Chapter 24: Advanced Indexing

Database System Concepts, 7th Ed.

Bloom Filters

- A **bloom filter** is a probabilistic data structure used to check membership of a value in a set
  - May return true (with low probability) even if an element is not present
  - But never returns false if an element is present
  - Used to filter out irrelevant sets
- Key data structure is a single bitmap
  - For a set with n elements, typical bitmap size is 10n
- Uses multiple independent hash functions
- With a single hash function h() with range=number of bits in bitmap:
  - For each element s in set S compute $h(s)$ and set bit $h(s)$
  - To query an element v compute $h(v)$, and check if bit $h(v)$ is set
- Problem with single hash function: significant chance of false positive due to hash collision
  - 10% chance with 10n bits

Bloom Filters (Cont.)

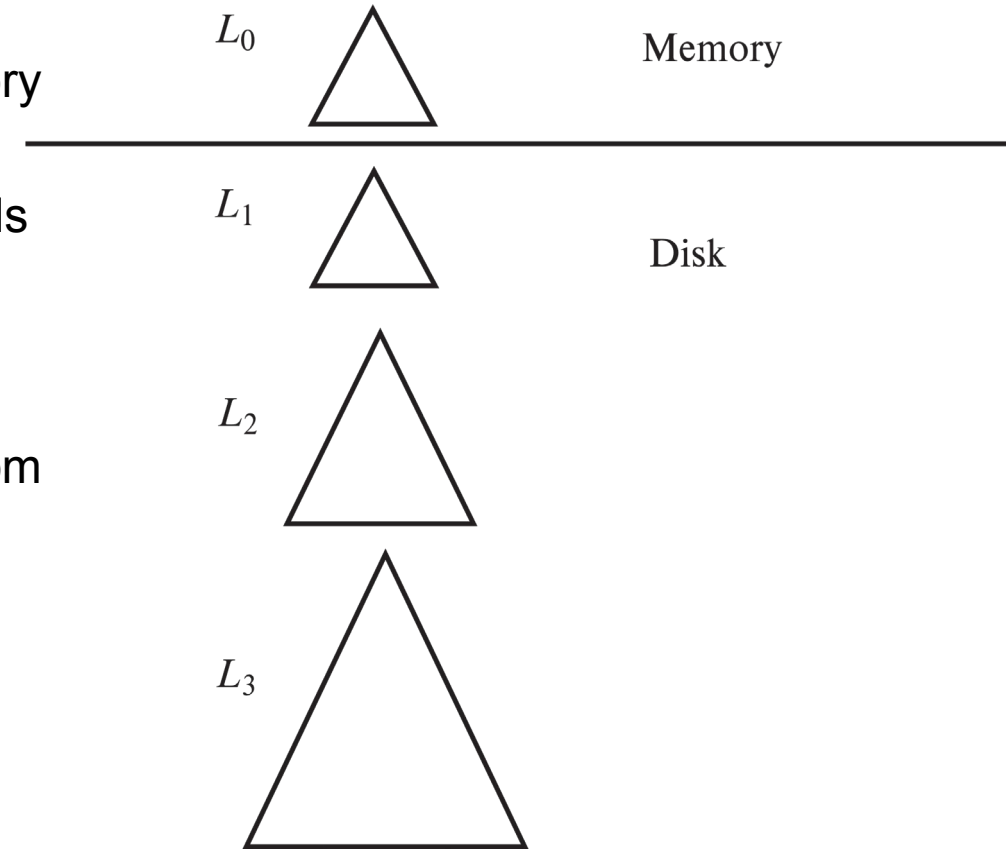
- Key idea of Bloom filter: reduce false positives by use multiple hash functions $h_i()$ for $i = 1..k$
  - For each element s in set S *for each* i compute $h_i(s)$ and set bit $h_i(s)$
  - To query an element v *for each* i compute $h_i(v)$, and check if bit $h_i(v)$ is set
    - If bit $h_i(v)$ is set for every i then report v as present in set
    - Else report v as absent
  - With 10n bits, and $k = 7$, false positive rate reduces to 1% instead of 10% with $k = 1$

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
  - 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

- Consider only inserts/queries for now
- Records inserted first into in-memory 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.)

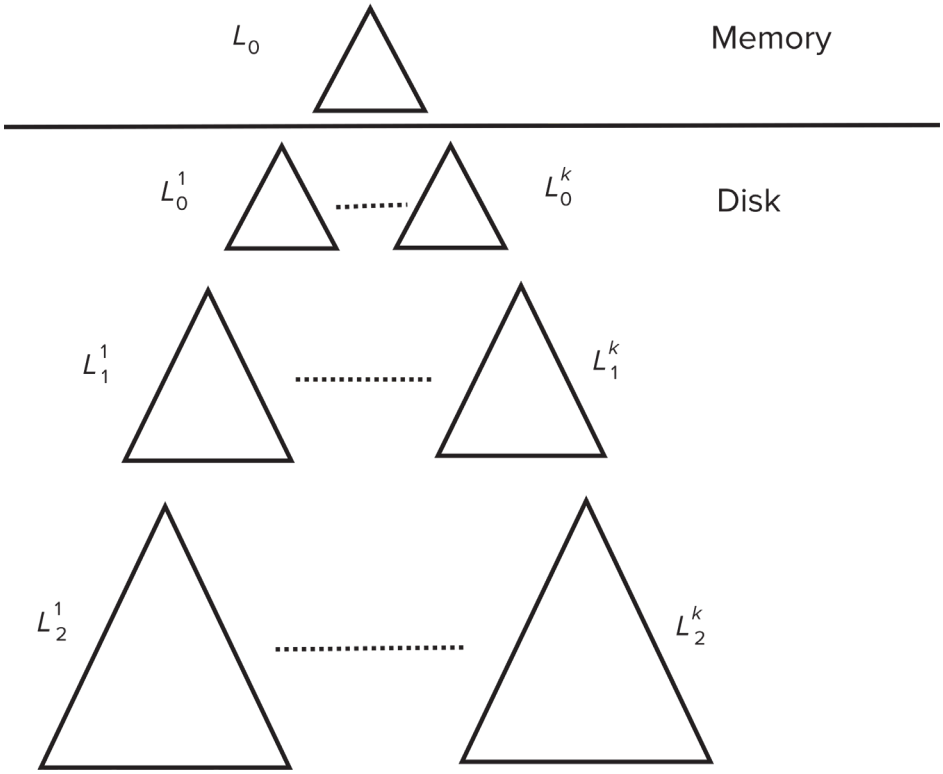
- Benefits of LSM approach
  - Inserts are done using only sequential I/O operations
  - Leaves are full, avoiding space wastage
  - Reduced number of I/O operations per record inserted as compared to normal B^+-tree (up to some size)
    - If each leaf has m entries, m/k entries merged in using 1 IO
    - 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

Optimizations of LSM

- **Rolling merge**
- LSM/Stepped Merge often implemented on a partitioned relation
  - Each partition size set to some max, split if over-sized
  - Spread partitions over multiple machines

