UNIT IV	IMPLEMENTATION TECHNIQUES
RAID – File Organization – Organization of Records in Files – Indexing and Hashing –Ordered Indices – B+ tree Index Files – B tree Index Files – Static Hashing – Dynamic Hashing – Query Processing Overview – Algorithms for SELECT and JOIN operations – Query optimization using Heuristics and Cost Estimation	

RAID

RAID (redundant array of independent disks) originally redundant array of inexpensive disks) is a way of storing the same data in different places on multiple hard disks to protect data in the case of a drive failure.

RAID: Redundant Arrays of Independent Disks

Disk organization techniques that manage a large numbers of disks, providing a view of a single disk of high capacity and high speed by using multiple disks in parallel, and high reliability by storing data redundantly, so that data can be recovered even if a disk fails

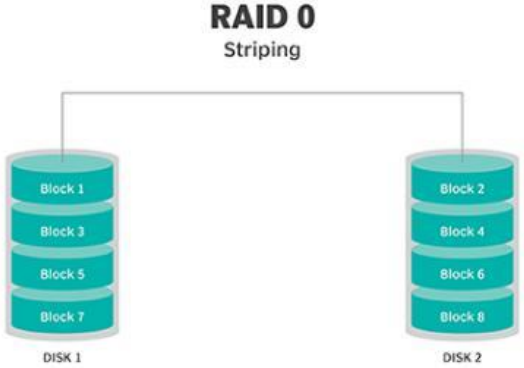
Motivation for RAID

- Just as additional memory in form of cache, can improve the system performance, in the same way additional disks can also improve system performance.
- In RAID we can use an array of disks which operates independently since there are many disks, multiple I/O requests can be handled in parallel if the data required is on separate disks
- A single I/O operation can be handled in parallel if the data required is distributed across multiple disks.

Benefits of RAID

- Data loss can be very dangerous for an organization
- RAID technology prevents data loss due to disk failure
- RAID technology can be implemented in hardware or software
- Servers make use of RAID Technology

- RAID level 0 divides data into block units and writes them across a number of disks. As data is placed across multiple disks it is also called “data Striping”.
- The advantage of distributing data over disks is that if different I/O requests are pending for two different blocks of data, then there is a possibility that the requested blocks are on different disks



There is no parity checking of data. So if data in one drive gets corrupted then all the data would be lost. Thus RAID 0 does not support data recovery Spanning is another term that is used with RAID level 0 because the logical disk will span all the physical drives. RAID 0 implementation requires minimum 2 disks.

Advantages

- I/O performance is greatly improved by spreading the I/O load across many channels & drives.
- Best performance is achieved when data is striped across multiple controllers with only one driver per controller

Disadvantages

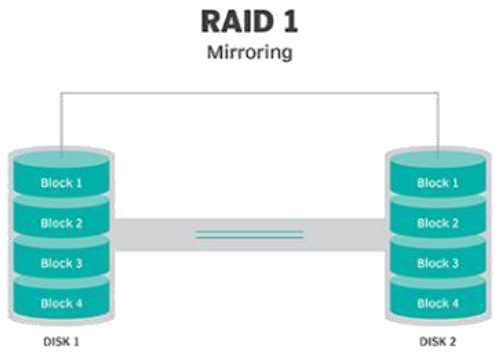
- It is not fault-tolerant, failure of one drive will result in all data in an array being lost

RAID Level 1: **Mirroring** (or **shadowing**)

- Also known as *disk mirroring*, this configuration consists of at least two drives that duplicate the storage of

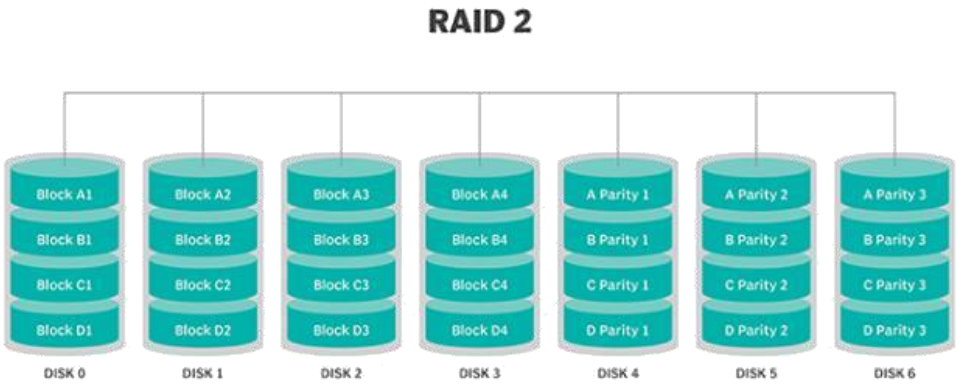
data. There is no striping.

- Read performance is improved since either disk can be read at the same time. Write performance is the same as for single disk storage.
- Every write is carried out on both disks. If one disk in a pair fails, data still available in the other.
- Data loss would occur only if a disk fails, and its mirror disk also fails before the system is repaired Probability of combined event is very small.



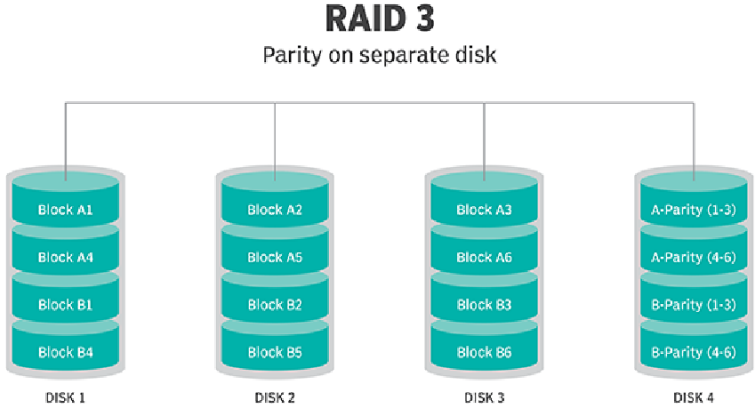
RAID Level 2:

This configuration uses striping across disks, with some disks storing error checking and correcting (ECC) information. It has no advantage over RAID 3 and is no longer used.



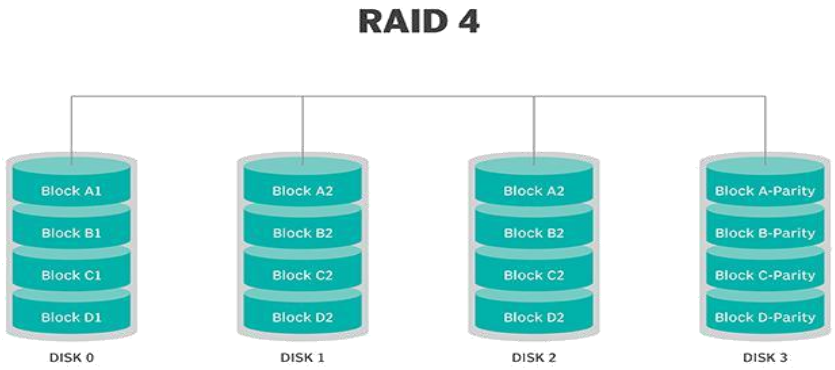
RAID Level 3: Bit-Interleaved Parity

- A single parity bit is enough for error correction, not just detection, since we know which disk has failed
  - When writing data, corresponding parity bits must also be computed and written to a parity bit disk
  - To recover data in a damaged disk, compute XOR of bits from other disks (including parity bit disk)
- I/O operation addresses all the drives at the same time, RAID 3 cannot overlap I/O. For this reason, RAID 3 is best for single-user systems with long record applications.



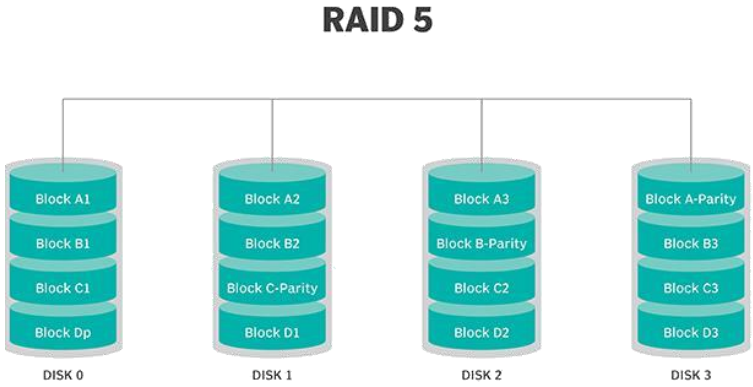
RAID Level 4: Block-Interleaved Parity

- When writing data block, corresponding block of parity bits must also be computed and written to parity disk
- To find value of a damaged block, compute XOR of bits from corresponding blocks (including parity block) from other disks.



