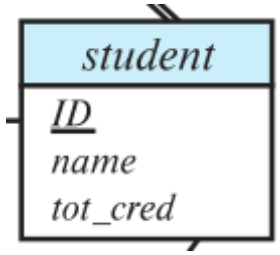
Outline

- Reducing ER Diagrams to Relational Tables
- Extended E-R Features
- Entity-Relationship Design Issues
- Normalisation
  - Decomposition
  - Normal Forms
- Data warehouse
- File organization
- Indexing
- NoSQL
  - ACID vs BASE
  - Sharding

Representing Entity Sets

- A strong entity set reduces to a table with the same attributes



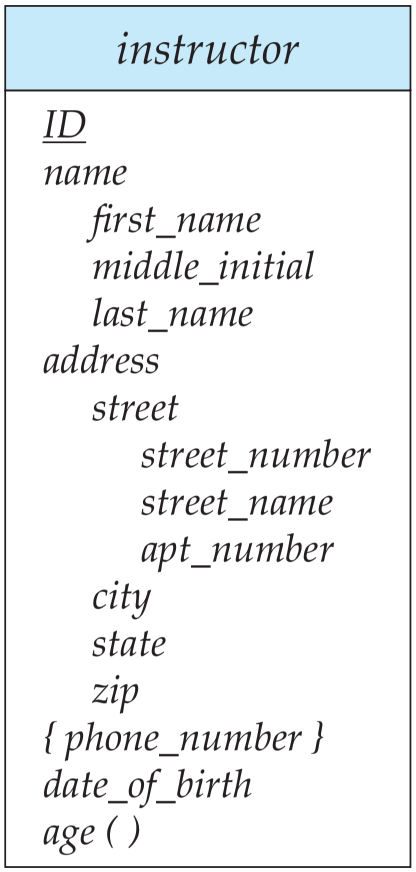
student(ID, name, tot_cred)

- A weak entity set becomes a table that includes a column for the primary key of the identifying strong entity set

section (course_id, sec_id, sem, year)

- Example

Representation of Entity Sets with Composite Attributes



- Composite attributes are flattened out by creating a separate attribute for each component attribute
  - Example: given entity set *instructor* with composite attribute *name* with component attributes *first_name* and *last_name* the schema corresponding to the entity set has two attributes *name_first_name* and *name_last_name*
    - Prefix omitted if there is no ambiguity (*name_first_name* could be *first_name)*
- Ignoring multivalued attributes, extended instructor table is
  - *instructor(ID, first_name, middle_initial, last_name, street_number, street_name, apt_number, city, state, zip_code, date_of_birth)*

Representation of Entity Sets with Multivalued Attributes

- A multivalued attribute *M* of an entity *E* is represented by a separate table *EM*
- Table *EM* has attributes corresponding to the primary key of *E* and an attribute corresponding to multivalued attribute *M*
- Example: Multivalued attribute *phone_number* of *instructor* is represented by a table:

  inst_phone= (*ID*, *phone_number*)

- Each value of the multivalued attribute maps to a separate tuple of the relation on table *EM*
  - For example, an *instructor* entity with primary key 22222 and phone numbers 456-7890 and 123-4567 maps to two tuples:
    (22222, 456-7890) and (22222, 123-4567)

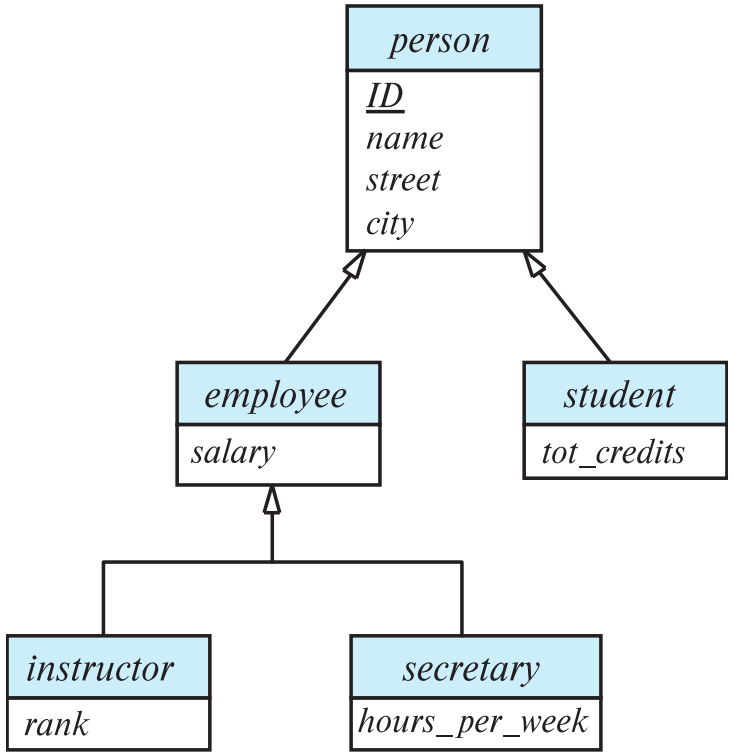
Choice of Primary key for Binary Relationship

- Many-to-Many relationships. The union of the primary keys is a minimal superkey and is chosen as the primary key.
- One-to-Many relationships . The primary key of the "Many" side is a minimal superkey and is used as the primary key.
- Many-to-one relationships. The primary key of the "Many" side is a minimal superkey and is used as the primary key.
- One-to-one relationships. The primary key of either one of the participating entity sets forms a minimal superkey, and either one can be chosen as the primary key.

Extended E-R Features

Specialization / Generalisation Example

- **Overlapping** – *employee* and *student*
  - An entity can belong to multiple specializations
- **Disjoint** – *instructor* and *secretary*
  - An entity can belong to only one specialization
- **Total and partial**
  - Total: each higher-level entity must belong to a lower-level entity set
  - Partial: some higher-level entities may not belong to any lower-level entity set.



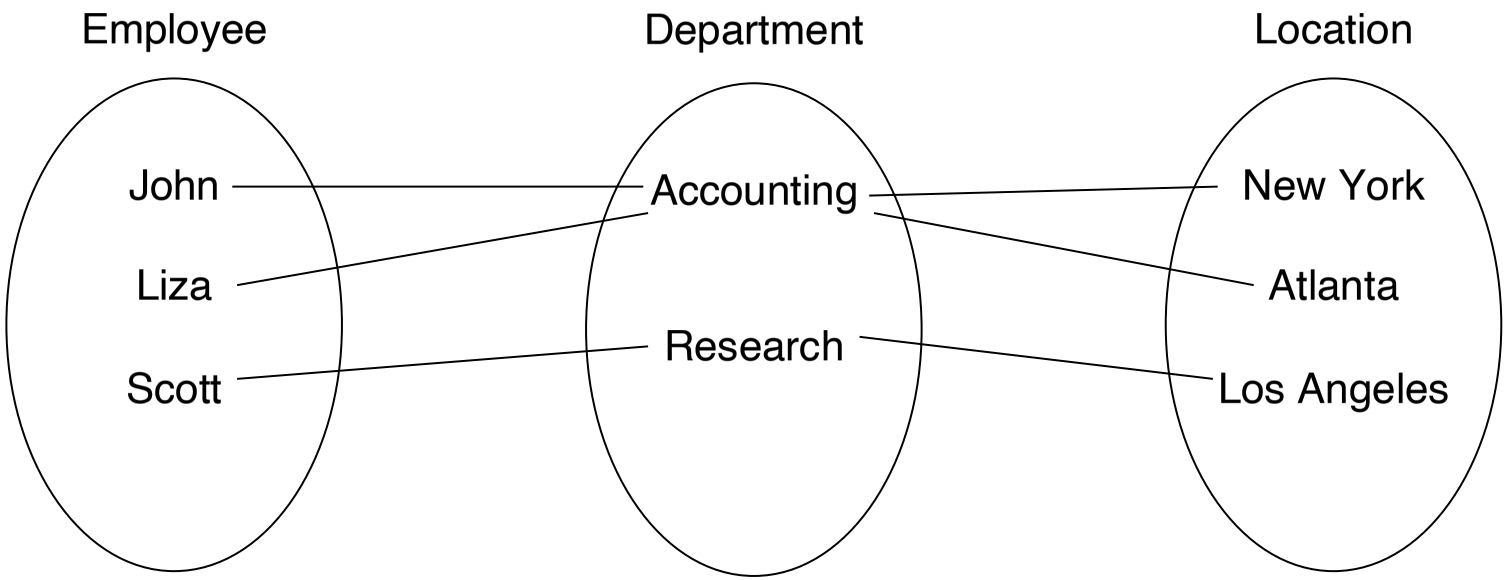
- Specialisation is top-down,
- Generalisation is bottom-up

Design Issues

Traps

- Traps occur when it is impossible to retrieve all of the necessary information from the diagram
  - Fan traps
  - Chasm traps

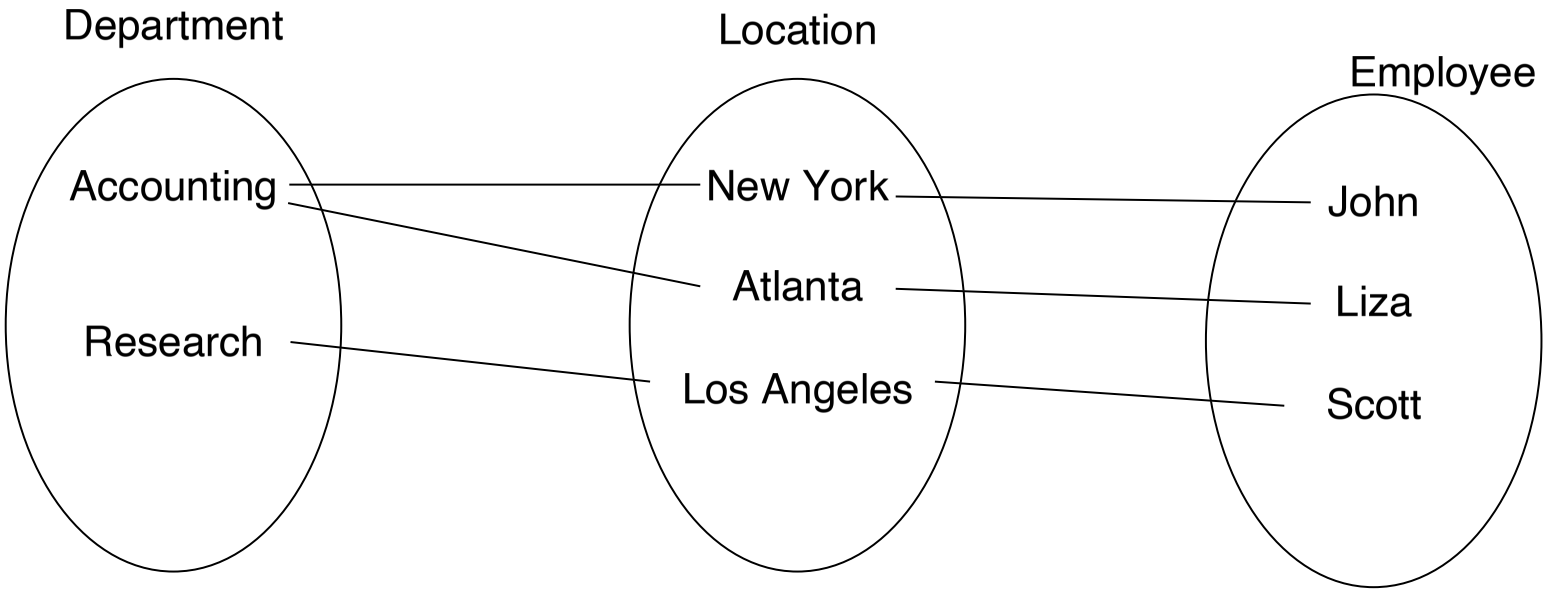
Fan Trap



Fan traps usually occur when there are 2 or more one-to-many relationships fan out from the same entity