Stepped Merge Index

- Stepped-merge index: variant of LSM tree with *k* trees at each level on disk
  - When all *k* indices exist at a level, merge them into one index of next level.
  - Reduces write cost compared to LSM tree
- But queries are even more expensive since many trees need to be queries
- Optimization for point lookups
  - Compute Bloom filter for each tree and store in-memory
  - Query a tree only if Bloom filter returns a positive result

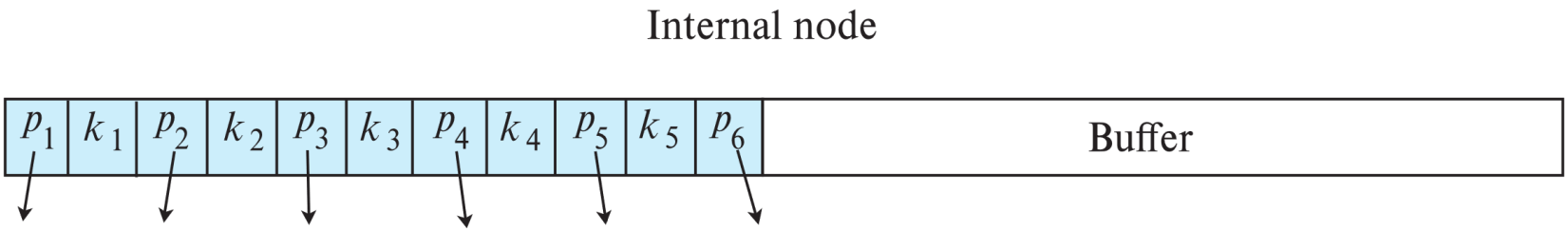


LSM Trees (Cont.)

- Deletion handled by adding special “delete” entries
  - Lookups will find both original entry and the delete entry, and must return only those entries that do not have matching delete entry
  - When trees are merged, if we find a delete entry matching an original entry, both are dropped.
- Update handled using insert+delete
- LSM trees were introduced for disk-based indices
  - But useful to minimize erases with flash-based indices
  - The stepped-merge variant of LSM trees is used in many BigData storage systems
    - Google BigTable, Apache Cassandra, MongoDB
    - And more recently in SQLite4, LevelDB, and MyRocks storage engine of MySQL

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
  - Can be used with any tree index structure
  - Used in PostgreSQL Generalized Search Tree (GiST) indices
- Drawback: more random I/O than LSM tree



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)
- 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
  - 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

Bitmap Indices (Cont.)

- Bitmap indices generally very small compared with relation size
  - E.g. if record is 100 bytes, space for a single bitmap is 1/800 of space used by relation.
    - If number of distinct attribute values is 8, bitmap is only 1% of relation size
- Deletion needs to be handled properly
  - **Existence bitmap** to note if 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
  - Above technique merges benefits of bitmap and B^+-tree indices

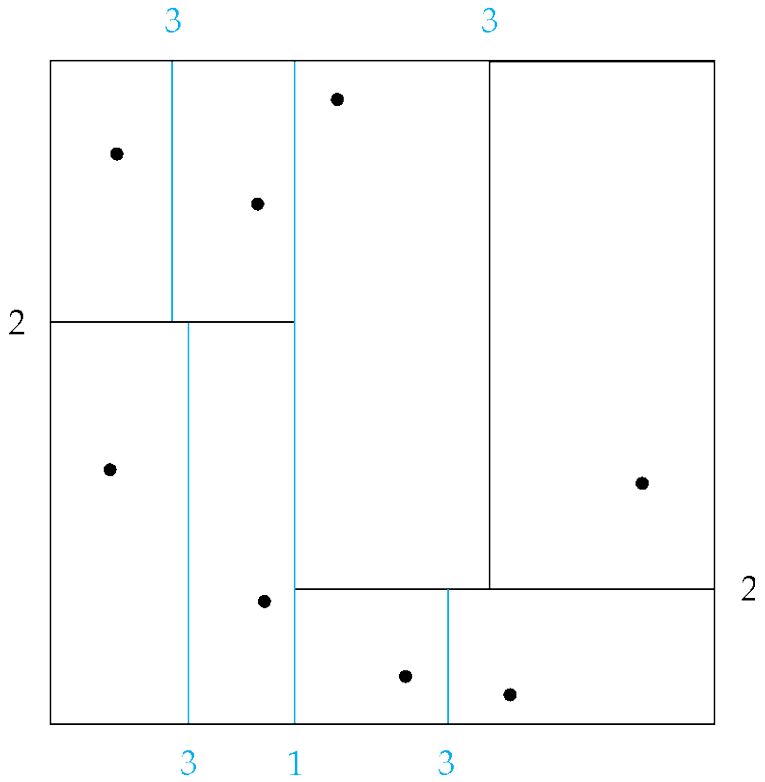
SPATIAL AND TEMPORAL INDICES

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 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.

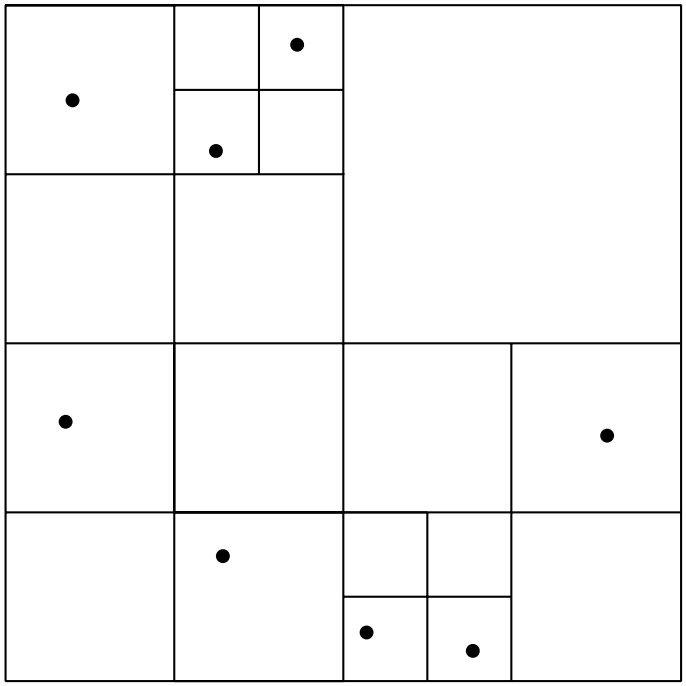


- The **k-d-B tree** extends the *k-d* tree to allow multiple child nodes for each internal node; well-suited for secondary storage.

Division of Space by Quadtrees

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 example).



Quadtrees (Cont.)