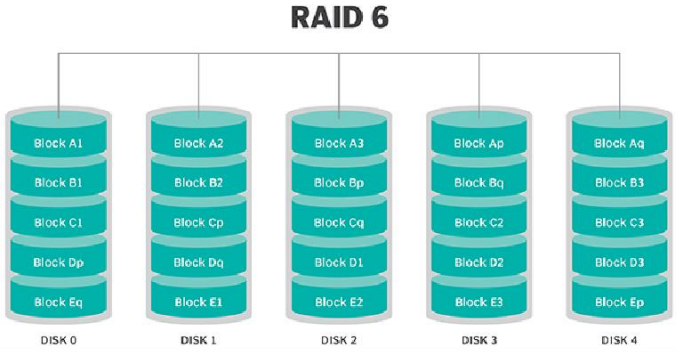
RAID Level 5:

- RAID 5 uses striping as well as parity for redundancy. It is well suited for heavy read and low write operations.
- Block-Interleaved Distributed Parity; partitions data and parity among all *N* + 1 disks, rather than storing data in *N* disks and parity in 1 disk.



RAID Level 6:

- This technique is similar to RAID 5, but includes a second parity scheme that is distributed across the drives in the array. The use of additional parity allows the array to continue to function even if two disks fail simultaneously. However, this extra protection comes at a cost.
- P+Q Redundancy scheme; similar to Level 5, but stores extra redundant information to guard against multiple disk failures.
    - Better reliability than Level 5 at a higher cost; not used as widely.



File Organization

- The database is stored as a collection of *files*.
- Each file is a sequence of *records.*
- A record is a sequence of fields.
- Classifications of records
  - Fixed length record
  - Variable length record
- Fixed length record approach:

  Assume record size is fixed each file has records of one particular type only different files are used for different relations

Simple approach

- Record access is simple

Example pseudo code

```
type account = record
                 account_number char(10);
                 branch_name char(22);
                  balance numeric(8);
               end
```

Total bytes 40 for a record

record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
record 2	A-215	Mianus	700
record 3	A-101	Downtown	500
record 4	A-222	Redwood	700
record 5	A-201	Perryridge	900
record 6	A-217	Brighton	750
record 7	A-110	Downtown	600
record 8	A-218	Perryridge	700

Two problems

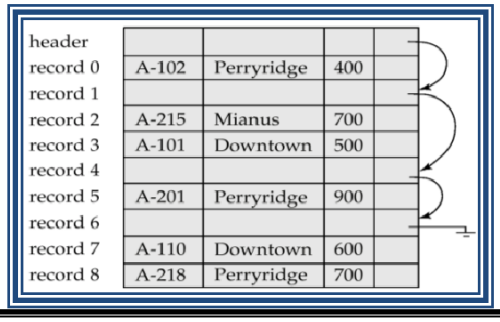
- Difficult to delete record from this structure.
- Some record will cross block boundaries, that is part of the record will be stored in one block and part in another. It would require two block accesses to read or write

Reuse the free space alternatives*:*

- move records i + 1, . . ., n to n i, . . . , n – 1
- do not move records, but link all free records on a free list
- Move the final record to deleted record place.

Free Lists

- Store the address of the first deleted record in the file header.
- Use this first record to store the address of the second deleted record, and so on



Variable-Length Records

Byte string representation

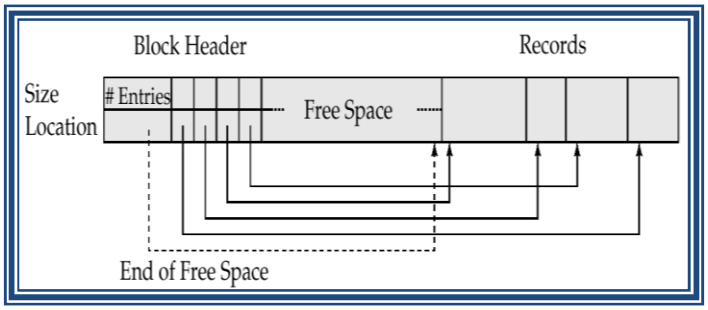
- Attach an *end-of-record* (⊥) control character to the end of each record
- Difficulty with deletion

0	perryridge	A-102	400	A-201	900	⊥
1	roundhill	A-305	350	⊥		
2	mianus	A-215	700	⊥		

Disadvantage

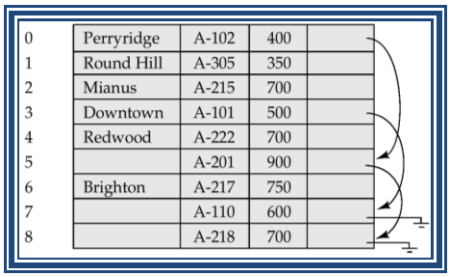
- It is not easy to reuse space occupied formerly by deleted record.
- There is no space in general for records grows longer

Slotted Page Structure



- Slotted page header contains:
  - number of record entries
  - end of free space in the block
  - location and size of each record

Pointer Method

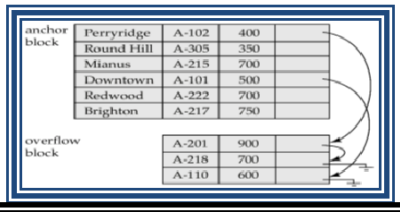


- A variable-length record is represented by a list of fixed-length records, chained together via pointers.
- Can be used even if the maximum record length is not known.

Disadvantage to pointer structure; space is wasted in all records except the first in a a chain.

Solution is to allow two kinds of block in file:

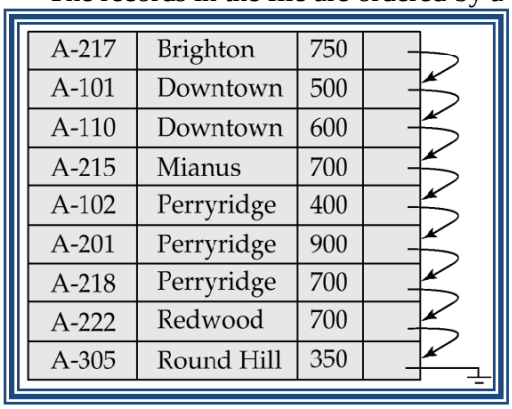
- Anchor block – contains the first records of chain
- Overflow block – contains records other than those that are the first records of chains.



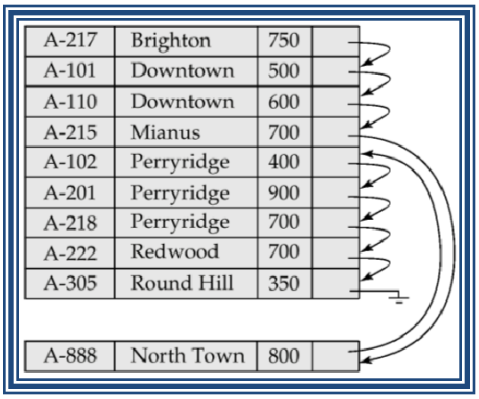
Organization of Records in Files

- Sequential – store records in sequential order, based on the value of the search key of each record
- Heap – a record can be placed anywhere in the file where there is space
- Hashing – a hash function computed on some attribute of each record; the result specifies in which block of the file the record should be placed

- Suitable for applications that require sequential processing of the entire file
- The records in the file are ordered by a search-key



A-217	Brighton	750	
A-101	Downtown	500	
A-110	Downtown	600	
A-215	Mianus	700	
A-102	Perryridge	400	
A-201	Perryridge	900	
A-218	Perryridge	700	
A-222	Redwood	700	
A-305	Round Hill	350	



A-217	Brighton	750	
A-101	Downtown	500	
A-110	Downtown	600	
A-215	Mianus	700	
A-102	Perryridge	400	
A-201	Perryridge	900	
A-218	Perryridge	700	
A-222	Redwood	700	
A-305	Round Hill	350	
A-888	North Town	800	

Deletion – use pointer chains

Insertion – locate the position where the record is to be inserted

- if there is free space insert there
- if no free space, insert the record in an overflow block
- In either case, pointer chain must be updated

Indexing and Hashing

Basic Concepts

- Indexing mechanisms used to speed up access to desired data.
  - E.g., author catalog in library
- **Search Key** - attribute to set of attributes used to look up records in a file.

