Today

- Finish Normalization from Lecture 5
- Start Database Internals

Recap
- Why redundancy is bad
- Functional dependencies
- Closure of attributes and functional dependencies

Boyce-Codd Normal Form (BCNF)

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

Acknowledgement:
- The following slides have been created adapting the instructor material of the [RG] book provided by the authors Dr. Ramakrishnan and Dr. Gehrke.
- Some slides have been adapted from slides by Prof. Jun Yang.

Normal Forms

- $R$ is in 4NF
  $\Rightarrow$ $R$ is in BCNF
  $\Rightarrow$ $R$ is in 3NF
  $\Rightarrow$ $R$ is in 2NF (a historical one)
  $\Rightarrow$ $R$ is in 1NF (every field has atomic values)

Only BCNF and 4NF are covered in the class.

Decomposition

- User id
- User name
- Twitter id
- Group id
- Joining Date (to a group)

Eliminates redundancy
- To get back to the original relation: $\Pi$

Next lecture: BCNF decomposition algorithm
Unnecessary decomposition

<table>
<thead>
<tr>
<th>attribute</th>
<th>foreign</th>
<th>table</th>
</tr>
</thead>
<tbody>
<tr>
<td>uid</td>
<td>Ralph</td>
<td>…</td>
</tr>
<tr>
<td></td>
<td>Lisa</td>
<td>…</td>
</tr>
<tr>
<td>twitterid</td>
<td>Milhouse</td>
<td>…</td>
</tr>
<tr>
<td></td>
<td>Bart</td>
<td>…</td>
</tr>
</tbody>
</table>


Fine: join returns the original relation

• Unnecessary: no redundancy is removed; schema is more complicated (and uid is stored twice!)

Lossless join decomposition

• Decompose relation \( R \) into relations \( S \) and \( T \)
  - \( \text{atrs}(R) = \text{atrs}(S) \cup \text{atrs}(T) \)
  - \( S = \pi_{\text{atrs}(S)}(R) \)
  - \( T = \pi_{\text{atrs}(T)}(R) \)

• The decomposition is a lossless join decomposition if, given known constraints such as FD’s, we can guarantee that \( R = S \bowtie T \)

• \( R \subseteq S \bowtie T \) or \( R \supseteq S \bowtie T \)?

• Any decomposition gives \( R \subseteq S \bowtie T \) (why?)
  - A lossy decomposition is one with \( R \nsubseteq S \bowtie T \)

BCNF decomposition algorithm

• Find a BCNF violation
  - That is, a non-trivial FD \( X \rightarrow Y \) in \( R \) where \( X \) is not a super key of \( R \)

• 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 \( Y \)

• Repeat until all relations are in BCNF

• Also gives a lossless decomposition!
BCNF decomposition example - 2

UserJoinsGroup (uid, uname, twitterid, gid, fromDate)

BCNF violation: uid → uname, twitterid

User (uid, uname, twitterid)

uid → uname, twitterid

Twitterid → uid

Member (uid, gid, fromDate)

uid, 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
  – BCNF decomposition is a lossless join decomposition
• BCNF: schema in this normal form has no redundancy due to FD’s

Multivalued dependencies

• A multivalued dependency (MVD) has the form
  \( X \rightarrow\!
  Y \), where \( X \) and \( Y \) are sets of attributes in a relation \( R \)
• \( X \rightarrow\!
  Y \) means that whenever two rows in \( R \) agree on all the attributes of \( X \), then we can swap their \( Y \) components and get two rows that are also in \( R \)

BCNF decomposition example - 3

UserJoinsGroup (uid, uname, twitterid, gid, fromDate)

BCNF violation: twitterid → uid

Userid (twitterid, uid)

BCNF

UserJoinedGroup (twitterid, uname, gid, fromDate)

BCNF violation: twitterid → uname

UserName (twitterid, uname)

BCNF

Member (twitterid, gid, fromDate)

BCNF

BCNF = no redundancy?