- **PR quadtree**: stores points; space is divided based on regions, rather than on the actual set of points stored.
- **Region quadtrees** store array (raster) information.
  - A node is a leaf node is all the array values in the region that it covers are the same. Otherwise, it is subdivided further into four children of equal area, and is therefore an internal node.
  - Each node corresponds to a sub-array of values.
  - The sub-arrays corresponding to leaves either contain just a single array element, or have multiple array elements, all of which have the same value.
- Extensions of *k-d* trees and PR quadtrees have been proposed to index line segments and polygons
  - Require splitting segments/polygons into pieces at partitioning boundaries
    - Same segment/polygon may be represented at several leaf nodes

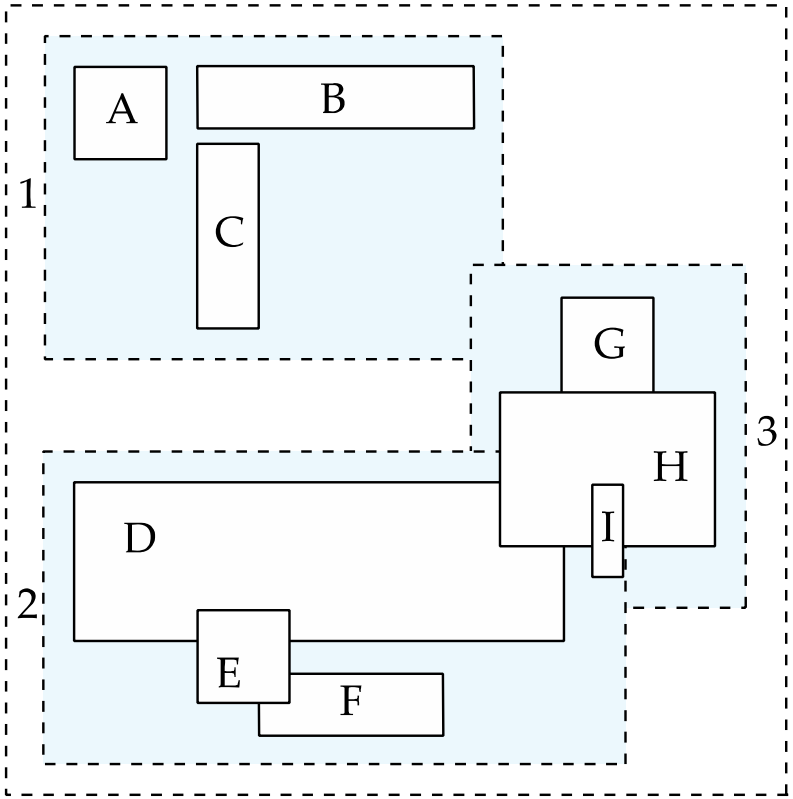
R-Trees

- **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 node to an N-dimensional interval, that is, an N-dimensional rectangle.
- Will consider only the two-dimensional case ($N = 2$)
  - generalization for $N > 2$ is straightforward, although R-trees work well only for relatively small N
- A polygon is stored only in one node, and the bounding box of the node must contain the polygon
  - The storage efficiency or R-trees is better than that of k-d trees or quadtrees since a polygon is stored only once

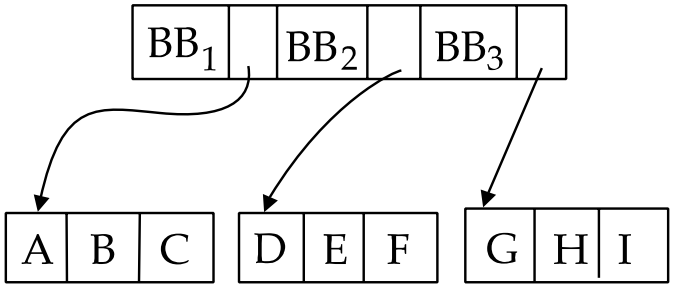
Example R-Tree

- 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 are allowed to overlap*

Rectangles being Indexed

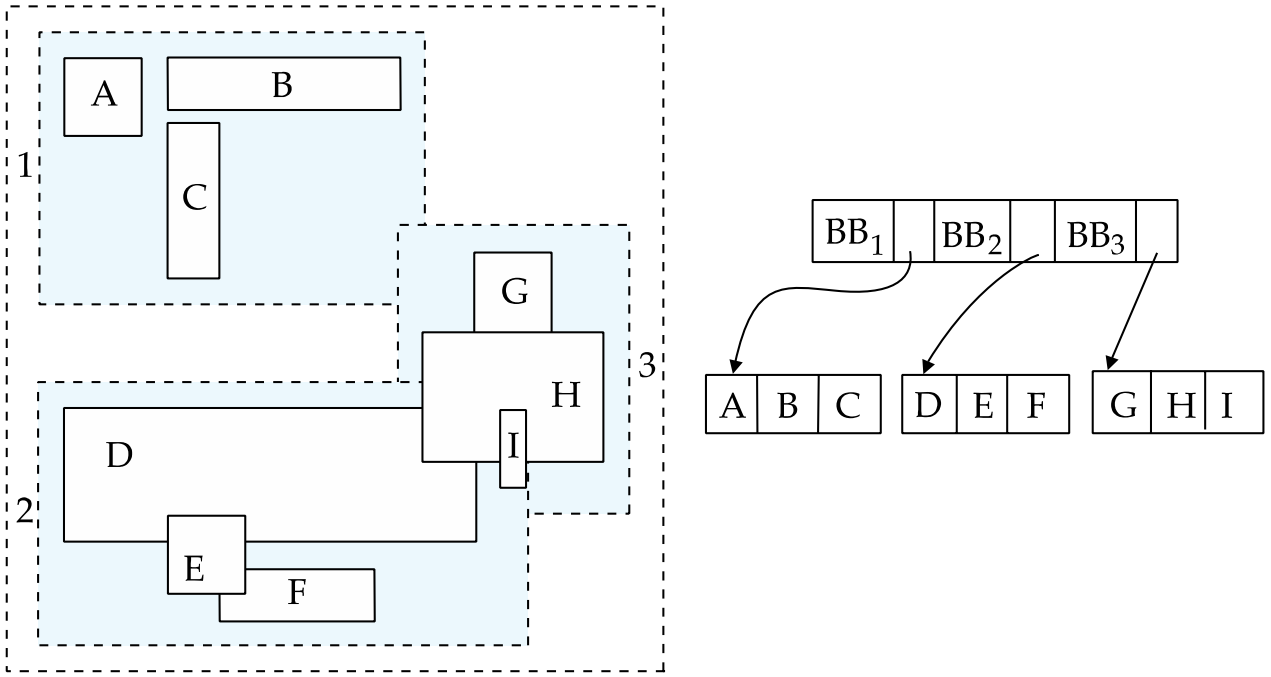


R-Tree



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



Search in R-Trees (Cont.)

