4.1
Clustered Indexes

Definition - What does *Clustered Index* mean?

A clustered index is a type of index where the table records are physically re-ordered to match the index.

Clustered indexes are efficient on columns that are searched for a range of values. After the row with first value is found using a clustered index, rows with subsequent index values are guaranteed to be physically adjacent, thus providing faster access for a user query or an application.

In other words, a clustered index stores the actual data, where a non-clustered index is a pointer to the data. In most DBMSs, you can only have one clustered index per table, though there are systems that support multiple clusters (DB2 being an example).

Like a regular index that is stored unsorted in a database table, a clustered index can be a composite index, such as a concatenation of first name and last name in a table of personal information.

4.2, 4.3

Indexing in Databases

Indexing is a way to optimize performance of a database by minimizing the number of disk accesses required when a query is processed.

An index or database index is a data structure which is used to quickly locate and

access the data in a database table.

Indexes are created using some database columns.

- The first column is the Search key that contains a copy of the primary key or candidate key of the table. These values are stored in sorted order so that the corresponding data can be accessed quickly (Note that the data may or may not be stored in sorted order).
- The second column is the Data Reference which contains a set of pointers holding the address of the disk block where that particular key value can be found.

Search Key	Data Reference

Structure of an index

There are two kinds of indices:

1. **Ordered indices:** Indices are based on a sorted ordering of the values.
2. **Hash indices:** Indices are based on the values being distributed uniformly across a range of buckets. The buckets to which a value is assigned is determined by function called a hash function.

There is no comparison between both the techniques, it depends on the database application on which it is being applied.

- **Access Types**: e.g. value based search, range access, etc.
- **Access Time**: Time to find particular data element or set of elements.
- **Insertion Time**: Time taken to find the appropriate space and insert a new data time.
- **Deletion Time**: Time taken to find an item and delete it as well as update the index structure.
- **Space Overhead**: Additional space required by the index.

Indexing Methods

Ordered Indices

The indices are usually sorted so that the searching is faster. The indices which are sorted are known as ordered indices.

- If the search key of any index specifies same order as the sequential order of the file, it is known as primary index or clustering index. **Note:** The search key of a primary index is usually the primary key, but it is not necessarily so.
- If the search key of any index specifies an order different from the sequential order of the file, it is called the secondary index or non-clustering index.

Clustered Indexing

Clustering index is defined on an ordered data file. The data file is ordered on a non-key field. In some cases, the index is created on non-primary key columns which may not be unique for each record. In such cases, in order to identify the records faster, we will

group two or more columns together to get the unique values and create index out of them. This method is known as clustering index. Basically, records with similar characteristics are grouped together and indexes are created for these groups.

For example, students studying in each semester are grouped together. i.e. 1st Semester students, 2nd semester students, 3rd semester students etc are grouped.

INDEX FILE	
SEMESTER	**INDEX ADDRESS**
1	
2	
3	
4	
5	

Data Blocks in Memory				
100	Joseph	Alaiedon Township	20	200
101				
110	Allen	Fraser Township	20	200
111				
120	Chris	Clinton Township	21	200
121				
200	Patty	Troy	22	205
201				
210	Jack	Fraser Township	21	202
211				
300				

Clustered index sorted according to first name (Search key)
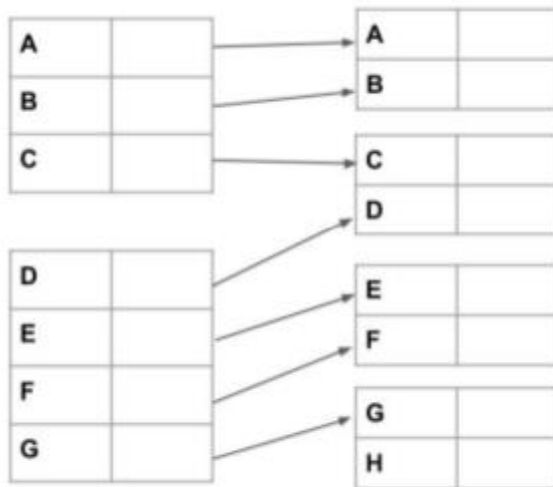## Primary Index

In this case, the data is sorted according to the search key. It induces sequential file organisation.

