Chapter 4: Data Storage and Indexing

Chapter 4: Data Storage and Indexing

- Overview of Physical Storage Media
- File Organization
- Organization of Records in Files
- Data-Dictionary Storage
- Database Buffer
- Basic Concepts - Indexing
- Ordered Indices
- B^+-Tree Index Files
- B-Tree Index Files
- Static Hashing
- Dynamic Hashing
- Comparison of Ordered Indexing and Hashing
- Index Definition in SQL
- Multiple-Key Access

Classification of Physical Storage Media

- Speed with which data can be accessed
- Cost per unit of data
- Reliability
  - data loss on power failure or system crash
  - physical failure of the storage device
- Can differentiate storage into:
  - **volatile storage:** loses contents when power is switched off
  - **non-volatile storage**:
    - Contents persist even when power is switched off.
    - Includes secondary and tertiary storage, as well as battery-back up main-memory.

Physical Storage Media

- **Cache** – fastest and most costly form of storage; volatile; small; managed by the computer system hardware.
- **Main memory**:
  - The general-purpose machine instructions operate on main memory.
  - fast access (10s to 100s of nanoseconds; 1 nanosecond = 10^{-9} seconds)
  - generally too small (or too expensive) to store the entire database
    - capacities of up to a few Gigabytes widely used currently
    - Capacities have gone up and per-byte costs have decreased steadily and rapidly (roughly factor of 2 every 2 to 3 years)
  - **Volatile** — contents of main memory are usually lost if a power failure or system crash occurs.

Physical Storage Media (Cont.)

- **Flash memory**
  - Data survives power failure
  - also known as EEPROM (Electrically Erasable Programmable Read-Only Memory)
  - Reading data from flash memory takes less than 100 nanosec.
  - Data can be written at a location only once, but location can be erased and written to again
    - Can support only a limited number (10K – 1M) of write/erase cycles.
    - Erasing of memory has to be done to an entire bank of memory
  - Reads are roughly as fast as main memory
  - But writes are slow (few microseconds), erase is slower
  - Cost per unit of storage roughly similar to main memory
  - Widely used in embedded devices such as digital cameras, phones, and USB keys

Physical Storage Media (Cont.)

- **Magnetic-disk**
  - Data is stored on spinning disk, and read/written magnetically
  - Primary medium for the long-term storage of data; typically stores entire database.
  - Data must be moved from disk to main memory for access, and written back for storage
    - Much slower access than main memory
  - **direct-access** – possible to read data on disk in any order, unlike magnetic tape
  - Capacities range from 80 GB to roughly 1.5 TB as of 2009
  - The **Samsung** 2.5 inch is the world's largest hard drive with 16 TB capacity.Aug 8, 2016.
  - As of August 2020, the largest hard drive is **20 TB** (while SSDs can be much bigger at 100 TB, mainstream consumer SSDs cap at 8 TB).
  - Smaller, 2.5-inch drives, are available at up to 2TB for laptops, and 5TB as external drives.

Physical Storage Media (Cont.)

- **Magnetic-disk**
  - Much larger capacity and cost/byte than main memory/flash memory
  - Growing constantly and rapidly with technology improvements (factor of 2 to 3 every 2 years)
- Survives power failures and system crashes
  - disk failure can destroy data, but is rare

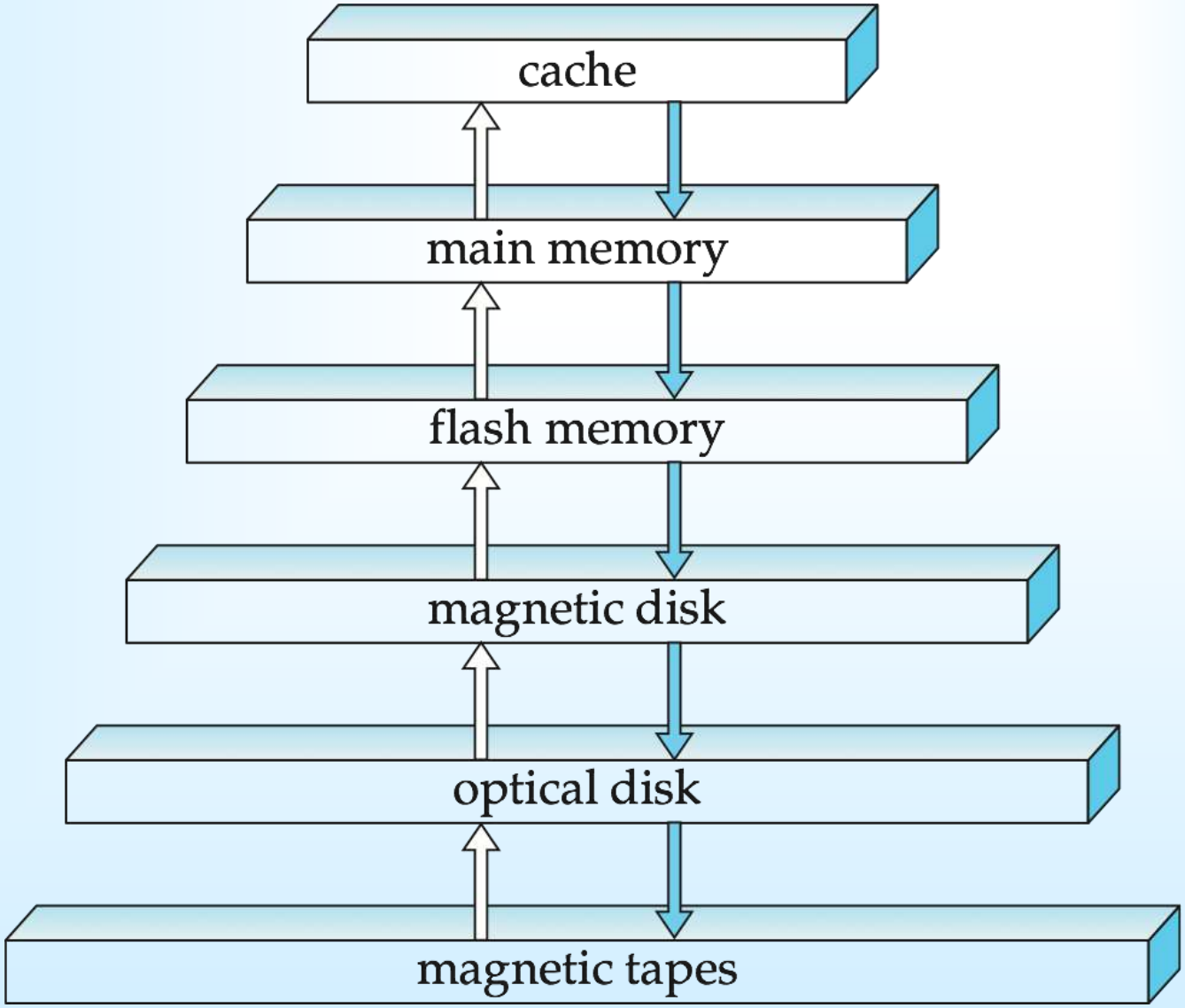
Physical Storage Media (Cont.)

- **Optical storage**
  - The most popular form of optical storage is compact-disk (CD) -70 MB of data.
  - non-volatile, data is read optically from a spinning disk using a laser
  - CD-ROM (640 MB) and DVD (4.7 to 17 GB) most popular forms
  - Blu-ray disks: 27 GB to 54 GB
  - Write-one, read-many (WORM) optical disks used for archival storage (CD-R, DVD-R, DVD+R)
  - Multiple write versions also available (CD-RW, DVD-RW, DVD+RW, and DVD-RAM)
  - Reads and writes are slower than with magnetic disk
  - **Juke-box** systems, with large numbers of removable disks, a few drives, and a mechanism for automatic loading/unloading of disks available for storing large volumes of data

Physical Storage Media (Cont.)

- **Tape storage**
  - non-volatile, used primarily for backup (to recover from disk failure), and for archival data
  - Cheaper than disk.
  - **sequential-access** – much slower than disk, because the tape must be accessed sequentially from beginning.
  - very high capacity (40 to 300 GB tapes available)
  - Sony announced in 2014 that they had developed a tape storage technology with the highest reported magnetic tape data density, potentially allowing tape capacity of **185 TB**.
  - tape can be removed from drive $\Rightarrow$ storage costs much cheaper than disk, but drives are expensive
  - Tape jukeboxes available for storing massive amounts of data
    - hundreds of terabytes (1 terabyte = 10^9 bytes) to even multiple **petabytes** (1 petabyte = 10^{12} bytes)

Storage Hierarchy



Pen Drive

- A small pen size flash memory device integrated with USB (Universal Serial Bus) interface offering easy, fast and reliable way for storing and transferring digital files.
- **Features:**
  - **Portable**: small, compact plug and play making it fit it into the hand.
  - **Non-volatile storage**: Contents are rewritable and does not lose its content even when removed from the USB port.
  - **Storage capacity**: available up to 512 GB and steadily incrementing in capacity and size.
  - **Compatibility**: advanced pen drives are compatible with Windows, Mac, Linux etc.
  - **Transfer rate**: varies between 5 MBPS to 60 MBPS or more depending upon the vendor.
  - **Cost**: one of the cheapest form of storage device.

Pen Drive

- **Uses**:
  - Transferring of files from one computer to another.
  - Secure storage of audio, video, data, application software's.
  - As a backup solution to store data before formatting your PC.
  - Using it as a temporary RAM to improve performance of the system.
  - As a booting solution to launch new operating system from pen drive.

Storage Hierarchy (Cont.)

- **primary storage:** Fastest media but volatile (cache, main memory).
- **secondary storage:** next level in hierarchy, non-volatile, moderately fast access time
  - also called **on-line storage**
  - E.g. flash memory, magnetic disks
- **tertiary storage:** lowest level in hierarchy, non-volatile, slow access time
  - also called **off-line storage**
  - E.g. magnetic tape, optical storage

File Organization

- The database is stored as a collection of *files*. These files reside permanently on disks.
- Each file is a sequence of *records.*
- A record is a sequence of fields.
- Each file is logically partitioned into fixed-length storage units called **blocks,** which are the units of both storage allocation and data transfer. The records are mapped onto disk blocks.
- Most databases use block sizes of 4 to 8 kilobytes by default

Instructor File

ID	*name*	*dept_name*	*salary*
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

Block 1

Block 2

Block 3

File Organization

- Two approaches:
  - **Fixed Length Records**:
    - Assume record size is fixed
    - Each file has records of one particular type only
    - Different files are used for different relations
  - **Variable length records:**
    - Storage of multiple record types in a file.
    - Record types that allow variable lengths for one or more fields.
    - Record types that allow repeating fields such as arrays or multiset.

Fixed-Length Records

- Let us consider a file of *instructor* records for our university database

```
type instructor = record
        ID char (5);
        name char(20);
        dept name char (20);
        salary numeric (8,2);
end
```

- Assume that each character occupies 1 byte and that numeric (8,2) occupies 8 bytes. Then, the *instructor* record is 53 bytes long.
- A simple approach is to use the first 53 bytes for the first record, the next 53 bytes for the second record, and so on.

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

- There are two problems with this simple approach:
  - 1. Unless the block size happens to be a multiple of 53 (which is unlikely), some records will cross block boundaries. That is, part of the record will be stored in one block and part in another. It would thus require two block accesses to read or write such a record.
  - 2. It is difficult to delete a record from this structure. The space occupied by the record to be deleted must be filled with some other record of the file, or we must have a way of marking deleted records so that they can be ignored.

  To avoid the first problem, we allocate only as many records to a block as would fit entirely in the block (this number can be computed easily by dividing the block size by the record size, and discarding the fractional part). Any remaining bytes of each block are left unused.

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
- 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*

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

Deleting record 3 and compacting

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
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

- When a record is deleted, we could move the record that came after it into the space formerly occupied by the deleted record, and so on, until every record following the deleted record has been moved ahead.
- Such an approach requires moving a large number of records

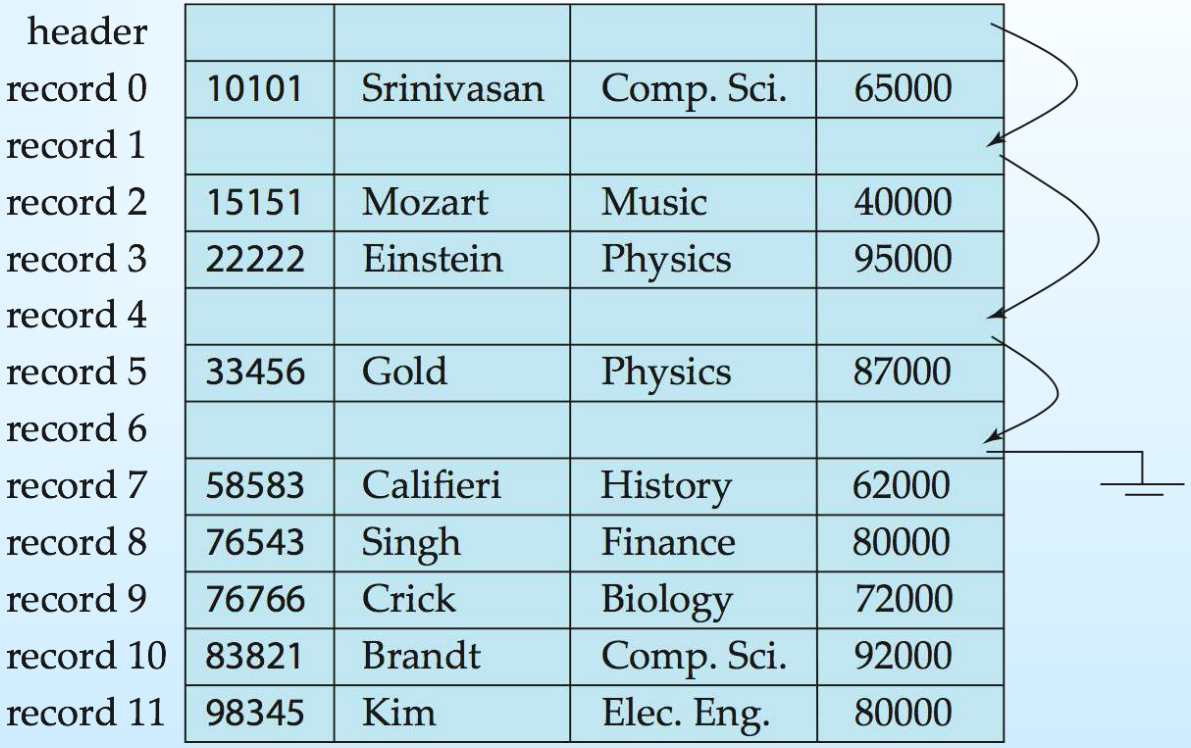
Deleting record 3 and moving last record

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 11	98345	Kim	Elec. Eng.	80000
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

- It might be easier simply to move the final record of the file into the space occupied by the deleted record.
- It is undesirable to move records to occupy the space freed by a deleted record, since doing so requires additional block accesses.

Free Lists

- Store the address of the first deleted record in the file header.
- Use this first record to store the address of the second deleted record, and so on
- Can think of these stored addresses as pointers since they "point" to the location of a record.
- More space efficient representation: reuse space for normal attributes of free records to store pointers. (No pointers stored in in-use records.)



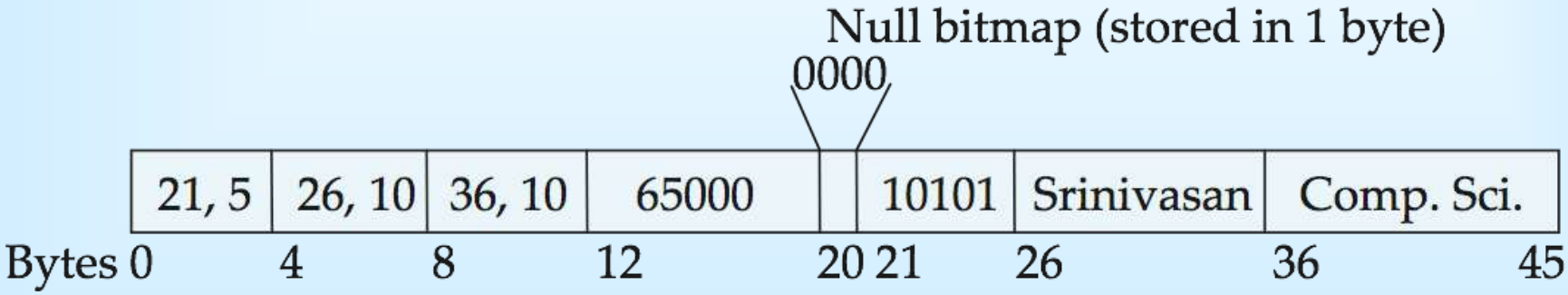
header				
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1				
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4				
record 5	33456	Gold	Physics	87000
record 6				
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology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

Variable-Length Records

- Variable-length records arise in database systems in several ways:
  - Storage of multiple record types in a file.
  - Record types that allow variable lengths for one or more fields.
  - Record types that allow repeating fields such as arrays or multiset.
- Two different problems must be solved by any such technique:
  - How to represent a single record in such a way that individual attributes can be extracted easily.
  - How to store variable-length records within a block, such that records in a block can be extracted easily.

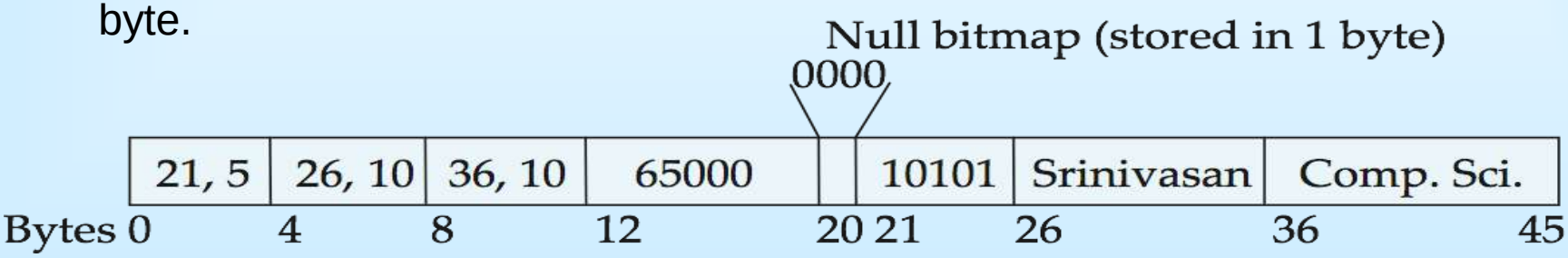
Variable-Length Records

- The representation of a record with variable-length attributes typically has two parts:
- An initial part with fixed length attributes, followed by data for variable length attributes.
- Fixed-length attributes, such as numeric values, dates, or fixed length character strings are allocated as many bytes as required to store their value.
- Variable-length attributes, such as varchar types, are represented by a pair (*offset, length), where offset* denotes *where* the data for that attribute begins within the record, and *length is the length in* bytes of the variable-sized attribute.