- Can be very inefficient in worst case since multiple paths may need to be searched
  - but works acceptably in practice.
- Simple extensions of search procedure to handle predicates *contained-in* and *contains*

Insertion in R-Trees

- To insert a data item:
  - Find a leaf to store it, and add it to the leaf
    - To find leaf, follow a child (if any) whose bounding box contains bounding box of data item, else child whose overlap with data item bounding box is maximum
  - Handle overflows by splits (as in B+ -trees)
    - Split procedure is different though (see below)
  - Adjust bounding boxes starting from the leaf upwards
- Split procedure:
  - Goal: divide entries of an overfull node into two sets such that the bounding boxes have minimum total area
    - This is a heuristic. Alternatives like minimum overlap are possible
  - Finding the "best" split is expensive, use heuristics instead
    - See next slide

Splitting an R-Tree Node

- **Quadratic split** divides the entries in a node into two new nodes as follows
  1. Find pair of entries with "maximum separation"
     - that is, the pair such that the bounding box of the two would has the maximum wasted space (area of bounding box – sum of areas of two entries)
  2. Place these entries in two new nodes
  3. Repeatedly find the entry with "maximum preference" for one of the two new nodes, and assign the entry to that node
     - Preference of an entry to a node is the increase in area of bounding box if the entry is added to the *other* node
  4. Stop when half the entries have been added to one node
     - Then assign remaining entries to the other node
- Cheaper **linear split** heuristic works in time linear in number of entries,
  - Cheaper but generates slightly worse splits.

Deleting in R-Trees

- Deletion of an entry in an R-tree done much like a B^+-tree deletion.
  - In case of underfull node, borrow entries from a sibling if possible, else merging sibling nodes
  - Alternative approach removes all entries from the underfull node, deletes the node, then reinserts all entries

Indexing Temporal Data

- Temporal data refers to data that has an associated time period (interval)
- 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
- 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

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

Figure 7.17 A temporal version of the *course* relation

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

Hashing

Database System Concepts, 7th Ed.

Static Hashing

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

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.

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

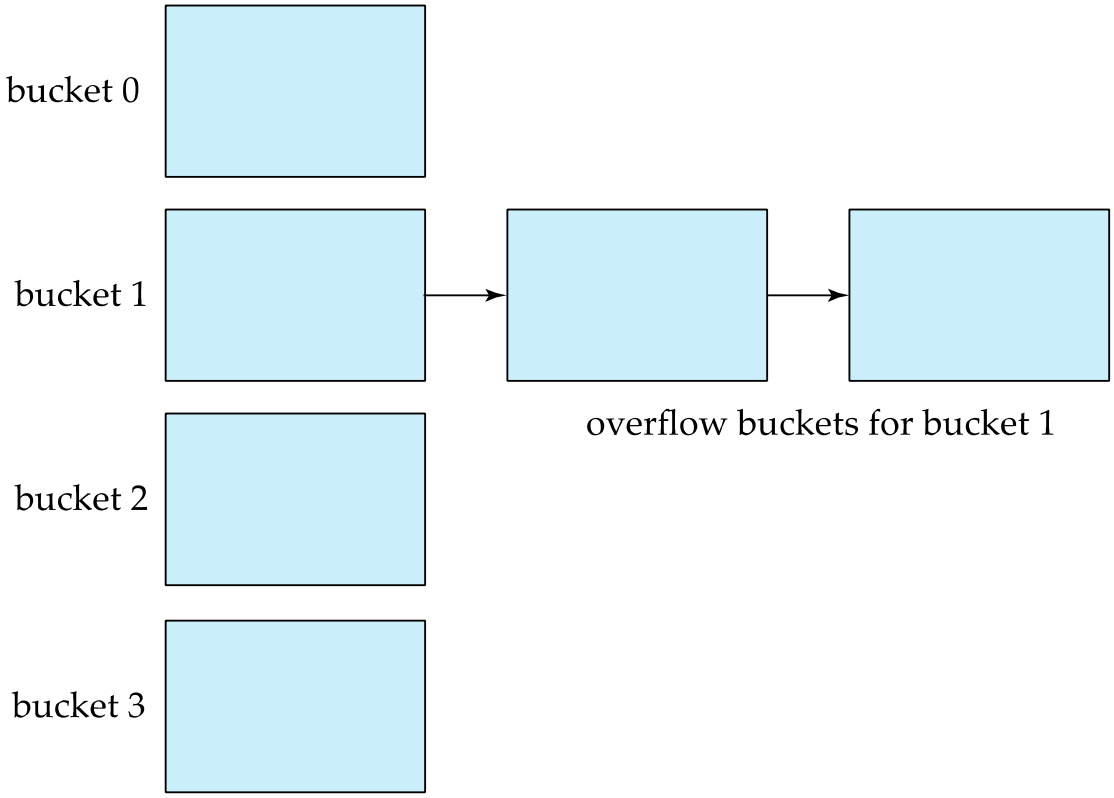
Hash file organization of *instructor* file, using *dept_name* as key.

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 overflow buckets, is not suitable for database applications.



Deficiencies of Static Hashing

- In static hashing, function *h* maps search-key values to a fixed set of *B* of bucket addresses. Databases grow or shrink 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.