An **index file** consists of records (called **index entries**) of the form

Search-key	**pointer**

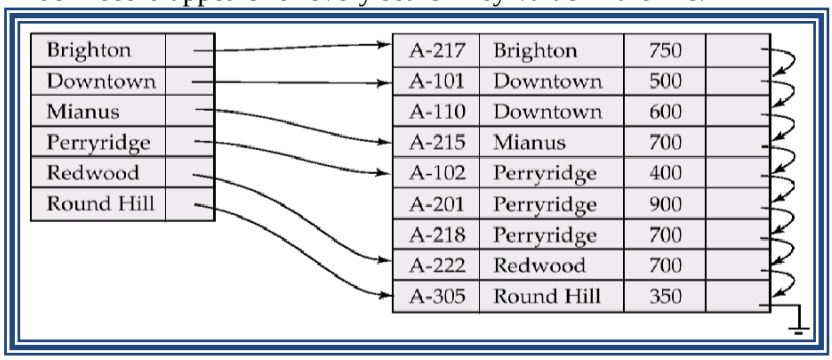
- Index files are typically much smaller than the original file
- Two basic kinds of indices:
  - **Ordered indices:** search keys are stored in sorted order
  - **Hash indices:** search keys are distributed uniformly across "buckets" and by using a "hash function" the values are determined.

- In an **ordered index,** index entries are stored sorted on the search key value.
- **Primary index:** in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
- **Secondary index**: an index whose search key specifies an order different from the sequential order of the file.

- Dense index
- Sparse index

Dense Index Files

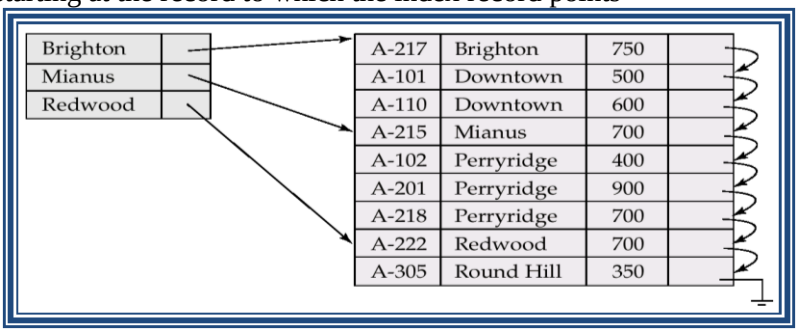
- Dense index — Index record appears for every search-key value in the file.



Sparse Index Files

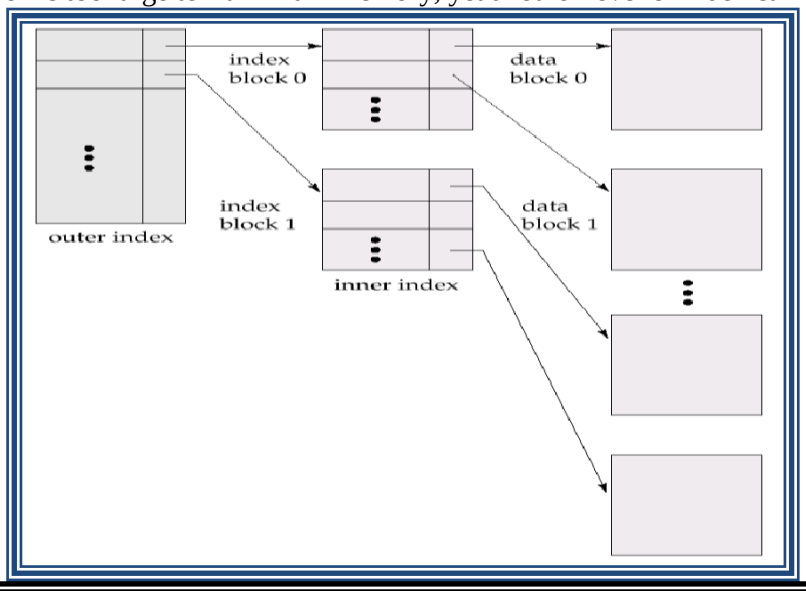
- Sparse Index
  contains index records for only some search-key values.
- To locate a record with search-key value *K* we:
  – Find index record with largest search-key value that is less than or equal to

Search file sequentially starting at the record to which the index record points



Multilevel Index

- If primary index does not fit in memory, access becomes expensive.
- To reduce number of disk accesses to index records, treat primary index kept on disk as a sequential file and construct a sparse index on it.
  – outer index – a sparse index of primary index
  – inner index – the primary index file
- If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.



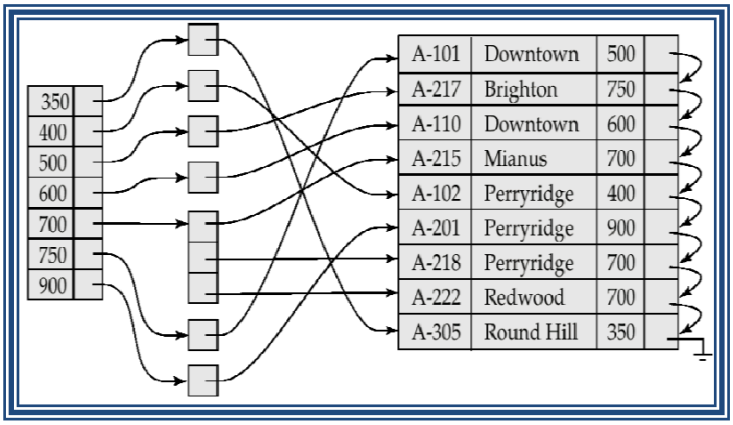
Index Update: Deletion

If deleted record was the only record in the file with its particular search-key value, the search-key is deleted from the index also.

- **Single-level index deletion:**
  - Dense indices – deletion of search-key is similar to file record deletion.
  - Sparse indices – if an entry for the search key exists in the index, it is deleted by replacing the entry in the index with the next search-key value in the file (in search-key order). If the next search-key value already has an index entry, the entry is deleted instead of being replaced.

- **Single-level index insertion:**
  - Perform a lookup using the search-key value appearing in the record to be inserted.
  - **Dense indices** – if the search-key value does not appear in the index, insert it.
  - **Sparse indices** – if index stores an entry for each block of the file, no change needs to be made to the index unless a new block is created. In this case, the first search-key value appearing in the new block is inserted into the index.

Secondary Index on ***balance*** field of ***account***



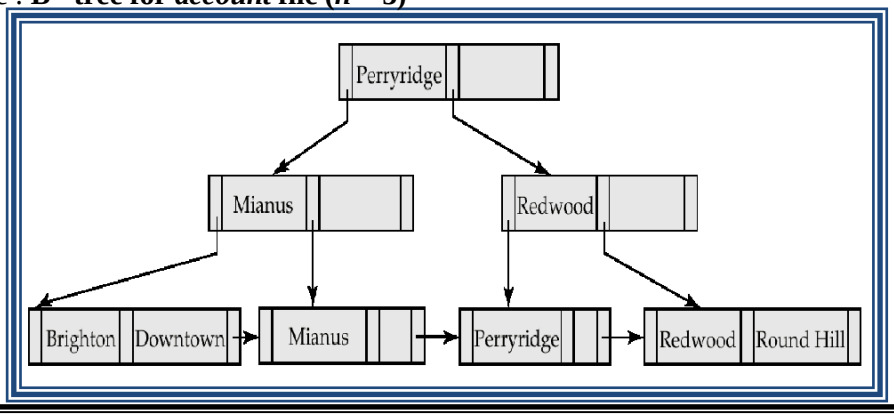
Primary and Secondary Indices

- Secondary indices have to be dense.
- Indices offer substantial benefits when searching for records.
- When a file is modified, every index on the file must be updated, Updating indices imposes overhead on database modification.
- Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive
  - each record access may fetch a new block from disk

`

B^+-Tree Index Files

Example of a B^+-tree : **B^+-tree for *account* file (*n* = 3)**



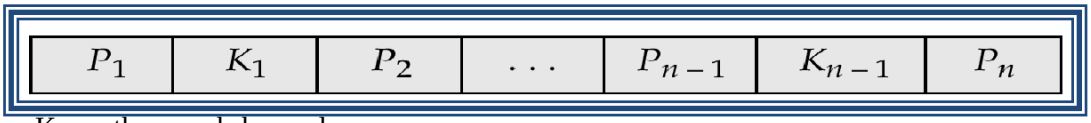
- **Disadvantage of indexed-sequential files**: performance degrades as file grows, since many overflow blocks get created. Periodic reorganization of entire file is required.
- **Advantage of B$^+$-tree index files**: automatically reorganizes itself with small, local, changes, in the face of insertions and deletions. Reorganization of entire file is not required to maintain performance.
- **Disadvantage of B$^+$-trees:** extra insertion and deletion overhead, space overhead.

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

- All paths from root to leaf are of the same length
- Each node that is not a root or a leaf has between [n/2] and n children.
- Special cases:
  - If the root is not a leaf, it has at least 2 children.
  - If the root is a leaf, it can have between 0 and (n–1) values.

