ADMINISTRIVIA

- HW5 out tonight
- OQ5 out Wednesday
- Both due February 21 (11:30 & 11:00)
- Exam grades on canvas by Wednesday
- Handed back in section on Thursday
TODAY

• Back to RDBMS
  • "Query plans" and DBMS planning
  • Management between SQL and execution
  • Optimization techniques
  • Indexing and data arrangement
QUERY EVALUATION STEPS

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

Disk

SQL query

Query optimization

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
Relational algebra expression is also called the “logical query plan”

$$\text{SELECT sname}$$
$$\text{FROM Supplier x, Supply y}$$
$$\text{WHERE x.sid = y.sid}$$
$$\quad \text{and } y.pno = 2$$
$$\quad \text{and } x.scity = 'Seattle'$$
$$\quad \text{and } x.sstate = 'WA'$$
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'
Supervisor($sid$, $sname$, scity, sstate)
Supply($sid$, pno, quantity)

**PHYSICAL QUERY PLAN 2**

(On the fly) $\Pi_{sname}$

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

(Hash join) $\sigma_{\text{sid} = \text{sid}}$

Supplier (File scan)

Supply (File scan)

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

Same logical query plan
Different physical plan
Supplier (sid, sname, scity, sstate)
Supply (sid, pno, quantity)

**PHYSICAL QUERY PLAN 3**

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

(Sort-merge join) sid = sid (c)

(Scan & write to T1) 

(a) \( \sigma_{scity='Seattle'} \) and \( sstate='WA' \)

Supplier (File scan)

(b) \( \sigma_{pno=2} \) (Scan & write to T2)

Supply (File scan)

Different but equivalent logical query plan; different physical plan

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

Each of the logical operators may have one or more implementations = physical operators

Will discuss several basic physical operators, with a focus on join
Logical operator:

\[
\text{Supplier} \bowtie_{\text{sid} = \text{sid}} \text{Supply}
\]

Propose three physical operators for the join, assuming the tables are in main memory:

1.

2.

3.
Logical operator:

\[ \text{Supplier} \bowtie_{\text{sid} = \text{sid}} \text{Supply} \]

Propose three physical operators for the join, assuming the tables are in main memory:

1. Nested Loop Join \( O(??) \)
2. Merge join \( O(??) \)
3. Hash join \( O(??) \)
Logical operator:

\[
\text{Supplier} \bowtie_{\text{sid}=\text{sid}} \text{Supply}
\]

Propose three physical operators for the join, assuming the tables are in main memory:

1. Nested Loop Join \( O(n^2) \)
2. Merge join \( O(n \log n) \)
3. Hash join \( O(n) \ldots O(n^2) \)
BRIEF REVIEW OF HASH TABLES

A (naïve) hash function:

\[ h(x) = x \mod 10 \]

Operations:

- \( \text{find}(103) = ?? \)
- \( \text{insert}(488) = ?? \)
BRIEF REVIEW OF HASH TABLES

insert(k, v) = inserts a key k with value v

Many values for one key
• Hence, duplicate k’s are OK

find(k) = returns the list of all values v associated to the key k
ITERATOR INTERFACE

Each operator implements three methods:

open()

next()

close()
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 ();
}
```
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

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();
    }
}
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 () {

    }

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 ();
}
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(in);
        }
    }
}
```
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

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(in);
        }
        return r;
    }
}

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(in);
        }
        return r;
    }
    void close () {
        child.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 ();
}

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(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

**PIPELINING**

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

(On the fly)
(On the fly)
(Nested loop)

Discuss: open/next/close for nested loop join

Suppliers
(File scan)

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

**PIPETLINING**

(On the fly)

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

(On the fly)

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

(Nested loop)

sno = sno

Suppliers
(File scan)

Supplies
(File scan)

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

**PIPELINING**

(On the fly)

$\pi_{sname}$

(On the fly)

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

(Nested loop)

$sno = sno$

Suppliers
(File scan)

Supplies
(File scan)

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

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

Supplies

\[ \text{Supplies}(\text{sid}, \text{pno}, \text{quantity}) \]

**PIPPLINING**

(On the fly)

(On the fly)

(On the fly)

(Nested loop)

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

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

\[ \text{sno = sno} \]

Suppliers

(File scan)

