ADMINISTRIVIA

• HW4 due today

• HW5 out already
  • Asterix and SQL++

• Section tomorrow on Asterix + review
  • bring laptop
  • install Asterix (see HW5 instructions)
REVIEW (PT 1)

• Query evaluation
  • process
  • logical and physical query plans
  • pipelining (using iterator interface)
QUERY EVALUATION STEPS

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

SQL query

Query optimization

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
REVIEW (PT 2)

• Physical plan details
  • join algorithms
    • nested loop, hash join, sorted merge join
  • file organization
    • heap or sequential
  • indexes (today)

• Eventual Goal: estimate cost of a query plan
JOIN ALGORITHMS

Logical operator:  \( \text{Supplier} \Join_{\text{sid}=\text{sid}} \text{Supply} \)

Potential physical operators (more shortly...):

1. **Nested Loop Join**
   - two nested loops to iterate through both sets of tuples

2. **Sorted Merge join**
   - sort the tuples from each (on disk)
   - pass through both sorted lists in order to find matches

3. **Hash join**
   - put tuples from second into a hash table with key sid
   - for each tuple from first, lookup sid in hash to get matches
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
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>
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

Most common types:

- hash index
- B+ tree index
B+ TREE INDEX BY EXAMPLE

d = 2

Find the key 40
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 becoming less so
SUMMARY SO FAR

Index = 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
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
Example

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
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

\[
\text{for } y \text{ in Takes}  
\text{if courseID > 300 then}  
\text{for } x \text{ in Student}  
\text{if } x.ID = y.studentID  
\text{output } * 
\]

\[
\text{for } y' \text{ in Takes_courseID where} y'.courseID > 300  
y = \text{fetch the Takes record pointed to by } y'  
\text{for } x' \text{ in Student_ID where} x'.ID = y.studentID  
x = \text{fetch the Student record pointed to by } x'  
\text{output } * 
\]
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

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

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
```

```
select * from V where M=77
```

```
select * from V where P=55 and 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
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:

\[
\text{SELECT } * \\
\text{FROM } V \\
\text{WHERE } N=\? \\
\]

100 queries:

\[
\text{SELECT } * \\
\text{FROM } V \\
\text{WHERE } P=\? \\
\]

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)
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:

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

100000 queries:

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

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

- Point queries
  - SELECT *
    FROM Movie
    WHERE year = ?
  - What data structure should be used for index?

- Range queries
  - SELECT *
    FROM Movie
    WHERE year >= ? AND year <= ?
  - What data structure should be used for index?
TWO TYPICAL KINDS OF QUERIES

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

- What data structure should be used for index?
  
  Hash tables

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

- What data structure should be used for index?
  
  B+ Trees
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.
\begin{verbatim}
SELECT * 
FROM R 
WHERE R.K>? and R.K<?
\end{verbatim}
Percentage of tuples retrieved

Cost

Sequential scan

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<?