Databases and Information Systems
CS303

Hashing

18-10-2023

Static Hashing

- File organizations based on the technique of hashing avoids accessing an index structure
- Bucket to denote a unit of storage that can store one or more records.
- Let K be the set of all search-keys and B be the set of all buckets. Hashing is a function h that maps every search-key to a bucket ($h: K \rightarrow B$)

Static Hashing

- To insert a record with search key K_i first compute $h(K_i)$ to get the address of the bucket.
  - If there is space in the bucket, the record is stored in that bucket
- To delete a record with search key K_i compute $h(K_i)$ to get the address of the bucket, go to the bucket, search the record to delete and delete it
- To search a record with search key K_i first compute $h(K_i)$ to get the address of the bucket.
  - Go to the bucket and find all records with search key K_i
  - There could be records with other search key too

Static Hashing : Hash functions

- Worst possible hash function maps all search-key values to the same bucket

- Ideal hash function distributes the stored keys uniformly across all the buckets
  - So that every bucket has the same number of records.

- Choose hash functions such that:
  - Distribution is uniform: Hash function assigns each bucket the same number of search-key values from the set of all possible search-key values.

  - Distribution is random: In the average case, each bucket will have nearly the same number of values assigned to it, regardless of how keys are distributed
    - The hash value will not be correlated to any externally visible ordering on the search-key values, such as alphabetic ordering or ordering by the length of the search keys; the hash function should appear to be random.

Static Hashing : Hash functions

- Consider the 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

- Have 26 buckets for dept_name
  - Put a record in i-th b
    i-th alphabet
  - Bad because there
    with S, B .. com
- Have 10 buckets for salary
  - If minimum salary is 30000 max is 130000
  - Put a record in i-th b
    30000+(i*10000) to 30000 + (i+1)*10000
  - Bad because mean salary bucket will have more records

Static Hashing : Hash functions

- Typical hashing function performs some computation on the binary representation of the string

- Example: Sum of the characters in the name, assuming i-th character has value i

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

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

Static Hashing : Hash functions

- Hash functions require careful design.
- A bad hash function may result in lookup taking time proportional to the number of search keys in the file.
- A well-designed function gives an average-case lookup time that is a (small) constant, independent of the number of search keys in the file.

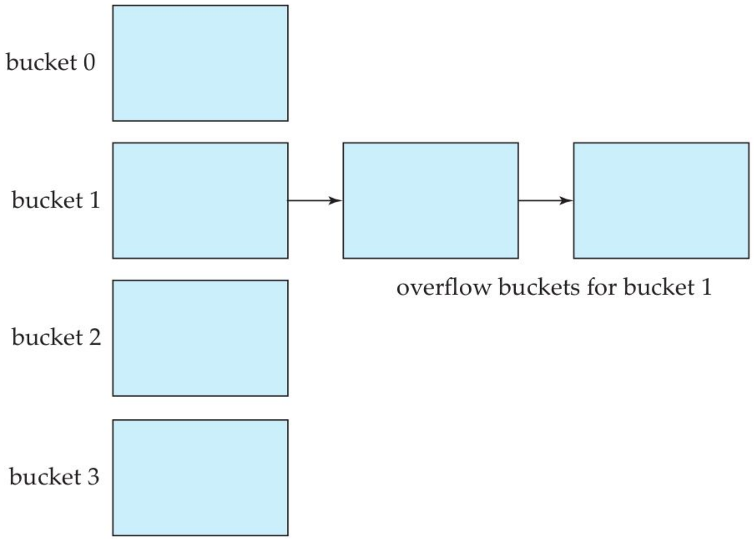
Static Hashing : Hash functions

- Handling Bucket Overflows: Bucket full when inserting a record
- Can occur for several reasons
  - Insufficient buckets : The number of buckets, which we denote n_B , must be chosen such that:
    - $n_B > n_r / f_r$,
      - n_r denotes the total number of records that will be stored
      - f_r denotes the number of records that will fit in a bucket.
    - To reduce probability of bucket overflow, n_B is chosen to be $(n_r/f_r)*(1+d)$ where d is fudge-factor (typically 0.2)
    - Not always possible to know the total number of records when the hash function is chosen
  - Skew : Some buckets are assigned more records than are others
    - Multiple records may have the same search key
    - Chosen hash function may result in nonuniform distribution of search keys.