B$^+$-Tree Node Structure

- **Typical node**

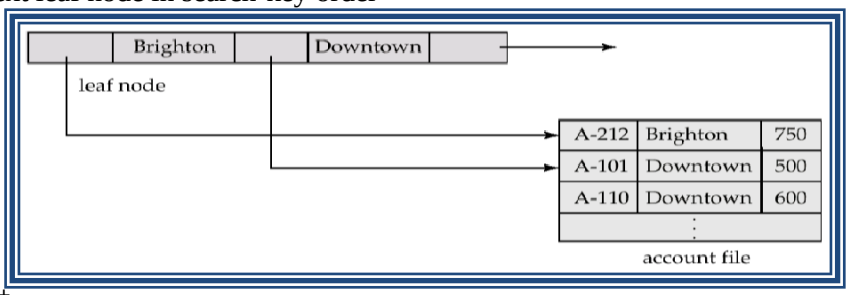


  - K_i are the search-key values
  - P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).
- The search-keys in a node are ordered

  $K_1 < K_2 < K_3 < \ldots < K_{n-1}$

Properties of Leaf Nodes

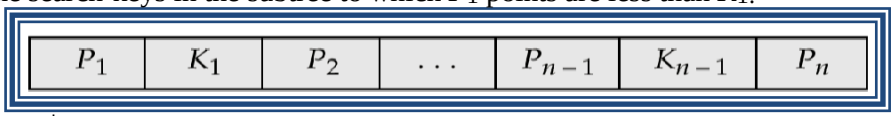
- For i = 1, 2, . . ., n–1, pointer P_i either points to a file record with search-key value K_i, or to a bucket of pointers to file records, each record having search-key value K_i.
- P_n points to next leaf node in search-key order



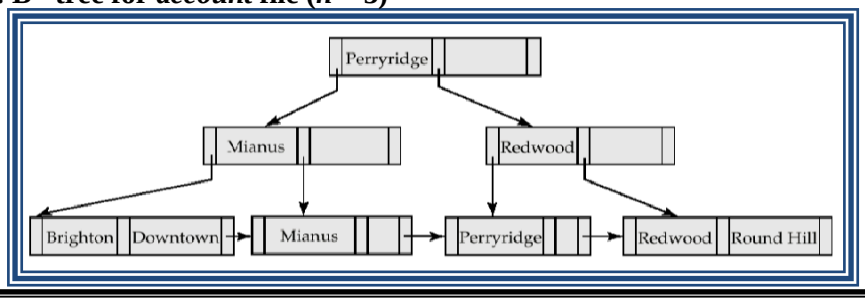
Non-Leaf Nodes in B$^+$-Trees

Non leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with m pointers:

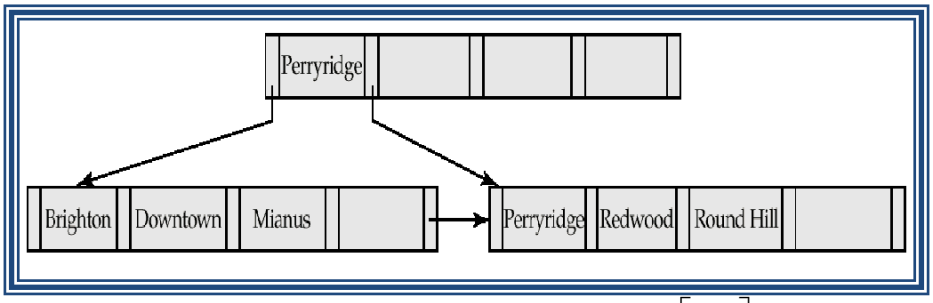
- All the search-keys in the subtree to which P_1 points are less than K_1.



Example of a B$^+$-tree**: B$^+$-tree for *account* file (n = 3)**



B^+-tree for *account* file (n = 5)



- Non-leaf nodes other than root must have between 3 and 5 children ($\lceil (n/2 \rceil$ and n with n =5).
- Root must have at least 2 children.

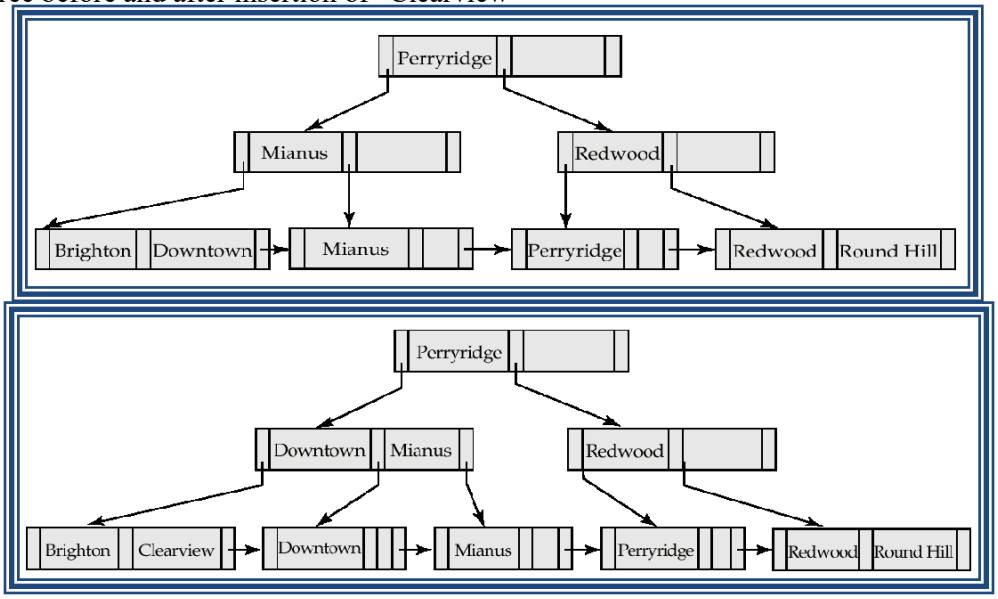
Observations about B^+-trees

- Since the inter-node connections are done by pointers, "logically" close blocks need not be "physically" close.
- The B^+-tree contains a relatively small number of levels thus searches can be conducted efficiently.
- Insertions and deletions to the main file can be handled efficiently.

Updates on B^+-Trees: Insertion

- Find the leaf node in which the search-key value would appear
- If the search-key value is already there in the leaf node, record is added to file and if necessary a pointer is inserted into the bucket.
- If the search-key value is not there, then add the record to the main file and create a bucket if necessary.Then:
  - If there is room in the leaf node, insert (key-value, pointer) pair in the leaf node otherwise, split the node.

Example: B^+-Tree before and after insertion of "Clearview"

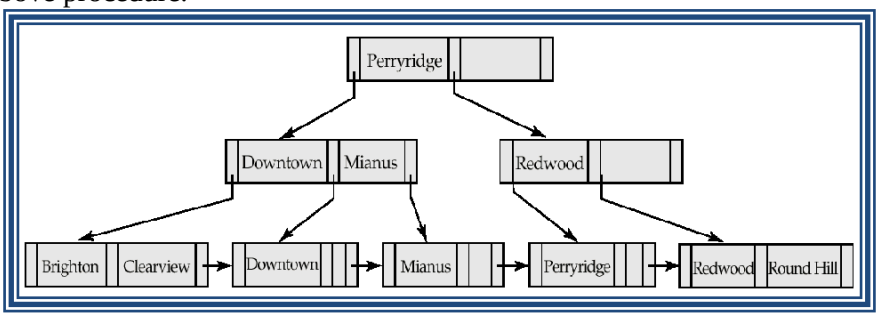


Updates on B^+-Trees: Deletion

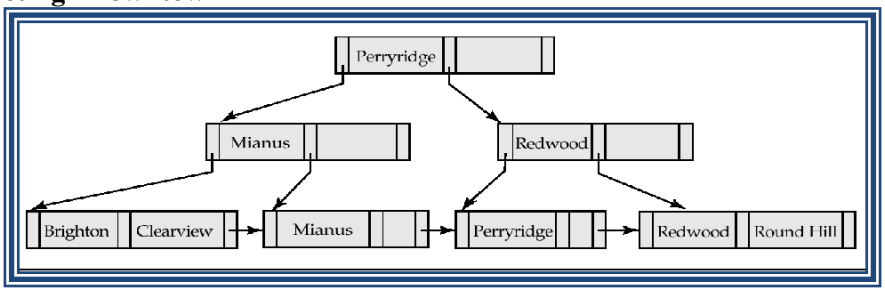
- Find the record to be deleted, and remove it from the main file and from the bucket (if present)
- Remove (search-key value, pointer) from the leaf node if there is no bucket or if the bucket has become empty
- If the node has too few entries due to the removal, and the entries in the node and a sibling fit into a single node, then
  - Insert all the search-key values in the two nodes into a single node (the one on the left), and delete

the other node.

– Delete the pair (K_{i-1}, P_i), where P_i is the pointer to the deleted node, from its parent, recursively using the above procedure.