Can't determine which city John or Liza work in

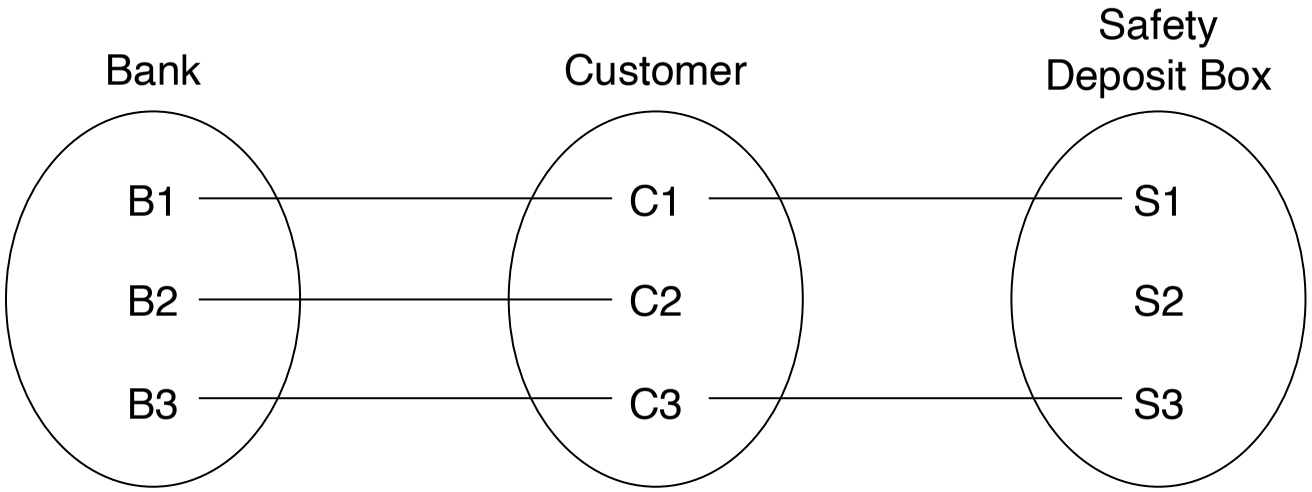
Fan Trap - Corrected



Now we can see which departments and locations each employee works in

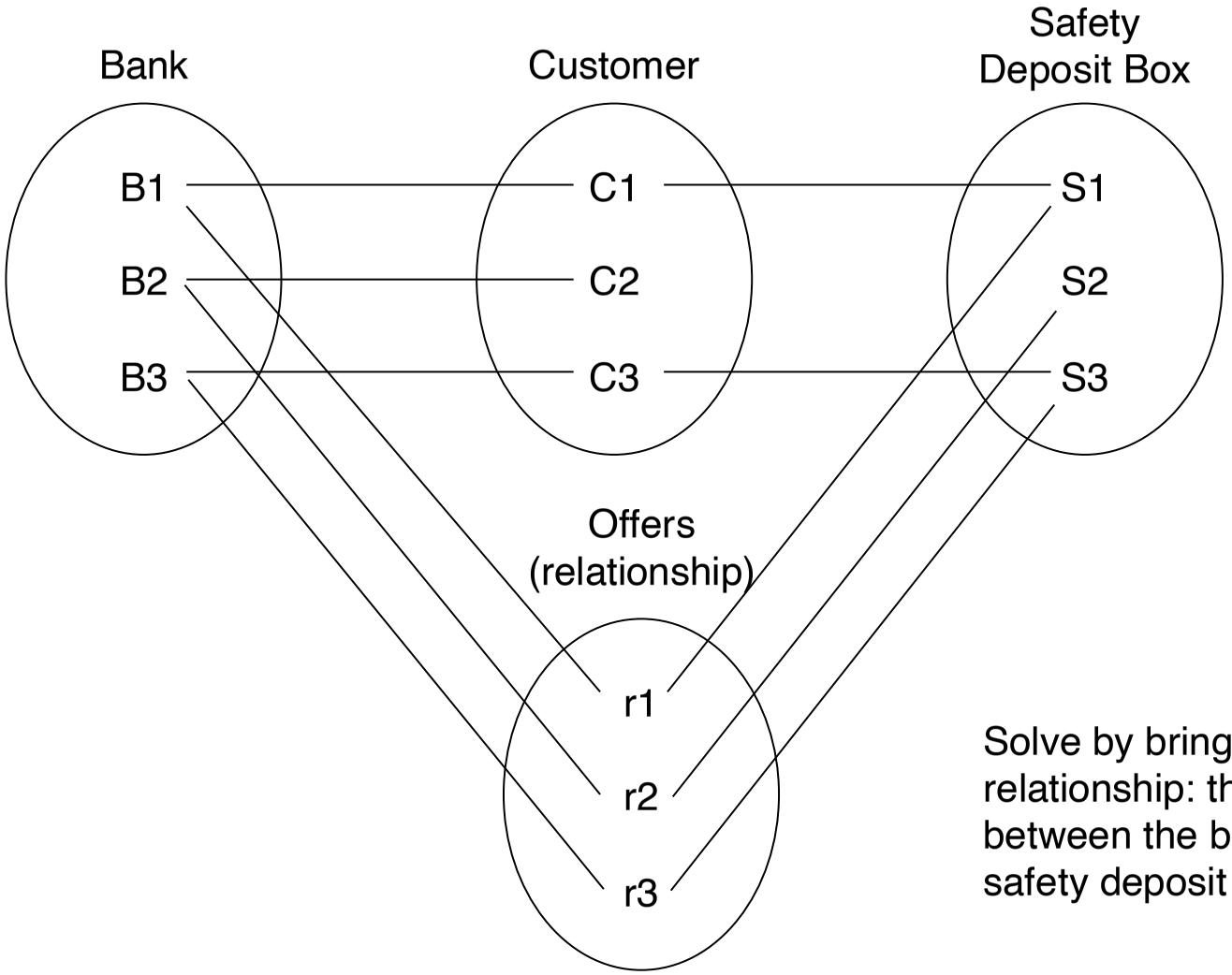
Chasm Trap

- Chasm traps occur when there can be an incomplete path from one entity to another
- This is often the result of a multiplicity that allows 0



Want to know which bank safety deposit box S2 belongs to

Chasm Trap - Correction



Solve by bringing in a 3rd relationship: this one between the bank and the safety deposit box.

Normalization

Example of Lossless Decomposition

- Decomposition of $R = (A, B, C)$

  $R_1 = (A, B)$ $\quad$ $R_2 = (B, C)$

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

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

$\Pi_A (r) \bowtie \Pi_B (r)$

A	B	C
α	1	A
β	2	B

Anomalies

Insert Anomalies

An example database

SalesStaff						
EmployeeID	**SalesPerson**	**SalesOffice**	**OfficeNumber**	**Customer1**	**Customer2**	**Customer3**
1003	Mary Smith	Chicago	312-555-1212	Ford	GM	
1004	John Hunt	New York	212-555-1212	Dell	HP	Apple
1005	Martin Hap	Chicago	312-555-1212	Boeing		

Insert Anomaly

EmployeeID	**SalesPerson**	**SalesOffice**	**OfficeNumber**	**Customer1**	**Customer2**	**Customer3**
1003	Mary Smith	Chicago	312-555-1212	Ford	GM	
1004	John Hunt	New York	212-555-1212	Dell	HP	Apple
1005	Martin Hap	Chicago	312-555-1212	Boeing		
???	???	Atlanta	312-555-1212			

There are facts we cannot record until we know information for the entire row. In our example we cannot open a new sales office until we also know at least one sales person.

Why? Because in order to create the record, we need provide a primary key. In our case this is the EmployeeID.

Update Anomalies

Example database

SalesStaff						
EmployeeID	**SalesPerson**	**SalesOffice**	**OfficeNumber**	**Customer1**	**Customer2**	**Customer3**
1003	Mary Smith	Chicago	312-555-1212	Ford	GM	
1004	John Hunt	New York	212-555-1212	Dell	HP	Apple
1005	Martin Hap	Chicago	312-555-1212	Boeing		

EmployeeID	**SalesPerson**	**SalesOffice**	**OfficeNumber**	**Customer1**	**Customer2**	**Customer3**
1003	Mary Smith	Chicago	312-555-1212	Ford	GM	
1004	John Hunt	New York	212-555-1212	Dell	HP	Apple
1005	Martin Hap	Chicago	312-555-1212	Boeing		

Here, we have the same information in several rows.

If the office number changes, then there are multiple updates that need to be made. If we don't update all rows, then inconsistencies appear.

Deletion Anomalies

Example database

SalesStaff						
EmployeeID	**SalesPerson**	**SalesOffice**	**OfficeNumber**	**Customer1**	**Customer2**	**Customer3**
1003	Mary Smith	Chicago	312-555-1212	Ford	GM	
1004	John Hunt	New York	212-555-1212	Dell	HP	Apple
1005	Martin Hap	Chicago	312-555-1212	Boeing		

EmployeeID	SalesPerson	SalesOffice	OfficeNumber	Customer1	Customer2	Customer3
1003	Mary Smith	Chicago	312-555-1212	Ford	GM	
~~1004~~	~~John Hunt~~	~~New York~~	~~212-555-1212~~	~~Dell~~	~~HP~~	~~Apple~~
1005	Martin Hap	Chicago	312-555-1212	Boeing		

Removal of a row causes removal of more than one set of facts.

For instance, if John Hunt retires, then deleting that row cause us to lose information about the New York office.

Complicated Searches

Example database

SalesStaff						
EmployeeID	**SalesPerson**	**SalesOffice**	**OfficeNumber**	**Customer1**	**Customer2**	**Customer3**
1003	Mary Smith	Chicago	312-555-1212	Ford	GM	
1004	John Hunt	New York	212-555-1212	Dell	HP	Apple
1005	Martin Hap	Chicago	312-555-1212	Boeing		

If you want to search for a specific customer such as Ford, you will have to write a query like:

```
SELECT SalesOffice
  FROM SalesStaff
WHERE Customer1 = 'Ford'
      OR Customer2 = 'Ford'
      OR Customer3 = 'Ford'
```

What is Normalisation?

- Normalization is a method of organizing the data in the database which helps you to
  - avoid data redundancy
  - insertion, update & deletion anomalies
  - Makes doing searches easier

- It is a process of analyzing the relation schemas based on their different functional dependencies and primary key.

What are Functional Dependencies?

- A functional dependency is a constraint that specifies the relationship between two sets of attributes, where one set can accurately determine the value of the other set
  - Given one value for X, do I find one and only one value for Y?

- Denoted as X -> Y
  - X is a set of attributes that is capable of determining the value of Y
  - X is called the Determinant
  - Y is called the Dependent

- Used to mathematically express relationships between database entities
  - Armstrong's Rules / Axioms – used to infer all of the functional dependencies in a relational database
  - https://en.wikipedia.org/wiki/Armstrong%27s_axioms

More Examples of Valid Dependencies

roll_no	name	dept_name	dept_building
42	abc	CO	A4
43	pqr	IT	A3
44	xyz	CO	A4
45	xyz	IT	A3
46	mno	EC	B2
47	jkl	ME	B2

Some more valid functional dependencies:

- dept_name → dept_building
  - dept_name can identify the dept_building accurately, since departments with different dept_name will also have a different dept_building
- More valid functional dependencies:
  - roll_no → name
  - {roll_no, name} ⇢ {dept_name, dept_building}, etc.

Just because there is a valid functional dependency X -> Y does not mean that the reverse Y-> X is also a valid functional dependency. In fact, often the reverse is not a valid functional dependency.

Example continued

roll_no	name	dept_name	dept_building
42	abc	CO	A4
43	pqr	IT	A3
44	xyz	CO	A4
45	xyz	IT	A3
46	mno	EC	B2
47	jkl	ME	B2

Some **invalid** functional dependencies:

- name → dept_name
  - Students with the same name can have different dept_name, hence this is not a valid functional dependency.
- dept_building → dept_name
  - There can be multiple departments in the same building, For example, in the above table departments ME and EC are in the same building B2, hence dept_building → dept_name is an invalid functional dependency.

