L07: Advanced Topics in SQL and Database Design

DSAN 6300/PPOL 6810: Relational Databases and SQL Programming

Irina Vayndiner

October 12 & 16 2023

GEORGETOWN UNIVERSITY

Agenda for today's class

- Logistics:
  - Midterm is **Thu, 10/19 and Mon, 10/23**
    - No late (or early) dates. No zoom option.
    - Closed books
  - HW2
    - Answers will be posted as soon as ALL students submit
      - Planning for Wed, 10/18
  - HW3 will be posted on Mon, 10/23
    - Due Tue 11/7
    - Lab from today is due after the mid-term
      - For Thu section: Tue, 10/24
      - For Mon section: Fri, 10/27
- Today:
  - Lecture: Advanced SQL and Database Design
  - Lab

Outline: Advanced Topics in SQL and Database Design

- Advanced SQL
  - Recursive queries
  - Advanced SQL Functions, including windowing
- Normalization
  - BCNF
  - 3NF
- Indexes (if time allows)
  - Clustered and Unclustered
  - B+-Tree
  - Bitmap

Advanced SQL Features

Recursive Queries

- Let's look at the instance of the relation *prereq.*
- It contains information about the various courses offered at the university, and the prerequisite for each course

course_id	*prereq_id*
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-190
CS-319	CS-101
CS-319	CS-315
CS-347	CS-319

- Suppose now that we want to find out which courses are a prerequisite *whether directly or indirectly*, for any course
- Since the depth of the list of pre-requisites is not limited, it takes either iterative (using stored procedures) or recursive algorithms to find all pre-requisites

Recursive Query (RQ) Structure

- The SQL standard supports recursion, using the **with recursive** clause, where a view (or *temporary view*) is expressed in terms of itself:

```
with recursive rq_name ([ col_1, … col_n]) as
(
  select  …# base
  union
  select …from rq_name … # recursive
)
select *
from rq_name
where …;
```

- The Recursive Query (RQ) has two parts separated by **union** or **union distinct**
- The first **select** (base) produces the initial row or rows for the RQ and does *not* refer to the rq_name.
- The second **select** (recursive) produces additional rows and *recurses* by referring to the rq_name in its FROM clause.
  - Recursion ends when the recursive select produces no new rows.

Recursive Query Example for pre-requisite problem

```
with recursive rec_prereq(course_id, prereq_id) as (
        select course_id, prereq_id
        from prereq
    union
        select rec_prereq.course_id, prereq.prereq_id
        from rec_prereq, prereq
        where rec_prereq.prereq_id = prereq.course_id
    )
select *
from rec_prereq;
```

- We first find all direct prerequisites of each course by executing the base query.
- On each recursive step, the recursive query adds one more level of courses in each iteration, until the maximum depth of the course-prereq relationship is reached
- In this example the view, *rec_prereq*, is called the **transitive closure** of the *prereq* relation

Advanced Functions: Rank

- **rank()** function finds the position of a value within a result set
- For instance, we may wish to assign students a rank in class based on their GPA, with the rank 1 going to the student with the highest GPA, the rank 2 to the student with the next highest GPA, and so on
- To illustrate ranking, let us assume we have a view

  student_grades (ID, GPA)

  with the grade-point average of each student
- Ranking is done by the attributes (or expressions) specified in **over** (**order by)** clause. In our example, for GPA
  - The following query gives the rank of each student:

    ```
    select ID, rank() over (order by (GPA) desc) as s_rank
    from student_grades
    order by s_rank;
    ```

  - **rank()** function can be used with **order by** (e.g. s_rank) to sort result rows into the desired order.

Advanced Functions: Rank (continued)

- The general syntax of the rank function is

  rank() **over** (**order by** *expr* [**asc**|**desc**] [, *expr* [**asc**|**desc**]] ...)

  - Each **order by** expression optionally can be followed by **asc** or **desc** to indicate sort direction.
    - The default is **asc** if no direction is specified.
    - **null** values sort order: first for ascending sorts, last for descending sorts.
- The **rank** function gives the same rank to all tuples that are equal on the **order by** attributes.
  - E.g. 1,1,1,4,4, 6
- There are other useful advanced functions, e.g. **lag**(*expr* , N) and **lead**(*expr*, N) that return the value of *expr* from the row that lags (precedes) and that leads (follows) the current row by *N* rows.
  - Queries more than one row in a table at a time without having to join the table to itself
    - Support of these functions varies significantly for different DBMS; therefore see the documentation.

