Chapter 14: Indexing

(and also chapter 24: Advanced Indexing)

Sistemas de Bases de Dados 2019/20

Capítulo refere-se a: Database System Concepts, 7th Ed

Outline

Multiple-Key Access

- Use multiple indices for certain types of queries.
- Example:

  select *ID*

  from *instructor*

  where *dept_name* = "Finance" **and** *salary* = 80000
- Possible strategies for processing query using 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.

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** *salary* = 80000
  - May fetch many records that satisfy the first but not the second condition

Covering Indices

- Indices on multiple attributes can be useful even for queries that do not require search in all the attributes of the index.
- Suppose you have a table of employees, with primary key *ID* and with several more attributes, including *name* and *salary*
  - Also, assume that there exists a clustering index on the attribute of the primary key
- Now, suppose that queries for knowing the salaries of employees given their name are quite frequent, i.e. of the form:

  select salary **from** employees **where** *name=* "…"
- Though in such a query we are only searching tuples by name, a non-clustering index (*name, salary*) may be quite useful.
- **Why?**
  - The salary of each employee is directly stored in the B+-tree (as part of the search key)
  - Thus, to know the salary of an employee one does not need to access the table!

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
  - Given a number *n* it must be easy to retrieve record *n*
    - Particularly easy if records are of fixed size
    - Otherwise, a list of pointer might be used
- 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)
- A bitmap is simply an array of bits

Bitmap Indices (Cont.)

- In its simplest form a bitmap index on an attribute has a **bitmap for each value** of the attribute
  - The bitmap has as many bits as records in the table
  - 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
- Example

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
  - In general, not particularly useful for single attribute queries
- Queries are answered using bitmap operations
  - Intersection (and)
  - Union (or)
- Each operation takes two bitmaps of the same size and applies the operation on corresponding bits to get the result bitmap
  - E.g., 100110 AND 110011 = 100010

    100110 OR 110011 = 110111

    NOT 100110 = 011001
  - Males with income level L1: 10010 AND 10100 = 10000
    - Can then retrieve required tuples.
    - Counting number of matching tuples is even faster
      - It doesn't even require access to the table file!

Bitmap Indices (Cont.)

- Bitmap indices are generally very small compared to the relation size
  - E.g., if a record is 1000 bytes, space for a single bitmap is 1/8000 of the space used by relation.
    - If the number of distinct attribute values is 8, bitmap is only 1/1000 of the relation size
- Deletion needs to be handled properly
  - **Existence bitmap** to mark whether there is a valid record at a record location
  - Needed for complementation
    - not(*A*=*v*): *(NOT bitmap-A-v) AND ExistenceBitmap*
- Should keep bitmaps for all values, even null value
  - To correctly handle SQL null semantics for NOT(*A*=*v*):
    - intersect above result with (NOT *bitmap-A-Null*)

Efficient Implementation of Bitmap Operations

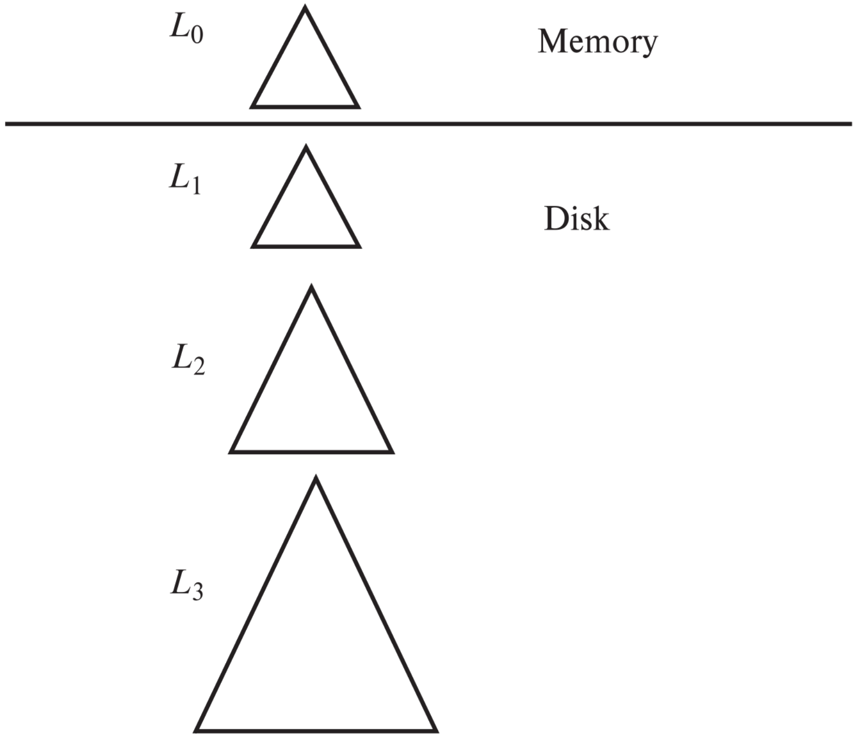
- Bitmaps are packed into words; a single word and (a basic CPU instruction) computes AND of 32 or 64 bits at once
  - E.g., 1-million-bit maps can be and-ed with just 31,250 instruction
- Counting number of 1s can be done fast by a trick:
  - Use each byte to index into a precomputed array of 256 elements each storing the count of 1s in the binary representation
    - Can use pairs of bytes to speed up further at a higher memory cost
  - Add up the retrieved counts
- Bitmaps can be used instead of Tuple-ID lists at leaf levels of B^+-trees, for values that have a large number of matching records
  - Worthwhile if > 1/64 of the records have that value, assuming a tuple-id is 64 bits
  - The technique above merges benefits of bitmap and B^+-tree indices

Write Optimized Indices

- Performance of B^+-trees can be poor for write-intensive workloads
  - One I/O per leaf, assuming all internal nodes are in memory
  - The I/O operation can be in non-contiguous areas, specially if insertions are done in non sequential order (of the search key)
  - With magnetic disks, < 100 inserts per second per disk
  - With flash memory, one page overwrite per insert
- Two approaches to reducing cost of writes
  - **Log-structured merge tree**
  - **Buffer tree**

Log Structured Merge (LSM) Tree

- To simplify, consider only inserts/queries (more in the book)
- Records inserted first into in-memory B+-tree (L_0 tree)
- When in-memory tree is full, records moved to disk (L_1 tree)
  - B^+-tree constructed using bottom-up build by merging existing L_1 tree with records from L_0 tree
- When L_1 tree exceeds some threshold, merge into L_2 tree
  - And so on for more levels
  - Size threshold for L_{i+1} tree is *k* times size threshold for L_i tree
  - Merge creates a new B^+-tree using bottom-up build

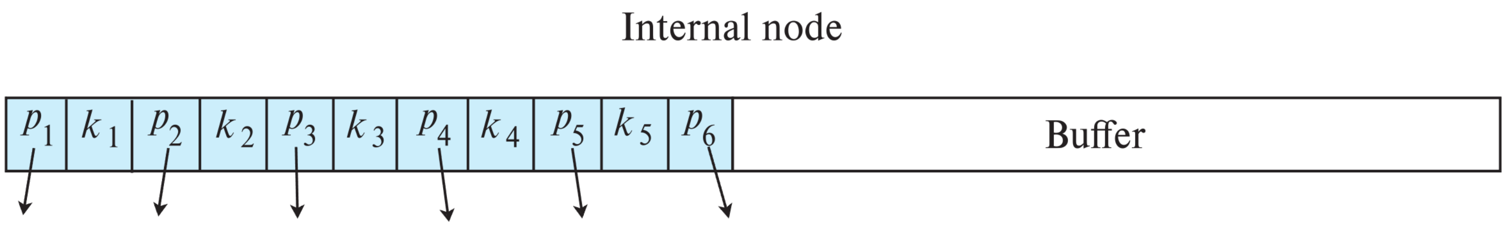


LSM Tree (Cont.)