Types of Dependencies

- There are mainly four types of Functional Dependency
  - Trivial Functional Dependency
  - Non-Trivial Functional Dependency
  - Multivalued Dependency
  - Transitive Dependency

Normalisation

- There are currently 10 different levels of normalization (not including unnormalized)

- From least to most normalized, the levels are:

UNF: Unnormalised form
1NF: First normal form
2NF: Second normal form
3NF: Third normal form
EKNF: Elementary key normal form
BCNF: Boyce-Codd normal form
4NF: Fourth normal form
ETNF: Essential tuple normal form
5NF: Fifth normal form
DKNF: Domain-key normal form
6NF: Sixth normal form

- Typically, we only use 1 – 3 normal forms, and maybe BCNF
  - Though we probably should also regularly be using 4NF

First Normal Form

- Attributes must be single-valued (atomic)
  - No multi-valued attributes
  - No composite attributes

- Example:

STUD_NO	STUD_NAME	STUD_PHONE	STUD_STATE	STUD_COUNTRY
1	RAM	9716271721, 9871717178	HARYANA	INDIA
2	RAM	9898297281	PUNJAB	INDIA
3	SURESH		PUNJAB	INDIA

Table 1

Conversion to first normal form

STUD_NO	STUD_NAME	STUD_PHONE	STUD_STATE	STUD_COUNTRY
1	RAM	9716271721	HARYANA	INDIA
1	RAM	9871717178	HARYANA	INDIA
2	RAM	9898297281	PUNJAB	INDIA
3	SURESH		PUNJAB	INDIA

Table 2

Second Normal Form

- Must be in First Normal Form **AND**
- There can be NO partial dependencies:
  - Each non-key attribute must be fully functionally dependent on the entire primary key
  - If an attribute depends on only part of a multi-valued key, move it to a separate table

2NF Example

STUD_NO	COURSE_NO	COURSE_FEE
1	C1	1000
2	C2	1500
1	C4	2000
4	C3	1000
4	C1	1000
2	C5	2000

Note: multiple courses can have the same fee

Key must be (STUD_NO, COURSE_NO)

- The key must be STUD_NO and COURSE_NO
- But, multiple courses can have the same fee
  - So fee is dependent on only part of the key

2NF – The Fix

Table 1

STUD_NO	COURSE_NO
1	C1
2	C2
1	C4
4	C3
4	C1

Table 2

COURSE_NO	COURSE_FEE
C1	1000
C2	1500
C3	1000
C4	2000
C5	2000

Need to split the table into two tables such as :
Table 1: STUD_NO, COURSE_NO
Table 2: COURSE_NO, COURSE_FEE

NOTE: 2NF tries to reduce the redundant data getting stored in memory. For instance, if there are 100 students taking C1 course, we don't need to store its Fee as 1000 for all the 100 records, instead, once we can store it in the second table as the course fee for C1 is 1000.

Third Normal Form

- Must be in Second Normal Form **AND**

- No non-prime attribute is transitively dependent on the primary key
  - Every **non-key** attribute must provide a fact about the key, the whole key, and nothing but the key
  - The table is in 3NF if and only if for every non-trivial functional dependency X -> Y, X is a super key or Y is a prime attribute

Third Normal Form - Example

STUD_NO	STUD_NAME	STUD_STATE	STUD_COUNTRY	STUD_AGE
1	RAM	HARYANA	INDIA	20
2	RAM	PUNJAB	INDIA	19
3	SURESH	PUNJAB	INDIA	21

Transitive dependency – If A->B and B->C are two FDs then A->C is called transitive dependency.

In this example, the Functional Dependencies are:

STUD_NO -> STUD_NAME
STUD_NO -> STUD_STATE
STUD_STATE -> STUD_COUNTRY
STUD_NO -> STUD_AGE

Primary Key: {STUD_NO}

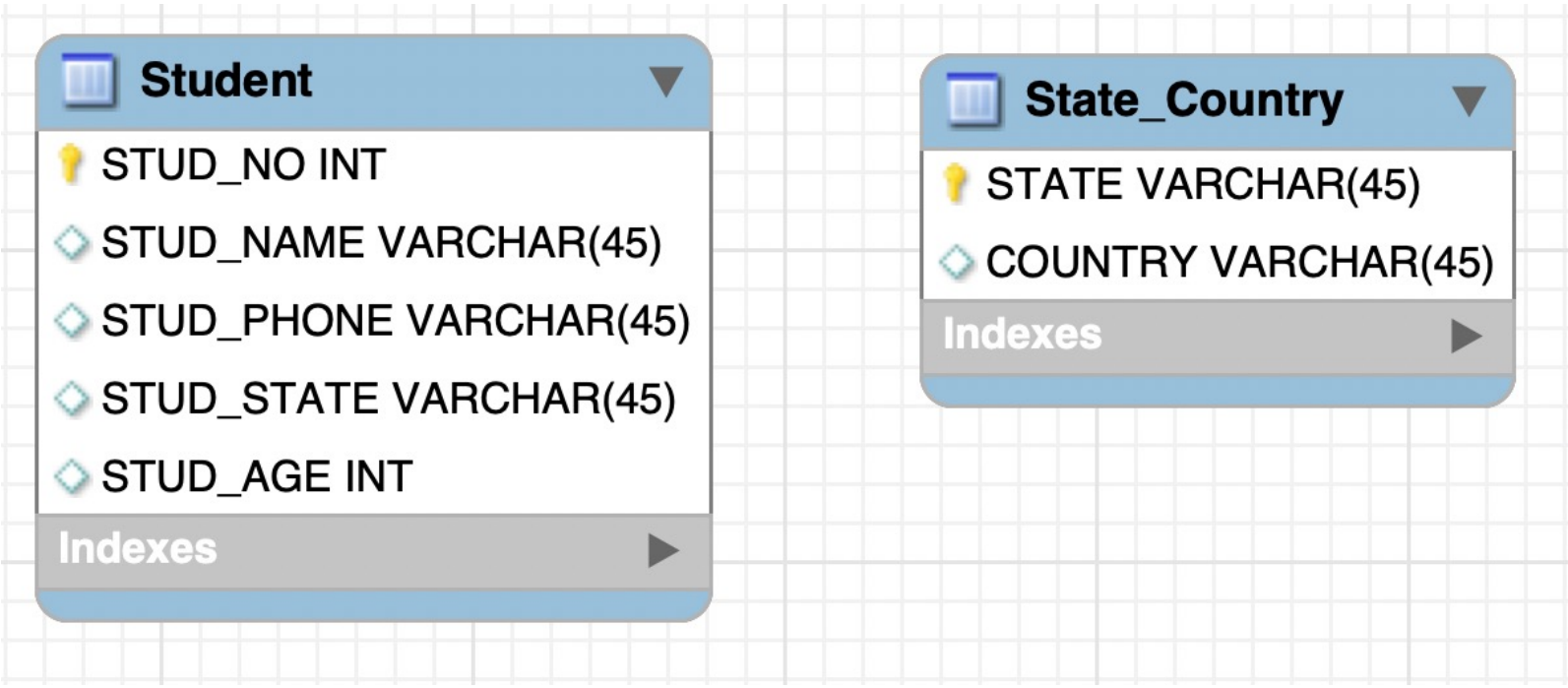
STUD_NO -> STUD_STATE and STUD_STATE -> STUD_COUNTRY are true.

This means that STUD_COUNTRY is transitively dependent on STUD_NO.
This violates the third normal form

Another similar example found in tables is:
City, state and zipcode

3NF – The Fix

Decompose into 2 tables:



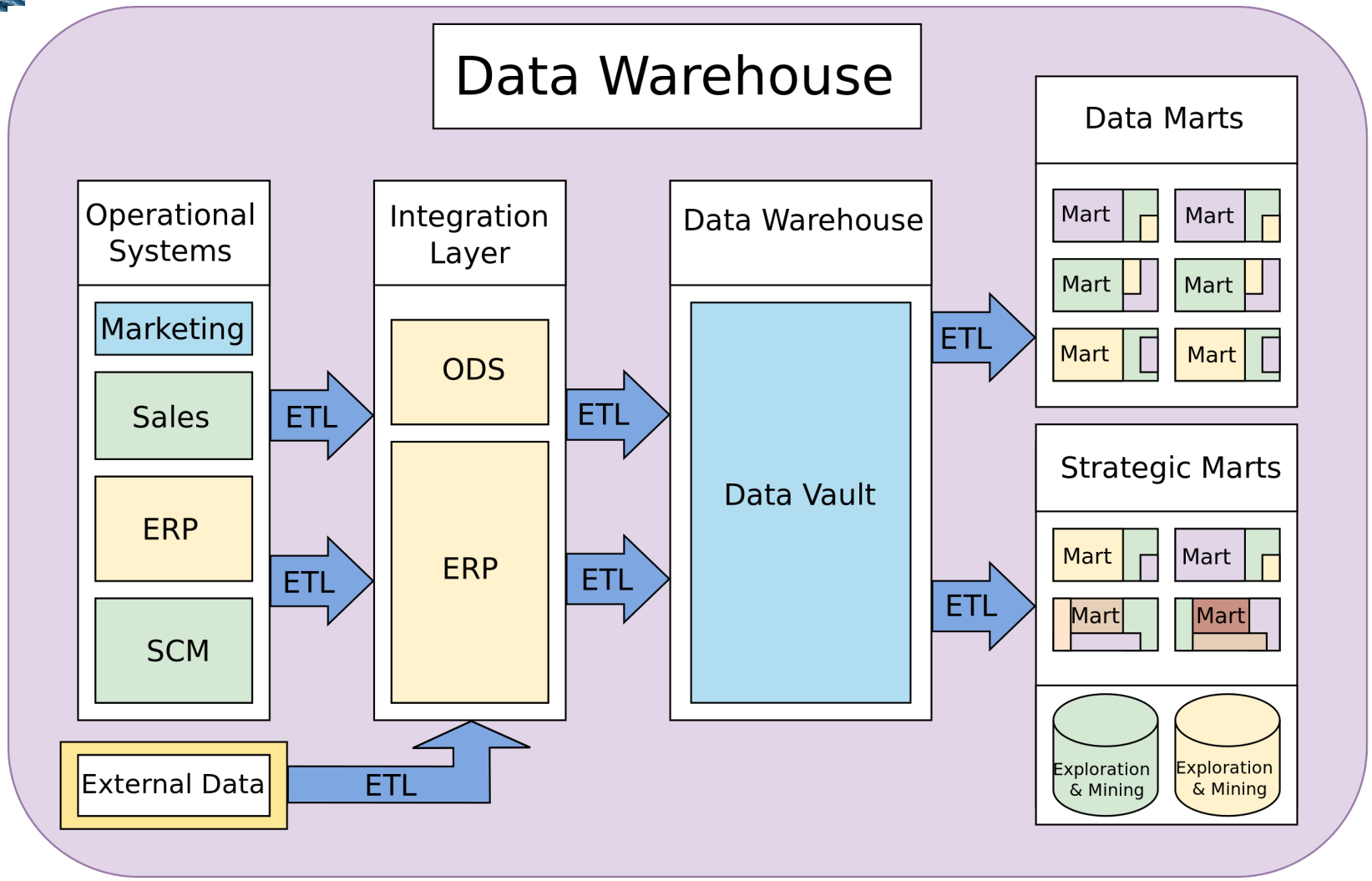
Denormalization for Performance

- May want to use non-normalized schema for performance
- For example, displaying *prereqs* along with *course_id,* and *title* requires join of *course* with *prereq*
- Alternative 1: Use denormalized relation containing attributes of *course* as well as *prereq* with all above attributes
  - faster lookup
  - extra space and extra execution time for updates
  - extra coding work for programmer and possibility of error in extra code
- Alternative 2: use a materialized view defined a *course* ⋈ *prereq*
  - Benefits and drawbacks same as above, except no extra coding work for programmer and avoids possible errors

Data Warehousing

- Data sources often store only current data, not historical data
- Corporate decision making requires a unified view of all organizational data, including historical data
- A **data warehouse** is a repository (archive) of information gathered from multiple sources, stored under a unified schema, at a single site
  - Greatly simplifies querying, permits study of historical trends
  - Shifts decision support query load away from transaction processing systems
  - Compare to a **data mart** which is a data warehouse that supports a specific business unit

