CS 457: Database Management Systems

Lecture 10
Architecture and Storage
What we already know…

- **Database** = collection of related files
- **DBMS** = program that manages the database
What we already know…

Relational Query Language:

- **Set-at-a-time**: instead of tuple-at-a-time

- **Declarative**: user says what they want and not how to get it

- **Query optimizer**: from *what* to *how*
Benefits of relational model

• Physical data independence
  – Can change physical data organization on disk for performance \textit{without affecting applications}
  – Thanks to logical data model and set-at-a-time query language

• Logical data independence
  – Can change logical schema \textit{without affecting applications}
  – Thanks to views and query rewriting
How to Implement a Relational DBMS?

Key challenge: Achieve high performance on large databases!
Goal for Today

Overview of DBMS architecture

Overview of storage management
DBMS Architecture

Process Manager
- Admission Control
- Connection Mgr

Query Processor
- Parser
- Query Rewrite
- Optimizer
- Executor

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

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

Buffer Manager

Page requests from higher-level code

Access methods
Buffer pool manager

Buffer pool

Disk page
Free frame

Main memory

Disk is a collection of blocks

1 page corresponds to 1 disk block

Disk
Buffer Manager

• Brings pages in from memory and caches them
• Eviction policies
  – Random page
  – Least-recently used
• Keeps track of which **pages are dirty**
  – A dirty page has changes not reflected on disk
  – Implementation: Each page includes a dirty bit
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
- Discussion:
  - OS files vs. DBMS files
  - OS memory manager vs. DBMS buffer manager
Consider a relation storing tweets:

Tweets(tid, user, time, content)

How should we store it on disk?
Design Exercise

• Design choice: **One OS file for each relation**
  – This does not always have to be the case! (e.g., SQLite uses one file for whole database)
  – DBMSs can also use disk drives directly

• An OS file provides an API of the form
  – Seek to some position (or “skip” over B bytes)
  – Read/Write B bytes
First Principle: Work with Pages

- Reading/writing to/from disk
  - Seeking takes a long time!
  - Reading sequentially is fast

- To simplify buffer manager, want to cache a collection of same-sized objects

- Solution: Read/write **pages** of data
  - A page should correspond to a disk block
Continuing our Design

Key questions:
• How do we organize pages into a file?
• How do we organize data within a page?

First, how could we store some tuples on a page? Let’s first assume all tuples are of the same size

Tweets(tid int, user char(10),
    time int, content char(140))
Design Exercise

• Think how you would store tuples on a page
  – Fixed length tuples
  – Variable length tuples

• Will use “tuple” and “record” interchangeably in the following discussion
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 future discussion on indexes
Page Format Approach 1

Fixed-length records: packed representation
Divide page into slots. Each slot can hold one tuple
Record ID (RID) for each tuple is (PageID,SlotNb)

<table>
<thead>
<tr>
<th>Slot₁</th>
<th>Slot₂</th>
<th>Slotₙ</th>
<th>Free space</th>
<th>Number of records</th>
</tr>
</thead>
</table>

How do we insert a new record?
Page Format Approach 1

Fixed-length records: packed representation

Divide page into slots. Each slot can hold one tuple

Record ID (RID) for each tuple is (PageID,SlotNb)

<table>
<thead>
<tr>
<th>Slot₁</th>
<th>Slot₂</th>
<th>Slotₙ</th>
<th>Slotₙ₊₁</th>
<th>Free Sp.</th>
<th>N</th>
</tr>
</thead>
</table>

How do we insert a new record?

Number of records
Page Format Approach 1

Fixed-length records: packed representation

Divide page into slots. Each slot can hold one tuple

Record ID (RID) for each tuple is (PageID, SlotNb)

How do we insert a new record?

How do we delete a record?
Page Format Approach 1

Fixed-length records: packed representation
Divide page into slots. Each slot can hold one tuple
Record ID (RID) for each tuple is (PageID, SlotNb)

How do we insert a new record?