- Benefits of LSM approach:
  - Except for L_0, which is in memory:
    - Inserts are done using only sequential I/O operations
    - Leaves are full, avoiding space wastage
    - Reduced number of I/O operations per record inserted as compared to normal B^+-tree (up to some size)
      - If each leaf has *m* entries, *m/k* entries merged in using 1 I/O
      - Total I/O operations: $k/m \log_k(I/M)$ where I = total number of entries, and M is the size of L_0 tree.
- Drawback of LSM approach
  - Queries have to search multiple trees
  - Entire content of each level copied multiple times
- Used in several BigData storage systems: Cassandra, MongoDB, …
  - MySQL also supports LSM trees (engine MyRocks)

Buffer Tree

- Alternative to LSM tree
- Key idea: each internal node of B$^+$-tree has a buffer to store inserts
  - Inserts are moved to lower levels when buffer is full
  - With a large buffer, many records are moved to lower level each time
  - Per record I/O decreases correspondingly
- Benefits
  - Less overhead on queries, and less writes
  - Can be used with any tree index structure
  - Used in PostgreSQL Generalized Search Tree (GiST) indices
- Drawback: more random I/O than LSM tree
  - Bad for magnetic disk, but not a problem for SSD


Internal node
p_1 k_1 p_2 k_2 p_3 k_3 p_4 k_4 p_5 k_5 p_6
Buffer

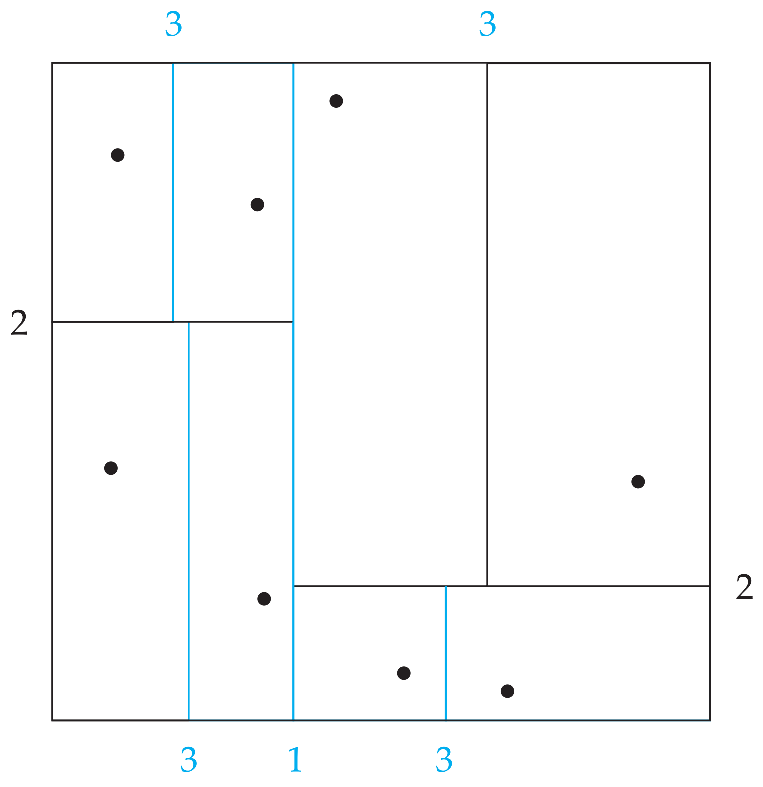
Spatial and Temporal Indices

Spatial Data

- Databases can store data types such as lines, polygons, in addition to raster images
  - allows relational databases to store and retrieve spatial information
  - Queries can use spatial conditions (e.g. contains or overlaps).
  - queries can mix spatial and nonspatial conditions
- **Nearest neighbor queries**, given a point or an object, find the nearest object that satisfies a given conditions.
- **Range queries** deal with spatial regions. e.g., ask for objects that lie partially or fully inside a specified region.
- Queries that compute intersections or unions of regions.
- **Spatial join** of two spatial relations with the location playing the role of join attribute.

Indexing of Spatial Data

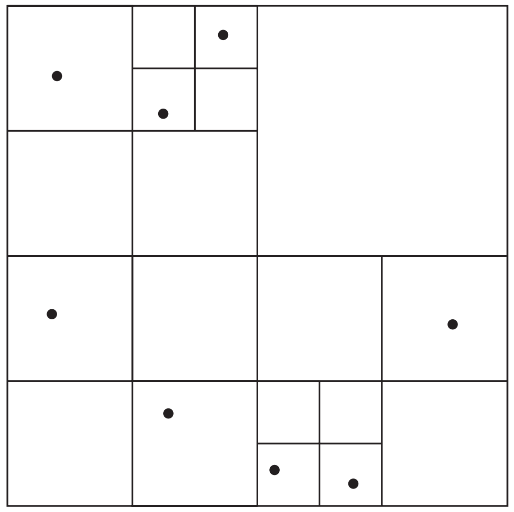
- **k-d tree** - early structure used for indexing in multiple dimensions.
- Each level of a *k-d* tree partitions the space into two.
  - Choose one dimension for partitioning at the root level of the tree.
  - Choose another dimensions for partitioning in nodes at the next level and so on, cycling through the dimensions.
- In each node, approximately half of the points stored in the sub-tree fall on one side and half on the other.
- Partitioning stops when a node has less than a given number of points.



- The **k-d-B tree** extends the *k-d* tree to allow multiple child nodes for each internal node; well-suited for secondary storage.

Division of Space by Quadtrees

- Each node of a quadtree is associated with a rectangular region of space; the top node is associated with the entire target space.
- Each non-leaf nodes divides its region into four equal sized quadrants
    - correspondingly each such node has four child nodes corresponding to the four quadrants and so on
- Leaf nodes have between zero and some fixed maximum number of points (set to 1 in the example below).