In this case, the primary key of the database table is used to create the index. As primary keys are unique and are stored in sorted manner, the performance of searching operation is quite efficient. The primary index is classified into two types : **Dense Index** and **Sparse Index**.
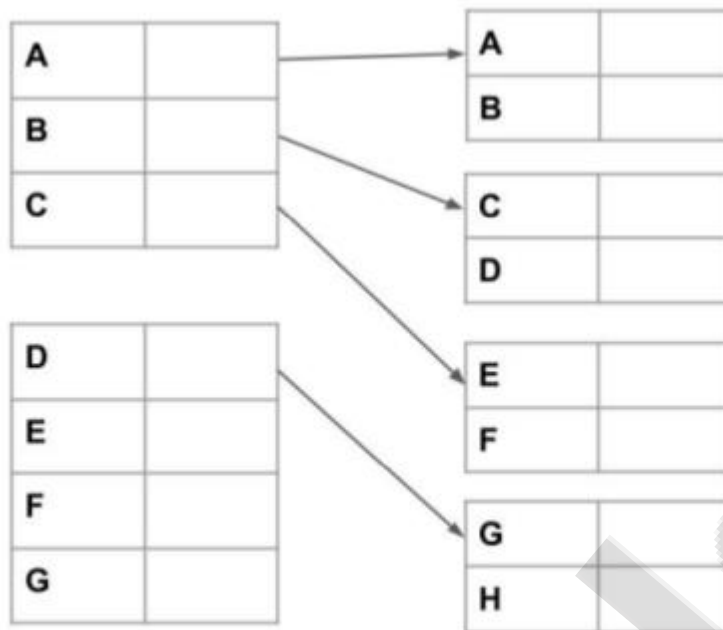
(I) Dense Index :

- For every search key value in the data file, there is an index record.
- This record contains the search key and also a reference to the first data record with that search key value.

Dense Index
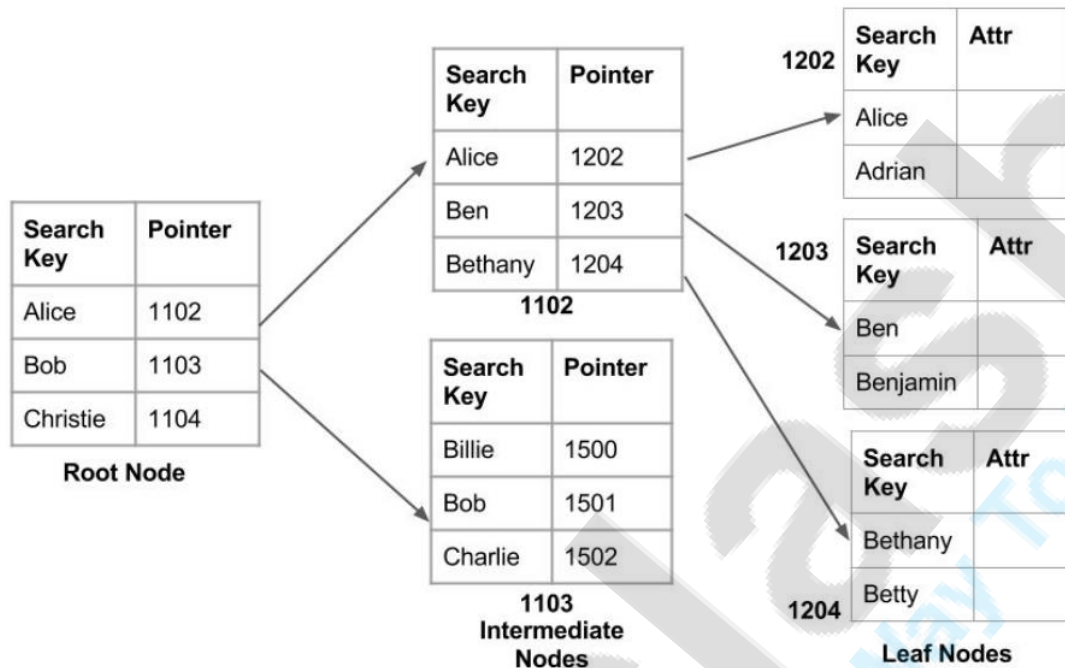
(II) Sparse Index :

- The index record appears only for a few items in the data file. Each item points to a block as shown.
- To locate a record, we find the index record with the largest search key value less than or equal to the search key value we are looking for.
- We start at that record pointed to by the index record, and proceed along the pointers in the file (that is, sequentially) until we find the desired record.

