CS186 Discussion Section Week 2

File Organization and Indexing

Today

- Terminology review
- Fixed and variable length records
- Slotted pages
- Index Introduction
- I/O examples
- Worksheet

Terminology

- File
  - have pages
- Pages
  - have records
- Record
  - stores a tuple in your table (e.g. a single Student)
- Buffer Pool holds pages in memory
- Fixed length record
  - A record stored with fixed length. All tuples in a table would be the same # bytes on disk.
- Variable length record
  - A record stored with variable length
  - Different tuples in the table will be stored with

Terminology

- Slotted pages
  - Manage variable length records in a page
  - Fit records into slots in the page
- Side note: Managing fixed length records is simple.
  - Let's say your page is 10 KB. Your fixed length record is 50 bytes. Your first record is at byte 0. Second record is at byte 50. Third record is at byte 100. How many records per page?
- Slotted directory
  - Manages the slots in a slotted page.
  - Each slot contains a record and the slotted directory is essentially a dictionary with key: slot # and value:

Fixed Length Records

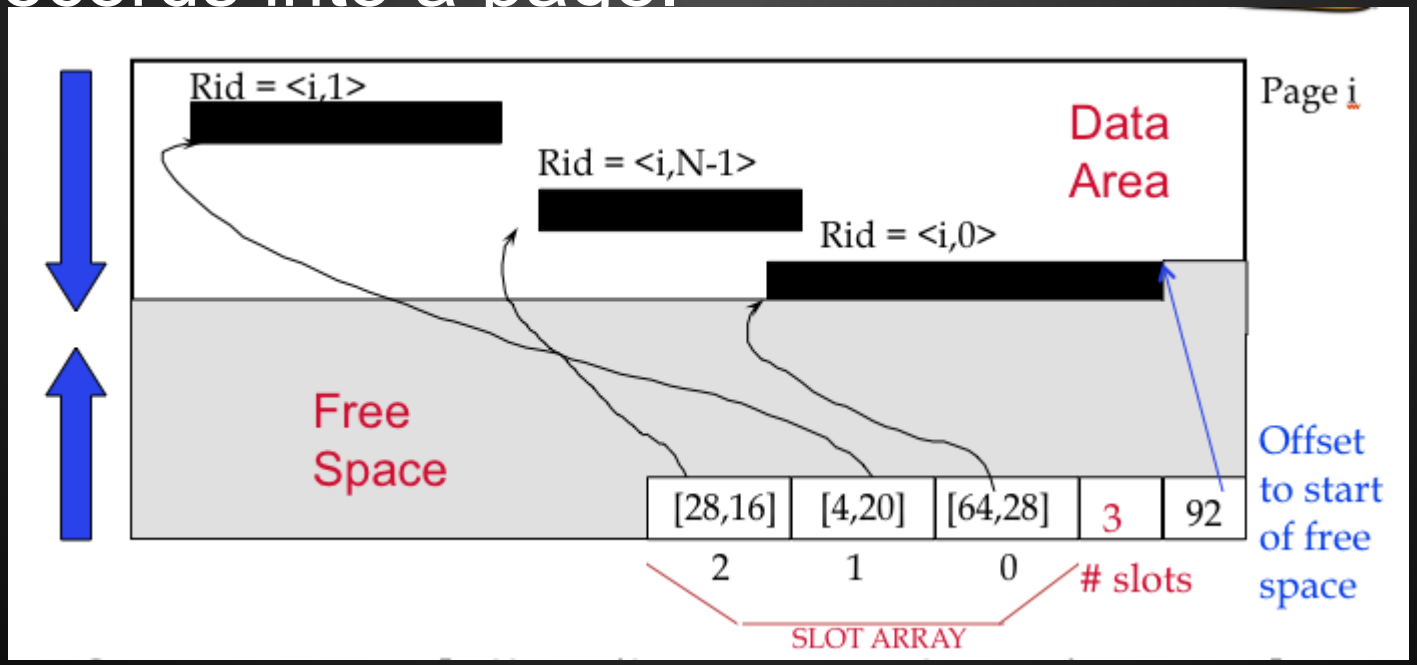
- When do we use fixed length records?
- What are some things we would store in our database as fixed length?
- Fixed length is easy. Easy to manage.
- Fixed length may take up more space.
- So we'll need to consider something else...

Variable Length Records

- When do we use variable length records?
- What are some examples of things we would store as variable length in our database?
- Variable length records a little bit harder to manage.
- How do we store variable length records?
  - Can store variable length record by using delimiter. For example Hive uses ‘\001’ to separate each field.
  - e.g. Joe’\001’Schmoe’\001’EECS stores the tuple (Joe, Schmoe,EECS)
  - Can use array of field offsets. [3,9]JoeSchmoeEECS stores the tuple (Joe, Schmoe, EECS)

Slotted Pages

- To use variable length records, we would need to somehow fit that into a page.
- Isn't quite as easy as fitting fixed length records into a page.



Slot Directory

- Slot directory is stored at the end of the page and as more and more slots are added, the slot directory grows upwards. (Kind of like the heap grows upwards)
- First 4 bytes holds offset to start of free space.
  - Why is this important?
  - If we want to insert another record, tells us where to fit our record into the next slot.
- Next 4 bytes is the number of slots.
  - Why is this important?
  - Tells us where our slot array ends.
- Slot array is essentially an array of [offset from start of page, length]
  - e.g. slot_array[0] contains information on how to

Slotted Page Example

- Query: SELECT * FROM Students WHERE sid=5135;
- Conveniently, the database has an index on sid, and tells you that you can find the result of the query at record id <page=10, slot#=3>
- Now, using the record id, you can look up the result of your query by loading page 10 into the buffer pool.
- The slot array for the slotted page looks like this: [<offset=10, length=10>, <offset=20, length=25>, <offset=0, length=10>, <offset=45, length=20>...]