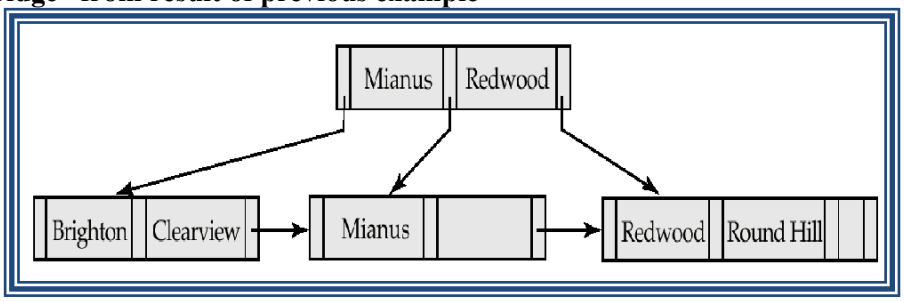


Before and after deleting "Downtown"



- The removal of the leaf node containing "Downtown" did not result in its parent having too little pointers. So the cascaded deletions stopped with the deleted leaf node's parent.

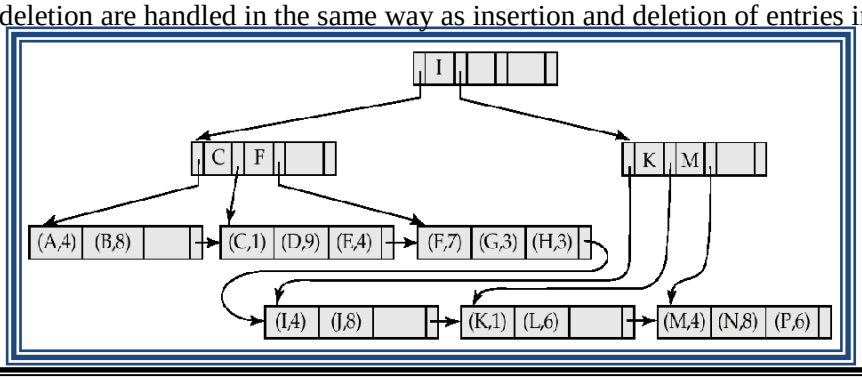
Deletion of "Perryridge" from result of previous example



- Node with "Perryridge" becomes empty and merged with its sibling.
- Root node then had only one child, and was deleted and its child became the new root node

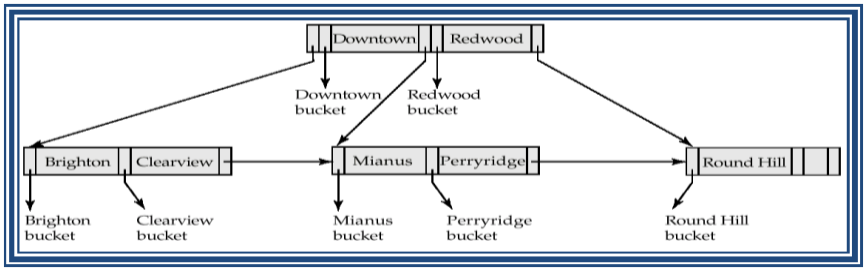
B^+-Tree File Organization

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

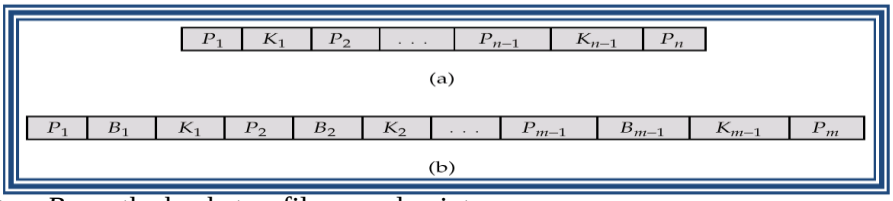


B-Tree Index Files

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

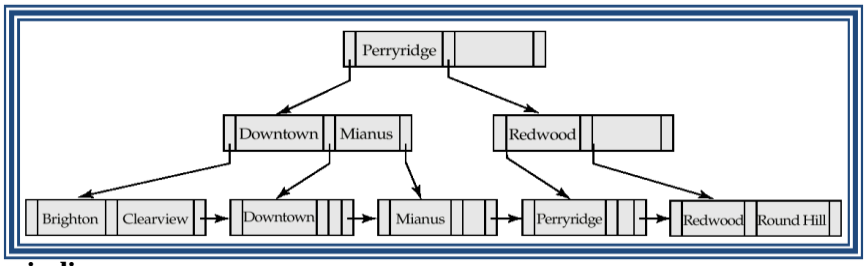


Generalized B-tree leaf node



Nonleaf node – pointers B_i are the bucket or file record pointers.

B+-tree on same data



Advantages of B-Tree indices:

- May use less tree nodes than a corresponding B^+-Tree.
- Sometimes possible to find search-key value before reaching leaf node.

Disadvantages of B-Tree indices:

- Only small fraction of all search-key values are found early
- Non-leaf nodes are larger, so fan-out is reduced (no. of pointers). Thus, B-Trees typically have greater depth than corresponding B^+-Tree
- Insertion and deletion more complicated than in B^+-Trees
- Implementation is harder than B^+-Trees.

HASHING

- 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.

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

- An ideal hash function is uniform, i.e., each bucket is assigned the same number of search-key values from the set of all possible values.
- Ideal hash function is random, so each bucket will have the same number of records.

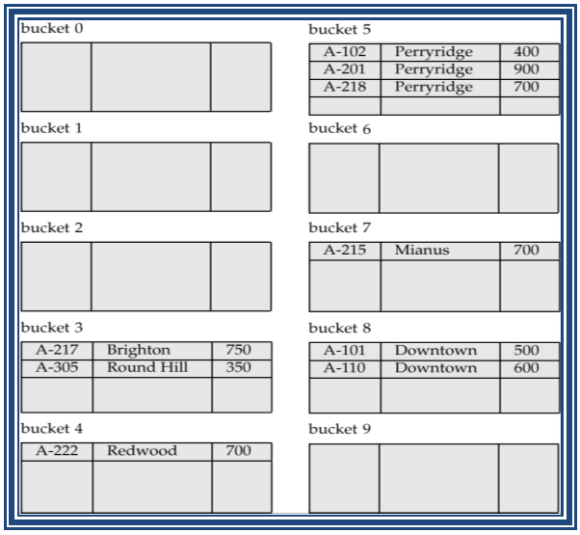
Types

- Static Hashing
- Dynamic Hashing

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.

- There are 10 buckets,
- The hash function returns the sum of the binary representations of the characters modulo 10
  - E.g. h(Perryridge) = 5 h(Round Hill) = 3 h(Brighton) = 3

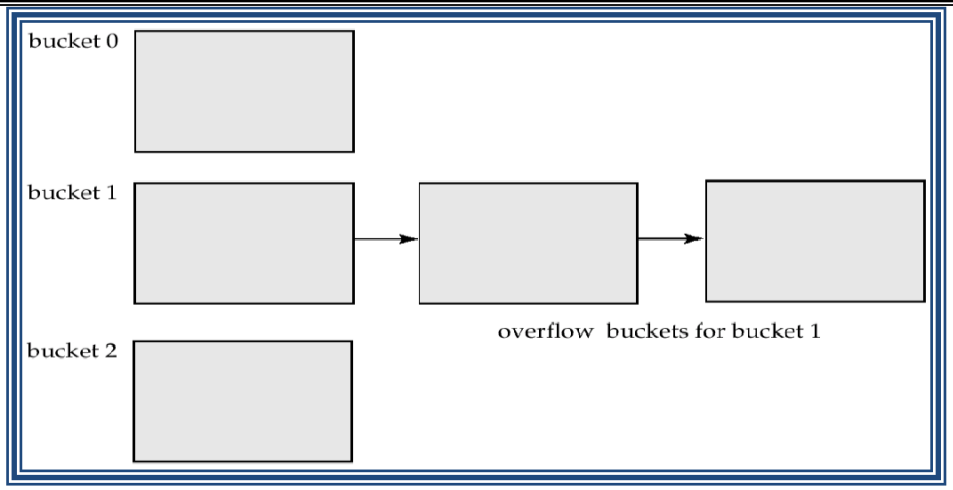


Operation

- **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.

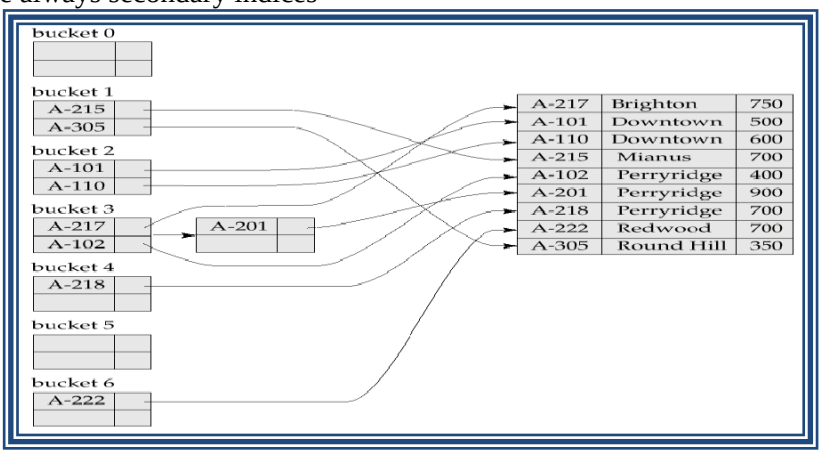
  Bucket address = h(K)
- **Search** − When a record needs to be retrieved, the same hash function can be used to retrieve the address of the bucket where the data is stored.
- **Delete** − This is simply a search followed by a deletion operation.