Sparse Index

Non-Clustered Indexing

A non clustered index just tells us where the data lies, i.e. it gives us a list of virtual pointers or references to the location where the data is actually stored. Data is not physically stored in the order of the index. Instead , data is present in leaf nodes. For eg. the contents page of a book. Each entry gives us the page number or location of the information stored. The actual data here(information on each page of book) is not organised but we have an ordered reference(contents page) to where the data points actually lie.
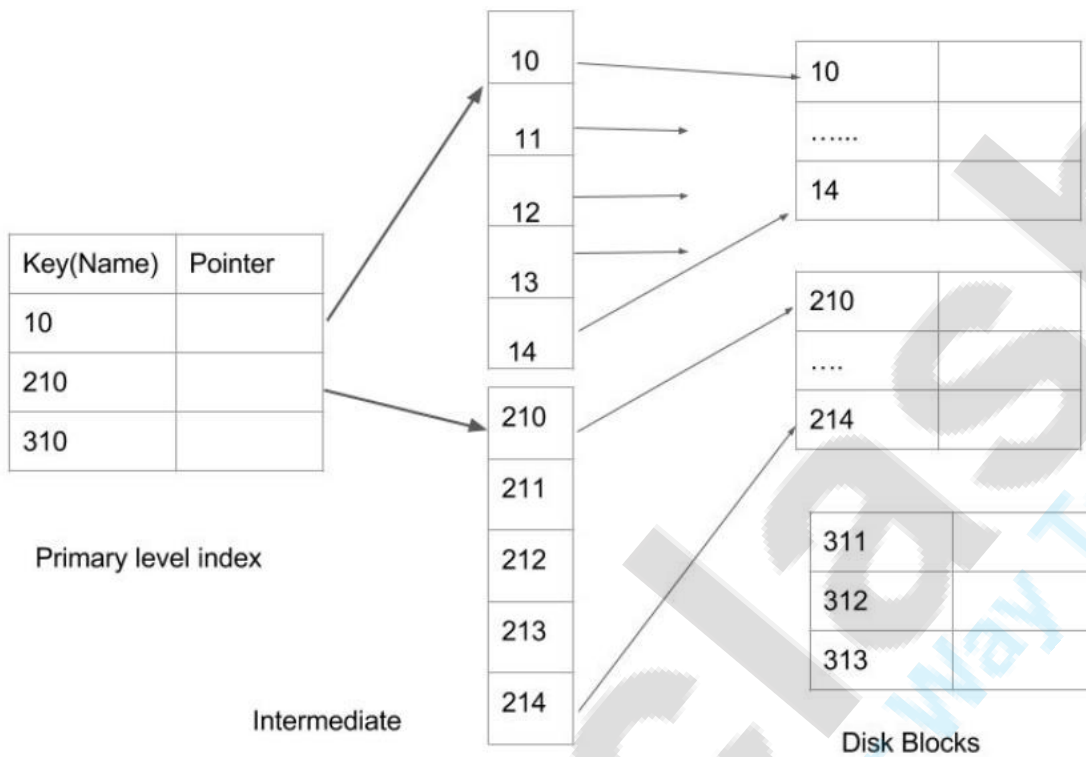
educlash Result / Revaluation Tracker
Track the latest Mumbai University Results / Revaluation as they happen, all in one App
Visit educlash.com for more

Search Key	Pointer
Alice	1202
Ben	1203
Bethany	1204

1102

Search Key	Pointer
Alice	1102
Bob	1103
Christie	1104

Root Node

Search Key	Pointer
Billie	1500
Bob	1501
Charlie	1502

1103
Intermediate Nodes

1202

Search Key	Attr
Alice	
Adrian	

1203

Search Key	Attr
Ben	
Benjamin	

Search Key	Attr
Bethany	
Betty	

1204

Leaf Nodes

Non clustered index

It requires more time as compared to clustered index because some amount of extra work is done in order to extract the data by further following the pointer. In case of clustered index, data is directly present in front of the index.