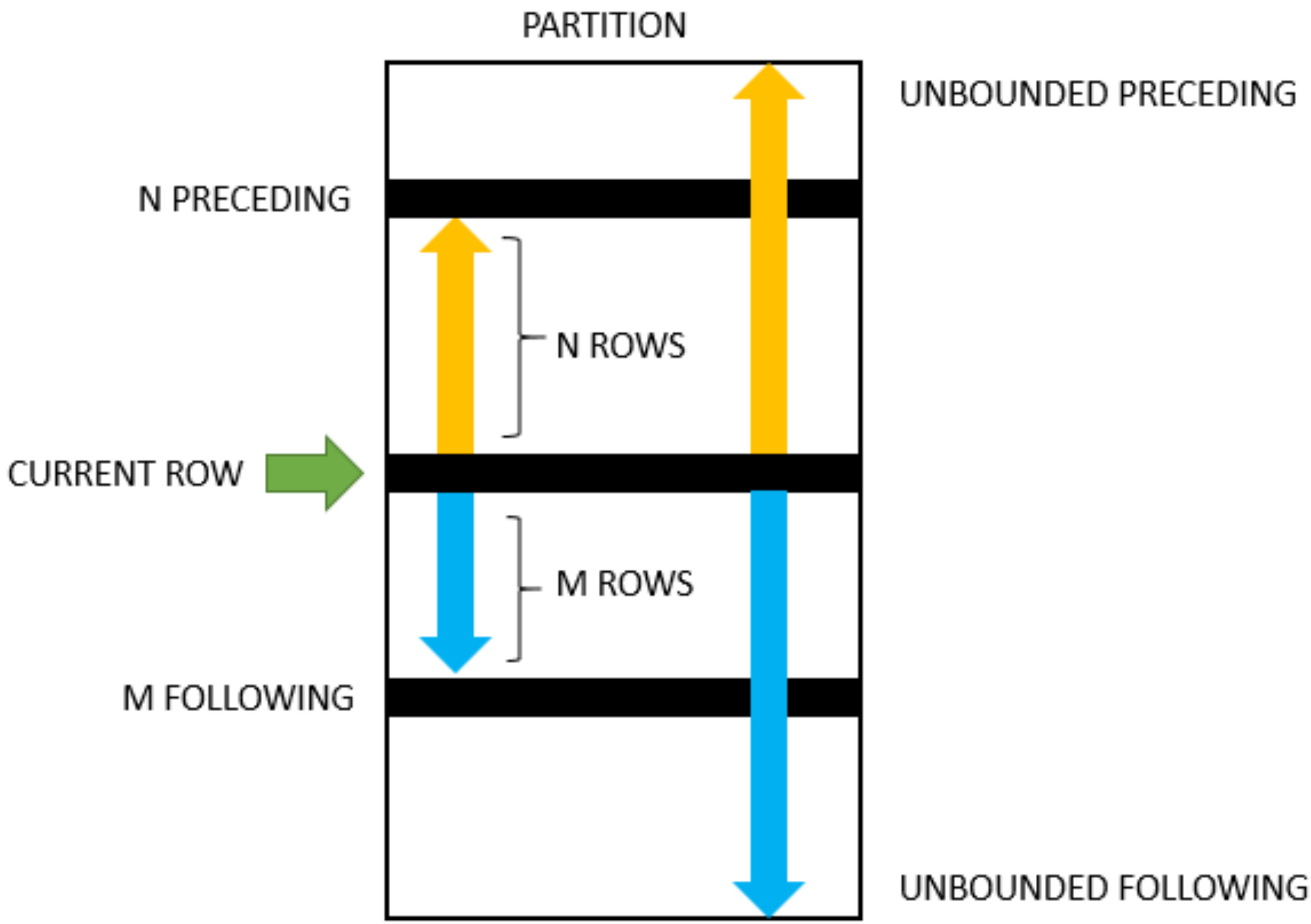
Advanced Aggregation: Window(ing) Functions

- Window functions compute an aggregate **over ranges of tuples**
  - Access to data in the records right before and after the current record.
  - A set of rows to which the this aggregation function applies is referred to as a window.
    - Useful, for example, when calculating moving averages
- Window function defines a **frame** or **window** of rows with a given length around the current row, and performs a calculation across the set of data in the window
  - Useful, for example, to compute an aggregate of a fixed range of time.
- Unlike regular aggregate functions, use of a window function does **not** cause rows to become grouped into a single output row — the rows retain their separate identities.
- Windows may overlap, in which case a tuple may contribute to more than one window.

Advanced Aggregation: Windowing (continued)

- General Syntax: OVER ([partition] [order] [frame])
- **Partition row ordering:** defines the subset of the rows that can be considered for a window
  - E.g. partition by year or by dept_num
- **Frame (window):** defined with respect to the current row
  - Allows a frame to move within a partition depending on the position of the current row within its partition.
- The offsets of the current row and frame rows are the row numbers if the frame unit is ROWS and row values if the frame unit is RANGE.
- Q: why **order** is needed?
- frame: {*frame_start* | *frame_between*}
- frame_between:
  BETWEEN *frame_start* AND *frame_end*
- *frame_start*, *frame_end*: {
  CURRENT ROW
  | UNBOUNDED PRECEDING
  | UNBOUNDED FOLLOWING
  | *expr* PRECEDING
  | *expr* FOLLOWING }

Advanced Aggregation: Windowing (continued)



Advanced Aggregation: Windowing Example

- Suppose there is a view

  tot_credits (year, num_credits)

  with the total number of credits taken by **all** students in each year.

