \]

- \( c_1 < A < c_2 \)
  
  \[ /* \sigma_{c_1<A<c_2}(R) */ \]
  
  \[ \text{Selectivity} = (c_2 - c_1)/(\max(R,A) - \min(R,A)) \]
Cost of Reading Data From Disk

• Sequential scan for relation R costs $B(R)$

• Index-based selection
  – Estimate selectivity factor $X$ (see previous slide)
  – Clustered index: $X \cdot B(R)$
  – Unclustered index $X \cdot T(R)$

Note: we ignore I/O cost for index pages
Index Based Selection

• Example:

<table>
<thead>
<tr>
<th>B(R) = 2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>T(R) = 100,000</td>
</tr>
<tr>
<td>V(R, a) = 20</td>
</tr>
</tbody>
</table>

• Table scan:

• Index based selection:

\[
\text{cost of } \sigma_{a=v}(R) = ?
\]
Index Based Selection

- Example:
  - Table scan: $B(R) = 2,000$ I/Os
  - Index based selection:

<table>
<thead>
<tr>
<th>$B(R)$</th>
<th>$T(R)$</th>
<th>$V(R, a)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>100,000</td>
<td>20</td>
</tr>
</tbody>
</table>

\[ \text{cost of } \sigma_{a=v}(R) = ? \]
Index Based Selection

- Example:
  - $B(R) = 2000$
  - $T(R) = 100,000$
  - $V(R, a) = 20$

- Table scan: $B(R) = 2,000$ I/Os

- Index based selection:
  - If index is clustered:
  - If index is unclustered:

$\text{cost of } \sigma_{a=v}(R) = ?$
Index Based Selection

- Example:
  \[
  \begin{align*}
  \text{B}(R) &= 2000 \\
  \text{T}(R) &= 100,000 \\
  \text{V}(R, a) &= 20
  \end{align*}
  \]

  \[\text{cost of } \sigma_{a=v}(R) = ?\]

- Table scan: B(R) = 2,000 I/Os

- Index based selection:
  - If index is clustered: B(R) * \(1/V(R,a)\) = 100 I/Os
  - If index is unclustered:
Index Based Selection

- Example:
  - Table scan: $B(R) = 2000$ I/Os
  - Index based selection:
    - If index is clustered: $B(R) \times \frac{1}{V(R, a)} = 100$ I/Os
    - If index is unclustered: $T(R) \times \frac{1}{V(R, a)} = 5,000$ I/Os

<table>
<thead>
<tr>
<th>$B(R)$</th>
<th>2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(R)$</td>
<td>100,000</td>
</tr>
<tr>
<td>$V(R, a)$</td>
<td>20</td>
</tr>
</tbody>
</table>

Cost of $\sigma_{a=v}(R) =$ ?
Index Based Selection

- Example:
  - Table scan: $B(R) = 2000$ I/Os
  - Index based selection:
    - If index is clustered: $B(R) \times 1/V(R, a) = 100$ I/Os
    - If index is unclustered: $T(R) \times 1/V(R, a) = 5,000$ I/Os

- Table scan: $B(R) = 2,000$ I/Os

Lesson: Don’t build unclustered indexes when $V(R, a)$ is small!
Cost of Executing Operators (Focus on Joins)
Outline

• **Join operator algorithms**
  – One-pass algorithms (Sec. 15.2 and 15.3)
  – Index-based algorithms (Sec 15.6)

• **Note about readings:**
  – In class, we discuss only algorithms for joins
  – Other operators are easier: read the book
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 $B(R) \leq M$
  - $M = \text{number of memory pages available}$
# Hash Join Example

Patient(pid, name, address)

Insurance(pid, provider, policy_nb)

Patient \( \bowtie \) Insurance

<table>
<thead>
<tr>
<th>Patient</th>
<th>Insurance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>'Bob'</td>
<td>'Blue'</td>
</tr>
<tr>
<td>'Seattle'</td>
<td>123</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>'Ela'</td>
<td>'Prem'</td>
</tr>
<tr>
<td>'Everett'</td>
<td>432</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>'Jill'</td>
<td>'Prem'</td>
</tr>
<tr>
<td>'Kent'</td>
<td>343</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>'Joe'</td>
<td>'GrpH'</td>
</tr>
<tr>
<td>'Seattle'</td>
<td>554</td>
</tr>
</tbody>
</table>

Two tuples per page
Hash Join Example

Patient \Join Insurance

Disk

Patient	Insurance
1	2	2	4	6	6
3	4	4	3	1	3
9	6	2	8
8	5	8	9

Memory M = 21 pages

Some large-enough #

Showing pid only

This is one page with two tuples
Hash Join Example

Step 1: Scan Patient and build hash table in memory
Can be done in method open()