Secondary Index

It is used to optimize query processing and access records in a database with some information other than the usual search key (primary key). In this two levels of indexing are used in order to reduce the mapping size of the first level and in general. Initially, for the first level, a large range of numbers is selected so that the mapping size is small. Further, each range is divided into further sub ranges.

In order for quick memory access, first level is stored in the primary memory. Actual physical location of the data is determined by the second mapping level.
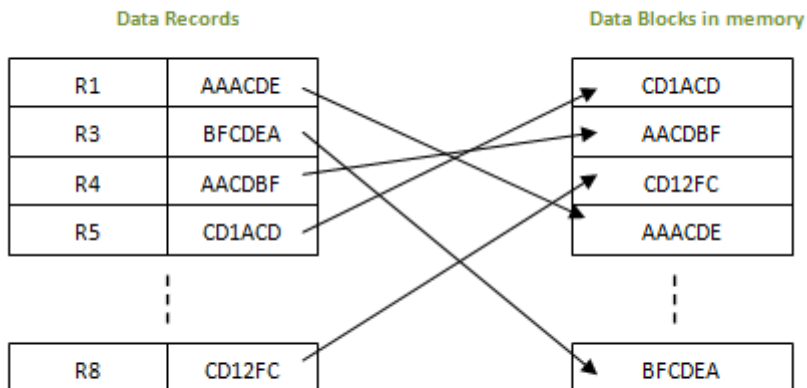
Secondary level Indexing

Indexed Sequential Access Methods

Indexed Sequential Access Method (ISAM)

This is an advanced sequential file organization method. Here records are stored in order of primary key in the file. Using the primary key, the records are sorted. For each primary key, an index value is generated and mapped with the record. This index is nothing but the address of record in the file.

In this method, if any record has to be retrieved, based on its index value, the data block address is fetched and the record is retrieved from memory.

Advantages of ISAM

- Since each record has its data block address, searching for a record in larger database is easy and quick. There is no extra effort to search records. But proper primary key has to be selected to make ISAM efficient.

- This method gives flexibility of using any column as key field and index will be generated based on that. In addition to the primary key and its index, we can have index generated for other fields too. Hence searching becomes more efficient, if there is search based on columns other than primary key.

- It supports range retrieval, partial retrieval of records. Since the index is based on the key value, we can retrieve the data for the given range of values. In the same way, when a partial key value is provided, say student names starting with 'JA' can also be searched easily.

Disadvantages of ISAM

- An extra cost to maintain index has to be afforded. i.e.; we need to have extra space in the disk to store this index value. When there is multiple

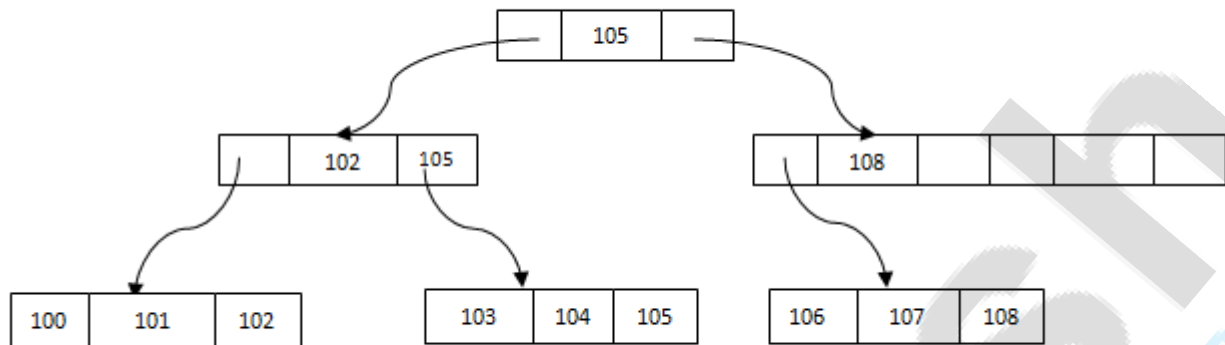key-index combinations, the disk space will also increase.

As the new records are inserted, these files have to be restructured to maintain the sequence. Similarly, when the record is deleted, the space used by it needs to be released. Else, the performance of the database will slow down.

