6. ALGORTIHM FOR SELECT AND JOIN OPERATION

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): σDNO=5 AND SALARY>30000 AND SEX = ‗F ‘ (EMPLOYEE)

(OP5): σESSN=‗123456789 ‘ AND PNO=10 (WORKS_ON)

6.1 ALGORTIHM FOR SELECT

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.

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.

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).

S4: Using a primary index to retrieve multiple records

Comparison condition is >, >=, <, or <= on a key field with a primary index

σ DNUMBER >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)

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.

σ DNO=5 (EMPLOYEE) - Use the index to retrieve all the records satisfying the condition

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.

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): σESSN=„123456789" 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.

6.2 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.
- he 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.

2. 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

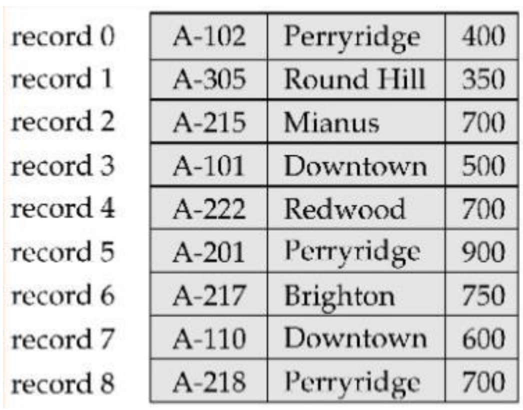
(i) Fixed length record approach:

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.

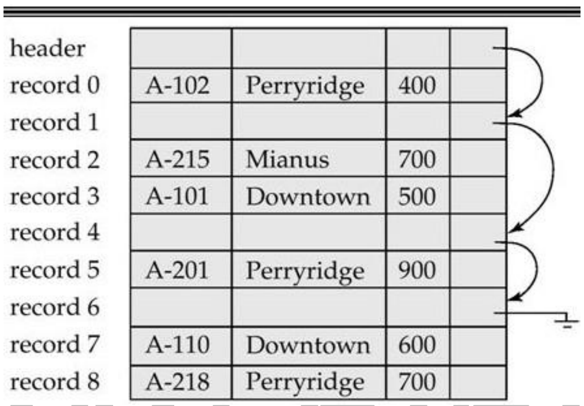
-Some record will cross block boundaries, that is part of the record will be stored in one block and part in another. It would require two block accesses to read or write Reuse the free space alternatives:

- 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



header			
record 0	A-102	Perryridge	400
record 1			
record 2	A-215	Mianus	700
record 3	A-101	Downtown	500
record 4			
record 5	A-201	Perryridge	900
record 6			
record 7	A-110	Downtown	600
record 8	A-218	Perryridge	700

Variable-Length Records

Byte string representation

Attach an end-of-record control character to the end of each

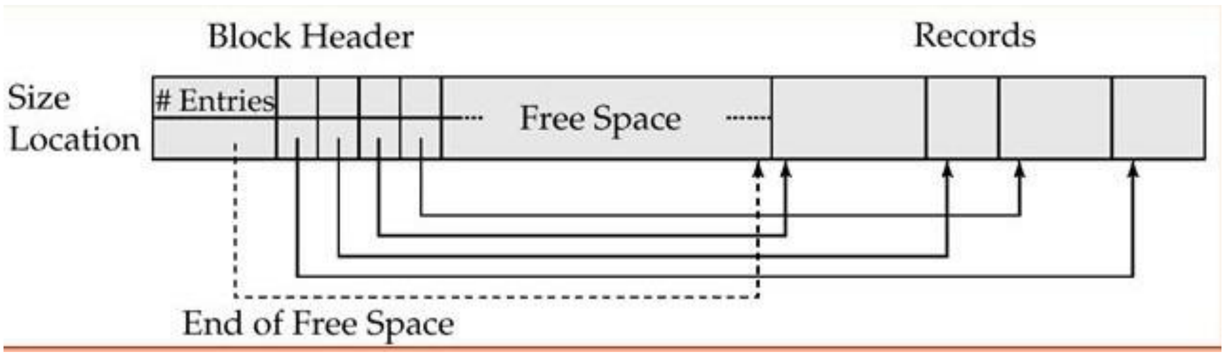
record. Difficulty with deletion

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

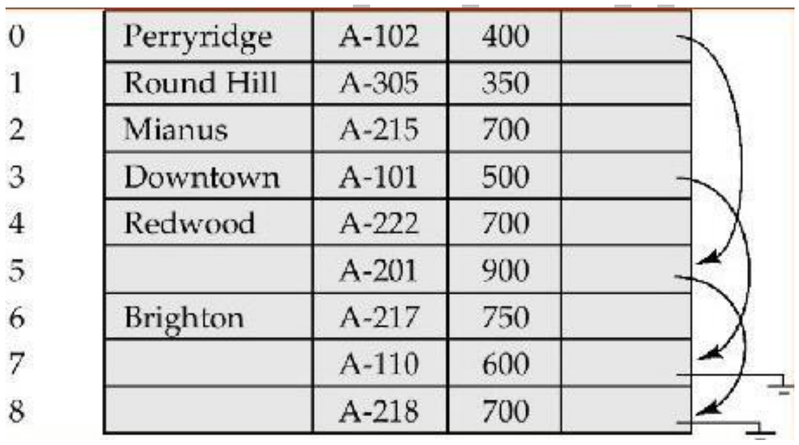
Slotted Page Structure



Slotted page header contains:

- number of record entries
- end of free space in the block
- location and size of each record

Pointer Method

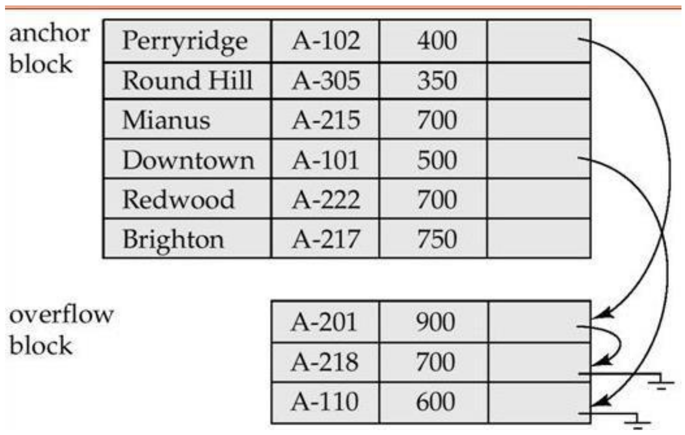


0	Perryridge	A-102	400	
1	Round Hill	A-305	350	
2	Mianus	A-215	700	
3	Downtown	A-101	500	
4	Redwood	A-222	700	
5		A-201	900	
6	Brighton	A-217	750	
7		A-110	600	
8		A-218	700	

- 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



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

1. SEQUENTIAL FILE ORGANIZATION

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

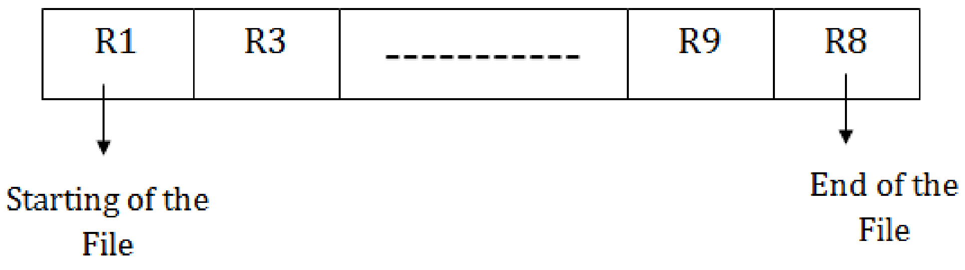
Deletion – use pointer chains

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

Sequential is the easiest method for file organization. In this method, files are stored sequentially. This method can be implemented in two ways:

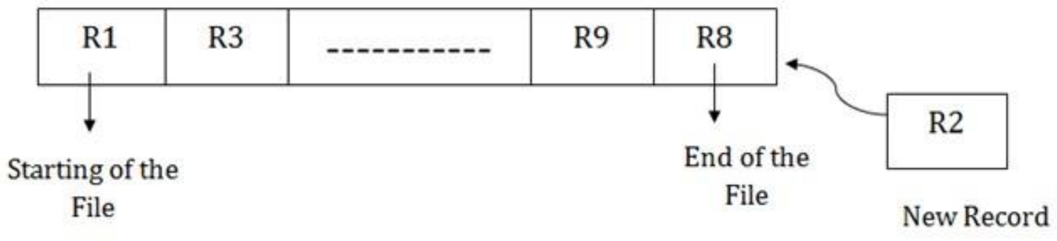
1. Pile File Method:

- It is a quite simple method. In this method, we store the record in a sequence, i.e., one after another. Here, the record will be inserted in the order in which they are inserted into tables.
- In case of updating or deleting of any record, the record will be searched in the memory blocks. When it is found, then it will be marked for deleting, and the new record is inserted.



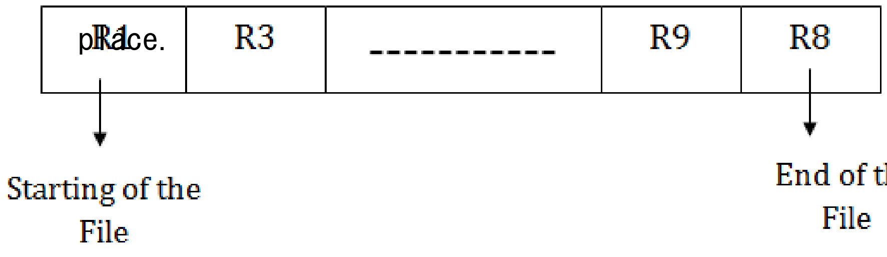
Insertion of the new record:

Suppose we have four records R1, R3 and so on upto R9 and R8 in a sequence. Hence, records are nothing but a row in the table. Suppose we want to insert a new record R2 in the sequence, then it will be placed at the end of the file. Here, records are nothing but a row in any table.