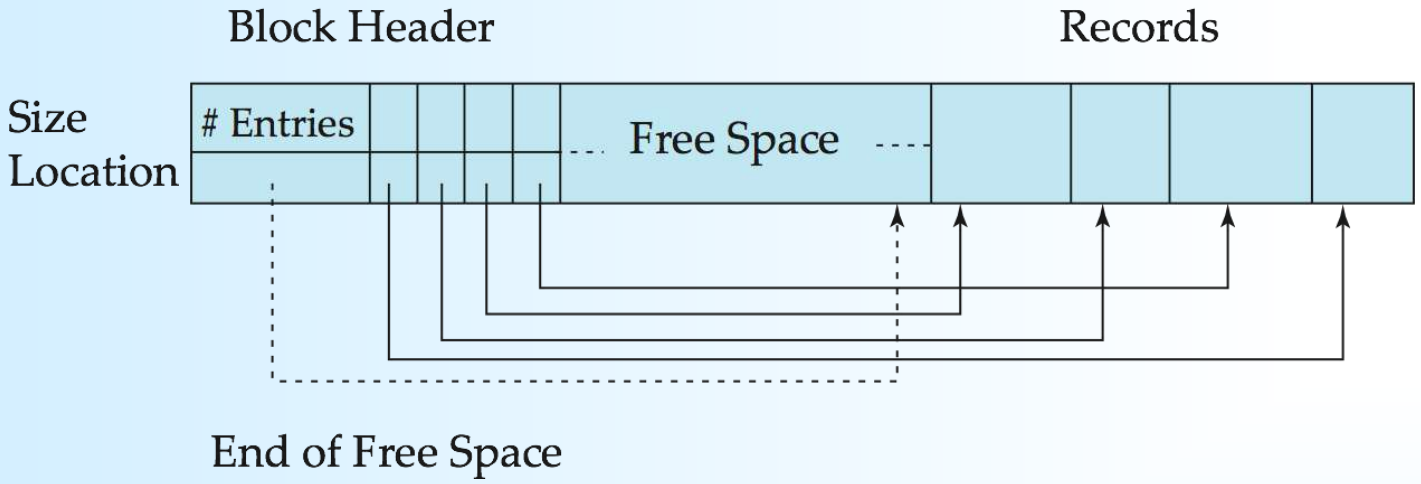


Variable-Length Records

- The figure shows an *instructor record, whose first three attributes ID, name, and dept name* are variable-length strings, and whose fourth attribute *salary is a fixed-sized number.*
- We assume that the offset and length values are stored in two bytes each, for a total of 4 bytes per attribute.
- The *salary attribute is assumed to be stored in 8 bytes,* and each string takes as many bytes as it has characters.
- The figure also illustrates the use of a **null bitmap,** which indicates which attributes of the record have a null value.
- In this particular record, if the salary were null, the fourth bit of the bitmap would be set to 1, and the *salary value stored* in bytes 12 through 19 would be ignored.
- Since the record has four attributes, the null bitmap for this record fits in 1 byte.



Variable-Length Records: Slotted Page Structure



- **Slotted page** header contains:
  - number of record entries
  - end of free space in the block
  - An array whose entries contain location and size of each record
- Records can be moved around within a page to keep them contiguous with no empty space between them; entry in the header must be updated.
- Pointers should not point directly to record — instead they should point to the entry for the record in header.

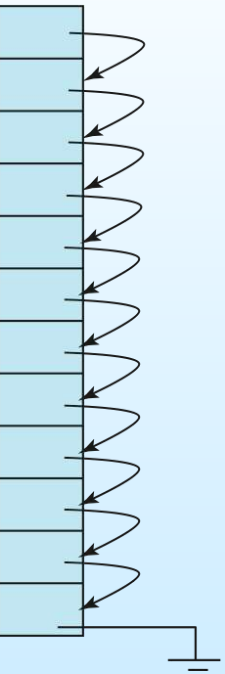
Organization of Records in Files

- **Heap -** Any record can be placed anywhere in the file where there is space for the record. There is no ordering of records. Typically, there is a single file for each relation.
- **Sequential** – Records are stored in sequential order, according to the “search key” value of each record..
- **Hashing** – a hash function computed on some attribute of each record; the result specifies in which block of the file the record should be placed.
- Generally, records of each relation may be stored in a separate file. In a **multitable clustering file organization** records of several different relations can be stored in the same file.
  - Motivation: store related records on the same block to minimize I/O.

Sequential File Organization

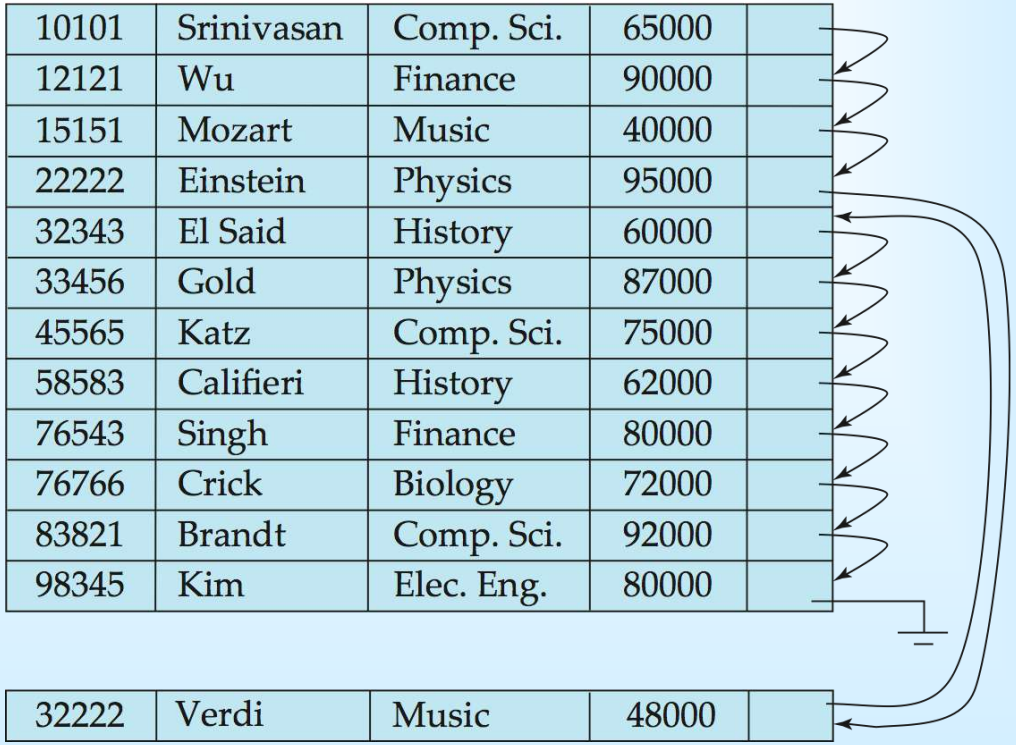
- **Sequential file:** Records are stored in sequential order, according to the "search key" value of each record..
- A **search key** is any attribute or set of attributes; it need not be the primary key, or even a superkey.
- Ex: The records in the instructor file are ordered by a search-key(ID).
- To permit fast retrieval of records in search-key order, we chain together records by pointers.
- The pointer in each record points to the next record 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	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	



Sequential File Organization (Cont.)

- 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.
- Such reorganizations are costly, and must be done during times when the system load is low.
- Deletion – use pointer chains

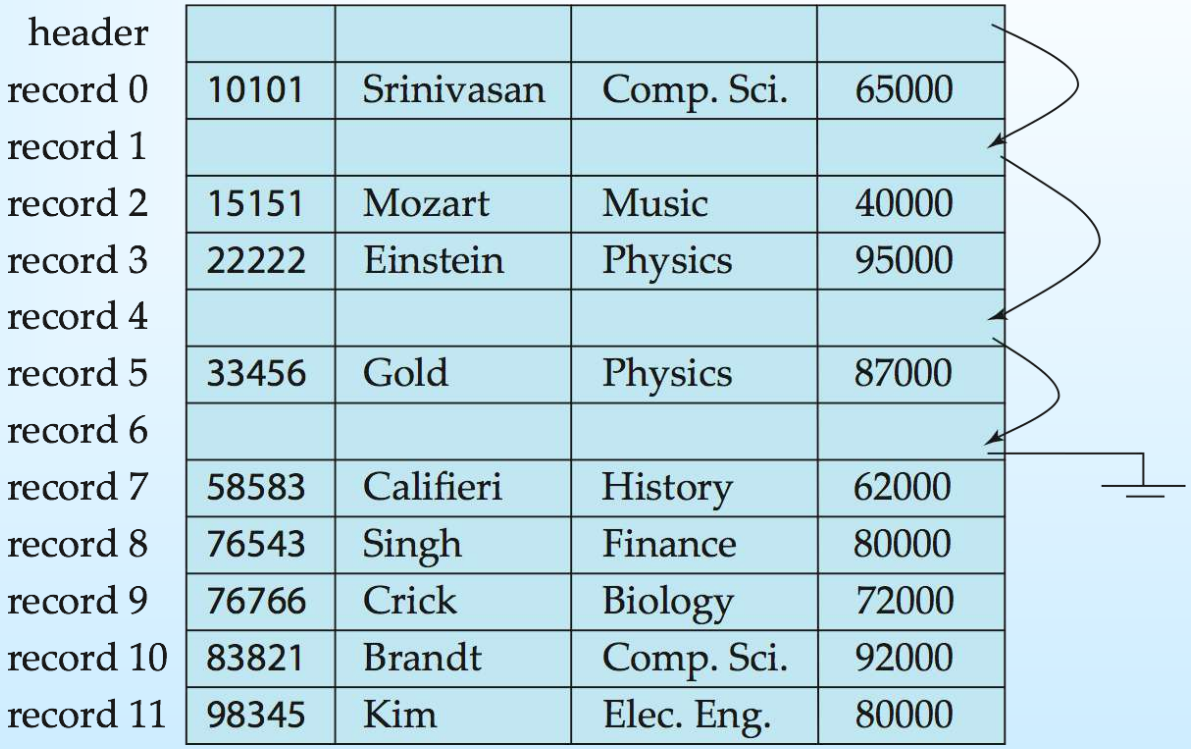


10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp.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

32222	Verdi	Music	48000	

Free Lists

- Store the address of the first deleted record in the file header.
- Use this first record to store the address of the second deleted record, and so on
- Can think of these stored addresses as pointers since they "point" to the location of a record.
- More space efficient representation: reuse space for normal attributes of free records to store pointers. (No pointers stored in in-use records.)



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

Multitable Clustering File Organization

Store several relations in one file using a **multitable clustering** file organization

department

dept_name	*building*	*budget*
Comp. Sci.	Taylor	100000
Physics	Watson	70000

instructor

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

multitable clustering of *department* and *instructor*

Comp. Sci.	Taylor	100000
45564	Katz	75000
10101	Srinivasan	65000
83821	Brandt	92000
Physics	Watson	70000
33456	Gold	87000

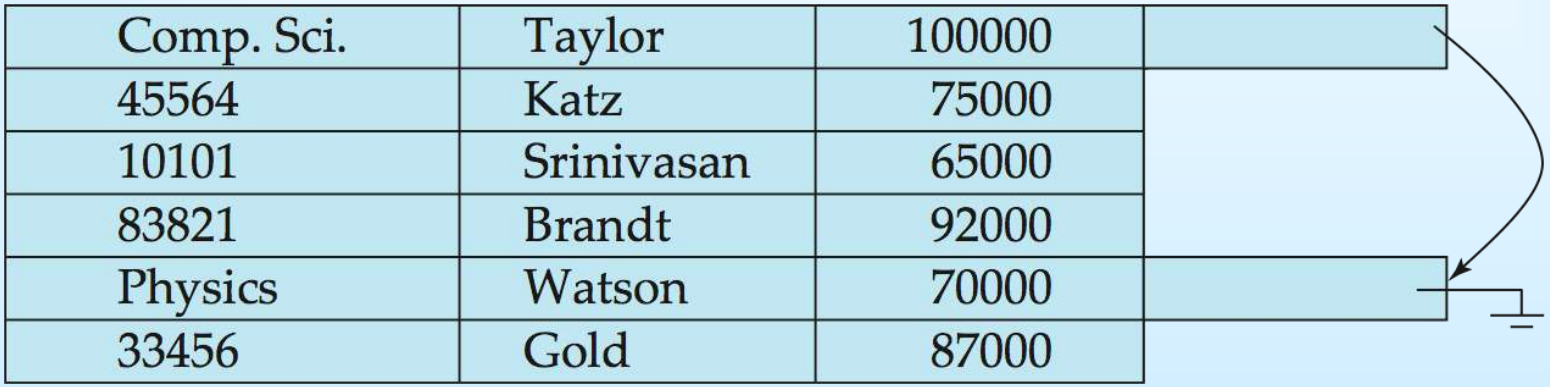
Multitable Clustering File Organization (cont.)

- Good for queries involving *department* ⋈ *instructor*, and for queries involving one single department and its instructors
- Bad for queries involving only *department or instructor*

  select *

  from *department;*

  requires more block accesses
- Can add pointer chains to link records of a particular relation
- Results in variable size records



Comp. Sci.	Taylor	100000	
45564	Katz	75000	
10101	Srinivasan	65000	
83821	Brandt	92000	
Physics	Watson	70000	
33456	Gold	87000	

Data Dictionary Storage

- A relational database system needs to maintain the data about relations (metadata), such as the schema of the relations.
- Relational schemas and other metadata about relations are stored in a structure called the **data dictionary or system catalog.**
  - Information about relations
    - Names of relations
    - Names of attributes of each relation
    - Domains and length of attributes
    - Names and definitions of views
    - integrity constraints
  - Data on users of the system
    - Names of authorized users
    - Accounting information about users
    - Passwords or other information used to authenticate users

- Statistical and descriptive data
  - number of tuples in each relation
  - Method of storage for each relation
- Physical file organization information
  - How relation is stored (sequential/hash/…)
  - Physical location of relation
    - operating system file name or
    - disk addresses of blocks containing records of the relation
- Information about indices
  - name of the index
  - name of relation being indexed
  - attribute on which the index is defined
  - type of index formed

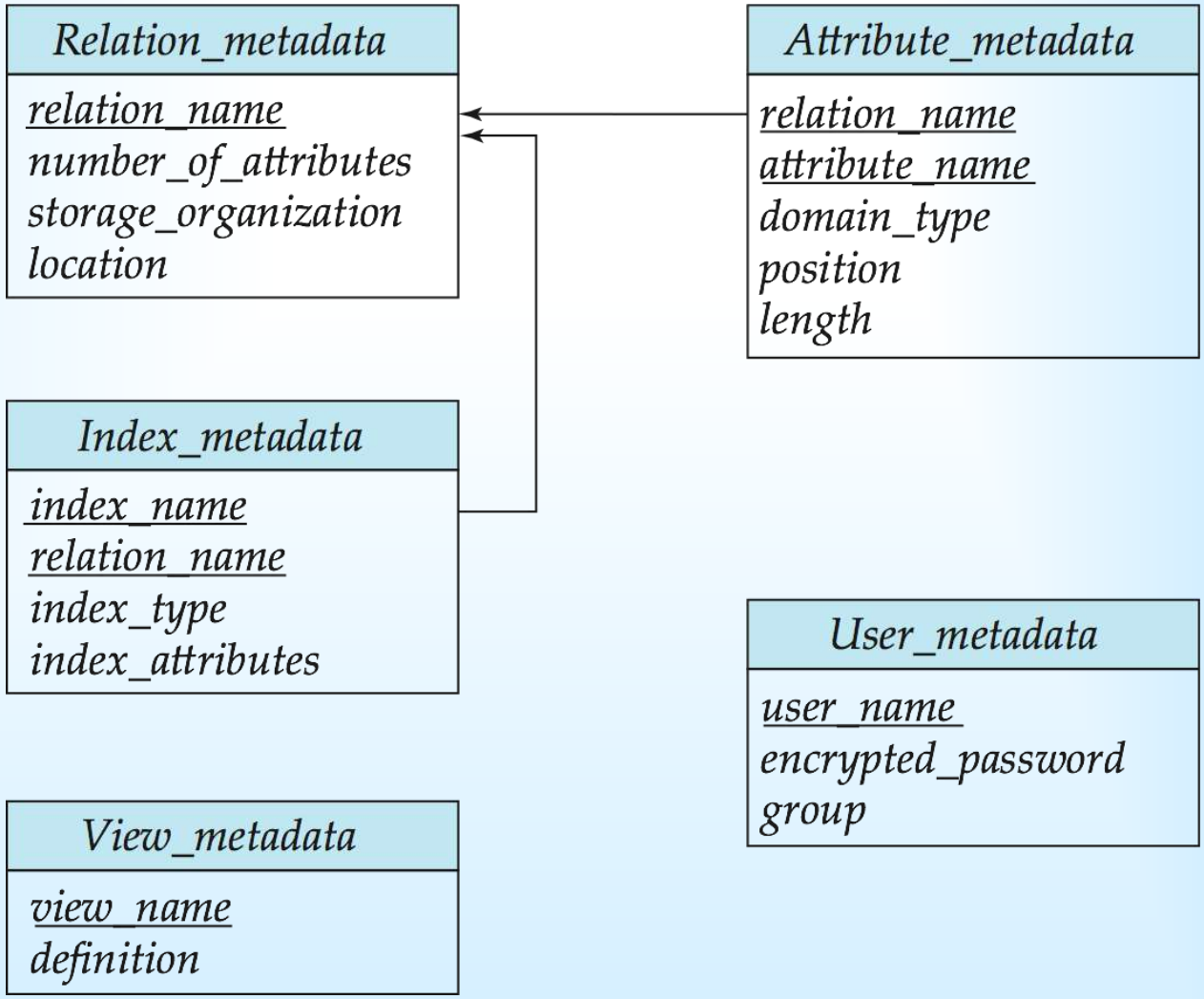
Data Dictionary Storage (Cont.)

- System designer decide how to represent system using relations. One possible representation as follows:

Relation-metadata = (relation-name, number-of-attributes, storage-organization, location)
Attribute-metadata = (attribute-name, relation-name, domain-type, position, length)
User-metadata = (user-name, encrypted-password, group)
Index-metadata = (index-name, relation-name, index-type, index-attributes)
View-metadata = (view-name, definition)

Relational Representation of System Metadata

- Relational representation on disk
- Specialized data structures designed for efficient access, in memory



Storage Access

- A database file is partitioned into fixed-length storage units called **blocks**. Blocks are units of both storage allocation and data transfer.
- Database system's major goal is to minimize the number of block transfers between the disk and memory. We can reduce the number of disk accesses by keeping as many blocks as possible in main memory.
- **Buffer** – portion of main memory available to store copies of disk blocks.
- **Buffer manager** – subsystem responsible for allocating buffer space in main memory.

Buffer Manager

- Programs call on the buffer manager when they need a block from disk.
  1. If the block is already in the buffer, buffer manager returns the address of the block in main memory to the requester.
  2. If the block is not in the buffer, the buffer manager
     1. Allocates space in the buffer for the block
        1. Replacing (throwing out) some other block, if required, to make space for the new block.
        2. Replaced block written back to disk only if it was modified since the most recent time that it was written to/fetched from the disk.
     2. Reads the block from the disk to the buffer, and returns the address of the block in main memory to requester.

