CH17

17.1. Define the following terms: *indexing field*, *primary key field*, *clustering field*, *secondary key field*, *block anchor*, *dense index*, and *nondense (sparse) index*.

Define the following terms:-

Indexing field:-

Record structure is consisting of several fields. The record fields are used to construct an index. An index access structure is usually defined on a single field of a file. Any field in a file can be used to create an index and multiple indexes on different fields can be constructed on a field.

Primary key field:-

A primary key is the ordering key field of the file. A field that is uniquely identifies a record.

Clustering field:-

A secondary index is also an ordered field with two fields. (like a primary index). The first field is of the same data type as some non-ordering field of the data file that is an indexing field. If the secondary access structure uses a key field, which has a distinct value for every record. Therefore, it is called as secondary key field.

Block anchor:-

The total number of entries in the index is the same as the number of disk block in the ordered data file.

The first record in each block of the data file is called as block anchor.

Dense index:

An index has an index entry for every search key value (and hence every record) in the data file. Index record contains the pointer and search key value to the records on the disk

Non-dense:-

An index has entries for only some of the search values.

17.2. What are the differences among primary, secondary, and clustering indexes? How do these differences affect the ways in which these indexes are implemented? Which of the indexes are dense, and which are not?

Differences among primary secondary and clustering indexes:-

Primary index	Clustering index	Secondary index
Primary index is specifies on the ordering key fields of an ordered file of a records.	The file with numerous records and have the same value for the ordering field.	Secondary index specifies on any non-ordering field.it has a unique value.
Number of index entries that blocks in data file.	Number of index entries that distinct index field values.	Number of index entries that records in data file.
Every record has unique value for that field.	It has at most one physical ordering field. So, it can have at most one primary index or one clustering index, but not both.	A file have several secondary indexes and in addition to its primary access method.
Primary index is non-dense.	Clustering index is non-dense.	Secondary index is dense.

17.3. Why can we have at most one primary or clustering index on a file, but several secondary indexes?

A file which is in an order has some fixed size of the records with some key fields is said to be the primary index. But the clustering index in which it has a block pointer and the data with a field of the same type as the clustering field.

Adding or removing records in the file cannot be done easily. It has some problems in which the data records are physically ordered.

To overcome this problem, a whole block can be reserved for each of the clustering fields.

A file which is not in an order is said to be secondary index. It can be defined on a single key field with a unique value and on a non-key field with repeated values.

The following is the reason behind why there are at most one primary or clustering indexes whereas several indexes for secondary index:

• Primary and clustering index can use a single key field such that both of them cannot be there in a file but for secondary index, a unique value can be taken as a key field in every records or a non-key field with the repeated values in which the pointers will point to another block that have pointers to the repeated values.

17.4. How does multilevel indexing improve the efficiency of searching an index file?

Multilevel indexing improves the efficiency of searching an indexing file.

• In multilevel indexing, the main idea is to reduce the blocks of the index that are searched.

• It is the blocking factor for the index.

So, the search space is reduced much faster.

A Multi-level defines the index file that will be referred first with an ordered file with a distinct k value.

• By using single level index, create the primary index and then create the second-level, third-level and so on.

• So that the multi-level index can be created with the single index blocks.

For improving the efficiency of searching the index file, multilevel index in is follows the following steps:

Step1:

- Multilevel index considers the index file. The distinct value with an ordered file for each key k (i)

Step 2:

- In first level, create a primary index.
- It is called primary index.
- Also, use block anchors.
- So, there is one entry in the level for each block.
- Hence, the second level blocking factor before, is some as the first level of the index.
- Here, before the blocking factor $= \mathrm{f}0$ the first level 1 has r_1 entries, then the first level needs $\left(\frac{r_1}{fo}\right)$ blocks.
- Then, in the second level r_2 index is needed.

Step 3:

- In next level, the primary index has an entry in the second level for the second-level blocks.

So, the entries in the third level is $r_3 = \left(\frac{r_2}{fo}\right)$

- Now repeat the process until all the entries fit in the single block of some index level t fit.

Use the formula to calculate the t **value,** $1 \le \left(\frac{r_1}{(fo)^t}\right)$

Hence in the multilevel index,

Approximately t levels will be corresponding to the r_1 first-level entries

Where $t = \left[\log_{fo}(r_1)\right]$

From the above steps and processer, we may improve the efficiency of the search an index file.

The following ways that the multilevel indexing improved the efficiency of searching an index file is:

• While searching the record, it reduces the access of number of blocks in the given indexing field value.

• The benefits of multi-level indexing include the reduction of insertion and deletion problems in indexing.

• While inserting new entries, it leaves some space that deals to tshe advantage to developers to adopt the multi-level indexing.

• By using B-tress and B+ trees, it is often implemented.

17.5. What is the order *p* of a B-tree? Describe the structure of B-tree nodes.

Order P of a B – tree:-

1

A tree, it consists that, each node contains at must p – 1 search values and *P* pointers in the order $P < P_1, K_1, P_2, K_2 .... P_{q-1}, K_{q-1}, P_q >$

Where $q <= P$:

Here each P_i is a pointer to child node and

K_i Is search value from some ordered set of values.

Structure of the B-tree

Structure of a B-tree follows the below steps.

Step 1:

Each internal node in the B-tree is in the form of

$< P_1, < K_1 P_{r1} >, P_2, < K_2, P_2 >, \ldots\ldots\ldots, < K_{q-1}, P_{rq-1} >, P_q >$

Here $q \leq P$

P_i is a tree pointer

P_{ri} is a data pointer. and

Search key value is equal to K;

Step 2:

With in each node, $K_1 < K_2 < \ldots\ldots\ldots K_{q-1}$

Step 3:

For all search key field values X in the Sub tree pointed at by P:

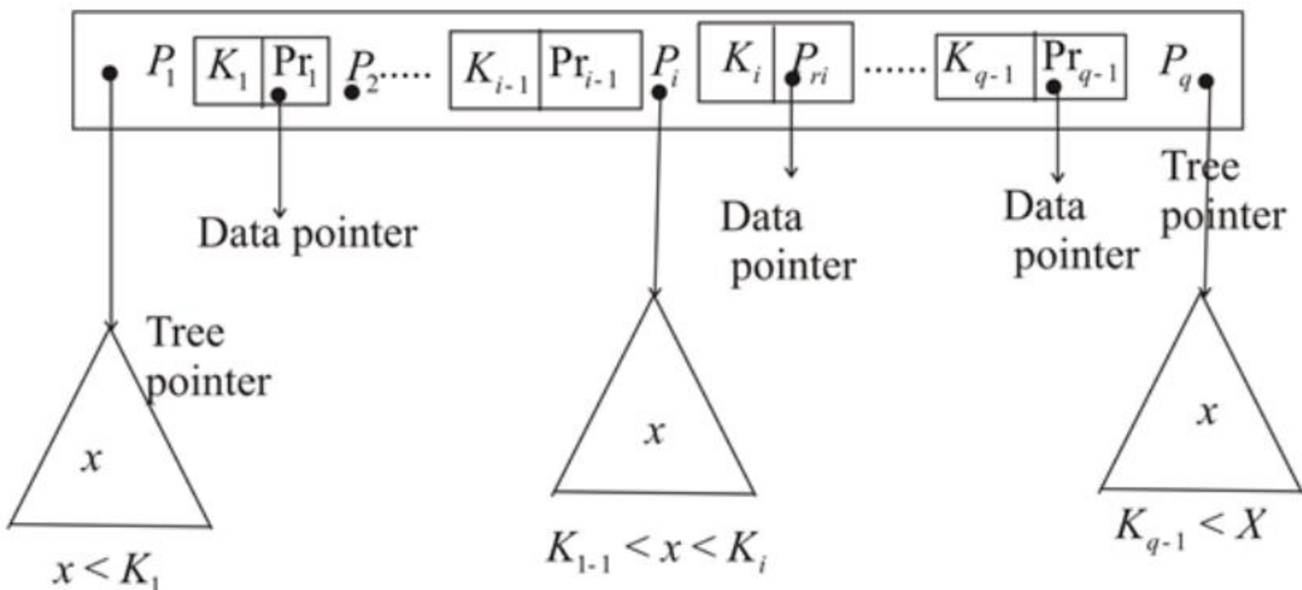
Step 4: Each node have at most p tree pointers.

Sep 5: Each node, except the root and leaf nodes, has at least two tree pointers unless it is the only node in the tree

Step 6: A node with a tree pointers, $q \leq p$, has $q-1$ search key field values.

Step 7: All nodes are at the same level. Leaf nodes have to same structure as internal nodes except that all of their tree pointers P_i are Null

Below figure shows the structure:-

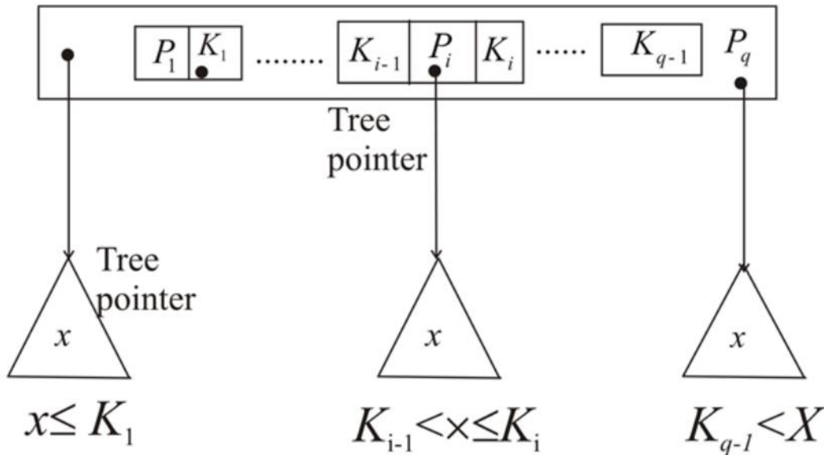


17.6. What is the order p of a B^+-tree? Describe the structure of both internal and leaf nodes of a B^+-tree.

Order P of a B^+-tree:-

Implementation of a dynamic multilevel index use a variation of the B-tree data structure is called as B^+-tree.

Structure of internal nodes of a B^+-tree:-



From the above figure,

Step 1

Each internal node is of the form of $< P_1, K_1, P_2, K_2, ..., P_{q-1}, K_{q-1}, P_q >$

Where $q \le P$ and each P_i is a tree pointer.

Step 2

Within each internal node, $K_1, < K_2 < .... < K_{q-1}$

Step 3

For all search field values X in the sub tree pointed at by P_i, we have $K_{i-1} < X \le K_i$ for $1 < i < q$; where $X \le K_i$ for $i = 1$ and $K_{i-1} < X$ for $i = q$

Step 4 each internal node has at most *P* tree pointers

Step 5

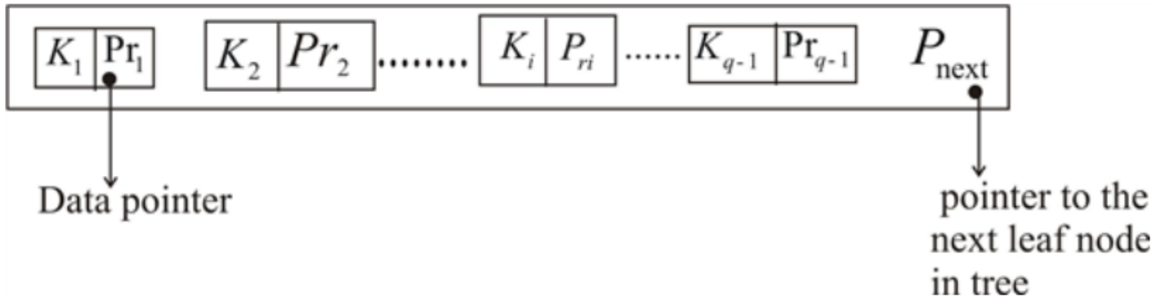
Each internal node, except the root, has at least $\left\lceil \frac{P}{2} \right\rceil$ tree pointers.

The root node has at least two tree pointers, if it is an internal node.

Step 6

An internal node with q pointers, $q \le P$. Has $q-1$ search field values.

Structure of leaf nodes of B^+-tree:-



From the above figure:-

Step 1:

Each leaf node is the form of

$<<K_1, Pr_1>, <K_2, Pr_2> .... <K_{q-1}, Pr_{q-1}>, P_{next}>$

Where $q \le P$, each Pr_i is a data pointer and P_{next} points to the next leaf node of the $B^+ - \text{tree}$.

Step 2:

Within each leaf node, $K_1 \le K_2 .... K_{q-1}, q \le P$

Step 3:

Each Pr_i is a data pointer that points to the record whose search field value is

K_i or to a file block containing the record.

Step 4:

Each leaf node has at least $\left[\left(\frac{P}{2}\right)\right]$ values.

Step 5:-

All leaf nodes are at the same values.

17.7. How does a B-tree differ from a B^+-tree? Why is a B^+-tree usually preferred as an access structure to a data file?

The main difference in B-tree and B^+ - tree is

→ A B-tree has data pointers in the both internal and leaf nodes, where as

→ In B^+-tree, it has only tree pointers in internal nodes and all data pointers are in leaf nodes.

B^+-tree preferred as an access structure to a data file because, entries in the internal nodes of a B^+-tree leading to fewer levels improving the search time.

In addition that, the entire tree can be traversed in order using the pent pointers.

17.8. Explain what alternative choices exist for accessing a file based on multiple search keys.

Choices for accessing file based on multiple fields are:

1. **Ordered Index on Multiple Attributes:** In this index is created on search key field that is a combination of attributes . If an index is created on attributes , the search key values are tuples with n values:

A lexicographic ordering of these tupl values establishes an order on this composite search key. Lexicographic ordering works similar to ordering of character strings. An index on a composite key of n attributes works similarly to primary or secondary indexing.

2. **Partitioned Hashing:** Partitioned hashing is an extension of static external hashing that allows access on multiple keys. It is only suitable for equality comparisons; range queries are not supported. In partitioned hashing, for a key consisting of n components, the hash function is designed to produce a result with n separate hash address. Th bucket address is a concatenation of these n addresses. It is then possible to search for the required composite search keys by looking up the appropriate buckets that match the parts of the address in which we are interested.

For example, consider the composite aearch key id Dno is hashed to 3 bits and Age to 5 bits; we get 8 bits of address. Suppose that Dno = has hash address '100' and for Age = 59 has address '10101' to search combination, search bucket address = 10010101.

An advantage of portioned hashing is is that it can be easily extended to any number of attributes. The bucket address can be designed so that high order bits in the address correspond to more frequently accessed attributes. Additionally, no separate access needs to be maintained for the individual attributes. The main drawback of portioned hashing is tat it cannot handle range queries on any of the component attributes.

