CompSci 516

Database Systems

Lecture 8

Normalization

Storage

Index

Instructor: Sudeepa Roy

Announcements

- HW1 Deadlines!
  - Today: parser and Q1-Q3 (last late day!)
  - Q4: next Tuesday 09/24
  - Q5 (RA questions posted on Sakai): next to next Tuesday 10/01
    - Check Piazza for submission instructions
- 2 late days with penalty apply for individual deadlines
  - It is important to start HWs from day-1!

Today's topic

- Finish Normalization
- New topic: Database Internals

Acknowledgement:
The following slides have been created adapting the instructor material of the [RG] book provided by the authors Dr. Ramakrishnan and Dr. Gehrke, and with the help of slides by Dr. Magda Balazinska and Dr. Dan Suciu

Recap: Functional Dependencies (FDs)

A	B	C	D
a1	b1	c1	d1
a1	b1	c1	d2
a1	b2	c2	d1
a2	b1	c3	d1

AB → C

ABD → C

AB → A (trivial)

But not

AB → D

A → D

A → C

Normal Forms

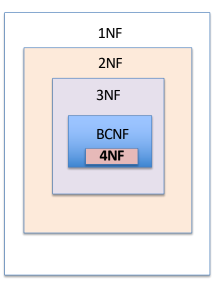
R is in 4NF

⇒ R is in BCNF

⇒ R is in 3NF

⇒ R is in 2NF (a historical one)

⇒ R is in 1NF (every field has atomic values)



Only BCNF and 4NF are covered in the class

Boyce-Codd Normal Form (BCNF)

- Relation R with FDs F is in BCNF
- if, for all X → A in F
  - A ϵ X (called a trivial FD), or
  - X contains a key for R
    - i.e. X is a superkey

No dependencies other than from superkeys can exist!

BCNF decomposition algorithm

1. Find a BCNF violation
   – That is, a non-trivial FD $X \rightarrow Y$ in R where X is not a super key of R
2. Decompose R into R_1 and R_2, where
   – R_1 has attributes $X \cup Y$
   – R_2 has attributes $X \cup Z$, where Z contains all attributes of R that are in neither X nor Y
3. Repeat until all relations are in BCNF

- Also gives a lossless decomposition!
  – Check yourself

BCNF decomposition example - 1

On blackboard

- CSJDPQV, key C, F = {JP → C, SD → P, J → S}
  – To deal with SD → P, decompose into SDP, CSJDQV.
  – To deal with J → S, decompose CSJDQV into JS and CJDQV
- Is JP → C a violation of BCNF?
- Note:
  – several dependencies may cause violation of BCNF
  – The order in which we pick them may lead to very different sets of relations
  – there may be multiple correct decompositions (can pick J → S first)

BCNF decomposition example - 2

uid → uname, twitterid
twitterid → uid
uid, gid → fromDate

UserJoinsGroup (*uid, uname, twitterid, gid, fromDate*)
BCNF violation: *uid → uname, twitterid*

User (*uid, uname, twitterid*)
uid → uname, twitterid
twitterid → uid
BCNF

Member (*uid, gid, fromDate*)
uid, gid → fromDate
BCNF

BCNF decomposition example - 3

It is not enough to only look at given FDs! You need to Consider the closure!

uid → uname, twitterid
twitterid → uid
uid, gid → fromDate

UserJoinsGroup (*uid, uname, twitterid, gid, fromDate*)
BCNF violation: *twitterid → uid*

apply Armstrong's axioms and rules!

UserId (*twitterid, uid*)
BCNF

UserJoinsGroup' (*twitterid, uname, gid, fromDate*)
twitterid → uname
twitterid, gid → fromDate
BCNF violation: *twitterid → uname*

UserName (*twitterid, uname*)
BCNF

Member (*twitterid, gid, fromDate*)
BCNF

Recap

- Functional dependencies: a generalization of the key concept
- Non-key functional dependencies: a source of redundancy
- BCNF decomposition: a method for removing redundancies
  – And gives lossless join decomposition
- BCNF = no redundancy due to FDs

But - the relation may still have redundancies! 4-NF (later)

Where are we now?