• User (uid, gid, place)
  – A user can belong to multiple groups
  – A user can register places she’s visited
  – Groups and places have nothing to do with other
  – FD’s?
    • None
    • BCNF?
      • Yes
      • Redundancies?
        • Tons!

MVD examples

User (uid, gid, place)

• uid → gid
• uid → place
  – Intuition: given uid, attributes gid and place are “independent”
• uid, gid → place
  – Trivial: LHS ∪ RHS = all attributes of \( R \)
• uid, gid → uid
  – Trivial: LHS ⊇ RHS
Complete MVD + FD rules

- FD reflexivity, augmentation, and transitivity
- MVD complementation: \( \text{If } X \rightarrow Y, \text{ then } X \rightarrow \text{attr}(R) - X - Y \)
- MVD augmentation: \( \text{If } X \rightarrow Y \text{ and } V \subseteq W, \text{ then } XW \rightarrow YV \)
- MVD transitivity: \( \text{If } X \rightarrow Y \text{ and } Y \rightarrow Z, \text{ then } X \rightarrow Z - Y \)
- Replication (FD is MVD): \( \text{If } X \rightarrow Y, \text{ then } X \rightarrow Y \)
- Coalescence: \( \text{If } X \rightarrow Y \text{ and } Z \subseteq Y \text{ and there is some } W \text{ disjoint from } Y \text{ such that } W \rightarrow Z, \text{ then } X \rightarrow Z \)

Try proving things using these?

An elegant solution: “chase”

- Given a set of FD’s and MVD’s \( \mathcal{D} \), does another dependency \( d \) (FD or MVD) follow from \( \mathcal{D} \)?
- Procedure
  - Start with the premise of \( d \), and treat them as “seed” tuples in a relation
  - Apply the given dependencies in \( \mathcal{D} \) repeatedly
    - If we apply an FD, we infer equality of two symbols
    - If we apply an MVD, we infer more tuples
  - If we infer the conclusion of \( d \), we have a proof
  - Otherwise, if nothing more can be inferred, we have a counterexample

Proof by chase

- In \( R(A, B, C, D) \), does \( A \rightarrow B \) and \( B \rightarrow C \) imply that \( A \rightarrow C \)?
  - Have: \[
  \begin{array}{cccc}
  A & B & C & D \\
  a & b_1 & c_1 & d_1 \\
  a & b_2 & c_2 & d_2
  \end{array}
  \]
  - Need: \[
  \begin{array}{cccc}
  A & B & C & D \\
  a & b_1 & c_1 & d_1 \\
  a & b_2 & c_2 & d_2
  \end{array}
  \]

Another proof by chase

- In \( R(A, B, C, D) \), does \( A \rightarrow B \) and \( B \rightarrow C \) imply that \( A \rightarrow C \)?
  - Have: \[
  \begin{array}{cccc}
  A & B & C & D \\
  a & b_1 & c_1 & d_1 \\
  a & b_2 & c_1 & d_2
  \end{array}
  \]
  - Need: \[
  \begin{array}{cccc}
  A & B & C & D \\
  a & b_1 & c_1 & d_1 \\
  a & b_2 & c_2 & d_2
  \end{array}
  \]
  \( c_1 = c_2 \)

Counterexample by chase

- In \( R(A, B, C, D) \), does \( A \rightarrow BC \) and \( CD \rightarrow B \) imply that \( A \rightarrow B \)?
  - Have: \[
  \begin{array}{cccc}
  A & B & C & D \\
  a & b_1 & c_1 & d_1 \\
  a & b_2 & c_2 & d_2
  \end{array}
  \]
  - Need: \( b_1 = b_2 \)

4NF

- A relation \( R \) is in Fourth Normal Form (4NF) if
  - For every non-trivial MVD \( X \rightarrow Y \) in \( R \), \( X \) is a superkey
  - That is, all FD’s and MVD’s follow from “key → other attributes” (i.e., no MVD’s and no FD’s besides key functional dependencies)
- 4NF is stronger than BCNF
  - Because every FD is also a MVD
4NF decomposition algorithm