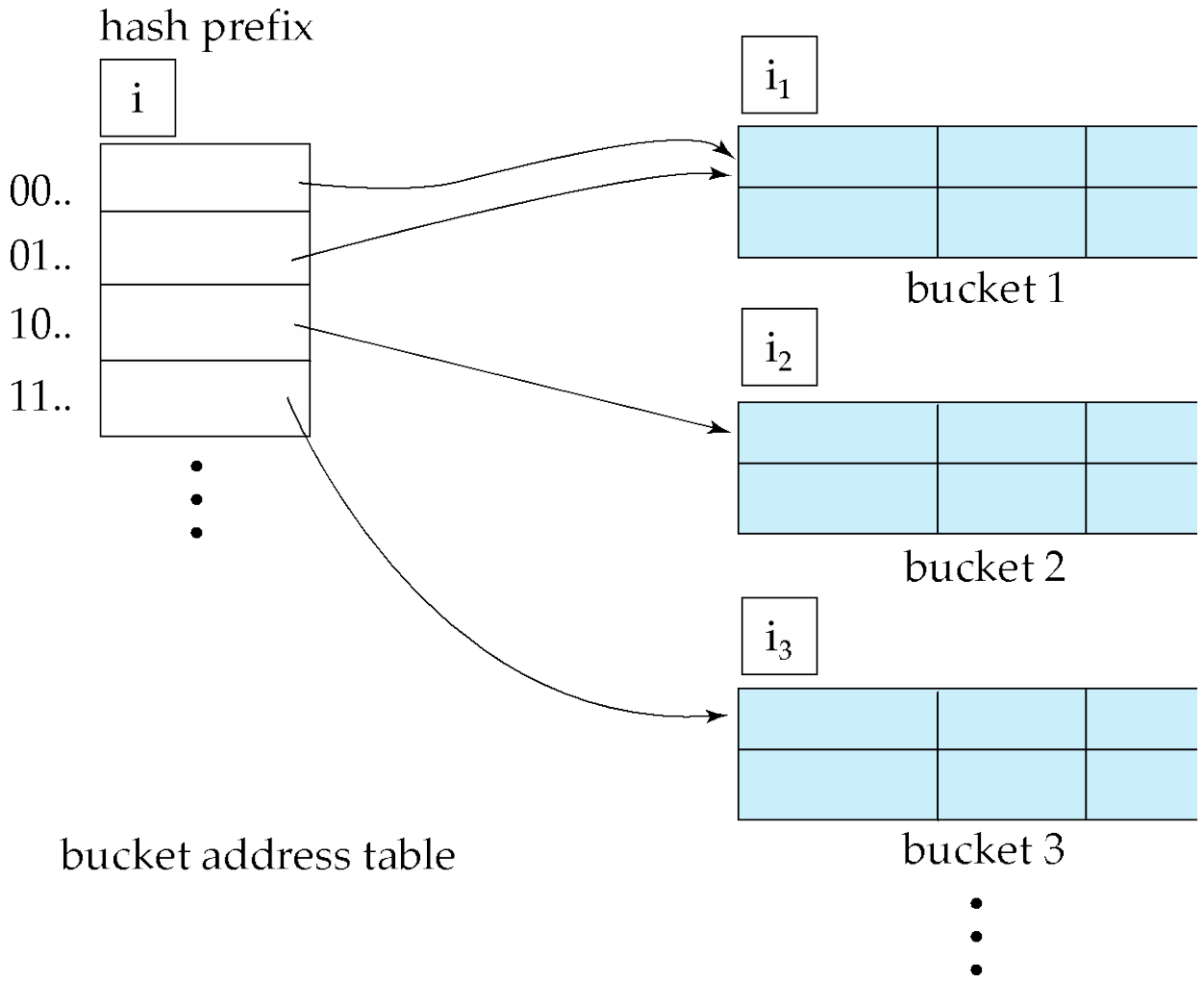
Dynamic Hashing

- 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 # of entries in hash table, without doubling # 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, $0 \le i \le 32$.
    - 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 a bucket (why?)
  - Thus, actual number of buckets is $< 2^i$
    - 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
  - 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 (will see shortly)

Insertion in Extendable Hash Structure (Cont)

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

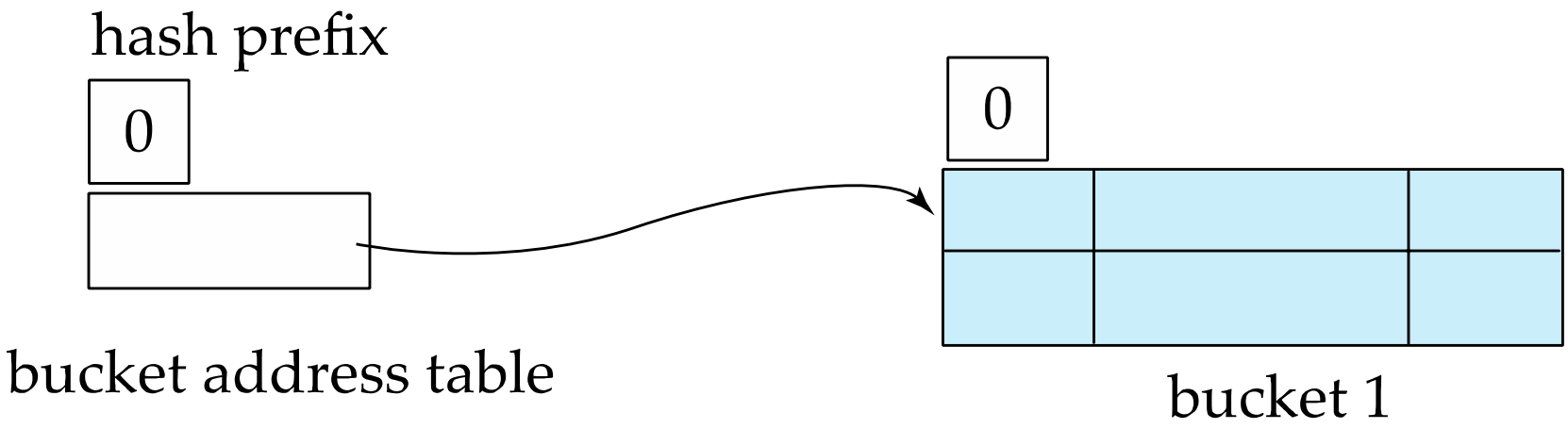
- If $i > i_j$ (more than one pointer to bucket j)
  - allocate a new bucket z, and set $i_j = i_z = (i_j + 1)$
  - Update the second half of the bucket address table entries originally pointing to j, to point to z
  - remove each record in bucket j and reinsert (in j or z)
  - recompute new bucket for 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

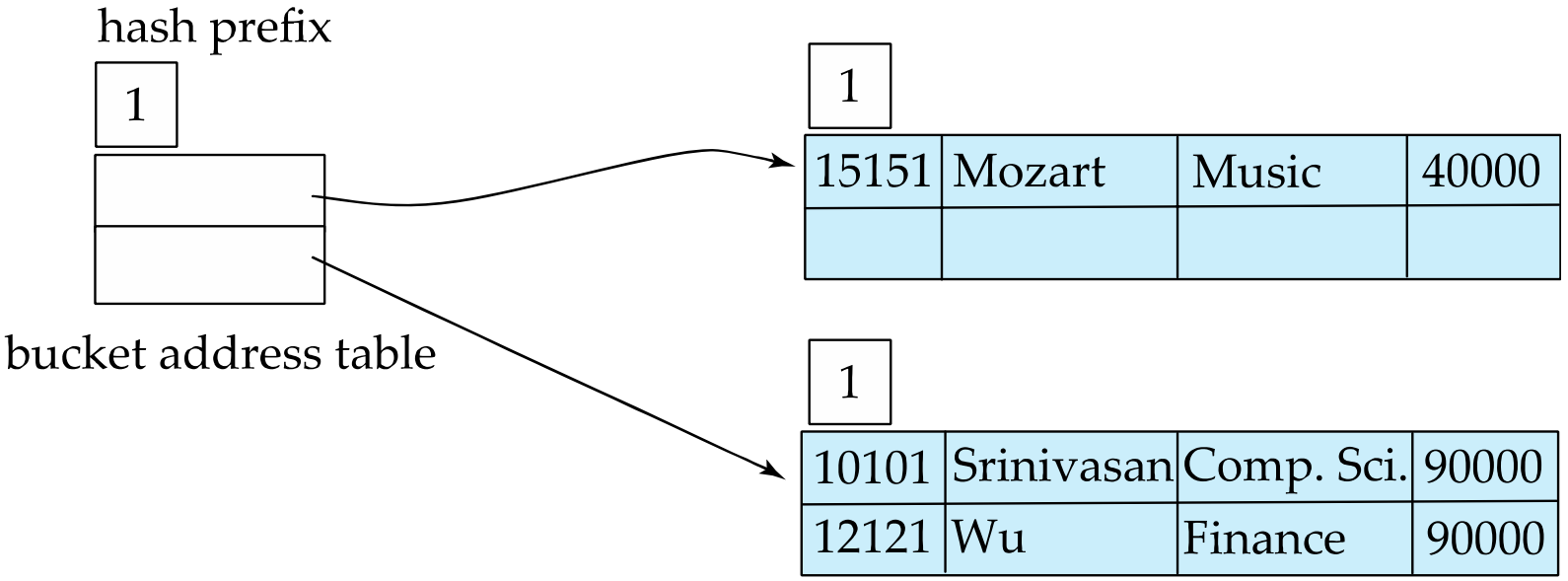
Example (Cont.)

