ADMINISTRIVIA

• HW4 due Wednesday
THIS WEEK

• Back to RDBMS
  • indexing, optimization, and execution
  • last material on the midterm (next Friday)
  • fitting 4 lectures into 3
    • disk-based techniques becoming less relevant
    • distributed techniques (next week) becoming more relevant
TODAY

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

- Parse & Rewrite Query
- Select Logical Plan
- Select Physical Plan
- Query Execution
- Disk

SQL query

Logical plan (RA)

Physical plan

Query optimization
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'
```
\text{Supplier}(\text{sid}, \text{sname}, \text{scity}, \text{sstate}) \\
\text{Supply}(\text{sid}, \text{pno}, \text{quantity})

\text{PHYSICAL QUERY PLAN 2}

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

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

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

Supplier (File scan)

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

Supply (File scan)

Different but equivalent logical query plan; different physical plan

```
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 queries 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
Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

MAIN MEMORY ALGORITHMS

Logical operator:

\[
\text{Supplier} \Join_{\text{id}=\text{sid}} \text{Supply}
\]

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

1.

2.

3.
Supplier(sid, sname, scity, sstate)
Supply(sid, pno, quantity)

MAIN MEMORY ALGORITHMS

Logical operator:

Supplier \bowtie_{\text{id}=\text{id}} \ Supply

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

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

\[ \text{Supplier} \Join_{\text{id} = \text{id}} \text{Supply} \]

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 \( O(n) \), ... \( O(n^2) \)

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

Example “on the fly” selection operator

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 {...}
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

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 () {

    }
}
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(); }
}
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);
}
Suppliers
Supply

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

\text{Nested loop}

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

\text{PIPELINING}

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

\pi_{ \text{sname} } \left( \sigma_{ \text{scity}'Seattle' \text{ and sstate}'WA' \text{ and pno}=2 } \left( \text{snos} \right) \right)

\text{Discuss: open/next/close for nested loop join}
Suppliers: $	ext{Supplier}(\text{sid}, \text{sname}, \text{scity}, \text{sstate})$

Supplies: $	ext{Supply}(\text{sid}, \text{pno}, \text{quantity})$

**Pipelining**

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

**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)  open()
\[ \Pi_{\text{sname}} \]
(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)

Supplies (File scan)

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

**PIPELINING**

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

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

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

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

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

\(\text{sno = sno}\)

**Suppliers**

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

**Supplies**

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

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

PIPEDLINING

(On the fly)

π_{sname}

(On the fly)

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

(Nested loop)

sno = sno

open() open() open()

Suppliers
(File scan)

Supplies
(File scan)

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

**PIPPELINING**

(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) $\text{suppliers} = \text{suppliers}$

<table>
<thead>
<tr>
<th>Suppliers</th>
<th>Supplies</th>
</tr>
</thead>
<tbody>
<tr>
<td>(File scan)</td>
<td>(File scan)</td>
</tr>
</tbody>
</table>

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

**PIPEDLINING**

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

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

(Nested loop)  

**Suppliers**  
(File scan)

**Supplies**  
(File scan)

Discuss: open/next/close for nested loop join
\textbf{PIPELINING}

\begin{align*}
\text{Supplier}(\text{sid, sname, scity, sstate}) \\
\text{Supply}(\text{sid, pno, quantity})
\end{align*}

\begin{itemize}
\item \textbf{(On the fly)} \hspace{2cm} \pi_{\text{sname}}
\item \textbf{(On the fly)} \hspace{2cm} \sigma_{\text{scity}=\text{‘Seattle’} \text{ and } \text{sstate}=\text{‘WA’} \text{ and } \text{pno}=2}
\item \textbf{(Nested loop)} \hspace{2cm} \text{sno} = \text{sno}
\end{itemize}

\textbf{Suppliers} \hspace{1cm} \textbf{Supplies}

\textbf{(File scan)} \hspace{1cm} \textbf{(File scan)}

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

PIPEGELING

(On the fly)

(On the fly)

(Nested loop)

π_{sname}

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

sno = sno

Suppliers

Supplies

(File scan)

(File scan)

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

PIPPLELINING

(On the fly)

(On the fly)

(Nested loop)

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

Supply(sid, pno, quantity)

PIEPLAINING


Nested loop

(On the fly)

(On the fly)

Next()

π_{sname}

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

sno = sno

next()

next()

(On the fly)

Suppliers

(File scan)

Supplies

(File scan)

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

Supply(sid, pno, 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)

Supplies

(File scan)

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

Tuples from here are pipelined
Suppliers

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

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

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

Tuples from here are "blocked"

Tuples from here are pipelined

(On the fly)

(On the fly)

(Hash Join)

Suppliers

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

Supplies

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

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

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

Merge Join

Suppliers
(File scan)

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

**BLOCKED EXECUTION**

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

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

\[
\text{(On the fly)} \quad \text{(On the fly)} \quad \text{(Merge Join)}
\]
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
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

• **Our focus** is on data too large to fit in memory
  • disk-based DBMSs this week
  • distributed DBMSs next week
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>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>
</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>
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.
INDEX

An additional file, that allows fast access to records in the data file given a search key
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>...</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>

| 10 | Tom   | Hanks |
| 20 | Amy   | Hanks |
| 50 | ...   | ...   |
| 200| ...   |       |
| 220|       |       |
| 240|       |       |
| 420|       |       |
| 800|       |       |
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)

[Diagram showing the hash table example with index and data file, along with a table of student data.]
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**
SCANNING A DATA FILE

Hard 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): $$$ more expensive but increasingly common
SUMMARY SO FAR

Index = a file that enables direct access to records in another data file

- B+ tree / Hash table
- Clustered/unclustered

Data resides on (hard) disk

- Organized in blocks
- Sequential reads are efficient
- Random access less efficient
- Random read 1-2% of data worse than sequential