B+ Tree File Organization

B+ Tree is an advanced method of ISAM file organization. It uses the same concept of key-index, but in a tree like structure. B+ tree is similar to binary search tree, but it can have more than two leaf nodes. It stores all the records only at the leaf node. Intermediary nodes will have pointers to the leaf nodes. They do not contain any data/records.

Consider a student table below. The key value here is STUDENT_ID. And each record contains the details of each student along with its key value and the index/pointer to the next value. In a B+ tree it can be represented as below.

STUDENT		
STUDENT_ID	STUDENT_NAME	ADDRESS
100	Joseph	Alaiedon Township
101	Allen	Fraser Township
102	Chris	Clinton Township
103	Patty	Troy
104	Jack	Fraser Township
105	Jessica	Clinton Township
106	James	Troy
107	Antony	Alaiedon Township
108	Jacob	Troy

Please note that the leaf node 100 means, it has name and address of student with ID 100, as we saw in R1, R2, R3 etc above.

From the above B+ tree structure, it is evident that

- There is one main node called root of the tree – 105 is the root here.

- There is an intermediary layer with nodes. They do not have actual records stored. They are all pointers to the leaf node. Only the leaf node contains the data in sorted order.

- The nodes to the left of the root nodes have prior values of root and nodes to the right have next values of the root. i.e.; 102 and 108 respectively.

- There is one final node, called leaf node, which has only values. i.e.; 100, 101, 103, 104, 106 and 107

- All the leaf nodes are balanced – all the leaf nodes at same distance from the root node. Hence searching any record is easier.

- Searching any record is linear in this case. Any record can be traversed through single path and accessed easily.

- Since the intermediary nodes have only pointers to the leaf node, the tree structure is of shorter height. Shorter the height, faster is the traversal and hence the retrieval of records.

Advantages of B+ Trees

- Since all records are stored only in the leaf node and are sorted sequential linked list, searching is becomes very easy.

- Using B+, we can retrieve range retrieval or partial retrieval. Traversing through the tree structure makes this easier and quicker.

- As the number of record increases/decreases, B+ tree structure grows/shrinks. There is no restriction on B+ tree size, like we have in ISAM.

- Since it is a balance tree structure, any insert/ delete/ update does not affect the performance.

- Since we have all the data stored in the leaf nodes and more branching of internal nodes makes height of the tree shorter. This reduces disk I/O. Hence it works well in secondary storage devices.

Disadvantages of B+ Trees

- This method is less efficient for static tables.

DBMS - Hashing

For a huge database structure, it can be almost next to impossible to search all the index values through all its level and then reach the destination data block to retrieve the desired data. Hashing is an effective technique to calculate the direct location of a data record on the disk without using index structure.
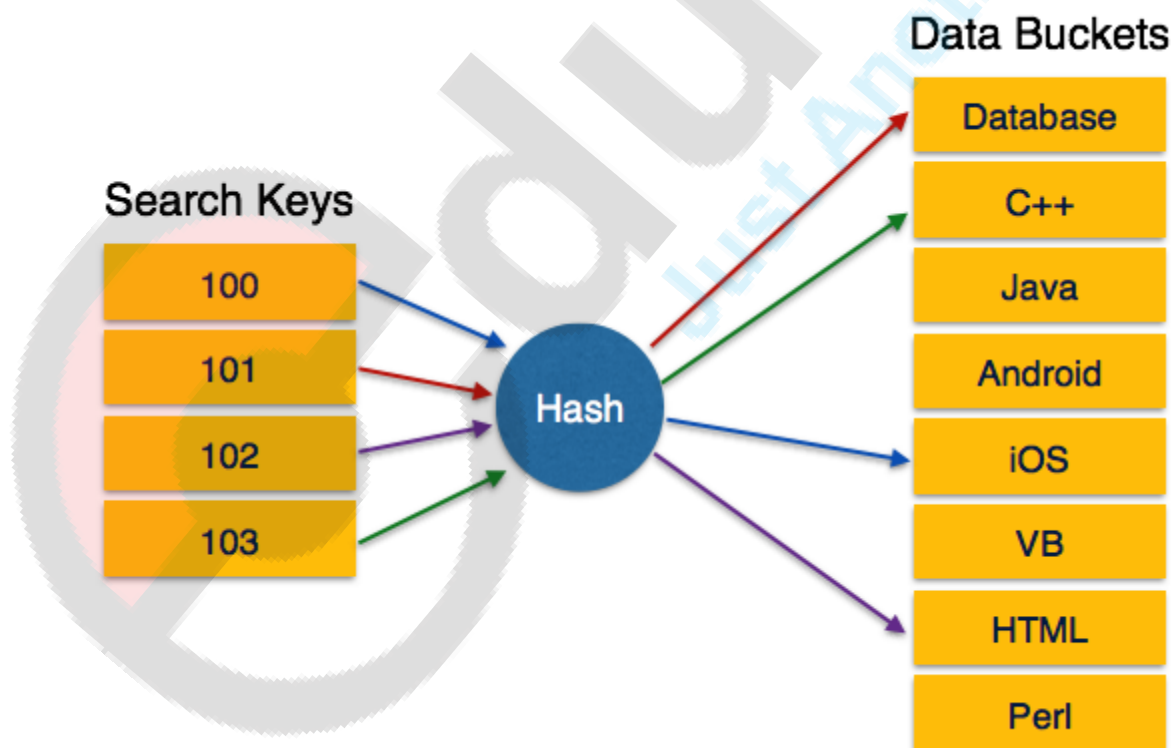
Hashing uses hash functions with search keys as parameters to generate the address of a data record.