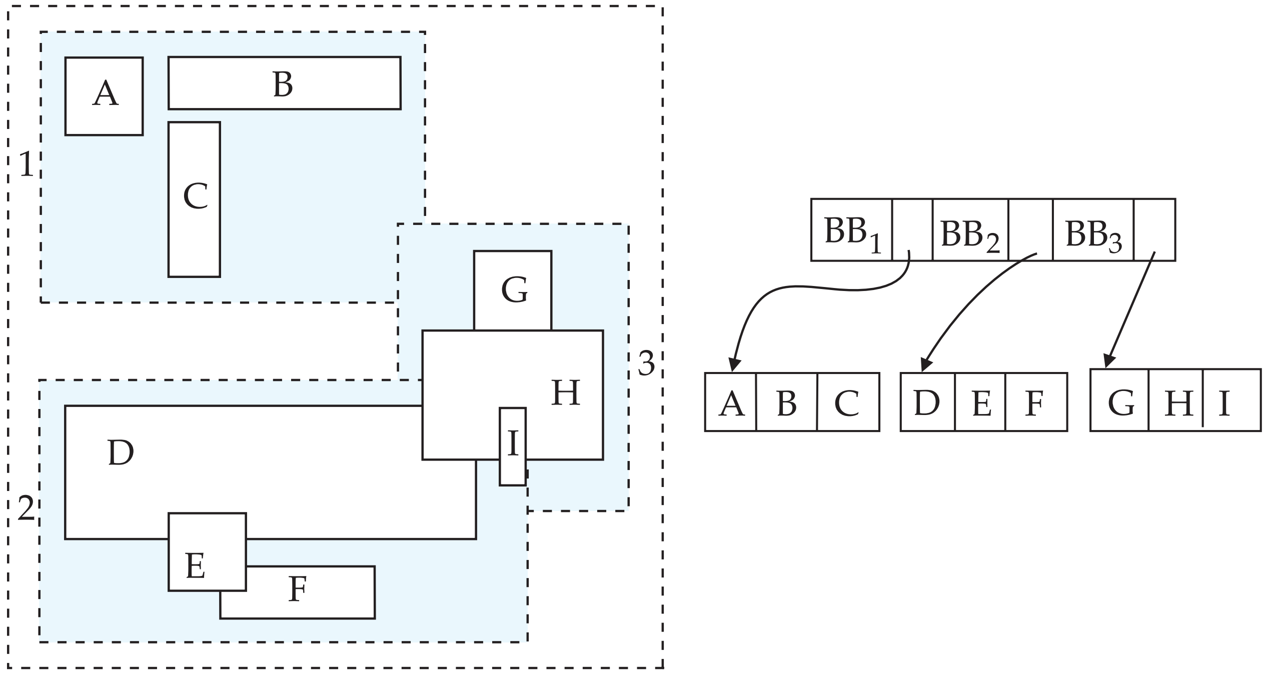


R-Trees

- **R-trees** are a N-dimensional extension of B$^+$-trees, useful for indexing sets of rectangles and other polygons.
- Supported in many modern database systems, along with variants like R$^+$ -trees and R*-trees.
- Basic idea: generalize the notion of a one-dimensional interval associated with each B$^+$-tree (intermediate) node to an N-dimensional interval, that is, an N-dimensional rectangle.
- We only consider the two-dimensional case ($N = 2$)
  - generalization for $N > 2$ is straightforward, although R-trees work well only for relatively small N
- The **bounding box** of a node is a minimum sized rectangle that contains all the rectangles/polygons associated with the node
  - *Bounding boxes of children of a node can overlap*

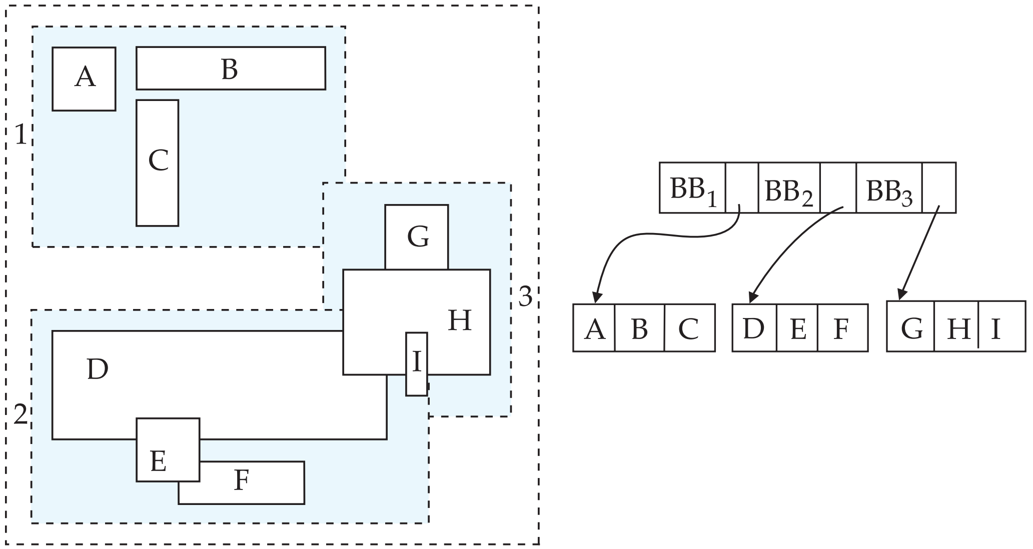
Example R-Tree

- A set of rectangles (solid line) and the bounding boxes (dashed line) of the nodes of an R-tree for the rectangles.
- The R-tree is shown on the right.



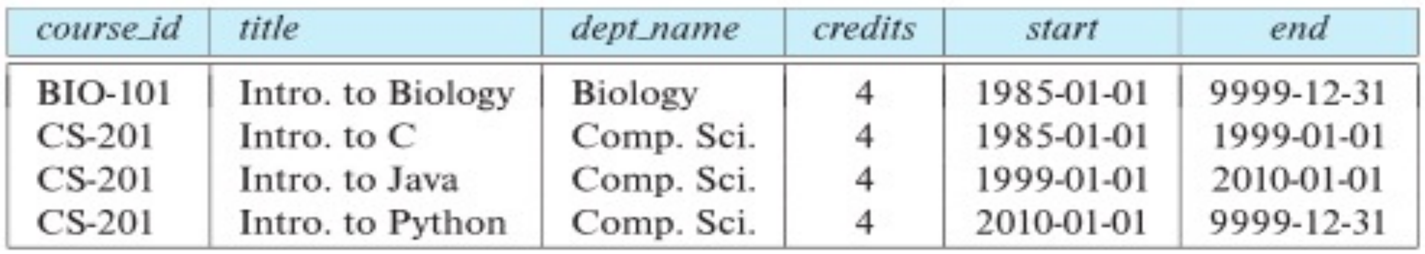
Search in R-Trees

- To find data items intersecting a given query point/region, do the following, starting from the root node:
  - If the node is a leaf node, output the data items whose keys intersect the given query point/region.
  - Else, for each child of the current node whose bounding box intersects the query point/region, recursively search the child
- Can be very inefficient in worst case since multiple paths may need to be searched but works acceptably in practice.



Indexing Temporal Data

- Temporal data refers to data that has an associated validity time period
  - Example: a temporal version of the *course* relation



course_id	*title*	*dept_name*	*credits*	*start*	*end*
BIO-101	Intro. to Biology	Biology	4	1985-01-01	9999-12-31
CS-201	Intro. to C	Comp. Sci.	4	1985-01-01	1999-01-01
CS-201	Intro. to Java	Comp. Sci.	4	1999-01-01	2010-01-01
CS-201	Intro. to Python	Comp. Sci.	4	2010-01-01	9999-12-31

- Time interval has a start and end time
  - End time set to infinity (or large date such as 9999-12-31) if a tuple is currently valid and its validity end time is not currently known
- A query may ask for all tuples that are valid at a point in time or during a time interval
  - Index on valid time period speeds up this task

Indexing Temporal Data (Cont.)