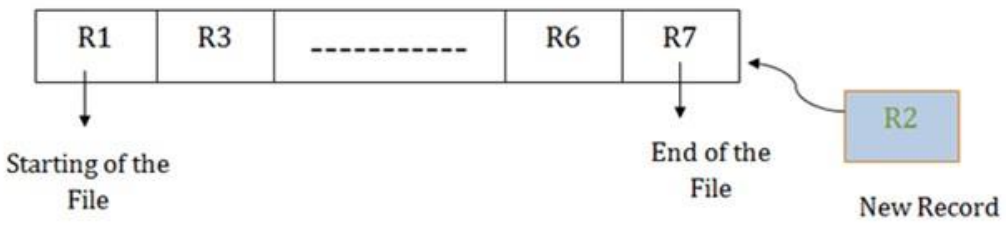
2. Sorted File Method:

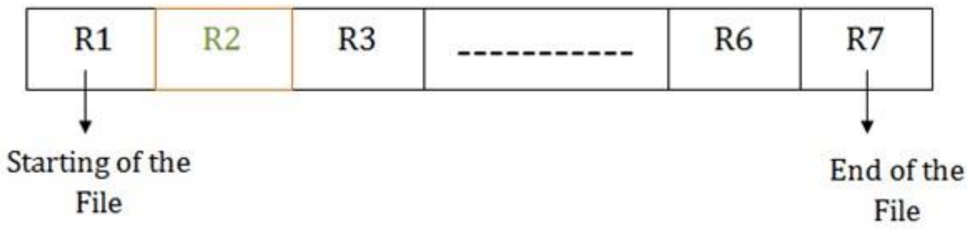
- In this method, the new record is always inserted at the file's end, and then it will sort the sequence in ascending or descending order. Sorting of records is based on any primary key or any other key.
- In the case of modification of any record, it will update the record and then sort the file, and lastly, the updated record is placed in the right place.



Insertion of the new record:

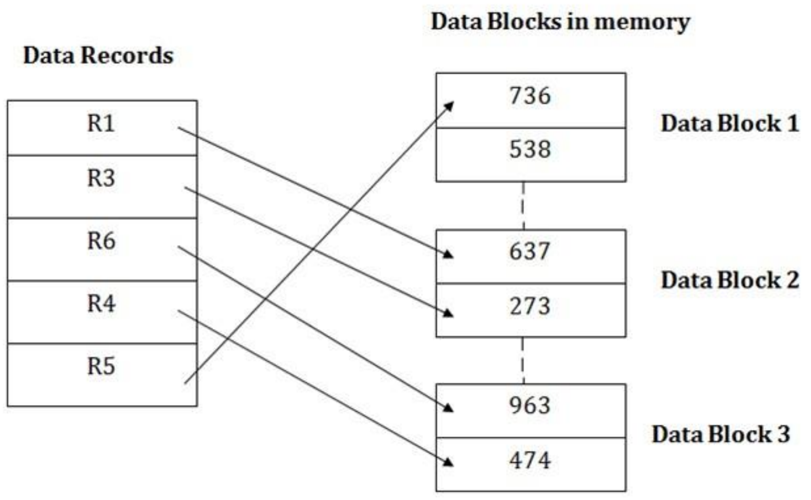
Suppose there is a preexisting sorted sequence of four records R1, R3 and so on upto R6 and R7. Suppose a new record R2 has to be inserted in the sequence, then it will be inserted at the end of the file, and then it will sort the sequence.





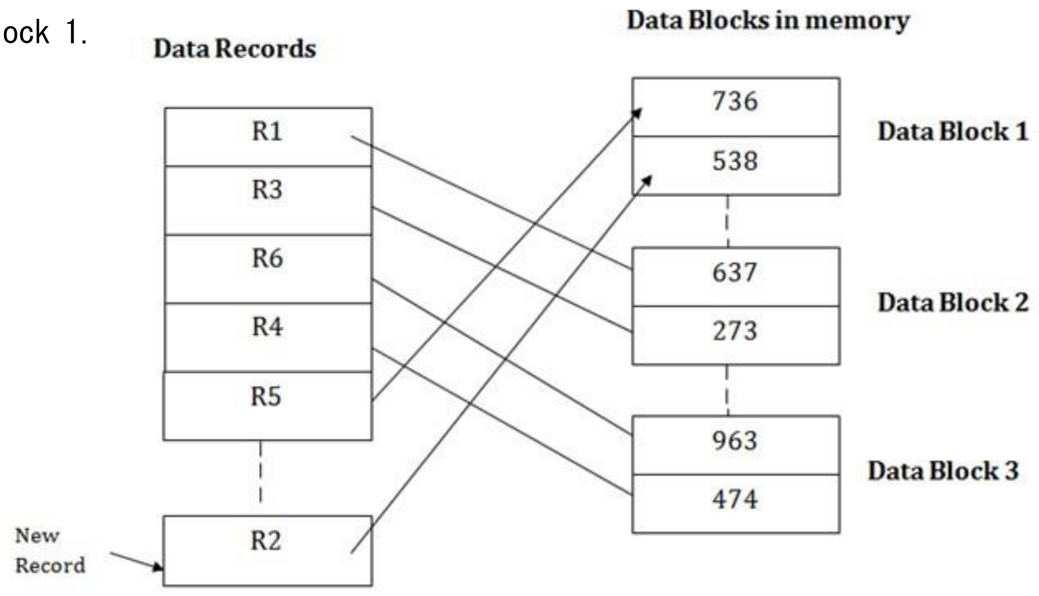
2. HEAP FILE ORGANIZATION

- It is the simplest and most basic type of organization.
- It works with data blocks.
- In heap file organization, the records are inserted at the file's end. When the records are inserted, it doesn't require the sorting and ordering of records.
- When the data block is full, the new record is stored in some other block. This new data block need not to be the very next data block, but it can select any data block in the memory to store new records. The heap file is also known as an unordered file.
- In the file, every record has a unique id, and every page in a file is of the same size. It is the DBMS responsibility to store and manage the new records.



Insertion of a new record

Suppose we have five records R1, R3, R6, R4 and R5 in a heap and suppose we want to insert a new record R2 in a heap. If the data block 3 is full then it will be inserted in any of the database selected by the DBMS, let's say data block 1.



If we want to search, update or delete the data in heap file organization, then we need to traverse the data from staring of the file till we get the requested record.

If the database is very large then searching, updating or deleting of record will be time- consuming because there is no sorting or ordering of records. In the heap file organization, we need to check all the data until we get the requested record.

4. 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

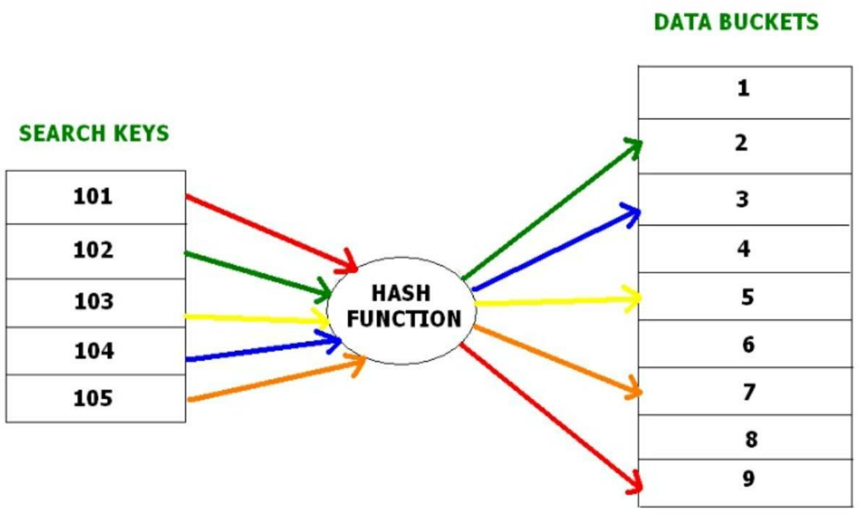
-Static Hashing

-Dynamic Hashing

4.1 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



Operations

(i) **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)

(ii) **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.

(iii) **Delete** – This is simply a search followed by a deletion operati on.

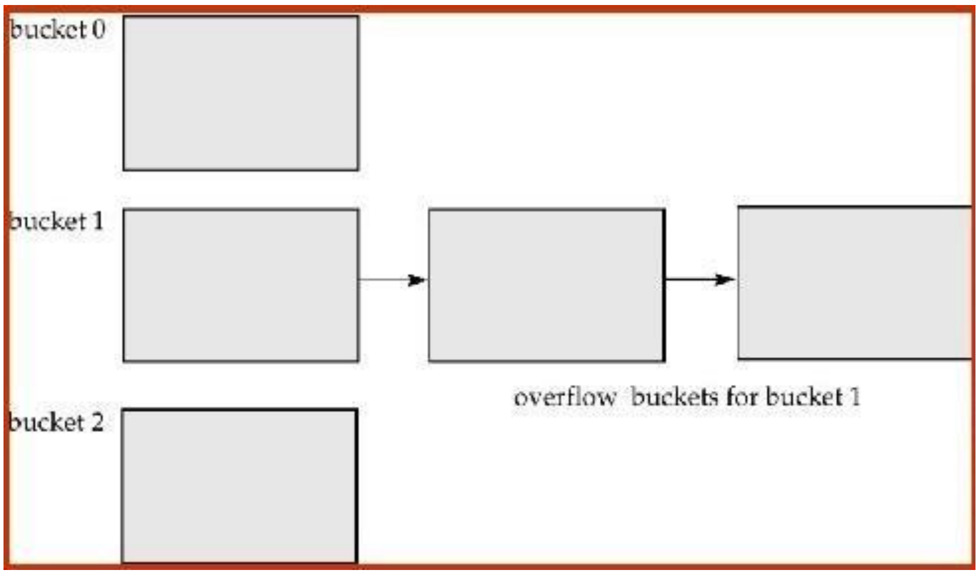
Bucket overflow can occur because of

- 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.

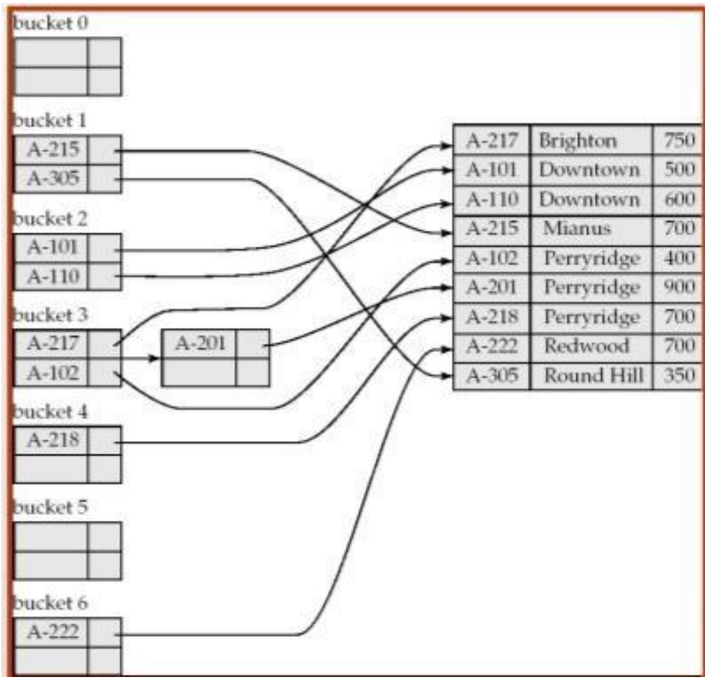
Overflow chaining

- 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.