Buffer Manager

Typical virtual-memory management schemes:

- **Buffer replacement strategy:** When there is no room left in the buffer, a block must be removed from the buffer before a new one can be read in. Most operating systems use a **least recently used (LRU) scheme, in which** the block that was referenced least recently is written back to disk and is removed from the buffer.
- **Pinned blocks: M**ost recovery systems require that a block should not be written to disk while an update on the block is in progress. A block that is not allowed to be written back to disk is said to be **pinned.**
- **Forced output of blocks.** There are situations in which it is necessary to write back the block to disk, even though the buffer space that it occupies is not needed. This write is called the **forced output of a block.**

Buffer-Replacement Policies

- Most operating systems replace the block **least recently used** (LRU strategy)
- However, a database system is able to predict the pattern of future references more accurately than an operating system.
- Queries have well-defined access patterns (such as sequential scans), and a database system can use the information in a user's query to predict future references
  - LRU can be a bad strategy for certain access patterns involving repeated scans of data
    - For example: when computing the join of 2 relations r and s by a nested loops
      for each tuple *tr* of *r* do
        for each tuple *ts* of *s* do
          if the tuples *tr* and *ts* match …
  - Mixed strategy with hints on replacement strategy provided by the query optimizer is preferable

Buffer-Replacement Policies

Ex: select *
from instructor natural join department;

for each tuple *i of instructor do*
for each tuple *d of department do*
if *i[dept name] = d[dept name]*
then begin
let *x be a tuple defined as follows:*
x[ID] := i[ID]
x[dept name] := i[dept name]
x[name] := i[name]
x[salary] := i[salary]
x[building] := d[building]
x[budget] := d[budget]
include tuple *x as part of result of instructor* ***natural join*** *department*
end
end
end

Buffer-Replacement Policies

Instructor Table

ID	*name*	*dept_name*	*salary*
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

Block 1

Block 2

Block 3

Block 1

Block 2

department relation

dept_name	*building*	*budget*
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Elec. Eng.	Taylor	85000
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000

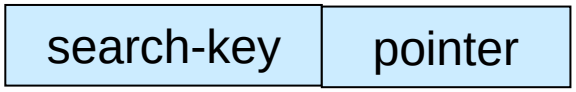
Buffer-Replacement Policies

- Once a tuple of instructor has been processed, that tuple is not needed again. Therefore, once processing of an entire block of instructor tuples is completed, that block is no longer needed in main memory, even though it has been used recently.
- The buffer manager should be instructed to free the space occupied by an instructor block as soon as the final tuple has been processed. This buffer-management strategy is called the **toss-immediate** strategy.
- When processing of a department block is completed, we know that that block will not be accessed again until all other department blocks have been processed. Thus, the most recently used department block will be the final block to be re-referenced, and the least recently used department block is the block that will be referenced next.
- The optimal strategy for block replacement for the above procedure is the **most recently used (MRU) strategy.**

Chapter 11: Indexing and Hashing

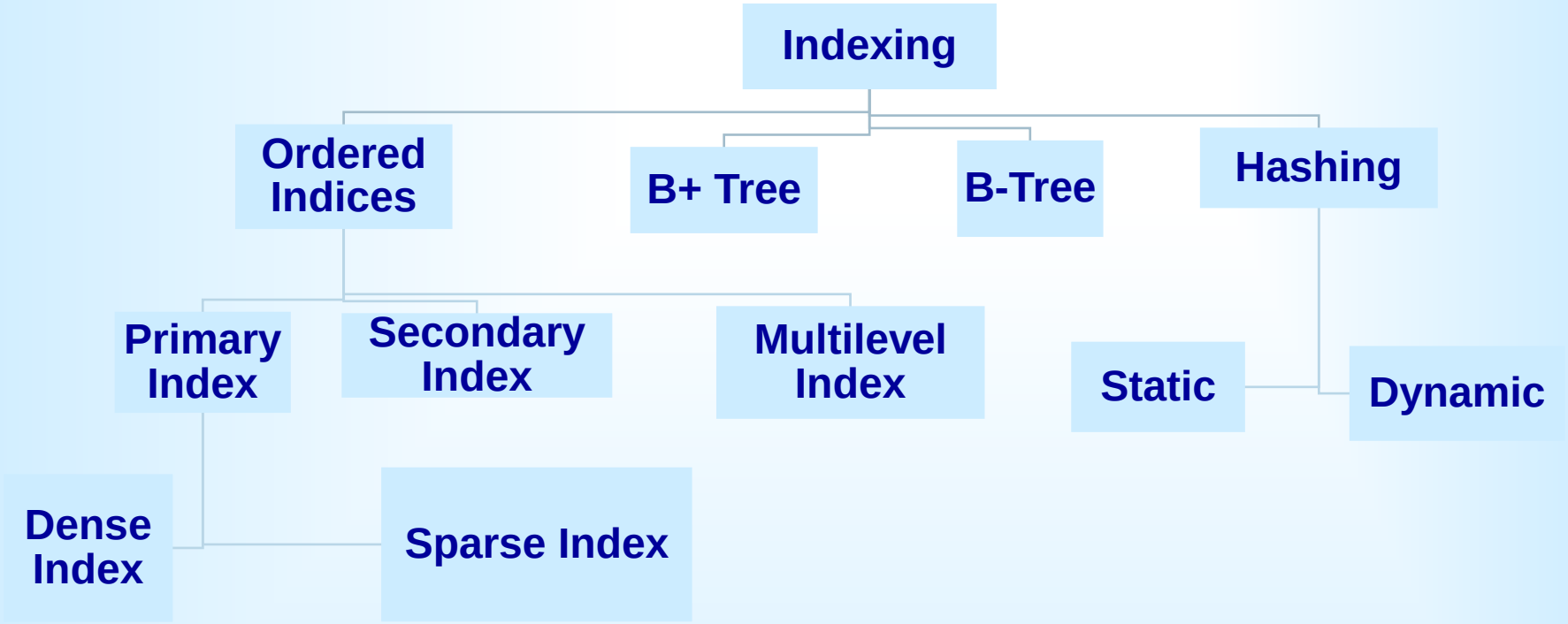
Indexing - Basic Concepts

- Indexing mechanisms used to speed up access to desired data.
  - E.g., author catalog in library
- An **index file** consists of index records (called **index entries**) of the form



- **Search Key** - attribute or set of attributes used to look up records in a file
- 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”.

Indexing - Basic Concepts



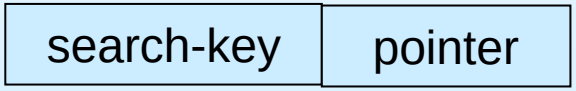
Index Evaluation Metrics

Indexing techniques evaluated on basis of:

- **Access types**: types of access that are supported efficiently.
  - Finding records with a specified attribute value
  - Or finding records with an attribute values falling in a specified range .
- **Access time**
  - The time it takes to find a particular data item, or set of items
- **Insertion time**
  - The time it takes to find the correct place to insert the new data item,
  - the time it takes to update the index structure.
- **Deletion time**
  - The time it takes to find the item to be deleted,
  - the time it takes to update the index structure.
- **Space overhead**
  - The additional space occupied by an index structure.

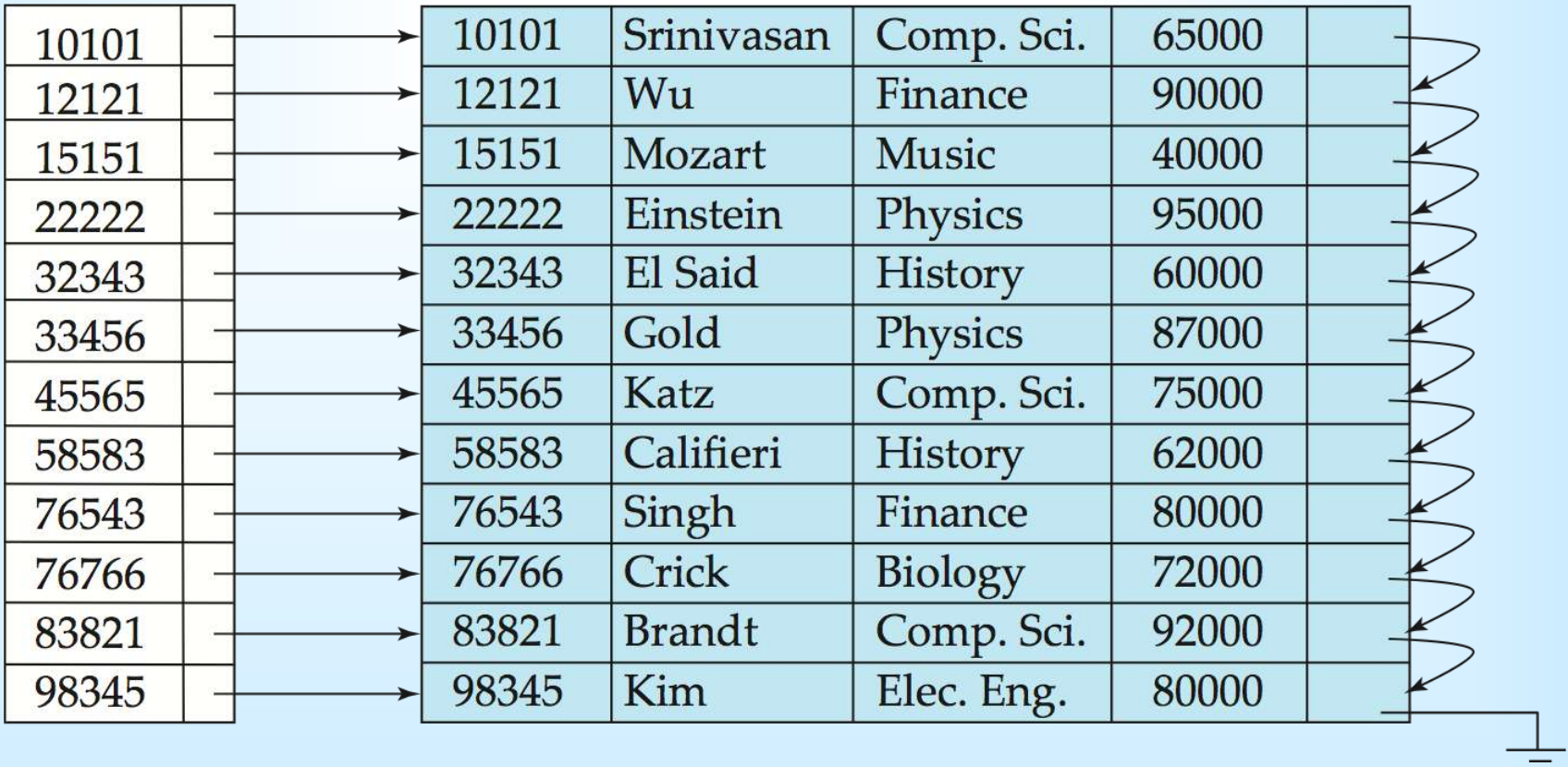
Ordered Indices

- In an ***ordered index***, index entries are stored in sorted order based on the search key value. E.g., author catalog in library.
- ***Primary index*****:** in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
  - Also called ***clustering 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 sequential order of the file. Also called non-clustering index**.**
- Index-sequential file**:** ordered sequential file with a primary index.
- An **index entry or index record** consists of



Dense Index Files

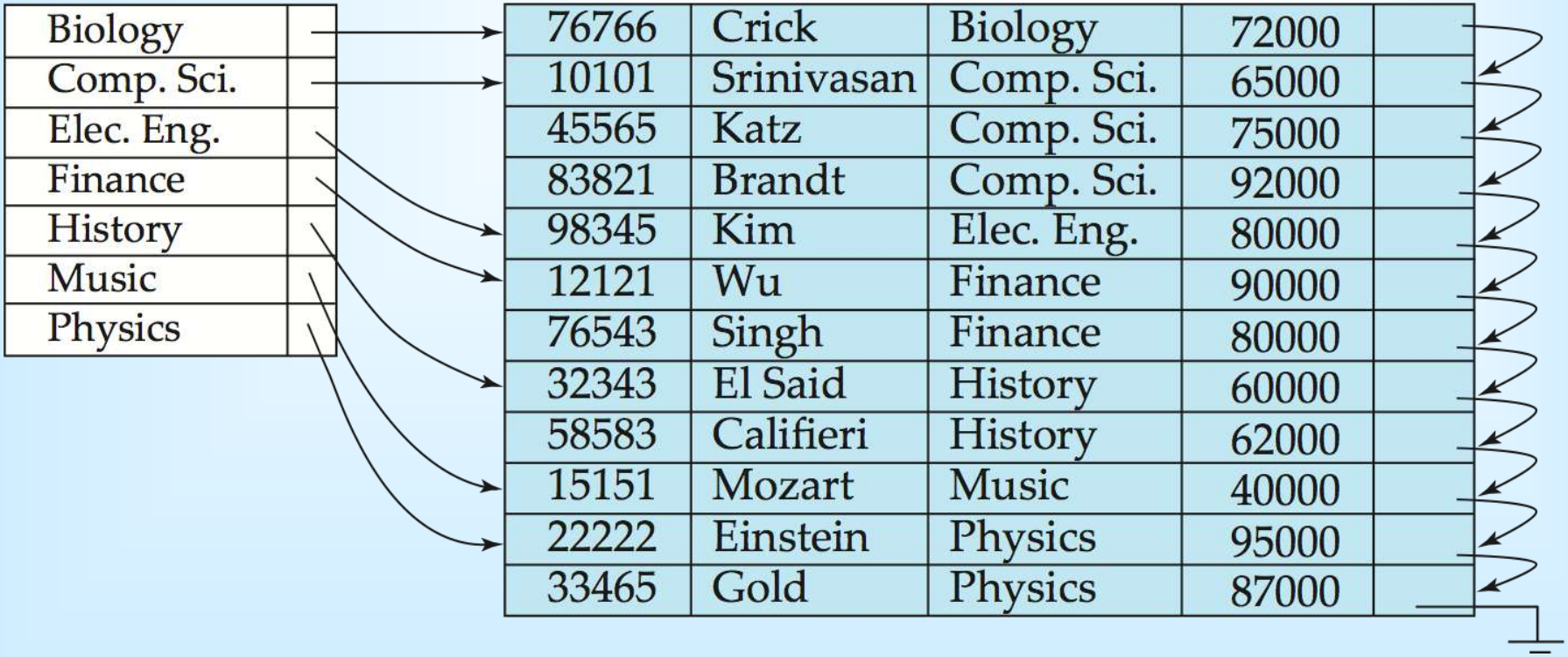
- **Dense index** — Index record appears for every search-key value in the file.
- E.g. index on *ID* attribute of *instructor* relation



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

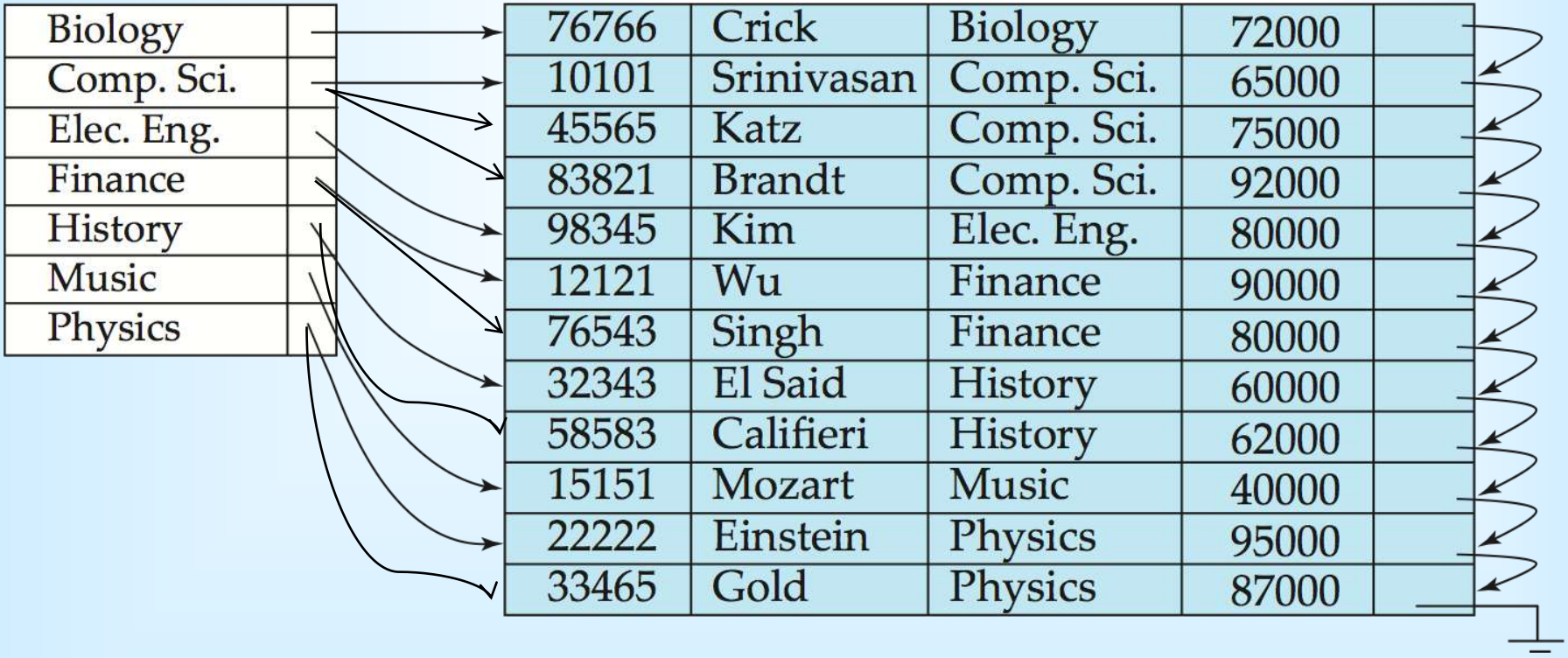
Dense Index Files (Cont.)