Data Warehouse – Another View

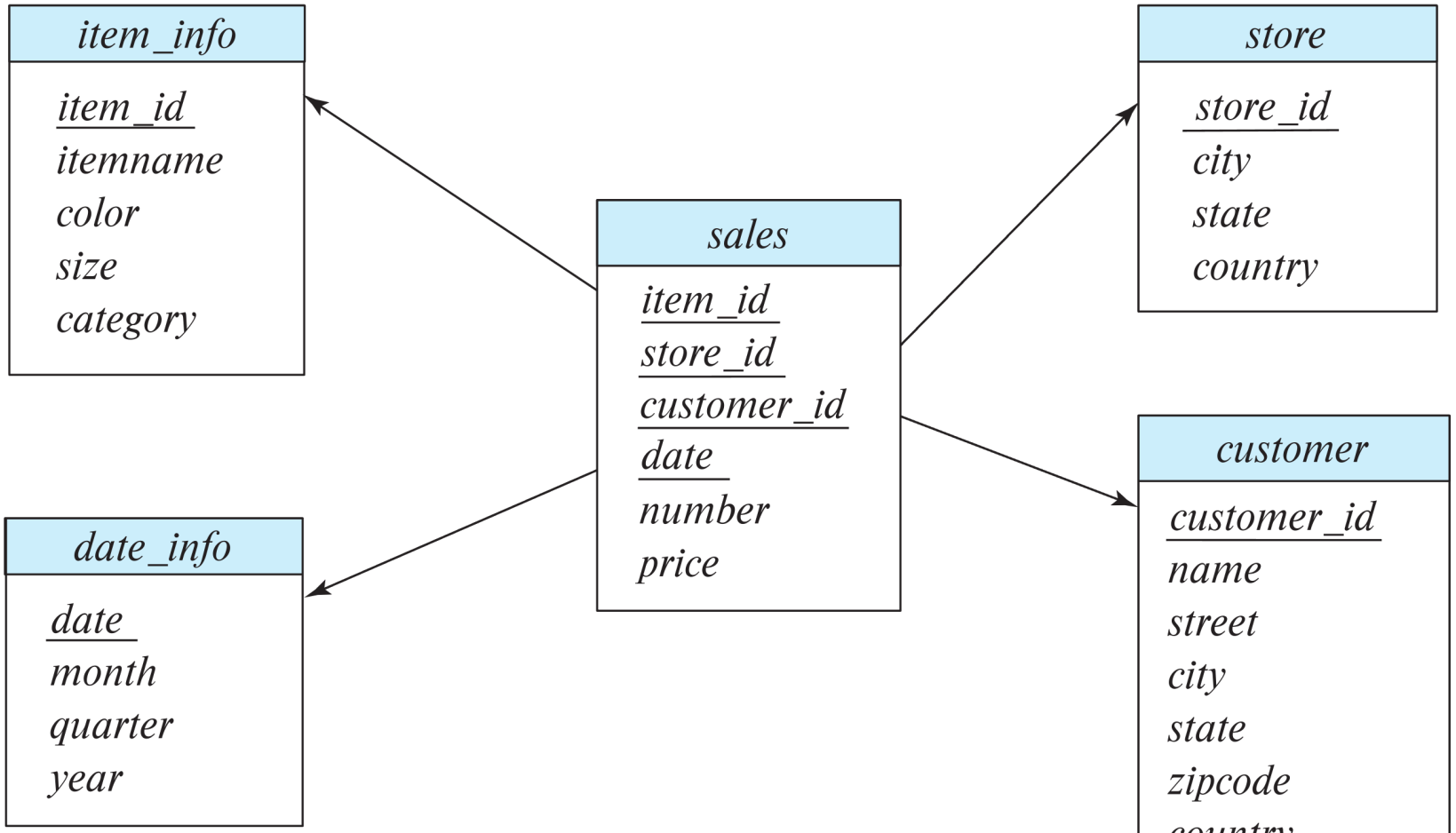


Multidimensional Data and Warehouse Schemas

- Data in warehouses can usually be divided into
  - **Fact tables**, which are large
    - E.g, *sales*(*item_id, store_id, customer_id, date, number, price*)
  - **Dimension tables**, which are relatively small
    - Store extra information about stores, items, etc.

- Attributes of **fact** tables can be usually viewed as
  - **Measure attributes**
    - measure some value, and can be aggregated upon
    - e.g., the attributes *number* or *price* of the *sales* relation
  - **Dimension attributes**
    - dimensions on which measure attributes are viewed
    - e.g., attributes *item_id, color,* and *size* of the *sales* relation
    - Usually small ids that are foreign keys to dimension tables

Data Warehouse Schema (Star Query)



The fact table is essentially a type of relationship table.

Chapter 12: Physical Storage Systems

Database System Concepts, 7th Ed.

Performance Measures (Cont.)

- **Disk block** is a logical unit for storage allocation and retrieval
  - 4 to 16 kilobytes typically
    - Smaller blocks: more transfers from disk
    - Larger blocks: more space wasted due to partially filled blocks
- **Sequential access pattern**
  - Successive requests are for successive disk blocks
  - Disk seek required only for first block
- **Random access pattern**
  - Successive requests are for blocks that can be anywhere on disk
  - Each access requires a seek
  - Transfer rates are low since a lot of time is wasted in seeks
- **I/O operations per second (IOPS)**
  - Number of random block reads that a disk can support per second
  - 50 to 200 IOPS on current generation magnetic disks

Improvement of Reliability via Redundancy

- **Redundancy** – store extra information that can be used to rebuild information lost in a disk failure
- E.g., **Mirroring** (or **shadowing**)
  - Duplicate every disk. Logical disk consists of two physical disks.
  - Every write is carried out on both disks
    - Reads can take place from either disk
  - If one disk in a pair fails, data still available in the other
    - Data loss would occur only if a disk fails, and its mirror disk also fails before the system is repaired
      - Probability of combined event is very small
        - Except for dependent failure modes such as fire or building collapse or electrical power surges
- **Mean time to data loss** depends on mean time to failure, and **mean time to repair**
  - E.g., MTTF of 100,000 hours, mean time to repair of 10 hours gives mean time to data loss of $500*10^6$ hours (or 57,000 years) for a mirrored pair of disks (ignoring dependent failure modes)

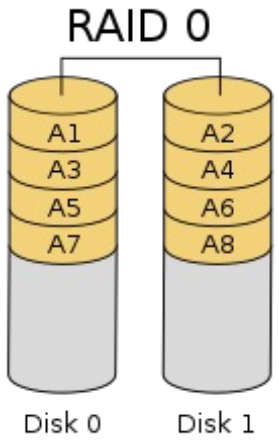
RAID

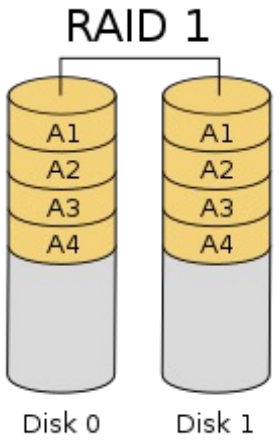
- **RAID: Redundant Arrays of Independent Disks**
  - disk organization techniques that manage a large numbers of disks, providing a view of a single disk of
    - **high capacity** and **high speed** by using multiple disks in parallel,
    - **high reliability** by storing data redundantly, so that data can be recovered even if a disk fails

- The chance that some disk out of a set of *N* disks will fail is much higher than the chance that a specific single disk will fail.
  - E.g., a system with 100 disks, each with MTTF of 100,000 hours (approx. 11 years), will have a system MTTF of 1000 hours (approx. 41 days)
  - Techniques for using redundancy to avoid data loss are critical with large numbers of disks

RAID Levels

- Schemes to provide redundancy at lower cost by using disk striping combined with parity bits
  - Different RAID organizations, or RAID levels, have differing cost, performance and reliability characteristics
- **RAID Level 0**: Block striping; non-redundant.
  - Used in high-performance applications where data loss is not critical.
- **RAID Level 1**: Mirrored disks with block striping
  - Offers best write performance.
  - Popular for applications such as storing log files in a database system.



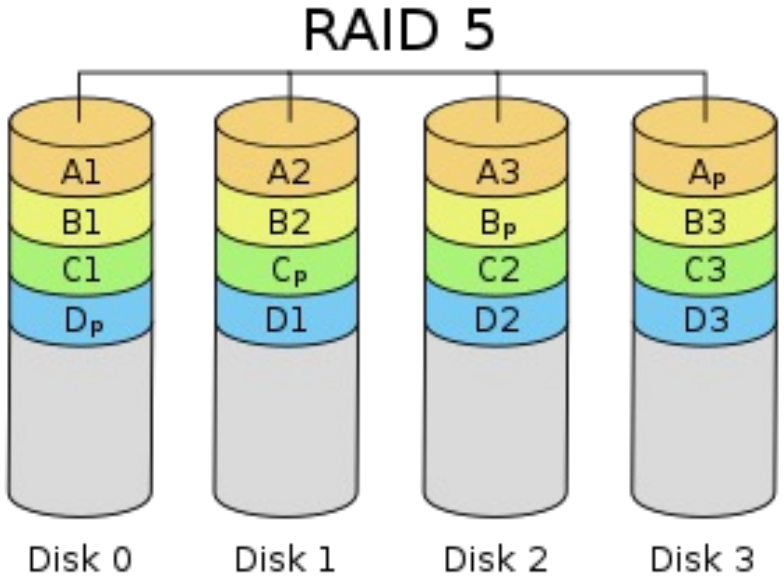


RAID Levels (Cont.)

- **RAID Level 5:** Block-Interleaved Distributed Parity
  - Parity blocks are distributed amongst the disks
  - E.g., with 4 disks, parity block for *n*th set of blocks is stored on disk (*n mod* 4) + 1, with the data blocks stored on the other 3 disks.
  - If a disk fails, can compute what is on it using the parity blocks
    - Slower performance, but it still runs
  - Time penalty for writes, but not reads

Parity bit
Simple error checking method.
Done using XOR.
Example:
Disk 1: 01101101
Disk 2: 11010100

Apply XOR:
10111001



Chapter 13: Data Storage Structures

Database System Concepts, 7th Ed.

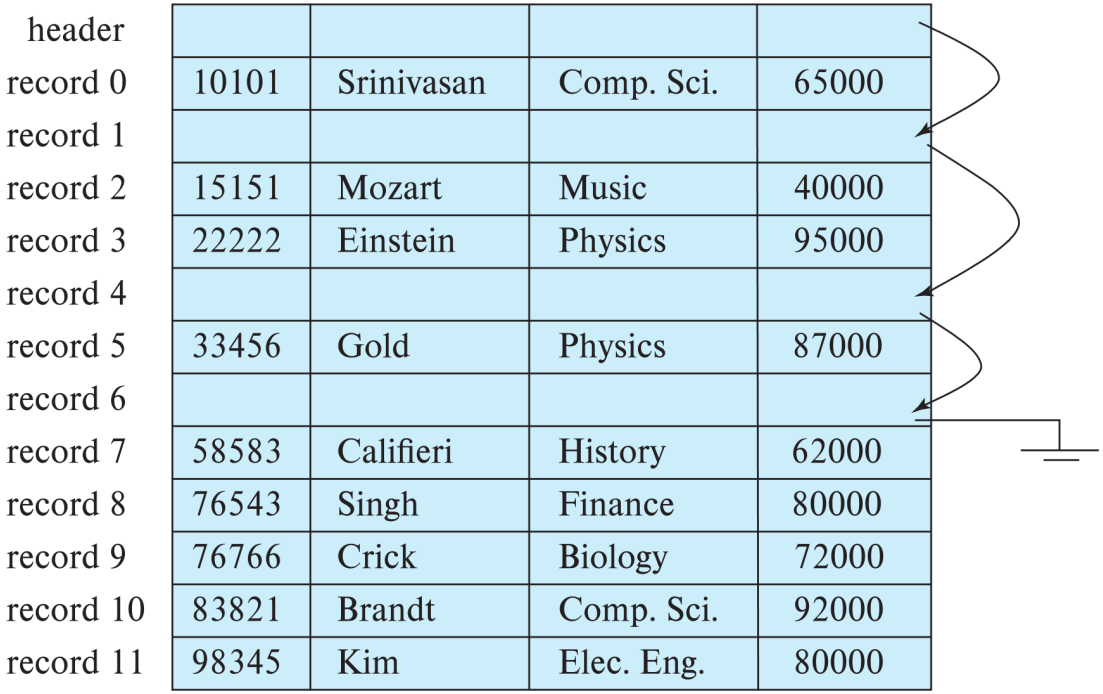
Fixed-Length Records

- Simple approach:
  - Store record *i* starting from byte $n * (i - 1)$, where n is the size of each record.
  - Record access is simple but records may cross blocks
    - Modification: do not allow records to cross block boundaries