Supplies

(File scan)

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

**PIPELINING**

- *(On the fly)*
  - \(\pi_{\text{sname}}\) open()
  - \(\sigma_{\text{scity} = 'Seattle' \text{ and } \text{sstate} = 'WA' \text{ and } \text{pno}=2}\) open()
  - open()

- *(Nested loop)*
  - \(\text{sno} = \text{sno}\)
  - Suppliers open()
    - *(File scan)*
  - Supplies open()
    - *(File scan)*

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

**PIPELINING**

(On the fly)

\[\prod_{\text{sname}}\]

(On the fly)

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

(Nested loop)

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

(On the fly)

(On the fly)

(On the fly)

File scan

File scan

File scan

Discuss: open/next/close for nested loop join


**PIPELINING**

\[
\begin{align*}
\text{Suppliers} & \text{(sid, sname, scity, sstate)} \\
\text{Supplies} & \text{(sid, pno, quantity)}
\end{align*}
\]

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

PIPELINING

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

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

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

Suppliers  
(File scan)

Supplies  
(File scan)

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

PIPELINING

(On the fly)

$\pi_{sname}$

(On the fly)

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

(Nested loop)

sno = sno

Suppliers
(File scan)

Supplies
(File scan)

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

**PIPELINING**

(On the fly)

\[ \pi_{sname} \]

(On the fly)

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

(Nested loop)

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

Suppliers

(File scan)

Supplies

(File scan)

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

**PIPELINING**

\(\pi_{\textit{sname}}\)

(On the fly)

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

(On the fly)

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

(Nested loop)

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

next()

Suppliers
(File scan)

Supplies
(File scan)

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

**PIPELINING**

(On the fly)

\[ \pi_{sname} \]

(On the fly)

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

(Nested loop)

\[ \text{sno = sno} \]

Discuss: open/next/close for nested loop join

Suppliers

(File scan)

Supplies

(File scan)
Supplier\((sid, sname, scity, sstate)\)
Supply\((sid, pno, quantity)\)

**PIPELINING**

(On the fly) \(\pi\_{sname}\)

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

(Hash Join) \(sno = sno\)

Suppliers
(File scan)

Supplies
(File scan)

Discuss hash-join in class
Suppliers
Supplies

PIPETLINING

(On the fly)  \[\pi_{\text{sname}}\]

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

(Hash Join)  \[\text{sno} = \text{sno}\]

Discuss hash-join in class

Tuples from here are pipelined

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

(File scan)  Suppliers
(File scan)  Supplies
Supplier($sid$, $sname$, $scity$, $sstate$)
Supply($sid$, $pno$, $quantity$)

**PIPELINING**

(On the fly)

$\pi_{sname}$

(On the fly)

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

(Hash Join)

$sno = sno$

Tuples from here are pipelined

Tuples from here are “blocked”

Discuss hash-join in class
Suppliers
d
Supply

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

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

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

Suppliers

(File scan)

Supplies

(File scan)

Discuss merge-join in class
**BLOCKED EXECUTION**

\[
\begin{align*}
\pi_{sname} &\left( \sigma_{\text{scity} = 'Seattle' \ \text{and} \ \text{sstate} = 'WA' \ \text{and} \ pno=2} \left( \text{Supplier}(\text{sid, sname, scity, sstate}) \right) \right) \\
\end{align*}
\]

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

- **Merge Join**
  - \( \pi_{sname} \)
  - \( \text{Supplier}(\text{sid, sname, scity, sstate}) \)
  - \( \text{Supply}(\text{sid, pno, quantity}) \)

**Blocked**
- Suppliers (File scan)
- Supplies (File scan)

Discuss merge-join in class.
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
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

<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 the example, we have 4 blocks with 2 tuples each
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>
DATA FILE TYPES

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.

<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>
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
KEYS IN INDEXING

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>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>240</td>
<td></td>
<td></td>
</tr>
<tr>
<td>420</td>
<td></td>
<td></td>
</tr>
<tr>
<td>800</td>
<td></td>
<td></td>
</tr>
<tr>
<td>950</td>
<td>...</td>
<td></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>
</tbody>
</table>

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

Index File (preferably in memory)

Data file (on disk)
B+ TREE INDEX BY EXAMPLE

d = 2

Find the key 40
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