Static Hashing : Hash functions

- Closed Hashing:
  If a bucket is already full, system provides an overflow bucket and the new record is added into the overflow bucket

- All overflow buckets are chained as a linked list



Static Hashing : Hash functions

- Open Hashing:
  If a bucket is full, the system inserts records in some other bucket
- One policy is to use the next bucket (in cyclic order) that has space
  - linear probing

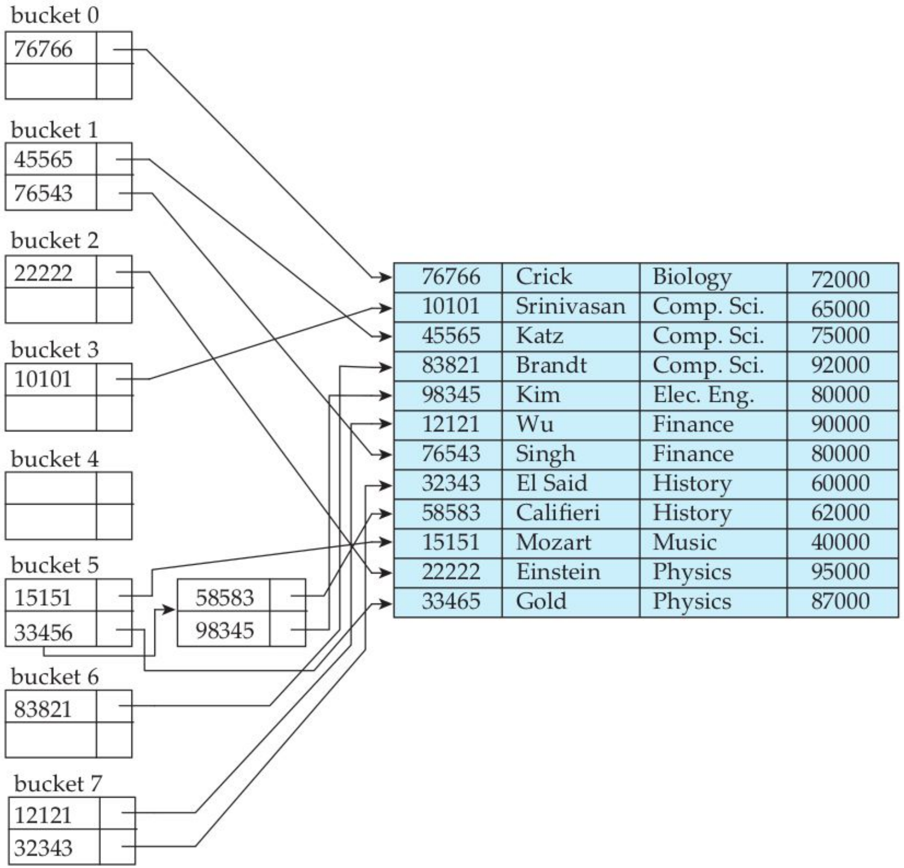
Static Hashing : Hash functions

- For Databases, Closed hashing is preferable.
- Drawback:
  - Hashing function should be chosen carefully at the beginning
  - Cannot be changed later on
  - Cannot handle varying size of databases (number of buckets is fixed)

Static Hashing : Hash indices

- Hashing can also be used to store search-key index structure

- A hash index is never needed as a clustering index structure
  - if a file itself is organized by hashing, there is no need for a separate hash index structure on it.