- To create a temporal index on attribute *a*:
  - Use spatial index, such as R-tree, with attribute *a* as one dimension, and time as another dimension
    - Valid time forms an interval in the time dimension
  - Tuples that are currently valid cause problems, since value is infinite or very large
    - Solution: store all current tuples (with end time as infinity) in a separate index, indexed on (*a, start-time*)
      - To find tuples valid at a point in time *t* in the current tuple index, search for tuples in the range (*a, 0*) to (*a,t*)
- Temporal index on primary key can help enforce temporal primary key constraint

course_id	*title*	*dept_name*	*credits*	*start*	*end*
BIO-101	Intro. to Biology	Biology	4	1985-01-01	9999-12-31
CS-201	Intro. to C	Comp. Sci.	4	1985-01-01	1999-01-01
CS-201	Intro. to Java	Comp. Sci.	4	1999-01-01	2010-01-01
CS-201	Intro. to Python	Comp. Sci.	4	2010-01-01	9999-12-31

Creation of Indices

- Example
  create index *takes_pk* **on** *takes* (*ID,course_ID, year, semester, section*)
  drop index *takes_pk*
- Most database systems allow specification of the type of index, and clustering.
- Indices on primary key are created automatically by almost all DBMS
- Some DBMS also create indices on foreign key attributes
  - Why might such an index be useful for this query?
    - $takes \bowtie \sigma_{name='Shankar'} (student)$
- Indices can greatly speed up lookups, but impose cost on updates
  - Index tuning assistants/wizards supported on several databases to help choose indices, based on query and update workload

Indexing in Oracle

- Oracle supports B$^+$-Tree indices as a default for the **create index** SQL command
  - B$^+$-Tree indices are created by default for every primary key and unique declaration
- A new non-null attribute *rowid* is added to all indices to non-unique attributes, so as to guarantee that all search keys are unique.
  - indices are supported on
    - attributes, and attribute lists,
    - on results of function over attributes
    - or using structures external to Oracle (Domain indices)
- Bitmap indices are also supported, but for that an explicit declaration is needed:

  create bitmap index <index-name> **on** <relation-name> (<attribute-list>)
- Oracle also has spatial indices, using R-Trees:

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

  indextype is mdsys.spatial_index

Hashing

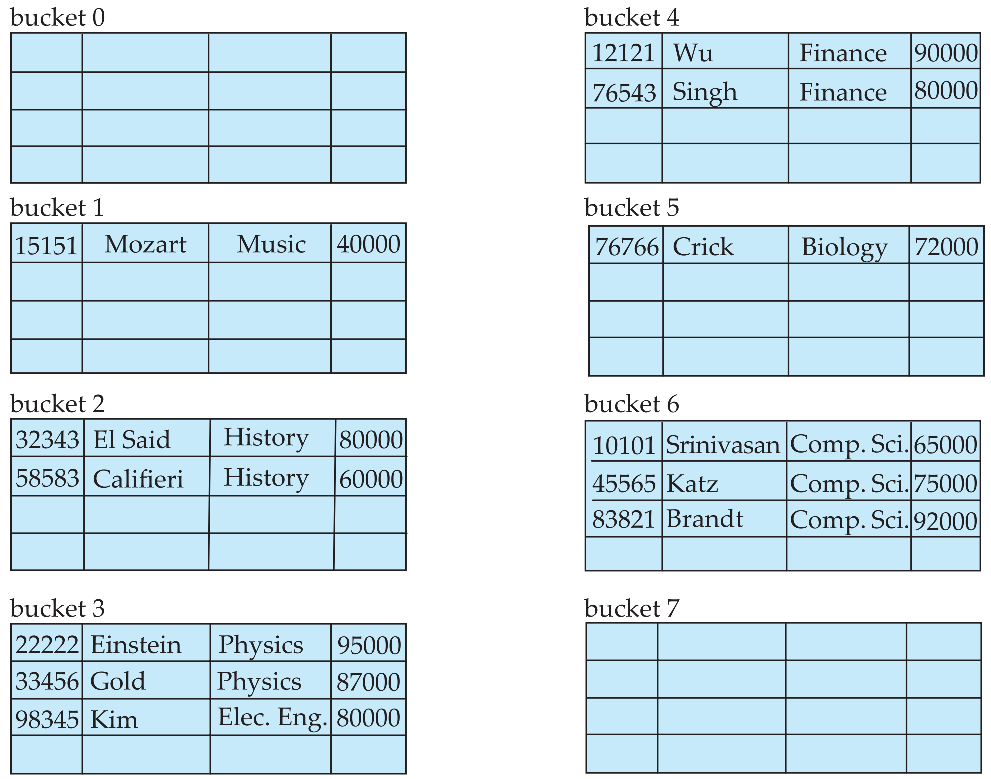
Static Hashing

- 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 must 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

Example of Hash File Organization

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

- There are 8 buckets,
- The binary representation of the *I* th character is assumed to be the integer *i.*
- 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



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 Functions

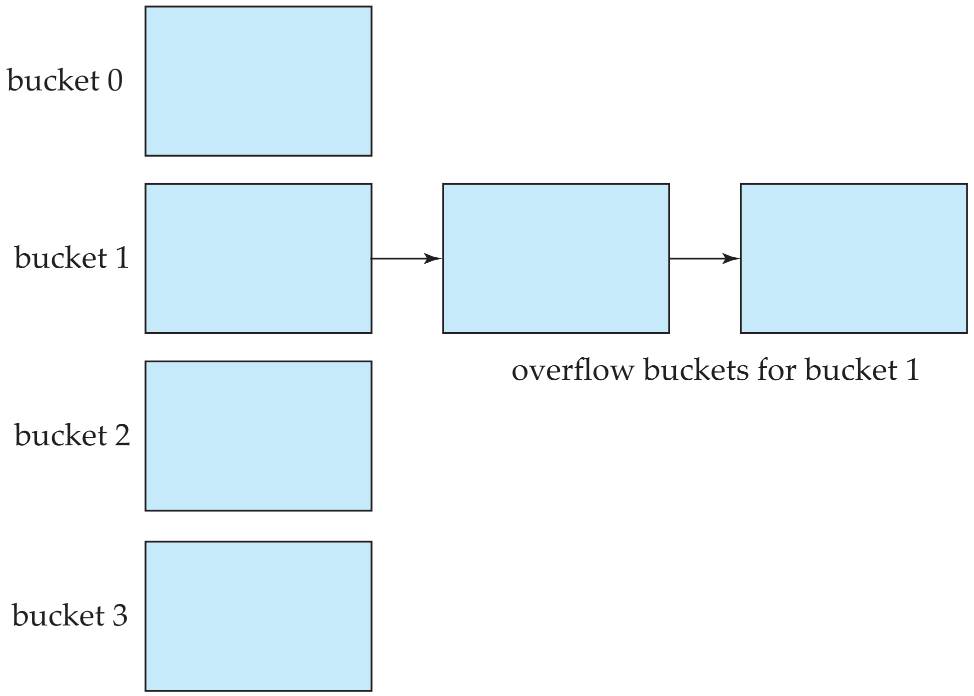
- Worst hash function maps all search-key values to the same bucket; this makes access time proportional to the number of search-key values in the file.
- An ideal hash function is **uniform**, i.e., each bucket is assigned the same number of search-key values from the set of all possible values.
- Ideal 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.
- 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 (as in previous example)

