Announcements

• HW1 is due next Thursday
  – How is it going?

• Projects:
  – Proposals are due next Wednesday (not graded)
  – Submit on dropbox
Where We Are

• What we have already seen
  – Overview of the relational model
    • Motivation and where model came from
    • Physical and logical independence
  – How to design a database
    • From ER diagrams to conceptual design
    • Schema normalization
  – How different data models work

• Where we go from here
  – How can we efficiently implement this model?
  – How can we run RA plans efficiently?
References


• Chapters 8 through 11 (in the R&G book, third ed.)
  – Disk and files: Sections 9.3 through 9.7
  – Index structures: Section 8.3
  – Hash-based indexes: Section 8.3.1 and Chapter 11
  – B+ trees: Section 8.3.2 and Chapter 10
DBMS Architecture

Admission Control
Connection Mgr
Parser
Query Rewrite
Optimizer
Executor

Access Methods
Buffer Manager
Lock Manager
Log Manager

Memory Mgr
Disk Space Mgr
Replication Services
Admin Utilities

DBMS Architecture

Process Model

Why not simply queue all user requests? (and serve them one at the time)

Alternatives

1. **Process per connection**
2. **Server process** (thread per connection)
   - OS threads or DBMS threads
3. **Server process with I/O process**

Advantages and problems of each model?
Process Per Connection

• **Overview**
  – DB server forks one process for each client connection

• **Advantages**
  – Easy to implement (OS time-sharing, OS isolation, debuggers, etc.)
  – Provides more physical memory than a single process can use

• **Drawbacks**
  – Need OS support
    • Since all processes access the same data on disk, need concurrency control
  – Not scalable: memory overhead and expensive context switches
    • Goal is efficient support for high-concurrency transaction processing
Server Process

• **Overview**
  - DB assigns one thread per connection (from a thread pool)

• **Advantages**
  - Shared structures can simply reside on the heap
  - Threads are lighter weight than processes (memory, context switching)

• **Drawbacks**
  - Concurrent programming is hard to get right (race conditions, deadlocks)
  - Portability issues can arise when using OS threads
  - **Big problem**: entire process blocks on synchronous I/O calls
    • Solution 1: OS provides asynchronous I/O (true in modern OS)
    • Solution 2: Use separate process(es) for I/O tasks
DBMS Threads vs OS Threads

• Why do some DBMSs implement their own threads?
  – Legacy: originally, there were no OS threads
  – Portability: OS thread packages are not completely portable
  – Performance: fast task switching

• Drawbacks
  – Replicating a good deal of OS logic
  – Need to manage thread state, scheduling, and task switching

• How to map DBMS threads onto OS threads or processes?
  – Rule of thumb: one OS-provided dispatchable unit per physical device
  – See page 9 and 10 of Hellerstein and Stonebraker’s paper
Commercial Systems

- **Oracle**
  - Unix default: process-per-user mode
  - Unix: DBMS threads multiplexed across OS processes
  - Windows: DBMS threads multiplexed across OS threads

- **IBM DB2**
  - Unix: process-per-user mode
  - Windows: OS thread-per-user

- **SQL Server**
  - Windows default: OS thread-per-user
  - Windows: DBMS threads multiplexed across OS threads
<table>
<thead>
<tr>
<th>Process Manager</th>
<th>Query Processor</th>
<th>Shared Utilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Admission Control</td>
<td>Parser</td>
<td>Memory Mgr</td>
</tr>
<tr>
<td>Connection Mgr</td>
<td>Query Rewrite</td>
<td>Disk Space Mgr</td>
</tr>
<tr>
<td></td>
<td>Optimizer</td>
<td>Replication Services</td>
</tr>
<tr>
<td></td>
<td>Executor</td>
<td>Admin Utilities</td>
</tr>
<tr>
<td>Access Methods</td>
<td>Buffer Manager</td>
<td></td>
</tr>
<tr>
<td>Lock Manager</td>
<td>Log Manager</td>
<td></td>
</tr>
<tr>
<td>Storage Manager</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Admission Control

• Why does a DBMS need admission control?
  – To avoid thrashing and provide “graceful degradation” under load