3. **Grid Files:** We can construct grid array with one linear scale for each of search attributes. This method is particularly useful for range queries that would map into a set of cells corresponding to a group of values along the linear scales. Conceptually, the grid file concept may be applied to any number of search keys. For n search keys, the grid array would have n dimensions of the search keys attributes and provides an access by combinations of value along those dimensions. Grid files perform well in terms of reduction in time for multiple key accesses. However, they represent a space overhead in terms of grid array structure. Moreover, with dynamic files, a frequent recognition of the files adds to maintenance cost.

17.9. What is partitioned hashing? How does it work? What are its limitations?

Partitioned flashing:-

It is an extension of static external hashing. That allows access on multiple keys. Means, hash values that are split into segments. That depend on each attribute of the search key.

Let take one example:

Let $n = 2$, for customer and search-key being (customer – street, customer – city)

Search – key value hash value

(Main,) 101111

(Main,) 101001

(Park,) 010010

(Spring,) 001001

(,) 110010

Working functionally of partitioned hashing:-

→In partitioned hashing, for a key consisting of n components. Hash function is designed to produce a result with n separate, hash addresses.

→ Bucket address is added to these n address

→Now, it is ready to search for the required composite search key by looking up eh appropriate buckets that mach the parts of the address in which we are interested.

Limitations of partitioned hashing:-

→It can be easily extended to any number of attributes.

→For individual attributes, it has no separate access structure.

→It cannot handle range queries on any of the component attributes.

17.10. What is a grid file? What are its advantages and disadvantages?

Grid files:-

If we access a file on two key, then we can construct a gird array with one linear scale/dimension for each of the search attribute. A grid file has

→ a single grid array

→array has number of dimensions equal to the number of searches attributes.

→one linear scale for each search key attribute.

Grid array has multiple cells, that can point to the same bucket.

Advantage:-

→ Gird file concept may be applied to any number of search keys.

Eg: for ‘ n ’ search keys, the grid array would have ‘ n ’ dimensions,

→ For range of queries, it is very use full

→It performs well in terms of reduction in time for multiple key accesses.

Disadvantage:-

→ Frequent reorganization of the file maintenance of grid file is costly in case of dynamic files.

17.11. Show an example of constructing a grid array on two attributes on some file.

Take grid array for the EMPLOYEE file with one linear scale D no and another for the age attribute.

D no

0 1,2
1 3,4
2 5
3 6,7
4 8
5. 9,10

Linear scale for Age

0	1	2	3	4	5
<20	21-25	26-30	31-40	41-50	>50

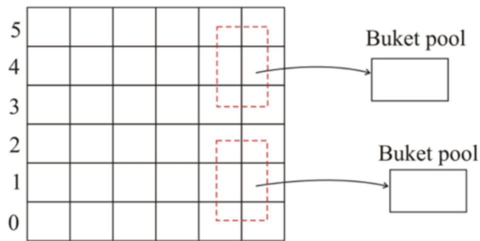
Through this data we want to show that the linear scale of D no has D no $=1,2$ combined as one value 0 on the scale while D no $=5$ corresponds to the value 2 on the scale and age is divided into its scale of 0 to 5 by grouping ages and distribute the employees uniformly by age.

For this the grid array shows 36 cells. And each cell points to some bucket address where the records corresponding to that cell are stored.

Now our request for D no $=4$ and age $=59$ maps into cell $(1,5)$. It is corresponding to grid array, and it will be found in the corresponding bucket. For 'n' search keys, the grid array would have 'n' dimensions.

Grid array on D no and AGE attributes.

Employee File.



17.12. What is a fully inverted file? What is an indexed sequential file?

Fully inverted file:-

Indexes that are all secondary and new records are inserted at the end of the file. Then the data file it self is an unordered file. So, a file that have secondary index on every one of its field is called as fully invented file. Usually, the indexes that are implemented as B^+- tree and up load dynamically to reflect insertion or deletion of records.

Indexed sequential file:-

→An indexed sequential file is a sequential file which has an index.

Sequential file means it stored into order of a key field.

→ Indexed sequential files are important for applications where data needs to be accessed through

Sequential and randomly using the index.

→An indexed sequential file allows fast access to a specific record.

Let an example.

An organization may store the details about it's employees as an indexed sequential file, and sometimes the file is accessed

Sequential:-

For example, when the whole of the file is processed to produce pay slips at the end of the month.

Randomly:

An example changes address, or a female employee gets married can changes her surname so, indexed sequential file can only be stored an a random access device.

Example magnetic disc, CD

17.13. How can hashing be used to construct an index?

Hashing technique is used for searching wherein fast retrieval of records is necessary. The reference file used for this is known as hash file. The search condition is validated using the hash key which is nothing but the reference name that has to be found.

Functions of hashing:

• A hash function 'f' or randomizing function is entered in the hash field value of a record and determines the address of it.

• It is also used as an internal search function within a program, whenever a group of records is accessed by using the value of only one field.

• Access structures similar to indexes that are based on hashing can be created; the **hash index** is a secondary structure to access the file by using hashing function on a search key.

• The index entries contains the key (K) and the pointer (P) used to point to the record containing the key or block containing the record for that key.

• The index files that contain these index entries can be organized as a dynamically expandable hash file, using dynamic or linear or extendible hashing techniques, searching for an entry is performed by using hash search algorithm on K.

• Once an entry is identified the pointer (P) is used to locate the corresponding record in the data file.

17.14. What is bitmap indexing? Create a relation with two columns and sixteen tuples and show an example of a bitmap index on one or both.

The bitmap index is a data structure that allows querying on more number of keys

• It is used for relations that contain a large number of rows so that it can be used identify the relation for the specific key value.

• It creates an index for one or more columns and each value or value range in those columns selected is/are indexed.

• A bitmap index is created for those columns that should contain only a small number of unique values.

Construction

• To create a bitmap index for a set of records in a relation or a table, the records must be numbered from 0 to n with an id that is used to be mapped to a physical address that contains a block number and a record offset within the block.

• It is created for one particular value of a particular field (or column) as an array of bits.

• For example a bitmap index is constructed for the column F and a value V of that column. A relation with n rows of n tuples and it contains n bits. The jth bit is set to 1 if the row j has the value V for column F, otherwise it is set to 0.

Example

S.No	Customer Name	Gender
1	Alan	M
2	Clara	F
3	John	M
4	Benjamin	M
5	Marcus	M
6	Alice	F
7	Joule	F
8	Louis	M
9	Samuel	M
10	Lara	F
11	Andy	F
12	Martin	M
13	Catherine	F
14	Fuji	F
15	Zarain	F
16	Ford	M

Bitmap index for Gender

For M: 1011100110010001, the row that contains the tuple M wherever it appears are set to 1, other are set to 0.

For F: 0100011001101110, the row that contains the tuple F wherever it appears are set to 1, other are set to 0.

17.15. What is the concept of function-based indexing? What additional purpose does it serve?

Function-based indexing is a new type of indexing that has been developed and used by the Oracle database systems as well as in some other organizational products that provides financial profit.

By applying any function to the value that belongs to the field or to the collection of fields, a result is obtained which is used as the key to the index that is used to create an index.

It ensures that Oracle Database System will use this index to search instead of performing the scan over full table, even when a function is used in the search value of a query.

Example,

The query that create an index, using function LOWER (CustomerName),

CREATE INDEX lower_ix ON Customer (LOWER (CustomerName));

It returns the customer name in lower case letter; LOWER ("MARTIN") results in “martin”, the query given below uses the index:

SELECT CustomerName

FROM Customer

WHERE Lower (CustomerName) = "martin".

If the functional-based indexing is not used, an Oracle database system perform scanning process for the entire table, as B^+-tree index is a searching process by using directly only the column value, any function that is used on a column avoids using such an index.

17.16. What is the difference between a logical index and a physical index?

Physical index

• The index entries with the key (K) and the physical pointer (P), used to point to the physical address of the record stored on the disk as a block number and offset. This is referred as physical index.

• For example, a primary file organization is based on extendible or linear hashing, and then at each time when a bucket is split, some of the records are allocated to a newer bucket and hence they are provided with new physical addresses.

• If there is a secondary indexing used on the file, the pointers that point to that record must be determined and updated (pointer must be changed if the record moved to another location) but it is considered to be a difficult task.

Logical index

• The index entries of logical index are a pair of keys K and K_S.

• Every entry of the records contains one value of K used for primary organization of files and another key K_S for the secondary indexing field matched with the value K of the field.

• While searching the secondary index on the value of K_S, a program can identify the location of the corresponding value of K and use this matching key terms to access the record through the primary organization of the file, thus it introduces an extra search level of indirection between the data and access structure.

17.17. What is column-based storage of a relational database?

Column-based storage of relations is a traditional way of storing the relations by row (one by one). It provides advantages especially for read-only queries, which are from read-only databases. It stores each column of data in relational databases individually and provides performance advantages.

Advantages

• Partitioning the table vertically column by column, so those tables with a two-column are constructed for each and every attribute of the table and thus only the columns that are needed can be accessed**.**

• Column-wise indexes and join indexes are used on multiple tables to provide answer to the queries without accessing the data tables.

• Materialized views are used to support queries on multiple columns.

Column-wise storage of data provides an extra feature in the index creation. The same column present in each table on number of projections creates indexes on each projection. For storing the values in the same column, various strategies, data compression, null value suppression, and various encoding techniques are used.

Exercises

17.18. Consider a disk with block size $B = 512$ bytes. A block pointer is $P = 6$ bytes long, and a record pointer is $P_R = 7$ bytes long. A file has $r = 30{,}000$ EMPLOYEE records of *fixed length*. Each record has the following fields: Name (30 bytes),Ssn (9 bytes), Department_code (9 bytes), Address (40 bytes), Phone (10 bytes), Birth_date (8 bytes), Sex (1 byte), Job_code (4 bytes), and Salary (4 bytes, real number). An additional byte is used as a deletion marker.

a. Calculate the record size R in bytes.

Disk operations on file using primary, secondary, clustering, B+ tree and B-tree methods

(a) **Calculation of Record Size**

Record size is calculated as follows

Record size=Name (in bytes)+ Ssn(in bytes)+Department_code(in bytes)

+Address (in bytes) + Phone (in bytes)+ Birth_date(in bytes)

+Sex (in bytes)+ Job_code(in bytes)+Salary(in bytes)

+1(1byte for deletion marker)

Record size $R = 30+9+9+40+9+8+1+4+4+1 = 115\,\text{bytes}$

b. Calculate the blocking factor *bfr* and the number of file blocks *b*, assuming an unspanned organization.

(b) **Calculation of Blocking factor and number of file blocks**

Blocking factor, bfr $= \text{floor}\left(\frac{B}{R}\right)$

$= \text{floor}\left(\frac{512}{115}\right)$

$= 4$ Records per block

Number of file blocks, $b = \text{Ceiling}\left(\frac{r}{bfr_i}\right)$

$= \text{Ceiling}\left(\frac{30000}{4}\right)$

$= 7500$

c. Suppose that the file is *ordered* by the key field Ssn and we want to construct a *primary index* on Ssn. Calculate (i) the index blocking factor bfr_i (which is also the index fan-out *fo*); (ii) the number of first-level index entries and the number of first-level index blocks; (iii) the number of levels needed if we make it into a multilevel index; (iv) the total number of blocks required by the multilevel index; and (v) the number of block accesses needed to search for and retrieve a record from the file—given its Ssn value—using the primary index.

(c) **Operations on file ordered by key field Ssn**

(i) **Calculation of Index blocking factor and**

Index record length, $R_i = (\text{VssN} + \text{P})$

$= 9 + 6$

$= 15\,\text{bytes}$

Blocking factor, bfr_i

$= \text{floor}\left(\frac{\text{B}}{\text{R}_i}\right)$

$= \text{floor}\left(\frac{512}{15}\right)$

$= 34$

(ii) **Calculation of number of first –level index and number of first level index blocks**

Number of first – level index entries r_1

=Number of first level index blocks b_1

Number of first-level index entries, r_1

$=$ number of file blocks b

$=$ 7500 entries

Number of first-level index blocks, b_1

$= \text{ceiling}(\frac{\text{r1}}{\text{bfr}_i})$

$= \text{ceiling}(\frac{7500}{34})$

=221 blocks

(iii) **Calculation of number of levels for multi-level index**

Number of second-level index entries r_2 =

Number of first-level blocks, b_1 = 221 entries

Number of second-level index blocks, b_2

$$= \text{ceiling}\left(\frac{r_2}{bfr_i}\right)$$
$$= \text{ceiling}\left(\frac{221}{34}\right)$$
$$= 7 \text{ blocks}$$

Number of third-level index entries, r_3 = number of second-level index blocks, b_2

= 7 entries

Number of third-level index blocks, b_3

$$=\text{ceiling}\left(\frac{r_3}{bfr_i}\right)$$
$$=\text{ceiling}\left(\frac{7}{34}\right)$$
$$=\text{ceiling}(0.20)$$
$$=1$$

It is the top index level because the third level has only one index.

Hence, the index has x = 3 levels

(iv) **Calculation of number of blocks for multi-level index**

Total number of the blocks for the index $b_i = b_1 + b_2 + b_3$

From bit (ii), Number of first-level index blocks, b_1 =221 blocks

From bit (iii), Number of second-level index blocks, b_2 =7 block

Number of second-level index blocks , b_3 =1 blocks

Therefore, the total number of blocks, $= 221 + 7 + 1$

$$= 229 \text{ blocks}$$

(v) **Calculation of number of block access to search and retrieve a record using primary index on a file.**

For primary index type of index, the number of block access is equal to

the access one block at each level plus one block from the data file.

Therefore, the number of block access =x+1

Since the file is ordered with a single key field, Ssn. So it is a type of primary index.

Number of blocks access to search for a record

$= x + 1$

$= 3 + 1$

$= 4$

d. Suppose that the file is *not ordered* by the key field Ssn and we want to construct a *secondary index* on Ssn. Repeat the previous exercise (part c) for the secondary index and compare with the primary index.

(d) **Repetition of part c for the secondary index**

(i) Index record length $R_i = (V\text{ssN} + P)$

$= 9 + 6$

$= 15\,\text{bytes}$

Index blocking factor bfr $i = (\text{fan} - \text{out})\,\text{fo}$

$= \text{floor}\left(\frac{\text{B}}{\text{R}_\text{i}}\right)$

$= \text{floor}\left(\frac{512}{15}\right)$

$= 34\,\text{index records/ block}$

In the ‘c’ part, the assumes that leaf-level index blocks contain block pointers. And it is possible to assume that they contain record pointers. And Record size is $V\text{ssN} + PR$

$= 9 + 7$

$= 16\,\text{bytes}$

So, leaf – level index blocking factor bfri.

$$\text{bf i} = \text{floor}\left(\frac{B}{Ri}\right)$$

$$= \text{floor}\left(\frac{512}{16}\right)$$

$= 32$ Index records/block for internal nodes, block pointers are always used, so the fan-out for internal nodes to is 34.