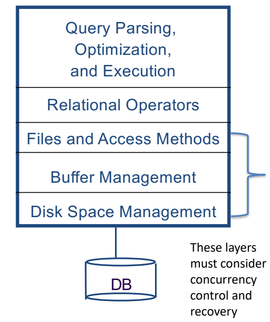
We learnt How to write queries and how to design a
- ✓ Relational Model and Query Languages
  - ✓ SQL, RA, RC
  - ✓ Postgres (DBMS)
  - ✓ XML (overview)
  - ▪ HW1
- ✓ Database Normalization

Next
- DBMS Internals
  – Storage
  – Indexing
  – Query Evaluation
  – Operator Algorithms
  – External sort
  – Query Optimization

Storage

DBMS Architecture

- A typical DBMS has a layered architecture
- The figure does not show the concurrency control and recovery components
  - to be done in "transactions"
- This is one of several possible architectures
  - each system has its own variations



Data on External Storage

- Data must persist on disk across program executions in a DBMS
  - Data is huge
  - Must persist across executions
  - But has to be fetched into main memory when DBMS processes the data
- The unit of information for reading data from disk, or writing data to disk, is a page
- Disks: Can retrieve random page at fixed cost
  - But reading several consecutive pages is much cheaper than reading them in random order

Disk Space Management

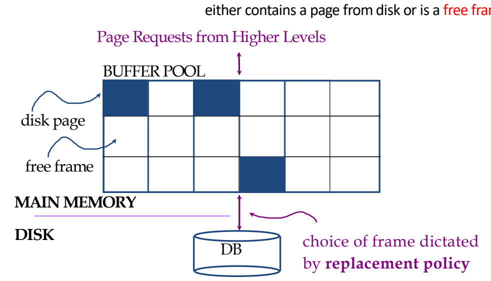
- Lowest layer of DBMS software manages space on disk
- Higher levels call upon this layer to:
  - allocate/de-allocate a page
  - read/write a page
- Size of a page = size of a disk block
  = data unit
- Request for a sequence of pages often satisfied by allocating contiguous blocks on disk
- Space on disk managed by Disk-space Manager
  - Higher levels don't need to know how this is done, or how free space is managed

Buffer Management

Suppose
- 1 million pages in db, but only space for 1000 in memory
- A query needs to scan the entire file
- DBMS has to
  - bring pages into main memory
  - decide which existing pages to replace to make room for a new page
  - called Replacement Policy
- Managed by the Buffer manager
  - Files and access methods ask the buffer manager to access a page mentioning the "record id" (soon)
  - Buffer manager loads the page if not already there

Buffer Management

Buffer pool = main memory is partitioned into frames either contains a page from disk or is a free frame



- Data must be in RAM for DBMS to operate on it
- Table of <frame#, pageid> pairs is maintained

When a Page is Requested ...

For every frame, store
- a dirty bit:
  - whether the page in the frame has been modified since it has been brought to memory
  - initially 0 or off
- a pin-count:
  - the number of times the page in the frame has been requested but not released (and no. of current users)
  - initially 0
  - when a page is requested, the count in incremented
  - when the requestor releases the page, count is decremented
  - buffer manager only reads a page into a frame when its pin-count is 0
  - if no frame with pin-count 0, buffer manager has to wait (or a transaction is aborted -- later)

When a Page is Requested ...