Hash Organization

- **Bucket** – A hash file stores data in bucket format. Bucket is considered a unit of storage. A bucket typically stores one complete disk block, which in turn can store one or more records.

- **Hash Function** – A hash function, **h,** is a mapping function that maps all the set of search-keys **K** to the address where actual records are placed. It is a function from search keys to bucket addresses.

Static Hashing

In static hashing, when a search-key value is provided, the hash function always computes the same address. For example, if mod-4 hash function is used, then it shall generate only 5 values. The output address shall always be same for that function. The number of buckets provided remains unchanged at all times.

Operation

- **Insertion** – When a record is required to be entered using static hash, the hash function **h** computes the bucket address for search key **K**, where the record will be stored.
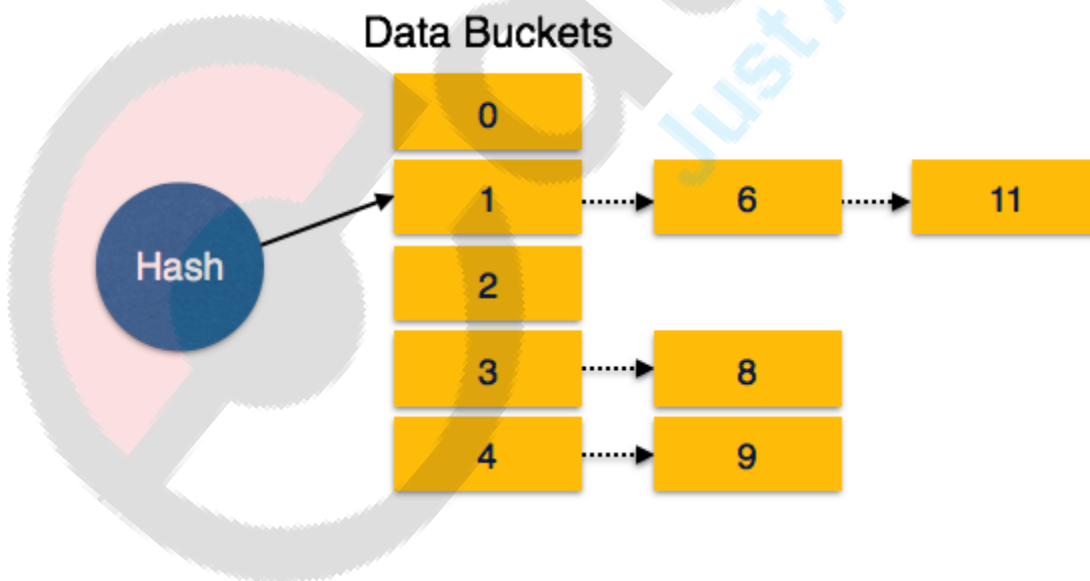
  Bucket address = h(K)

- **Search** – When a record needs to be retrieved, the same hash function can be used to retrieve the address of the bucket where the data is stored.