  Q: How many tuples per year in this view per year? (1)

- Suppose that *for each year*, we need to compute average number of credits *over three preceding years*.

```
select year, avg(num_credits)
over (order by year rows 3 preceding)
as avg_total_credits
from tot_credits;
```

- Most aggregate functions (avg, sum, min, max, etc.) can be used as window functions.
  - See mysql window documentation for details: https://docs.oracle.com/cd/E17952_01/mysql-8.0-en/window-functions-usage.html
  - Note: expression **over (order by) is** similar to what we used in rank()

Normalization

Redundancy Issues Overview

- **Redundancy** is at the root of several problems associated with relational schemas

- Redundant Storage where some information is stored repeatedly.
  - Leads to Increased storage costs
- Update Anomalies
  - If one copy of such repeated data is updated, an inconsistency is created unless all copies are similarly updated.
- Insertion Anomalies
  - It may not be possible to store certain information unless some other, unrelated, information is stored as well.
- Deletion Anomalies
  - It may not be possible to delete certain information without losing some other, unrelated, information as well.

Redundancy Example

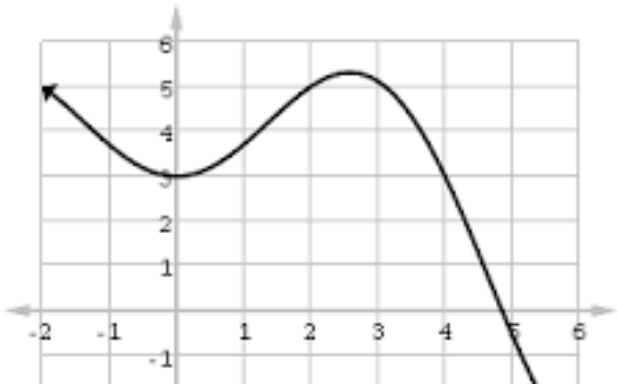
- Suppose we combine *instructor* and *department* into *in_dep,* which represents the natural join on the relations *instructor* and *department*

ID	*name*	*salary*	*dept_name*	*building*	*budget*
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El Said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000
76766	Crick	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
83821	Brandt	92000	Comp. Sci.	Taylor	100000
15151	Mozart	40000	Music	Packard	80000
33456	Gold	87000	Physics	Watson	70000
76543	Singh	80000	Finance	Painter	120000

- For *each department* information in *building* and *budget* columns is repeated
- Update anomaly: What if budget for Physics dept changed?
- Deletion anomaly: If we delete all instructors from Physics department, we lose the information about its building and budget!
- Insertion anomaly: What if we want to insert an *instructor*, but don't know the *budget* for his department?

Functional Dependencies (FD) Definitions

- Mathematically, redundancy is expressed through the concept of **functional dependency (FD)**



- Value of an argument X determines uniquely the value of a function F(X)

- In Relational Algebra: the value for a certain set of attributes determines uniquely the value for another set of attributes.

Functional Dependencies (continued)

- Formally, let R be a schema

$$\alpha \subseteq R \text{ and } \beta \subseteq R$$

- The **functional dependency**

$$\alpha \rightarrow \beta$$

holds on R *if and only if* for any relations $r(R)$, whenever any two tuples t_1 and t_2 of r agree on the attributes α, they also agree on the attributes β.

That is, $t_1[\alpha] = t_2[\alpha] \Rightarrow t_1[\beta] = t_2[\beta]$

ID	*name*	*salary*	*dept_name*	*building*	*budget*
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El Said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000
76766	Crick	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
83821	Brandt	92000	Comp. Sci.	Taylor	100000
15151	Mozart	40000	Music	Packard	80000
33456	Gold	87000	Physics	Watson	70000
76543	Singh	80000	Finance	Painter	120000

α β

FD/Redundancy and Normal Forms

- Role of FDs in detecting redundancy:
  - Consider a relation R with 3 attributes A, B, C.
    - If no FDs, then there is no redundancy here.
    - FD A->B, then several tuples could have the same A value, and if so, they'll all will have the same B value

- If a relation is in a certain **normal form,** defined via functional dependencies, then redundancy-related problems are avoided/minimized.

- We will look today at two normal forms
  - Boyce-Codd Normal Form (BCNF)
  - Third Normal Form (3NF)

Boyce-Codd Normal Form (BCNF)

- A schema *R* is in BCNF, if for all functional dependencies of the form $\alpha \rightarrow \beta$

  where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:
  - $\alpha \rightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$)
  - α is a superkey for *R*
- It means, as Bill Kent jokingly put it, that in a BCNF relation each attribute is uniquely defined by
  - the (super) key
  - the whole key
  - and nothing but the key
  - (so help me Codd ☺)

Kent, William. "A Simple Guide to Five Normal Forms in Relational Database Theory", *Communications of the ACM* 26 (2), Feb. 1983, pp. 120–125.

Boyce-Codd Normal Form (BCNF) Example

- Example of schema that is **NOT** in BCNF

  in_dep (*ID*, *name, salary, dept_name, building, budget*)

ID	*name*	*salary*	*dept_name*	*building*	*budget*
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El Said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000
76766	Crick	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
83821	Brandt	92000	Comp. Sci.	Taylor	100000
15151	Mozart	40000	Music	Packard	80000
33456	Gold	87000	Physics	Watson	70000
76543	Singh	80000	Finance	Painter	120000

α β

because

- There is FD *dept_name* $\rightarrow$ *building, budget*
- BUT *dept_name* is NOT a superkey

Decomposition into BCNF

- If a schema is in BCNF, there is no redundancy
- But what if it is not in BCNF?
  - The idea is to *decompose* it into several schemas, ideally, each a BCNF schema.
- In our example

  in_dep (*ID,* *name, salary, dept_name, building, budget*)

  can be decomposed into

  instructor(*ID*, *name, salary, dept_name*)

  department (*dept_name, building, budget*)

- Is it always possible? How can we do it?