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

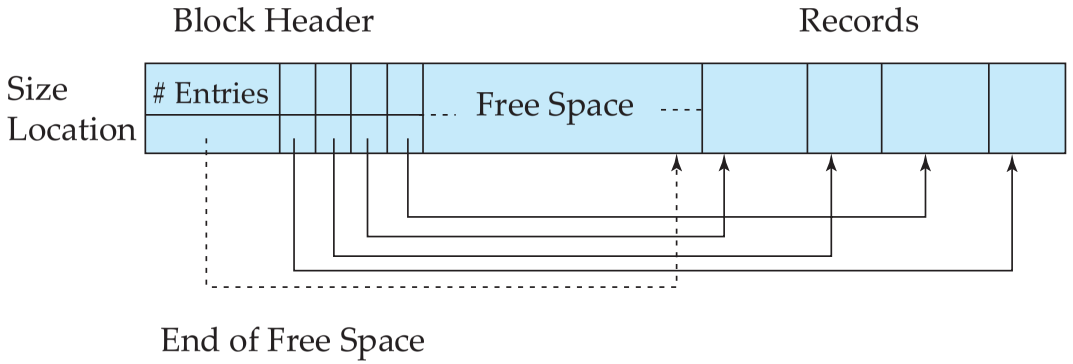
Fixed-Length Records

- Deletion of record *i:* alternatives*:*
  - move records $i + 1, \ldots, n$ to $i, \ldots, n - 1$
  - move record n to i
  - **do not move records, but link all free records on a *free list***



header				
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1				
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4				
record 5	33456	Gold	Physics	87000
record 6				
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

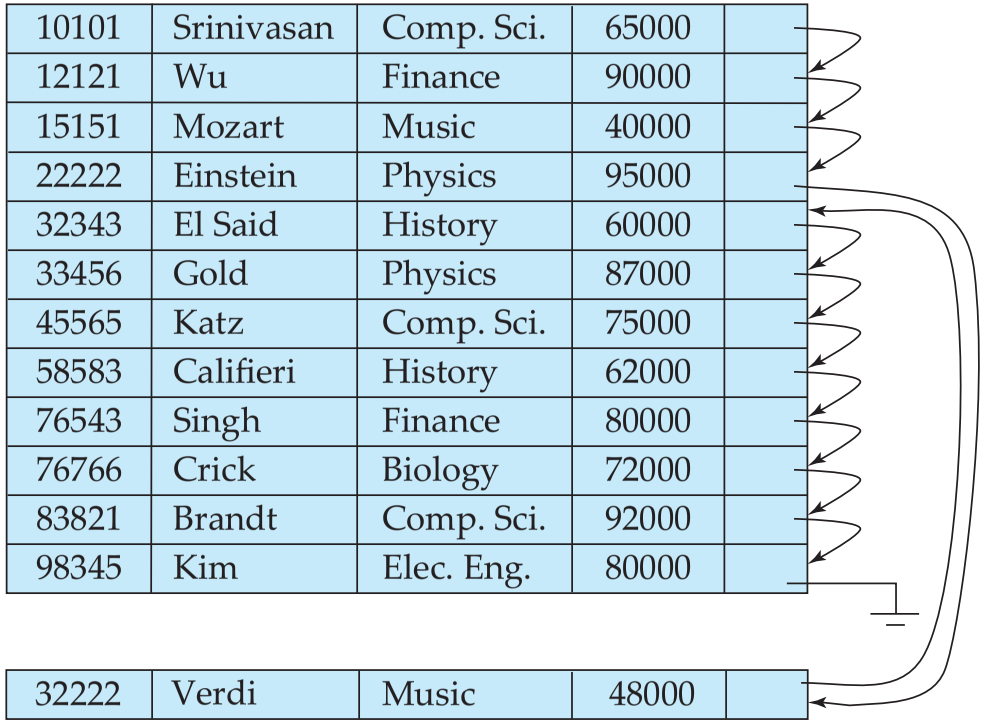
Variable-Length Records: Slotted Page Structure



- **Slotted pages** work by splitting up the page into a few independent memory regions that grow into each other.
- The header contains:
  - number of record entries
  - end of free space in the block
  - location and size of each record
- The variable-length records reside in a contiguous manner within the block.

Sequential File Organization (Cont.)

- Deletion – use pointer chains
- Insertion –locate the position where the record is to be inserted
  - if there is free space insert there
  - if no free space, insert the record in an overflow block
  - In either case, pointer chain must be updated
- Need to reorganize the file from time to time to restore sequential order



10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

32222	Verdi	Music	48000	

Chapter 14: Indexing

Database System Concepts, 7th Ed.

Basic Concepts

- Indexing mechanisms used to speed up access to desired data.
  - E.g., author catalog in library, or a book index
- **Search Key** - attribute to set of attributes used to look up records in a file.

- An **index file** consists of records (called **index entries**) of the form

search-key	pointer

- Index files are typically much smaller than the original file

- Two basic kinds of indices:
  - **Ordered indices:** search keys are stored in sorted order
  - **Hash indices:** search keys are distributed uniformly across "buckets" using a "hash function".

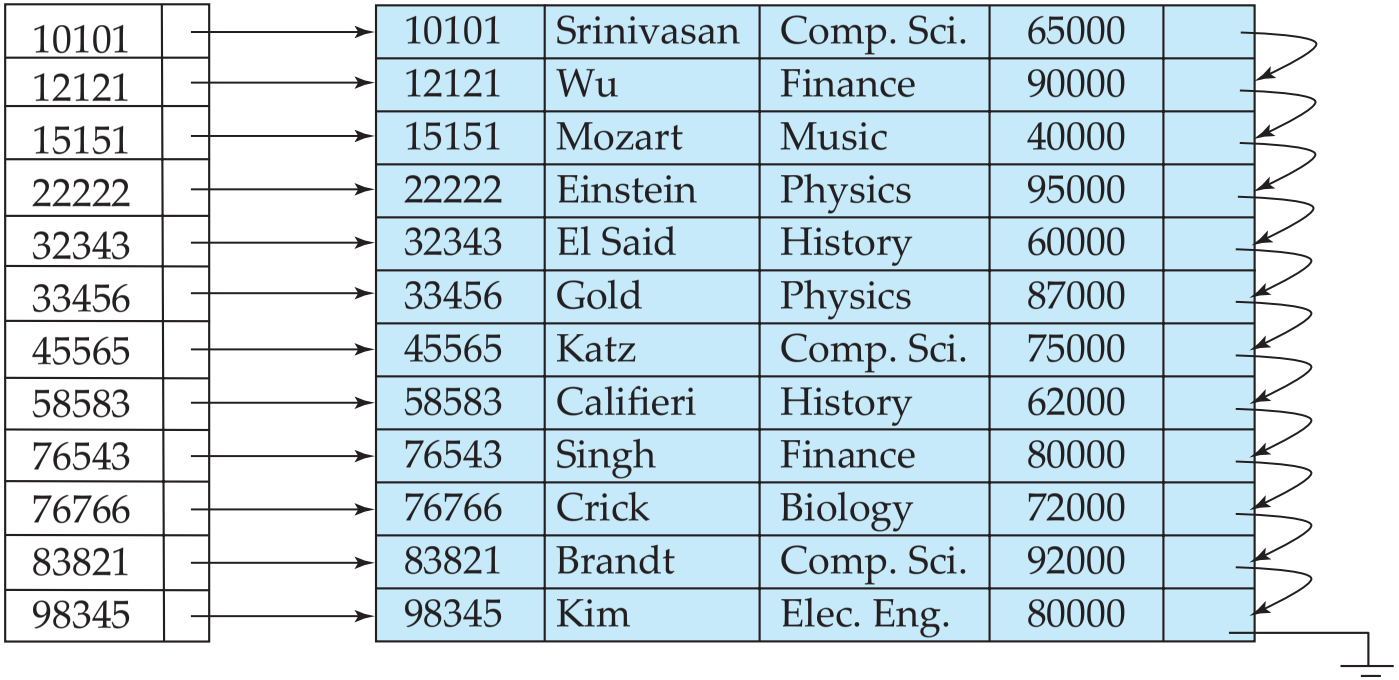
Ordered Indices

- In an **ordered index,** index entries are stored sorted on the search key value.
  - Similar in concept to a book index
- **Clustering index:** determines the sequential sort order of the file
  - Also called **primary index**
  - The search key of a primary index is usually but not necessarily the primary key.
- **Secondary index**: an index whose search key specifies an order different from the sorted order of the file. Also called a **nonclustering index.**
- **Index-sequential file:** sequential file ordered on a search key, with a clustering index on the search key.

Dense Index Files

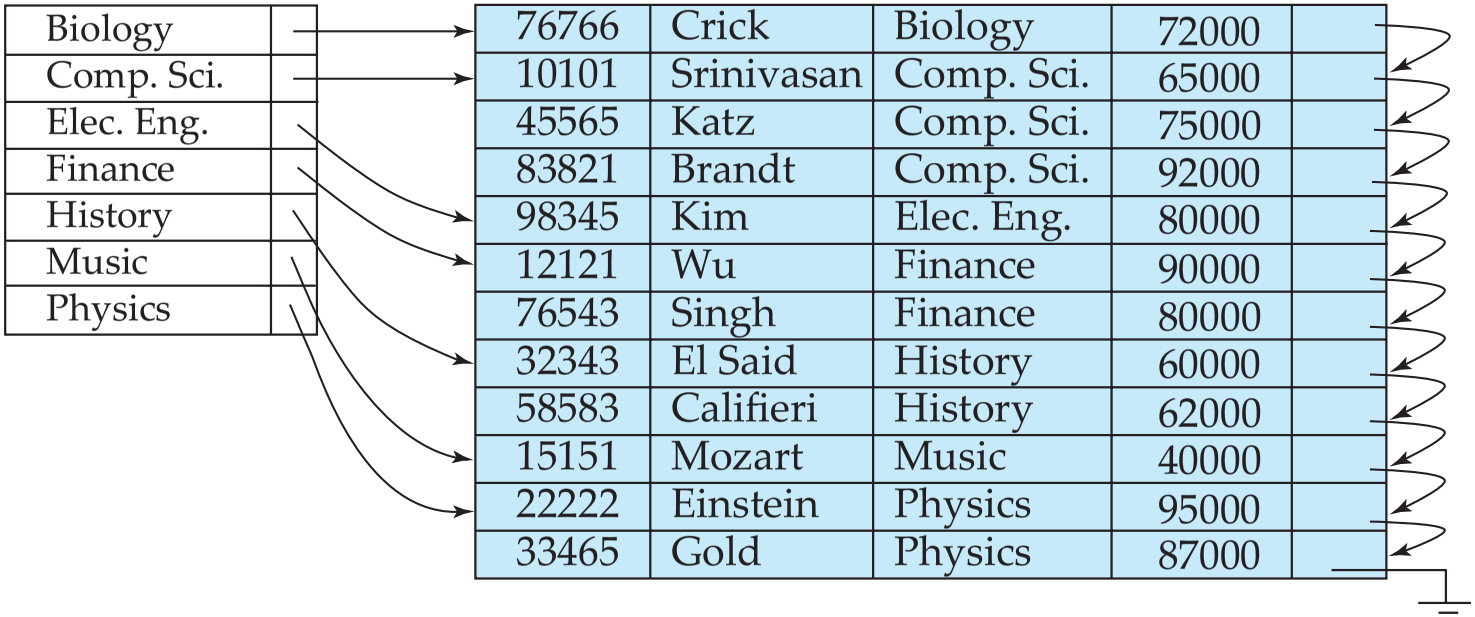
Dense index — Index record appears for every search-key value in the file.

- E.g. index on *ID* attribute of *instructor* relation
- Search the index for a value, e.g. 22222, then use the pointer to find the disk block that the record resides in



Dense Index Files (Cont.)