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**.*

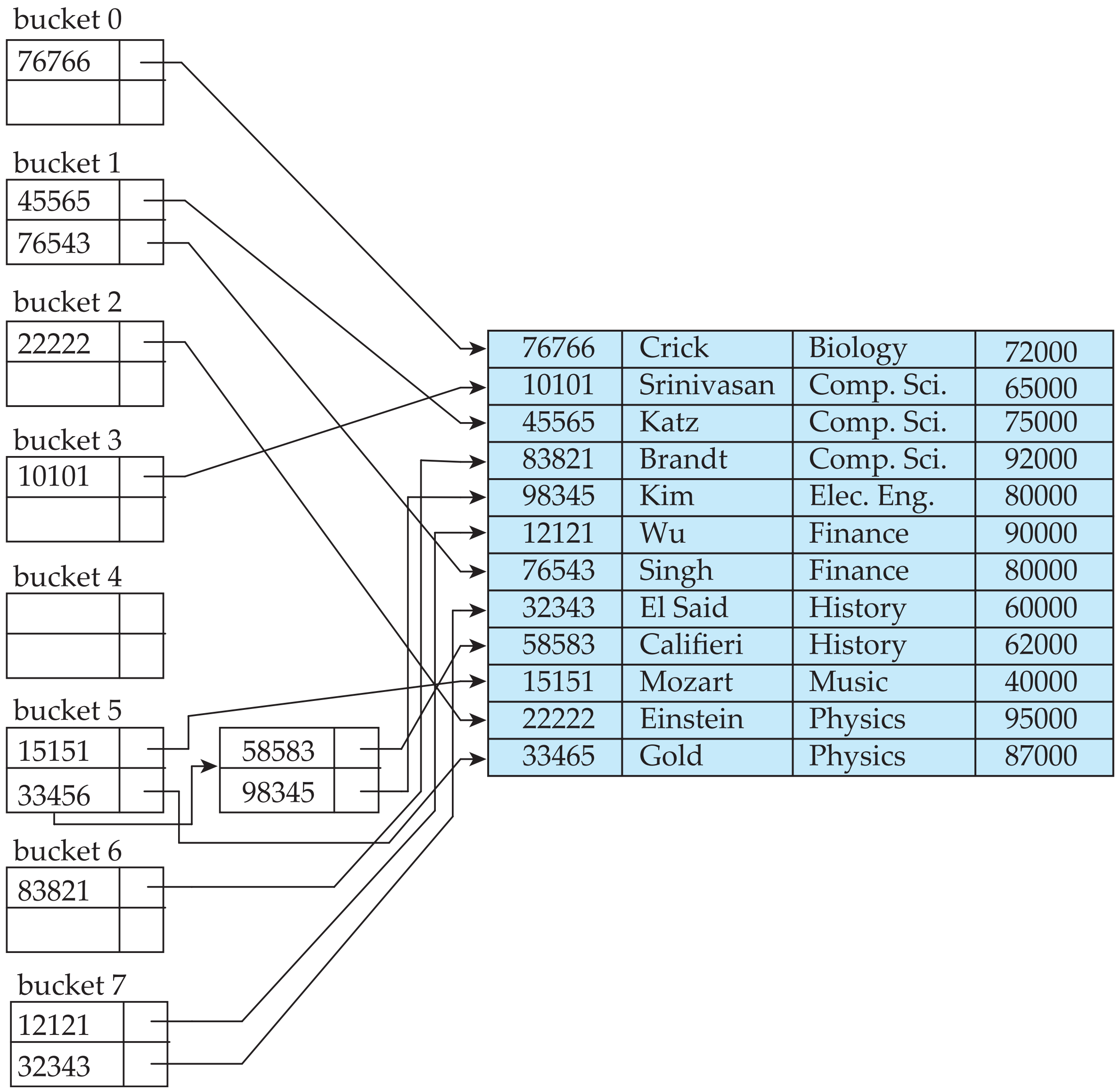
Handling of Bucket Overflows (Cont.)

- **Overflow chaining** – the overflow buckets of a given bucket are chained together in a linked list.
- Above scheme is called **closed addressing (**also called **closed hashing or open hashing** depending on the book you use**)**
  - An alternative, called **open addressing** **(**also called **open hashing** or **closed hashing** depending on the book you use) which does not use over-flow buckets, is not suitable for database applications.



Hash Indices

- Hashing can be used not only for file organisation, but also for index-structure creation.
- A **hash index** organises the search keys, with their associated record pointers, into a hash file structure.



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

Deficiencies of Static Hashing

- In static hashing, function *h* maps search-key values to a fixed set of *B* of bucket addresses. Databases shrink or grow (a lot) with time.
  - If initial number of buckets is too small, and file grows, performance will degrade due to too much overflows.
  - If space is allocated for anticipated growth, a significant amount of space will be wasted initially (and buckets will be underfull).
  - If database shrinks, again space will be wasted.
- One solution: periodic re-organization of the file with a new hash function
  - Expensive, disrupts normal operations
- Better solution: allow the number of buckets to be modified dynamically.

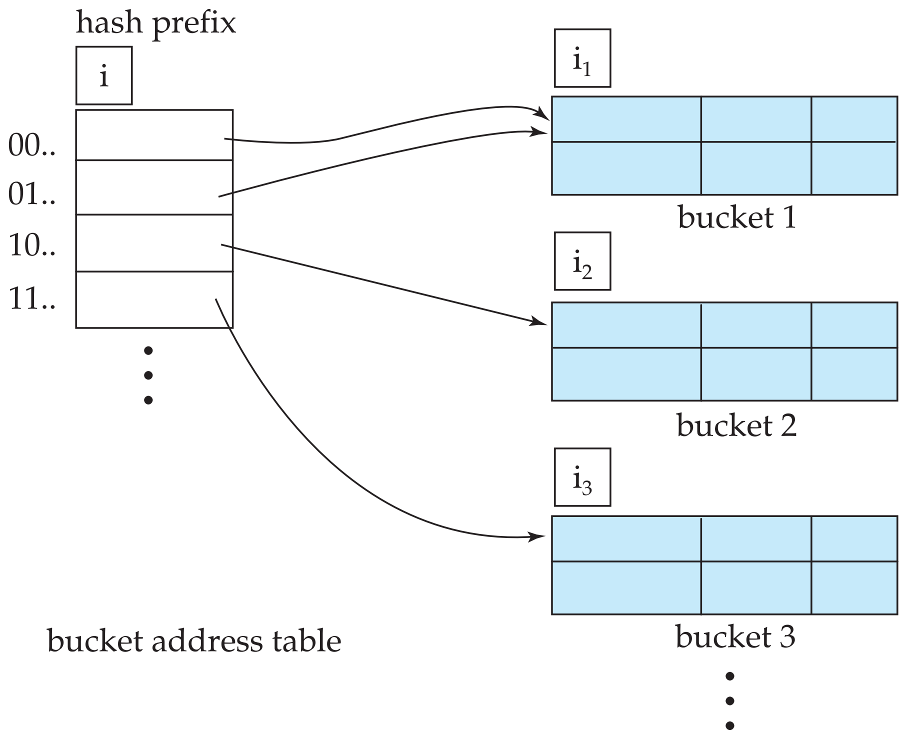
Dynamic Hashing

- Periodic rehashing
  - If number of entries in a hash table becomes (say) 1.5 times size of hash table,
    - create new hash table of size (say) 2 times the size of the previous hash table
    - Rehash all entries to new table
- Linear Hashing
  - Do rehashing in an incremental manner
- Extendable Hashing
  - Tailored to disk based hashing, with buckets shared by multiple hash values
  - Doubling of number of entries in hash table, without doubling number of buckets

Extendable Hashing

- **Extendable hashing** – one form of dynamic hashing
  - Hash function generates values over a large range — typically *b*-bit integers, with $b = 32$.
  - At any time use only a prefix of the hash function to index into a table of bucket addresses.
  - Let the length of the prefix be *i* bits, $0 \leq i \leq 32$.
    - 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 the same bucket
  - Thus, actual number of buckets is $< 2^i$
      - (with b = 32, this allows for up to 4G buckets)
    - The number of buckets also changes dynamically due to coalescing and splitting of buckets.