(ii) Number of first-level index entries

$r_1 =$ Number of file records $r = 30000$

Number of first level index blocks $b_1 = \text{Ceiling}\left(\frac{r_1}{\text{bfri}}\right)$

$$= \text{Ceiling}\left(\frac{30{,}000}{34}\right)$$

$= 883 \text{ blocks}$

Number of first level index entries r_1

$r_1 =$ Number of file records $r = 30000$

Number of first-level index block b_1

$$b_1 = \text{Ceiling}\left(\frac{r_1}{\text{bfr i}}\right)$$

$$= \text{Ceiling}\left(\frac{30000}{32}\right)$$

$= 938 \text{ blocks}$

(iii) Calculate the number of levels

Number of second –level index entries r_2

r_2 = Number of first-level index blocks $b_1 = 883 \text{ entries}$

Number of second – level index blocks b_2

$$b_2 = \text{Ceiling}\left(\frac{r_2}{\text{bfri}}\right)$$

$$= \text{Ceiling}\left(\frac{883}{34}\right)$$

$$= 26 \text{ blocks}$$

Number of third-level index block b_3

$$b_3 = \text{Ceiling}\left(\frac{r_3}{\text{bfri}}\right)$$

$$= \text{Ceiling}\left(\frac{26}{34}\right)$$

$$= 1$$

So, the third level has one block and it is the top of the level

So, index has total 3 levels $\Rightarrow x = 3 \text{ levels}$

(iv) Total number of blocks for the index (b_i)

$$\begin{aligned} bi &= b_1 + b_2 + b_3 \\ &= 938 + 28 + 1 \\ &= 967 \end{aligned}$$

(v) Number of blocks accesses to search for a record $= x + 1$

$$= 3 + 1$$

$$= 4$$

e. Suppose that the file is *not ordered* by the nonkey field Department_code and we want to construct a *secondary index* on Department_code, using

option 3 of Section 17.1.3, with an extra level of indirection that stores record pointers. Assume there are 1,000 distinct values of Department_code and that the EMPLOYEE records are evenly distributed among these values. Calculate (i) the index blocking factor bfr_i (which is also the index fan-out *fo*); (ii) the number of blocks needed by the level of indirection that stores record pointers; (iii) the number of first-level index entries and the number of first-level index blocks; (iv) the number of levels needed if we make it into a multilevel index; (v) the total number of blocks required by the multilevel index and the blocks used in the extra level of indirection; and (vi) the approximate number of block accesses needed to search for and retrieve all records in the file that have a specific Department_code value, using the index.

(e) **Operations on the file which is constructed using secondary index on Department_code**

(i) **Calculation of index blocking factor**

Index record size $R_i = (V \text{ DEPARTMENT CODE } + \text{P})$

$= 9 + 6$

$= 15 \text{ bytes}$

Index blocking factor (bfri)

$\text{bfri} = (\text{fan} - \text{out}) \text{ to floor } \left(\frac{\text{B}}{\text{Ri}}\right)$

$= \text{floor} \left(\frac{512}{15}\right)$

$= 34 \text{ index records/block}$

(ii) **Calculation of number of blocks for indirection**

Here 1000 distinct values of Department_code.

Number of records for each value is $\frac{r}{1000}$

$$=\frac{30000}{1000}$$

$$=30$$

So, we know that record pointer size $P_R = 7\text{bytes},$

Number of bytes need at the level of indirection for each value of

Department_code is $7 \times 30 = 210\,\text{bytes}$

It is fits on the block

So, 1000 blocks are needed for the level of indirection.

(iii) **Calculation of number of first-level index entries and number of first level blocks**

Number of first-level index entries (r_1)

$r_1 =$ Number of distinct values of Department_code

$$=1000\,\text{entries}$$

Number of first level index blocks (b_1)

$$b_1 = \text{Ceiling}\left(\frac{r_1}{\text{bfri}}\right)$$

$$=\text{Ceiling}\left(\frac{1000}{34}\right)$$

$$=30\,\text{blocks}$$

(iv) **Calculation of number of levels for multi-level index**

We can calculate the number of levels by number of second level index entries (r_2)

$r_2 =$ Number of first level index blocks b_1

$\Rightarrow b_1 = 30$ Entries

Number of second-level index blocks (b_2)

$b_2 =$ Ceiling $\left(\frac{r_2}{\text{bfr i}}\right)$

$= \text{Ceiling}\left(\frac{30}{34}\right)$

$= 1$

The index has $x = 1$ level

(v) **Calculation of number of blocks for multi-level index**

Total number of blocks for the index (b_i)

$b_i = b_1 + b_2 + \text{bindrection}$

$= 30 + 1 + 1000$

$= 1031\,\text{bytes}$

(vi) **Calculation of number of block access to search and retrieve all records in the file for a Department_code value**

Number of block accesses to search for and retrieve the block containing the record pointers at the level of indirection

$= x + 1$

$= 2 + 1$

$= 3\,\text{block accesses}$

If 30 records are distributed over 30distrinct blocks, we need an additional 30 blocks.

So, total block accesses needed on average to retrieve all the records with in a given value for Department_code $= x + 1 + 30$

$= 2 + 1 + 30$

$= 33$

f. Suppose that the file is *ordered* by the nonkey field Department_code and we want to construct a *clustering index* on Department_code that uses block anchors (every new value of Department_code starts at the beginning of a new block). Assume there are 1,000 distinct values of Department_code and that the EMPLOYEE records are evenly distributed among these values. Calculate (i) the index blocking factor bfr_i (which is also the index fan-out fo); (ii) the number of first-level index entries and the number of first-level index blocks; (iii) the number of levels needed if we make it into a multilevel index; (iv) the total number of blocks required by the multilevel index; and (v) the number of block accesses needed to search for and retrieve all records in the file that have a specific Department_code value, using the clustering index (assume that multiple blocks in a cluster are contiguous).

(f) **Operations on the file which is constructed using clustering index on Department_code**

(i) Calculation of index blocking factor

Index blocking factor (bfr i)

$$\text{bfr i} = (\text{fan-out})\,\text{f}_0$$

$$= \text{floor}\left(\frac{\text{B}}{\text{Ri}}\right)$$

$$= \text{floor}\left(\frac{512}{15}\right)$$

$$= \text{floor}(34.13)$$

$$= 34 \text{ index records/block}$$

Where $Ri = (\text{V DEPARTMENT CODE } + P)$

$$= 9 + 6$$

$$= 15$$

(ii) **Calculation of number of first-level index entries and number of first level blocks**

Number of first level index entries (r_1)

$r_1 =$ Number of distinct DEPARTMENT CODE values

$= 1000$ entries.

Number of first-level index blocks (b_1)

$$\Rightarrow b_1 = \text{Ceiling}\left(\frac{r_1}{\text{bfr i}}\right)$$

$$= \text{Ceiling}\left(\frac{1000}{34}\right)$$

$$= 30 \text{ blocks}$$

(iii) **Calculation of number of levels for multi-level index**

Calculate the number of levels as number of second-level index entries (r_2)

$r_2 =$ Number of first-level index blocks (b_1)

$= 30\,\text{entries}$

Number of second-level index blocks b_2

$b_2 =$ Ceiling $\left(\frac{r_2}{\text{bfr i}}\right)$

$= \text{Ceiling}\left(\frac{30}{34}\right) = 1$

Second level has one block and it is in the top index level

The index has $x = 1\,\text{level}$

(iv) **Calculation of number of blocks for multi-level index**

Total number of blocks for the index (b_i)

$\Rightarrow b_i = b_1 + b_2$

$= 30 + 1$

$= 31\,\text{blocks}$

(v) **Calculation of number of block access to search and retrieve all records in the file for a Department_code value**

Number of block accesses to search for the first block in the cluster of blocks $= x + 1$

$= 2 + 1$

$= 3$

So, the 30 records are clustered in ceiling $\left(\frac{30}{\text{bfr}}\right)$

$= 30\ \text{Ceiling}\left(\frac{30}{4}\right)$

$= 8\,\text{blocks}$

So, total block accesses needed on average to retrieve all the records with a given

DEPARTMENT CODE $= x + 8$

$= 2 + 8$

$= 10\,\text{block accesses}$

g. Suppose that the file is *not* ordered by the key field Ssn and we want to construct a B^+-tree access structure (index) on Ssn. Calculate (i) the orders p and p_{leaf} of the B^+-tree; (ii) the number of leaf-level blocks needed if blocks are approximately 69% full (rounded up for convenience); (iii) the number of levels needed if internal nodes are also 69% full (rounded up for convenience); (iv) the total number of blocks required by the B^+-tree; and (v) the number of block accesses needed to search for and retrieve a record from the file—given its Ssn value—using the B^+-tree.

(g) **Operations on B+ tree**

(i) **Calculation of orders p and p-leaf of B+ tree**

Orders P and P leaf of the $B^+ - \text{tree}$

Each internal node has

$$(p \times P) + ((p-1) \times \text{VssN}) < B \text{ or}$$

$$((p \times 6) + (p-1) \times 9) < 512$$

$$15p < 512$$

$$\Rightarrow p = \frac{512}{15}$$

$$= 34$$

So, $p = 34$

For leaf nodes, the record pointers are included in the leaf nodes, and it satisfied the

$$(P\text{leaf} \times (\text{VssN} + \text{P}_\text{R})) + P < B$$

(Or)

$$(P\text{leaf} \times (9+7)) + 6 < 512$$

$$16P\text{leaf} < 512$$

$$\Rightarrow P\text{leaf} = 31$$

(ii) **Calculation of leaf nodes if the blocks are 69 percent full**

Nodes are 69% full on the average, so the average number of key values in a leaf node is

$\Rightarrow 0.69 \times \text{P leaf}$

$0.69 \times 31 = 21.39$

If we round up this for convenience, we get 22 key values and 22 record pointers per leaf node.

So, the file has 30000 records and hence 30000 values of ssN, the number of leaf-level nodes needed is

$$b_1 = \text{Ceiling}\left(\frac{30000}{22}\right)$$
$$= \text{Ceiling}(1363.63)$$
$$= 1364$$

(iii) **Calculation of number of levels if internal nodes are 69 percent full**

Calculate the number of levels as average fan-out for the internal nodes is (f_0)

$$f_0 = \text{Ceiling}\ (0.69 \times p)$$

$$= \text{Ceiling}\ (0.69 \times 34)$$

$$= \text{Ceiling}\ (23.46)$$

$$= 24$$

Number of second level tree blocks (b_2)

$$b_2 = \text{Ceiling}\left(\frac{b_1}{f_0}\right)$$

$$= \text{Ceiling}\left(\frac{1364}{24}\right)$$

$$= 57\ \text{blocks}$$

Number of third level tree blocks (b_3)

$$b_3 = \text{Ceiling}\left(\frac{b_2}{f_0}\right)$$

$$= \text{Ceiling}\left(\frac{57}{24}\right)$$

$$= 3\ \text{blocks}$$

Number of fourth-level tree blocks (b_4)

$$b_4 = \text{Ceiling}\left(\frac{b_3}{f_0}\right)$$

$$= \text{Ceiling}\left(\frac{3}{24}\right)$$

$$= 1$$

So, the fourth level has one block and the tree has $x = 4$ levels

So,

$$x = \text{Ceiling}\left(\log f_0 (b_1)\right) + 1$$

$$= \text{Ceiling}\left(\log 24\ 1364\right) + 1$$

$$= 3 + 1 = 4 \text{ levels}$$

(iv) **Calculation of total number of blocks**

Total number of blocks for the tree b_i

$$b_i = b_1 + b_2 + b_3 + b_4$$

$$= 1364 + 57 + 3 + 1$$

$$= 1425 \text{ blocks}$$

h. Repeat part g, but for a B-tree rather than for a B^+-tree. Compare your results for the B-tree and for the B^+-tree.

(h) **Repetition of part (g) for B-tree**

(i) *p* and *p leaf* order of the B -tree

Each internal node has

$$(p \times P) + ((p-1) \times (P_R + VssN)) < B \text{ or}$$

$$= ((p \times 6) + (p-1) \times (7+9)) \le 512$$

$$= ((p \times 6) + (p-1) \times 16) \le 512$$

$$= 22p - 16 \le 512$$

$$= 22p \le 512 + 16$$

$$= 22p \le 528$$

Choose p value as large value that satisfies the inequality

So, $p = 23$

For leaf nodes, the record pointers are included in the leaf nodes, and it satisfied the

$$= (p\,\text{leaf} \times (\text{VssN} + P_R)) + P \le B$$

$$= (pleaf \times (9+7) + 6) \le 512$$

$$= 16\,pleaf \le 512$$

$$pleaf = 31$$

(ii) Each node of B-Tree is 69% full .So the average number of key values in a leaf node is

$$= 0.69 \times p\ \text{leaf}$$

$$= 0.69 \times 31$$

$$= 21.39$$

If we get ceiling of 21.39 for convenience, we get 22 key values and 22 record pointers per leaf node.

So, the file has 30000 records and hence 30000 values of ssN , the number of leaf-level nodes needed is

$$b_1 = \text{Ceiling}\left(\frac{30000}{22}\right)$$

$$= \text{Ceiling}(1363.63)$$

$$= 1364$$

(iii) Calculate the number of levels as average fan-out for the internal nodes

is (f_0)

$$f_0 = \text{Ceiling } (0.69 \times p)$$

$$= \text{Ceiling } (0.69 \times 23)$$

$$= \text{Ceiling } (15.87)$$

$$= 16$$

Number of second level tree blocks (b_2)

$$b_2 = \text{Ceiling } \left(\frac{b_1}{f_0}\right)$$

$$= \text{Ceiling } \left(\frac{1364}{16}\right)$$

$$= \text{Ceiling}(85.25)$$

$$= 86 \text{ blocks}$$

Number of third level tree blocks (b_3)

$$b_3 = \text{Ceiling } \left(\frac{b_2}{f_0}\right)$$

$$= \text{Ceiling } \left(\frac{86}{16}\right)$$

$$= \text{Ceiling}(5.375)$$

$$= 6$$

Number of fourth-level tree blocks (b_4)

$$b_4 = \text{Ceiling } \left(\frac{b_3}{f_0}\right)$$

$$= \text{Ceiling } \left(\frac{6}{16}\right)$$

$$= \text{Ceiling}(0.375)$$

$$= 1$$

So, the fourth level has one block and the tree has $x = 4$ levels

So,

$$x = \text{Ceiling } (\log f_0 (b_1)) + 1$$

$$= \text{Ceiling } (\log 16\ 1364) + 1$$

$$= 3 + 1 = 4 \text{ levels}$$

(iv) Total number of blocks for the tree b_i

$$b_i = b_1 + b_2 + b_3 + b_4$$
$$= 1364 + 16 + 6 + 1$$
$$= 1387$$

(v) Number of blocks accesses to search for a record $= x + 1$

$$= 4 + 1$$
$$= 5$$

Comparison of B+ tree and B-tree

Calculation of approximate number of entries in B+ tree

At root level, each node on average will have 34 pointers and 33 (p-1) search fields

Root	1 node	33 entries	34 pointers
Level1	34 nodes	1122 entries	1156 pointers
Level2	1156 nodes	38148 entries	39304 pointers
Level3	39304 nodes	1297032 entries	

Calculation of approximate number of entries in B tree

At root level, each node on average will have 23 pointers and 22 (p-1) search fields

Root	1 node	23 entries	22 pointers
Level1	22 nodes	506 entries	484 pointers
Level2	484 nodes	11132 entries	10648 pointers
Leaf Level	10648 nodes	244904 entries	

For given block size, pointer size and search key field size, a three level B+ tree holds 1336335 entries on average .Similarly, for given block size, pointer size and search key field size, a leaf level B tree holds 256565 entries on average .Therefore, average entries stored on B+ tree are more than the average entries stored in B tree.