- Initial Hash structure; bucket size = 2



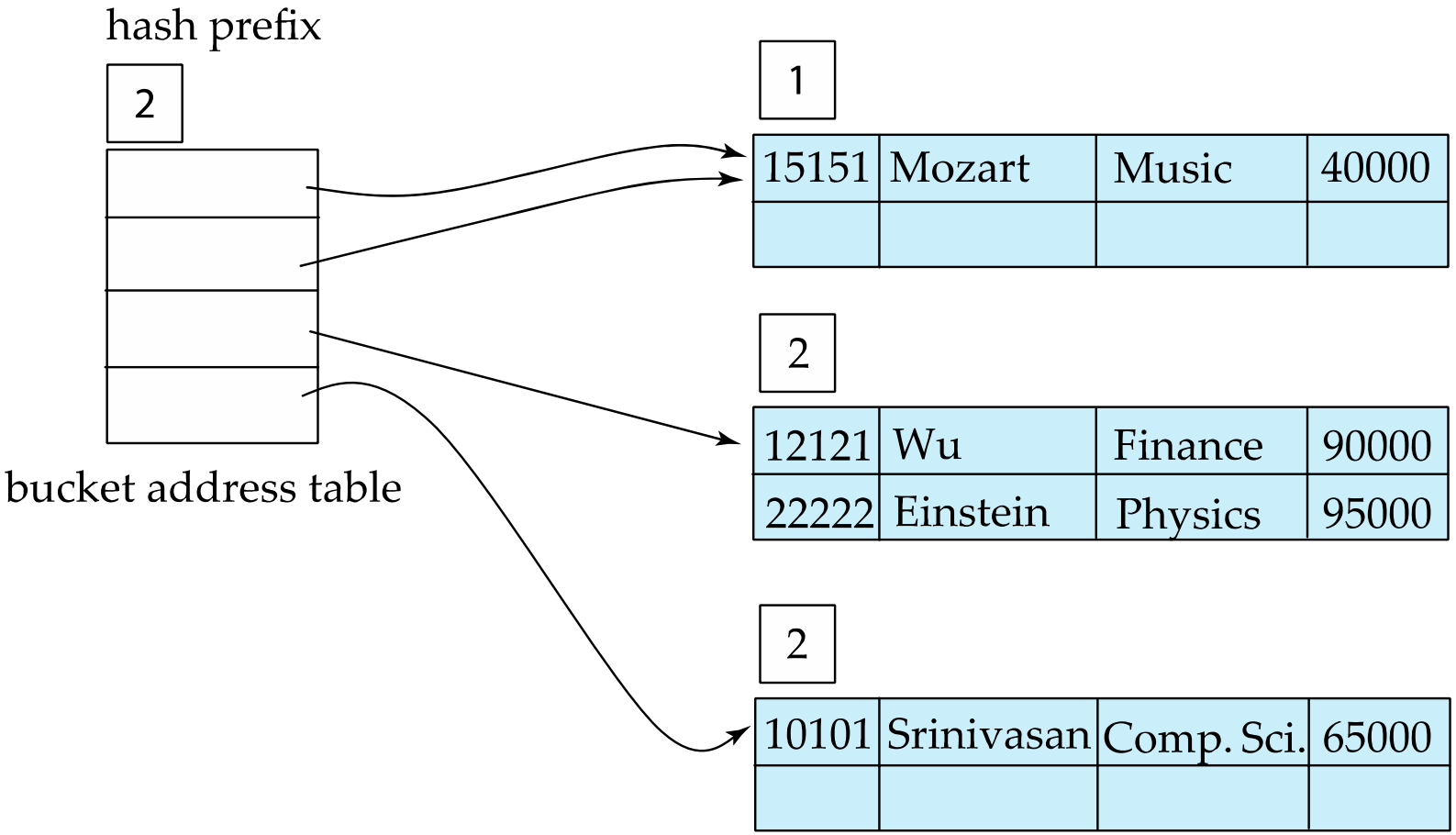
Example (Cont.)

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



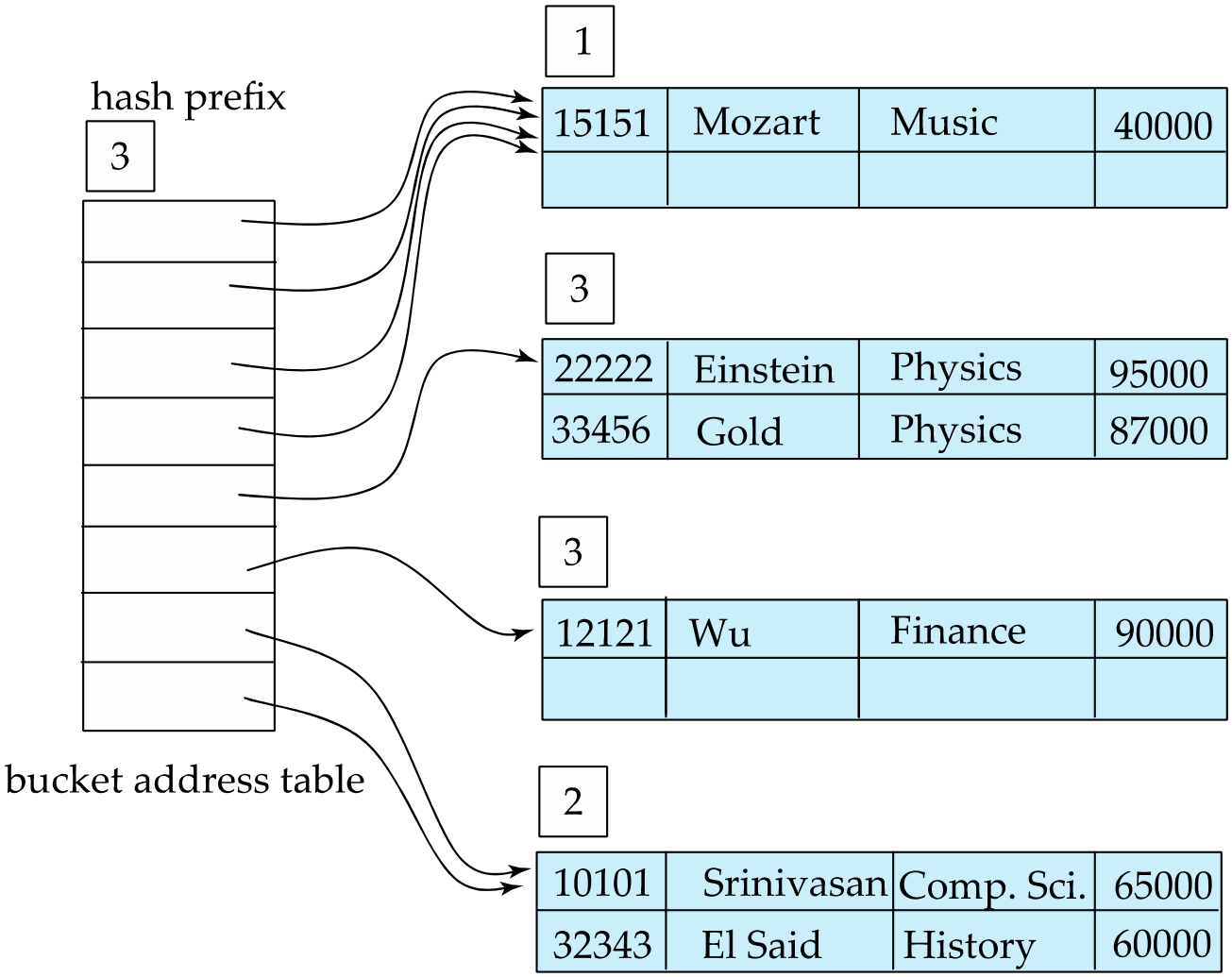
Example (Cont.)