  Now what?
- You know that slot # is 3. You look up the slot # 3 and see that the record you are looking for starts at offset=45 and has length 20 and now we can access it on the page!

Indexes

- From Wikipedia: “A database index is a data structure that improves the speed of data retrieval operations on a database table at the cost of slower writes and increased storage
  - Read speed goes up
  - Write speed goes down
  - Increased storage space

Indexes

Example: Wikipedia pages

• Each page has a lot of data, like name, page, URL, and other meta data.

• But we often just want to look up by URL.

• In order to avoid reading the entire database for looking up by URL, what can we do?

– Sort by URL

– Create an index!

Basic Hash Index vs Tree Based Index

- Hash Index is only good for equality and not for range queries.
  - For example, if you had a HashMap<Integer, String> in Java, there is no query HashMap.getGreaterThan(10) to get the values of for the keys of all Integers greater than 10.
  - You would need to do HashMap.get(11), HashMap.get(12)... HashMap.get(2147483648)
  - Doesn't make much sense does it?
- Tree Based Index can be used for both range queries and equality.
- A little bit more overhead with tree based index, so deciding which kind of index to use is based on what types of queries you are going to run on the table

Why don’t we index all of the fields?

Reasoning through Average Case I/O Counts for Operations (B = # disk blocks in file)

	Heap File	Sorted File
Scan All Records		
Equality Search (1 match)		
Range Search		
Insert		
Delete		

Reasoning through Average Case I/O Counts for Operations (B = # disk blocks in file)

	Heap File	Sorted File
Scan All Records	B	
Equality Search (1 match)		
Range Search		
Insert		
Delete		

Why?

Scanning through all records requires loading every single page into memory.

Reasoning through Average Case I/O Counts for Operations (B = # disk blocks in file)

	Heap File	Sorted File
Scan All Records	B	
Equality Search (1 match)	.5B	
Range Search		
Insert		
Delete		

Why?

Database needs to scan, on average, half of the pages to find what you're looking for.

What about 0 matches?

Reasoning through Average Case I/O Counts for Operations (B = # disk blocks in file)

	Heap File	Sorted File
Scan All Records	B	
Equality Search (1 match)	.5B	
Range Search	B	
Insert		
Delete		

Why?

Any page can contain a record within the range, so you'll have to look at all the pages.

Reasoning through Average Case I/O Counts for Operations (B = # disk blocks in file)

	Heap File	Sorted File
Scan All Records	B	
Equality Search (1 match)	.5B	
Range Search	B	
Insert	2	
Delete		

Why?

Database needs to load a page into memory (1 I/O). Write a record into the page. Now this page is dirty and we write the page back onto disk (1 I/O)

Reasoning through Average Case I/O Counts for Operations (B = # disk blocks in file)

	Heap File	Sorted File
Scan All Records	B	
Equality Search (1 match)	.5B	
Range Search	B	
Insert	2	
Delete	.5B + 1	

Why?

Database needs to find the record you want to delete first. Like equality search, loads on average half the tables pages before finding a match. (.5B I/O). Last I/O comes from writing the dirty page back to disk.

Reasoning through Average Case I/O Counts for Operations (B = # disk blocks in file)

	Heap File	Sorted File
Scan All Records	B	B
Equality Search (1 match)	.5B	
Range Search	B	
Insert	2	
Delete	.5B + 1	

Why?

Same reasoning as heap file

Reasoning through Average Case I/O Counts for Operations (B = # disk blocks in file)

	Heap File	Sorted File
Scan All Records	B	B
Equality Search (1 match)	.5B	log_2(B) (if on sort key) .5B (otherwise)
Range Search	B	
Insert	2	
Delete	.5B + 1	

Why?

If on sort key, we just do a binary search on the pages to find the right page. (log_2(B) pages loaded for binary search)

If not on sort key :((Same thing as heap file)

Reasoning through Average Case I/O Counts for Operations (B = # disk blocks in file)

	Heap File	Sorted File
Scan All Records	B	B
Equality Search (1 match)	.5B	log_2(B) (if on sort key) .5B (otherwise)
Range Search	B	log_2(B) + selectivity * B
Insert	2	
Delete	.5B + 1	

Why?

log_2(B) to find the first page in beginning of range.

selectivity * B to read consecutive pages until end of range.

e.g. SELECT * FROM Students where age > 20 and age < 30;

Reasoning through Average Case I/O Counts for Operations (B = # disk blocks in file)

	Heap File	Sorted File
Scan All Records	B	B
Equality Search (1 match)	.5B	log_2(B) (if on sort key) .5B (otherwise)
Range Search	B	log_2(B) + selectivity * B
Insert	2	log_2(B) + B
Delete	.5B + 1	

Why?

Finding the page to insert to takes log_2(B).

What about the other B?

Remember the assumption that pages are fully packed. Inserting a single record, forces every page following the page the record was written to to be rewritten.

On average .5B pages after insertion. 0.5B I/O reads. 0.5B I/O writes

Reasoning through Average Case I/O Counts for Operations (B = # disk blocks in file)

	Heap File	Sorted File
Scan All Records	B	B
Equality Search (1 match)	.5B	log_2(B) (if on sort key) .5B (otherwise)
Range Search	B	log_2(B) + selectivity * B
Insert	2	log_2(B) + B
Delete	.5B + 1	log_2(B) + B

Why?

Same reasoning as insertion, except use assumption that files are compacted after deletion.