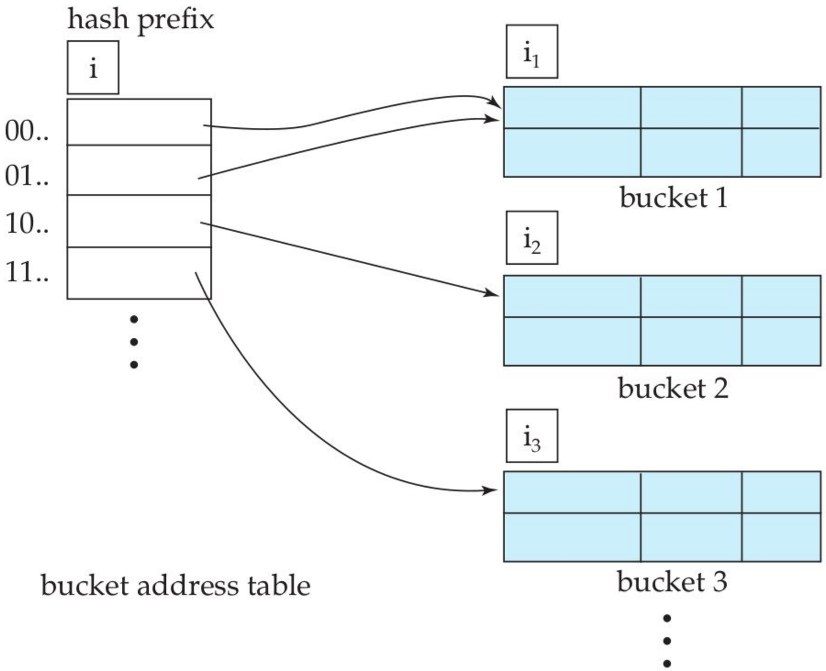


Dynamic Hashing

- The need to fix the number of buckets at the beginning is bad.
  - Most database size changes over time
- If we want to use static hashing:
  - Choose a hash function based on the current file size.
    - Cannot handle larger databases in the future
  - Choose a hash function based on the anticipated size of the file at some point in the future.
    - Significant amount of space may be wasted initially.
  - Periodically reorganize the hash structure in response to file growth.
    - 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.
- Dynamic hashing allows the hash function to be modified dynamically to accommodate the growth or shrinkage of the database.

Dynamic Hashing

- Data Structure:
  - Should be able to split / merge buckets
  - Create buckets on demand, as records are inserted into the file
  - Image of a typical hash function is a 32 bit number
  - At any point, we use i bit for hashing where $0 \le i \le b$.
    - These i bits are used as an offset into an additional table of bucket addresses.
  - The value of i grows and shrinks with the size of the database.



Dynamic Hashing

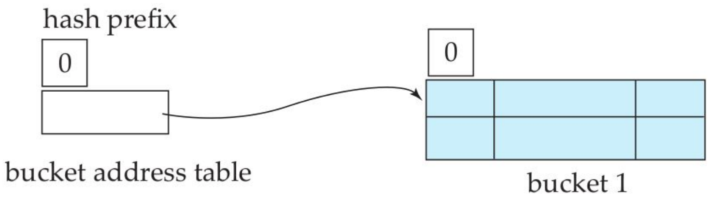
- Querying: To locate the bucket containing search-key value K_a
  - Compute the first i high-order bits of $h(K_a)$
  - Look at the corresponding table entry for this bit string
  - Follow the bucket pointer in the table entry.