- Hash structure after insertion of Einstein record



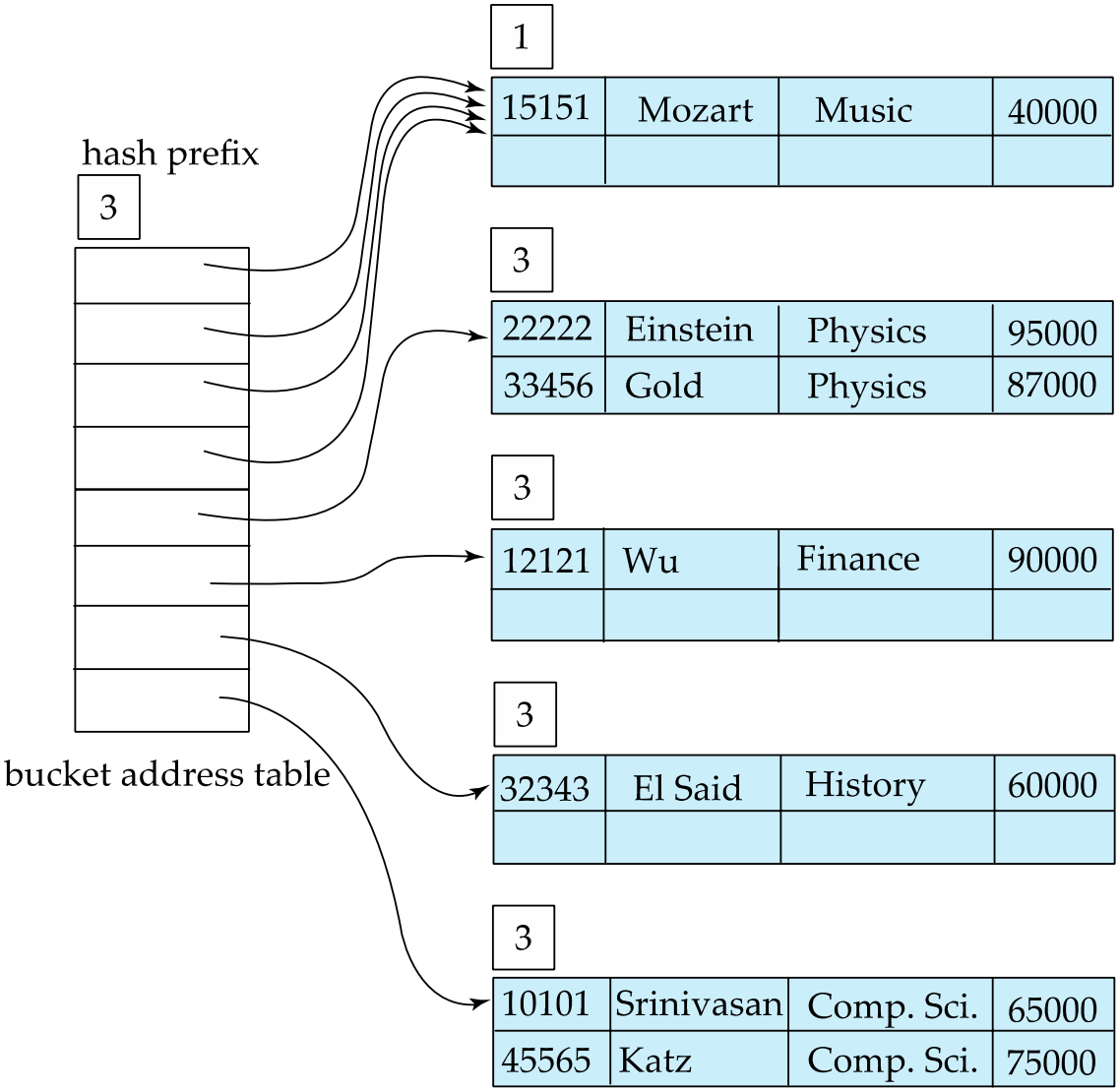
Example (Cont.)