General Extendable Hash Structure



In this structure, $i_2 = i_3 = i$, whereas $i_1 = i - 1$ (see next slide for details)

Use of Extendable Hash Structure

- Each bucket *j* stores a value i_j
    - *A*ll 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 (will see shortly)

Insertion in Extendable Hash Structure (Cont.)

To split a bucket *j* when inserting record with search-key value K_j:

- If $i > i_j$ (more than one pointer to bucket *j*)
  - allocate a new bucket *z*, and set $i_j = i_z = (i_j + 1)$
  - Update the second half of the bucket address table entries originally pointing to *j*, to point to *z*
  - remove each record in bucket *j* and reinsert (in *j* or *z*)
  - recompute new bucket for K_j and insert record in the bucket (further splitting is required if the bucket is still full)
- If $i = i_j$ (only one pointer to bucket *j*)
  - If *i* reaches some limit *b*, or too many splits have happened in this insertion, create an overflow bucket
  - Else
    - increment *i* and double the size of the bucket address table.
    - replace each entry in the table by two entries that point to the same bucket.
    - recompute new bucket address table entry for K_j
      Now $i > i_j$ so use the first case above.

Deletion in Extendable Hash Structure

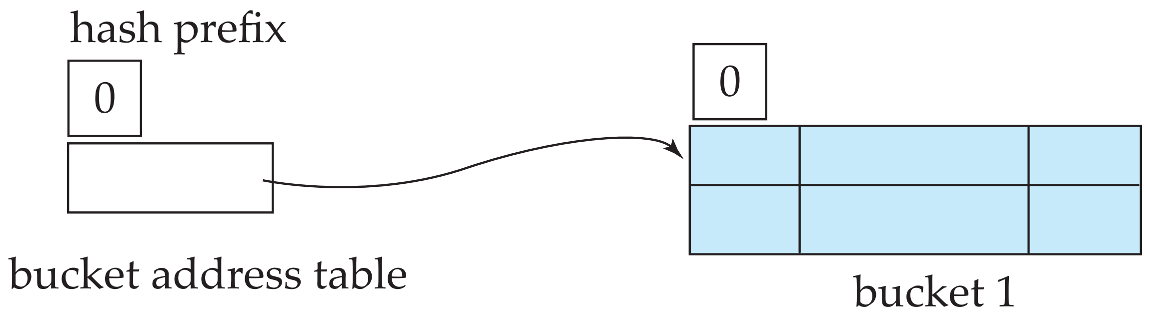
- To delete a key value,
  - locate it in its bucket and remove it.
  - The bucket itself can be removed if it becomes empty (with appropriate updates to the bucket address table).
  - Coalescing of buckets can be done (can coalesce only with a "*buddy*" bucket having same value of i_j and same $i_j - 1$ 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

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

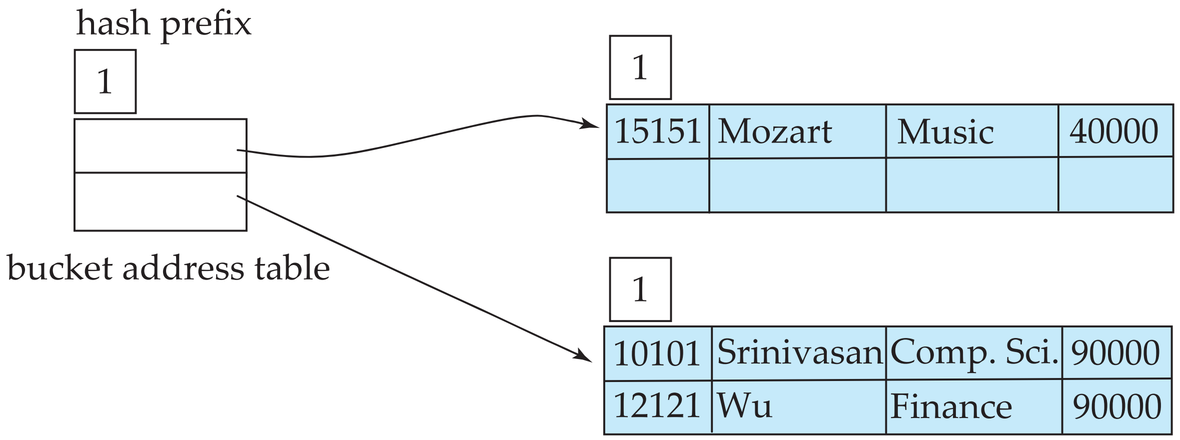
Example (Cont.)

- Initial hash structure; bucket size = 2



Example (Cont.)

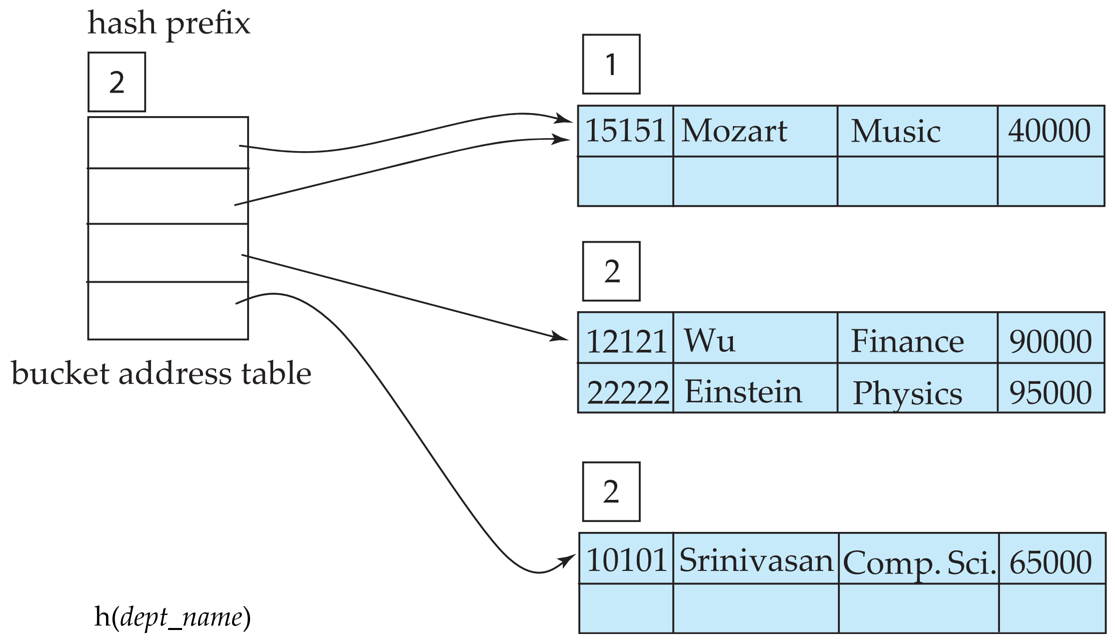
- Hash structure after insertion of "Mozart", "Srinivasan", and "Wu" records



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

Example (Cont.)