• When does DBMS perform admission control?
  – In the dispatcher process: want to drop clients as early as possible to avoid wasting resources on incomplete requests
    • This type of admission control can also be implemented before the request reaches the DBMS (e.g., application server or web server)
  – Before query execution: delay queries to avoid thrashing
    • Can make decisions based on estimated resource needs for a query
DBMS Architecture

- Admission Control
- Connection Mgr
- Access Methods
- Lock Manager

- Parser
- Query Rewrite
- Buffer Manager
- Log Manager

- Optimizer
- Executor

- Memory Mgr
- Disk Space Mgr
- Replication Services

- Admin Utilities

Storage Model

• **Problem**: DBMS needs spatial and temporal control over storage
  – Spatial control for performance
  – Temporal control for correctness and performance

Alternatives

• **Use “raw” disk device interface directly**
  – Interact directly with device drivers for the disks

• **Use OS files**
Spatial Control
Using “Raw” Disk Device Interface

• **Overview**
  – DBMS issues low-level storage requests directly to disk device

• **Advantages**
  – DBMS can ensure that important queries access data sequentially
  – Can provide highest performance

• **Disadvantages**
  – Requires devoting entire disks to the DBMS
  – Reduces portability as low-level disk interfaces are OS specific
  – Many devices are in fact “virtual disk devices”
Spatial Control
Using OS Files

• **Overview**
  – DBMS creates one or more very large OS files

• **Advantages**
  – Allocating large file on empty disk can yield good physical locality

• **Disadvantages**
  – OS can limit file size to a single disk
  – OS can limit the number of open file descriptors
  – But these drawbacks have mostly been overcome by modern OSs
Commercial Systems

• Most commercial systems offer both alternatives
  – Raw device interface for peak performance
  – OS files more commonly used

• In both cases, we end-up with a DBMS file abstraction implemented on top of OS files or raw device interface
DBMS Architecture

Process Manager
- Admission Control
- Connection Mgr

Query Processor
- Parser
- Query Rewrite
- Optimizer
- Executor

Storage Manager
- Access Methods
- Lock Manager
- Buffer Manager
- Log Manager

Shared Utilities
- Memory Mgr
- Disk Space Mgr
- Replication Services
- Admin Utilities

Temporal Control
Buffer Manager

• Correctness problems
  – DBMS needs to control when data is written to disk in order to provide transactional semantics (we will study transactions later)
  – OS buffering can delay writes, causing problems when crashes occur

• Performance problems
  – OS optimizes buffer management for general workloads
  – DBMS understands its workload and can do better
  – Areas of possible optimizations
    • Page replacement policies
    • Read-ahead algorithms (physical vs logical)
    • Deciding when to flush tail of write-ahead log to disk
Buffer Manager

Page requests from higher-level code

Files and access methods
Buffer pool manager

Buffer pool

Disk page

Free frame

Main memory

Disk space manager

Disk is a collection of blocks

Disk

1 page corresponds to 1 disk block

CSE 544 - Fall 2015
Commercial Systems

- DBMSs implement their own buffer pool managers

- Modern filesystems provide good support for DBMSs
  - Using large files provides good spatial control
  - Using interfaces like the mmap suite
    - Provides good temporal control
    - Helps avoid double-buffering at DBMS and OS levels
DBMS Architecture

Process Manager
- Admission Control
- Connection Mgr

Query Processor
- Parser
- Query Rewrite
- Optimizer
- Executor

Storage Manager
- Files and Access Methods
- Lock Manager
- Buffer Manager
- Log Manager

Shared Utilities
- Memory Mgr
- Disk Space Mgr
- Replication Services
- Admin Utilities

Access Methods

- A DBMS stores data on disk by breaking it into pages
  - A page is the size of a disk block.
  - A page is the unit of disk IO
- Buffer manager caches these pages in memory
- Access methods do the following:
  - They organize pages into collections called DB files
  - They organize data inside pages
  - They provide an API for operators to access data in these files
Data Storage

• Basic abstraction
  – Collection of records or file
  – Typically, 1 relation = 1 database file
  – A file consists of one or more pages

• How to organize pages into files?
• How to organize records inside a file?