Problems with Decompositions

- There are three potential problems to consider:
  - (1) Loss of information: Given instances of the decomposed relations, we may not be able to reconstruct the corresponding instance of the original relation!
    - Unacceptable
  - (2) Dependency Preservation: Checking some dependencies may require joining the instances of the decomposed relations.
    - Undesirable
  - (3) Some queries become more complex. It may be required to join the tables back to get the information
    - Depends on business requirements
- Let's look into each of these problems in more details

(1) Loss of Information

- Decomposition may lead to the loss of information
- Consider an example where we choose to decompose the schema

  employee (ID, name, street, city, salary)

  into the following two schemas:

  employee1 (ID, name)

  employee2 (name, street, city, salary)
- In order to query the database we now need to do a join of these two schemas

Loss of Information (continued)

ID	name	street	city	salary
⋮				
57766	Kim	Main	Perryridge	75000
98776	Kim	North	Hampton	67000
⋮				

employee

ID	name
⋮	
57766	Kim
98776	Kim
⋮	

name	street	city	salary
⋮			
Kim	Main	Perryridge	75000
Kim	North	Hampton	67000
⋮			

natural join

ID	name	street	city	salary
⋮				
57766	Kim	Main	Perryridge	75000
57766	Kim	North	Hampton	67000
98776	Kim	Main	Perryridge	75000
98776	Kim	North	Hampton	67000
⋮				

- The join result has **lost information** about which employee identifiers correspond to which names, addresses and salaries!

Lossless Decomposition definition

- Let R be a schema and let R_1 and R_2 form a decomposition of R . That is R = $R_1 \cup R_2$
- The decomposition is a **lossless decomposition** if there is no loss of information by replacing R with two schemas $R_1 \cup R_2$
- That is, if for any relation r(R)

$$\prod_{R_1}(r) \bowtie \prod_{R_2}(r) = r$$

- And, conversely a decomposition is lossy if

$$r \subset \prod_{R_1}(r) \bowtie \prod_{R_2}(r)$$

Example of Lossless Decomposition

- Decomposition of *R = (A, B, C) into*
  $R_1 = (A, B)$ *and* $R_2 = (B, C)$

A	*B*	*C*
α	1	A
β	2	B

r

A	*B*
α	*1*
β	*2*

$\Pi_{A,B}(r)$

B	*C*
1	A
2	B

$\Pi_{B,C}(r)$

$\Pi_{A,B}(r) \bowtie \Pi_{B,C}(r)$

A	*B*	*C*
α	1	A
β	2	B

(2) Dependency Preservation

- Functional Dependency is a constraint
- Enforcing that functional dependency constraint each time the database is updated can be costly
  - It is useful to design the database in a way that constraints can be enforced efficiently
- If enforcing a functional dependency can be done by considering just one relation, then the cost of enforcing this constraint is low
- When decomposing a relation, attributes that are parts of the same FD may get into different schemas.
  - In that case enforcing of FD will require to perform a Cartesian product, which is costly
- A decomposition that makes it computationally hard to enforce functional dependency is said to be **NOT** **dependency preserving**.

Dependency Preservation Example

- Consider a schema:

  dept_advisor(s_ID, i_ID, dept_name)

  Assume that each student can have no more than one instructor from any department. Thus, we have functional dependencies:

  $i\_ID \rightarrow dept\_name$

  $s\_ID, dept\_name \rightarrow i\_ID$

- In the above design we are forced to repeat the department name once for each time an instructor participates in a *dept_advisor* relationship.
  - To fix this, we need to decompose *dept_advisor*
- Any decomposition will not include all three original attributes in one schema. Therefore, the attributes from FD:

  $s\_ID, dept\_name \rightarrow i\_ID$

  will need to be in different schemas
- In this example, any decomposition will NOT be functional dependency preserving

Goals of Normalization

- The process of reducing redundancy is call **normalization**
- Ideally, our goals are:
  - BCNF (no redundancy)
  - Losslessness
  - Dependency preservation

- In a general case, it is not possible to satisfy all three!
  - Need to choose **two**

- Since loss of information is not acceptable, available options are:
  - Losslessness and BCNF
  - Losslessness and dependency preservation

Losslessness + BCNF

- It is always possible to do a Lossless decomposition into BCNF as:
  - If a schema R_i is not in BCNF and has a FD $\alpha \rightarrow \beta$, we substitute R_i with two schemas ($R_i - \beta$) and (α, β)
  - Repeat until all schemas are in BCNF
  - Now each R_i is in BCNF, and decomposition is lossless.
- Example: schema that is **NOT** in BCNF
  - We saw it earlier today:

    in_dep (*<u>ID,</u> name, salary, dept_name, building, budget*)

    with FD *dept_name* $\rightarrow$ *building, budget*

  can be decomposed into:

    instructor (*<u>ID,</u> name, salary, dept_name*)

    dept (*<u>dept_name</u>, building, budget*)