- In a **dense clustering index**, the index record contains the search-key value and a pointer to the first data record with that search-key value.
- The rest of the records with the same search-key value would be stored sequentially after the first record.
- Dense index on *dept_name*, with *instructor* file sorted on *dept_name*



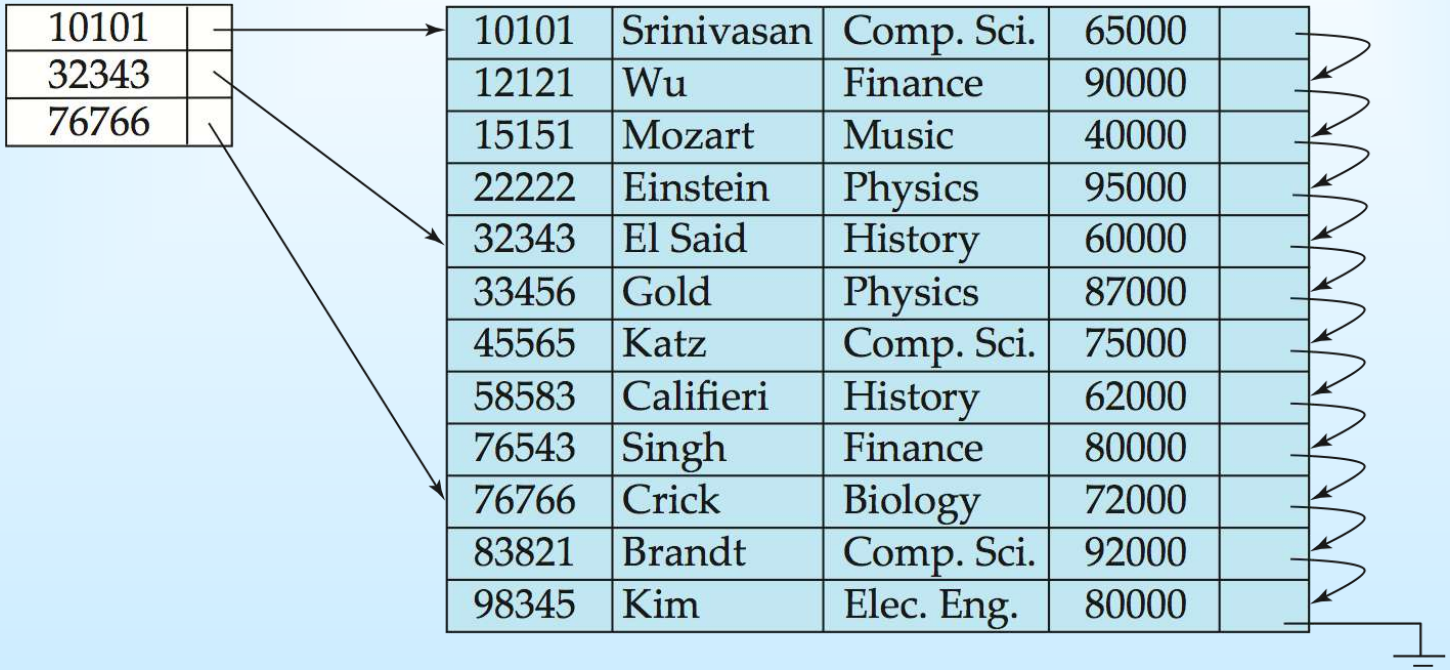
Dense Index Files (Cont.)

- In a **dense non clustering index**, the index must store a list of pointers to all records with the same search-key value.
- Dense index on *dept_name*, with *instructor* file sorted on *dept_name*

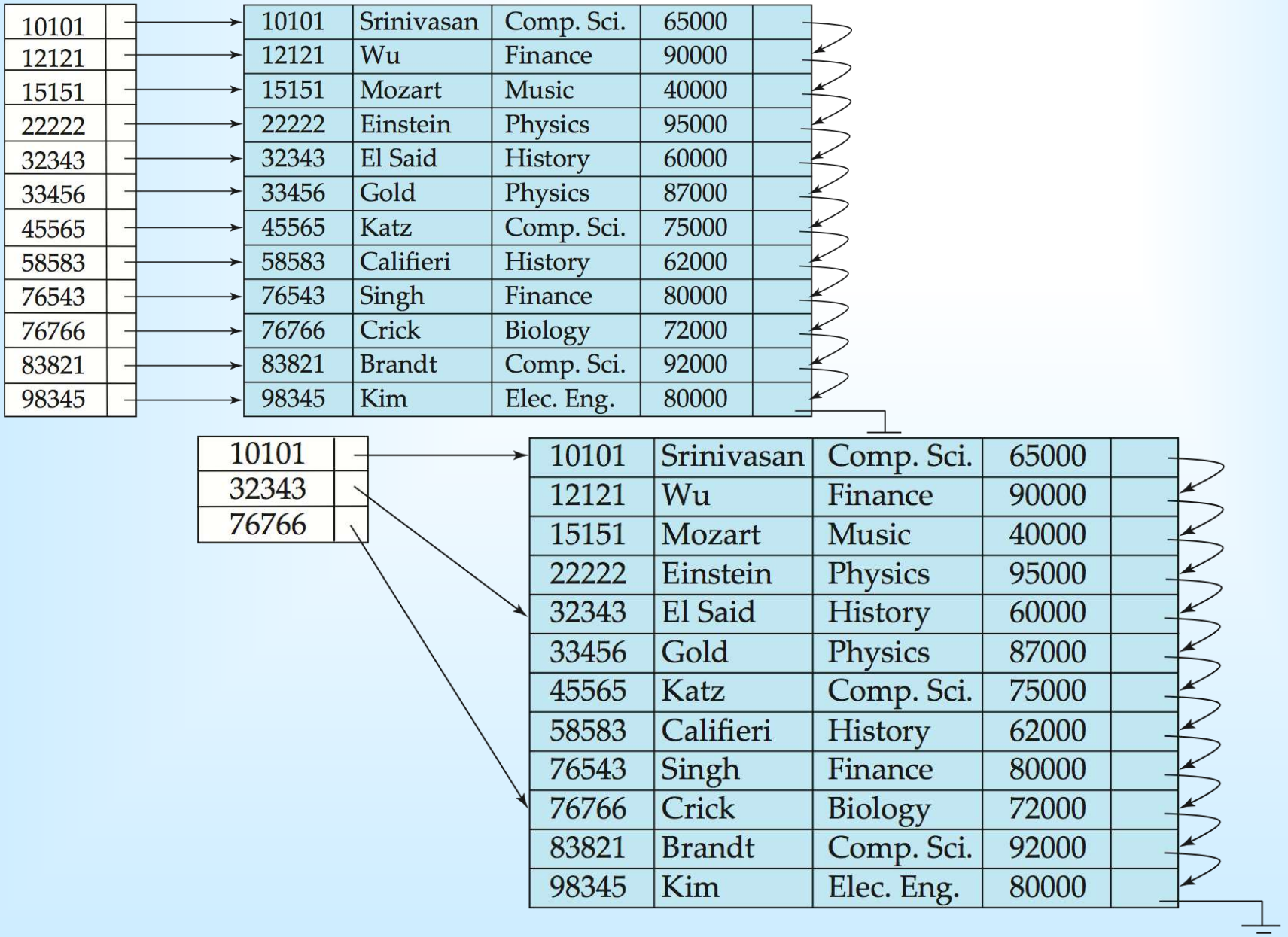


Sparse Index Files

- **Sparse Index**: contains index records for only some search-key values.
  - 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 <= *K*
  - Search file sequentially starting at the record to which the index record points

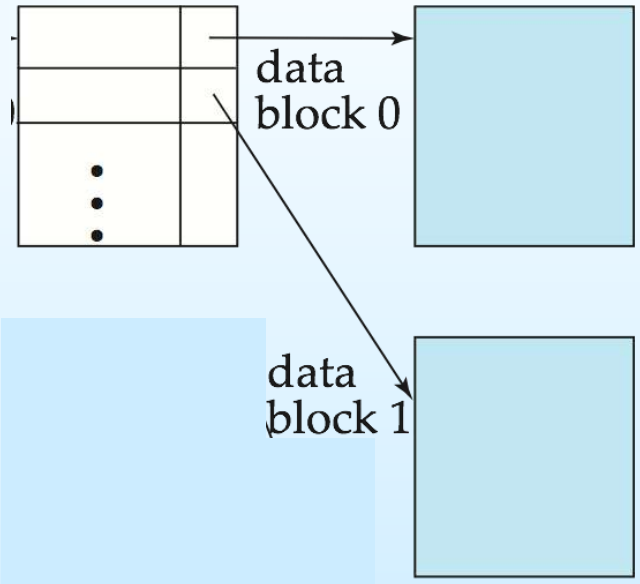


Dense and Sparse Index

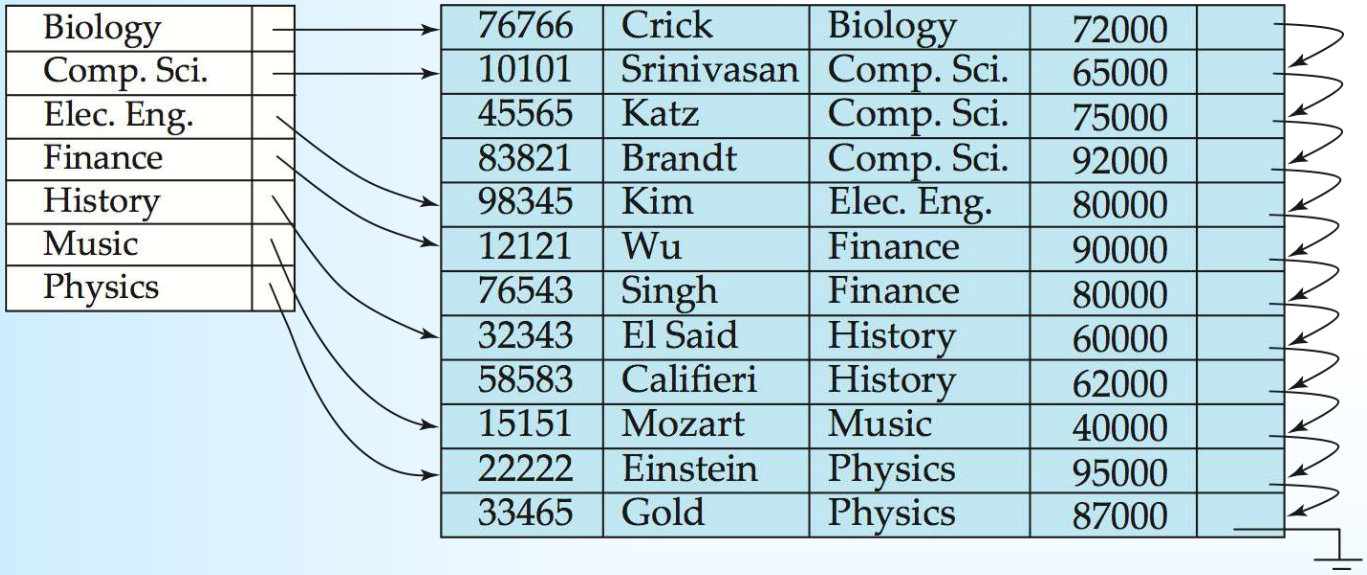


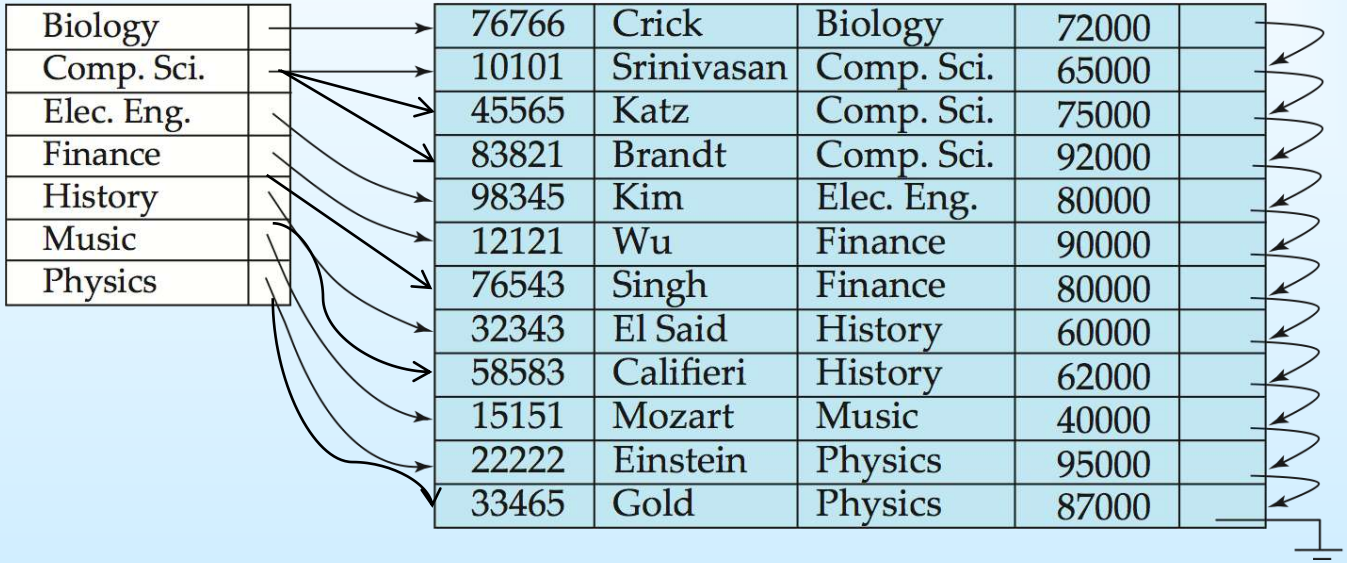
Sparse Index Files (Cont.)

- Compared to dense indices:
  - Less space and less maintenance overhead for insertions and deletions.
  - Generally slower than dense index for locating records.
- **Good tradeoff**: sparse index with an index entry for every block in file, corresponding to least search-key value in the block.

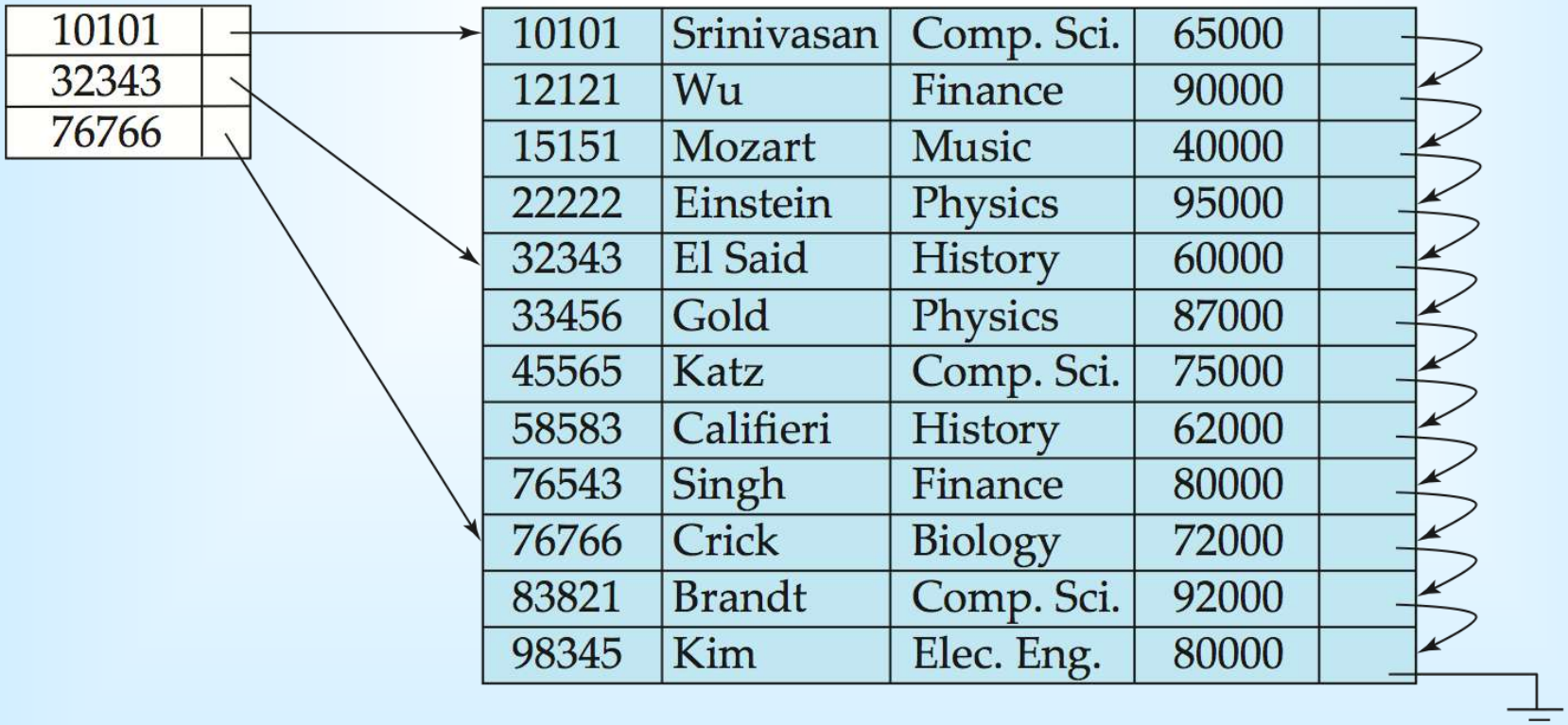


Index Update





Index Update



Index Update: Insertion

- Single-level index insertion:
  - Perform a lookup using the search-key value appearing in the record to be inserted.
  - **Dense indices –**
    - ❖ if the search-key value does not appear in the index, insert it at appropriate position.
    - ❖ Otherwise the following actions are taken:
      a. If the index record stores pointers to all records with the same search-key value, the system adds a pointer to the new record to the index record.
      b. Otherwise, if the index record stores a pointer to only the first record with the search-key value, the system then places the record being inserted after the last records with the same search-key values.
  - **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. In this case, the first search-key value appearing in the new block is inserted into the index.

Index Update: Deletion

- To delete a record, the system first looks up the record to be deleted.
- Single-level index deletion:
  - **Dense indices –**
    - ❖If deleted record was the only record in the file with its particular search-key value, the search-key is deleted from the index.
    - ❖Otherwise the following actions are taken:
      a. If the index record stores pointers to all records with the same search-key value, the system deletes the pointer to the deleted record from the index record.
      b. Otherwise, the index record stores a pointer to only the first record with the search-key value. In this case, if the deleted record was the first record with the search-key value, the system updates the index record to point to the next record.

