CSE 344

FEBRUARY 14TH – INDEXING
EXAM

• Grades posted to Canvas
• Exams handed back in section tomorrow
• Regrades: Friday office hours
EXAM

- Overall, you did well
  - Average: 79
  - Remember: lowest between midterm/final is only worth 25% of your grade
  - Still ~50% of points are still up for grabs
COURSE SCHEDULE

- Staggered assignments for next few weeks
  - HW6 on AWS, some setup time, easier assignment
  - HW7: Written Assignment – Feedback back before finals week
  - HW8: Java/JDBC assignment
# COURSE SCHEDULE

<table>
<thead>
<tr>
<th>Monday</th>
<th>Wednesday</th>
<th>Friday</th>
</tr>
</thead>
<tbody>
<tr>
<td>HW5 Out</td>
<td></td>
<td>HW6 Out</td>
</tr>
<tr>
<td></td>
<td>HW5 Due</td>
<td>HW7 Out</td>
</tr>
<tr>
<td>HW8 Out</td>
<td>HW6 Due</td>
<td>HW7 Due</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This week

Finals week
COURSE SCHEDULE

• Multiple assignments out at once
  • 2 additional late days – total of 5
  • Only 2 per assignment
  • HW8 – use only one tag
• HW3/4 Feedback by Wednesday (21st)
TODAY

• RDBMS
  • Physical plans
  • Pipelining
  • Indexing
QUERY EVALUATION STEPS

SQL query → Parse & Rewrite Query → Select Logical Plan → Select Physical Plan → Query Execution → Disk

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
 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 ();
}

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 ();
}
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();
  }
}

Example “on the fly” selection operator
Suppliers

Supplies

**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)
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)  

$sno = sno$

Tuples from here are pipelined

---

Suppliers  
(File scan)

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

**PIPELINING**

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

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

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

Tuples from here are pipelined

Tuples from here are “blocked”

Suppliers (File scan)

Supplies (File scan)
**BLOCKED EXECUTION**

**Suppliers**

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

**Supply**

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

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

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

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

- Suppliers (File scan)
- Supplies (File scan)

Discuss merge-join in class


**BLOCKED EXECUTION**

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

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

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

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

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

\( \text{Blocked} \)  
\( \text{Blocked} \)

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**
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

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>...</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**

<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>

**Data File Student**
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
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 \(\approx\) 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, still too expensive for everything
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 disk

- Organized in blocks
- Sequential reads are efficient
- Random access less efficient
- Random read 1-2% of data worse than sequential
SELECT *
FROM Student x, Takes y
WHERE x.ID=y.studentID AND y.courseID > 300

Assume the database has indexes on these attributes:

- **Takes_courseID** = index on Takes.courseID
- **Student_ID** = index on Student.ID
Student(ID, fname, lname)
Takes(studentID, courseID)

EXAMPLE

```
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:
- **Takes_courseID** = index on Takes.courseID
- **Student_ID** = index on Student.ID

```
for y' in Takes_courseID where y'.courseID > 300
    y = fetch the Takes record pointed to by y'
    for x' in Student_ID where x'.ID = y.studentID
        x = fetch the Student record pointed to by x'
        output *
```
SELECT *
FROM Student x, Takes y
WHERE x.ID = y.studentID AND y.courseID > 300

Assume the database has indexes on these attributes:
• Takes_courseID = index on Takes.courseID
• Student_ID = index on Student.ID

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

for y’ in Takes_courseID where y’.courseID > 300
  y = fetch the Takes record pointed to by y’
  for x’ in Student_ID where x’.ID = y.studentID
    x = fetch the Student record pointed to by x’
    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)
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)

What does this mean?
GETTING PRACTICAL: 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);

SELECT * FROM V WHERE P=55 AND M=77

SELECT * FROM V WHERE P=55

SELECT * FROM V WHERE M=77
GETTING PRACTICAL: 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);

select * from V where P=55
select * from V where M=77
select * from V where P=55 and M=77

yes
yes
no
GETTING PRACTICAL: 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)

select *
from V
where P=55 and M=77

select *
from V
where P=55

select *
from V
where M=77

Not supported in SQLite
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

<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 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:

SELECT *
FROM V
WHERE N=?

100 queries:

SELECT *
FROM V
WHERE P=?

What indexes?
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=?

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:

100 queries:

100000 queries:

SELECT * 
FROM V 
WHERE N>? and N<?

SELECT * 
FROM V 
WHERE P=?

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)
THE INDEX SELECTION PROBLEM 3

V(M, N, P);

Your workload is this

100000 queries:
SELECT * 
FROM V 
WHERE N=?

1000000 queries:
SELECT * 
FROM V 
WHERE N=? and P>?

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

What indexes?
THE INDEX SELECTION PROBLEM 3

V(M, N, P);

Your workload is this

100000 queries:
SELECT * FROM V WHERE N=?

1000000 queries:
SELECT * FROM V WHERE N=? and P>?

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

A: V(N, P)

How does this index differ from:
1. Two indexes V(N) and V(P)?
2. An index V(P, N)?
THE INDEX SELECTION PROBLEM 4

V(M, N, P);

Your workload is this

1000 queries:

SELECT *
FROM V
WHERE N>? and N<?

100000 queries:

SELECT *
FROM V
WHERE P>? and P<?

What indexes?
THE INDEX SELECTION
PROBLEM 4

V(M, N, P);

Your workload is this

1000 queries:

100000 queries:

SELECT *
FROM V
WHERE N>? and N<?

SELECT *
FROM V
WHERE P>? and P<?

A: V(N) secondary, V(P) primary index
TWO TYPICAL KINDS OF QUERIES

- **Point queries**
  
  ```sql
  SELECT *
  FROM Movie
  WHERE year = ?
  ```

- **Range queries**
  
  ```sql
  SELECT *
  FROM Movie
  WHERE year >= ? AND year <= ?
  ```

- 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 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<??