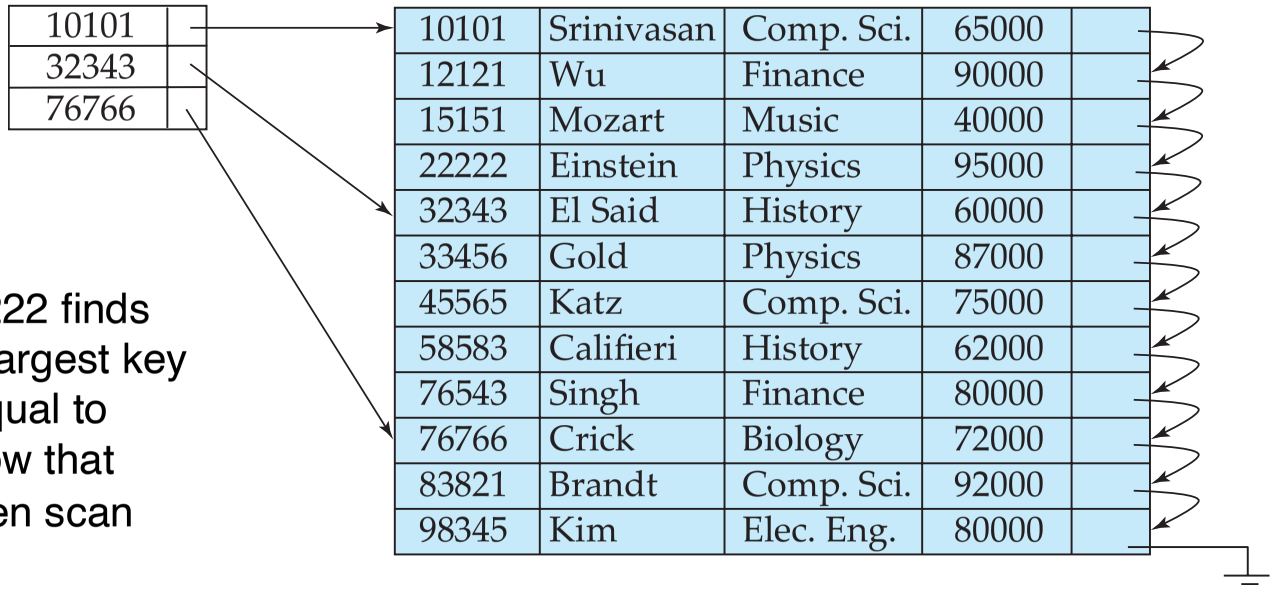
- Dense index on *dept_name*, with *instructor* file sorted on *dept_name*
- Because the file is sorted, only have pointer to the first entry
  - Rest of the records with that same search key are accessed sequentiantially after the first one since this is a clustering index (if not unique)



Sparse Index Files

- **Sparse Index**: contains index records for only some search-key values.
  - Only applicable when records are sequentially ordered on search-key

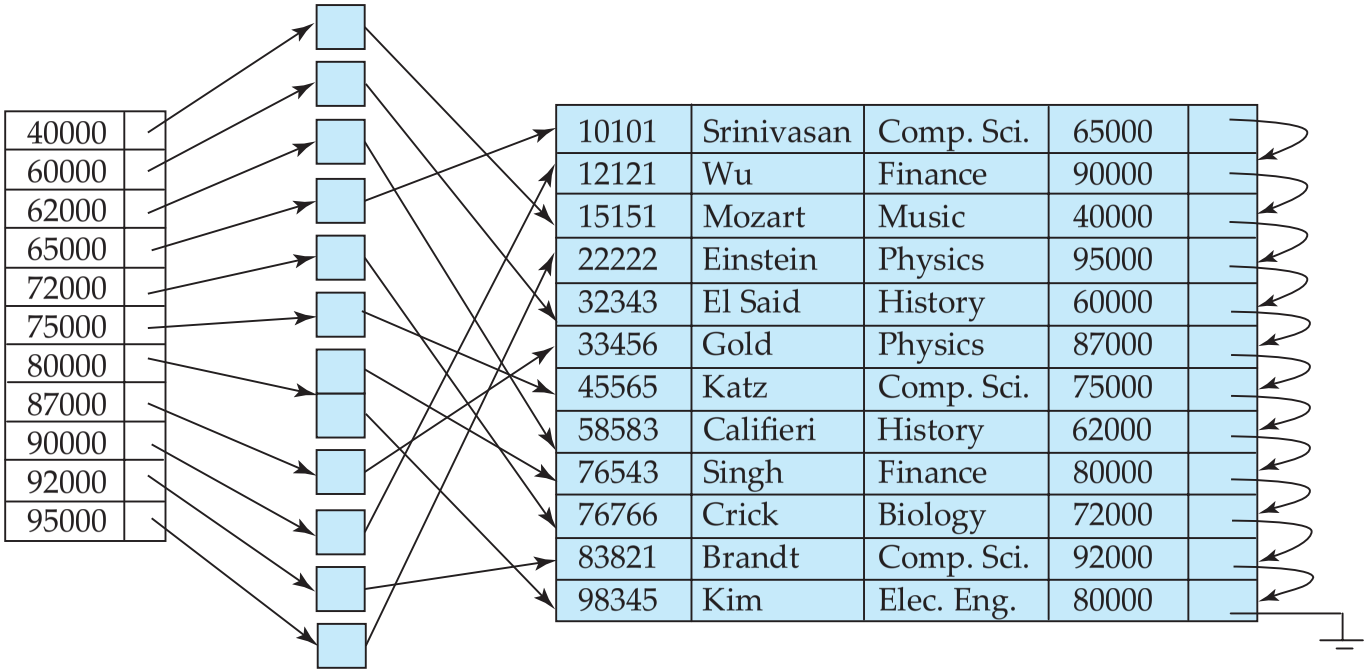
- To locate a record with search-key value *K* we:
  - Find index record with largest search-key value less than or equal to *K*
  - Search file sequentially starting at the record to which the index record points



Search on 22222 finds 10101 as the largest key less than or equal to 22222, so follow that pointer and then scan

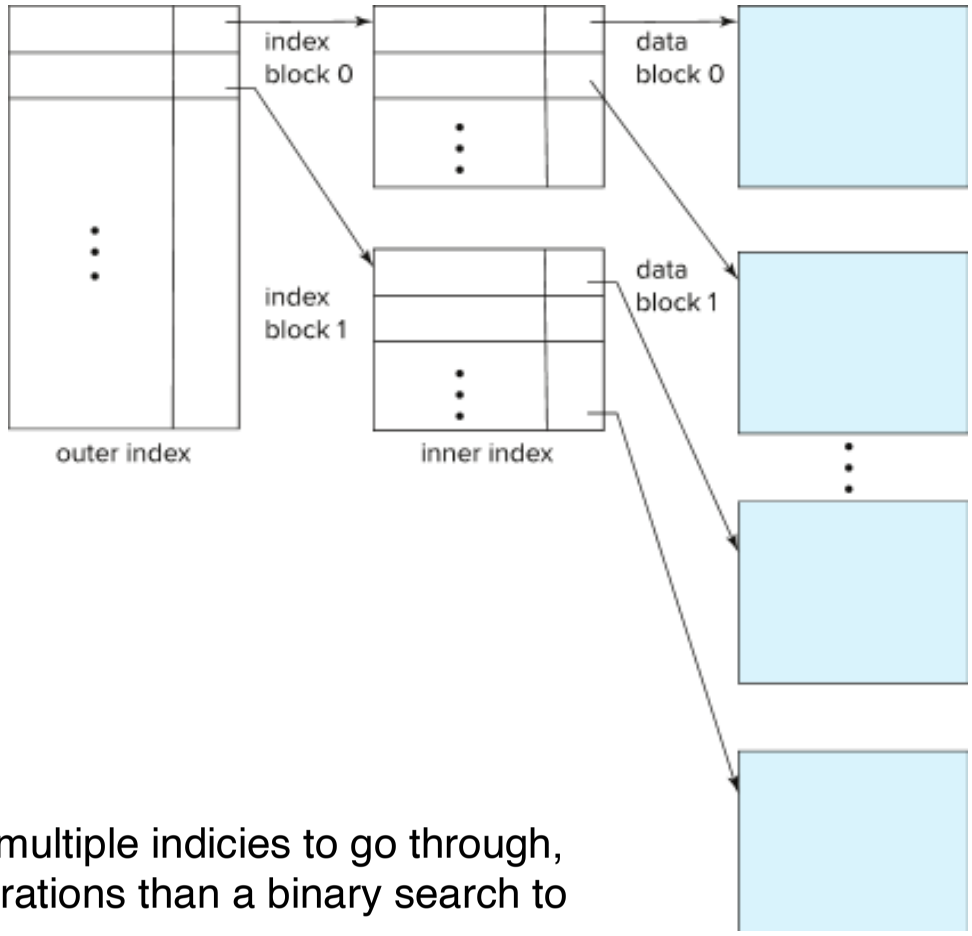
Secondary Indices Example

- Secondary index on salary field of instructor
  - Can't rely on the file being sorted in a way to accommodate a secondary index



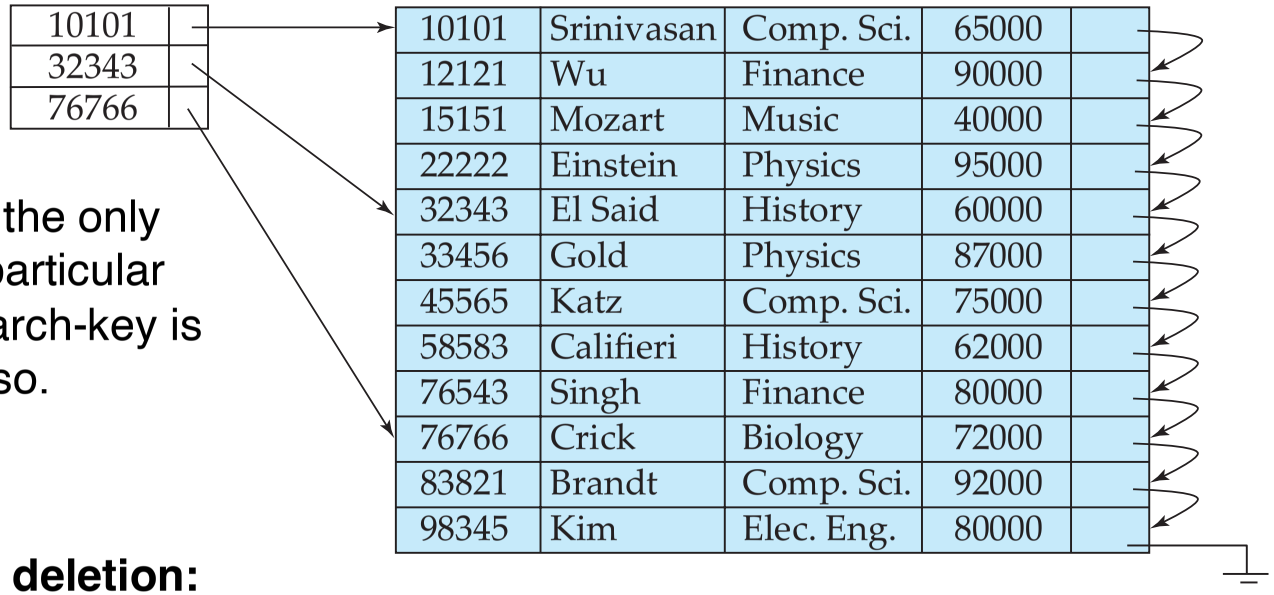
- Index record points to a bucket that contains pointers to all the actual records with that particular search-key value.
- Secondary indices have to be dense

Multilevel Index (Cont.)



Even though there are multiple indicies to go through, there are fewer I/O operations than a binary search to find a record

Index Update: Deletion



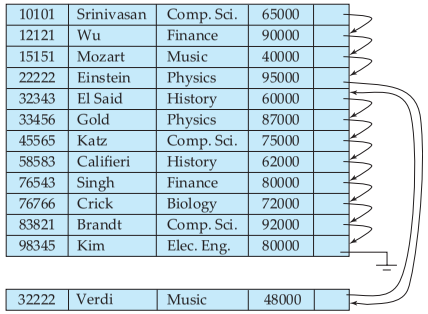
If the deleted record was the only record in the file with its particular search-key value, the search-key is deleted from the index also.

Single-level index entry deletion:

- **Dense indices** – deletion of search-key is similar to file record deletion.
- **Sparse indices** –
  - if an entry for the search key exists in the index, it is deleted by replacing the entry in the index with the next search-key value in the file (in search-key order).
  - If the next search-key value already has an index entry, the entry is deleted instead of being replaced.

Index Update: Insertion

- **Single-level index insertion:**
  - Perform a lookup using the search-key value of the record to be inserted.
  - **Dense indices** – if the search-key value does not appear in the index, insert it
    - Indices are maintained as sequential files
    - Need to create space for new entry, overflow blocks may be required



10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

32222	Verdi	Music	48000	

  - **Sparse indices** – if index stores an entry for each block of the file, no change needs to be made to the index unless a new block is created.
    - If a new block is created, the first search-key value appearing in the new block is inserted into the index.

- **Multilevel insertion and deletion:** algorithms are simple extensions of the single-level algorithms

What is a B+-Tree?

- A B+-tree is a rooted tree satisfying the following properties:
- All paths from root to leaf are of the same length
- 3 types of nodes: root, internal, and leaf
- Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children.
  - n is a number set when the database is set up
  - Can be large (e.g. n = 100)
- A leaf node has between $\lceil (n–1)/2 \rceil$ and $n–1$ values
- Special cases:
  - If the root is not a leaf, it has at least 2 children.
  - If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and $(n–1)$ values.