17.19. A PARTS file with Part# as the key field includes records with the following Part# values: 23, 65, 37, 60, 46, 92, 48, 71, 56, 59, 18, 21, 10, 74, 78, 15, 16, 20, 24, 28, 39, 43, 47, 50, 69, 75, 8, 49, 33, 38. Suppose that the search field values are inserted in the given order in a B^+-tree of order $p = 4$ and $p_{leaf} = 3$; show how the tree will expand and what the final tree will look like.

B+ Tree Insertion:

Here, the given a set of keys to be inserted into a B^+-tree of order $P=4$

- The Order $P=4$ implies that each node in the tree should have at most 4 pointers.
- $P_{leaf}=3$ Means the leaf nodes must have at least 2 keys and at most 3 keys.
- The insertion first start from the root, when root or any node overflows its capacity, it must split.
- When a leaf node is full the first $\lceil (p_{leaf}+1)/2 \rceil$ elements will keep in that node and rest elements should form the right node.
- The element at that rightmost position of the left partition will propagate up to the parent node.
- If the propagation is from the leaf node, a copy of the element should maintain at leaf. Else, just move that element to its parent node.
- All the elements in the key list should be there in the leaf nodes.

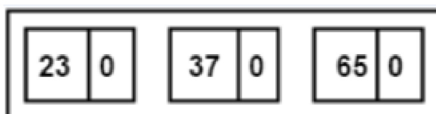
In problem given a set of keys to insert into the B+ tree in order.

The given list is,

23, 65, 37, 60, 46, 92, 48, 71, 56, 59, 18, 21, 10, 74, 78, 15, 16, 20,24, 28, 39, 43, 47, 50, 69, 75, 8, 49, 33, 38.

First, insert the first three keys into the root; it will not result in overflow. Since, the capacity of the node is also 3.

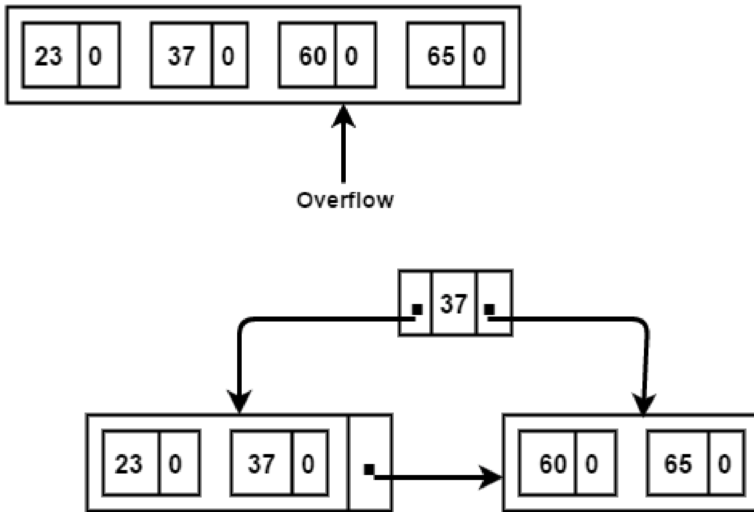
The resultant B+ tree will be as below:



Since, the node is also a leaf node and there is no pointer.

Insert 60:

After the insertion of 60 into this node, it will results in an overflow, So the node to be split into two and a new level will created as below:



Insert 46:

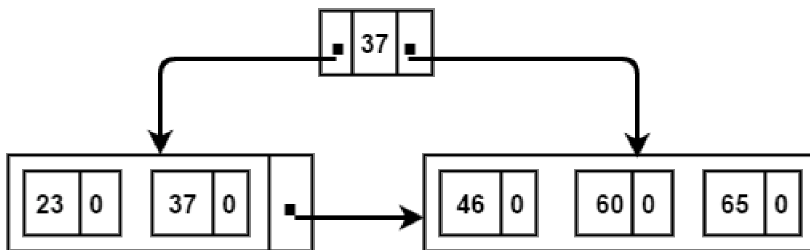
Insertion of 46 will not affect the capacity constraint of the second node in level 2.

The resultant tree will be,

Level1- 37

Level2: 23,37: 46,60,65

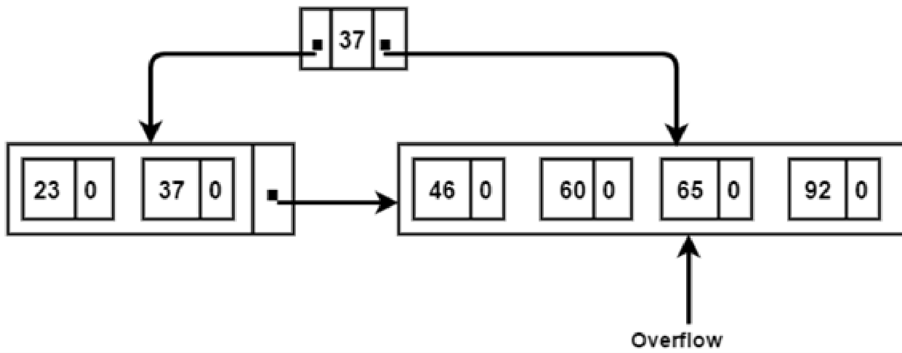
The tree will look as below:

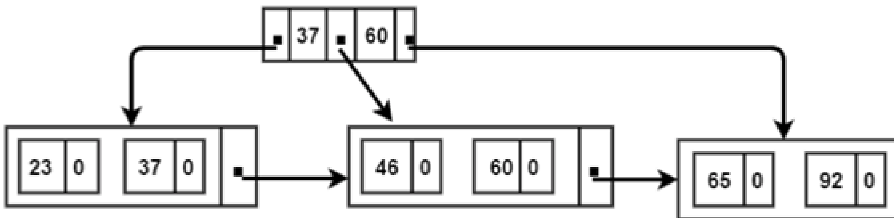


Insert 92:

Insertion of next key, 92 will results in the overflow of the second node in the level2, it will be 46,60,65,92.

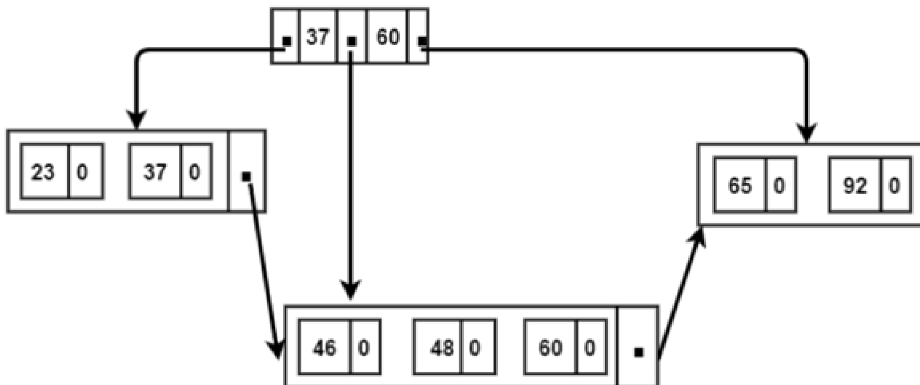
• Therefore, we need to split that node from 60 and create one new node in level2 and duplicate 60 in the parent node as below:





Insert 48:

Insertion of 48 will not prompt any overflow it will insert to the second node in the level2 as below:



Insert 71:

Insertion of **71** into B+ tree also will not prompt any overflow.

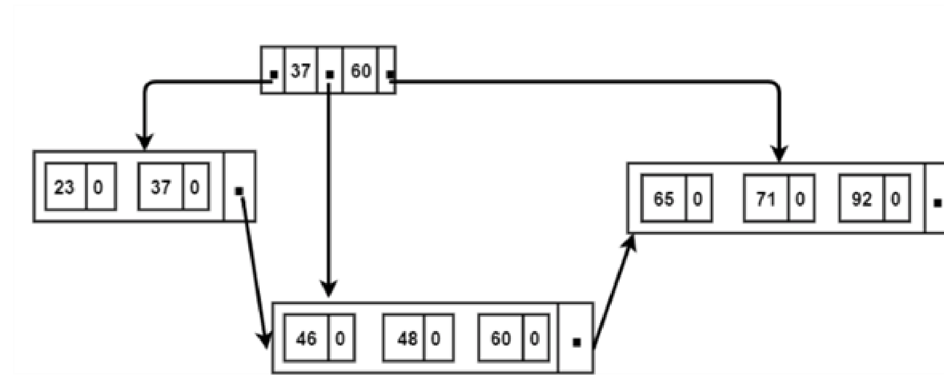
• It can insert into third node of level 2 without violating order constraints.

• Therefore, the updated tree will be as below:

Level1: 37, 60

Level2: 23, 37: 46, 48, 60: 65, 71, 92

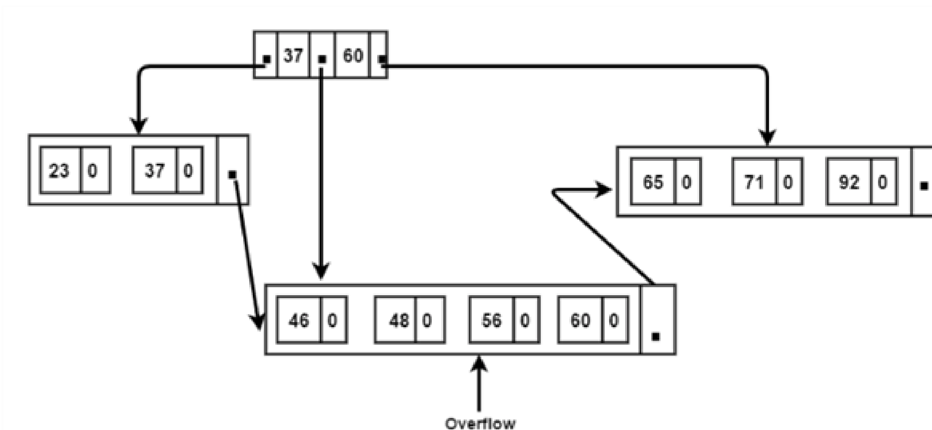
The tree will be look as below:



Insert 56:

Next insertion is of **56.**

• Clearly 56 is belongs to the second node of level 2 but it will results in an overflow as shown below:



• So, need to split that node (46, 48, 56, 60).

• The first two (46, 48) will form the first node of split and (56, 60) will form the second, the last element of the first set (48) will propagate to up.

• Since it is a leaf node, it will be only duplication.

However, the resultant B+ will be as below:

Level 1: 37, 48, 60

Level 2: 23, 37: 46, 48: 56, 60: 65, 71, 92

The rest insertion operations can be performed as below:

• The level is counts from root to leaves, that is; root will have level value 1 and increment 1 downwards.

Insert 59:

Level 1: 37, 48, 60

Level 2: 23, 37: 46, 48: 56**,** **59**, 60: 65, 71, 92

Insert 18:

Level 1: 37, 48, 60

Level 2: **18,** 23, 37: 46, 48: 56**,** 59, 60: 65, 71, 92

Insert 10:

Level 1: 37

Level 2: 21: 48, 60

Level 3: **10,** 18, 21**:** 23, 37: 46, 48: 56**,** 59, 60: 65, 71, 92

Insert 74:

Level 1: 37

Level 2: 21: 48, 60

Level 3: 10, 18, 21**:** 23, 37: 46, 48: 56**,** 59, 60: 65, 71, **74,** 92

Overflow in level 3. Split overloaded node at 71

Level 1: 37

Level 2: 21: 48, 60, **71**

Level 3: 10, 18, 21**:** 23, 37: 46, 48: 56**,** 59, 60: 65, 71: **74**, 92

Insert 78:

Level 1: 37

Level 2: 21: 48, 60, 71

Level 3: 10, 18, 21**:** 23, 37: 46, 48: 56**,** 59, 60: 65, 71: 74, **78**, 92

Insert 15:

Level 1: 37

Level 2: 21: 48, 60, 71

Level 3: 10, **15,** 18, 21**:** 23, 37: 46, 48: 56**,** 59, 60: 65, 71: 74, 78, 92

Overflow in the first node of level 3, split it at 15 and propagate 15 up.

Level 1: 37

Level 2: **15**, 21: 48, 60, 71

Level 3: 10, **15:** 18, 21**:** 23, 37: 46, 48:

56**,** 59, 60: 65, 71: 74, 78, 92

Insert 16:

Level 1: 37

Level 2: 15, 21: 48, 60, 71

Level 3: 10, 15 **16,** 18, 21**:** 23, 37: 46, 48:

56**,** 59, 60: 65, 71: 74, 78, 92

Insert 20:

Level 1: 37

Level 2: 15, 21: 48, 60, 71

Level 3: 10, 15 16, 18, **20**, 21**:** 23, 37: 46, 48:

56**,** 59, 60: 65, 71: 74, 78, 92

Overflow at the inserted node, split it at 18 and propagate 18 up.

Level 1: 37

Level 2: 15, **18**, 21: 48, 60, 71

Level 3: 10, 15 16, 18: **20**, 21**:** 23, 37: 46, 48:

56**,** 59, 60: 65, 71: 74, 78, 92

Insert 24:

Level 1: 37

Level 2: 15, 18, 21: 48, 60, 71

Level 3: 10, 15 16, 18: 20, 21**:** 23, **24,** 37: 46, 48:

56**,** 59, 60: 65, 71: 74, 78, 92

Insert 28:

Level 1: 37

Level 2: 15, 18, 21: 48, 60, 71

Level 3: 10, 15 16, 18: 20, 21**:** 23, 24, **28,** 37: 46, 48: 56**,** 59, 60: 65, 71: 74, 78, 92

Overflow in the fourth node of level 3, split it at 24 and propagate 24 up as below.