• Simplest approach: heap file (unordered)
Heap File Operations

- **Create** or **destroy** a file
- **Insert** a record
- **Delete** a record with a given rid (rid)
  - rid: unique tuple identifier
  - used to identify disk address of page containing record
- **Get** a record with a given rid
- **Scan** all records in the file
Heap File Implementation 1

Linked list of pages:

- Header page
  - Data page
  - Data page
    - Full pages
      - Data page
      - Data page
      - Data page
  - Pages with some free space
    - Data page
    - Data page
    - Data page
Heap File Implementation 2

Better: directory of pages

Header page

Directory

Data page

Data page

Data page

Directory contains **free-space count** for each page.

Faster inserts for variable-length records
Page Formats

Issues to consider

- 1 page = 1 disk block = fixed size (e.g. 8KB)
- Records:
  - Fixed length
  - Variable length
- Record id = RID
  - Typically RID = (PageID, SlotNumber)

Why do we need RID’s in a relational DBMS?
See discussion about indexes later in the lecture
Types of Files

• Heap file (what we discussed so far)
  – Unordered
• Sorted file (also called sequential file)
• Clustered file (aka indexed file)
File is **not sorted** on any attribute

**Student(sid: int, age: int, ...)**

<table>
<thead>
<tr>
<th>30</th>
<th>18 ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>70</td>
<td>21</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>20</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>19</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>80</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>18</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>10</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>22</td>
</tr>
</tbody>
</table>

1 record

1 page
Heap File Search Example

- 10,000 students
- 10 student records per page
- Total number of pages: 1,000 pages
- Find student whose sid is 80
  - Must read on average 500 pages
- Find all students older than 20
  - Must read all 1,000 pages
- Can we do better?
Sequential File

File **sorted on an attribute**, usually on primary key

Student \( (\text{sid}: \text{int}, \text{age}: \text{int}, ...) \)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>21</td>
<td>...</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>19</td>
<td></td>
</tr>
</tbody>
</table>
Sequential File Example

- Total number of pages: 1,000 pages
- Find student whose sid is 80
  - Could do binary search, read $\log_2(1,000) \approx 10$ pages
- Find all students older than 20
  - Must still read all 1,000 pages
- Can we do even better?
Index:

Index: data structure that organizes data records on disk to optimize selections on the search key fields for the index.

- An index contains a collection of data entries, and supports efficient retrieval of all data entries with a given search key value $k$. 
Indexes

• **Search key** = can be any set of fields
  – not the same as the primary key, nor a key

• **Index** = collection of data entries

• **Data entry** for key k can be:
  – The actual record with key k
    • In this case, the index is also a special file organization
    • Called: “indexed file organization”
  – (k, RID)
  – (k, list-of-RIDs)
Primary Index

- **Primary index**: determines location of indexed records
- **Dense index**: each record in data file is pointed to by a (key, rid) pairs in index

![Diagram showing Index File and Data File with key, rid pairs]
Primary Index with Duplicate Keys

- **Sparse** index: pointer to lowest search key on each page:

- Search for 20

  20 is here...

...but need to search here too

<table>
<thead>
<tr>
<th>10</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>40</td>
<td>40</td>
</tr>
</tbody>
</table>
Primary Index Example

• Let’s assume all pages of index fit in memory

• Find student whose sid is 80
  – Index (dense or sparse) points directly to the page
  – Only need to read 1 page from disk.

• Find all students older than 20
  – Must still read all 1,000 pages.

• How can we make both queries fast?
Secondary Indexes

- To index other attributes than primary key
- Always dense (why?)

```
<table>
<thead>
<tr>
<th>18</th>
<th>18</th>
<th>19</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>21</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>30</td>
<td>18</td>
<td>40</td>
<td>19</td>
</tr>
<tr>
<td>50</td>
<td>22</td>
<td>60</td>
<td>18</td>
</tr>
<tr>
<td>70</td>
<td>21</td>
<td>80</td>
<td>19</td>
</tr>
</tbody>
</table>
```
Clustered vs. Unclustered Index

CLUSTERED
Clustered = records close in index are close in data

UNCLUSTERED
Index Classification Summary

• **Primary/secondary**
  – Primary = determines the location of indexed records
  – Secondary = cannot reorder data, does not determine data location

• **Dense/sparse**
  – Dense = every key in the data appears in the index
  – Sparse = the index contains only some keys

• **Clustered/unclustered**
  – Clustered = records close in index are close in data
  – Unclustered = records close in index may be far in data

• **B+ tree / Hash table / …**
Large Indexes

• What if index does not fit in memory?