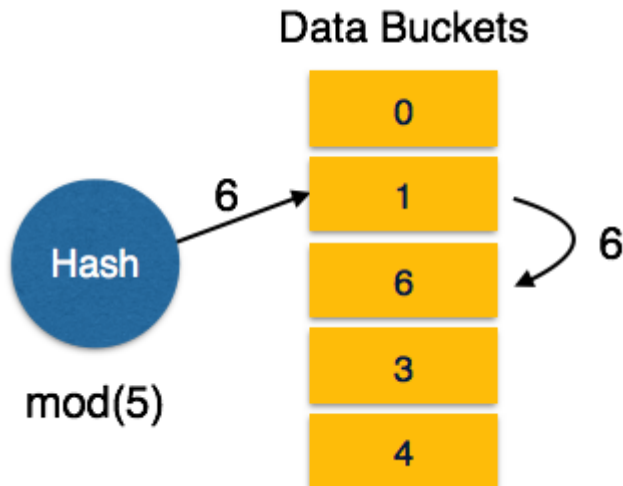- **Delete** – This is simply a search followed by a deletion operation.

Bucket Overflow

The condition of bucket-overflow is known as **collision**. This is a fatal state for any static hash function. In this case, overflow chaining can be used.

- **Overflow Chaining** – When buckets are full, a new bucket is allocated for the same hash result and is linked after the previous one. This mechanism is called **Closed Hashing**.
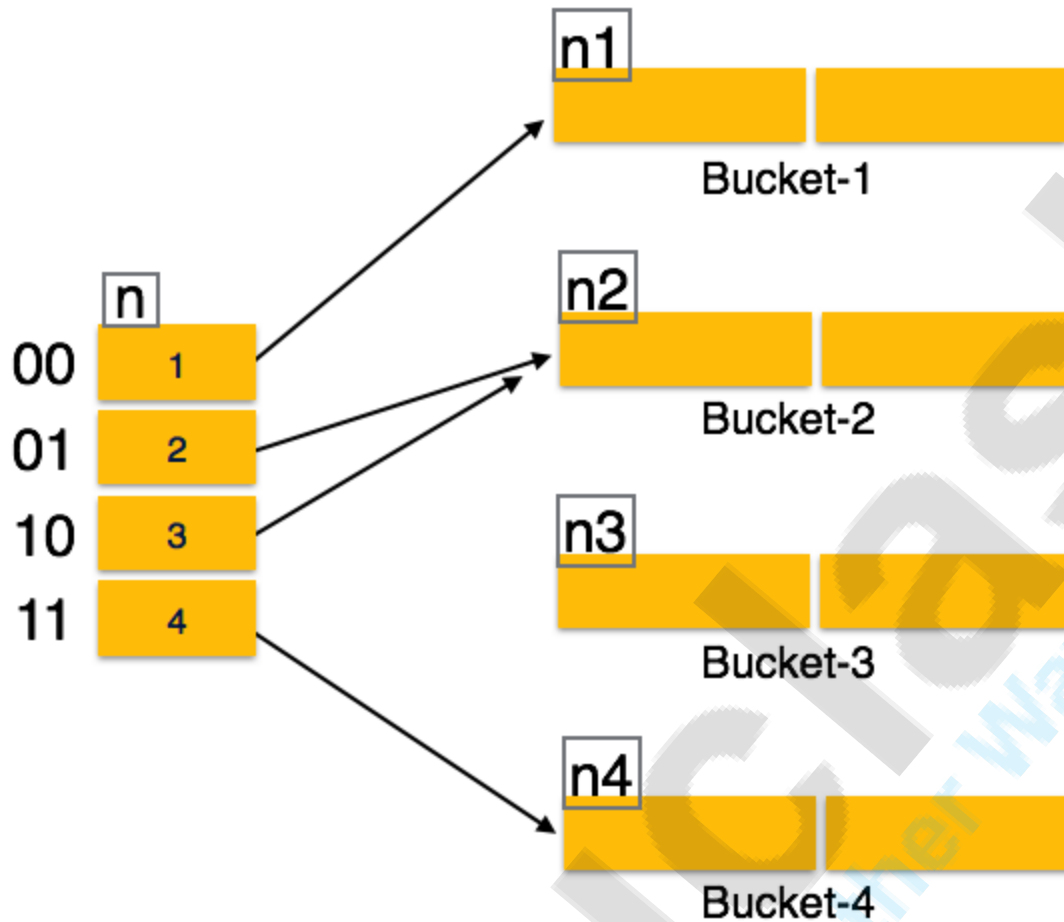
- **Linear Probing** − When a hash function generates an address at which data is already stored, the next free bucket is allocated to it. This mechanism is called **Open Hashing**.