- Bucket overflow can occur because of
  - Insufficient buckets
  - Skew in distribution of records. This can occur due to :
    - multiple records have same search-key value
- Although the probability of bucket overflow can be reduced, it cannot be eliminated; it is handled by using overflow buckets.
- Overflow chaining – the overflow buckets of a given bucket are chained together in a linked list.
- Above scheme is called **closed hashing**.
  - An alternative, called **open hashing**, which does not use overflow buckets, is not suitable for database applications.



Hash Indices

- Hashing can be used not only for file organization, but also for index-structure creation.
- A hash index organizes the search keys, with their associated record pointers, into a hash file structure.
- Hash indices are always secondary indices



Deficiencies of Static Hashing

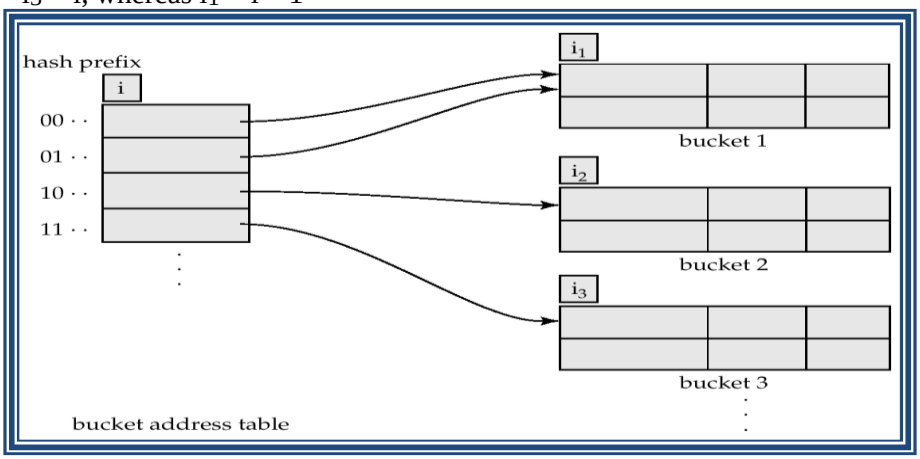
- In static hashing, function h maps search-key values to a fixed set of B of bucket addresses.
  - Databases grow with time. If initial number of buckets is too small, performance will degrade due to too much overflows.
  - If file size at some point in the future is anticipated and number of buckets allocated accordingly, significant amount of space will be wasted initially.
  - If database shrinks, again space will be wasted.
- These problems can be avoided by using techniques that allow the number of buckets to be **modified dynamically**.

- Good for database that grows and shrinks in size
- Allows the hash function to be modified dynamically
- 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.
  - 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

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



Insertion in Extendable Hash Structure

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

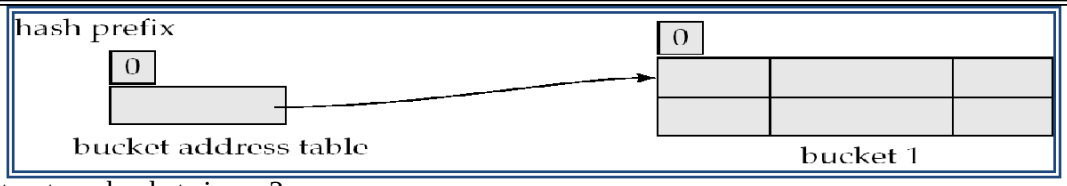
    Now $i > i_j$ so use the first case above.

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

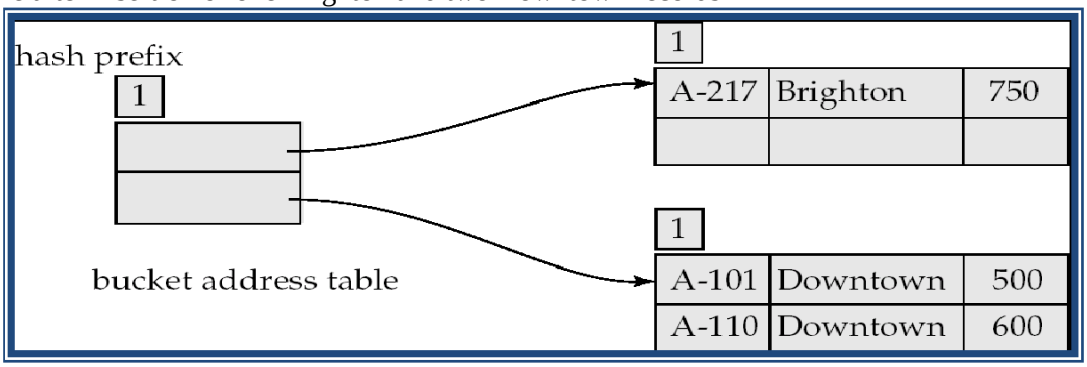
Example

branch_name	h(*branch_name*)
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111 1001 1100 0000 0001

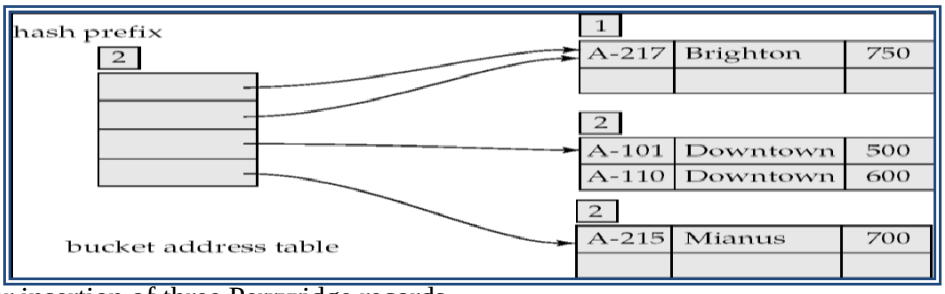


Initial Hash structure, bucket size = 2

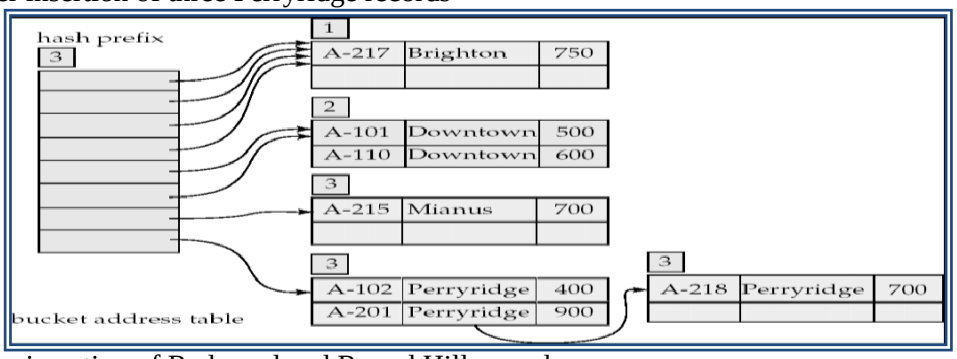
Hash structure after insertion of one Brighton and two Downtown records



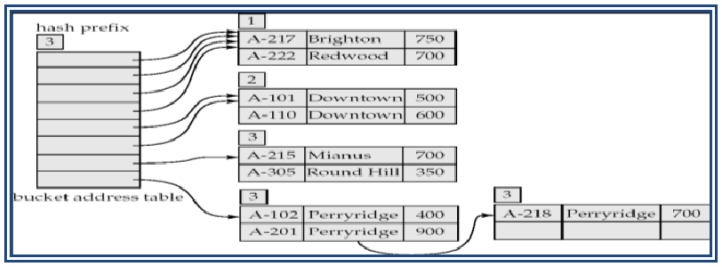
Hash structure after insertion of Mianus record



Hash structure after insertion of three Perryridge records



Hash structure after insertion of Redwood and Round Hill records



Use of Extendable Hash Structure

- 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

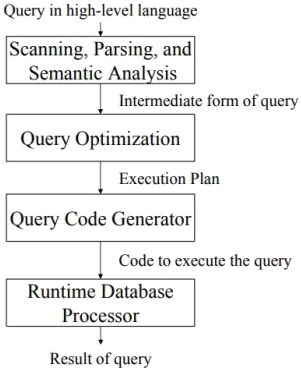
Updates in Extendable Hash Structure

- **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.
  - Overflow buckets used instead in some cases.
- **To delete a key value,**
  - locate it in its bucket and remove it.
  - The bucket itself can be removed if it becomes empty
  - Coalescing of buckets can be done
  - Decreasing bucket address table size is also possible
- **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.