Index Update: Deletion

- **Sparse indices –**
  - ❖ If the index does not contain an index record with the search-key value of the deleted record, nothing needs to be done to the index.
  - ❖ Otherwise the system takes the following actions:

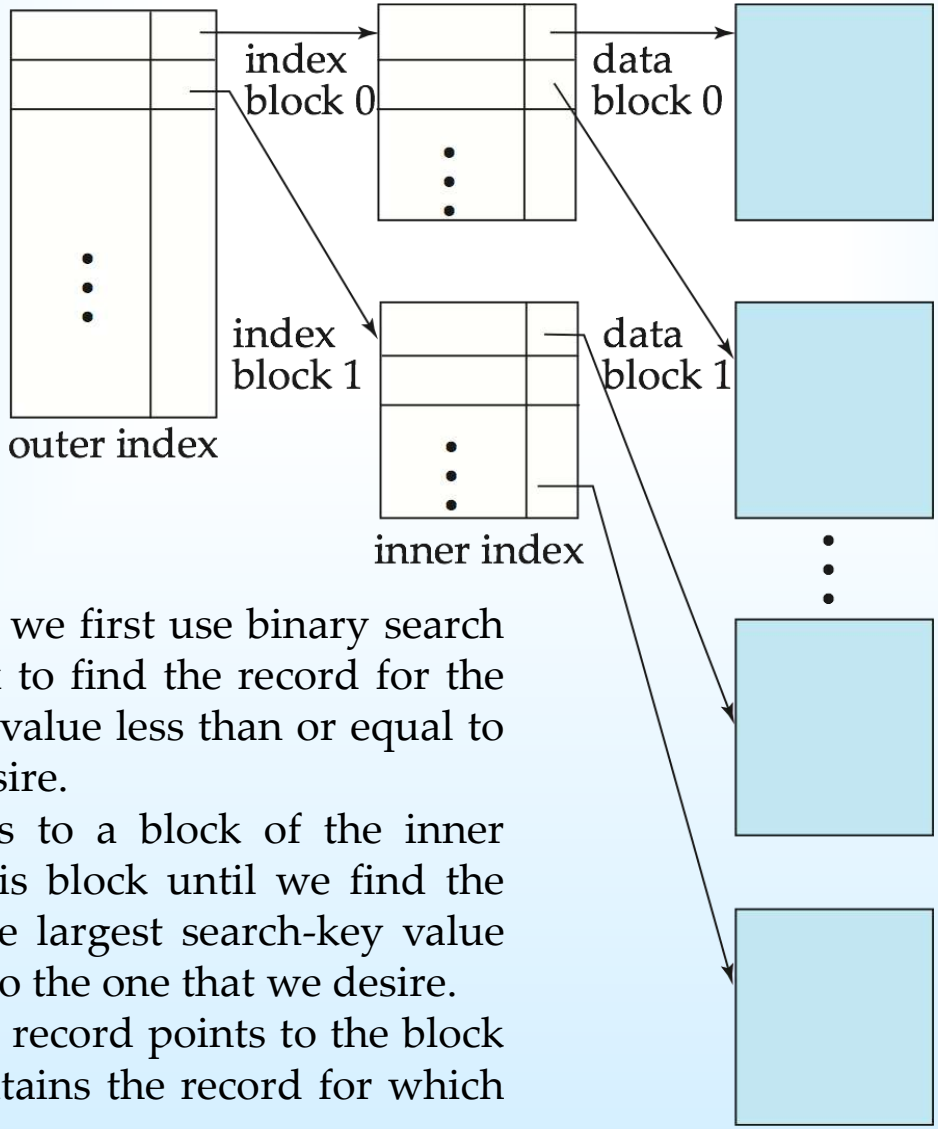
    a. If the deleted record was the only record with its search key, the system replaces the corresponding index record with an index record for the next search-key value (in search-key order). If the next search-key value already has an index entry, the entry is deleted instead of being replaced.

    b. Otherwise, if the index record for the search-key value points to the record being deleted, the system updates the index record to point to the next record with the same search-key value.

Multilevel Index

- If an index is small enough to be kept entirely in main memory, the search time to find an entry is low.
- However, if the index is so large that it can not be kept in main memory, index blocks must be fetched from disk when required.
- Solution: treat primary index kept on disk as a sequential file and construct a sparse index on it.
  - inner index – the primary index file
  - outer index – a sparse index of primary index
- If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.
- Indices at all levels must be updated on insertion or deletion from the file.

Multilevel Index (Cont.)

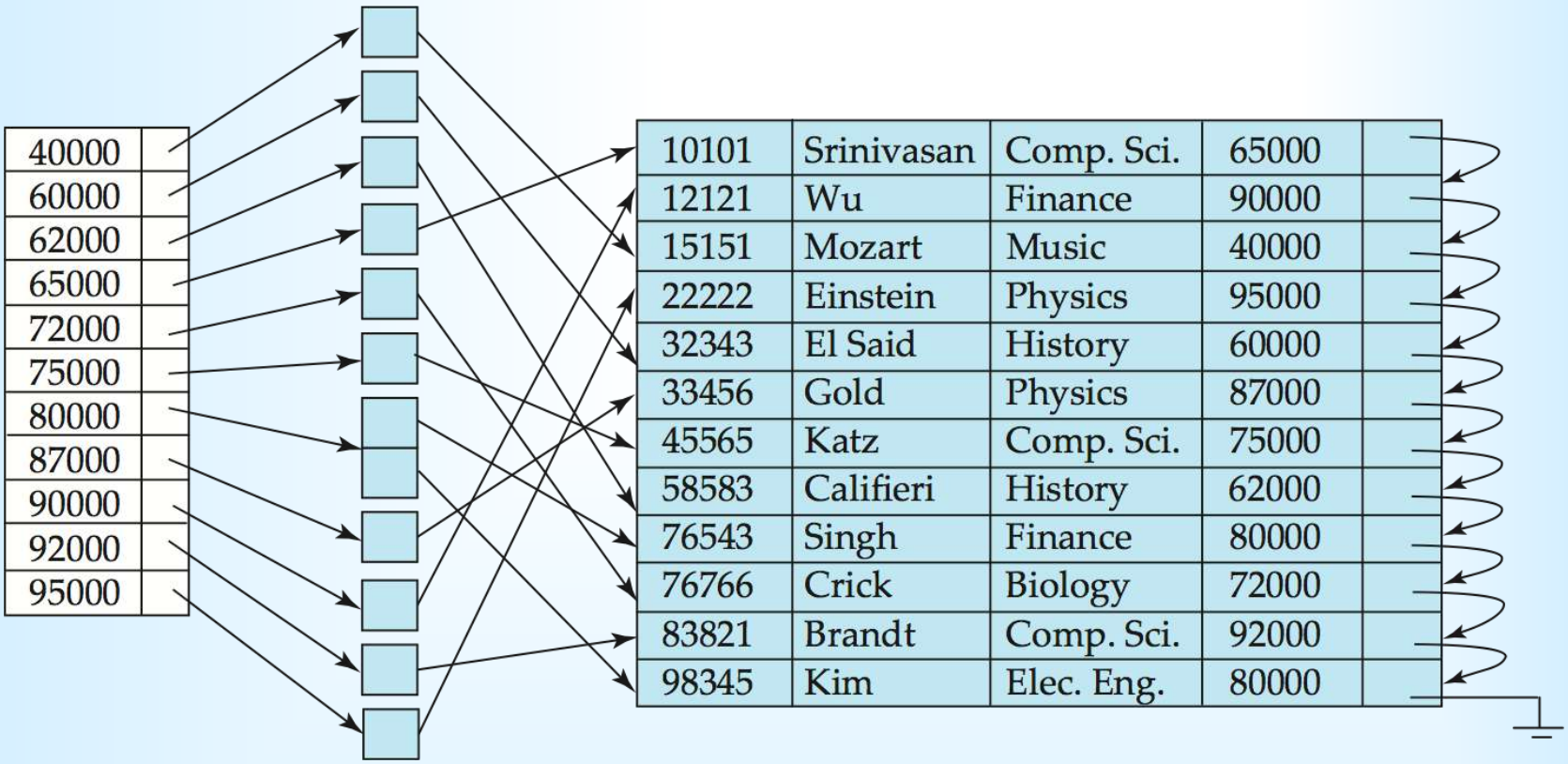


- To locate a record, we first use binary search on the outer index to find the record for the largest search-key value less than or equal to the one that we desire.
- The pointer points to a block of the inner index. We scan this block until we find the record that has the largest search-key value less than or equal to the one that we desire.
- The pointer in this record points to the block of the file that contains the record for which we are looking.

Secondary Indices

- Whose search key value specifies an order other than sequential order.
- We can have a secondary index must be dense, with an index record for each search-key value and a pointer to every record in the file.
- Index record points to a bucket that contains pointers to all the actual records with that particular search-key value.
- Frequently, one wants to find all the records whose values in a certain field (which is not the search-key of the primary index) satisfy some condition.
  - Example 1: In the *instructor* relation stored sequentially by ID, we may want to find all instructors in a particular department
  - Example 2: as above, but where we want to find all instructors with a specified salary or with salary in a specified range of values

Secondary Indices Example



Secondary index on *salary* field of *instructor*

Primary and Secondary Indices

- Indices offer substantial benefits when searching for records.
- BUT: Updating indices imposes overhead on database modification --when a file is modified, every index on the file must be updated,
- Secondary indices have to be dense, while primary indices have either dense or sparse.
- Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive
  - Each record access may fetch a new block from disk
  - Block fetch requires about 5 to 10 milliseconds, versus about 100 nanoseconds for memory access

B^+-Tree Index Files

B^+-tree indices are an alternative to indexed-sequential files.

- Disadvantage of indexed-sequential files
  - Performance degrades as file grows, since many overflow blocks get created in case of insertion.
  - Periodic reorganization of entire file is required.
- Advantage of B^+-tree index files:
  - Automatically reorganizes itself with small, local changes, in the case 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 Index Files (Cont.)

A B^+-tree is a rooted tree satisfying the following properties:

- All paths from the root of the tree to the leaf of the tree are of the same length (balance tree).
- Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children.
- 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 Node Structure

- Typical node

P_1	K_1	P_2	...	P_{n-1}	K_{n-1}	P_n

  - K_i are the search-key values. It contains up to n-1 search key values K_1 , K_2 , K_3 , . . . , $K_{n–1}$
  - It contains up to n pointers P_1, P_2, … , P_n .
  - P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).
- The search-keys in a node are ordered

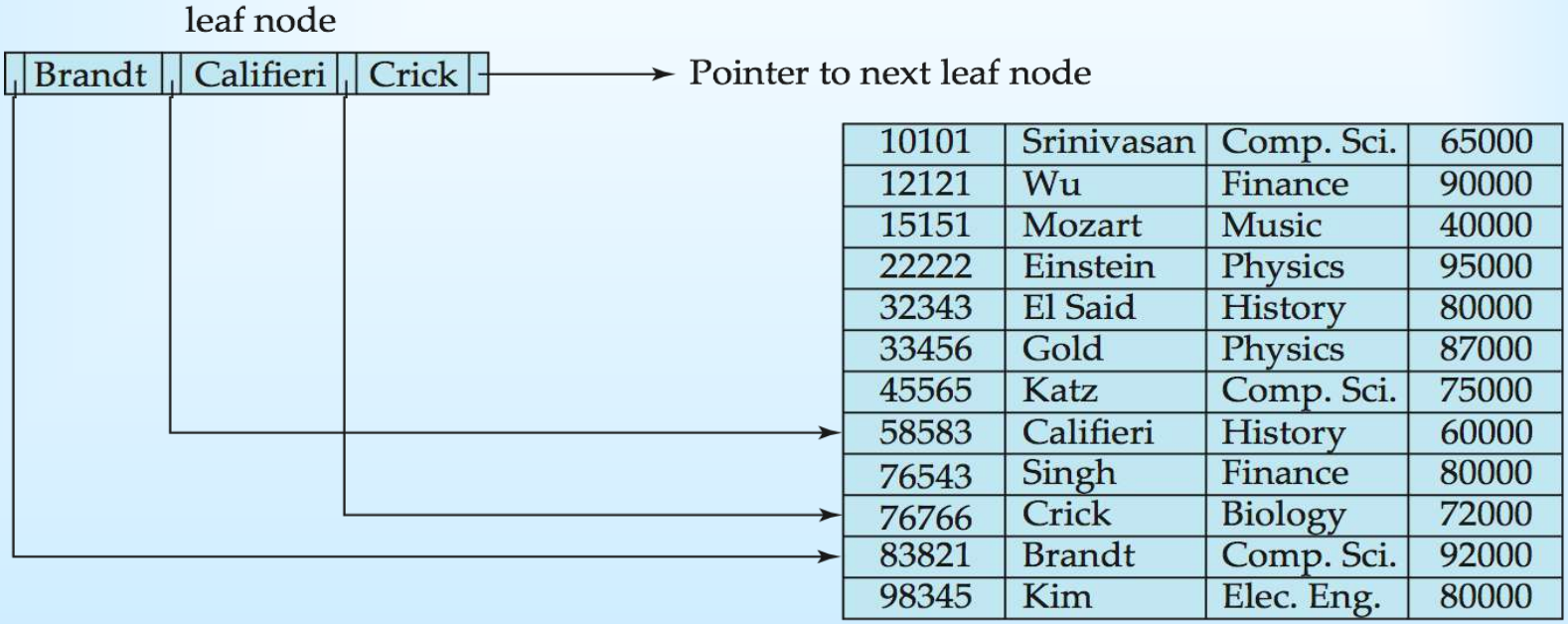
$$K_1 < K_2 < K_3 < . . . < K_{n–1}$$

(Initially assume no duplicate keys, address duplicates later)

Leaf Nodes in B+-Trees

Properties of a leaf node:

- For $i = 1, 2, \ldots, n-1$, pointer P_i points to a file record with search-key value K_i,
- 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.
- The B+-tree index is used as a dense index that every search-key value must appear in some leaf node.



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 *n* pointers:
  - Pointer P_1 points to the subtree that contains those search key values less than K_1
  - For $2 \leq i \leq n - 1$, pointer P_i points to the subtree that contains those search key values less than K_i *and* greater than or equal to K_{i-1}.
  - P_n points to the subtree that contains those key values greater than or equal to K_{n-1}.

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

  - Non leaf nodes are also referred to as internal nodes.

Example of B^+-Tree

Names in alphabetic order:

Brandt

Caiifieri

Crick

Einstein

El Said

Gold

Katz

Kim

Mozart

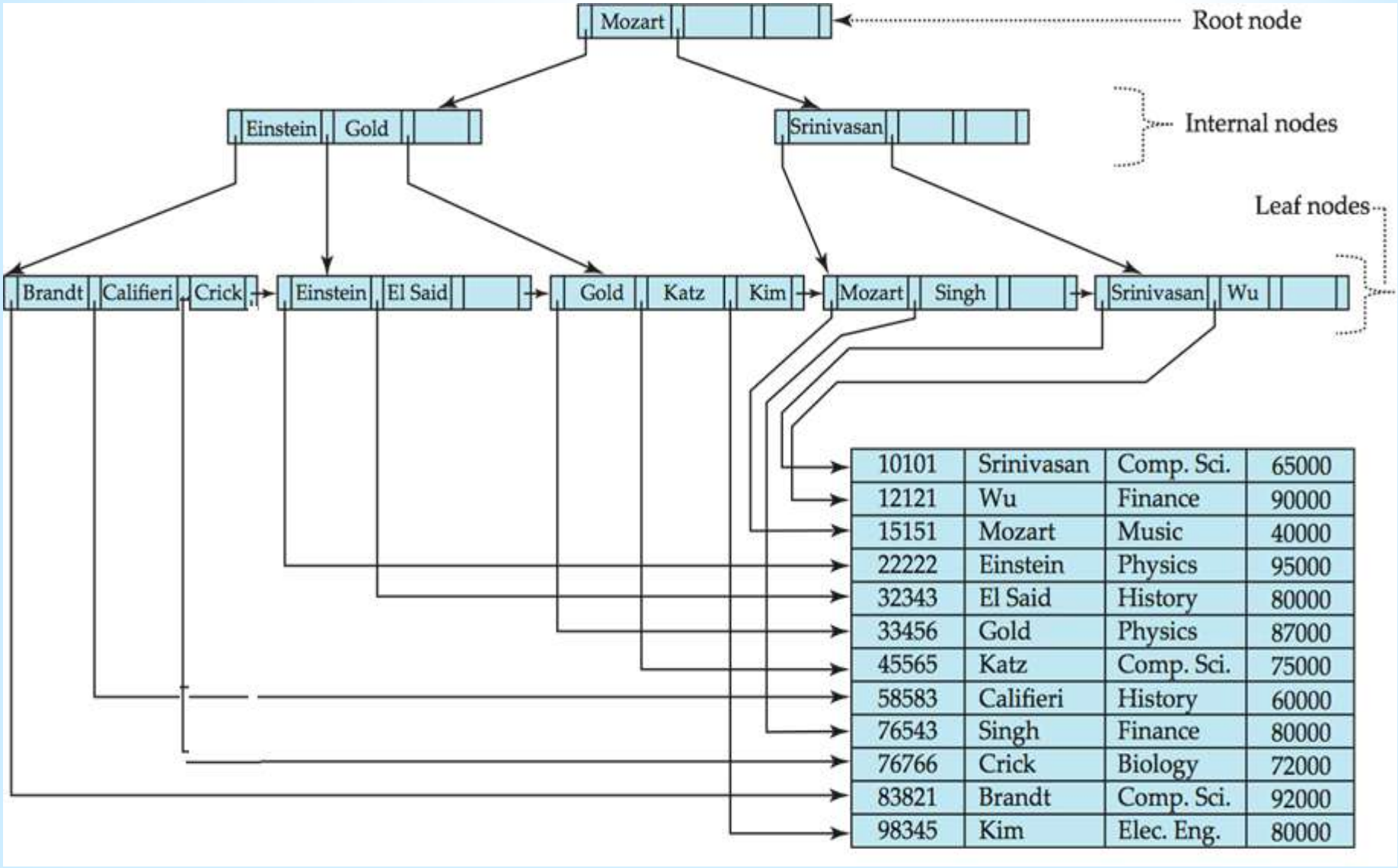
Singh

Shrinivasan

Wu

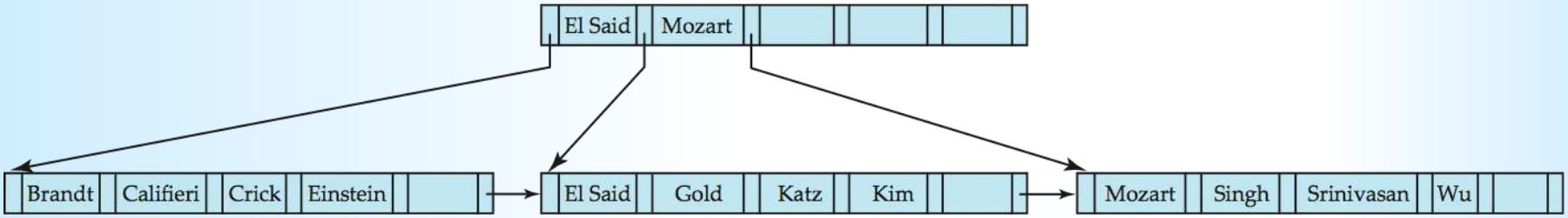
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000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

Example of B+-Tree



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

Example of B+-tree



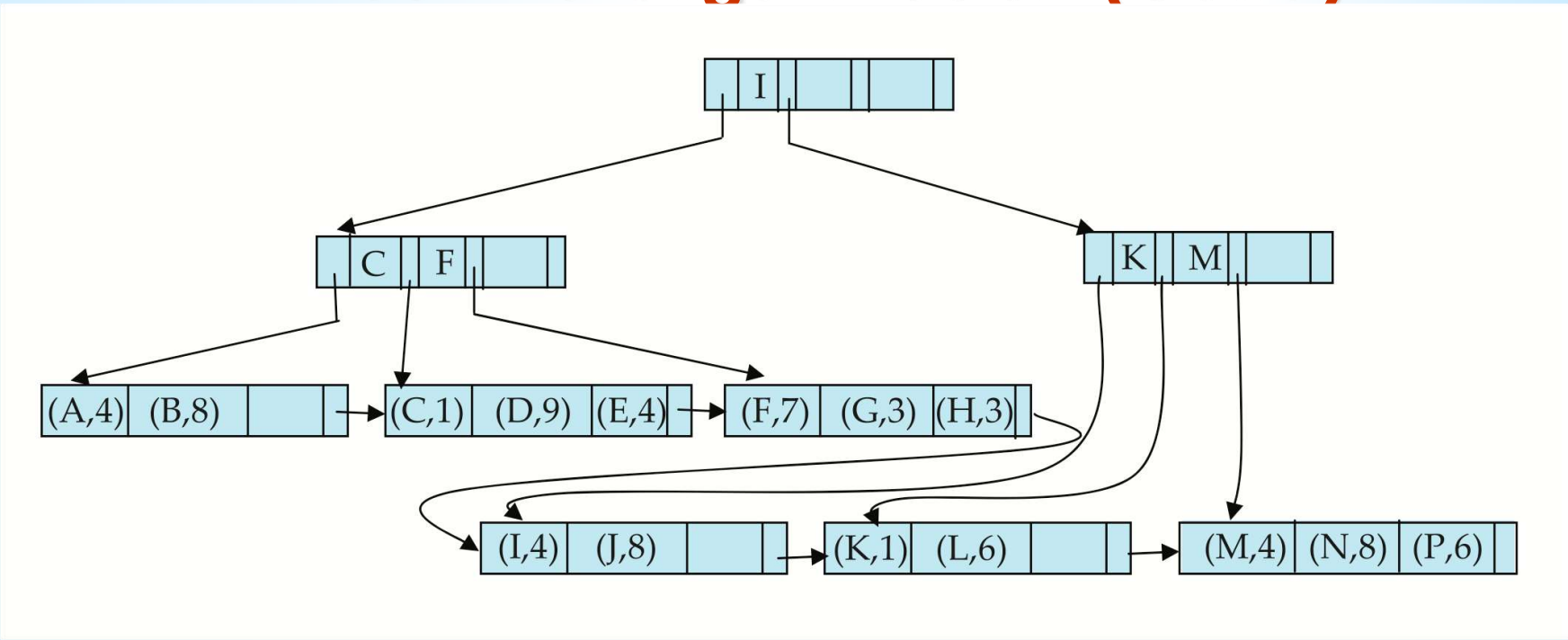
B+-tree for *instructor* file (n = 6)

- 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.