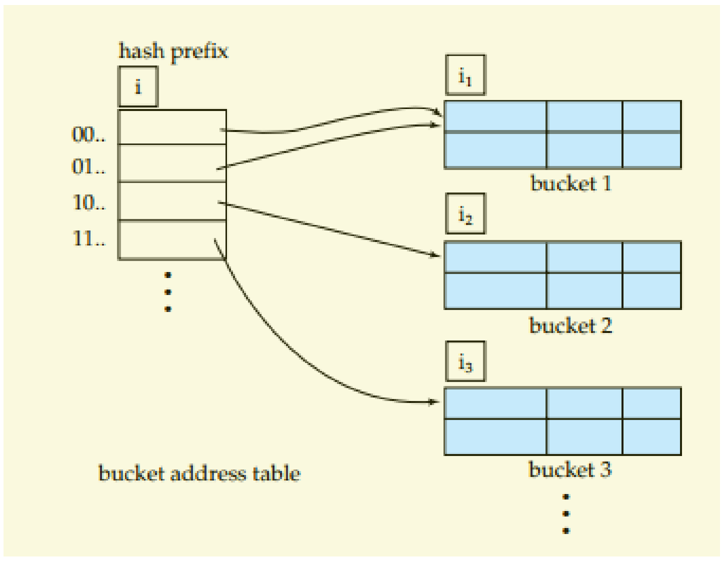
- o Databases grow with time. If initial number of buckets is too small, performance will degrade due to too much overflows.
- o If file size at some point in the future is anticipated and number of buckets allocated accordingly, significant amount of space will be wasted initially.
- o If database shrinks, again space will be wasted.
- o These problems can be avoided by using techniques that allow the number of buckets to be modified dynamically.

4.2 Dynamic Hashing

- o Good for database that grows and shrinks in size
- o Allows the hash function to be modified dynamically
- o Extendable hashing - one form of dynamic hashing
  - o Hash function generates values over a large range
  - o Typically b-bit integers, with b = 32.
  - o At any time use only a prefix of the hash function to index into a table of bucket addresses.
  - o Let the length of the prefix be i bits, 0 <= i <=32.
  - o Bucket address table size = 2i. Initially i = 0
  - o Value of i grows and shrinks as the size of the database grows and shrinks.
  - o Multiple entries in the bucket address table may point to a bucket.
  - o Thus, actual number of buckets is < 2i
  - o The number of buckets also changes dynamically due to coalescing and splitting of buckets.

4.3 General Extendable Hash

In this structure, i2 = i3 = i, whereas i1 = i - 1



Insertion in Extendable Hash Structure

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

- If i > ij (more than one pointer to bucket j)
  - Allocate a new bucket z, and set ij = iz = (ij + 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 Kj and insert record in the bucket (further splitting is required if the bucket is still full)
- If i = ij (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 Kj
  - Now i > ij so use the first case above.

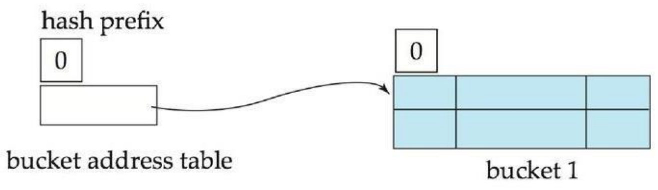
Deletion in Extendable Hash Structure

To delete a key value,

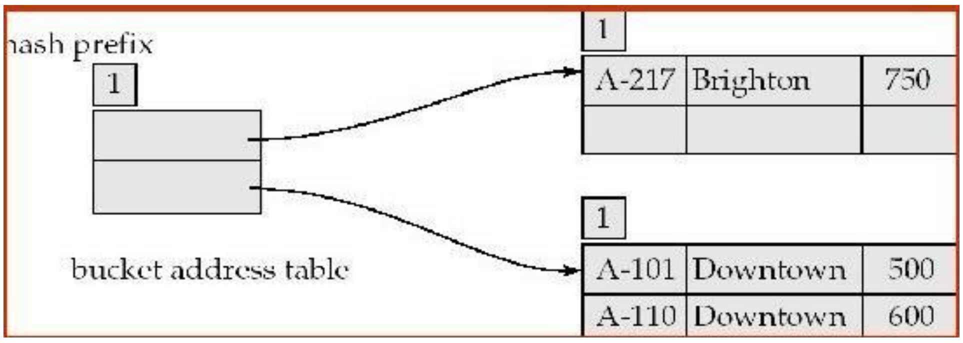
- Locate it in its bucket and remove it.
- The bucket itself can be removed if it becomes empty (with appropriate updates to the bucket address table).
- Coalescing of buckets can be done (can coalesce only with a "buddy" bucket having same value of ij and same ij -1 prefix, if it is present)
- Decreasing 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

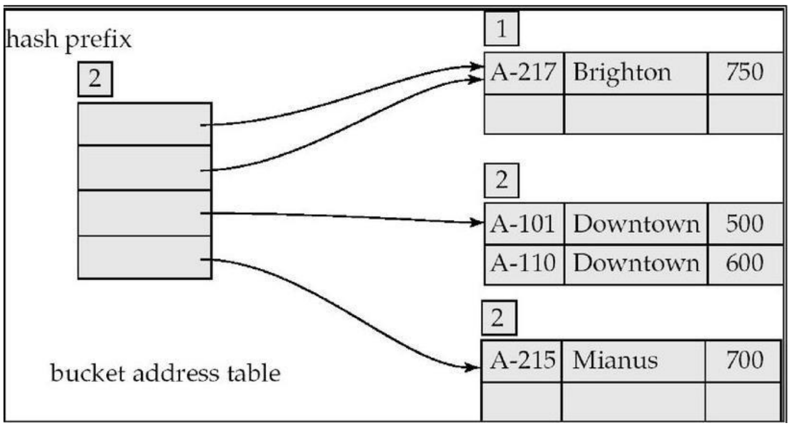
- Initial hash structure; bucket size = 2



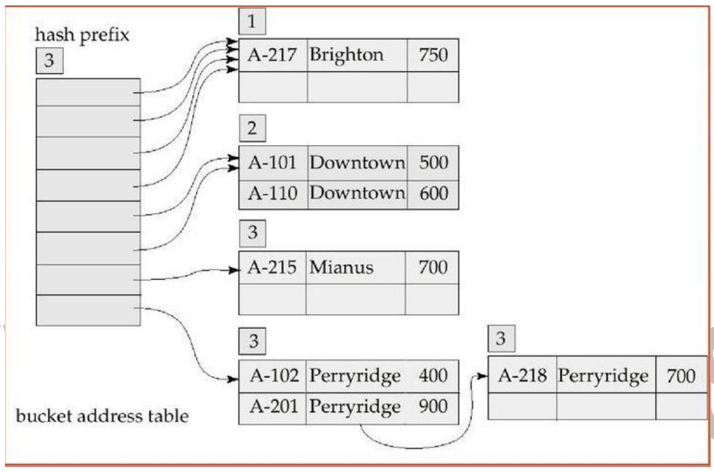
(i) **Hash structure after insertion of one Brighton and two Downtown records**



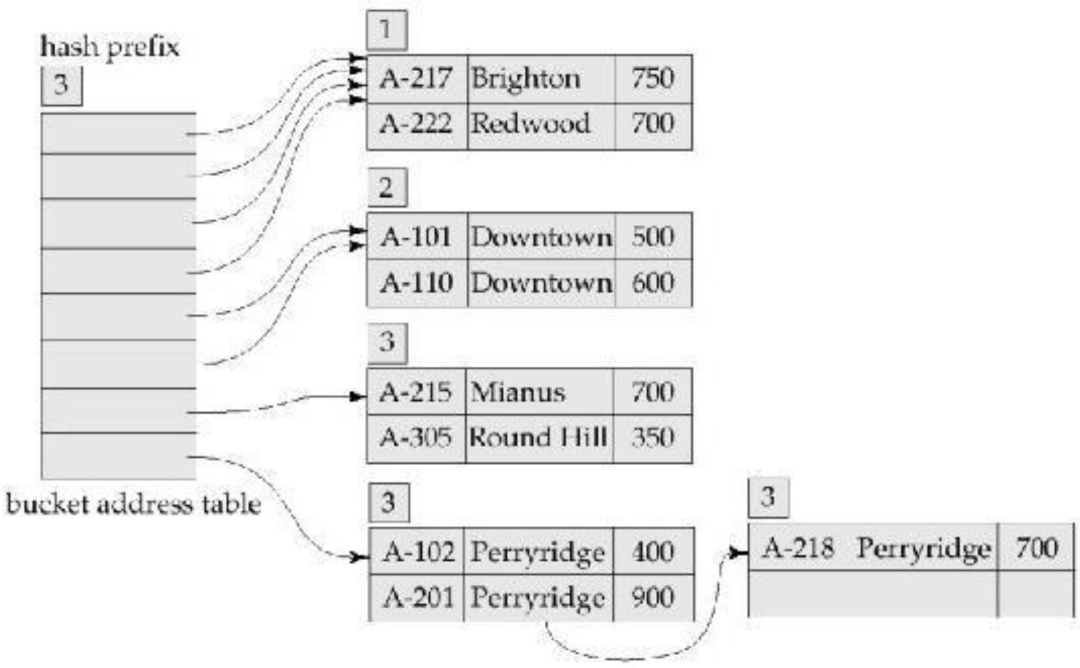
(ii) **Hash structure after insertion of Mianus record**



(iii) Hash structure after insertion of three Perryridge records



(iv) Hash structure after insertion of Redwood and Round Hill records



Use of Extendable Hash Structure

To locate the bucket containing search-key Kj:

1. Compute h(Kj) = 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 Kj
  - 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.

7. HEURISTIC-BASED QUERY OPTIMIZATION

In general, many different relational algebra expressions-and hence many different query trees can be **equivalent**; that is, they can represent the *same query*.

The query parser will typically generate a standard **initial query tree** to correspond to an SQL query, without doing any optimization.

For example, for a SELECT-PROJECT-JOIN query, such as Q2, the initial tree is shown in Figure. The CARTESIAN PRODUCT of the relations specified in the FROM clause is first applied; then the selection and join conditions of the WHERE clause are applied, followed by the projection on the SELECT clause attributes.

Such a canonical query tree represents a relational algebra expression that is *very inefficient if executed directly,* because of the CARTESIAN PRODUCT (×) operations.

The heuristic query optimizer will transform this initial query tree into an equivalent **final query tree** that is efficient to execute.