- Check if the page is already in the buffer pool
- if yes, increment the pin-count of that frame
- If no,
  - Choose a frame for replacement using the replacement policy
  - If the chosen frame is dirty (has been modified), write it to disk
  - Read requested page into chosen frame
- Pin (increase pin-count of) the page and return its address to the requestor

- If requests can be predicted (e.g., sequential scans), pages can be pre-fetched several pages at a time
- Concurrency Control & recovery may entail additional I/O when a frame is chosen for replacement
  - e.g. Write-Ahead Log protocol : when we do Transactions

Buffer Replacement Policy

- Frame is chosen for replacement by a replacement policy
- Least-recently-used (LRU)
  - add frames with pin-count 0 to the end of a queue
  - choose from head
- Clock (an efficient implementation of LRU)
- First In First Out (FIFO)
- Most-Recently-Used (MRU) etc.

Buffer Replacement Policy

- Policy can have big impact on # of I/O's
- Depends on the access pattern
- Sequential flooding: Nasty situation caused by LRU + repeated sequential scans
  - What happens with 10 frames and 9 pages?
  - What happens with 10 frames and 11 pages?
  - # buffer frames < # pages in file means each page request in each scan causes an I/O
  - MRU much better in this situation (but not in all situations, of course)

DBMS vs. OS File System

- Operating Systems do disk space and buffer management too:
- Why not let OS manage these tasks?
- DBMS can predict the page reference patterns much more accurately
  - can optimize
  - adjust replacement policy
  - pre-fetch pages – already in buffer + contiguous allocation
  - pin a page in buffer pool, force a page to disk (important for implementing Transactions concurrency control & recovery)
- Differences in OS support: portability issues
- Some limitations, e.g., files can't span disks

Next..

- How are pages stored in a file?
- How are records stored in a page?
  - Fixed length records
  - Variable length records
- How are fields stored in a record?
  - Fixed length fields/records
  - Variable length fields/records

Files of Records

- Page or block is OK when doing I/O, but higher levels of DBMS operate on records, and files of records
- FILE: A collection of pages, each containing a collection of records
- Must support:
  - insert/delete/modify record
  - read a particular record (specified using record id)
  - scan all records (possibly with some conditions on the records to be retrieved)

File Organization

- File organization: Method of arranging a file of records on external storage
  - One file can have multiple pages
  - Record id (rid) is sufficient to physically locate the page containing the record on disk
  - Indexes are data structures that allow us to find the record ids of records with given values in index search key fields
- NOTE: Several uses of "keys" in a database
  - Primary/foreign/candidate/super keys
  - Index search keys

Alternative File Organizations

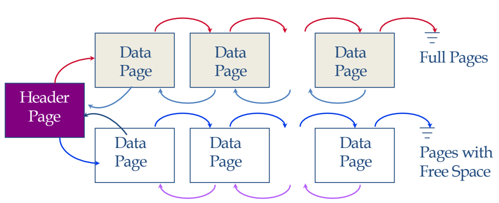
Many alternatives exist, each ideal for some situations, and not so good in others:

- Heap (random order) files: Suitable when typical access is a file scan retrieving all records
- Sorted Files: Best if records must be retrieved in some order, or only a "range" of records is needed.
- Indexes: Data structures to organize records via trees or hashing
  - Like sorted files, they speed up searches for a subset of records, based on values in certain ("search key") fields
  - Updates are much faster than in sorted files

Unordered (Heap) Files

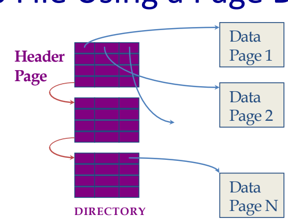
- Simplest file structure contains records in no particular order
- As file grows and shrinks, disk pages are allocated and de-allocated
- To support record level operations, we must:
  - keep track of the pages in a file
  - keep track of free space on pages
  - keep track of the *records* on a page
- There are many alternatives for keeping track of this

Heap File Implemented as a List



- The header page id and Heap file name must be stored someplace
- Each page contains 2 `pointers' plus data
- Problem?
  - to insert a new record, we may need to scan several pages on the free list to find one with sufficient space

Heap File Using a Page Directory

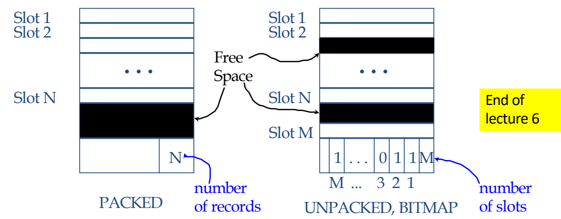


- The entry for a page can include the number of free bytes on the page.
- The directory is a collection of pages
  - linked list implementation of directory is just one alternative
  - Much smaller than linked list of all heap file pages!