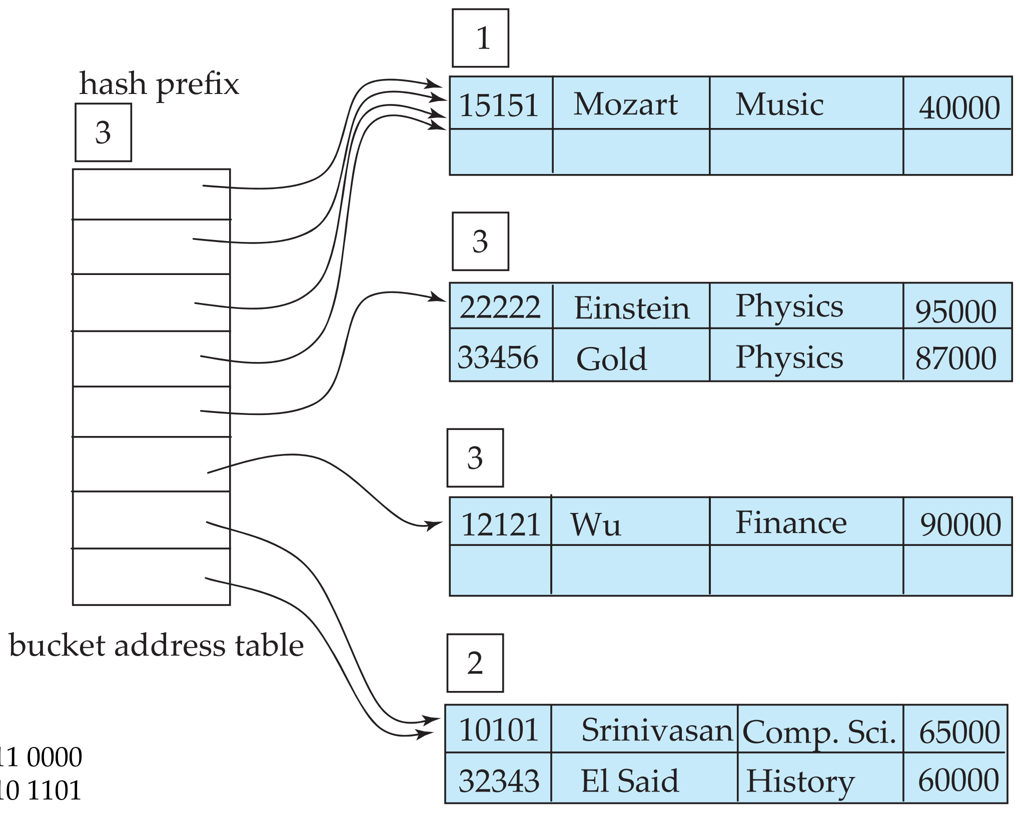
- Hash structure after insertion of Einstein record



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

Example (Cont.)

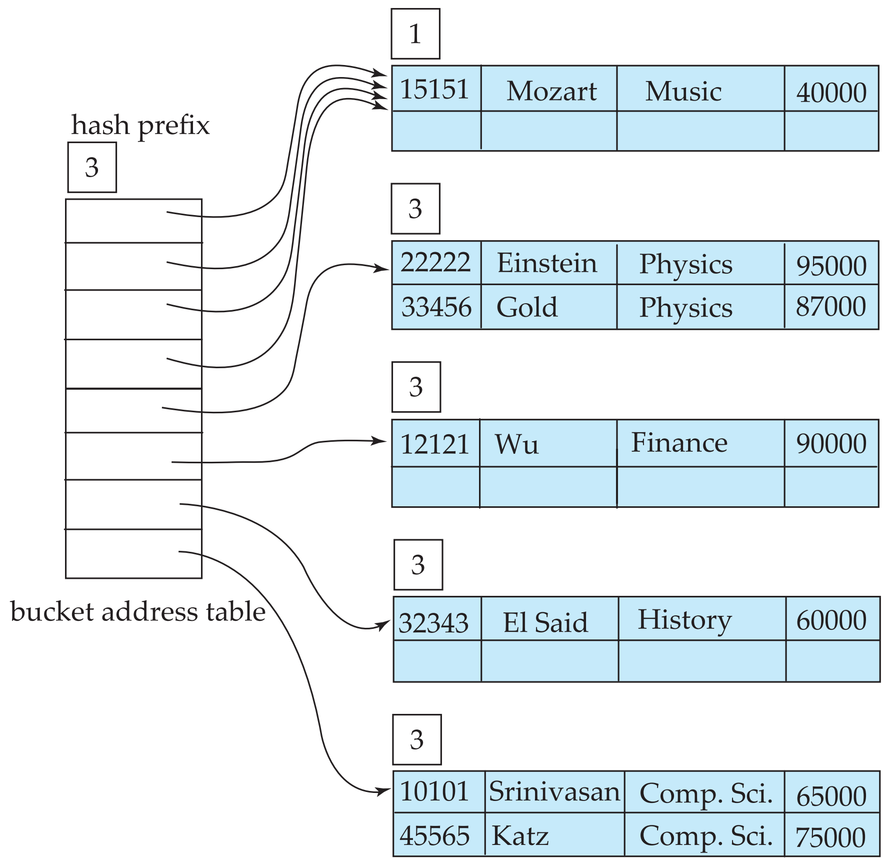
- Hash structure after insertion of Gold and El Said records



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

Example (Cont.)

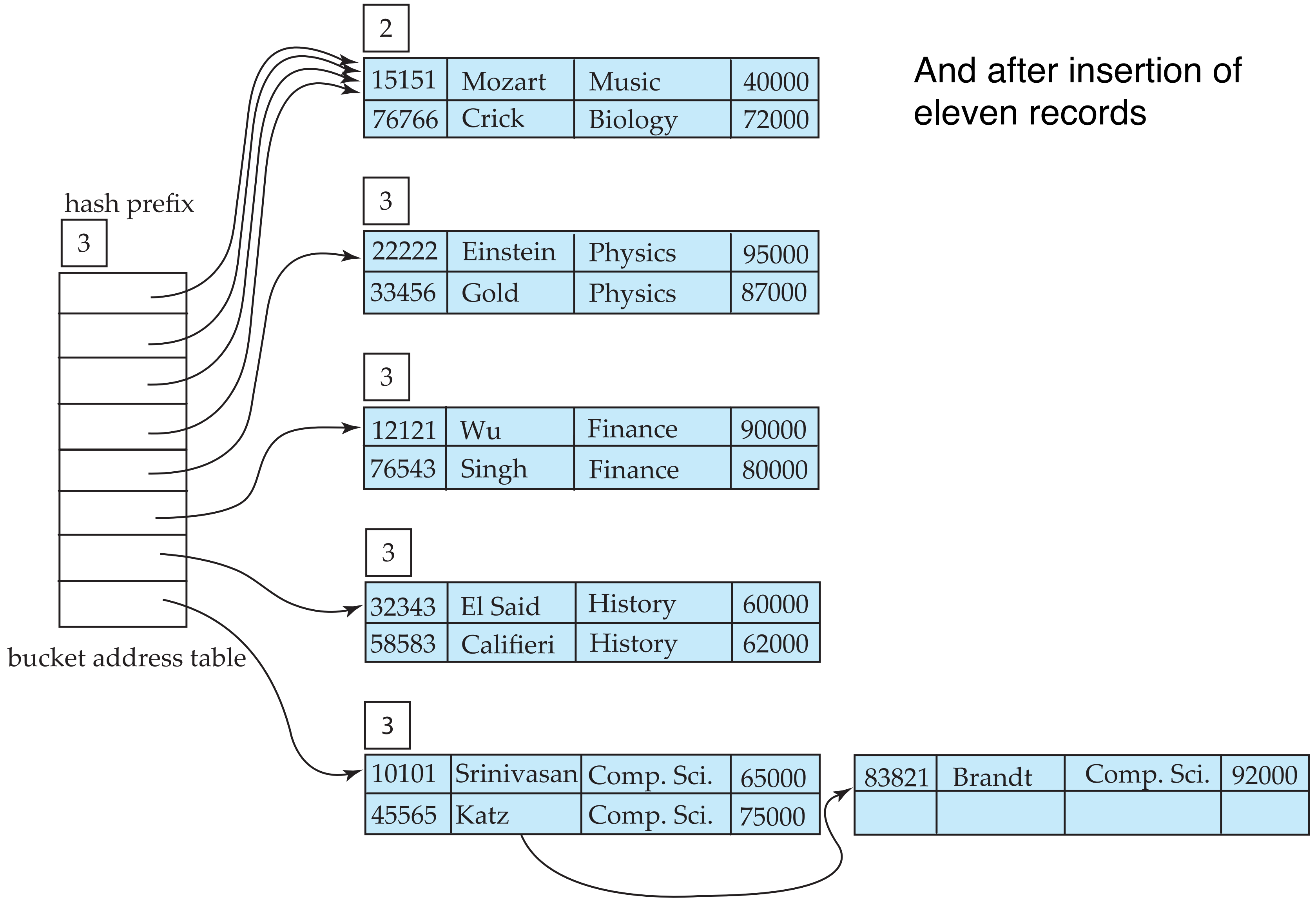
- Hash structure after insertion of Katz record