B+-Tree Index Files

- Disadvantage of indexed-sequential files
  - Performance degrades as file grows, since many overflow blocks get created.
  - Periodic reorganization of entire file is required.
- Advantage of B+-tree index files:
  - Automatically reorganizes itself with small, local, changes, in the face of insertions and deletions.
  - Reorganization of entire file is not required to maintain performance.
- (Minor) disadvantage of B+-trees:
  - Extra insertion and deletion overhead, space overhead.
- Advantages of B+-trees outweigh disadvantages
  - B+-trees are used extensively

B+-Tree Node Structure

- Typical node

  - K_i are the search-key values
  - P_i are pointers to children for non-leaf nodes
  - P_i are pointers to records or buckets of records for leaf nodes.

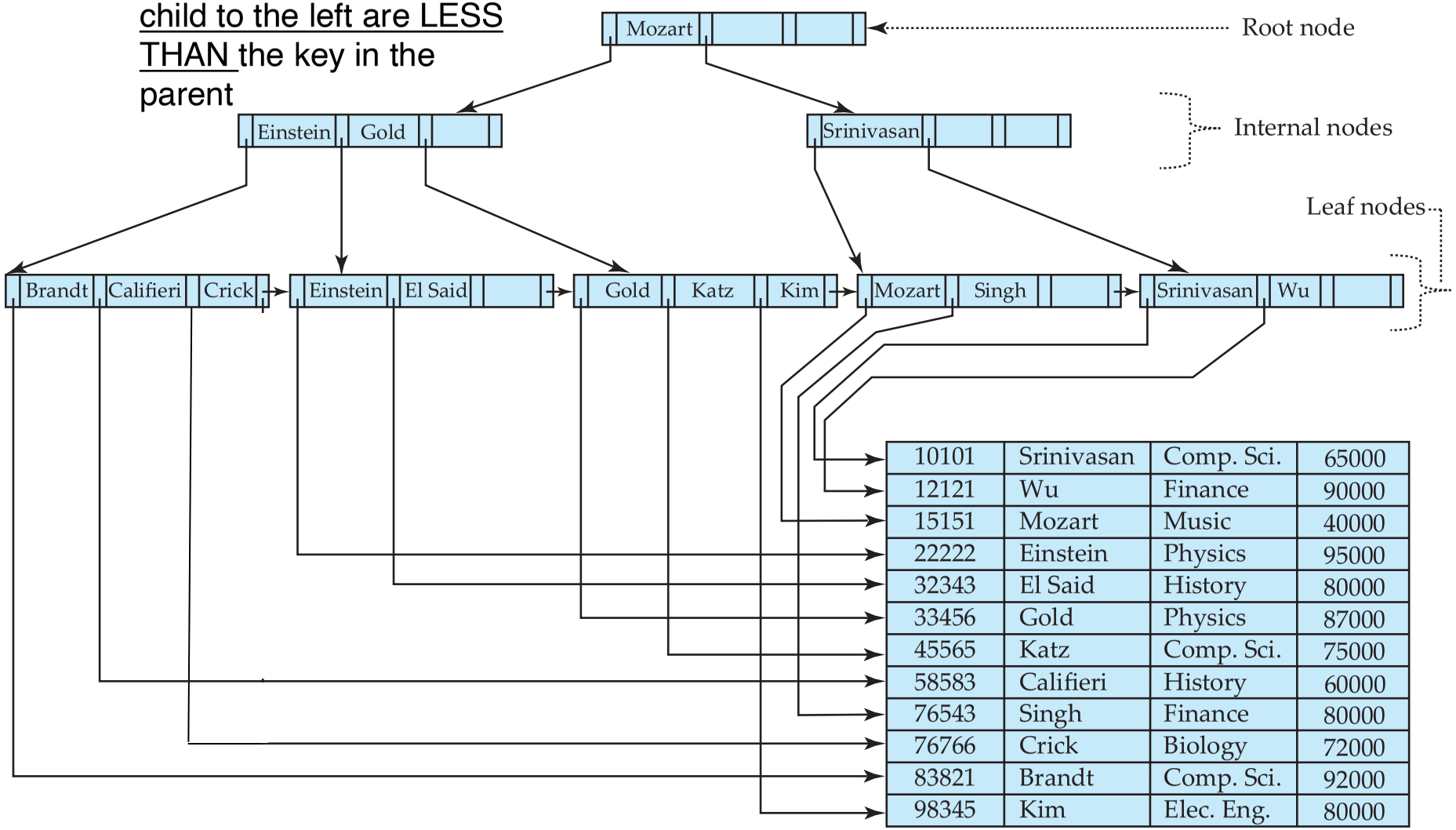
- The search-keys in a node are ordered

$$K_1 < K_2 < K_3 < \ldots < K_{n-1}$$

(Initially assume no duplicate keys, address duplicates later)

Example of B+-Tree

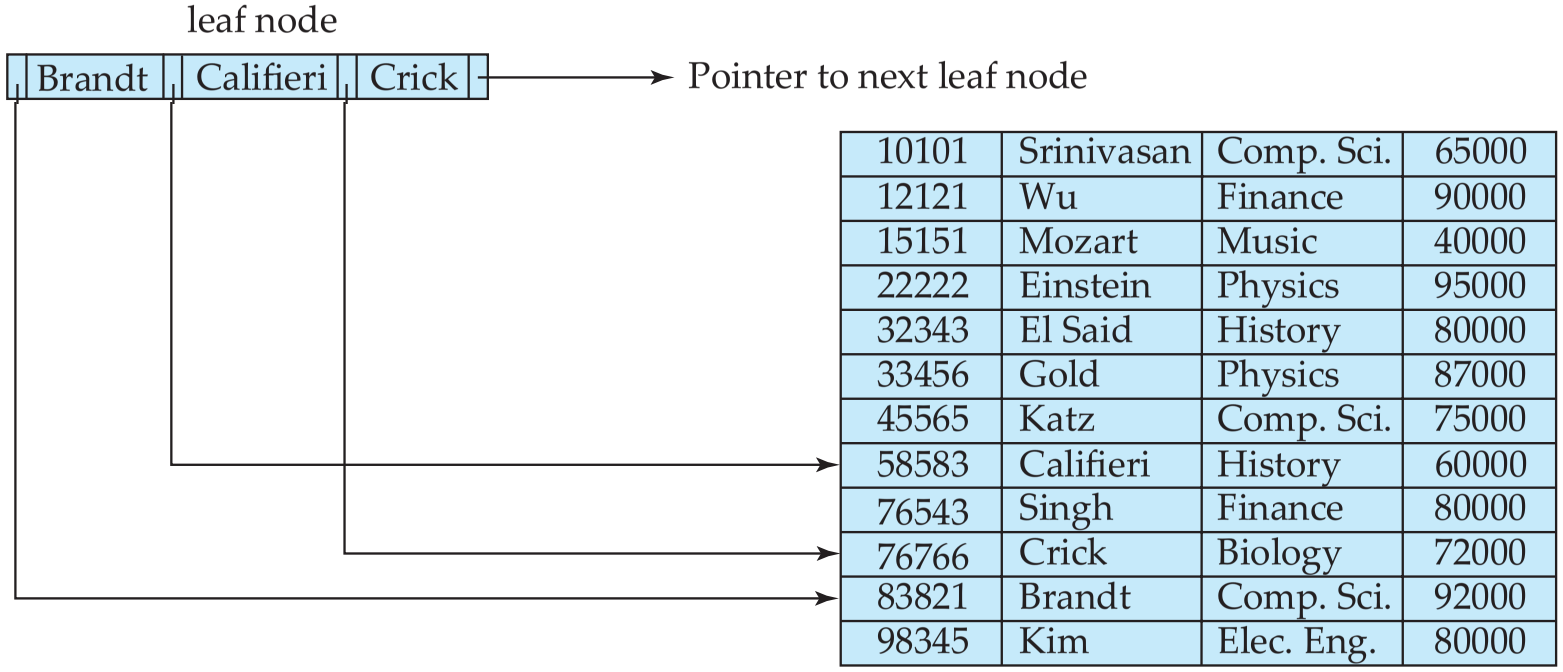
Note that the keys in the child to the left are LESS THAN the key in the parent



10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Leaf Nodes in B+-Trees

- For $i = 1, 2, \ldots, n-1$, pointer P_i points to a file record with search-key value K_i,
  - P_1 points to the record for K_1
- If L_i, L_j are leaf nodes and $i < j$:
  - L_i's search-key values are less than or equal to L_j's search-key values
- P_n points to next leaf node in search-key order



10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Non-Leaf Nodes in B+-Trees

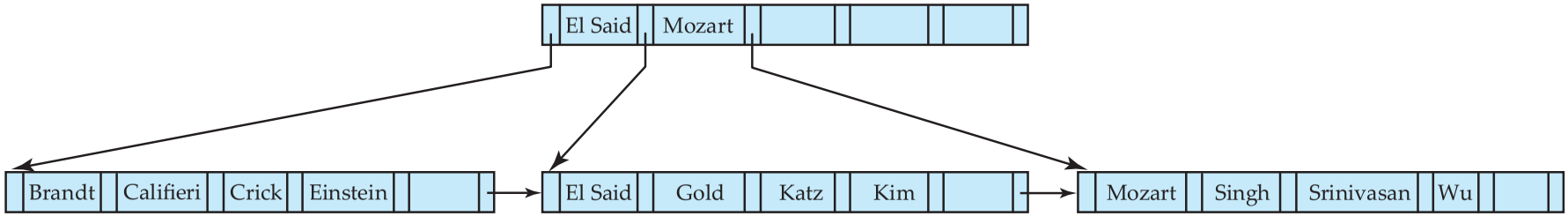
- Non leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with m pointers:
  - General structure

P_1	K_1	P_2	$\cdots$	P_{n-1}	K_{n-1}	P_n

  - All the search-keys in the subtree to which P_1 points are less than K_1
  - For $2 \leq i \leq n-1$, all the search-keys in the subtree to which P_i points have values greater than or equal to K_{i-1} and less than K_i
  - All the search-keys in the subtree to which P_n points have values greater than or equal to K_{n-1}

Example of B+-tree

- B+-tree for *instructor* file ($n = 6$, n = number of pointers)



- Leaf nodes must have between 3 and 5 values ($\lceil (n–1)/2 \rceil$ and $n – 1$, with $n = 6$).
- Non-leaf nodes other than root must have between 3 and 6 children ($\lceil (n/2 \rceil$ and n with $n = 6$).
- Root must have at least 2 children.
  - Otherwise, just have the root node

Updates on B+-Trees: Insertion

- Splitting a leaf node:
  - take the n (search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first $\lceil n/2 \rceil$ in the original node, and the rest in a new node.
  - let the new node be p, and let k be the least key value in p. Insert (k,p) in the parent of the node being split.
  - If the parent is full, split it and **propagate** the split further up.

- Splitting of nodes proceeds upwards till a node that is not full is found.
  - In the worst case the root node may be split increasing the height of the tree by 1.

