Introduction to Database Systems
CSE 414

Lecture 25:
Basics of Data Storage and Indexes
Announcements

• HW8 and WQ7
  – Due on 5/30

• OH changes
  – Alvin will be away next Wed
  – Jonathan will give next Wed’s lecture

• Final on Thurs 6/7
  – Final review on 6/3 afternoon
Recap: Transactions

• Protocols discussed:
  – Nothing
  – 2PL $\rightarrow$ unrecoverable schedules
  – Strict 2PL $\rightarrow$ phantom problem
  – Predicate locking $\rightarrow$ expensive!

• Recall our execution model!
Isolation Levels in SQL

1. “Dirty reads”
   SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

2. “Committed reads”
   SET TRANSACTION ISOLATION LEVEL READ COMMITTED

3. “Repeatable reads”
   SET TRANSACTION ISOLATION LEVEL REPEATABLE READ

4. Serializable transactions
   SET TRANSACTION ISOLATION LEVEL SERIALIZABLE

Try these in HW8!
Beware!

In commercial DBMSs:

- Default level is often NOT serializable
- Default level differs between DBMSs
- Some engines support subset of levels!
- Serializable may not be exactly ACID
  - Locking ensures isolation, not atomicity
- Also, some DBMSs do NOT use locking and different isolation levels can lead to different pbs

- Bottom line: RTFM for your DBMS!
Class Overview

- Unit 1: Intro
- Unit 2: Relational Data Models and Query Languages
- Unit 3: Non-relational data
- Unit 4: RDMBS internals and query optimization
- Unit 5: Parallel query processing
- Unit 6: DBMS usability, conceptual design
- Unit 7: Transactions
- Unit 8: Advanced topics: Query optimization
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>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Student 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>
<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>

CSE 414 - Spring 2018
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
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>
</tbody>
</table>

...
Example 2: Index on fName

Index \textit{Student\_fName} on \textit{Student.fName}

Data File \textit{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 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
Recall binary trees from CSE 143!

d = 2

Find the key 40

Index File
(preferably in memory)

Data file
(on disk)
Clustered vs Unclustered

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

• **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, still too expensive for everything
**Example**

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

**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 *
```

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

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

Student(ID, fname, lname)
Takes(studentID, courseID)
Example

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
Example

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

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

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

```python
for y' in Takes_courseID where y'.courseID > 300 
    y = fetch the Takes record pointed to by y'
```
Example

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’
```
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 *
```
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 *
```
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);
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)

What does this mean?
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?

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

yes
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)

What does this mean?

select *
from V
where P=55

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

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

What does this mean?

yes

yes
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);

What does this mean?

- 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);

What does this mean?

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

yes
no
yes
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)

What does this mean?

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