How do we arrange a collection of records on a page?

- Each page contains several slots
  - one for each record
- Record is identified by <page-id, slot-number>
- Fixed-Length Records
- Variable-Length Records
- For both, there are options for
  - Record formats (how to organize the fields within a record)
  - Page formats (how to organize the records within a page)

Page Formats: Fixed Length Records

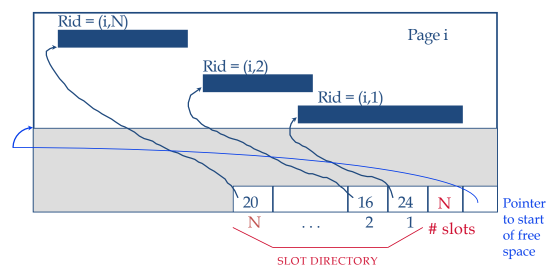


- Record id = <page id, slot #>
- Packed: moving records for free space management changes rid; may not be acceptable
- Unpacked: use a bitmap – scan the bit array to find an empty slot
- Each page also may contain additional info like the id of the next page (not shown)

Page Formats: Variable Length Records

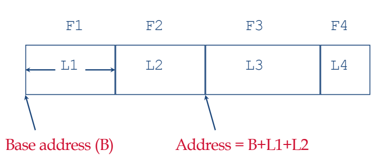
- Need to find a page with the right amount of space
  - Too small – cannot insert
  - Too large – waste of space
- if a record is deleted, need to move the records so that all free space is contiguous
  - need ability to move records within a page
- Can maintain a directory of slots (next slide)
  - Slot contains <record-offset, record-length>
  - deletion = set record-offset to -1
- Record-id rid = <page, slot-in-directory> remains unchanged

Page Formats: Variable Length Records



- Can move records on page without changing rid
  - so, attractive for fixed-length records too
- Store (record-offset, record-length) in each slot
- rid-s unaffected by rearranging records in a page

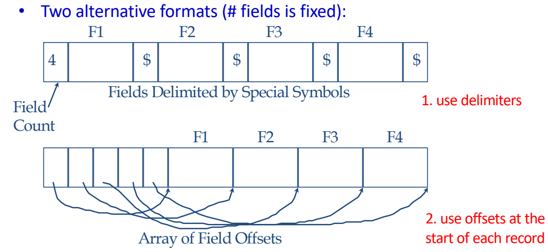
Record Formats: Fixed Length



- Each field has a fixed length
  - for all records
  - the number of fields is also fixed
  - fields can be stored consecutively
- Information about field types same for all records in a file
  - stored in system catalogs
- Finding i-th field does not require scan of record
  - given the address of the record, address of a field can be obtained easily

Record Formats: Variable Length

- Cannot use fixed-length slots for records
- Two alternative formats (# fields is fixed):



- Second offers direct access to i-th field, efficient storage of nulls (special don't know value); small directory overhead
- Modification may be costly (may grow the field and not fit in the page)

Indexes

Indexes

- An index on a file speeds up selections on the search key fields for the index
  - Any subset of the fields of a relation can be the search key for an index on the relation.
  - "Search key" is not the same as "key"
    key = minimal set of fields that uniquely identify a tuple
- An index contains a collection of data entries, and supports efficient retrieval of all data entries k* with a given key value k