The optimizer must include rules for *equivalence among relational algebra expressions* that can be applied to transform the initial tree into the final, optimized query tree. First we discuss informally how a query tree is transformed by using heuristics, and then we discuss general transformation rules and show how they can be used in an algebraic heuristic optimizer.

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

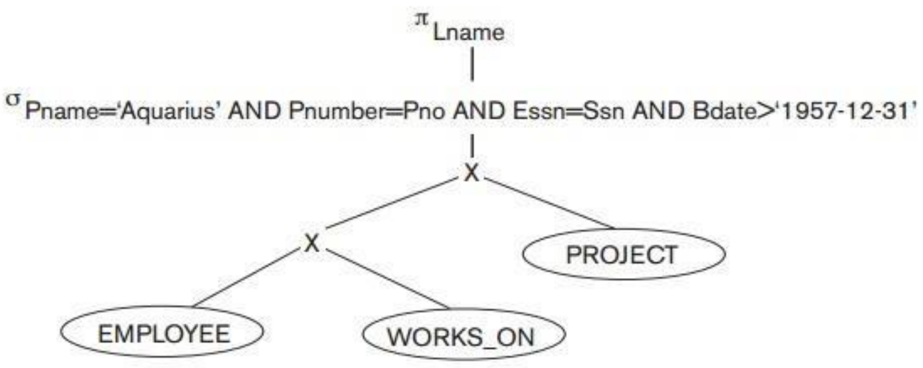
6. Identify sub-trees that represent groups of operations that can be executed by a single algorithm

Query ″Find the last names of employees born after 1957 who work on a project named _Aquarius ‘.″

SQL

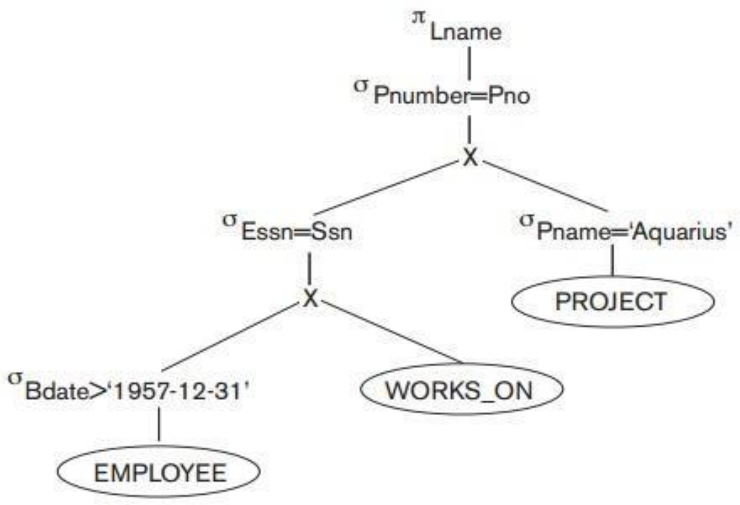
```
SELECT LNAME
FROM EMPLOYEE, WORKS_ON, PROJECT
WHERE PNAME=_Aquarius ‘AND PNUMBER=PNO AND ESSN=SSN AND BDATE._1957-12- 31 ‘;
```

(a)



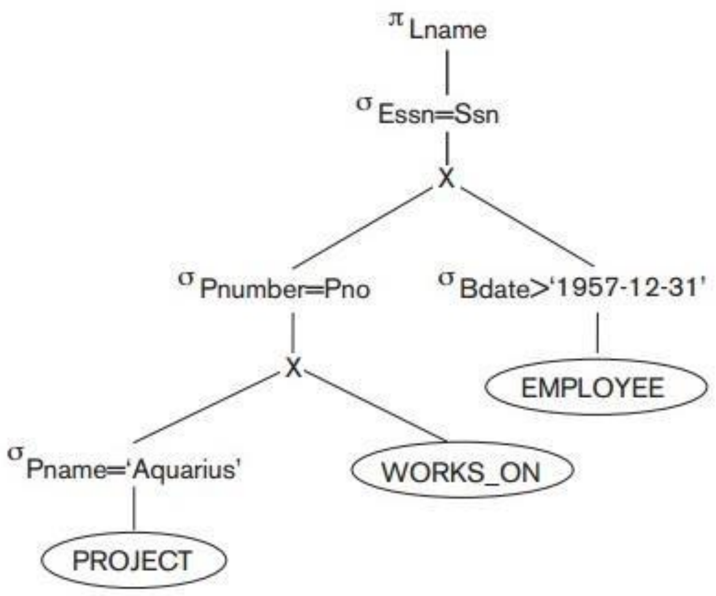
π_{Lname}
$\sigma_{\text{Pname='Aquarius' AND Pnumber=Pno AND Essn=Ssn AND Bdate>'1957-12-31'}}$
X
X
PROJECT
EMPLOYEE
WORKS_ON

(b)

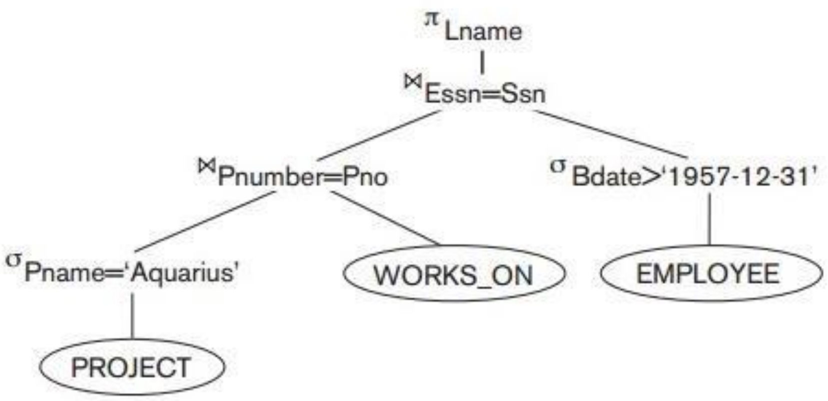


π_{Lname}
$\sigma_{\text{Pnumber=Pno}}$
X
$\sigma_{\text{Essn=Ssn}}$
$\sigma_{\text{Pname='Aquarius'}}$
X
PROJECT
$\sigma_{\text{Bdate>'1957-12-31'}}$
WORKS_ON
EMPLOYEE

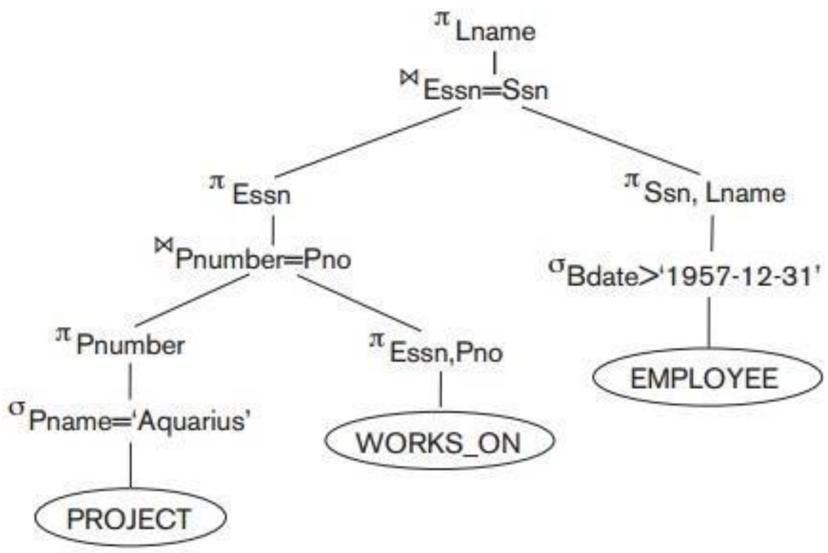
(c)


π Lname
σ Essn=Ssn
X
σ Pnumber=Pno
σ Bdate>'1957-12-31'
X
EMPLOYEE
σ Pname='Aquarius'
WORKS_ON
PROJECT

(d)


π Lname
⋈ Essn=Ssn
⋈ Pnumber=Pno
σ Bdate>'1957-12-31'
σ Pname='Aquarius'
WORKS_ON
EMPLOYEE
PROJECT

(e)


π Lname
⋈ Essn=Ssn
π Essn
π Ssn, Lname
⋈ Pnumber=Pno
σ Bdate>'1957-12-31'
π Pnumber
π Essn,Pno
EMPLOYEE
σ Pname='Aquarius'
WORKS_ON
PROJECT

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 :

[log2(nBlocks(R))], if equality condition is on key attribute, because SCA(R) = 1 in this case. [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.

3. INDEXED FILE ORGANIZATION

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.

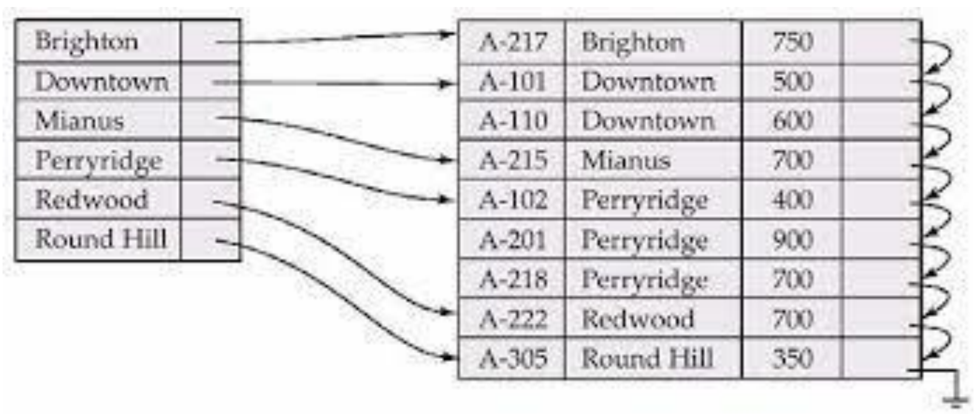
Primary index: in a sequentially ordered file, the index whose search key specifies the sequential order of the file.

Secondary index: an index whose search key specifies an order different from the sequential order of the file.