How do we delete a record? What is the problem?
Page Format Approach 1

Fixed-length records: packed representation
Divide page into slots. Each slot can hold one tuple
Record ID (RID) for each tuple is (PageID, SlotNb)

- How do we insert a new record?
- How do we delete a record?
- How do we handle variable-length records?
Page Format Approach 2

Can handle variable-length records
Can move tuples inside a page without changing RIDs
RID is (PageID, SlotID) combination

Header contains slot directory
+ Need to keep track of number of slots
+ Also need to keep track of free space (F)

Slot directory
Each slot contains <record offset, record length>

Free space

4 F
Next: Record Formats
Record Formats

Fixed-length records => Each field has a fixed length (i.e., it has the same length in all the records)

| Field 1 | Field 2 | ... | ... | Field K |

Information about field lengths and types is in the catalog (think of the table metadata in your first project)
Record Formats

Variable length records

Record header

Remark: NULLS require no space (why ?)
Long Records Across Pages

- When records are very large
- Or even medium size: saves space in blocks
- Commercial RDBMSs avoid this
  - Hard to maintain
  - Space saving is not significant
LOB

• Large objects
  – Binary large object: BLOB
  – Character large object: CLOB

• Supported by modern database systems
• E.g. images, sounds, texts, etc.

• Storage: attempt to cluster blocks together
Continuing our Design

Our key questions:

• How do we organize pages into a file?
• How do we organize data within a page?

Now, how should we group pages into files?
Heap File Implementation 1

A sequence of pages

Data page | Data page | Data page | Data page | Data page | Data page | Data page

Some pages have space and other pages are full
Add pages at the end when need more space

Works well for small files
But finding free space requires scanning the file
Heap File Implementation 2

Linked list of pages:

- Header page

Full pages:

- Data page
- Data page
- Data page

Pages with some free space:

- Data page
- Data page
- Data page
Heap File Implementation 3

Better: directory of pages

Directory contains **free-space count** for each page.
Faster inserts for variable-length records
Modifications: Insertion

- File is unsorted (\textit{= heap file})
  - add it wherever there is space (easy \text{🙂})
  - add more pages if out of space

- File is sorted
  - Is there space on the right page?  
    - Yes: we are lucky, store it there
  - Is there space in a neighboring page?  
    - Look 1-2 pages to the left/right, shift records
  - If anything else fails, create \textit{overflow page}
Overflow Pages

- After a while the file starts being dominated by overflow pages: time to reorganize.
Modifications: Deletions

• Free space in page, shift records
  – Be careful with slots
  – RIDs for remaining tuples must NOT change

• May be able to eliminate an overflow page
Modifications: Updates

• If new record is shorter than previous, easy 😊
• If it is longer, need to shift records
  – May have to create overflow pages
Alternate Storage Manager Design: Column Store

Rows stored contiguously on disk

Columns stored contiguously on disk
More Detailed Example

Row-based
(4 pages)

Page

| A | 1 |
| A | 2 |
| A | 2 |
| B | 2 |
| B | 4 |
| C | 4 |
| C | 4 |

Column-based
(4 pages)

Page

| A | 1 |
| A | 2 |
| A | 2 |
| B | 2 |
| B | 4 |
| C | 4 |
| C | 4 |

C-Store potentially avoids large tuple headers (why?)
Continuing our Design

We know how to store tuples on disk in a heap file

How do these files interact with rest of engine?
How Components Fit Together

- **Query Processor**: Process data
- **Access methods**: Organize data to support fast access to desired subsets of records
- **Buffer manager**: Caches data in memory. Reads/writes data to/from disk as needed
- **Disk-space manager**: Allocates space on disk for files/access methods

**Operators**: Sequential Scan, etc.

**Access Methods**: HeapFile, etc.

**Buffer Manager**

**Disk Space Mgr**

Data on disk
Access Methods

• Operators view relations as collections of records

• The access methods worry about how to organize these collections
Heap File Access Method API

• **Create** or **destroy** a file
• **Insert** a record
• **Delete** a record with a given rid (rid)
  – rid: unique tuple identifier (more later)
• **Get** a record with a given rid
  – Not necessary for sequential scan operator
  – But used with indexes (more next lecture)
• **Scan** all records in the file
Pushing Updates to Disk

- When inserting a tuple, HeapFile inserts it on a page but does not write the page to disk.
- When deleting a tuple, HeapFile deletes tuple from a page but does not write the page to disk.
- The buffer manager worries when to write pages to disk (and when to read them from disk).
- When need to add a new page to the file, HeapFile adds page to the file on disk and then gets it again through the buffer manager.
Conclusion

- Row-store storage managers are most commonly used today
- They offer high-performance for transactions
- But column-stores win for analytical workloads
- They are gaining traction in that area