Level 1: 37

Level 2: 15, 18, 21,**24**: 48, 60, 71

Level 3: 10, 15 16, 18: 20, 21**:** 23, 24: **28,** 37: 46, 48: 56**,** 59, 60: 65, 71: 74, 78, 92

Again, overflow at level 2, need one more split at 18 as below.

Level 1: **18,** 37

Level 2: 15: 21,24: 48, 60, 71

Level 3: 10, 15 16, 18: 20, 21**:** 23, 24: **28,** 37: 46, 48: 56**,** 59, 60: 65, 71: 74, 78, 92

Insert 39:

Level 1: 18**,** 37

Level 2: 15: 21, 24: 48, 60, 71

Level 3: 10, 15 16, 18: 20, 21**:** 23, 24: 28, 37: **39,** 46, 48: 56**,** 59, 60: 65, 71: 74, 78, 92

Insert 43:

Level 1: 18**,** 37

Level 2: 15: 21, 24: 48, 60, 71

Level 3: 10, 15 16, 18: 20, 21**:** 23, 24: 28, 37: 39, **43**, 46, 48: 56**,** 59, 60: 65, 71: 74, 78, 92

Over flow at the inserted node, so split that node at second element 43 as below.

Level 1: 18**,** 37

Level 2: 15: 21, 24: **43,** 48, 60, 71

Level 3: 10, 15 16, 18: 20, 21**:** 23, 24: 28, 37: 39, **43:** , 46, 48: 56**,** 59, 60: 65, 71: 74, 78, 92

Again, overflow at level 2.

Level 1: 18**,** 37, **48**

Level 2: 15: 21, 24: **43:** 60, 71

Level 3: 10, 15 16, 18: 20, 21**:** 23, 24: 28, 37: 39, **43:** , 46, 48: 56**,** 59, 60: 65, 71: 74, 78, 92

Insert 47:

Level 1: 18**,** 37, 48

Level 2: 15: 21, 24: 43: 60, 71

Level 3: 10, 15 16, 18: 20, 21**:** 23, 24: 28, 37: 39, 43: 46, **47,** 48: 56**,** 59, 60: 65, 71: 74, 78, 92

Insert 50:

Level 1: 18**,** 37, 48

Level 2: 15: 21, 24: 43: 60, 71

Level 3: 10, 15 16, 18: 20, 21**:** 23, 24: 28, 37: 39, 43: 46, 47**,** 48: **50,** 56**,** 59, 60: 65, 71:

74, 78, 92

Overflow at the inserted node. Split the node at 56, the second element and propagate it up as below.

Level 1: 18**,** 37, 48

Level 2: 15: 21, 24: 43: **56,** 60, 71

Level 3: 10, 15 16, 18: 20, 21**:** 23, 24: 28, 37: 39, 43: 46, 47**,** 48: **50,** 56**:** 59, 60: 65, 71: 74, 78, 92

Insert 69:

Level 1: 18**,** 37, 48

Level 2: 15: 21, 24: 43: 56, 60, 71

Level 3: 10, 15 16, 18: 20, 21**:** 23, 24: 28, 37: 39, 43: 46, 47**,** 48: 50, 56**:** 59, 60: 65, **69,** 71: 74, 78, 92

Insert 75:

Level 1: 18**,** 37, 48

Level 2: 15: 21, 24: 43: 56, 60, 71

Level 3: 10, **15** 16, 18: 20, 21**:** 23, 24: 28, 37: 39, 43: 46, 47**,** 48: 50, 56**:** 59, 60: 65, 69, 71: 74, **75** 78, 92

Overflow at the inserted node, split and propagate up the node at the second element.

Level 1: 18**,** 37, 48

Level 2: 15: 21, 24: 43: 56, 60, 71, **75**

Level 3: 10, **15** 16, 18: 20, 21**:** 23, 24: 28, 37: 39, 43: 46, 47**,** 48: 50, 56**:** 59, 60: 65, 69, 71: 74, **75**: 78, 92

Again, overflow at the inserted node, split it at 60 and propagate up.

Level 1: 18**,** 37, 48, **60**

Level 2: 15: 21, 24: 43: 56: 71, **75**

Level 3: 10, **15** 16, 18: 20, 21**:** 23, 24: 28, 37: 39, 43: 46, 47**,** 48: 50, 56**:** 59, 60: 65, 69, 71: 74, **75**: 78, 92

Again overflow at the inserted node of 60. Split it at 37 and propagate 37 into a new level.

Level 1: 37

Level 2: 18**:** 48, **60**

Level 3: 15: 21, 24: 43: 56: 71, **75**

Level 4: 10, **15** 16, 18: 20, 21**:** 23, 24: 28, 37: 39, 43: 46, 47**,** 48: 50, 56**:** 59, 60: 65, 69, 71: 74, **75**: 78, 92

Insert 8:

Level 1: 37

Level 2: 18**:** 48, 60

Level 3: 15: 21, 24: 43: 56: 71**,** 75

Level 4: **8,** 10, 15 16, 18: 20, 21**:** 23, 24:

28, 37: 39, 43: 46, 47**,** 48: 50, 56**:**

59, 60: 65, 69, 71: 74, 75: 78, 92

Insert 49:

Level 1: 37

Level 2: 18**:** 48, 60

Level 3: 15: 21, 24: 43: 56: 71**,** 75

Level 4: 8, 10, **15** 16, 18: 20, 21**:** 23, 24:

28, 37: 39, 43: 46, 47**,** 48: **49,** 50, 56**:** 59, 60: 65, 69, 71: 74, 75: 78, 92

Insert 33:

Level 1: 37

Level 2: 18**:** 48, 60

Level 3: 15: 21, 24: 43: 56: 71**,** 75

Level 4: 8, 10, 15 16, 18: 20, 21**:** 23, 24:

28, **33,** 37: 39, 43: 46, **47,** 48: 49, 50, 56**:** 59, 60: 65, 69, 71: 74, 75: 78, 92

Insert 38:

Level 1: 37

Level 2: 18**:** 48, 60

Level 3: 15: 21, 24: 43: 56: 71**,** 75

Level 4: 8, 10, 15 16, 18: 20, 21**:** 23, 24:

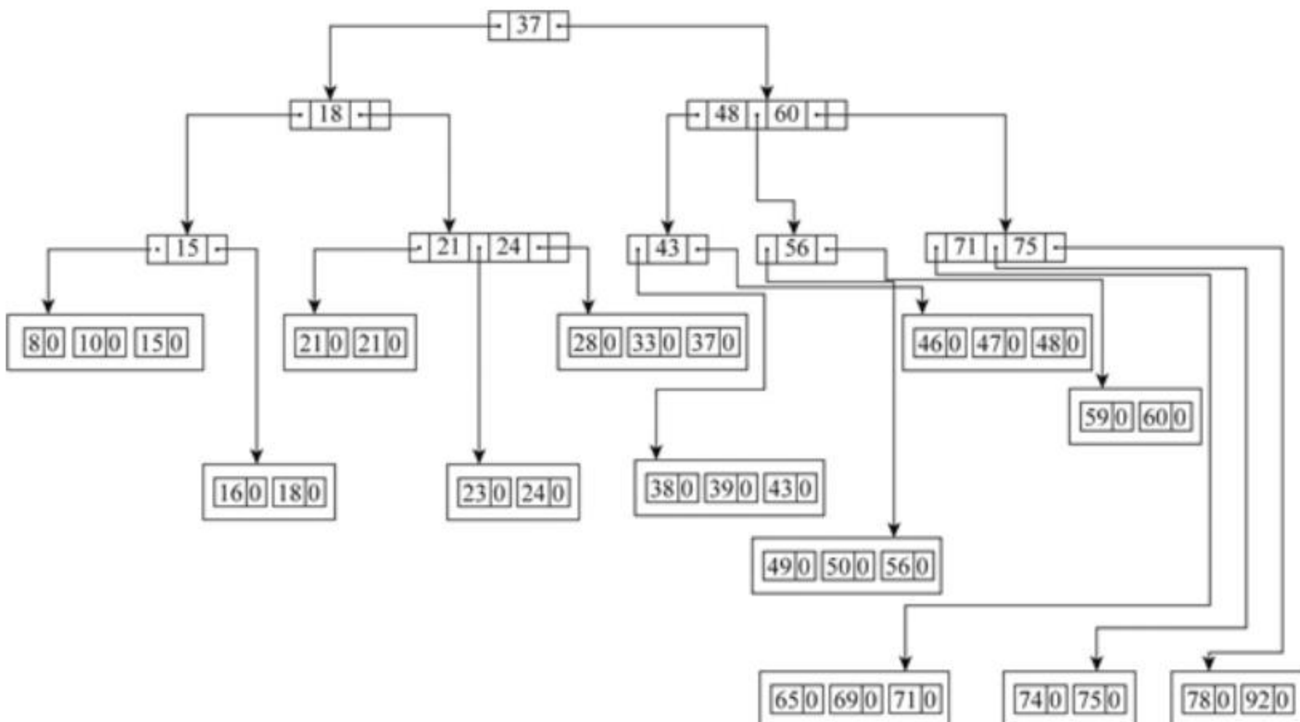
28, 33, 37: **38,** 39, 43: 46, 47**,** 48: 49, 50, 56: 59, 60: 65, 69, 71: 74, 75: 78, 92

The tree after the insertion of the last key 38 will give us the final B+ tree.

• From each node except the leaf nodes, a left pointer is there to the child nodes in which left pointer points to node having keys less than that parent node and right pointer points to the node having key values larger than that parent node.

• Each set in the above tree levels will form a node and set elements are the keys present in that node.

Graphically the final tree after the insertion of keys will look as below:



17.21. Suppose that the following search field values are deleted, in the given order, from the B$^+$-tree of Exercise 17.19; show how the tree will shrink and show the final tree. The deleted values are 65, 75, 43, 18, 20, 92, 59, 37.

In the B^+- tree deletion algorithm, the deletion of a key value from a leaf node is

(1) It is less than half full.

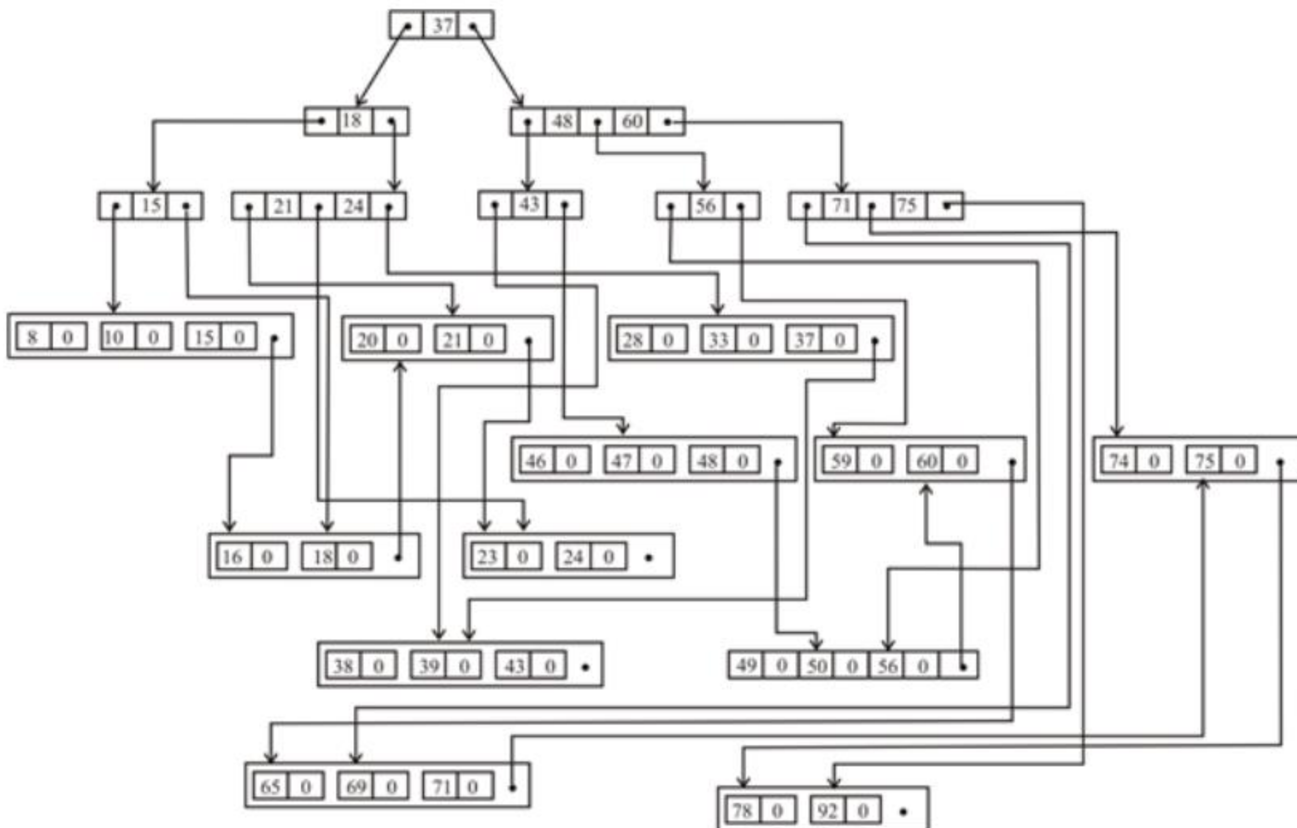
In this case, we may combine with the next leaf node.

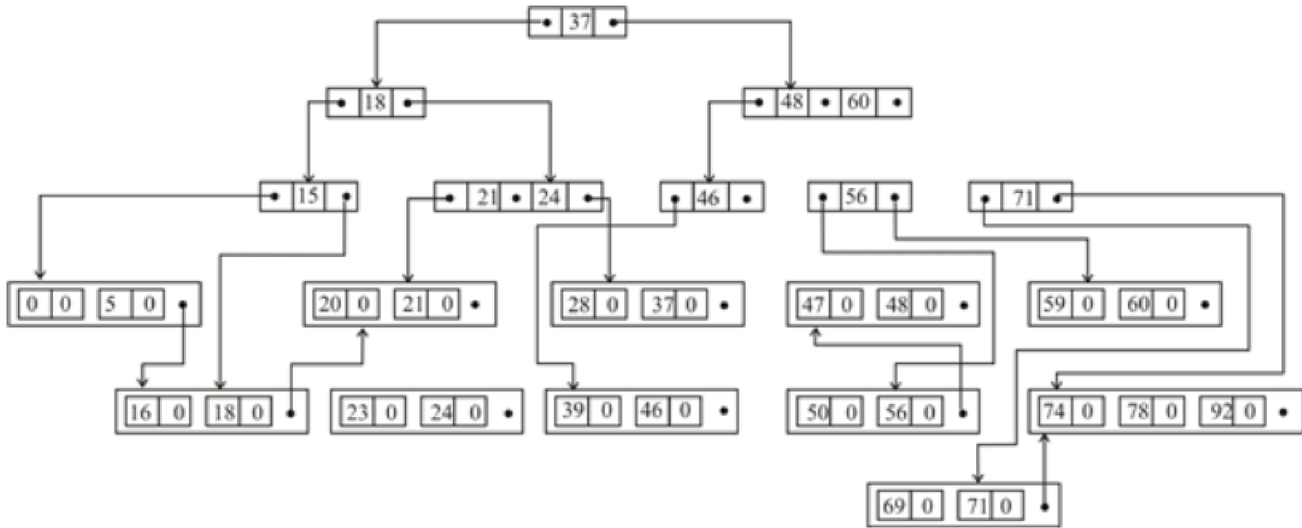
(2) If the key value is deleted from right most value. Then its value will appear in an internal node.