Losslessness + Dependency preservation Third Normal Form (3NF)

- It is often a good compromise to keep dependency preservation vs BCNF
- 3NF is a minimal relaxation of BCNF to ensure dependency preservation
- A relation schema *R* is in **third normal form (3NF),** if for all $\alpha \rightarrow \beta$ at least one of the following holds:
  - $\alpha \rightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$)
  - α is a superkey for *R*
  - Each attribute *A* in $\beta - \alpha$ is contained in a *candidate* key for *R.*
    - Note*:* each attribute may be in a different candidate key
- Note: If a relation is in BCNF it is in 3NF
  - Since in BCNF one of the first two conditions above must hold
- 3NF imposes “dependency on the key” condition similar to BCNF
  - But not ALL attributes need to depend on a key as was in BCNF
    - Attributes that are contained in **any** candidate key are exempt (called prime attributes)

3NF Example

- Consider following schema for a relation that records only courses that a student passed

  student_course(s_ID, course_ID, ssn)

- Two candidate keys are:
  - {*s_ID, course_ID*}
  - {*ssn, course_ID* }
- Function dependency:

  ssn$\rightarrow$ s_*ID*

- *student_course* is **not** in BCNF since
  - *s_ID* depends on *ssn*, but *ssn* is NOT a candidate key by itself
- *student_course,* however, is in 3NF
  - since *ssn* is a part of a candidate key {*ssn*, *course_ID* }

- Note: There is a redundancy in *student_course* since if a student passed several courses her *ssn* will be present multiple times

Comparison of BCNF and 3NF

- Advantages of 3NF vs BCNF
  - It is always possible to obtain a 3NF design without sacrificing losslessness and dependency preservation.
  - Practically, it is much more often used as a requirement for database design
- Disadvantages of 3NF vs BCNF
  - Some redundancy is allowed
    - Some information is repeated
    - We may have to use null values to represent some of the possible meaningful relationships among data items.

(3) Over-normalization

- We already mentioned a potential problem with normalization
  - Some queries become more complex, since they require to join the tables back to get the information.
- Consider an example:
  - Address entity:

    address (firstname, lastname, number, street, city, state, zip)

    (Bob, Jones, 12, Main Street, Springfield, IL, 12345)
  - There is a FD (*number, street, city, state) -> zip*
  - Decomposition to bring to BCNF is
    - (*firstname, lastname, number, street, city, state*) and
    - (*number, street, city, state, zip)*
- Do we really want to do this?
  - So what if zip codes are a bit redundant?
  - Insert, update, delete anomalies are manageable

Normalization Takeaways

- What normalization does for database design
  - Reduces redundancy
  - Improves data consistency (by having only one copy)
  - Improves concurrency (less needs to be “locked”)
- Potential issues with normalization
  - Requiring (potentially lossy) joins to put your data back together
  - Requiring you to understand all FDs before you can design and use your database (can be an issue for Big Data)
  - This takes time (doesn’t lend itself to rapid prototyping).
- Normalization is a tool (like ER design).
  - Designing great databases is up to you! ☺

Indexing

Indexing: Basic Concepts

- An **index** is a data structure that organizes data records (typically) on a disk to optimize retrieval operations.
- It allows to efficiently retrieve all records that satisfy search conditions on the **search-key**
- **Search-Key** is a set of attributes to look up records in a file.
- An **index file** consists of records (called **index entries**) of the form

search-key	pointer

- Basic types of indices:
  - **Ordered indices:** search-keys are stored in sorted order
  - **Hash indices:** search-keys are distributed uniformly across "buckets" using a hash function.
  - **Bitmap index**

Ordered Index Types

- **Primary vs. secondary**: If search-key contains primary key, then it is called a primary index.
  - **Unique** index: Search key contains a candidate key.
- **Clustered vs. unclustered:** If order of data records (typically on disk) is the same as, or 'close to', order of index entries (both ordered by search key), then it is called clustered index.
  - A file can be clustered on at most one search-key.
  - Cost of retrieving data records through index varies *greatly* based on whether index is clustered or not!