Remember Terminology

- Index search key (key): k
  - Used to search a record
- Data entry : k*
  - Pointed to by k
  - Contains record id(s) or record itself
- Records or data
  - Actual tuples
  - Pointed to by record ids

INDEX does this

Alternatives for Data Entry **k*** in Index **k**

- In a data entry k* we can store:
  1. (Alternative 1) The actual data record with key value **k,** or
  2. (Alternative 2) <**k**, rid>
     - rid = record of data record with search key value **k**, or
  3. (Alternative 3) <**k**, rid-list>
     - list of record ids of data records with search key **k**>
- Choice of alternative for data entries is orthogonal to the indexing technique used to locate data entries with a given key value **k**

Alternatives for Data Entries: Alternative 1

- In a data entry k* we can store:
  1. The actual data record with key value **k**
  2. <**k**, rid>
     - rid = record of data record with search key value **k**
  3. <**k**, rid-list>
     - list of record ids of data records with search key **k**>

Advantages/ Disadvantages?

- Index structure is a file organization for data records
  - instead of a Heap file or sorted file
- How many different indexes can use Alternative 1?
- At most one index can use Alternative 1
  - Otherwise, data records are duplicated, leading to redundant storage and potential inconsistency
- If data records are very large, #pages with data entries is high
  - Implies size of auxiliary information in the index is also large

Alternatives for Data Entries: Alternative 2, 3

- In a data entry k* we can store:
  1. The actual data record with key value **k**
  2. <**k**, rid>
     - rid = record of data record with search key value **k**
  3. <**k**, rid-list>
     - list of record ids of data records with search key **k**>

Advantages/ Disadvantages?

- Data entries typically much smaller than data records
  - So, better than Alternative 1 with large data records
  - Especially if search keys are small.
- Alternative 3 more compact than Alternative 2
  - but leads to variable-size data entries even if search keys have fixed length.

Index Classification

- Primary vs. secondary
- Clustered vs. unclustered
- Tree-based vs. Hash-based

Primary vs. Secondary Index

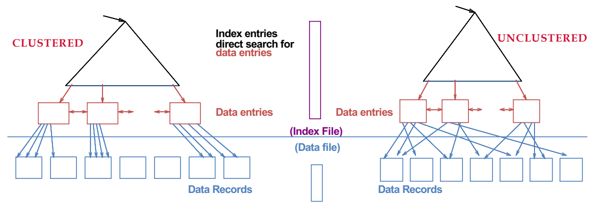
- If search key contains primary key, then called primary index, otherwise secondary
  - Unique index: Search key contains a candidate key
- Duplicate data entries:
  - if they have the same value of search key field k
  - Primary/unique index never has a duplicate
  - Other secondary index can have duplicates

Clustered vs. Unclustered Index

- If order of data records in a file is the same as, or `close to', order of data entries in an index, then clustered, otherwise unclustered
  - Alternative 1 implies clustered
  - Alternative 2, 3 are typically unclustered
    - unless sorted according to the search key
  - Sometimes, clustered also implies Alternative 1
    - since sorted files are rare
  - A file can be clustered on at most one search key
  - Cost of retrieving data records (range queries) through index varies greatly based on whether index is clustered or not

Clustered vs. Unclustered Index

- Suppose that Alternative (2) is used for data entries, and that the data records are stored in a Heap file
- To build clustered index, first sort the Heap file
  - with some free space on each page for future inserts
  - Overflow pages may be needed for inserts
  - Thus, data records are `close to', but not identical to, sorted



Methods for indexing

- Tree-based
- Hash-based
- (in detail later)

System Catalogs

- For each index:
  - structure (e.g., B+ tree) and search key fields
- For each relation:
  - name, file name, file structure (e.g., Heap file)
  - attribute name and type, for each attribute
  - index name, for each index
  - integrity constraints
- For each view:
  - view name and definition
- Plus statistics, authorization, buffer pool size, etc.
- (described in [RG] 12.1)

Catalogs are themselves stored as relations!