- Dense index
- Sparse index

(I) Dense Index Files

Dense index – Index record appears for every search-key value in the



Brighton
Downtown
Mianus
Perryridge
Redwood
Round Hill

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

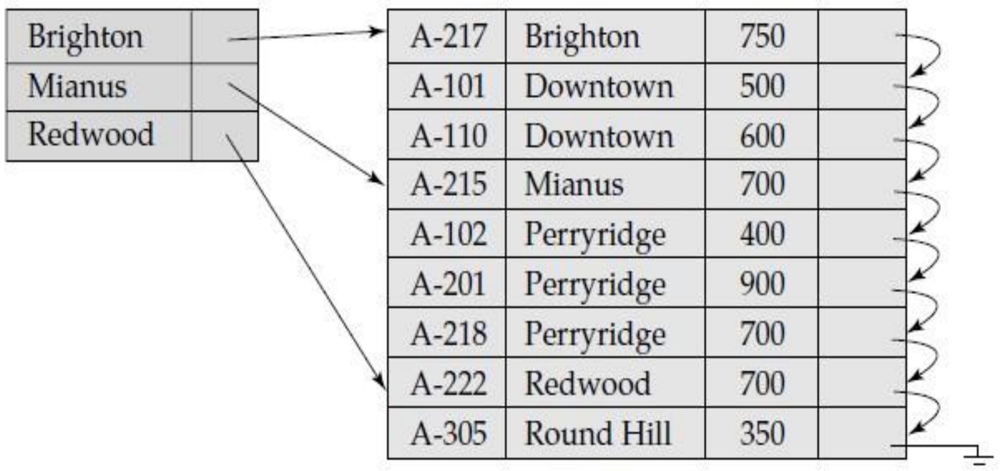
(II) Sparse Index Files

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



(III) Multilevel Index

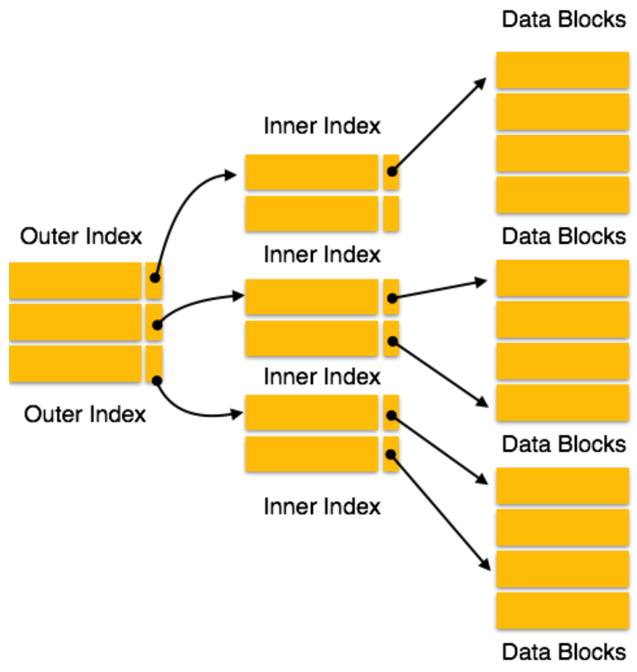
If primary index does not fit in memory, access becomes expensive.

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.

(i) 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.

(ii) **Single-level index insertion**:

- Perform a lookup using the search-key value appearing in the record to be inserted.
- **Dense indices**

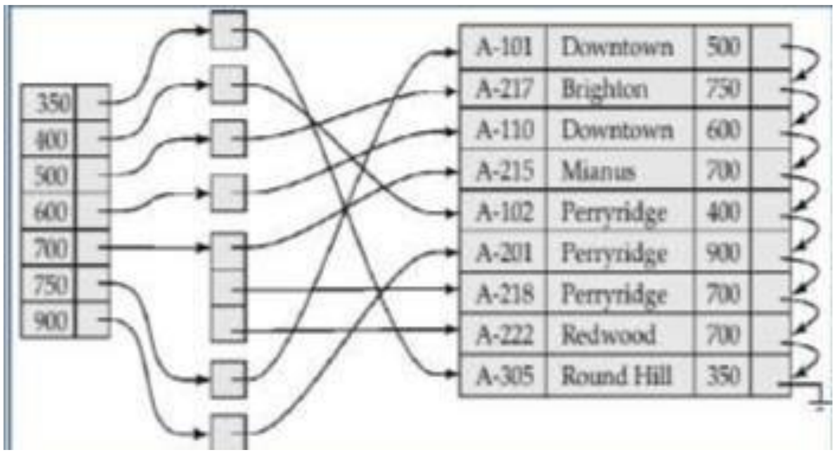
  if the search-key value does not appear in the index, insert it.
- **Sparse indices**

  - if index stores an entry for each block of the file, no change needs to be made

-key value appearing in to the index unless a new block is created. In this case, the first search

the new block is inserted into the index.

Secondary Index on balance field of account



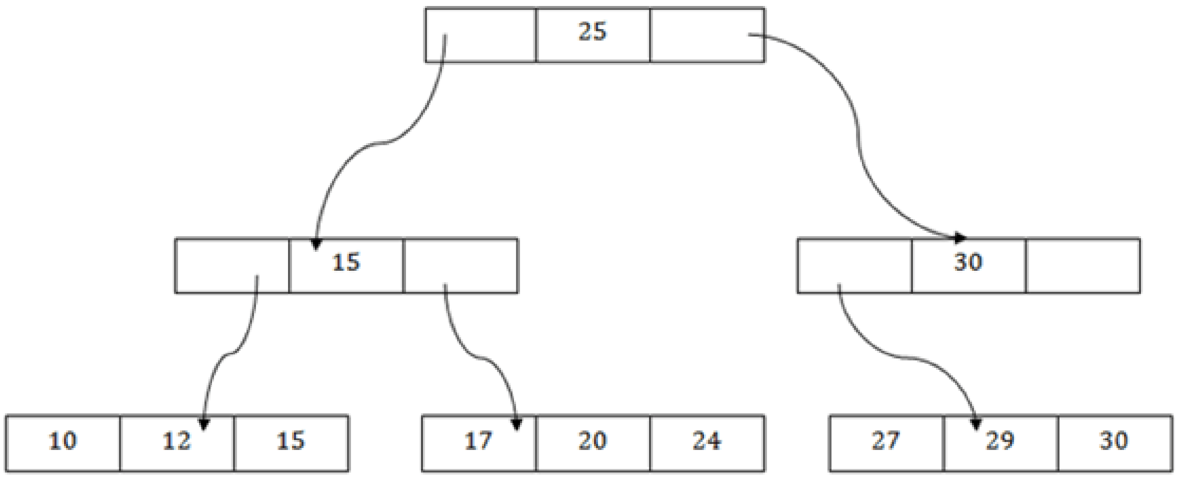
Primary and Secondary Indices

- 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

1. B+-TREE INDEX FILES

B+ File Organization

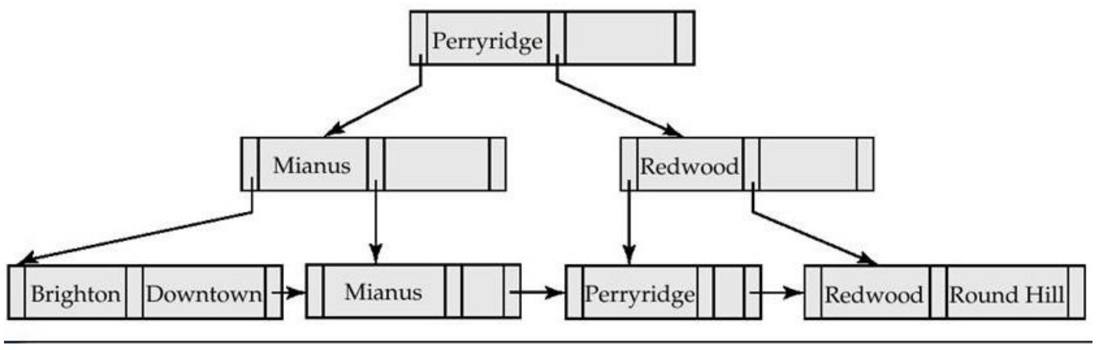
- B+ tree file organization is the advanced method of an indexed sequential access method. It uses a tree-like structure to store records in File.
- It uses the same concept of key-index where the primary key is used to sort the records. For each primary key, the value of the index is generated and mapped with the record.
- The B+ tree is similar to a binary search tree (BST), but it can have more than two children. In this method, all the records are stored only at the leaf node. Intermediate nodes act as a pointer to the leaf nodes. They do not contain any records.



The above B+ tree shows that:

- There is one root node of the tree, i.e., 25.
- There is an intermediary layer with nodes. They do not store the actual record. They have only pointers to the leaf node.
- The nodes to the left of the root node contain the prior value of the root and nodes to the right contain next value of the root, i.e., 15 and 30 respectively.
- There is only one leaf node which has only values, i.e., 10, 12, 17, 20, 24, 27 and 29.
- Searching for any record is easier as all the leaf nodes are balanced.
- In this method, searching any record can be traversed through the single path and accessed easily
- **Example for 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:

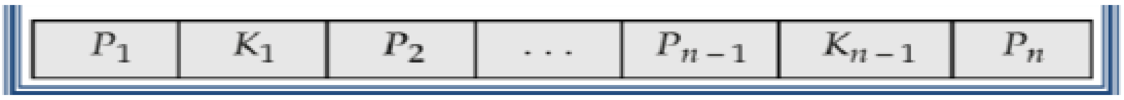
Extra insertion and deletion overhead, space overhead.

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



- Ki are the search-key values

- Pi 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

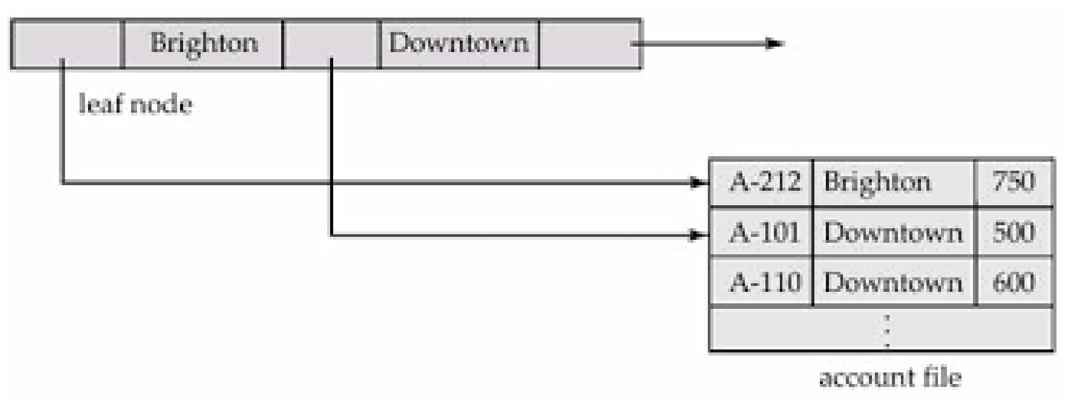
ordered K1 < K2 < K3 < . . .

< Kn-1

Properties of Leaf Nodes

For i = 1, 2, . . ., n-1, pointer Pi either points to a file record with search-key value Ki, or to a bucket of pointers to file records, each record having search-key value Ki.