Clustered Index Example

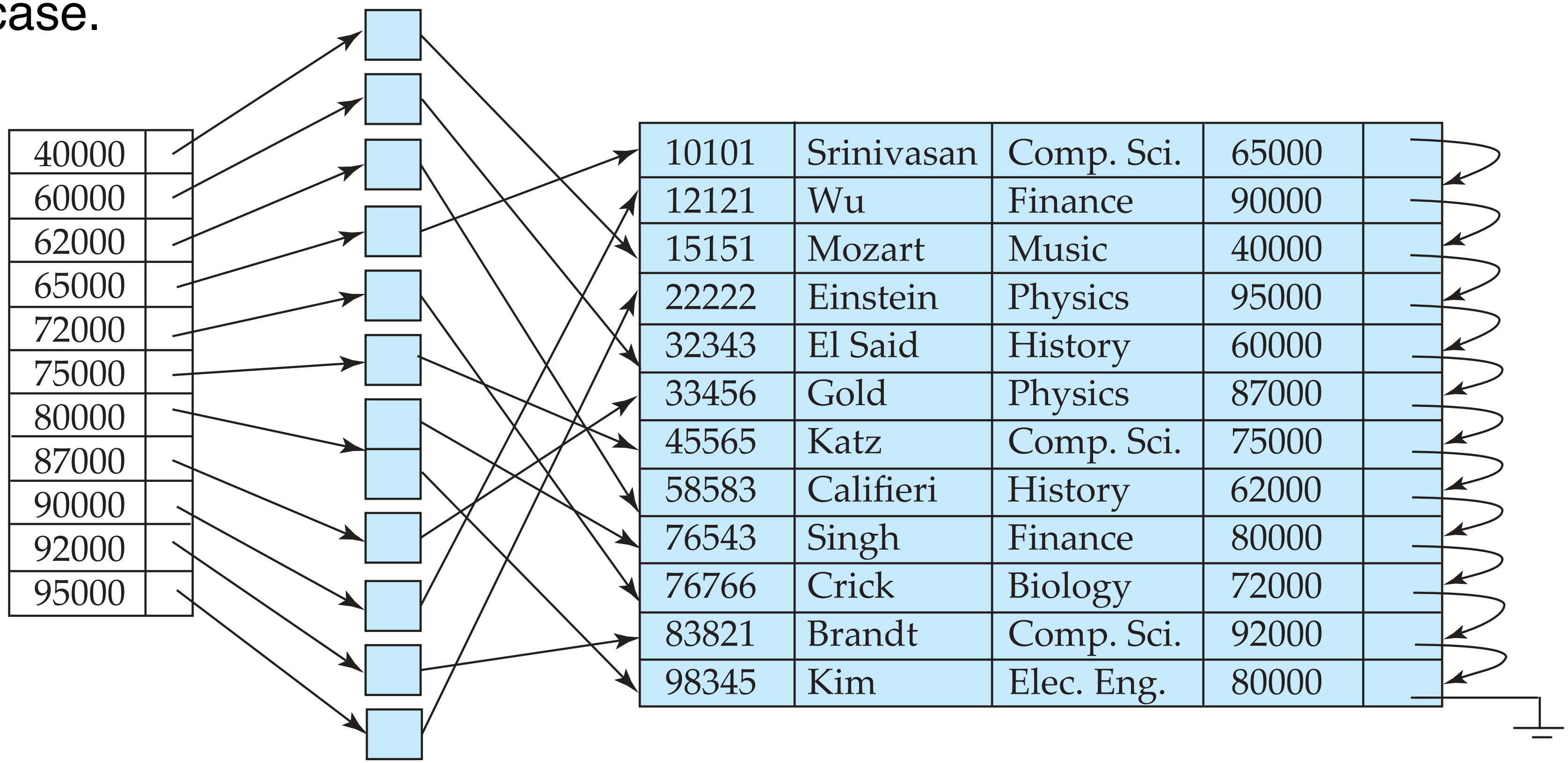
- Let's choose to index on *dept_name*, with *instructor* file sorted on *dept_name*

Biology	
Comp. Sci.	
Elec. Eng.	
Finance	
History	
Music	
Physics	

76766	Crick	Biology	72000	
10101	Srinivasan	Comp. Sci.	65000	
45565	Katz	Comp. Sci.	75000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	
12121	Wu	Finance	90000	
76543	Singh	Finance	80000	
32343	El Said	History	60000	
58583	Califieri	History	62000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
33465	Gold	Physics	87000	

Non-clustered Secondary Index Example

- Primary clustered index is based on IDs
- Secondary Index points to a bucket that contains pointers to all the actual records with that particular search-key value, salary in this case.