hash prefix
3
bucket address table
1
15151 Mozart Music 40000
3
22222 Einstein Physics 95000
33456 Gold Physics 87000
3
12121 Wu Finance 90000
3
32343 El Said History 60000
3
10101 Srinivasan Comp. Sci. 65000
45565 Katz Comp. Sci. 75000

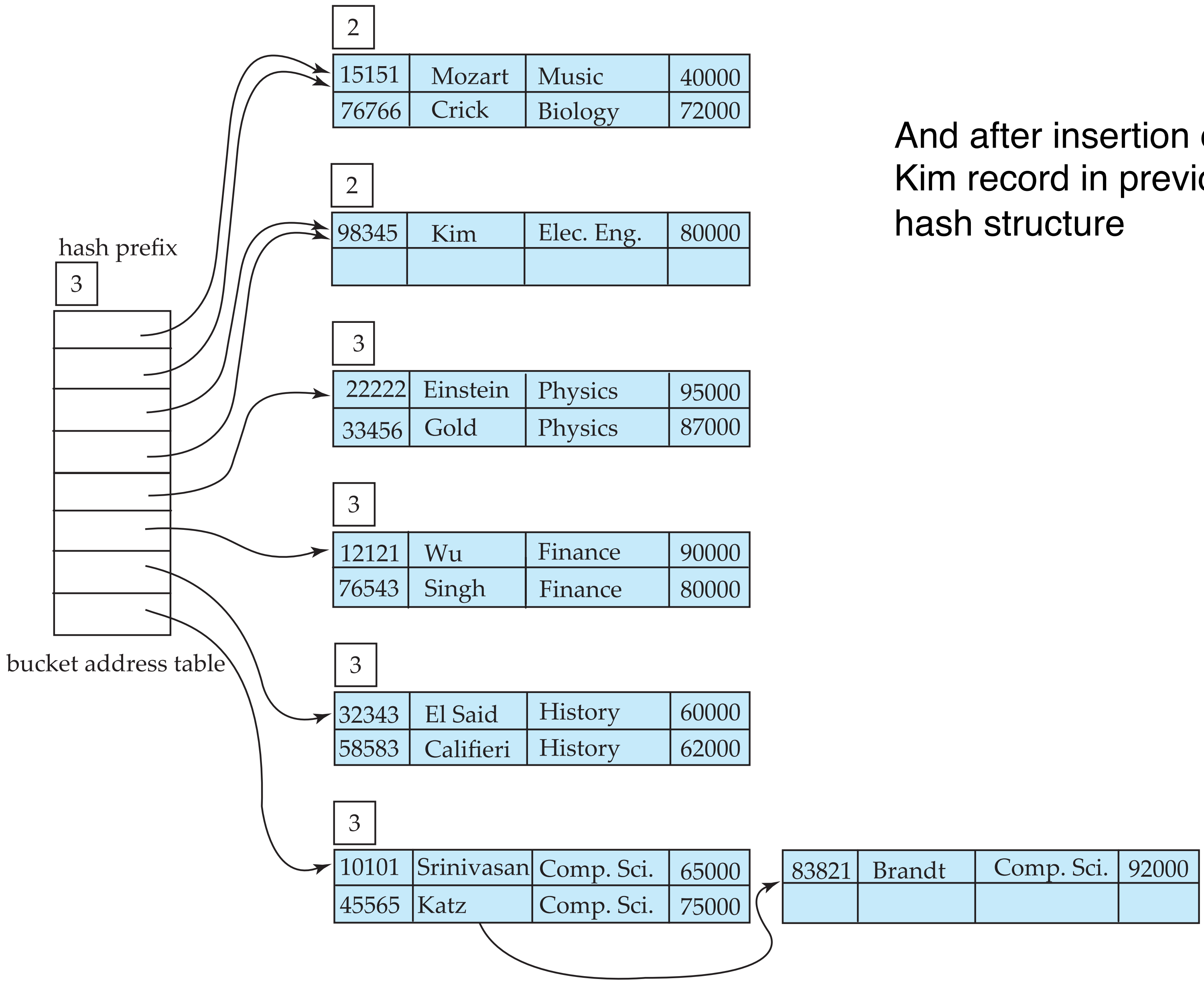
Example (Cont.)

And after insertion of eleven records



hash prefix
3
bucket address table
2
15151 | Mozart | Music | 40000
76766 | Crick | Biology | 72000
3
22222 | Einstein | Physics | 95000
33456 | Gold | Physics | 87000
3
12121 | Wu | Finance | 90000
76543 | Singh | Finance | 80000
3
32343 | El Said | History | 60000
58583 | Califieri | History | 62000
3
10101 | Srinivasan | Comp. Sci. | 65000
45565 | Katz | Comp. Sci. | 75000
83821 | Brandt | Comp. Sci. | 92000

Example (Cont.)


hash prefix
3
bucket address table
2
15151 Mozart Music 40000
76766 Crick Biology 72000
2
98345 Kim Elec. Eng. 80000
3
22222 Einstein Physics 95000
33456 Gold Physics 87000
3
12121 Wu Finance 90000
76543 Singh Finance 80000
3
32343 El Said History 60000
58583 Califieri History 62000
3
10101 Srinivasan Comp. Sci. 65000
45565 Katz Comp. Sci. 75000
83821 Brandt Comp. Sci. 92000

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
  - 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 and MySQL do not support hashing

Hashing in Oracle

- Hash indices are not supported
- However (limited) static hash file organisation is supported for partitions

  create table … **partition by hash**(<attribute-list>)
  partitions <N>
  stored in (<tables>)

- Index files can also be partitioned using hash function

  create index … global partition by **hash**(<attribute-list>)
  partitions <N>

- This creates a global index partitioned by the hash function
- (Global) indexing over hash partitioned table is also possible
- Hashing may also be used to organise clusters in multitable clusters