Dynamic Hashing

- Inserting: Example
- Assume bucket can hold only 2 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

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	



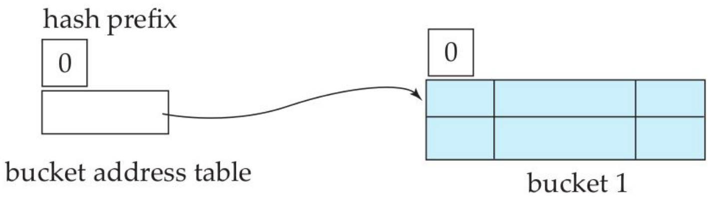
- i is the number of bits considered by bucket address table
- i_j is the number of bits considered by bucket j
- Inserting first two records will go into Bucket 1

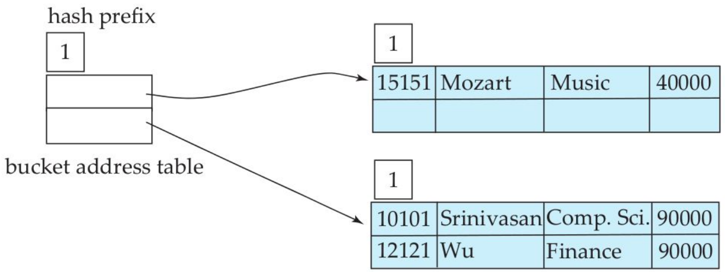
Dynamic Hashing

- Insert Mozart ($i = i_0 = 0$)
- Increase i to 1
- Create new bucket and rehash

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



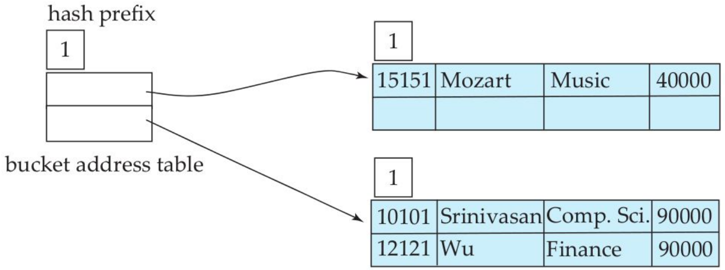


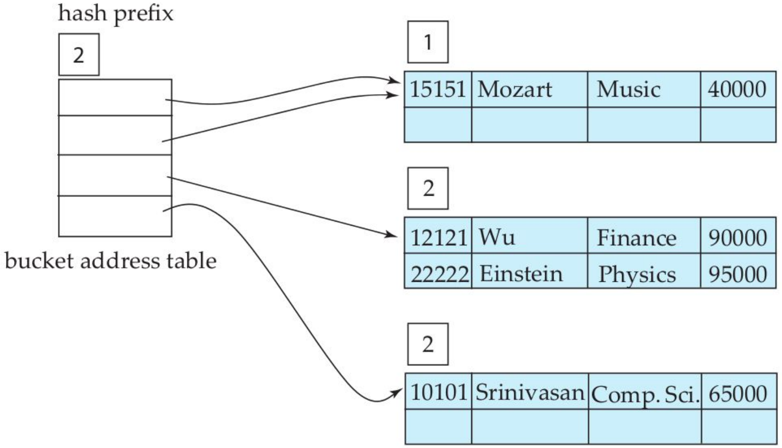
Dynamic Hashing

- Insert Einstein ($i = i_1 = 1$)
- Increase i to 2
- Create new bucket and rehash