- Find a 4NF violation
  - A non-trivial MVD $X \rightarrow Y$ in $R$ where $X$ is not a superkey
- 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 $R$ attributes not in $X$ or $Y$)
- Repeat until all relations are in 4NF
- Almost identical to BCNF decomposition algorithm
- Any decomposition on a 4NF violation is lossless

4NF decomposition example

<table>
<thead>
<tr>
<th>User (uid, gid, place)</th>
<th>4NF violation: ( \text{uid} \rightarrow \text{gid} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{uid} )</td>
<td>( \text{gid} )</td>
</tr>
<tr>
<td>142</td>
<td>dps</td>
</tr>
<tr>
<td>456</td>
<td>abc</td>
</tr>
<tr>
<td>456</td>
<td>gov</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Member (uid, gid)</th>
<th>4NF</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{uid} )</td>
<td>( \text{gid} )</td>
</tr>
<tr>
<td>142</td>
<td>dps</td>
</tr>
<tr>
<td>456</td>
<td>abc</td>
</tr>
<tr>
<td>456</td>
<td>gov</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Visited (uid, place)</th>
<th>4NF</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{uid} )</td>
<td>( \text{place} )</td>
</tr>
<tr>
<td>142</td>
<td>Springfield</td>
</tr>
<tr>
<td>456</td>
<td>Springfield</td>
</tr>
<tr>
<td>456</td>
<td>Morocco</td>
</tr>
</tbody>
</table>

Summary

- Philosophy behind BCNF, 4NF:
  - Data should depend on the key, the whole key, and nothing but the key!
  - You could have multiple keys though
- Redundancy is not desired typically
  - not always, mainly due to performance reasons
- Functional/multivalued dependencies – capture redundancy
- Decompositions – eliminate dependencies
- Normal forms
  - Guarantees certain non-redundancy
  - BCNF, and 4NF
- Lossless join
- How to decompose into BCNF, 4NF
- Chase

Other kinds of dependencies and normal forms

- Dependency preserving decompositions
- Join dependencies
- Inclusion dependencies
- 5NF, 3NF, 2NF
- See book if interested (not covered in class)

Where are we now?

- We learnt
  - Relational Model and Query Languages
  - SQL, RA, HC
  - PostgreSQL (DBMS)
  - XML (overview)
  - HW1
- Database Normalization

Next

- DBMS Internals
  - Storage
  - Indexing
  - Query Evaluation
  - Operator Algorithms
  - External sort
  - Query Optimization
Reading Material

- [RG]
  - Storage: Chapters 8.1, 8.2, 8.4, 9.4-9.7
  - Index: 8.3, 8.5
  - Tree-based index: Chapter 10.1-10.7
  - Hash-based index: Chapter 11

Additional reading
- [GUW]
  - Chapters 8.3, 14.1-14.4

Acknowledgement:
The following slides have been created adapting the instructor material of the [RG] book provided by the authors Dr. Ramakrishnan and Dr. Gehrke.

What will we learn?

- How does a DBMS organize files?
  - Record format, Page format

- What is an index?

- What are different types of indexes?
  - Tree-based indexing:
    - B+ tree
    - insert, delete
  - Hash-based indexing
    - Static and dynamic (extendible hashing, linear hashing)

- How do we use index to optimize performance?

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

When a Page is Requested ...
For every frame, store
• a dirty bit:
  – whether the page has been modified since it has been brought to memory
  – initially 0 or off
• a pin-count:
  – the number of times a page has been requested but not released (and no. of current users)
  – initially 0
  – when a page is requested, the count is 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 page with pin-count 0, buffer manager has to wait (or a transaction is aborted – later)

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.
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

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
9/12/17

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)
  - <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
  - not attractive for fixed-length records too
- Store (record-offset, record-length) in each slot
- rid is 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 (if fields is fixed):
  
  1. use delimiters
  
  2. use offsets at the start of each record

- 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)