Disk

<table>
<thead>
<tr>
<th>Patient</th>
<th>Insurance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2</td>
<td>2 4 6 6</td>
</tr>
<tr>
<td>3 4</td>
<td>4 3 1 3</td>
</tr>
<tr>
<td>9 6</td>
<td>2 8</td>
</tr>
<tr>
<td>8 5</td>
<td>8 9</td>
</tr>
</tbody>
</table>

Memory M = 21 pages

Hash h: pid % 5

| 5 | 1 | 6 | 2 | 3 | 8 | 4 | 9 |

Input buffer
Hash Join Example

Step 2: Scan Insurance and probe into hash table
Done during calls to next()

Memory M = 21 pages
Hash h: pid % 5

Disk

Patient | Insurance
---|---
1 2 | 2 4 6 6
3 4 | 4 3 1 3
9 6 | 2 8
8 5 | 8 9

Write to disk or pass to next operator
Hash Join Example

Step 2: Scan Insurance and probe into hash table
Done during calls to next()

Memory M = 21 pages
Hash h: pid % 5

Disk

<table>
<thead>
<tr>
<th>Patient</th>
<th>Insurance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2</td>
<td>2 4</td>
</tr>
<tr>
<td>3 4</td>
<td>4 3</td>
</tr>
<tr>
<td>9 6</td>
<td>2 8</td>
</tr>
<tr>
<td>8 5</td>
<td>8 9</td>
</tr>
</tbody>
</table>

Input buffer

Output buffer

5 1 6 2 3 8 4 9

2 4

4 4
Hash Join Example

Step 2: Scan Insurance and probe into hash table
Done during calls to next()

Memory $M = 21$ pages
Hash $h$: pid $\% 5$

Input buffer
Output buffer
Keep going until read all of Insurance

Cost: $B(R) + B(S)$
Nested Loop Joins

- Tuple-based nested loop \( R \bowtie S \)
- \( R \) is the outer relation, \( S \) is the inner relation

\[
\text{for each tuple } t_1 \text{ in } R \text{ do } \\
\text{ for each tuple } t_2 \text{ in } S \text{ do } \\
\text{ if } t_1 \text{ and } t_2 \text{ join then output } (t_1,t_2)
\]

What is the Cost?
Nested Loop Joins

- Tuple-based nested loop \( R \bowtie S \)
- \( R \) is the outer relation, \( S \) is the inner relation

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

- Cost: \( B(R) + T(R)B(S) \)
- Multiple-pass since \( S \) is read many times

What is the Cost?
Page-at-a-time Refinement

for each page of tuples r in R do
  for each page of tuples s in S do
    for all pairs of tuples \( t_1 \) in r, \( t_2 \) in s
      if \( t_1 \) and \( t_2 \) join then output \((t_1, t_2)\)

- Cost: \( B(R) + B(R)B(S) \)

What is the Cost?
Page-at-a-time Refinement

Disk

Patient  Insurance

Patient  Insurance

Input buffer for Patient

Input buffer for Insurance

Output buffer

1 2

2 4

6 6

4 3

1 3

2 8

8 9

168
Page-at-a-time Refinement

Disk

Patient
1 2
3 4
9 6
8 5

Insurance
2 4
4 3
2 8
8 9

Input buffer for Patient
1 2
4 3

Input buffer for Insurance

Output buffer

169
Page-at-a-time Refinement

Cost: $B(R) + B(R)B(S)$
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
Sort-Merge Join Example

Step 1: Scan Patient and sort in memory

Memory M = 21 pages

Patient | Insurance
--------|--------
1 2     | 2 4    | 6 6
3 4     | 4 3    | 1 3
9 6     | 2 8    |
8 5     | 8 9    |
Sort-Merge Join Example

Step 2: Scan Insurance and sort in memory

Disk

Patient | Insurance
--- | ---
1 2 | 2 4 6 6
3 4 | 4 3 1 3
9 6 | 2 8
8 5 | 8 9

Memory M = 21 pages

1 2 3 4 5 6 8 9
1 2 2 3 3 4 4 6
6 8 8 9
Sort-Merge Join Example

Step 3: **Merge** Patient and Insurance

Disk

<table>
<thead>
<tr>
<th>Patient</th>
<th>Insurance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2</td>
<td>2 4</td>
</tr>
<tr>
<td>3 4</td>
<td>4 3</td>
</tr>
<tr>
<td>9 6</td>
<td>2 8</td>
</tr>
<tr>
<td>8 5</td>
<td>8 9</td>
</tr>
</tbody>
</table>

Memory M = 21 pages

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2</td>
<td>3 4</td>
<td>5 6</td>
<td>8 9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 2</td>
<td>2 3</td>
<td>3 4</td>
<td>4 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 8</td>
<td>8 9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Output buffer

1 1
Sort-Merge Join Example

Step 3: **Merge** Patient and Insurance

Memory \( M = 21 \) pages

Keep going until end of first relation
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:
    \[ B(R) + T(R) \times (T(S) \times \frac{1}{V(S,a)}) \]