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

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

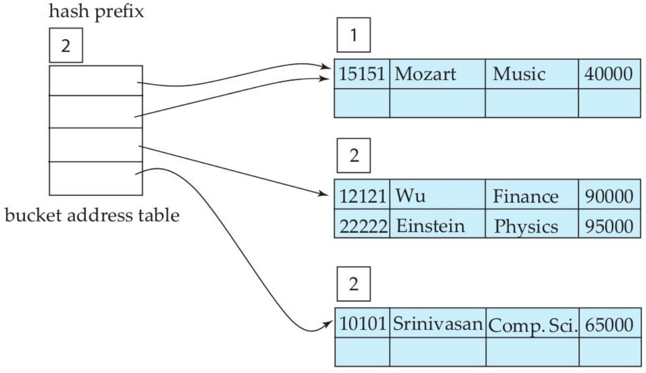




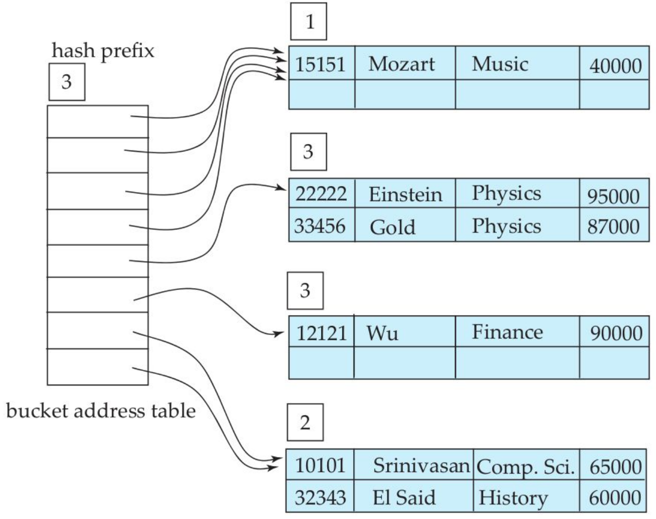
Dynamic Hashing

- Insert El Said (No overflow)
- Insert Gold
- Increase i to 3

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	

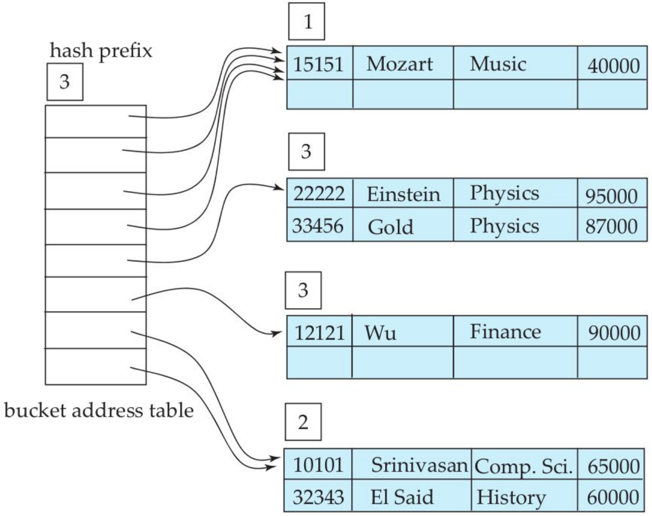