QUERY PROCESSING OVERVIEW

1. The scanning, parsing, and validating module produces an internal representation of the query.
2. The query optimizer module devises an execution plan which is the execution strategy to retrieve the result of the query from the database files. A query typically has many possible execution strategies differing in performance, and the process of choosing a reasonably efficient one is known as query optimization. Query optimization is beyond this course. The code generator generates the code to execute the plan.The runtime database processor runs the generated code to produce the query result.



Evaluation of SQL Statement

The query is evaluated in a different order.

- The tables in the from clause are combined using Cartesian products.The where predicate is then applied.
- The resulting tuples are grouped according to the group by clause.The having predicate is applied to each group, possibly eliminating some groups.The aggregates are applied to each remaining group.The select clause is performed last.

- SQL query is first translated into an equivalent extended relational algebra expression.
- SQL queries are decomposed into query blocks, which form the basic units that can be translated into the algebraic operators and optimized.
- Query block contains a single SELECT-FROM-WHERE expression, as well as GROUP BY and HAVING clauses.
- Nested queries within a query are identified as separate query blocks.

Example:

```
SELECT LNAME, FNAME
FROM EMPLOYEE
WHERE SALARY > (SELECT MAX(SALARY)
                FROM EMPLOYEE
                WHERE DNO=5);
```

- The inner block
  - (SELECT MAX (SALARY) FROM EMPLOYEE WHERE DNO=5)
  - Translated in:
    - 3 MAX SALARY(σDNO=5(EMPLOYEE))
- The Outer block
  - SELECT LNAME, FNAME FROM EMPLOYEE WHERE SALARY > C
  - Translated in:
    - ∏ LNAZME, FNAME (σSALARY>C(EMPLOYEE))

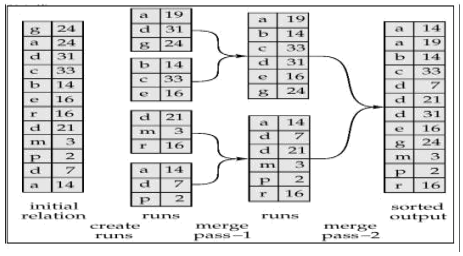
      (C represents the result returned from the inner block.)
- The query optimizer would then choose an execution plan for each block.
- The inner block needs to be evaluated only once. (Uncorrelated nested query).
- It is much harder to optimize the more complex correlated nested queries.

External Sorting

It refers to sorting algorithms that are suitable for large files of records on disk that do not fit entirely in main memory, such as most database files..

- ORDER BY.
- Sort-merge algorithms for JOIN and other operations (UNION, INTERSECTION). Duplicate elimination algorithms for the PROJECT operation (DISTINCT). Typical external sorting algorithm uses a sort-merge strategy: Sort phase: Create sort small sub-files (sorted sub-files are called runs).
- Merge phase: Then merges the sorted runs. N-way merge uses N memory buffers to buffer input runs, and 1 block to buffer output. Select the 1st record (in the sort order) among input buffers, write it to the output buffer and delete it from the input buffer. If output buffer full, write it to disk. If input buffer empty, read next block from the corresponding run..

E.g. 2-way Sort-Merge



Basic Algorithms for Executing Relational Query Operations

- An RDBMS must include one or more alternative algorithms that implement each relational algebra operation (SELECT, JOIN,…) and, in many cases, that implement each combination of these operations.
- Each algorithm may apply only to particular storage structures and access paths (such index,…).
- Only execution strategies that can be implemented by the RDBMS algorithms and that apply to the particular query and particular database design can be considered by the query optimization module.

- These algorithms depend on the file having specific access paths and may apply only to certain types of selection conditions.
- We will use the following examples of SELECT operations:
  - (OP1):σSSN='123456789' (EMPLOYEE)
  - (OP2):σ DNUMBER > 5 (DEPARTMENT)

- (OP3):σDNO=5 (EMPLOYEE)
- (OP4):σ DNO=5 AND SALARY>30000 AND SEX = 'F' (EMPLOYEE)
- (OP5):σESSN='123456789' AND PNO=10 (WORKS_ON)

- Many search methods can be used for simple selection: S1 through S6
- **S1: Linear Search (brute force) –full scan in Oracle's terminology-**
  - Retrieves every record in the file, and test whether its attribute values satisfy the selection condition: an expensive approach.
  - Cost: b/2 if key and b if no key
- **S2: Binary Search**
  - If the selection condition involves an equality comparison on a key attribute on which the file is ordered.
  - σSSN='1234567' (EMPLOYEE), SSN is the ordering attribute.
  - Cost: log2b if key.
- **S3: Using a Primary Index (hash key)**
  - An equality comparison on a key attribute with a primary index (or hash key).
  - This condition retrieves a single record (at most).
  - Cost :primary index : bind/2 + 1(hash key: 1bucket if no collision).
- **S4: Using a primary index to retrieve multiple records**
  - Comparison condition is >, >=, <, or <= on a key field with a primary index
  - σDNUMBER >5(DEPARTMENT)
  - Use the index to find the record satisfying the corresponding equality condition (DNUMBER=5), then retrieve all subsequent records in the (ordered) file.
  - For the condition (DNUMBER <5), retrieve all the preceding records.
  - Method used for range queries too(i.e. queries to retrieve records in certain range)
  - Cost: bind/2 + ?. '?' could be known if the number of duplicates known.
- **S5: Using a clustering index to retrieve multiple records**
  - If the selection condition involves an equality comparison on a non-key attribute with a clustering index.
  - σDNO=5(EMPLOYEE)
  - Use the index to retrieve all the records satisfying the condition.
  - Cost: log2bind + ?. '?' could be known if the number of duplicates known.
- **S6: Using a secondary (B+-tree) index on an equality comparison**
  - The method can be used to retrieve a single record if the indexing field is a key or to retrieve multiple records if the indexing field is not a key.
  - This can also be used for comparisons involving >, >=, <, or <=.
  - Method used for range queries too.
  - Cost to retrieve: a key= height + 1; a non key= height+1(extra-level)+?, comparison=(height-1)+?+?
- Many search methods can be used for complex selection which involve a Conjunctive Condition: S7 through as S9.
  - Conjunctive condition: several simple conditions connected with the AND logical connective.
  - (OP4): s DNO=5 AND SALARY>30000 AND SEX = 'F' (EMPLOYEE).
- **S7:Conjunctive selection using an individual index.**
  - If an attribute involved in any single simple condition in the conjunctive condition has an access path that permits the use of one of the Methods S2 to S6, use that condition to retrieve the records.
  - Then check whether each retrieved record satisfies the remaining simple conditions in the conjunctive condition
- **S8:Conjunctive selection using a composite index:**
  - If two or more attributes are involved in equality conditions in the conjunctive condition and a composite index (or hash structure) exists on the combined fields.
  - Example: If an index has been created on the composite key (ESSN, PNO) of the WORKS_ON file,

we can use the index directly.
- (OP5):$\sigma_{ESSN='123456789' \text{ AND } PNO=10}$ (WORKS_ON).

- **S9: Conjunctive selection by intersection of record pointers**
  - If the secondary indexes are available on more than one of the fields involved in simple conditions in the conjunctive condition, and if the indexes include record pointers (rather than block pointers), then each index can be used to retrieve the set of record pointers that satisfy the individual condition.
  - The intersection of these sets of record pointers gives the record pointers that satisfy the conjunctive condition.
  - If only some of the conditions have secondary indexes, each retrieval record is further tested to determine whether it satisfies the remaining conditions.

Algorithms for implementing JOIN Operation

- **Join: time-consuming operation. We will consider only natural join operation**
  - Two-way join: join on two files.
  - Multiway join: involving more than two files.
- **The following examples of two-way JOIN operation (RΘ A=BS) will be used:**
  - OP6: EMPLOYEE Θ DNO=DNUMBER DEPARTMENT
  - OP7: DEPARTMENT Θ MGRSSN=SSN EMPLOYEE
- **J1: Nested-loop join (brute force)**
  - For each record t in R (outer loop), retrieve every record s from S (inner loop) and test whether the two records satisfy the join condition t[A] = s[B].
- **J2: Single-loop join (using an access structure to retrieve the matching records)**
  - If an index (or hash key) exists for one of the two join attributes (e.g B of S), retrieve each record t in R, one at a time (single loop), and then use the access structure to retrieve directly all matching records s from S that satisfy s[B] = t[A]
- **J3: Sort-merge join:**
  - If the records of R and S are physically sorted (ordered) by value of the join attributes A and B, respectively, we can implement the join in the most efficient way.
  - Both files are scanned concurrently in order of the join attributes, matching the records that have the same values for A and B.
  - If the files are not sorted, they may be sorted first by using external sorting.
  - Pairs of file blocks are copied into memory buffers in order and records of each file are scanned only once each for matching with the other file if A & B are key attributes.
  - The method is slightly modified in case where A and B are not key attributes.
- **J4: Hash-join**
  - The records of files R and S are both hashed to the same hash file using the same hashing function on the join attributes A of R and B of S as hash keys.
- **Partitioning Phase**
  - First, a single pass through the file with fewer records (say, R) hashes its records to the hash file buckets.
  - Assumption: The smaller file fits entirely into memory buckets after the first phase.
  - (If the above assumption is not satisfied, the method is a more complex one and number of variations have been proposed to improve efficiency: partition has join and hybrid hash join.)
- **Probing Phase**
  - A single pass through the other file (S) then hashes each of its records to probe appropriate bucket, and that record is combined with all matching records from R in that bucket.