In this case, the key value to the left of the deleted key and left node will replaces the deleted key value in the internal node.

From the data, deleting 65 will only affect the leaf node.

Deleting 75 will cause a leaf node to be less than half. So, it is combined with the next node and also 75 is removed than the internal node.

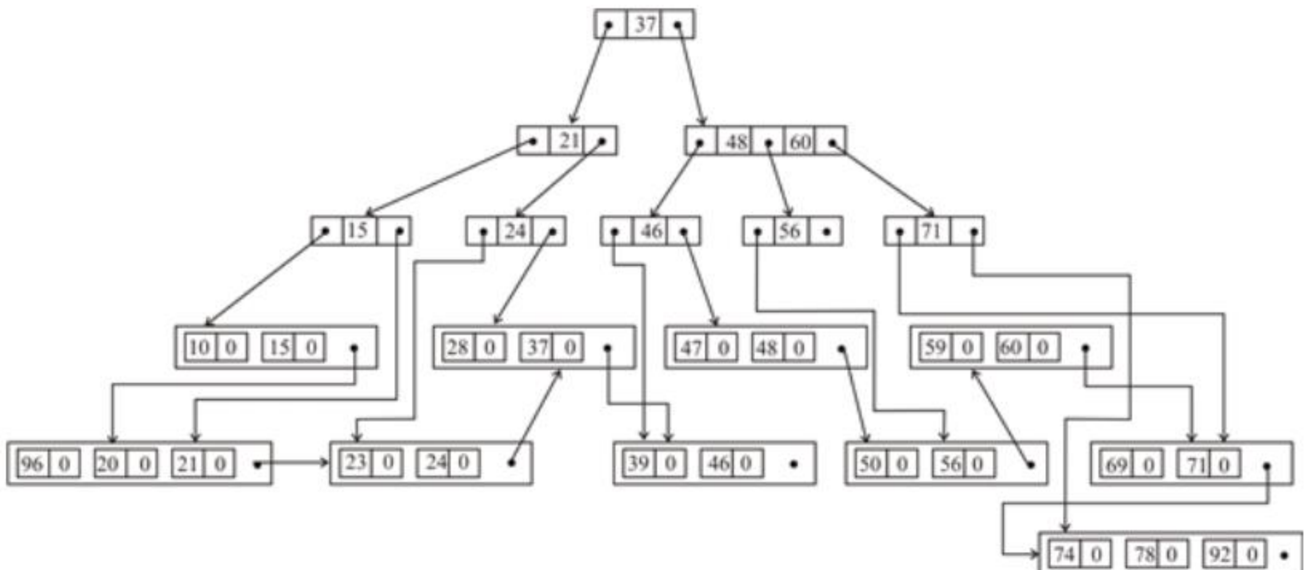




In the next step we may delete 18, it is in the right most entry in a leaf node and appears in an internal node of the $B^+ - \text{tree}$. Now the leaf node is less than half full and combined with the next node.

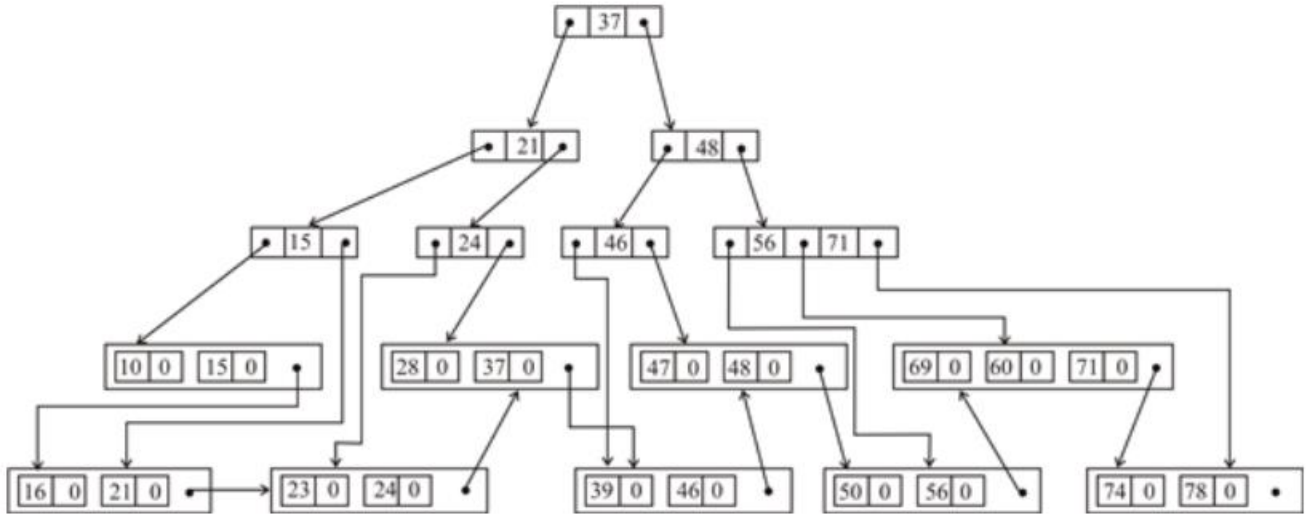
The value 18 must be removed from the internal node. Causing underflow in the internal

One approach for dealing with under flow internal nodes is to reorganize the values of the under flow node with its child nodes, so 21 is moved up into the under flow node leading to the following free.

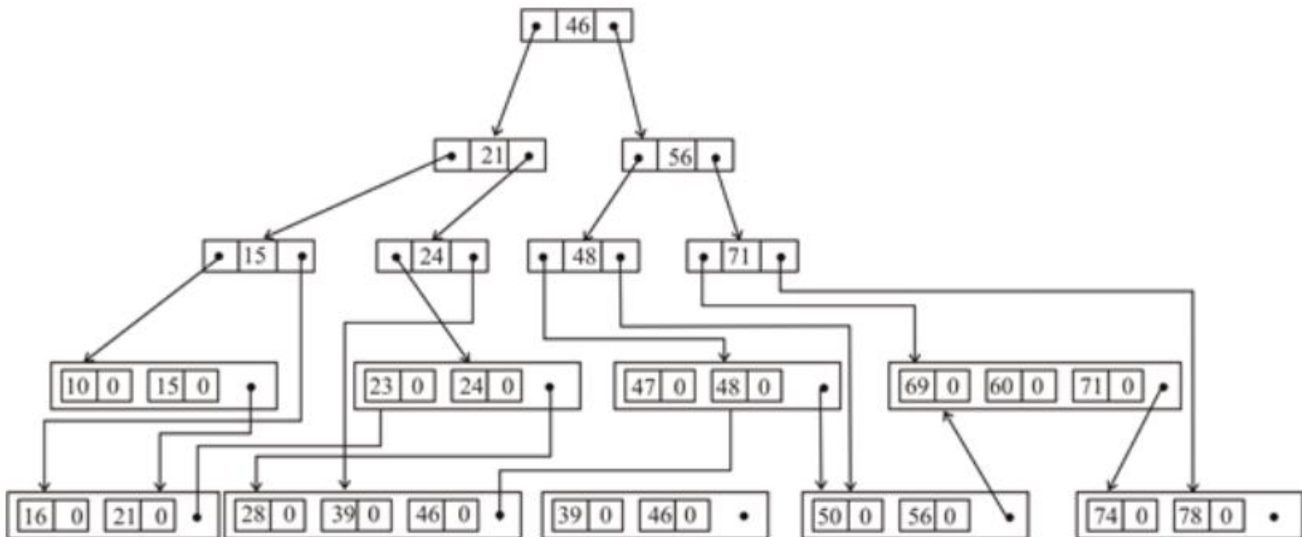


Deleting 20 and 92 will not cause under flow.

Deleting 59 causes under flow and the remaining value go is combined with the next leaf node. Hence 60 is no larger a right most entry in a leaf node. This is normally don by moving 56 up to replace 60 in the internal node, but since this leads to under flow in the node that used to contains 56 the nodes can be reorganized as follows.



Finally removing 37 causes serious underflow, leading to a reorganization of the whole tree. One approach to deleting the value on the root node is to use the right mast value in the root node is to use the right mast value in the next leaf node to replace the root an move this leaf node to the left sub tree. In this case the resulting tree may book as follows.



17.24. Suppose that several secondary indexes exist on nonkey fields of a file, implemented using option 3 of Section 17.1.3; for example, we could have secondary indexes on the fields Department_code, Job_code, and Salary of the EMPLOYEE file of Exercise 17.18. Describe an efficient way to search for and retrieve records satisfying a complex selection condition on these fields, such as (Department_code = 5 AND Job_code = 12 AND Salary = 50,000), using the record pointers in the indirection level.

An efficient way of retrieving values for Department_code = 5 AND Job_code = 12 AND Salary = 50,000 using record pointers in indirection level is:

1. Using secondary index on Deparment_code retrieve list of all record pointers for all records for which Department_code = 5

2.) Using secondary index on Job_code get value of record pointers that are among record pointers obtained in step 1 and have Job_code = 12.

Using third secondary index may cause delay, so value of salary can be matched in list of records whose pointers are obtained from step two.

If size of file is very large use of third secondary index can also be used for gaining result in a quicker way.

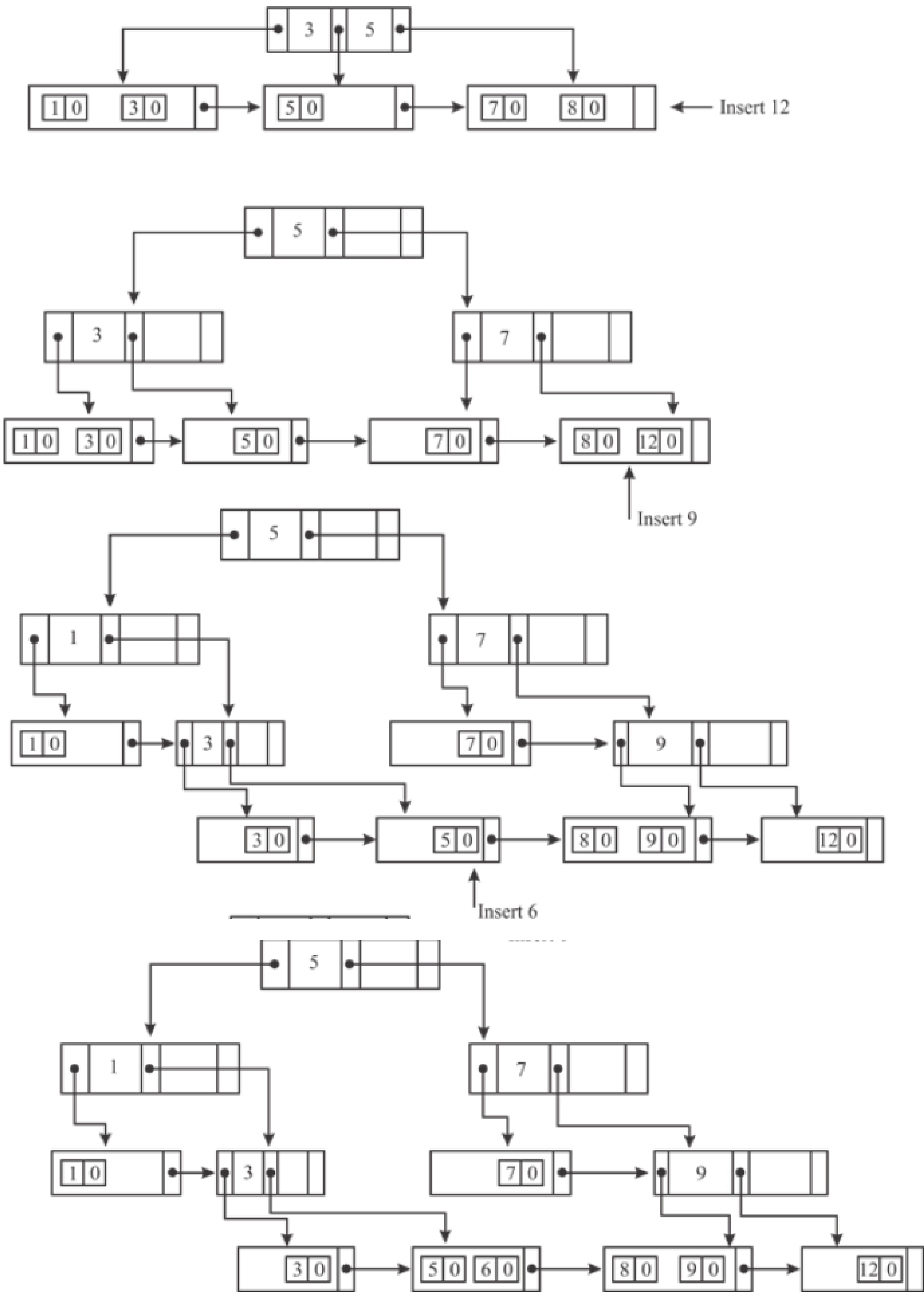
17.26. It is possible to modify the B^+-tree insertion algorithm to delay the case where a new level is produced by checking for a possible *redistribution* of values among the leaf nodes. Figure 17.17 illustrates how this could be done for our example in Figure 17.12; rather than splitting the leftmost leaf node when 12 is inserted, we do a *left redistribution* by moving 7 to the leaf node to its left (if there is space in this node). Figure 17.17 shows how the tree would look when redistribution is considered. It is also possible to consider *right redistribution*. Try to modify the B^+-tree insertion algorithm to take redistribution into account.

Refer to figure 17.17 for the redistribution of the values among the leaf nodes at a new level. The figure shows inserting the values 12, 9 and 6. In the figure, value 12 is inserted into the leaf node by moving 7 to its left leaf node through left redistribution.