• Why not index the index itself?
  – Hash-based index
  – Tree-based index
Hash-Based Index

Good for point queries but not range queries

Another example of secondary index

Another example of primary index and indexed-file organization
Tree-Based Index

• How many index levels do we need?
• Can we create them automatically?
  – Yes!
• Can do something even more powerful!
B+ Trees

• Search trees

• Idea in B Trees
  – Make 1 node = 1 page (= 1 block)
  – Keep tree balanced in height

• Idea in B+ Trees
  – Make leaves into a linked list : facilitates range queries
B+ Trees

Note: can also store data records directly as data entries
B+ Trees Basics

- Parameter $d =$ the **degree**
- Each node has $d \leq m \leq 2d$ keys (except root)

```
30  120  240
```

Keys $k < 30$

```
40  50  60
```

Data records 40  50  60

Next leaf

Each node also has $m+1$ pointers

Keys $30 \leq k < 120$

Keys $120 \leq k < 240$

Keys $240 \leq k$
B+ Tree Example

Degree $d = 2$

Find the key 40
Searching a B+ Tree

• Exact key values:
  – Start at the root
  – Proceed down, to the leaf

• Range queries:
  – Find lowest bound as above
  – Then sequential traversal

Index on Student(age)

Select name
From Student
Where age = 25

Select name
From Student
Where 20 <= age
and age <= 30
B+ Tree Design

• How large should d be?
• Example:
  – Key size = 4 bytes
  – Pointer size = 8 bytes
  – Block size = 4096 bytes
• \[2d \times 4 + (2d+1) \times 8 \leq 4096\]
• \(d = 170\)
B+ Trees in Practice

• Typical order: 100. Typical fill-factor: 67%.
  – average fanout = 133

• Typical capacities
  – Height 4: $133^4 = 312,900,700$ records
  – Height 3: $133^3 = 2,352,637$ records

• Can often hold top levels in buffer pool
  – Level 1 = 1 page = 8 Kbytes
  – Level 2 = 133 pages = 1 Mbyte
  – Level 3 = 17,689 pages = 133 Mbytes
Insert (K, P)
- Find leaf where K belongs, insert
- If no overflow (2d keys or less), halt
- If overflow (2d+1 keys), split node, insert in parent:
  - If leaf, also keep K3 in right node
  - When root splits, new root has 1 key only
Insertion in a B+ Tree

Insert K=19
Insertion in a B+ Tree

After insertion
Insertion in a B+ Tree

Now insert 25
Insertion in a B+ Tree

After insertion
Insertion in a B+ Tree

But now have to split!

10 15 18 19
20 25 30 40 50
60 65 80 85 90
10 15 18 19 20 25 30 40 50 60 65 80 85 90
Insertion in a B+ Tree

After the split

...diagram...
Deletion from a B+ Tree

Delete 30
Deletion from a B+ Tree

After deleting 30

May change to 40, or not
Deletion from a B+ Tree

Now delete 25
Deletion from a B+ Tree

After deleting 25
Need to rebalance
Rotate
Deletion from a B+ Tree

Now delete 40
Deletion from a B+ Tree

After deleting 40
Rotation not possible
Need to *merge* nodes

![Diagram of B+ Tree deletion process](image-url)
Deletion from a B+ Tree

Final tree
Summary on B+ Trees

• Default index structure on most DBMSs

• Very effective at answering ‘point’ queries:
  productName = ‘gizmo’

• Effective for range queries:
  50 < price AND price < 100

• Less effective for multirange:
  50 < price < 100 AND 2 < quant < 20
Indexes in Postgres

CREATE TABLE V(M int, N varchar(20), P int);

CREATE INDEX V1_N ON V(N)

CREATE INDEX V2 ON V(P, M)

CREATE INDEX VVV ON V(M, N)

CLUSTER V USING V2

Makes V2 clustered
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=?

Which indexes should we create?
Index Selection Problem 1

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)
Index Selection Problem 2

V(M, N, P)

Your workload is this

100000 queries:
```sql
SELECT * FROM V WHERE N>? and N<?
```

100 queries:
```sql
SELECT * FROM V WHERE P=?
```

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

Which indexes should we create?
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)
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>?

10000000 queries: INSERT INTO V VALUES (?, ?, ?)

Which indexes should we create?
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) (must be B-tree)
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<?
```

Which indexes should we create?
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