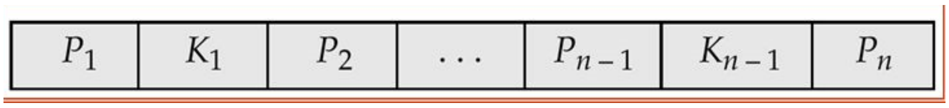
Pn points to next leaf node in search-key order



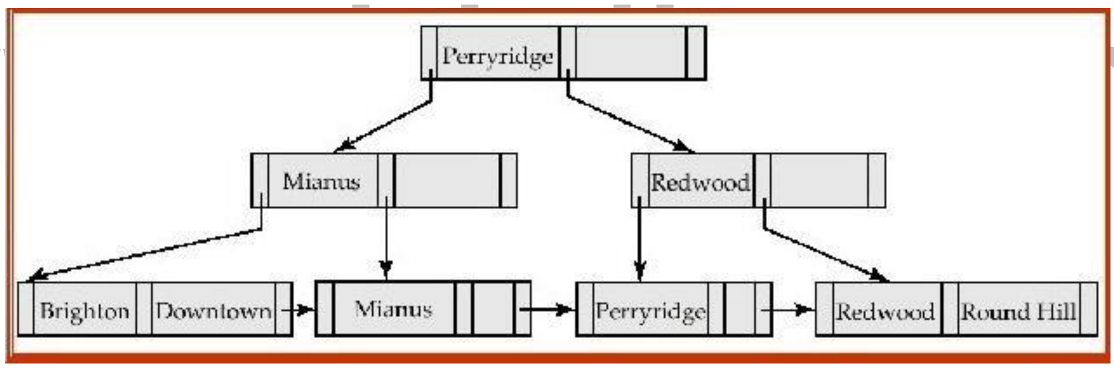
Non-Leaf Nodes in B+-Trees

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

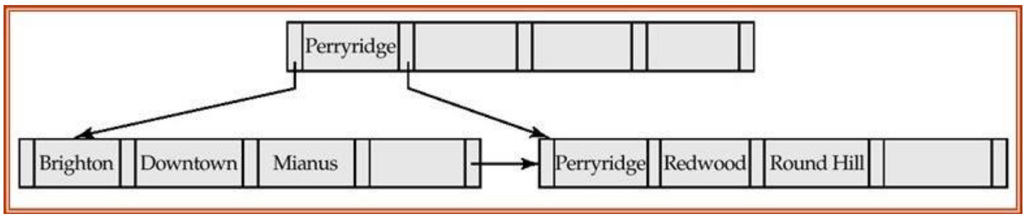
- All the search-keys in the subtree to which P1 points are less than K1.



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 ((n/2⌉ 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.

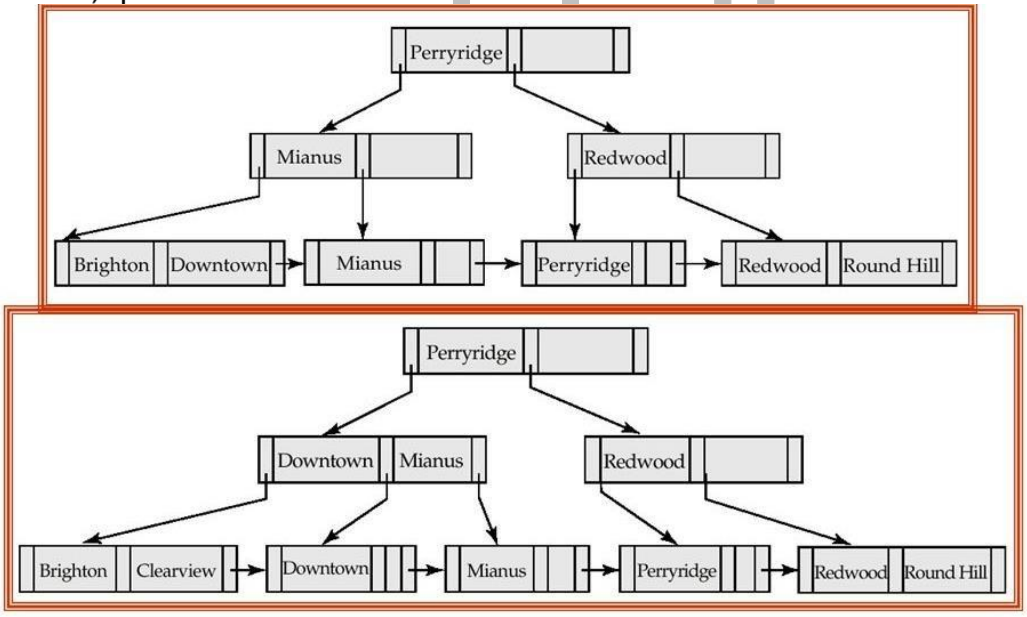
Updates on B+-Trees: Insertion

- 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

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

otherwise, split the node.



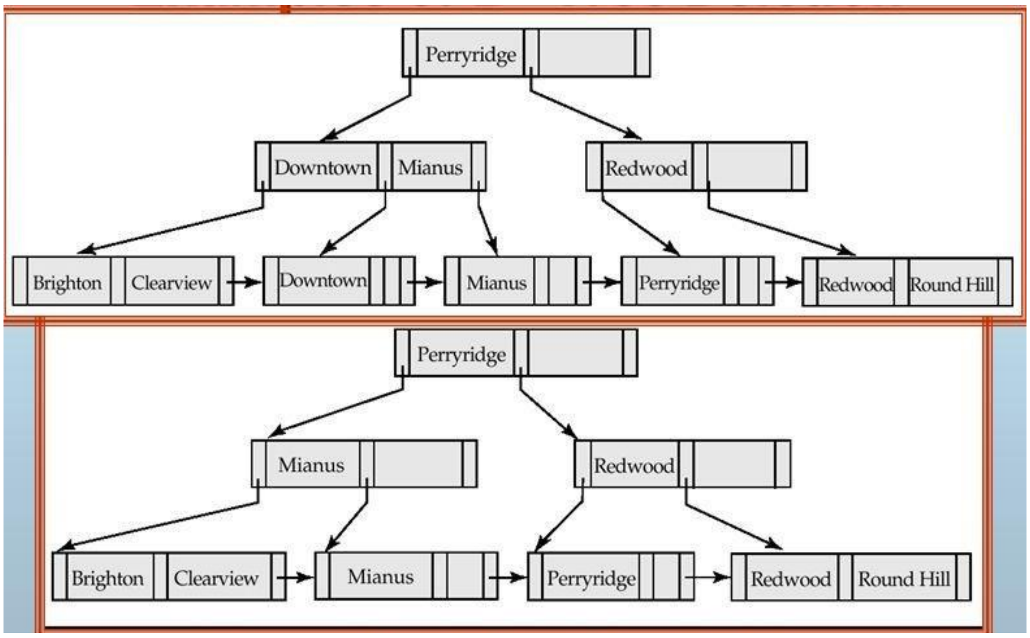
B$^+$-Tree before and after insertion of "Clearview"

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

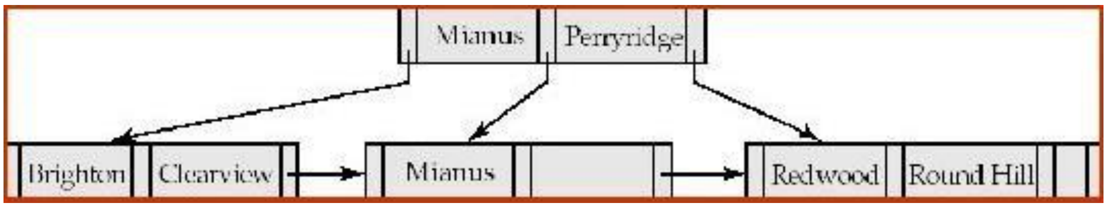
- Insert all the search-key values in the two nodes into a single node (the one on the left), and delete the other node.
- Delete the pair (Ki-1, Pi), where Pi is the pointer to the deleted node, from its parent,

  recursively using the above



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.



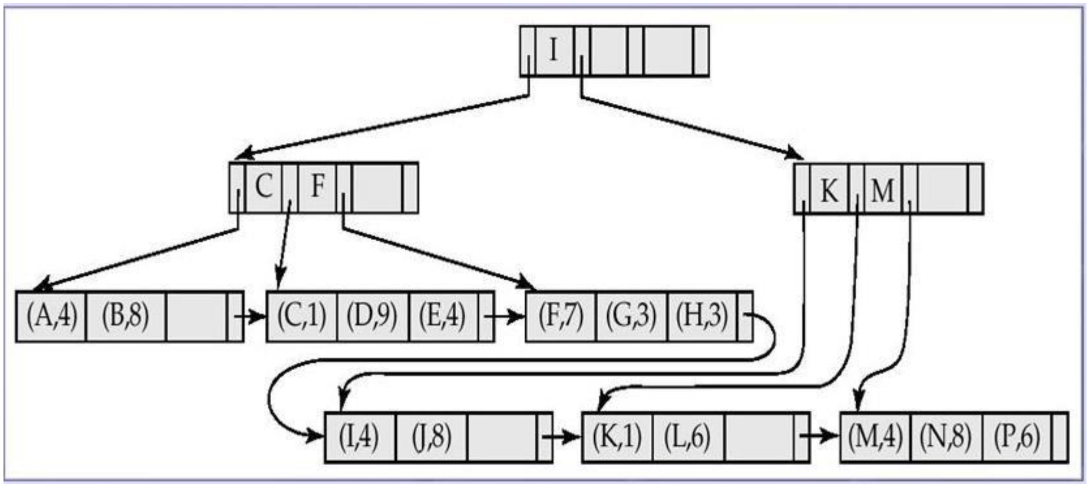
- Node with "Perryridge" becomes empty and merged with its sibling.
- Root node then had only one child, and was deleted and its child became the new root node

B+-Tree File Organization

- The leaf nodes in a B+-tree file organization store records, instead of pointers.