B^+-Tree File Organization

- Index sequential file degradation problem is solved by using B^+-Tree indices.
- Data file degradation problem is solved by using B^+-Tree File Organization.
- The leaf nodes in a B^+-tree file organization store records, instead of pointers.
- Leaf nodes are still required to be half full
  - Since records are larger than pointers, the maximum number of records that can be stored in a leaf node is less than the number of pointers in a nonleaf node.
- Insertion and deletion are handled in the same way as insertion and deletion of entries in a B^+-tree index.

B+-Tree File Organization (Cont.)



Example of B+-tree File Organization

- Good space utilization important since records use more space than pointers.
- To improve space utilization, involve more sibling nodes in redistribution during splits and merges
  - Involving 2 siblings in redistribution (to avoid split / merge where possible) results in each node having at least $\lfloor 2n/3 \rfloor$ entries

B-Tree Index Files

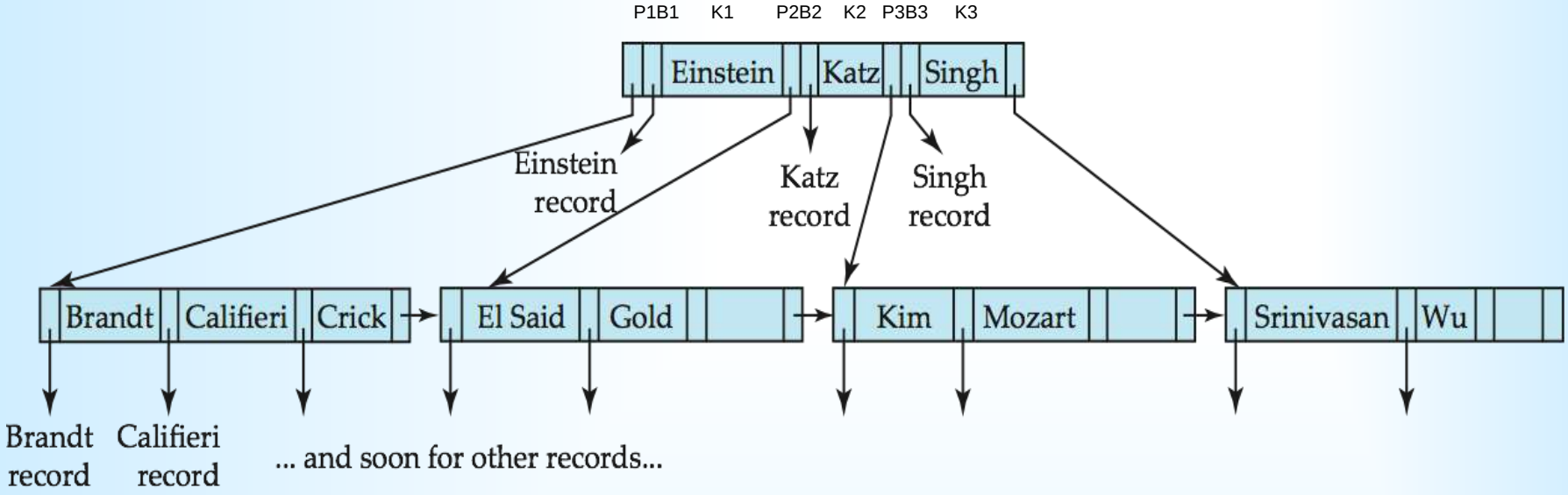
- Similar to B+-tree, but B-tree allows search-key values to appear only once; eliminates redundant storage of search keys.
- Search keys in nonleaf nodes appear nowhere else in the B-tree; an additional pointer field for each search key in a nonleaf node must be included.
- Generalized B-tree leaf node (a) and nonleaf node (b).

(a)

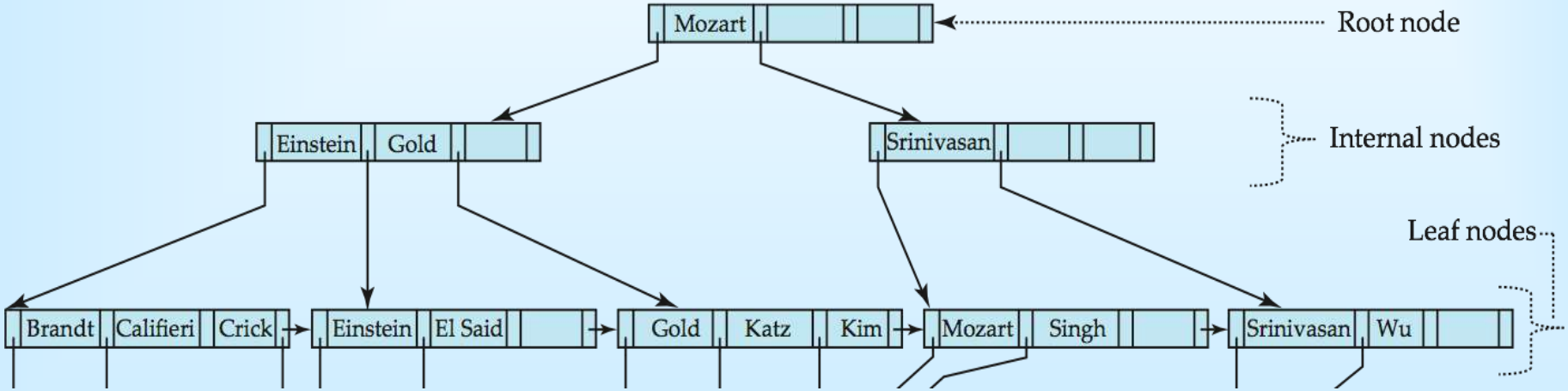
(b)

- Nonleaf node – pointers Bi are the bucket or file record pointers.

B-Tree Index File Example



B-tree (above) and B+-tree (below) on same data



B-Tree Index Files (Cont.)

- Advantages of B-Tree indices:
  - May use less tree nodes than a corresponding B^+-Tree.
  - Sometimes possible to find search-key value before reaching leaf node.
  - Avoid redundant storage of search key values.
- Disadvantages of B-Tree indices:
  - Only small fraction of all search-key values are found early.
  - Fewer search keys appear in a non-leaf B-tree node, compared to B^+-trees, implies that a B-tree has a smaller fan out and therefore may have depth greater than that of the corresponding B+-tree.
  - Insertion and deletion more complicated than in B^+-Trees
  - Implementation is harder than B^+-Trees.
- Typically, advantages of B-Trees do not out weigh disadvantages.

Multiple-Key Access(Using Mupltiple Single Key Indices)

- Use multiple indices or use an index built on a multi attribute search key for certain types of queries.
- Example: Two Indices *dept_name* and *salary*

select *ID*

from *instructor*

where *dept_name* = “Finance”

and *salary* = 80000

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

- Possible strategies for processing query using two indices on single attributes:

1. Use index on *dept_name* to find instructors with department name Finance; test *salary = 80000*
2. Use index on *salary* to find instructors with a salary of $80000; test *dept_name* = “Finance”.
3. Use *dept_name* index to find pointers to all records pertaining to the “Finance” department. Similarly use index on *salary*. Take intersection of both sets of pointers obtained.

Multiple-Key Access (Indices on Multiple Keys)

- **Composite search keys** are search keys containing more than one attribute
  - E.g. (*dept_name, salary*)
- Lexicographic ordering: $(a_1, a_2) < (b_1, b_2)$ if either
  - $a_1 < b_1$, or
  - $a_1 = b_1$ and $a_2 < b_2$

Indices on Multiple Attributes

Suppose we have an index on combined search-key (*dept_name, salary*).

- With the **where** clause
  where *dept_name* = “Finance” **and** *salary* = 80000
  the index on (*dept_name, salary*) can be used to fetch only records that satisfy both conditions.
  - Using separate indices in less efficient — we may fetch many records (or pointers) that satisfy only one of the conditions.
- Can also efficiently handle
  where *dept_name* = “Finance” **and** *salary* < 80000
- But cannot efficiently handle
  where *dept_name* < “Finance” **and** *balance* = 80000
  - May fetch many records that satisfy the first but not the second condition

Index Definition in SQL

- Create an index

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

  E.g.: **create index** *dept_index* **on** *instructor (dept_name)*

- Use **create unique index** to indirectly specify and enforce the condition that the search key is a candidate key.
  - Not really 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.

Hashing

Static Hashing

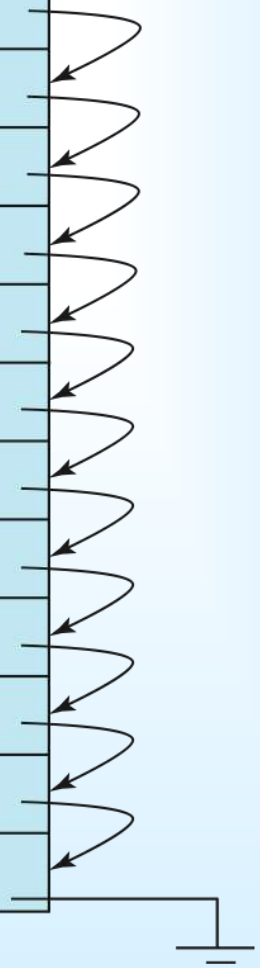
- A **bucket** is a unit of storage containing one or more records (a bucket is typically a disk block).
- In a **hash file organization** we obtain the address of the bucket containing desired record directly by computing a **hash function** on its search-key value**.**
- Let ***K*** denote the set of all search-key values, ***B*** denote the set of all bucket addresses
- A 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 records for access, insertion as well as deletion.
- Records with different search-key values may be mapped to the same bucket; thus entire bucket has to be searched sequentially to locate a record.

Hash Functions

- Worst hash function maps all search-key values to the same bucket; Such a function is undesirable because all records have to be kept in the same bucket.
- An ideal hash function is **uniform**, i.e., each bucket is assigned the same number of search-key values from the set of *all* possible values search key values.
- Ideal hash function is **random**, so each bucket will have the same number of records assigned to it irrespective of the *actual distribution* of search-key values in the file.

Instructor Relation

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	



Example of Hash File Organization

- Let us choose a hash function for the *instructor* file using the search key *dept_name*.
- Assume that we decide to have 26 buckets, and we define a hash function that maps names beginning with the *i*th letter of the alphabet to the *i*th bucket.
- It fails to provide a uniform distribution, since we expect more names to begin with such letters as B and R than Q and X.
- Now suppose that we want a hash function on the search key *salary*. Suppose that the minimum salary is $30,000 and the maximum salary is $130,000, and we use a hash function that divides the values into 10 ranges, $30,000–$40,000,$40,001–$50,000 and so on.
- The distribution of search-key values is uniform, but is not random.
- Records with salaries between $60,001 and $70,000 are far more common than are records with salaries between $30,001 and $40,000.
- As a result, the distribution of records is not uniform—some buckets receive more records than others do.

Example of Hash File Organization

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

- Typical hash functions perform computation on the internal binary representation of the search-key.

  For example, for a string search-key, the binary representations of all the characters in the string could be added and the sum modulo the number of buckets could be returned. .

- The binary representation of the *i*th character is assumed to be the integer *i.*
- There are 8 buckets(0 to 7),
- The hash function returns the sum of the binary representations of the characters modulo 8
  - E.g. h(Music) = 1 h(History) = 2
    h(Physics) = 3 h(Elec. Eng.) = 3

Example of Hash File Organization

- M = 01001101
- u = 01110101
- s = 01110011
- i = 01101001
- c = 01100011
- **Binary Addition = 01000000001**
- **Decimal Value = 513**
- **513 MOD 8 = 1**
- **Bucket 1 assign Music.**
- **ASCII**

Example of Hash File 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

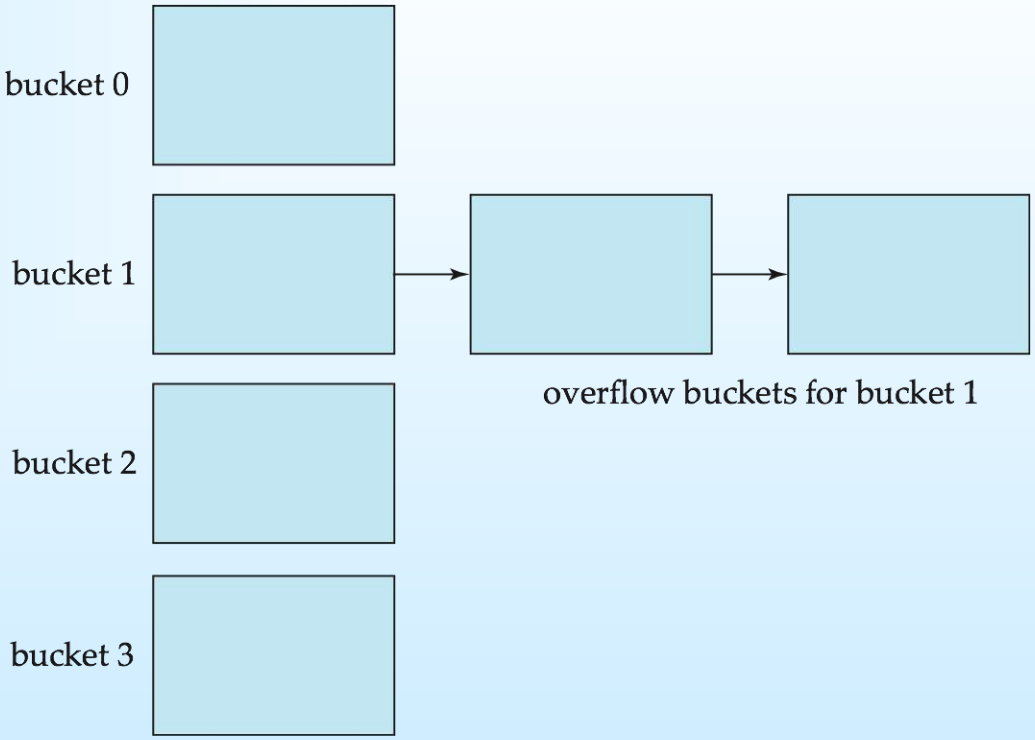
Hash file organization of *instructor* file, using *dept_name* as key (see previous slide for details).

Handling of Bucket Overflows

- When a record is inserted, the bucket to which it is mapped has space to store the record. If the bucket does not have enough space, a **bucket overflow** is said to occur.
- Bucket overflow can occur for several reasons:
  - **Insufficient buckets**: The number of buckets, which we denote n_B, must be chosen such that $n_B > n_r / f_r$, where n_r denotes the total number of records that will be stored and f_r denotes the number of records that will fit in a bucket.
  - **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*.**

Handling of Bucket Overflows (Cont.)

- **Overflow chaining** – the overflow buckets of a given bucket are chained together in a linked list.
- Above scheme is called **closed hashing.**
  - An alternative, called **open hashing**, which does not use overflow buckets-the set of bucket is fixed, is not suitable for database applications.



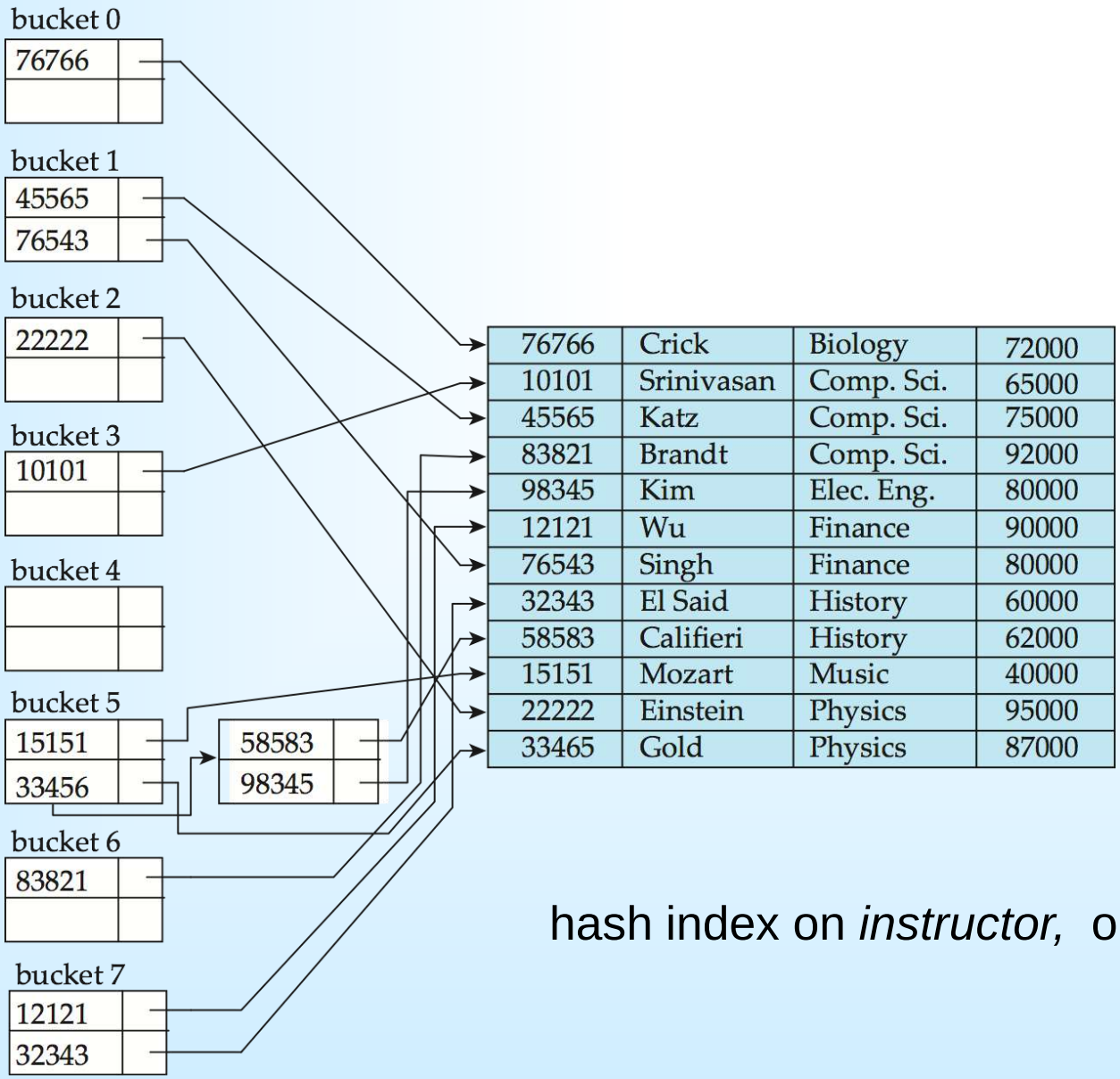
Closed and open hashing

Closed Hashing	Open hashing
the overflow buckets of a given bucket are chained together in a linked list.	does not use overflow buckets-the set of bucket is fixed.
Closed hashing is preferable for database systems.	Open hashing has been used to construct symbol tables for compilers and assemblers.
In a database system, it is important to be able to handle deletion as well as insertion.	Usually, compilers and assemblers perform only lookup and insertion operations on their symbol tables.