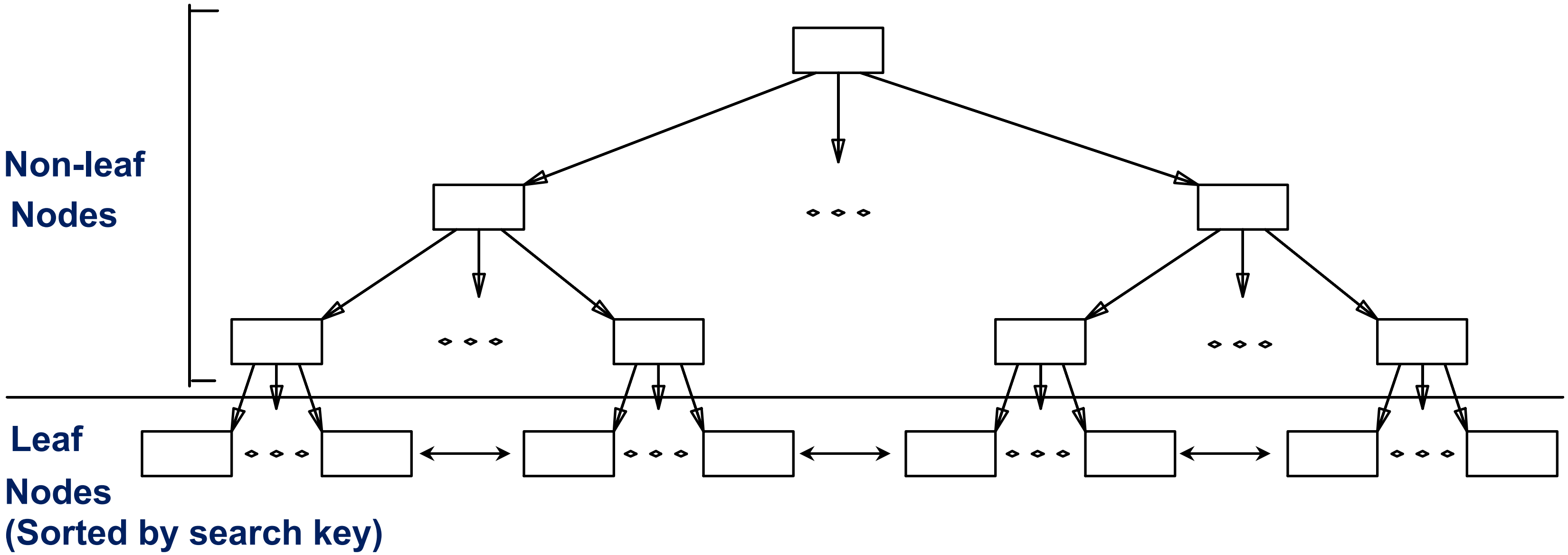


Secondary index on *salary* field of *instructor*

B^+-Tree Index Files

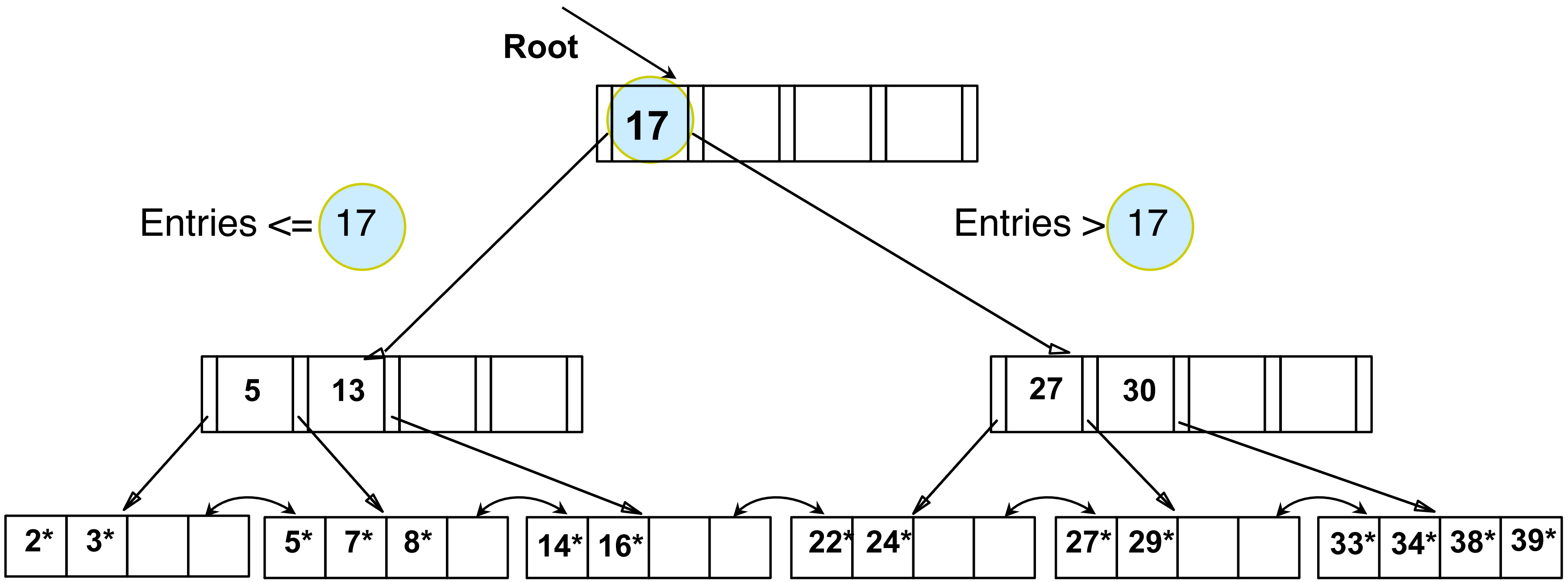
- The main disadvantage of the index-sequential file organization that we considered so far that performance degrades as the file grows.
  - Although this degradation can be remedied by reorganization of the file, frequent reorganizations are undesirable.
- The **B+-tree index** structure is the most widely used of several index structures that maintain their efficiency despite insertion and deletion of data.
- A B+-tree index takes the form of a balanced tree in which every path from the root of the tree to a leaf of the tree is of the same length.

B+ Tree Index Structure



- Non-leaf nodes have *index entries;* used only to direct searches
- Leaf nodes contain *data entries*

Example B+ Tree



Note that data entries in leaf level are sorted
(*) marks actual value

- Find 24*? 29*? All > 5* and < 8*?
- Insert/delete: Find data entry in leaf, then change it. Need to adjust parent sometimes.
  - And change sometimes bubbles up the tree

Hash-Based Indexes

- Records are grouped into *buckets.*
- A **bucket** is a unit of storage containing one or more entries (a bucket is typically a disk block for clustered index).
  - We obtain a bucket from its search-key value using a **hash function**
- Hash function *h* is a function from the set of all search-key values *K* to the set of all bucket addresses *B.*
- Hash function is used to locate entries for access, e.g. insertion and deletion.
- Entries with different search-key values may be mapped to the same bucket; thus entire bucket has to be searched sequentially to locate an entry.