- Hash structure after insertion of Gold and El Said records

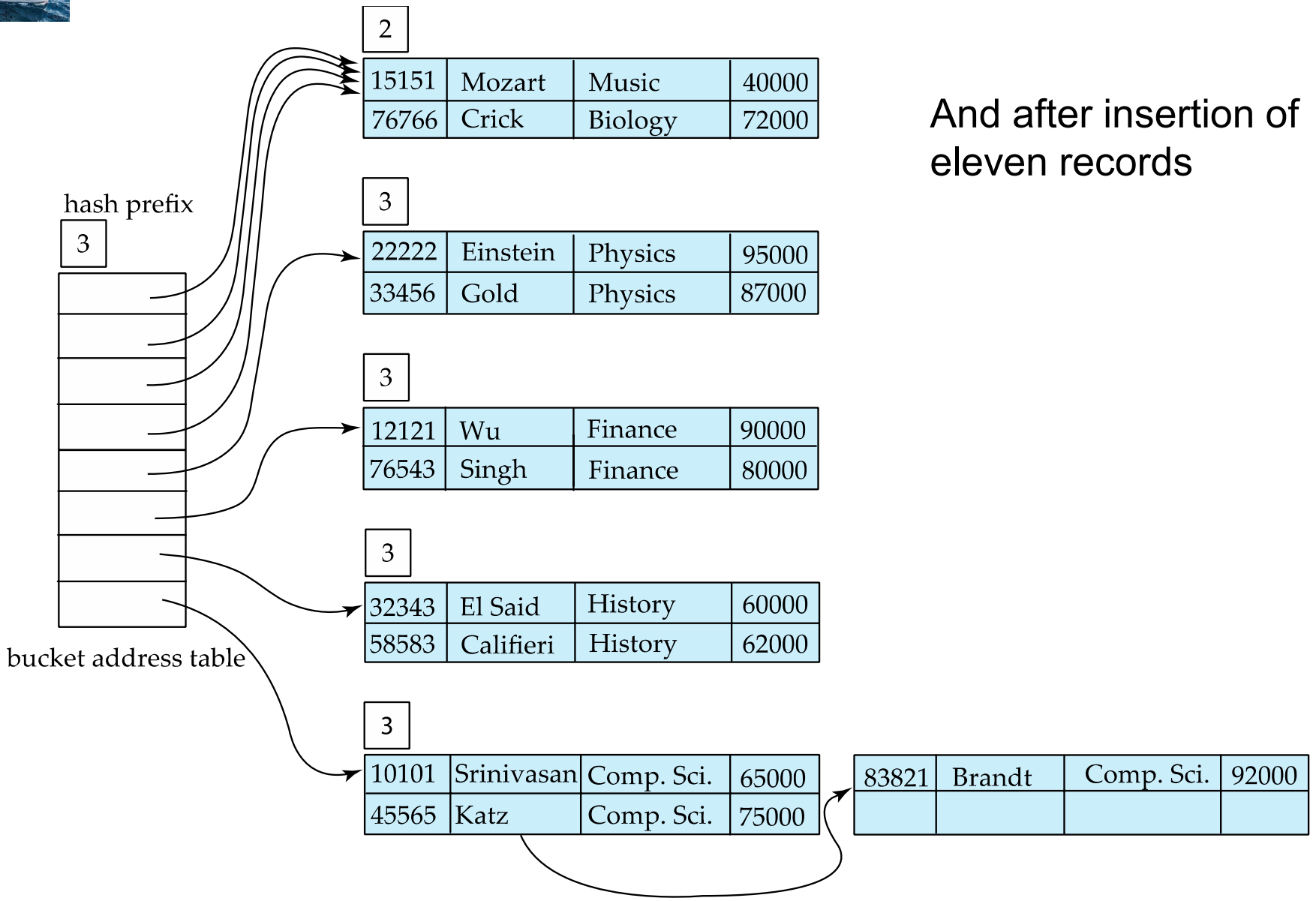


Example (Cont.)

- Hash structure after insertion of Katz record

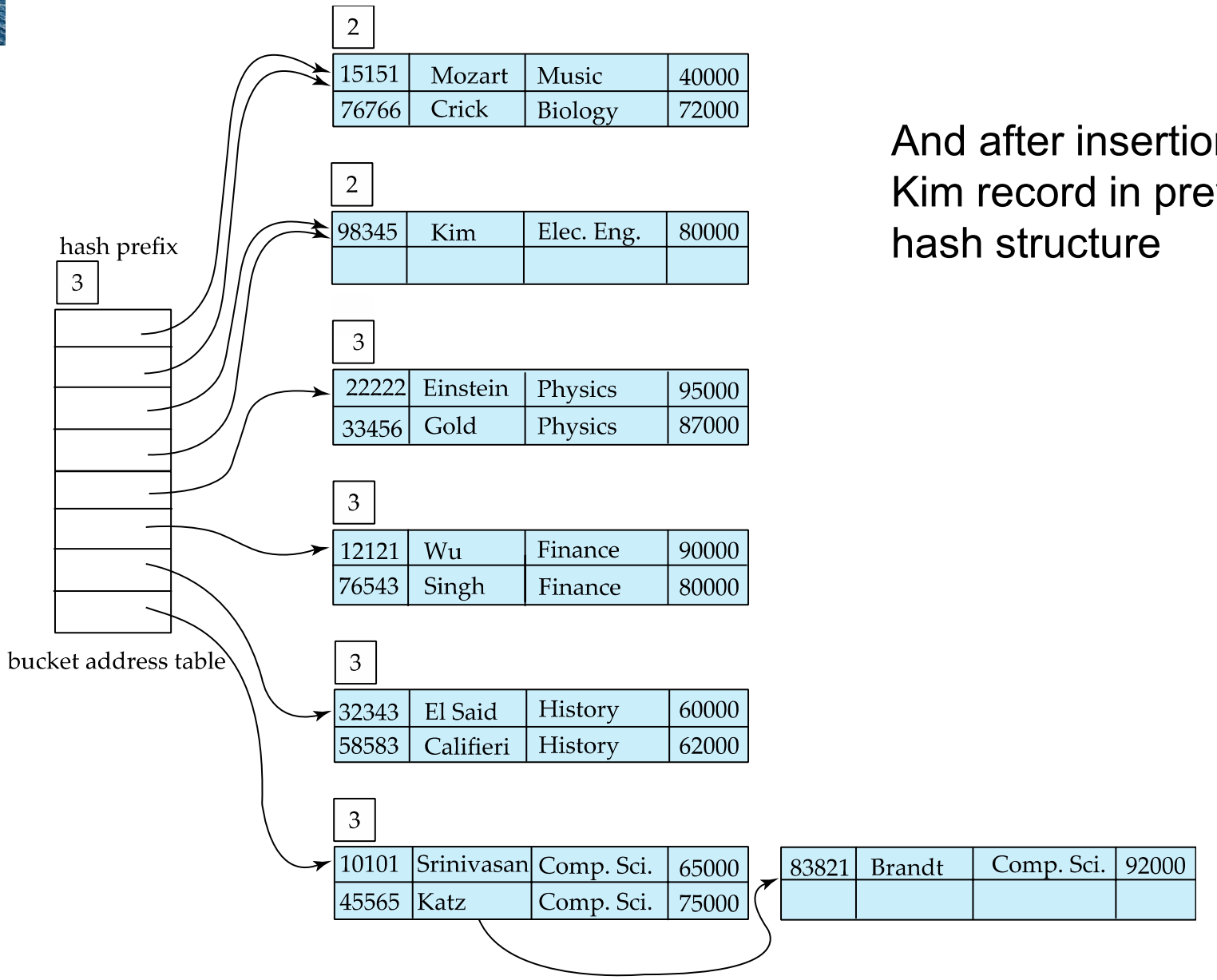


Example (Cont.)



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
  - 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 preferred
- In practice:
  - Hash-indices are extensively used in-memory
  - But not used much on fisk
    - Oracle supports static hash organization, but not hash indices
    - SQL Server and PostgreSQL do not support hashing on disk

End of Chapter 24

Partitioned Hashing

- Hash values are split into segments that depend on each attribute of the search-key.

  $(A_1, A_2, \ldots, A_n)$ for n attribute search-key

- Example: $n = 2$, for *customer,* search-key being (*customer-street, customer-city*)

search-key value	*hash value*
(Main, Harrison)	101 111
(Main, Brooklyn)	101 001
(Park, Palo Alto)	010 010
(Spring, Brooklyn)	001 001
(Alma, Palo Alto)	110 010

- To answer equality query on single attribute, need to look up multiple buckets. Similar in effect to grid files.