Hash Indices

- Hashing can be used not only for file organization, but also for index-structure creation.
- A **hash index** organizes the search keys, with their associated record pointers, into a hash file structure.
- The hash function in the figure computes the sum of the digits of the *ID* modulo 8.
- The hash index has eight buckets, each of size 2. One of the buckets has three keys mapped to it, so it has an overflow bucket.
- Strictly speaking, hash indices are always secondary indices
  - if the file itself is organized using hashing, a separate primary hash index on it using the same search-key is unnecessary.
  - However, we use the term hash index to refer to both secondary index structures and hash organized files.

Example of Hash Index



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

Deficiencies of Static Hashing

- We must choose the hash function when we implement the system, and it cannot be changed easily thereafter if the file being indexed grows or shrinks
- Most databases grow larger over time. If we are to use static hashing for such a database, we have three classes of options:
  - Choose a hash function based on the current file size. This option will result in performance degradation as the database grows.
  - Choose a hash function based on the anticipated size of the file at some point in the future. But, a significant amount of space may be wasted initially.
  - Periodically reorganize the hash structure in response to file growth. Such a reorganization involves choosing a new hash function, recomputing the hash function on every record in the file, and generating new bucket assignments. This reorganization is a massive, time-consuming operation.
- Allow the hash function to be modified dynamically to accommodate the growth or shrinkage of the database known as **dynamic hashing.**

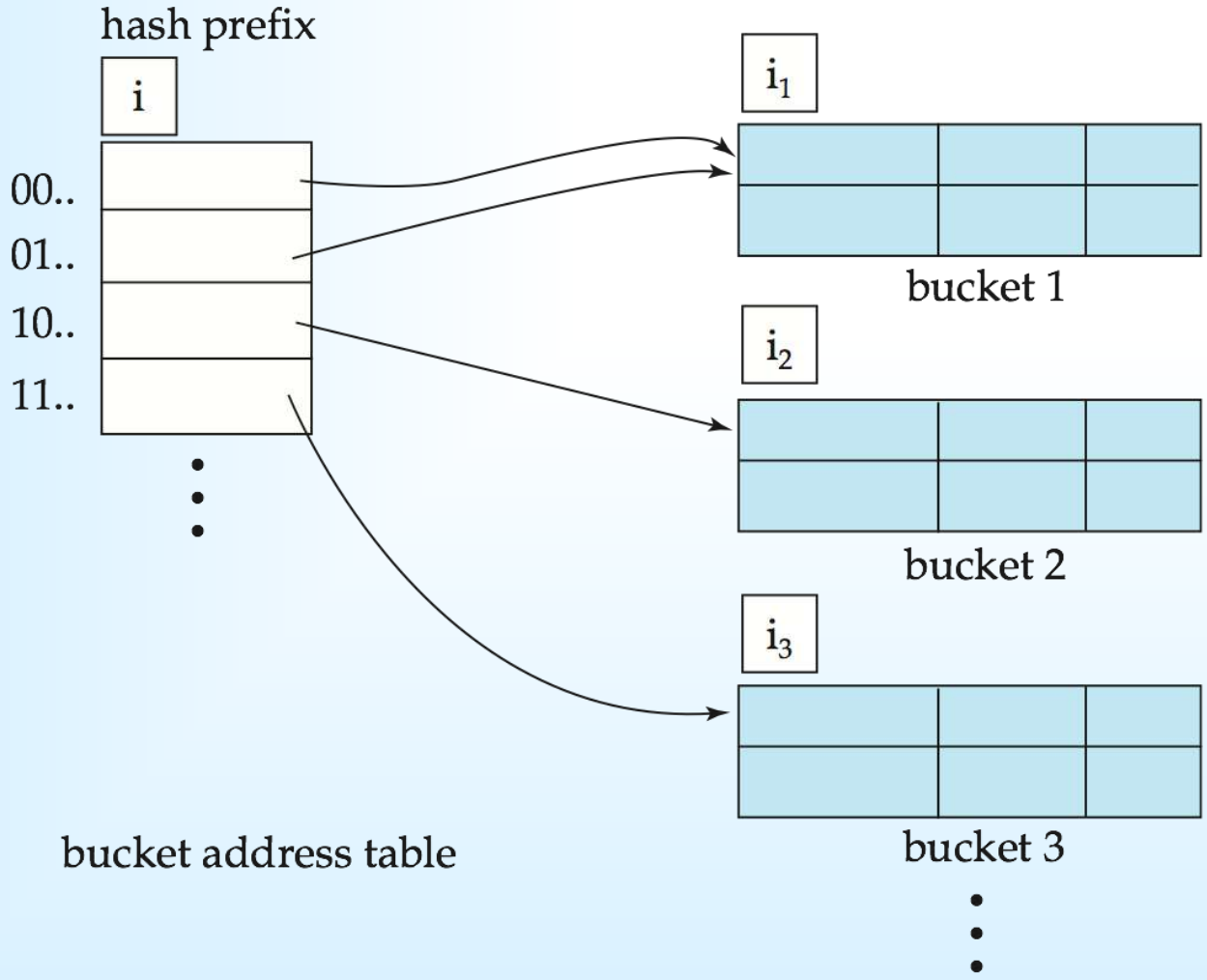
Dynamic Hashing

- Good for database that grows and shrinks in size.
- Allows the hash function to be modified dynamically
- **Extendable hashing** – one form of dynamic hashing
- Extendable hashing copes with changes in database size by splitting and combining buckets as the database grows and shrinks.
  - Hash function generates values over a large range — typically *b*-bit integers, with *b* = 32.
  - We do not create a bucket for each hash value. Indeed, $2^{32.}$ is over 4 billion,
  - Instead, we create buckets on demand, as records are inserted into the file.
  - We do not use the entire b bits of the hash value initially.

Dynamic Hashing

- At any point, we use i bits, where $0 \leq i \leq b$.
- Let the length of the prefix be *i* bits, $0 \leq i \leq 32$.
- Bucket address table is used.
- Bucket address table size = 2^{i}. Initially $i = 0$
- Value of *i* grows and shrinks as the size of the database grows and shrinks.
- Multiple entries in the bucket address table may point to a bucket.
- Thus, actual number of buckets is $\leq 2^{i}$
- The number of buckets also changes dynamically due to combining and splitting of buckets.

General Extendable Hash Structure



In this structure, $i_2 = i_3 = i$, whereas $i_1 = i - 1$

Queries on Extendable Hash Structure

- Each bucket j stores a value i_j
  - All the entries that point to the same bucket have the same values on the first i_j bits.
- To locate the bucket containing search-key K_j:
  1. Compute $h(K_j) = X$
  2. Use the first i high order bits of X as a displacement into bucket address table, and follow the pointer to appropriate bucket
- To insert a record with search-key value K_j
  - follow same procedure as look-up and locate the bucket, say j.
  - If there is room in the bucket j insert record in the bucket.
  - Else the bucket must be split and insertion re-attempted (next slide.)
    - Overflow buckets used instead in some cases.

Insertion in Extendable Hash Structure (Cont)

To split a bucket *j* when inserting record with search-key value K_j, the system must first determine from the hash value whether it needs to increase the number of bits that it uses:

- If $i > i_j$ (more than one pointer in the bucket address table points to bucket *j*)
  - Split bucket *j* without increasing the size of the bucket address table.
  - allocate a new bucket *z*, and set i_j and i_z to *i*.
  - Next, the system needs to adjust the entries in the bucket address table that previously pointed to bucket *j.*
  - Leaves the first half of the entries as they were pointing to bucket *j*.
  - make the second half of the bucket address table entries pointing *z*
  - remove and reinsert each record in bucket *j.*

Insertion in Extendable Hash Structure (Cont)

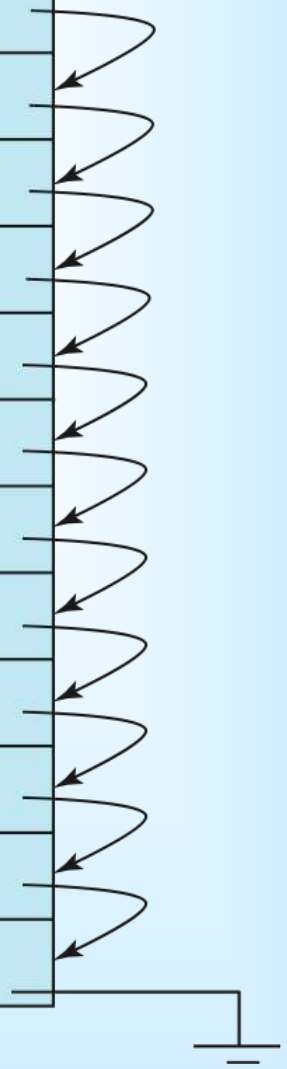
- If $i = i_j$ (only one pointer in the bucket address table points to bucket j)
  - Increase the size of bucket address table.
  - increment value of i by 1 and double the size of the bucket address table.
  - replace each entry in the table by two entries that point to the same bucket.
  - Now two entries in the bucket address table points to bucket j.
  - System allocates a new bucket z and set the second entry to point to the new bucket z.
  - recompute new bucket address table entry for K_j
- When inserting a value, if the bucket is full after several splits (that is, i reaches some limit b) create an overflow bucket instead of splitting bucket address table entry further.

Deletion in Extendable Hash Structure

- To delete a key value,
  - Locate it in its bucket and remove it.
  - The bucket itself can be removed if it becomes empty (with appropriate updates to the bucket address table).
  - Combining of buckets can be done (can coalesce only with a "*buddy*" bucket having same value of i_j and same i_j –1 prefix, if it is present)
  - Decreasing bucket address table size is also possible
    - Note: decreasing bucket address table size is an expensive operation and should be done only if number of buckets becomes much smaller than the size of the table

Instructor Relation

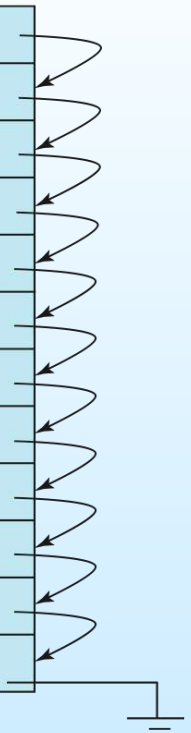
10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp.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



Use of Extendable Hash Structure: Example

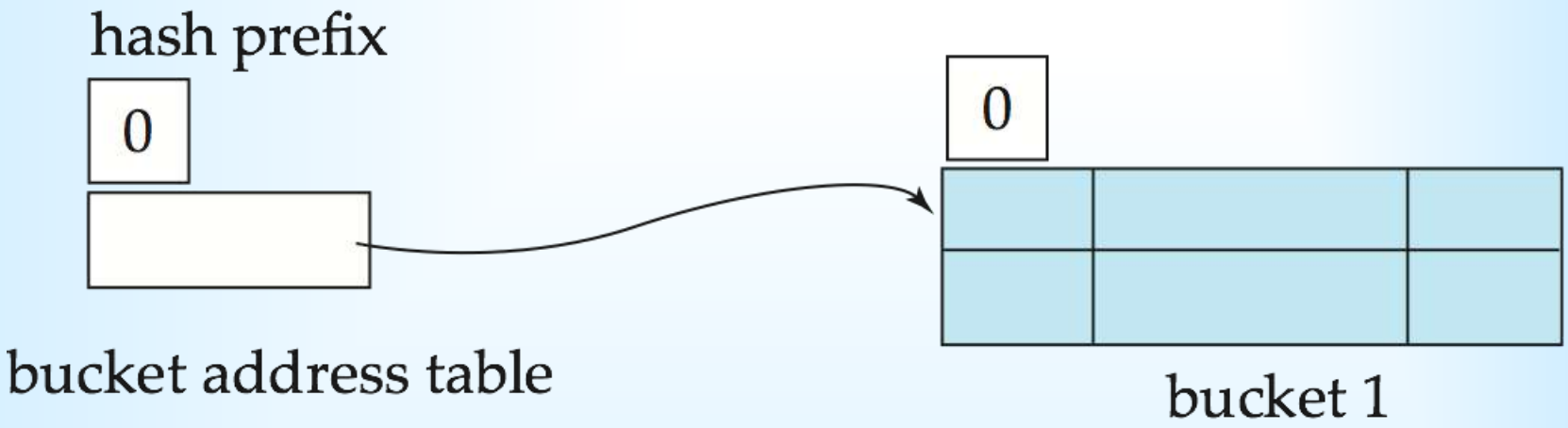
dept_name	h(*dept_name*)
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100 1001 1110 1011
Physics	1001 1000 0011 1111 1001 1100 0000 0001

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	



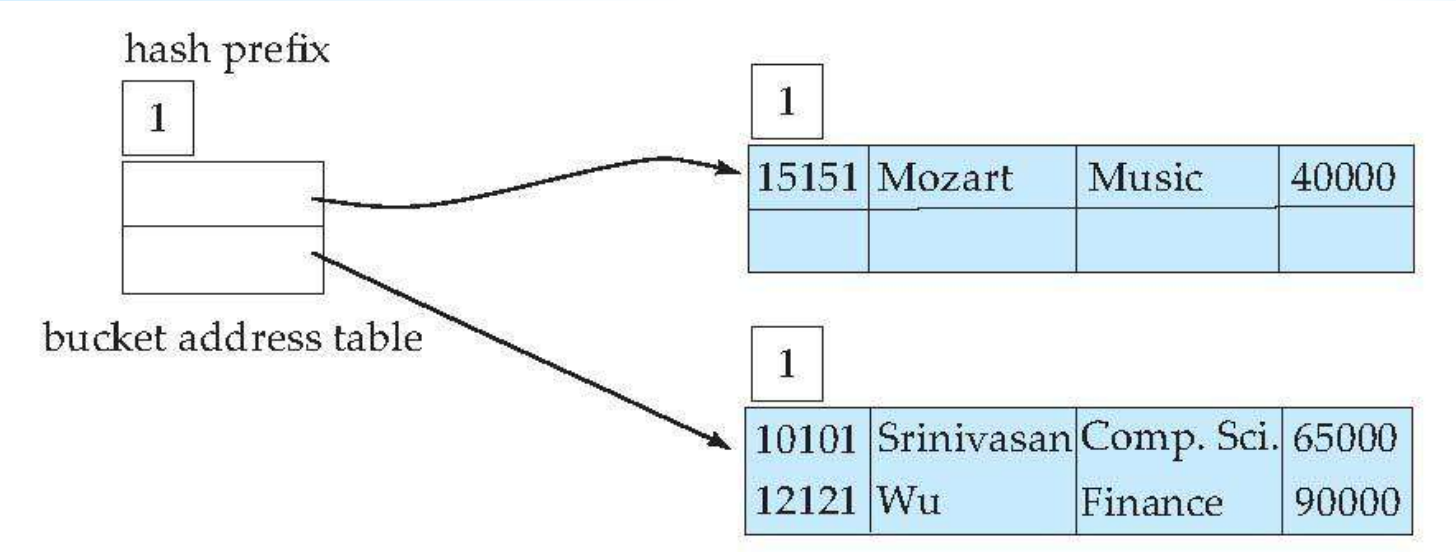
Example (Cont.)

- Initial Hash structure; bucket size = 2



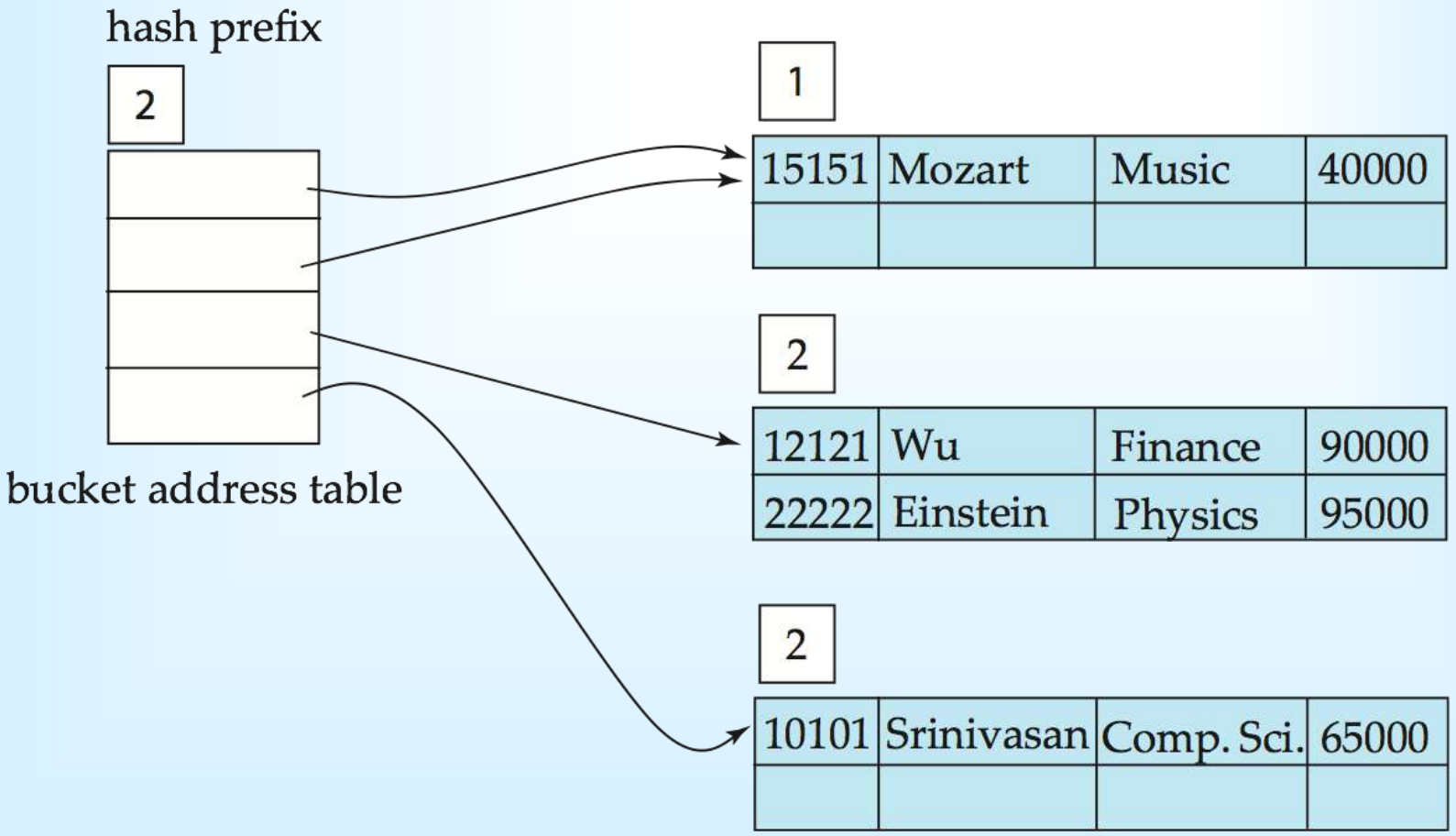
Example (Cont.)