Example of Hash Organization

bucket 0

bucket 1

15151	Mozart	Music	40000

bucket 2

32343	El Said	History	80000
58583	Califieri	History	60000

bucket 3

22222	Einstein	Physics	95000
33456	Gold	Physics	87000
98345	Kim	Elec. Eng.	80000

bucket 4

12121	Wu	Finance	90000
76543	Singh	Finance	80000

bucket 5

76766	Crick	Biology	72000

bucket 6

10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

bucket 7

Hash file organization of *instructor* file, using *dept_name* as key.

The hash function returns the sum of the binary representations of the characters modulo 10

- E.g. h(Music) = 1 h(History) = 2 h(Physics) = 3 h(Elec. Eng.) = 3

Bitmap Indices

- **Bitmap indices** are a special type of index designed for efficient querying on columns that have relatively small number of distinct values (or, low-cardinality)
  - E.g. gender, country, state, …
  - E.g. income-level
    - income broken up into a small number of levels such as 0-9999, 10000-19999, 20000-50000, 50000- infinity

- Records in a relation are assumed to be numbered sequentially from, say, 0
  - Given a number *n* it must be easy to retrieve record *n*
    - Particularly easy if records are of fixed size

- A bitmap is simply an array of bits

Bitmap Indices (continued)

- In its simplest form, a bitmap index on an attribute has a bitmap for each value of the attribute
  - Bitmap has as many bits as the number of records
  - In a bitmap for value v, the bit for a record is 1 if the record has the value v for the attribute, and is 0 otherwise

record number	*ID*	*gender*	*income_level*
0	76766	m	L1
1	22222	f	L2
2	12121	f	L1
3	15151	m	L4
4	58583	f	L3

Bitmaps for *gender*

m	10010
f	01101

Bitmaps for *income_level*

L1	10100
L2	01000
L3	00001
L4	00010
L5	00000

Bitmap Indices (continued)

- Queries are processed using bitmap operations
  - Intersection (and)
  - Union (or)
- Each operation takes two bitmaps of the same size and applies the operation on corresponding bits to get the result bitmap
  - E.g. 100110 AND 110011 = 100010

    100110 OR 110011 = 110111

    NOT 100110 = 011001
  - Example: Males with income level L1: 10010 AND 10100 = 10000
    - Can then easily retrieve required tuples.
    - Counting number of matching tuples is fast

Bitmap Indices Summary

- Bitmap indices have generally small size, e.g. compared with relation size
- Bitmap indexes are used on the columns which have low number of unique values (low cardinality) while B-tree indexes used for high cardinality. Relation can have both types.
- Bitmaps don't do well when need to update the bitmap-indexed column often
- Ok for inserts but not great for deletes and updates
  - Often that is enough to work with Big Data

Index Definition in SQL (reminder)

- Create an index

  create index <index-name> **on** <relation-name>
  (<attribute-list>)

  E.g.: **create index** *b-index* **on** *branch(branch_name)*

- Use **create unique index** to indirectly specify and enforce the condition that the search key is a candidate key.
  - Not required if SQL **unique** integrity constraint is supported
- To drop an index

  drop index <index-name>

- Most database systems allow specification of type of index, and clustering type.
- Indices on primary key are created automatically by most databases
- Some databases also create indices on foreign key attributes

Indexes Analysis

- Indices can greatly speed up reads (lookups)
- Cost:
  - There is a cost of maintaining the index
    - e.g. on updates and deletes
  - There is storage cost
- What indexes do we need?
  - Which relations should have indexes?
  - What field(s) should be the search key?
  - Should we build several indexes?
- For each index, what kind of an index should it be?
  - Clustered? Hash/tree? Bitmap?
- Starting point
  - Consider the most important queries
  - Plan for less indices, add more if/when needed

Index Selection Best Practices

- Attributes in WHERE clause are candidates for index keys.
  - Exact match condition suggests hash index.
  - Range query suggests tree index.
    - Clustering is especially useful for range queries; can also help on equality queries if there are many duplicates.
- Multi-attribute search keys should be considered when a WHERE clause contains several conditions.
- Try to choose indexes that benefit as many queries as possible.
- Since only one index can be clustered per relation, choose it based on important queries that would benefit the most from clustering.

Index Consideration Summary

- Understanding the nature of the *workload* for the application, and the performance goals, is essential to developing a good design.
  - What are the important queries and updates?
  - What attributes/relations are involved?
- Indexes must be chosen to speed up important queries (and perhaps some updates!).
  - Choose indexes that can help many queries, if possible.
  - Clustering is an important decision; only one index on a given relation can be clustered!
  - Order of fields in composite index key *can* be important.
  - Consider index maintenance overhead on updates to key fields.

Lab 07: Problems to submit are marked so