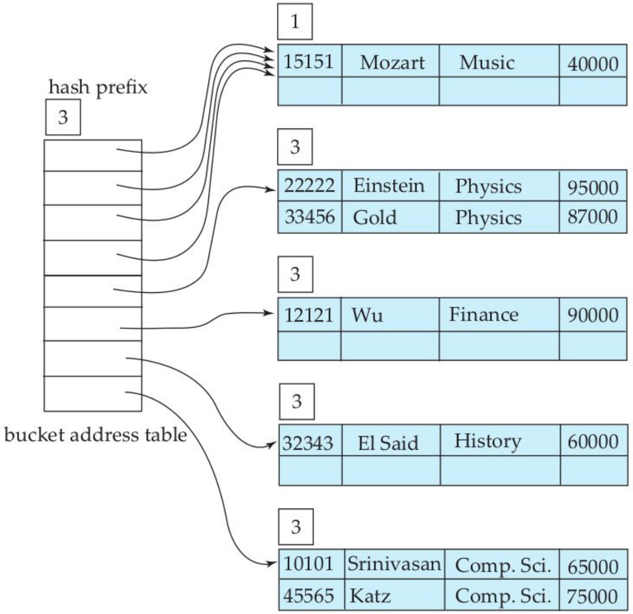
Dynamic Hashing

- Insert Katz
- No increase in i

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

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



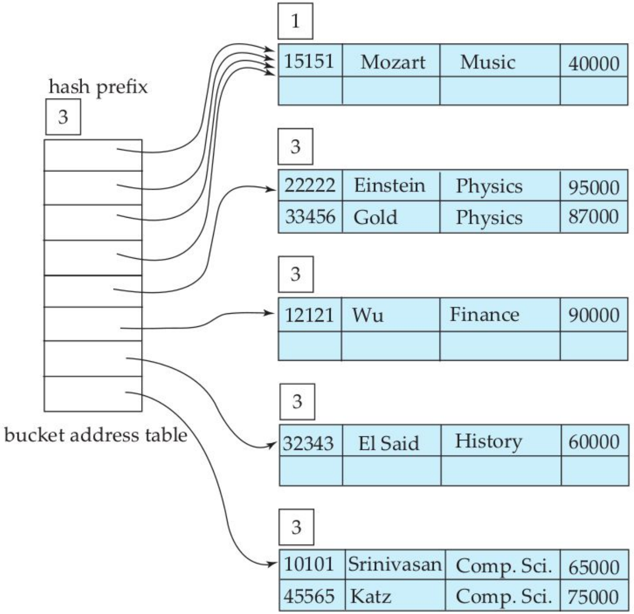


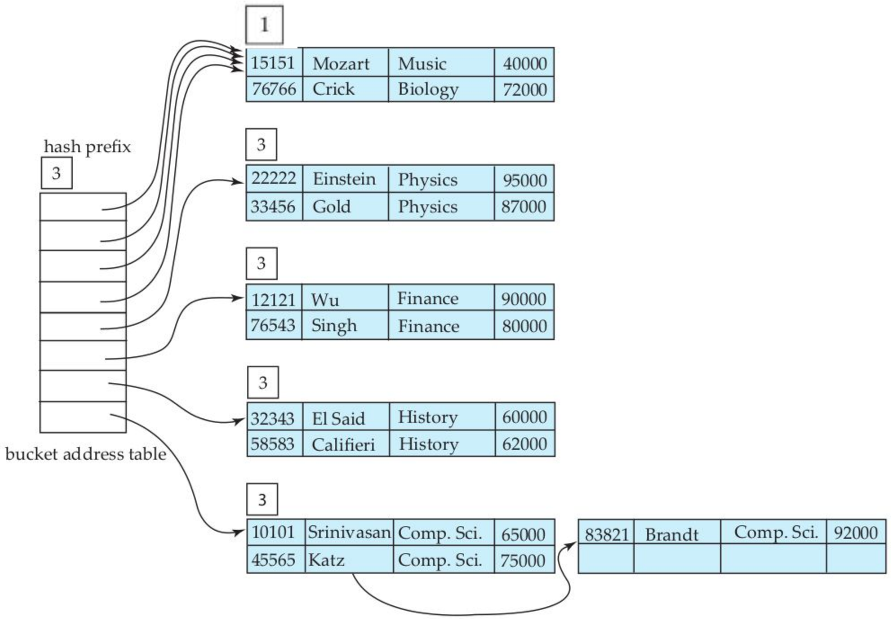
Dynamic Hashing

- Insert Califieri, Singh, Crick
  (No overflow)
- Insert Brandt
  (Cannot be handled by splitting)