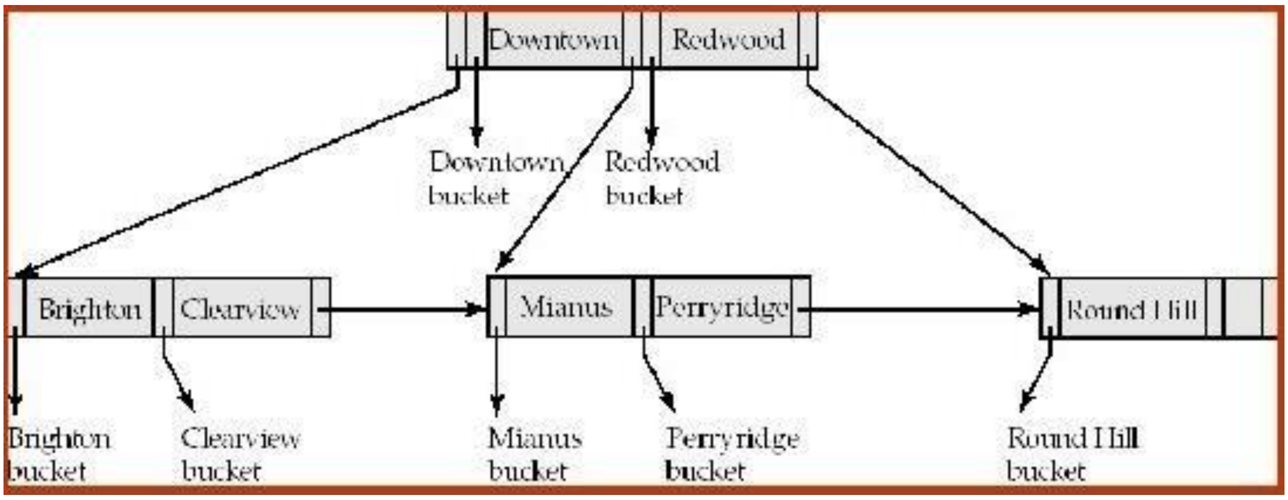
- Since records are larger than pointers, the maximum number of records that can be stored in a leaf node is less than the number of pointers in a nonleaf node.

- 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.

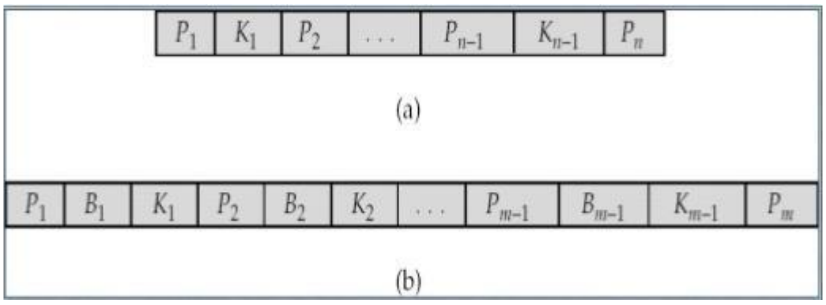


2. B -TREE INDEX FILES

- 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 Bi are the bucket or file record pointers.

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:

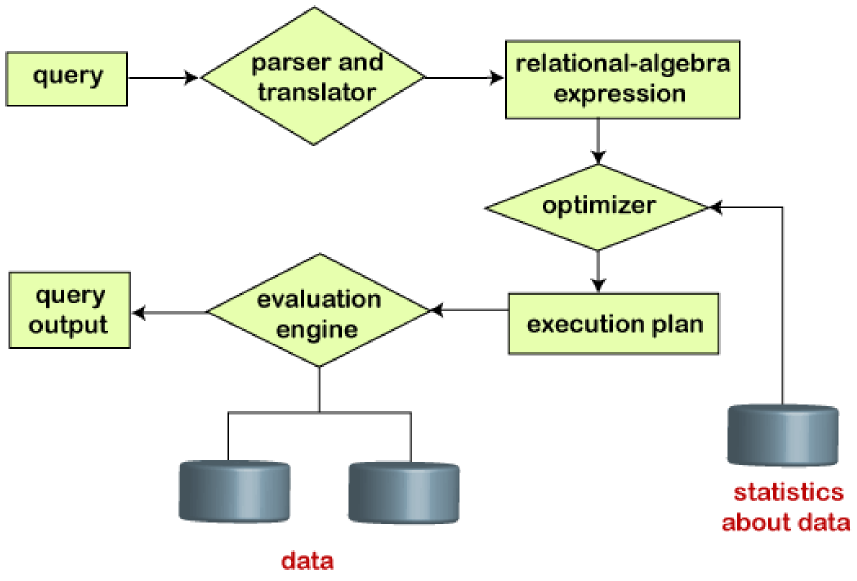
- Only small fraction of all search-key values are found early
- Non-leaf nodes are larger, so fan-out is reduced (no. of pointers). Thus, B-Trees typically have greater depth than corresponding B+-Tree
- Insertion and deletion more complicated than in B+-Trees
- Implementation is harder than B+-Trees.

5. QUERY PROCESSING IN DBMS.

Query Processing is the activity performed in extracting data from the database.

In query processing, it takes various steps for fetching the data from the database. The steps involved are:

1. Parsing and translation
2. Optimization



Steps in query processing

The query processing works in the following way:

Parsing and Translation

- The scanning, parsing, and validating module produces an internal representation of the

  query. 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.
- The code generator generates the code to execute the plan. The runtime database processor runs the generated code to produce the query result.
- Relational algebra is well suited for the internal representation of a query.

The translation process in query processing is similar to the parser of a query. When a user executes any query, for generating the internal form of the query, the parser in the system checks the syntax of the query, verifies the name of the relation in the database, the tuple, and finally the required attribute value. The parser creates a tree of the query, known as 'parse-tree.' Further, translate it into the form of relational algebra. With this, it evenly replaces all the use of the views when used in the query.

It is done in the following steps:

Step-1:

Parser: During parse call, the database performs the following checks- Syntax check, Semantic check and Shared pool check, after converting the query into relational algebra.

Parser performs the following checks as (refer detailed diagram):

1. **Syntax check** – concludes SQL syntactic validity.

   Example: SELECT * FORM employee

   Here error of wrong spelling of FROM is given by this check.
2. **Semantic check** – determines whether the statement is meaningful or not. Example: query contains a table name which does not exist is checked by this check.
3. **Shared Pool check** – Every query possess a hash code during its execution. So, this check determines existence of written hash code in shared pool if code exists in shared pool then database will not take additional steps for optimization and execution.

Hard Parse and Soft Parse –

If there is a fresh query and its hash code does not exist in shared pool then that query has to pass through from the additional steps known as hard parsing otherwise if hash code exists then query does not passes through additional steps. It just passes directly to execution engine (refer detailed diagram). This is known as soft parsing.

Hard Parse includes following steps - Optimizer and Row source generation.

Step-2:

Optimizer: During optimization stage, database must perform a hard parse atleast for one unique DML statement and perform optimization during this parse. This database never optimizes DDL unless it includes a DML component such as subquery that require optimization.

It is a process in which multiple query execution plan for satisfying a query are examined and most efficient query plan is satisfied for execution. Database catalog stores the execution plans and then optimizer passes the lowest cost plan for execution.

Step-3:

Execution Engine: Finally runs the query and display the required result.

Thus, we can understand the working of a query processing in the below-described diagram:

Suppose a user executes a query. As we have learned that there are various methods of extracting the data from the database. In SQL, a user wants to fetch the records of the employees whose salary is greater than or equal to 10000. For doing this, the following query is undertaken:

SELECT EMP_NAME FROM EMPLOYEE WHERE SALARY>10000;

Thus, to make the system understand the user query, it needs to be translated in the form of relational algebra. We can bring this query in the relational algebra form as:

- $\sigma_{salary>10000}\ (\pi_{Emp\_Name}(Employee))$
- $\pi_{Emp\_Name}(\sigma_{salary>10000}\ (Employee))$

After translating the given query, we can execute each relational algebra operation by using different algorithms. So, in this way, a query processing begins its working.

Evaluation

For this, with addition to the relational algebra translation, it is required to annotate the translated relational algebra expression with the instructions used for specifying and evaluating each operation. Thus, after translating the user query, the system executes a query evaluation plan.

Query Evaluation Plan

- In order to fully evaluate a query, the system needs to construct a query evaluation plan.

- A query evaluation plan defines a sequence of primitive operations used for evaluating a query. The query evaluation plan is also referred to as **the query execution plan**.
- A **query execution engine** is responsible for generating the output of the given query. It takes the query execution plan, executes it, and finally makes the output for the user query.

Optimization

- The cost of the query evaluation can vary for different types of queries. Although the system is responsible for constructing the evaluation plan, the user does need not to write their query efficiently.
- Usually, a database system generates an efficient query evaluation plan, which minimizes its cost. This type of task performed by the database system and is known as Query Optimization.
- For optimizing a query, the query optimizer should have an estimated cost analysis of each operation. It is because the overall operation cost depends on the memory allocations to several operations, execution costs, and so on.

Finally, after selecting an evaluation plan, the system evaluates the query and produces the output of the query.
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: Π MAX SALARY (σDNO=5(EMPLOYEE))

The Outer block
SELECT LNAME, FNAME FROM EMPLOYEE WHERE SALARY > C

- Translated in: Π 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.

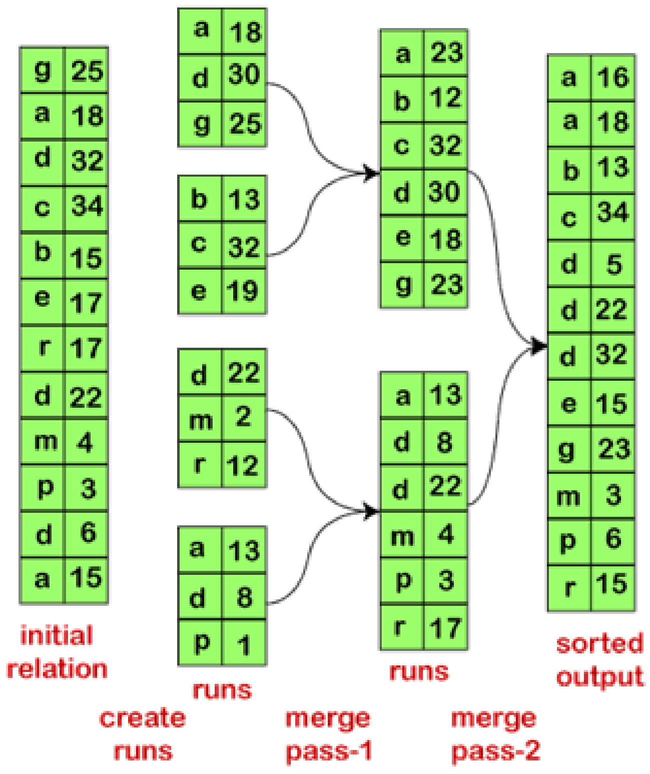
External Sorting

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



External sorting using sort-merge

UNIT IV - IMPLEMENTATION TECHNIQUES

1. 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

- These levels contain the following characteristics:
- It contains a set of physical disk drives.
- In this technology, the operating system views these separate disks as a single logical disk.
- In this technology, data is distributed across the physical drives of the array.
- Redundancy disk capacity is used to store parity information.
- In case of disk failure, the parity information can be helped to recover the data.