- Note: Should review the slides for inserts from lecture 16

Static Hashing

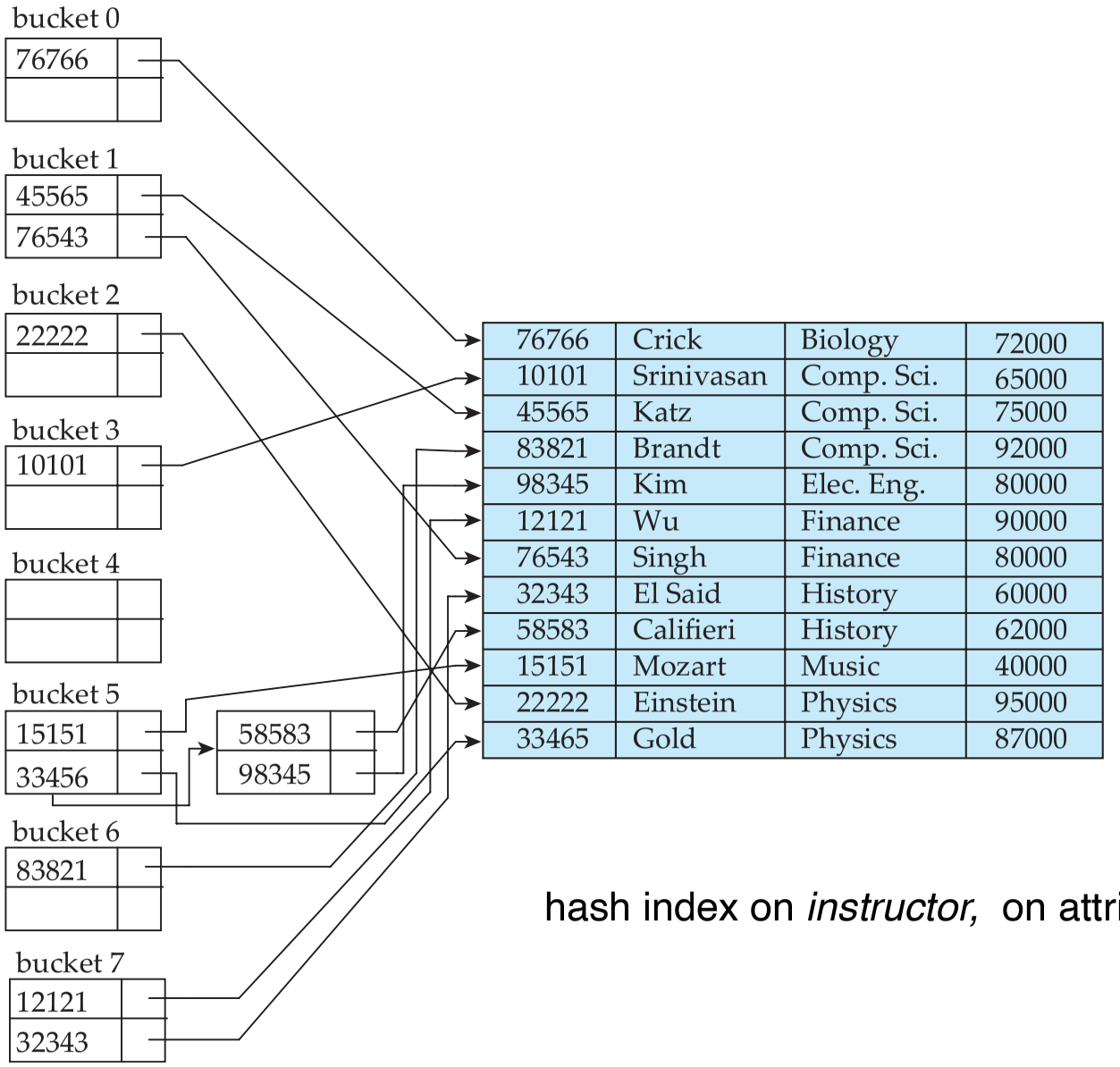
- A **bucket** is a unit of storage containing one or more entries (a bucket is typically a disk block).
  - we obtain the bucket of an entry from its search-key value using a **hash function**
- Hash function *h* is a function from the set of all search-key values *K* to the set of all bucket addresses *B.*
- Hash function is used to locate entries for access, insertion as well as 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.
- In a **hash index**, buckets store entries with pointers to records
- In a **hash file-organization** buckets store records

Handling of Bucket Overflows

- Bucket overflow can occur because of
  - Insufficient buckets
  - Skew in distribution of records. This can occur due to two reasons:
    - multiple records have same search-key value
    - chosen hash function produces non-uniform distribution of key values

- Although the probability of bucket overflow can be reduced, it cannot be eliminated; it is handled by using ***overflow buckets***.

Example of Hash Index



hash index on *instructor,* on attribute *ID*

NoSQL Database Systems

- No SQL / Not Only SQL
- Store and retrieve data in formats other than tables
- Primary objectives of NoSQL
  - Design simplicity
  - Horizontal scaling
    - Adding more servers
    - Scaling for relational database systems is “vertical”
      - Improve performance by adding more CPU or RAM
  - Finer control over data availability
    - Still issues around data protections

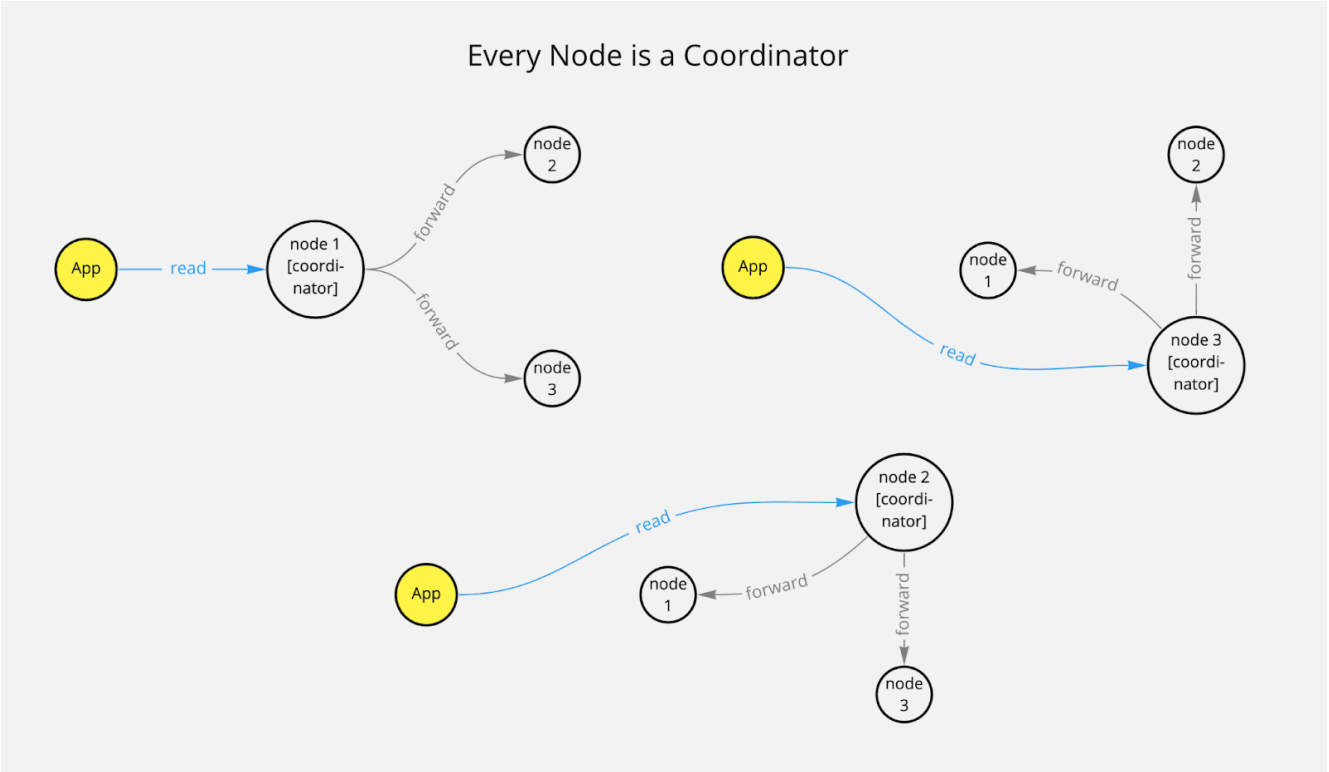
Relational Databases vs NoSQL

Relational Database	NoSql Database
Supports powerful query language.	Supports very simple query language.
It has a fixed schema.	No fixed schema.
Follows ACID (Atomicity, Consistency, Isolation, and Durability).	It is only “eventually consistent”.
Supports transactions.	Does not support transactions.

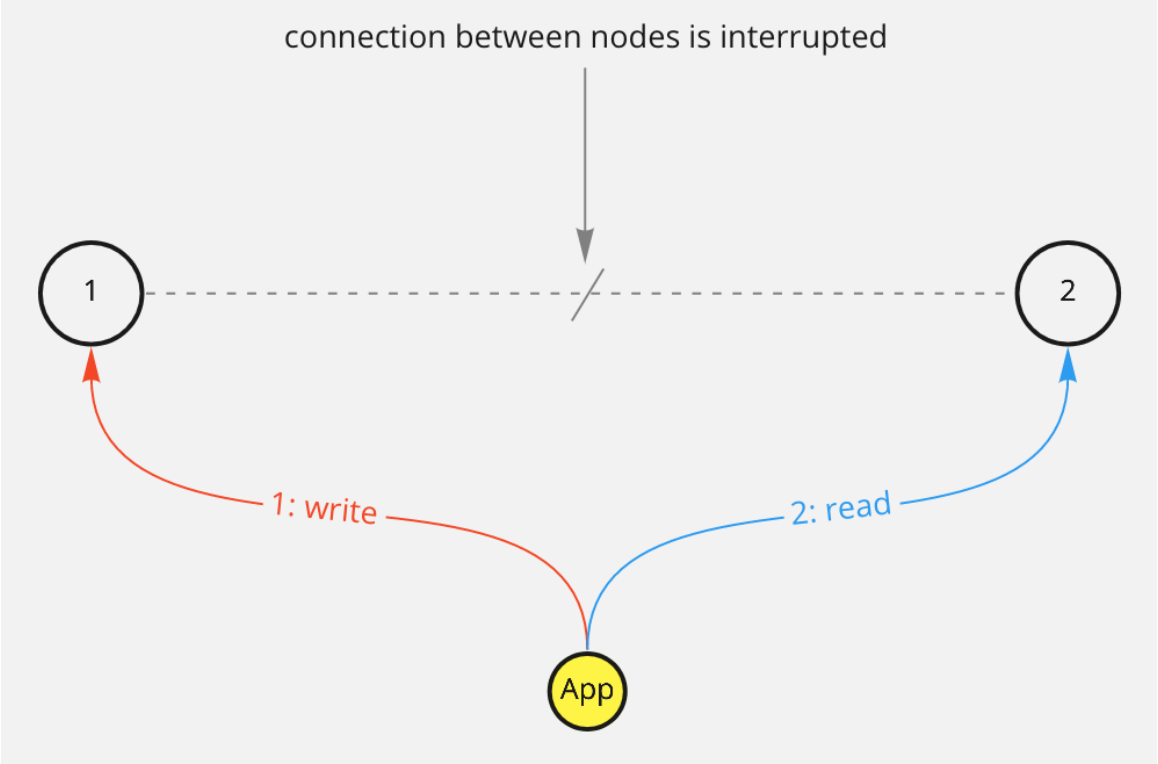
Data Replication

- Data is replicated to different nodes. If certain data is requested, a request can be processed from any node.

- Every Node Is a Coordinator:
  - handles read and write requests
  - Figures out which node has which data



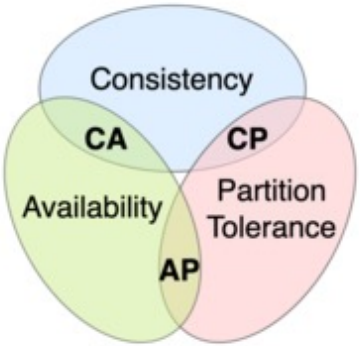
Consistency



- Should you disallow writes to all nodes to ensure consistency? This means availability would be sacrificed to ensure consistency and correctness.

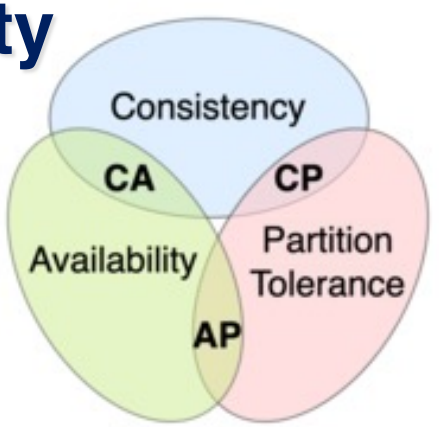
- Accept the write to node 1 and keep serving reads from both nodes. This would keep the system available but depending on what node you read from, the answer will be different, which means sacrificing consistency over availability.

Consistency Issues



- Consistency vs Availability
  - **CAP Theorem**
    - Consistency: Every read receives the most recent write or an error
    - Availability: Every request receives a (non-error) response, **without** the guarantee that it contains the most recent write
    - Partition tolerance: The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes.
    - Can only optimize for 2 of these, not all 3
  - When the data is partitioned, there exists a possibility that data written to one node may not yet have made it to the other nodes.
    - What do you do?

Consistency vs Availability



- ACID compliance => choosing consistency over availability
- NoSQL databases are often BASE compliant
  - Basically Available
    - Reading and writing are available as much as possible, but might not be consistent
  - Soft-state
    - Only have some probability of knowing the state of the system
  - Eventually consistent
    - If we wait long enough, the data will become consistent across the nodes

Partitioning / Sharding