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

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



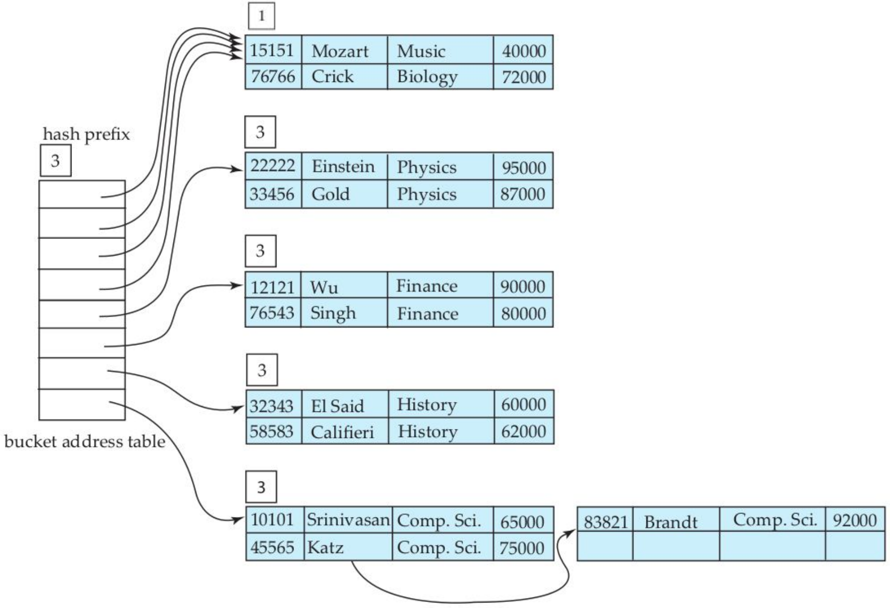


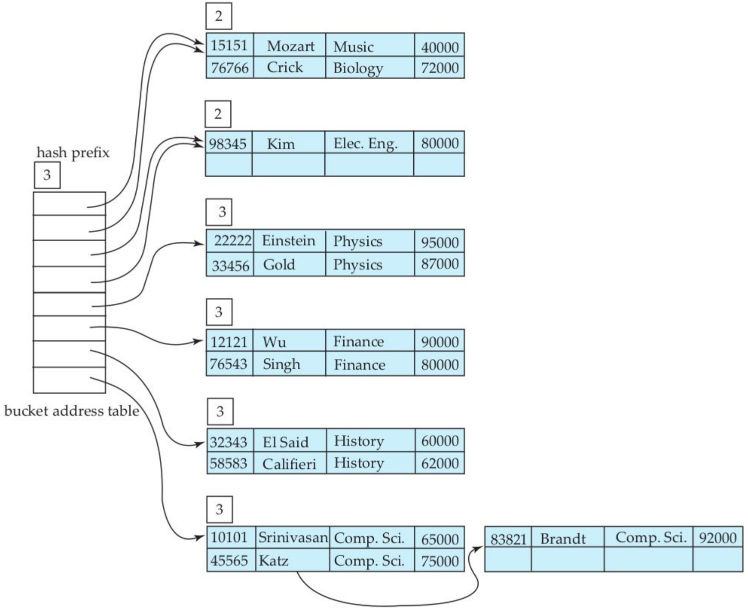
Dynamic Hashing

- Insert Kim

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

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





Dynamic Hashing

- Inserting: Suppose inserting a record with key K_a leads to bucket j
  - If bucket j has some space, insert the new record in bucket j
  - Else : Should split the bucket and decide if the number of bits used should be increased
    - If $i = i_j$ (Increase the size of the bucket address table).
      - Increment the value of i by 1 (doubling the size of the bucket address table)
      - Replace each entry by two entries in bucket address table, both of which contain the same pointer as the original entry (two entries in the bucket address table point to bucket j)
      - Allocate a new bucket (bucket z), and set the second entry to point to the new bucket. Set i_j and i_z to i
      - Rehash each record in bucket j and, depending on the first i bits (now i is increased by 1 and put it in bucket j or bucket z)
      - Now insert the new element (generally it succeeds)
        - If it fails again, then repeat the steps again
      - If all records in bucket j have same search-key then no amount of splitting will help
        - Create overflow buckets

Dynamic Hashing

- Inserting: Suppose inserting a record with key K_a leads to bucket j
  - If bucket j has some space, insert the new record in bucket j
  - Else : Should split the bucket and decide if the number of bits used should be increased
    - If $i = i_j$ (Increase the size of the bucket address table).
    - If $i > i_j$ (Split can be done without increasing i)
      - Create new bucket z and assign i_j and i_z to old $i_j + 1$
      - Among the entries in the bucket address table that mapped to bucket j
        - Leave the first half as it is, Update the second half to point to bucket z
      - Rehash the records in bucket j and distribute them to bucket j and bucket z
      - Now attempt to reinsert (will most likely succeed)
        - If it fails, repeat depending on whether $i = i_j$ or $i > i_j$
- In both cases, we recompute hashing only for records in a single bucket