The values 12, 9 and 6 can be distributed among the leaf nodes, at a new level, using right redistribution as follows:

- When a new value is inserted in a leaf node, the tree is divided into leaf nodes and internal nodes. Every value that appears in the internal node also appears as the rightmost value at the leaf level, such that the tree pointer to the left of this value points to this value.
- If a new values needs to be inserted in the leaf node and the leaf node is full, then it is split. The first $j = \lceil ((p_{\text{leaf}} + 1)/2) \rceil$ values, where p_{leaf} denotes the order of leaf nodes, present in the original node are retained and rests of the values are moved to a new leaf node. The duplicate value of the *j*th search value is retained at the parent node and a pointer pointing to the new node is created.
- This new node is inserted in the parent node. If the parent node is full then it is split. The *j*th search value is moved to the parent and values present in the internal nodes up to P_j are kept, where P_j is the *j*th tree pointer and $j = \lfloor ((p+1)/2) \rfloor$.
- The values from P_{j+1} till the last value present in the node are kept in the new internal node. The splitting of parent node and leaf nodes continues in this way and results in new level for the B^+ tree.

The modified B^+ tree insertion algorithm based on the right redistribution is as follows:



CH18

18.1. Discuss the reasons for converting SQL queries into relational algebra queries before optimization is done.

Following are the reasons for converting SQL queries into relational algebra queries before performing optimization:

- SQL queries are declarative whereas relational algebra queries are procedural.
- SQL queries do not specify a query execution plan. Relational algebra queries specify a query execution plan.

Optimization is performed well when there is a query execution plan. Hence, SQL queries are converted into relational algebra queries before performing optimization.

18.2. Discuss semi-join and anti-join as operations to which nested queries may be mapped; provide an example of each.

Semi-join and anti-join operations:

Different SQL queries like random queries, report generation queries, canned queries from different type of enterprise applications are served by RDBMS. Some of these queries are changed into operations, called semi-join and anti-join such that these operations are not the part of relational algebra.

Consider the schema given below that will be used for example:

Order (Oname, Onumber, Orderdate, Cid)

Customer (Ssn, Address, Cname, Customerid)

Semi-join operation on nested query:

- Semi-join is used with nested queries, for removing the IN, ANY and EXISTS nested subqueries.
- Consider the query given below, which is joined to the nested subquery by the connector IN.

SELECT COUNT (*)

FROM Customer C

WHERE C.Customerid IN (SELECT Order.Cid

FROM Order O

WHERE O.Orderdate = '3/30/2014')

- The above query counts the number of customers who ordered on the date 3/30/2014. The operation retrieves the customer rows whose Customerid attribute matches the values for Cid attribute in Order table, for a particular order date.

• To remove the nested subquery, semi-join operation will be performed, which is displayed by non-standard notation “S=”, as follows:

SELECT COUNT *

FROM Customer C, Order O

WHERE C.Customerid S= O.Cid and O.Orderdate='3/30/2014';

Anti-join operation on nested query:

• Anti-join is used with nested queries, for removing the NOT IN, ALL and NOT EXISTS nested subqueries.

• Consider the query given below, which is joined to the nested subquery by the connector NOT IN.

SELECT COUNT (*)

FROM Customer C

WHERE C.Customerid NOT IN (SELECT Order.Cid

FROM Order O

WHERE O.orderdate = '3/30/2014');

• The query counts the number of customers who did not order on date 3/30/2014. The operation finds out the customers whose Customerid does not matches the values for Cid attribute of Order table, for a particular order date.

• To remove the nested query, anti-join operation will be performed, which is displayed by the symbol “A=”, as follows:

SELECT COUNT (*)

FROM Customer C, Order O

WHERE C.Customerid A= O.Cid and orderdate='3/30/2014';

18.3. How are large tables that do not fit in memory sorted? Give the overall procedure.

Large tables that do not fit in main memory are sorted using external sorting and they have to reside in secondary storage like hard disk, tap. The external sorting can handle massive amount of data.

The external sorting algorithm is based on sort-merge strategy in which small sub-files, known as **runs**, are sorted and then these sorted sub files are merged into large sub-files that are sorted.

The buffer space in main memory is required by the sort-merge algorithm, which is used for actual sorting and merging strategies. Buffer space is the part of DBMS cache—a DBMS controlled are in computer's main memory.

The buffer space is partitioned into small buffers, its size is equal to the size of disk block to hold individual disk blocks.

The two phases of external sort are as follows:

- **Sorting phase:**

- All the runs of the large file are first read and then sorted in main memory using internal sorting algorithm and written back to disk in the form of sub-files (runs) that are temporarily sorted.

- Individual run size and number of initial runs (n_R) are determined by the number of file blocks (b) and available buffer space (n_B).

- For example, if $b = 1024 \text{ disk blocks}$ and $n_B = 10 \text{ disk blocks}$ then $n_R = \lceil b / n_B \rceil = 103 \text{ initial runs}$. Hence 103 sorted runs are stored in the form of sub-files that are temporarily sorted.

- **Merging phase:**

- This phase requires one or more merge passes for merging the sorted runs into one output file.

- The degree of merging (d_M) is the number of sorted runs (sub-files) that are to be combined in each merging step.

- In order to store individual disk block belonging to each sorted runs, one buffer block is required and one additional buffer block is required to store one disk block corresponding to the merge result.

Therefore, degree of merging is the smaller of n_R and $(n_B - 1)$ and number of merge passes is represented by $\lceil (\log_{d_M}(n_R)) \rceil$.

- In the previous example, $n_B = 10$, also $d_M = 9$. This implies that at the end of first merge pass, 103 initial runs would be merged into 12 larger sorted runs taken 9 at a time.

- Then these 12 sorted runs are then merged into 1 sorted run taken 9 at a time and then finally into 1 fully sorted file. This merging process requires 3 passes.

18.4. Discuss the different algorithms for implementing each of the following relational operators and the circumstances under which each algorithm can be used: SELECT, JOIN, PROJECT, UNION, INTERSECT, SET DIFFERENCE, CARTESIAN PRODUCT.

Various algorithms for implementing relational operator:

1.) **SELECT**: There are various search algorithms that can be used to implement a SELECT operation:

Algorithms for simple selection:

a.) Linear Search (brute force): Retrieve every record in the file, and test weather its attribute values satisfy selection condition.

b.) Binary Search: If the selection condition involves an equality comparison on key attribute on which the file is ordered, binary search- which is more efficient than linear search- can be used.

c.) Using primary key (or hash key): If the selection condition involves a comparison on key attribute with primary index. This condition retrieves a single record.

d.) Using a primary index to retrive multiple records: If comparison condition is >, >=, <, <= on a key field with primary index use the index to find the record satisfying the corresponding equality condition.

e.) Using clustering index to retrieve multiple records: If the selection condition involves an equality comparison on a non key attribute with a clustering index, use index to retrieve all the records satisfying the condition.

f.) Using a secondary index on an equality comparison: This search 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 comparison involving >, >=, <, <=

Algorithms for complex selection:

a.) 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 for simple search but linear search, use that condition to retrive records and then check weather each retrieved record satisfies the remaining simple condition in the conjunctive condition.

b.) Conjunctive selection using composite index: If two or more attributes are involved in equality conditions in the conjunctive condition and a composite index exists on the combined fields, we can use the index directly.

c.) Conjunctive selection by intersection of record pointers: Id 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, then each index can be used to retrieve the set o record pointers that satisfy the individual condition. The intersection of these sets of record pointers gives the record pointer that satisfy the conjunctive condition, which are then used to retrieve those records directly. If only some of the conditions have secondary indexes, each retrieved record is further tested to determine weather it satisfies the given conditions.

2.) **JOIN**: Methods for implementing Joins are:

a.) Nested- Loop join (brute force): For each record t in R(outer loop), retrieve every record s from S (inner loop) and test weather the two records satisfy the join condition (t[A] = s[B])

b.) Single loop join (using an access structure to retrieve the matching records): If an index exists for one of the two join attributes-say, 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 tha satisfy s[B] = t[A].

c.) Sort- merge Join: If records of R and S and physically sorted by value of the join attributes A and B, resp., we can implement the join in the most efficient way possible. Both files are scanned concurrent in order of the join attributes, matching the records that have the same values for A and B. If the files are not sorted, thy ma be sorted first by using external sorting. In this method, pairs of file blocks are copied into memory buffers in order and the records of each file are scanned only once each for matching with the other file- unless both A and B are non key attributes, in which case the method needs to be modified slightly.

d.) 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. First, a single pass through the file with fewer records hashes its record to the hash file buckets; this is called partitioning phase. In second phase, called the probing phase, a single pass through the other file (S) then hashes each of its records to probe the appropriate bucket, and that record is combined with all matching records from R in that bucket. This simplified description of hash- join assumes that the smaller of to the files fits entirely into memory bucket after the first phase.

3.)**PROJECT**: A PROJECT operation is straight forward t implement if includes a key of relation R, because in this case relation will have same number of tuples as R, but with only the values for the attributes in in each tuple. If does not include a key of R, duplicate tuples may not be eliminated. This is usually done by sorting the result of the operation and thn eliminating duplicate tuples, which appear consecutively after sorting.

Hashing can also be used to eliminate duplicates: as each record hashed and inserted into a bucket of the hash file in memory, it is checked against those already in the bucket; if it is a duplicate, it is not inserted.

4.) **, INTERSECTION, SET DIFFERENCE**: Customary way to implement these operations is to use variations of the sort- merge technique: the two relations are sorted on the same attributes, and, after sorting, a single scan through relation is sufficient to produce the result.

For operation can be implemented by scanning and merging both sorted files concurrently, and weather the same tuple exists in both relations, only one is kept in the merged result.

For INSERTION operation, merged result only those tuples that appear in both relations

Hashing can also be used to implement , INSERTION, and SET DIFFERENCE. One table is portioned and the other is use to probe appropriate partition.

To implement UNION, first hash the records of R; then hash the records of S, but do not insert duplicate records in the buckets.

To implement INTERSECTION, first partition the records of R to hash file. Then, while hashing each record of S, probe to check id an identical record from R is found in the bucket, and if so add record to result file.

To implement SET DIFFERENCE, first hash the records of R to the hash file buckets. While hashing each record of S, if an identical record is found in th bucket, remove that record from the bucket.

7.) **CARTESIAN PRODUCT**: The Cartesian product operation is quite operation R X S is quite expensive because its result includes records from R and S. Additionally; the attributes of the result include all attributes of R and S. If R has n record and j attributes and S has m records and k attributes, the result relation will have n*m records and j+k attributes. Hence it is important to avoid CARTESIAN PRODUCT operation and to substitute other equivalent operations during query optimization. For example if we need Cartesian product on selected attribute value for a relation and another selected value of other relation we must apply select operation prior to CARTESIAN PRODUCT this reducing number of tuples that will be part of product.

18.4. Give examples of a conjunctive selection and a disjunctive selection query and discuss how there may be multiple options for their execution.

Conjunctive selection query is composed of various conditions that are connected to each other via AND logical connector. An example of conjunctive selection query is:

SELECT DIRECTOR.Dname

FROM EMPLOYEE, DIRECTOR

WHERE EMPLOYEE.Deptno = DEPARTMENT.Deptno AND

LOCATION = DELHI;

Disjunctive selection query is composed of various conditions that are connected to each other via OR logical connector. An example of conjunctive selection query is:

SELECT DIRECTOR

FROM CAST, PLAYS

WHERE CAST.Act = PLAYS.Act OR COMPOSER = Al Pacino;

Above conjunctive selection query can be executed through various options as discussed below:

• **S8—Using an individual index:** If one of the search methods from S2 to S6 is applicable to the access path of an attribute, related to the conjunctive select condition, then that condition can be used to fetch the records. Fetching the records is followed by checking, whether rest of the conditions are satisfied by the fetched record.

• **S9—Using a composite index:** If the equality conditions involves two or more attributes in the conjunctive select condition and a composite index is present on the combined fields, then index can be used directly.

• **S10—By intersection of record pointers:**

• If various fields, that are included in simple conditions of conjunctive select condition, contains the secondary indexes and if indexes contains the record pointers, then the set of record pointers, satisfying the given conditions, can be fetched through each index.

• The record pointers that satisfy the conjunctive select condition are obtained through intersection of record pointers. These record pointers can be further used to fetch the records.

• In case only few conditions comprises of secondary indexes, then verification of each fetched record is performed to determine whether fetched record satisfies the remaining conditions.

The disjunctive selection query can be executed through various options as discussed below:

• **Search method for disjunctive selection:** Disjunctive condition comprises of simple conditions that are connected by the OR logical operator. This selection process is harder to process and optimize.

Consider the condition:

$$\sigma_{\text{Dno=5 OR Salary>30000 OR Sex='F'}}(\text{EMPLOYEE})$$

• The union of the records, that satisfies the individual conditions, produces the records that comply with the disjunctive condition. If any condition does not have access path, then brute force or liner search is used.

The selection can be optimized only in case, if there exists an access path corresponding to every disjunctive select condition. This optimization takes place by fetching the records corresponding to satisfied conditions, followed by eliminating duplicate records using union operation.

• The query optimizer should use appropriate search methods, from S1 to S7, for the successful execution of each SELECT operation in a disjunctive selection query.

18.5. Discuss alternative ways of eliminating duplicates when a "SELECT Distinct <attribute>" query is evaluated.

Instead of using DISTINCT operator to retrieve unique records, some other alternatives are as follows:

1. **Union:** Union operator can also be used to eliminate duplicate rows when combining two or more SQL statements.

SELECT City FROM Suppliers UNION SELECT City FROM Customers ORDER BY City;

In the above query, if various customers and suppliers share the same city then every city will be listed only once.

2. **Intersect:** Intersect operator can be used to eliminate duplicity by returning the rows from the first select statement that matches with rows from second select statement.

SELECT Department_name FROM Department INTERSECT SELECT Department_name FROM Employee ORDER BY Department_name;

The above query returns all the department names only once, which are common in both the tables.

3. **Except:** Except operator returns all the rows that are present in the first select statement and not in the second select statement.

SELECT supplier_id FROM Location

EXCEPT

SELECT supplier_id FROM Suppliers

ORDER BY Supplier_id;

The above query returns all the supplier ids that are present in the Location table but not present in the Suppliers table.

18.6. How are aggregate operations implemented?

Aggregate operations:

Table scan or an appropriate index can be used for computing aggregate operators. Various available aggregate operators are MIN, MAX, AVERAGE, COUNT, SUM.

Implementation of aggregate operations:

• Consider relation STUDENT and an attribute Marks.

SELECT MIN (Marks)

FROM STUDENT;

• Assume B^+ tree index exists on Marks for STUDENT relation. The query optimizer uses Marks index to search the lowest marks value in the index node from the STUDENT relation and follows the leftmost pointer in each node from the root to the leftmost leaf.

The lowest Marks value would be the last entry of that index node. Using such operators is more efficient than full table scan.

• The B^+ tree index can also be used to calculate the aggregate functions AVERAGE and SUM, provided it should be a dense index that is, if there exists an index entry corresponding to each record in the main file. In this case computation is applied on index value.

• In a nondense index, actual number of records associated with individual index value is a prerequisite for computation.

• When COUNT (*) function is applied to STUDENT, the result is retrieved directly from catalog. Catalog is the place where the output of the COUNT function, that is the number of records in STUDENT relation, is stored.