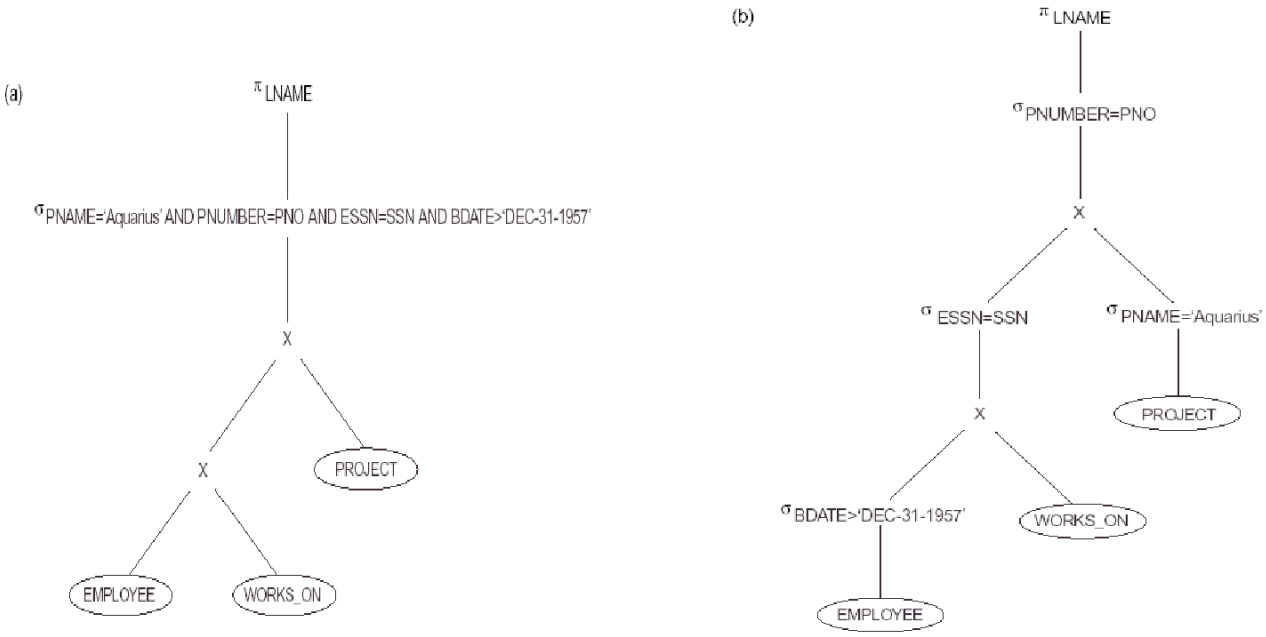
Heuristic-Based Query Optimization

1. Break up SELECT operations with conjunctive conditions into a cascade of SELECT operations
2. Using the commutativity of SELECT with other operations, move each SELECT operation as far down the query tree as is permitted by the attributes involved in the select condition

3. Using commutativity and associativity of binary operations, rearrange the leaf nodes of the tree
4. Combine a CARTESIAN PRODUCT operation with a subsequent SELECT operation in the tree into a JOIN operation, if the condition represents a join condition
5. Using the cascading of PROJECT and the commuting of PROJECT with other operations, break down and move lists of projection attributes down the tree as far as possible by creating new PROJECT operations as needed
6. Identify sub-trees that represent groups of operations that can be executed by a single algorithm

- Query
  "Find the last names of employees born after 1957 who work on a project named 'Aquarius'."
- SQL

SELECT LNAME
FROM EMPLOYEE, WORKS_ON, PROJECT
WHERE PNAME='Aquarius' **AND** PNUMBER=PNO **AND** ESSN=SSN **AND** BDATE.'1957-12-31';



Cost Estimation in Query Optimization

The main aim of query optimization is to choose the most efficient way of implementing the relational algebra operations at the lowest possible cost.

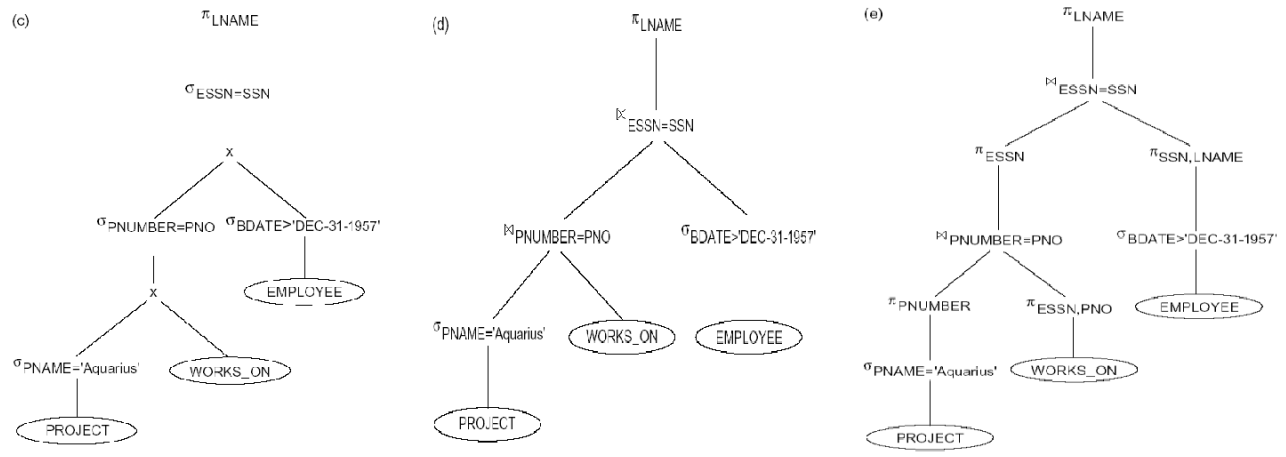
- The query optimizer should not depend solely on heuristic rules, but, it should also estimate the cost of executing the different strategies and find out the strategy with the minimum cost estimate.

The cost functions used in query optimization are estimates and not exact cost functions.

- The cost of an operation is heavily dependent on its selectivity, that is, the proportion of select operation(s) that forms the output.
- In general the different algorithms are suitable for low or high selectivity queries.
- In order for query optimizer to choose suitable algorithm for an operation an estimate of the cost of executing that algorithm must be provided

The cost of an algorithm depends on cardinality of its input.

- To estimate the cost of different query execution strategies, the query tree is viewed as containing a series of basic operations which are linked in order to perform the query.
- It is also important to know the expected cardinality of an operation's output because this forms the input to the next operation.



Cost Components of Query Execution

The cost of executing the query includes the following components:

– Access cost to secondary storage.
– Storage cost.
– Computation cost.
– Memory uses cost.
– Communication cost.

Importance of Access cost

Out of the above five cost components, the most important is the secondary storage access cost.

- The emphasis of the cost minimization depends on the size and type of database applications.
- For example in smaller database the emphasis is on the minimizing computing cost as because most of the data in the files involve in the query can be completely store in the main memory.
- For large database, the main emphasis is on minimizing the access cost to secondary device.
- For distributed database, the communication cost is minimized as because many sites are involved
- for the data transfer.

– [nBlocks(R)/2], if the record is found.
– [nBlocks(R)], if no record satisfied the condition.

Binary Search :

o [log2(nBlocks(R))], if equality condition is on key attribute, because SCA(R) = 1 in this case.
o[log2(nBlocks(R))] + [SCA(R)/bFactor(R)] – 1, otherwise.

Equity condition on Primary key

– [nLevelA(I) + 1]

– [nLevelA(I) + 1] + [nBlocks(R)/2]

Cost functions for JOIN Operation

Join operation is the most time consuming operation to process.

- An estimate for the size (number of tuples) of the file that results after the JOIN operation is required to develop reasonably accurate cost functions for JOIN operations.
- The JOIN operations define the relation containing tuples that satisfy a specific predicate F from the Cartesian product of two relations R and S.