- Hash structure after insertion of “Mozart”, “Srinivasan”, and “Wu” records



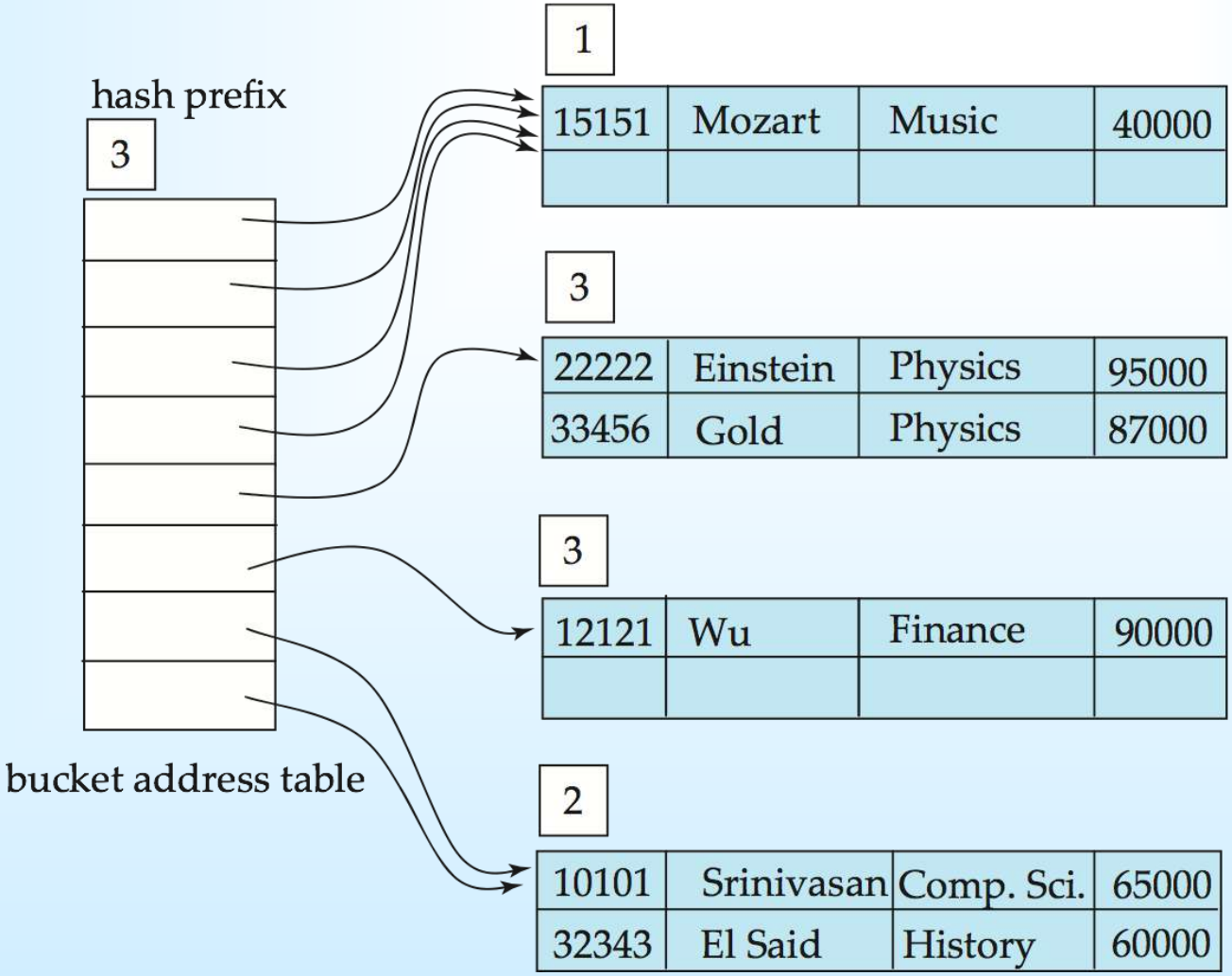
Example (Cont.)

- Hash structure after insertion of Einstein record



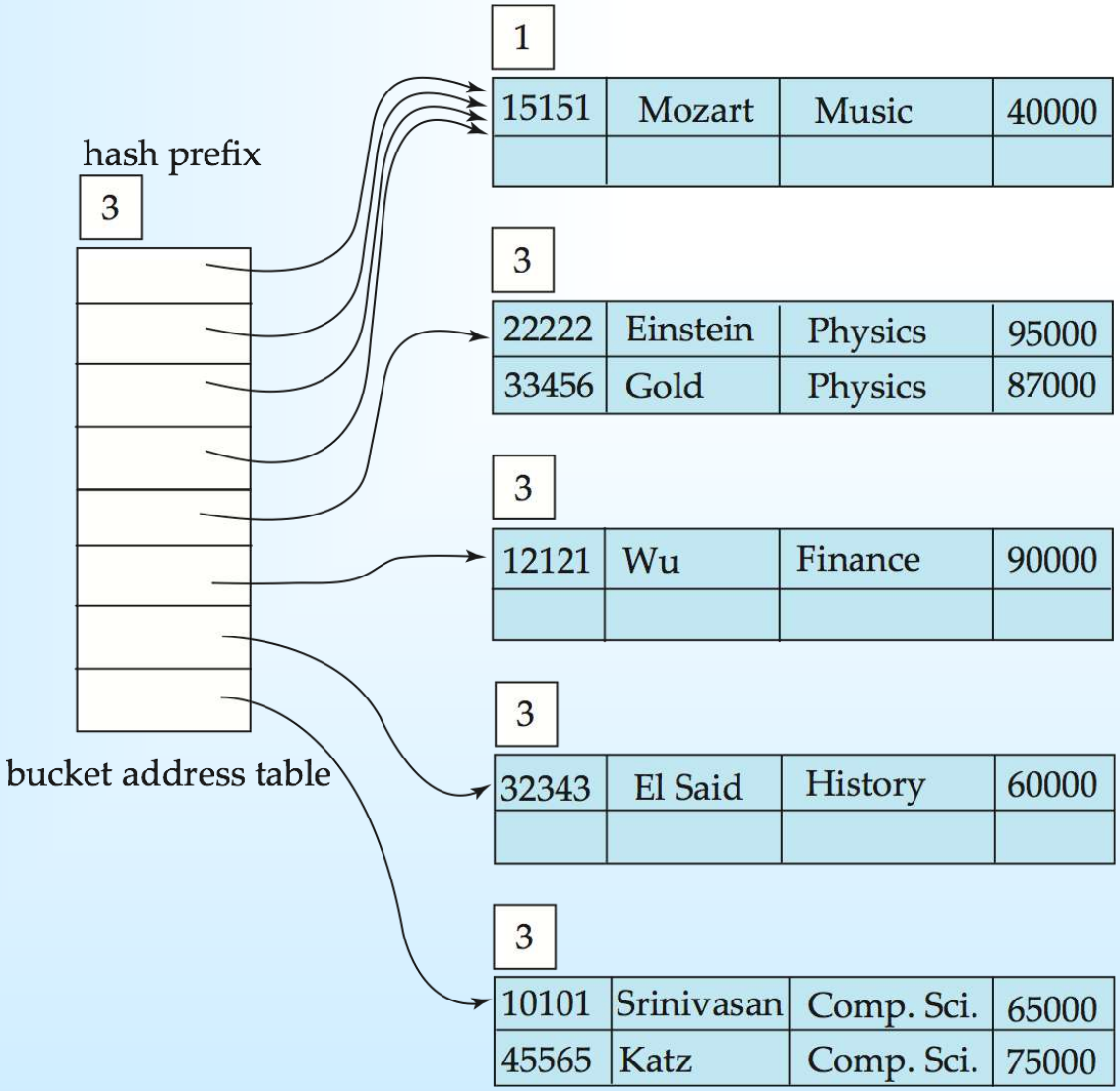
Example (Cont.)

- Hash structure after insertion of Gold and El Said records

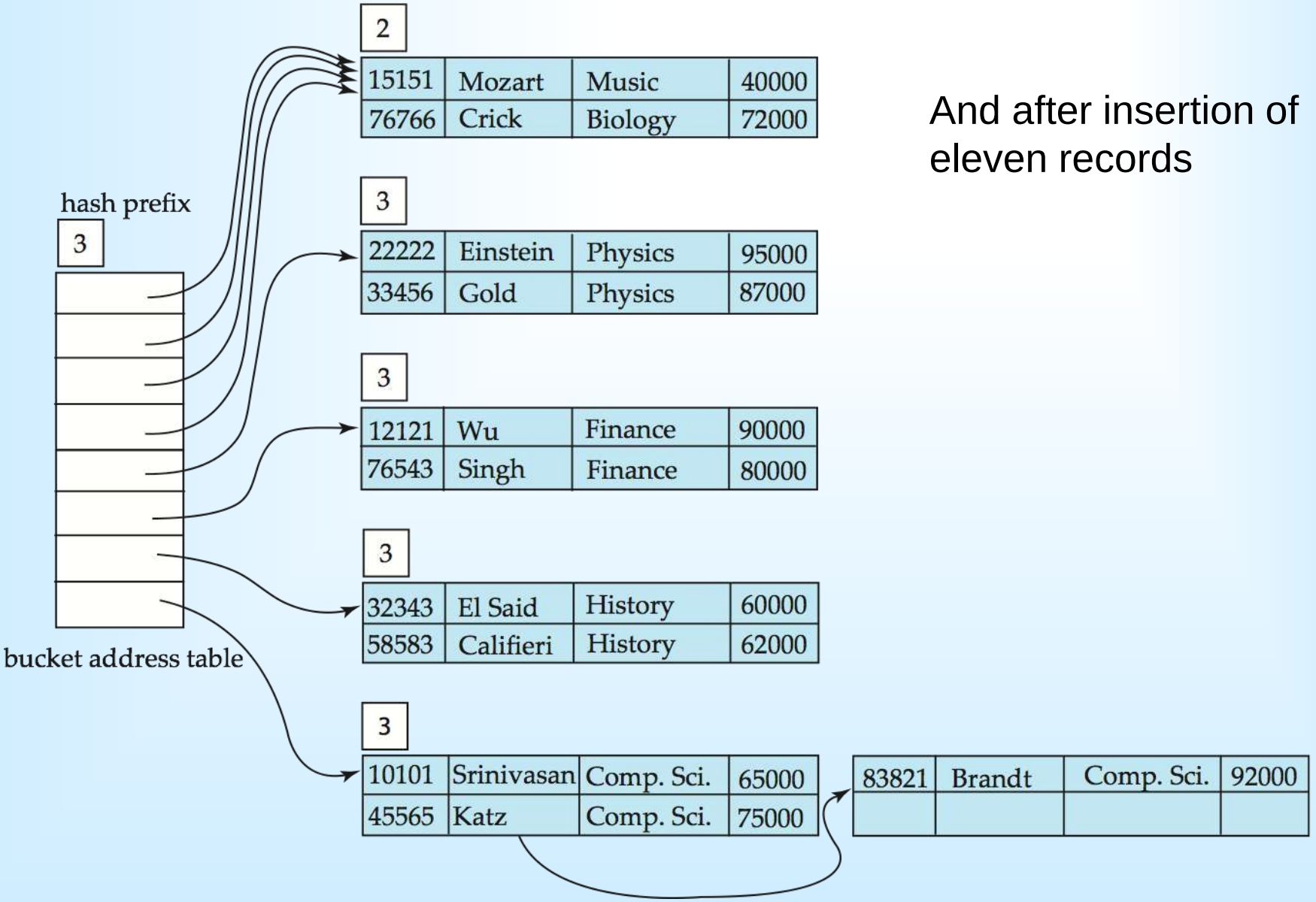


Example (Cont.)

- Hash structure after insertion of Katz record

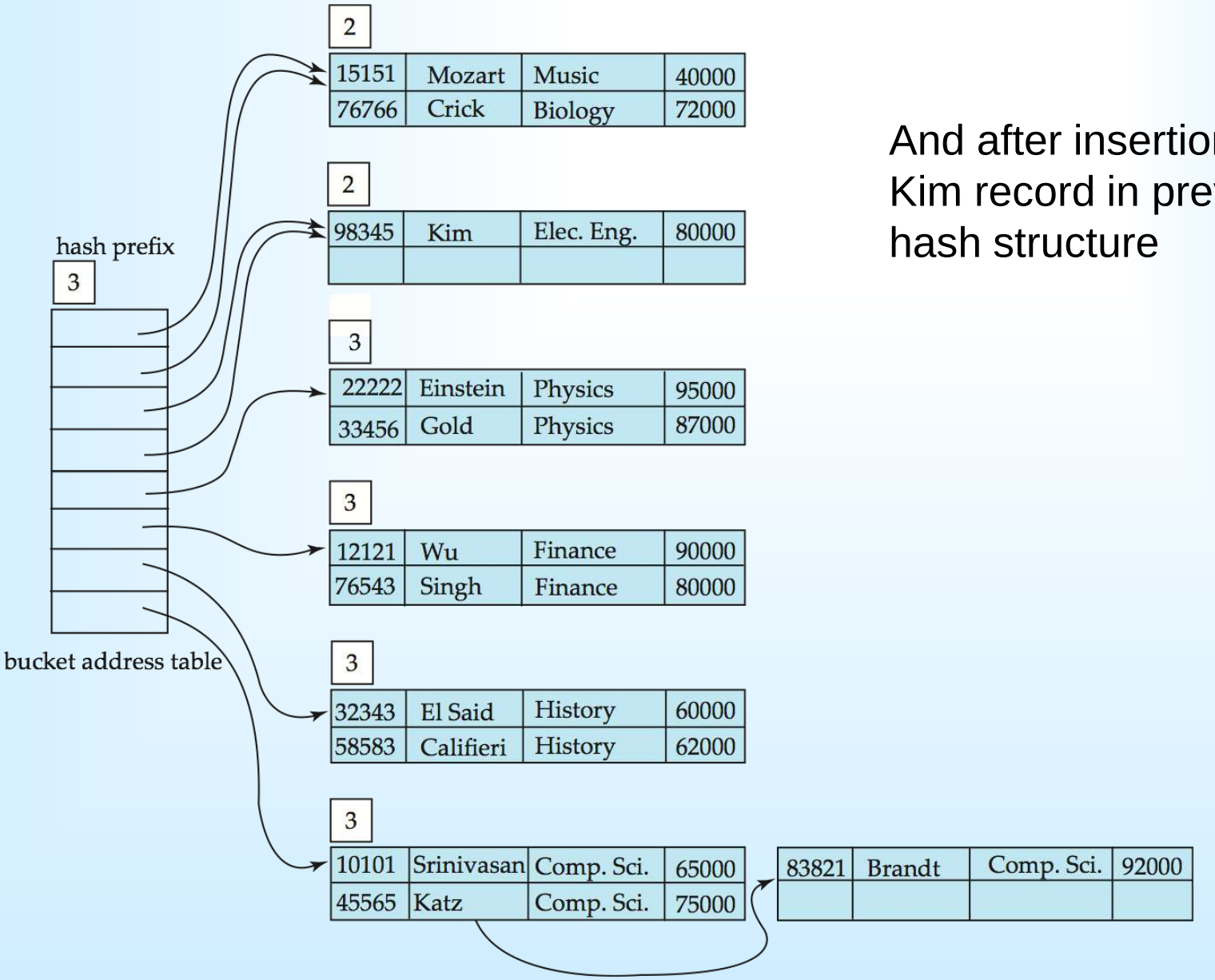


Example (Cont.)



And after insertion of eleven records

Example (Cont.)



And after insertion of Kim record in previous hash structure

Extendable Hashing vs. Other Schemes

- Benefits of extendable hashing:
  - Hash performance does not degrade with growth of file.
  - Minimal space overhead
- Disadvantages of extendable hashing
  - Extra level of indirection to find desired record, since the system must access the bucket address table before accessing the bucket itself.
  - Bucket address table may itself become very big (larger than memory)
    - Cannot allocate very large contiguous areas on disk either
    - Solution: B^+-tree structure to locate desired record in bucket address table
  - Changing size of bucket address table is an expensive operation
- Linear hashing is an alternative mechanism
  - Allows incremental growth of its directory (equivalent to bucket address table)
  - At the cost of more bucket overflows

Comparison of Ordered Indexing and Hashing

- Cost of periodic re-organization
- Relative frequency of insertions and deletions
- Is it desirable to optimize average access time at the expense of worst-case access time?
- Expected type of queries:
  - Hashing is generally better at retrieving records having a specified value of the key.
  - If range queries are common, ordered indices are to be preferred
- In practice:
  - PostgreSQL supports hash indices, but discourages use due to poor performance
  - Oracle supports static hash organization, but not hash indices
  - SQLServer supports only B^+-trees

Bitmap Indices

- Bitmap indices are a special type of index designed for efficient querying on multiple keys
- 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
- Applicable on attributes that take on a relatively small number of distinct values
  - 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)

Bitmap Indices (Cont.)

- A bitmap is simply an array of bits
- In its simplest form a bitmap index on an attribute A of relation r has one bitmap for each value of the attribute
  - Bitmap has as many bits as 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 (Cont.)

- Bitmap indices are useful for queries on multiple attributes
  - not particularly useful for single attribute queries
- Queries are answered using bitmap operations
  - Intersection (and)
  - Union (or)
  - Complementation (not)
- 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
  - Males with income level L1: 10010 AND 10100 = 10000
    - Can then retrieve required tuples.
    - Counting number of matching tuples is even faster

End of Unit 4