Dynamic Hashing

- Deleting: To delete a record with search-key value K_a :
  - Suppose record is in bucket—say, j
  - Remove both the search key from the bucket and the record from the file
  - The bucket is removed if it becomes empty
    - If several buckets can be merged, and the size of the bucket address table can be cut in half
      - How to detect and implement the merge ? Exercise
- Detect when can the bucket address table can be reduced in size?
  - Exercise
- Changing the size of the bucket address table is expensive
  - Performed only if the result reduces the number of buckets significantly

Ordered Indexing V/s Hashing

- Should we store a file as
  - Index sequential organized
  - As a B^+ tree
  - As one of the Hashing
- Depends on the following factors:
  - Is the cost of periodic reorganization of the index or hash organization acceptable?
  - What is the relative frequency of insertion and deletion?
  - Is it desirable to optimize average access time at the expense of increasing the worst-case access time?
  - What types of queries are users likely to pose?
- Most implementations use B^+ trees

Ordered Indexing V/s Hashing

- SELECT $A_1.....A_n$ FROM r WHERE $A_m = c$
  - Hashing is preferable
- SELECT $A_1.....A_n$ FROM r WHERE $A_m >= c_1$ AND $A_m <= c_2$
  - Ordered indexing is preferable
- Usually the designer will choose ordered indexing
  - Unless it is known in advance that range queries will be infrequent
    - Then hashing would be chosen
- Hash organizations are useful for temporary files created during query processing
  - if lookups based on a key value are required, but no range queries will be performed
    - Example : Computing Natural Joins

Bitmap indices

- Specialized type of index designed for easy querying on multiple keys,
  - Although each bitmap index is built on a single key
  - Array of bits

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

- select * from r where gender = ‘f’ and income level = ‘L_2’;
  - Take bitwise intersection of f and L_2
- Can also be used to count the number of tuples satisfying some conditions

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

- Deletion of records creates holes in the bitmap
  - Shifting the records is expensive
  - Create a new Existence bitmap
- Insert can be at the end or in the place of a deleted record

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

- Implementing Bitmap operations:
  - Bitmap Intersection can be done using for loop
    - More efficient to use bitwise AND (can handle 32 or 64 bits at once)

  - Bitmap union computes 'OR' operation
  - Bitmap complement computes NOT operation
    - How to do complement in presence of existence bitmap ? (Exercise)

  - Counting 1's in a bitmap:
    - One for loop
    - Better way: Maintain an array of length 256
      - i-th index stores the number of 1's in the binary representation of i
      - Take each byte of bitmap and hash it to the index (via identity)
        - Initially total count to 0 and keep adding the hashed value
        - Essentially 8 bits are processed together

Defining index in SQL

- The SQL standard does not provide any way for the database user or administrator to control what indices are created and maintained in the database system.
- Indices are not required for correctness, since they are redundant data structures
- Database system automatically decides what indices to create.
    - Not easy to automate the right choice of index
- So most SQL implementations provide the programmer control over creation and removal of indices

Defining index in SQL

- CREATE INDEX <index-name> ON <relation-name> (<attribute-list>);
  - CREATE INDEX dept_index ON instructor (dept_name);
  - If <attribute_list> is a candidate key
    - CREATE UNIQUE INDEX dept_index ON instructor (dept_name);
    - Displays error if dept_name is not a candidate key in instructor table

- Some database systems also provide a way to specify the type of index to be used (such as B+ -tree or hashing)

- DROP INDEX <index-name>;

Reference:

Database System Concepts by Silberschatz, Korth and Sudarshan

(6th edition)

Chapter 11