Dynamic Hashing

The problem with static hashing is that it does not expand or shrink dynamically as the size of the database grows or shrinks. Dynamic hashing provides a mechanism in which data buckets are added and removed dynamically and on-demand. Dynamic hashing is also known as **extended hashing**.

Hash function, in dynamic hashing, is made to produce a large number of values and only a few are used initially.

Organization

The prefix of an entire hash value is taken as a hash index. Only a portion of the hash value is used for computing bucket addresses. Every hash index has a depth value to signify how many bits are used for computing a hash function. These bits can address 2n buckets. When all these bits are consumed − that is, when all the buckets are full − then the depth value is increased linearly and twice the buckets are allocated.

Operation

- **Querying** − Look at the depth value of the hash index and use those bits to compute the bucket address.

- **Update** − Perform a query as above and update the data.

- **Deletion** – Perform a query to locate the desired data and delete the same.

- **Insertion** – Compute the address of the bucket

  - If the bucket is already full.

    - Add more buckets.

    - Add additional bits to the hash value.

    - Re-compute the hash function.

  - Else

    - Add data to the bucket,

  - If all the buckets are full, perform the remedies of static hashing.

Hashing is not favorable when the data is organized in some ordering and the queries require a range of data. When data is discrete and random, hash performs the best.

Hashing algorithms have high complexity than indexing. All hash operations are done in constant time.

Extensible Hashing

Extendible hashing is a type of hash system which treats a hash as a bit string, and uses a trie for bucket lookup.[1] Because of the hierarchical nature of the system, re-hashing is an incremental operation (done one bucket at a time, as needed). This means that time-sensitive applications are less affected by table growth than by standard full-table rehashes.

Extendible hashing was described by Ronald Fagin in 1979. Practically all modern filesystems use either extendible hashing or B-trees. In particular,

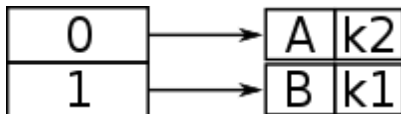the Global File System, ZFS, and the SpadFS filesystem use extendible hashing.[2]
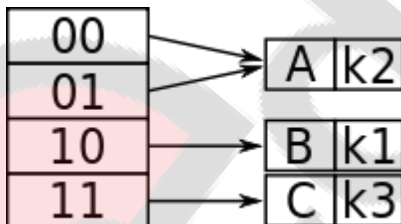
Example[edit]

Assume that the hash function   returns a string of bits. The first i bits of each string will be used as indices to figure out where they will go in the "directory" (hash table). Additionally, i is the smallest number such that the index of every item in the table is unique.

Keys to be used:

Let's assume that for this particular example, the bucket size is 1. The first two keys to be inserted, k_1 and k_2, can be distinguished by the most significant bit, and would be inserted into the table as follows:

Now, if k_3 were to be hashed to the table, it wouldn't be enough to distinguish all three keys by one bit (because both k_3 and k_1 have 1 as their leftmost bit). Also, because the bucket size is one, the table would overflow. Because comparing the first two most significant bits would give each key a unique location, the directory size is doubled as follows:

And so now k_1 and k_3 have a unique location, being distinguished by the first two leftmost bits. Because k_2 is in the top half of the table, both 00 and 01 point to it because there is no other key to compare to that begins with a 0.

Linear Hashing

Linear hashing is a dynamic hash table algorithm invented by Witold Litwin (1980),[1] and later popularized by Paul Larson. Linear hashing allows for the expansion of the hash table one slot at a time. The frequent single slot expansion can very effectively control the length of the collision chain. The cost of hash table expansion is spread out across each hash table insertion operation, as opposed to being incurred all at once.[2] Linear hashing is therefore well suited for interactive applications.