Motivation for RAID

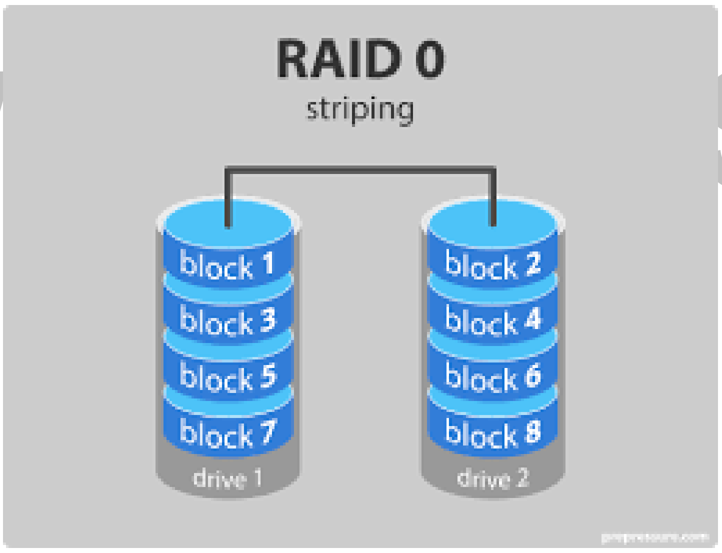
- Just as additional memory in form of cache, can improve the system performance, in the same way additional disks can also improve system performance.
- In RAID we can use an array of disks which operates independently since there are many disks, multiple I/O requests can be handled in parallel if the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

- 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

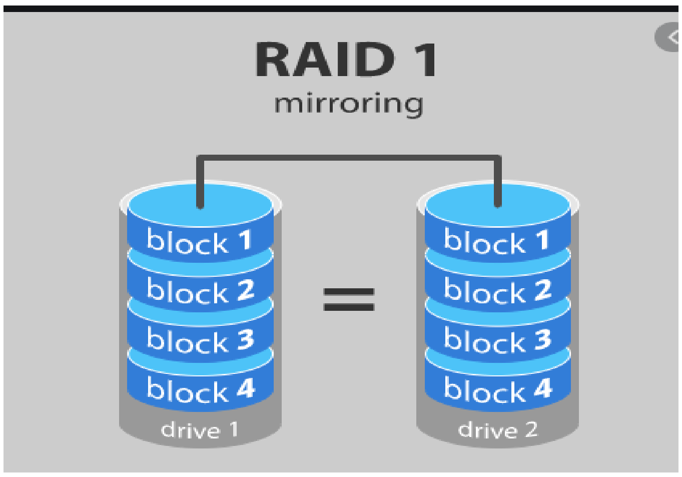
- I/O performance is greatly improved by spreading the I/O load across many channels & drives.
- Best performance is achieved when data is striped across multiple controllers with only one driver per controller

Disadvantages

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

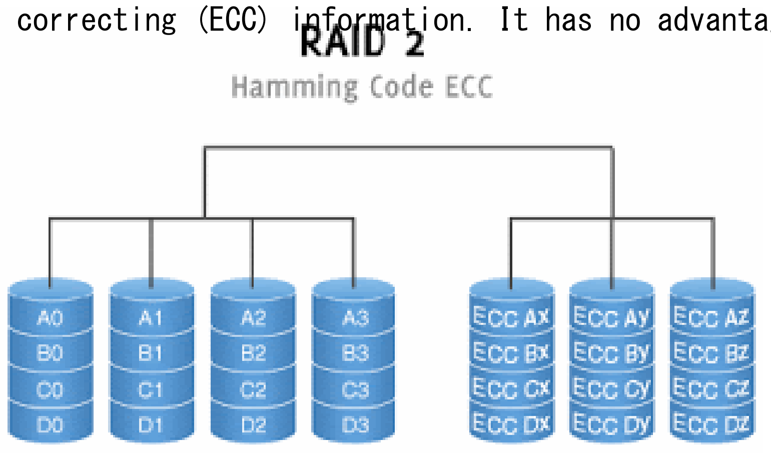
RAID LEVEL 1: Mirroring (or shadowing)

- 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



- Each bit of data word is written to a data disk drive (4 in this example: 0 to 3).

- Each data word has its Hamming Code ECC word recorded on the ECC disks.
- On Read, the ECC code verifies correct data or corrects single disk errors.

Advantages-

- On the fly' data error correction
- Extremely high data transfer rates possible
- The higher the data transfer rate required, the better the ratio of data disks to ECC disks.

Disadvantages-

- Very high ratio of ECC disks to data disks with smaller word sizes
- Entry level cost very high
- Requires very high transfer rate requirement to

  justify. No commercial implementations exist

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.
- The data block is subdivided ('striped') and written on the data disks. Stripe parity is generated on Writes, recorded on the parity disk and checked on Reads.
- RAID Level 3 requires a minimum of 3 drives to implement.

  Advantages

  -Very high Read data transfer rate

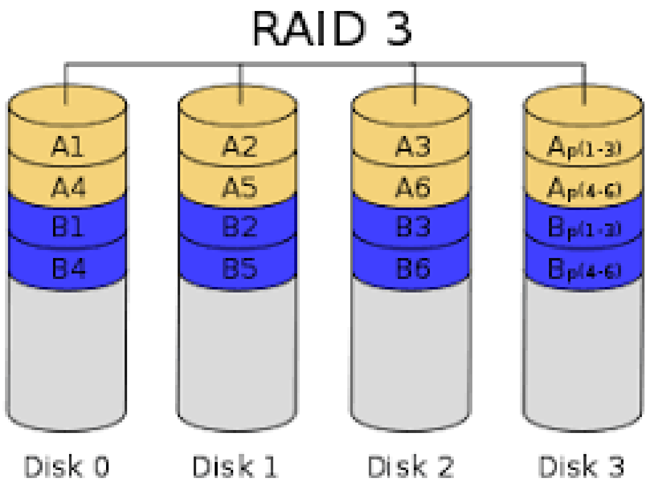
  -Very high Write data transfer rate

-Disk failure has an insignificant impact on throughput

-Low ratio of ECC (Parity) disks to data disks means high efficiency

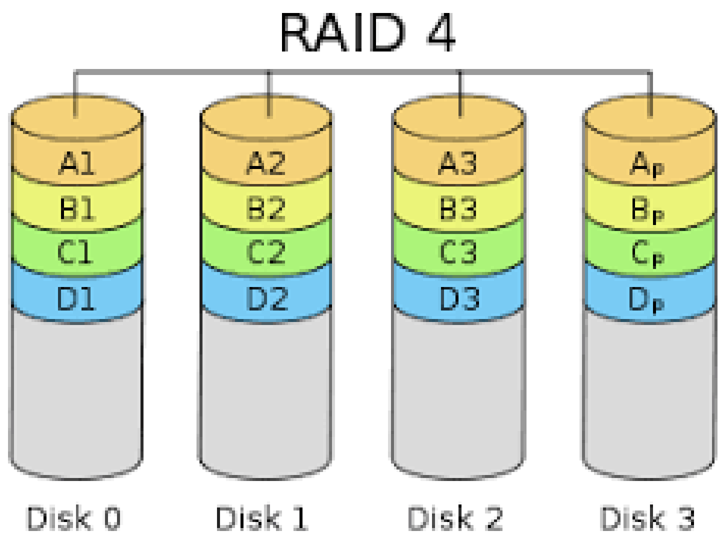
Disadvantages

- Transaction rate equal to that of a single disk drive at best (if spindles are
- Controller design is fairly complex.
- Very difficult and resource intensive to do as a



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



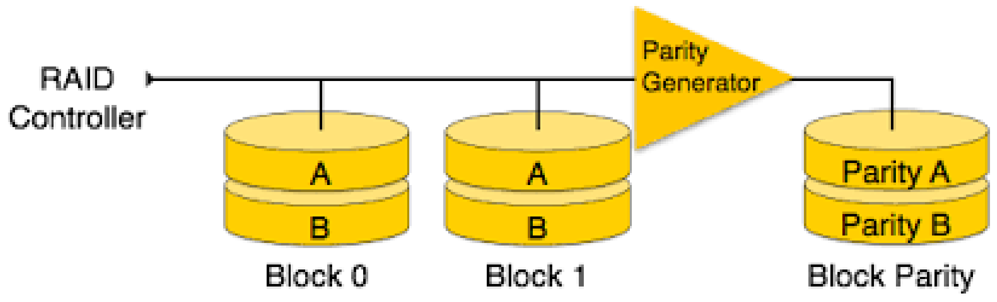
Each entire block is written onto a data disk. Parity for same rank blocks is generated on Writes, recorded on the parity disk and checked on Reads. RAID Level 4 requires a minimum of 3 drives to implement.

Advantages

- Very high Read data transaction rate
- Low ratio of ECC (Parity) disks to data disks means high efficiency
- High aggregate Read transfer rate
- Low ratio of ECC (Parity) disks to data disks means high efficiency

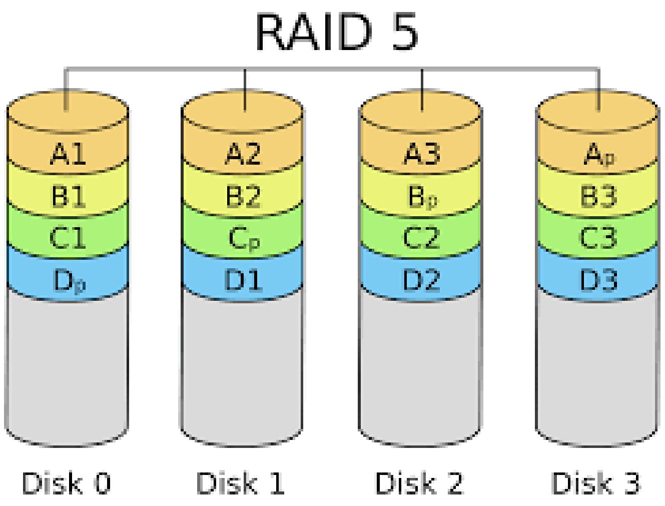
Disadvantages

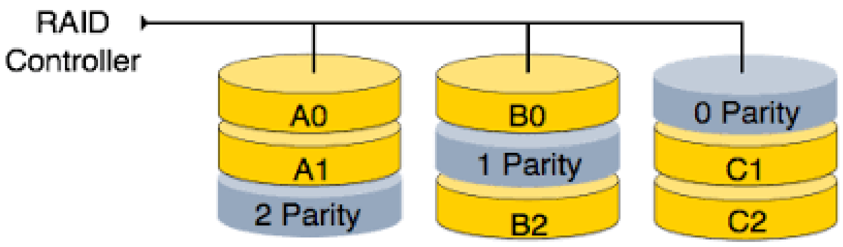
- Quite complex controller design
- Worst Write transaction rate and Write aggregate transfer rate
- Difficult and inefficient data rebuild in the event of disk failure
- Block Read transfer rate equal to that of a single disk



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.