• The aggregate operator should be separately applied to each group of tuples, in case GROUP BY clause is used. The relation must be segregated into tuples, such that for the grouping attributes every partition carries the same value.

Consider the query:

SELECT roll_no, AVG (Marks)

FROM STUDENT

GROUP By roll_no;

In such queries, first sorting or hashing should be used on grouping attributes. The above query involves the grouping of tuples of STUDENT, corresponding to every roll number, in a partition. Also, average marks would be computed for each group.

18.7. How are outer join and non–equi-join implemented?

Outer join is implemented as follows:

Outer join combines records from two tables such that both the tables may not be having matching records. Outer join is further divided in three types:

• **Left outer join:** The result of left outer join of two tables is all the rows of the left table that matches with the rows of the right table. In case the rows of left table don't matches with the rows of right table, NULL is displayed.

Consider the following query:

SELECT E.Ename, D.Dname

FROM EMPLOYEE LEFT OUTER JOIN DEPARTMENT

ON EMPLOYEE.Deptno = DEPARTMENT.Deptno;

The above query will retrieve all the rows containing the name of the employees and all the departments associated these employees, as a result. If left table does not have associated department, then its value is entered as NULL.

The sort-merge and hash algorithm can be used to perform outer join. The left outer join is combination of inner join and anti-join.

Algebra notation for left outer join:

• Perform the inner join of DEPARTMENT and EMPLOYEE tables.

QUERY1 ← $\pi_{Ename,Dname}$(EMPLOYEE ? Deptno = Deptno DEPARTMENT)

• Retrieve the rows from EMPLOYEE that are not present in the inner join, by performing anti-join operation.

QUERY2 ← π_{Ename}(EMPLOYEE) − π_{Ename} (QUERY1)

• Perform null padding on Dname field in QUERY2.

QUERY2 ← QUERY2 × NULL

• Obtain the left outer join result by performing union operation on QUERY1 and QUERY2.

RESULT ← QUERY1 ∪ QUERY2

The estimated cost of outer join is the combined cost of all the above steps.

Computing left outer join using algebraic notation creates and stores temporary tables, and later these tables are processed several times. In order to reduce running and storing overhead of outer join, inner join anti-join combination should be used.

• **Right outer join:**

Right outer join is similar to left outer join, but involves the switching of operands.

• **Full outer join:**

Full outer join combines the result obtained through inner join of two tables and extra unmatched rows from the left and right tables.

Non-equi-join is implemented as follows:

Non equi join uses inequality operator for performing inner join and outer join operations. Here rows are matched from different tables using inequality operators and desired result is obtained. The operators like $<,\ >,\ \leq,\ \geq,\ \neq$ are used in this type of join. In non-equi-join, hash based algorithms can’t be used.

18.8. What is the iterator concept? What methods are part of an iterator?

Iterator:

• Reading some one or more files as input, processing it and producing a corresponding output file in the form of relation, are some of the steps involved in various algorithms for algebraic operations. If in this process, the output generated is one tuple at a time then it is referred to as iterator.

• Iterator uses an approach opposite to materialization. In Materialization approach entire relation is created as a result, stored on primary memory and verified again by the next algorithm.

• The query plan can be executed depending on invocation of iterators in a certain order.

The iterator consists of following three methods:

• **Open ():** When Open () is invoked, it initializes operator by loading the data structure required for operator and allocating buffers for the operator's input and output.

Open() method passes the necessary arguments required to perform the operation and further calls Open () to retrieve the needed arguments.

• **Get_Next ():** This method works on input arguments and calls the code corresponding to the input's operation.

The function returns the next output tuple and updates the state of iterator to determine the amount of input processed. Get_Next () puts some special value in output buffer, when it is not possible to return more tuples.

• **Close():** This method is called when all the requested tuples are retrieved or the needed tuples have been returned, and ends the iteration process.

Iterator method can be used in accessing method. Hash based index or accessing B^+ tree can be thought of a function that can be executed in the form of iterator; it also produces output as a series of tuples. Pipelining strategy can also be used with iterator when implemented operator can process the tuple completely.

18.9. What are the three types of parallel architectures applicable to database systems? Which one is most commonly used?

Parallel architecture for query processing in DBMS:

In parallel database architecture, system is designed in such a way that various operations like executing queries, loading data can be performed by multiple processors and memory in parallel.

Three types of parallel database architecture are as follows:

• **Shared memory architecture**:

• Shared memory is like conventional architecture, in which a common main memory is accessed by interconnected multiple processors. All the processors can access main memory, but it creates bottleneck and memory access becomes slow.

• One disadvantage of this architecture is that there is increasing conflict on the common memory as the number of processor increases

• **Shared disk architecture:**

• All the processors has their own main memory and all systems can share each other's disks that are interconnected on a common network.

• It is not necessary that every processor has a disk of its own.

• There is more conflict for the limited network bandwidth. Storage area network (SAN), network attached system (NAS) are examples of this type of architecture.

SANs transfers data in the form of units of blocks and NAS employs file transfer protocol for transferring the files.

• **Shared nothing architecture**:

• In this architecture neither memory nor disk is shared. Each processor has its own memory and disk storage.

Most commonly used architecture:

• The shared nothing architecture is most widely used architecture. It overcome limitations of other two type of architectures. It is less expensive.

• When a processor require some data from another processor, it sends a request to the processor containing data, then the processor collect data from its disk and send it back to requested server.

• It provides parallelism in query processing in three ways:

• Individual operator parallelism

• Inter-query parallelism

• Intra-query parallelism

• As the number of processors and memory increases, shared nothing architecture provides linear speed-up (reduce the time taken in processing) and scale-up (sustain performance even more resources are added).

18.10. What are the parallel implementations of join?

The parallel join is used to split the relation which has to be joined in such a manner that the join is partitioned into multiple *n* smaller joins. After portioning, perform the smaller joins in parallel on n processor. At the end, take a union of the result.

Parallel join techniques:

Equality based partitioned join:

If two relations A and B are partitioned in *n* partitions and being processed by *n* processors, and one partition of relation A_i and one partition of relation B_i are processed by same processor then natural join or equi-join will take place.

But the partitions must not be overlapped on join key.

Inequality join with partition and replication:

In join condition, if comparison operators (<, >, <=, >=, ≠) are used, then parallel natural join is not possible.

Two cases in which parallel join can happen:

a) **Asymmetric case:** If relation A is much smaller than relation B then, the relation B is partitioned and relation A is replicated to all the processers where B_i is, and join operation is performed between B_i and A at the processor P_i.

b) **Symmetric case:** In this case both the relations are partitioned and replicated in corresponding *n, m* ways. To accomplish parallel join *n***m* processors are used.

Each processor has replicas of both the relations that are used in local join. In case of equi-join, the cost of partitioning with replication costs more.

Parallel partitioned hash join:

When two partitions to be joined are very large in size and even after partition, join them locally is too costly, the hash join is used.

- Using hash function h_1, each tuple of relations are mapped to each processor. Within each processor tuples of the relations are partitioned using hash function h_2.
- Partitioning hash algorithm is executed locally on each processor and local join is performed between partitions of both the relations.
- The union operation is performed on the result from all the processors to retrieve final result.

18.11. What are intraquery and interquery parallelisms? Which one is harder to achieve in the shared-nothing architecture? Why?

Inter-query and intra-query parallelism:

Inter-query and intra-query are the level on which parallelism is achieved in shared nothing architecture.

Intra-query parallelism:

It is form of parallelism, in which a single query is partitioned or executed among multiple processors and then the processing takes place parallel.

• One more method to execute operations in parallel is operation tree where independent operations are executed in parallel.

• If the output of the operation is retrieved as tuple by tuple and entering into another operation, then it is called pipelines parallelism.

The intra-query parallelism is used to speed up a single complex queries which is long running. It is also used for scientific calculations.

Inter-query parallelism:

This is the execution of multiple queries in parallel with one another on many processors. It increases transaction throughput.

The goal of incorporating inter-query parallelism is to scale-up query processing system. It is because the single-processor multiuser system themselves are designed to support concurrency control among transactions with the aim of increasing transaction throughput.

Implementation of **inter-query architecture is harder** to achieve in the shared nothing architecture.

The implementation of inter-query architecture is harder to achieve in the shared nothing architecture due to the following reason:

• In shared nothing architecture logging and locking must be done through message passing that is much costlier than shared memory architecture, it causes cache coherency problem.

• In shared memory architecture, it is easy to implement inter-query parallelism because lock information and logs are maintained in same memory, so easy to handle query.

18.12. Under what conditions is pipelined parallel execution of a sequence of operations prevented?

Pipelining:

In the case of materialized evaluation, large temporary materialized files are created. These files are then written on the disk, which is a time consuming process and increases the query processing overhead.

The process of combining various relational operations into one and avoiding the writing of temporary files onto a disk is called pipelining.

Blocking of pipelined parallel execution:

• A parallel execution of a query can be achieved by using a parallel algorithm corresponding to every operation involved in the query. The data which is input to a particular operation is partitioned.

Another way to achieve parallel execution of a query is through the evaluation of an operator tree. In this evaluation, the operations do not depend on each other and hence can be executed in parallel on separate processors.

If the output from one of these operations can be generated tuple by tuple and this output acts as input to another operator, then this result is known as pipelined parallelism.

• An operator, that is not relevant to execution and does not produce any output, until it has taken all the inputs, blocks the pipelining process. Hence, such operators can prevent the execution of pipelined parallelism.

Exercises

18.13. Consider SQL queries Q1, Q8, Q1B, and Q4 in Chapter 6 and Q27 in Chapter 7.

a. Draw at least two query trees that can represent *each* of these queries. Under what circumstances would you use each of your query trees?

b. Draw the initial query tree for each of these queries, and then show how the query tree is optimized by the algorithm outlined in Section 18.7.

c. For each query, compare your own query trees of part (a) and the initial and final query trees of part (b).

Take the queries from given data in text book.

Q1

SELECT F name, L name , Address

FROM EMPLOYEE, DEPARTMENT,

WHERE D name = ‘ Research ‘ AND D number = D no ;

Q1B

SELECT E. F name , E. Name , E . Address

FROM EMPLOYEE E, DEPARTMENT D.

WHERE D. Name = ‘Research ‘ AND D.D number = E. D number ;

Q8

SELECT E. F name , E. L name , S.F name , S.L name

FROM EMPLOYEE AS E, EMPLOYEE AS S;

WHERE E. Super – ssn = SSsn, ;

Q4

SELECT DISTINCT P number

FROM PROJECT, DEPARTMENT, EMPLOYEE.

WHERE Dnum = Dnumber AND Mgr-SSn = SSn

AND Lname = ‘Smilth’;

Q27

SELECT Pnumber, Pname, COUNT (*)

FROM PROJECT, WORKS ON, EMPLOYEE

WHERE Pnamer = Pno AND SSn = AND Dno = 5.

GROUP BY Pnumber, Pname.

So, in these above queries, let consider the SQL query $4 form the chapter 8.

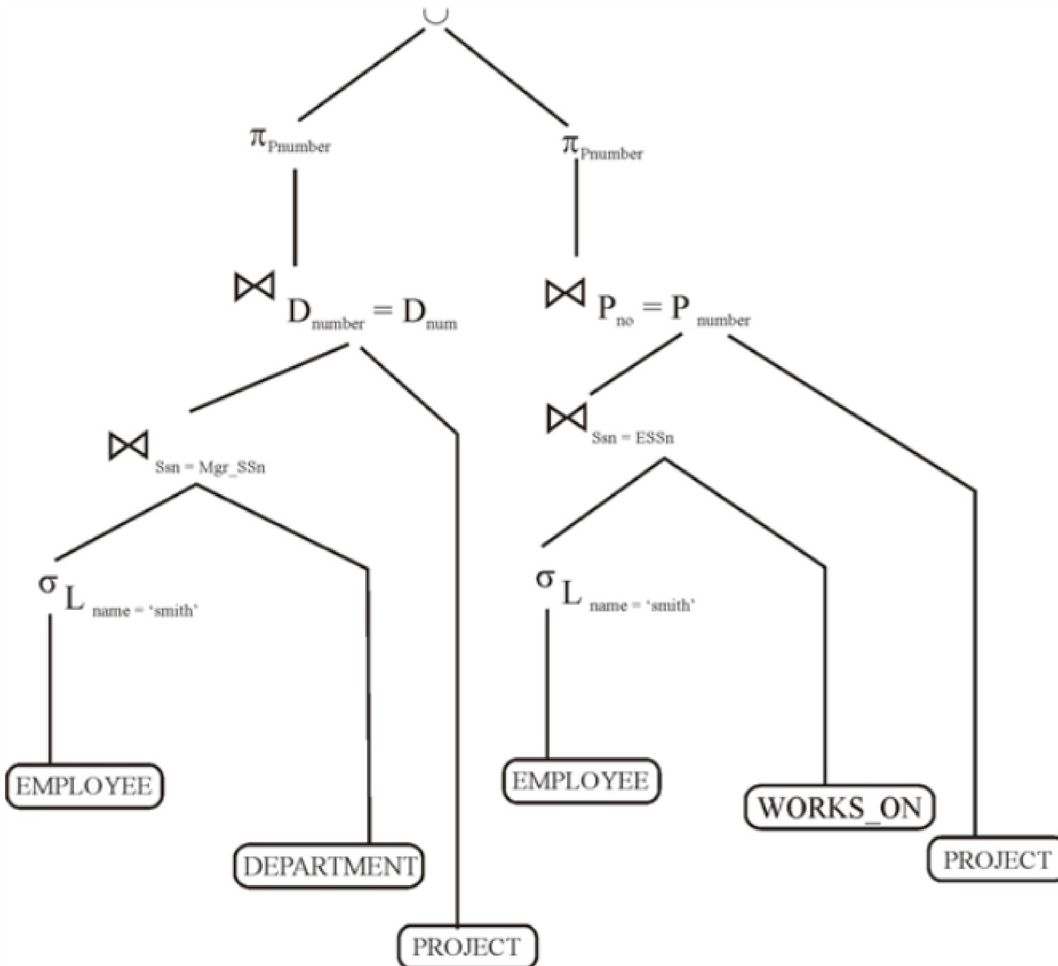
Now we may write the Relational Algebra Expression of the Query 4.

$$A \leftarrow (\text{PROJECT} \bowtie_{\text{Dnum=Dnumber}} \text{DEPARTMENT} \bowtie_{\text{Mgr_SSn=SSn}} (\sigma_{\text{Lname='smith'}} \text{EMPLOYEE}))$$

$$B \leftarrow (\text{PROJECT} \bowtie_{\text{P number=P No}} \text{WORKS_ON} \bowtie_{\text{ESSN=SSn}} (\sigma_{\text{Lname='smith'}} \text{EMPLOYEE}))$$

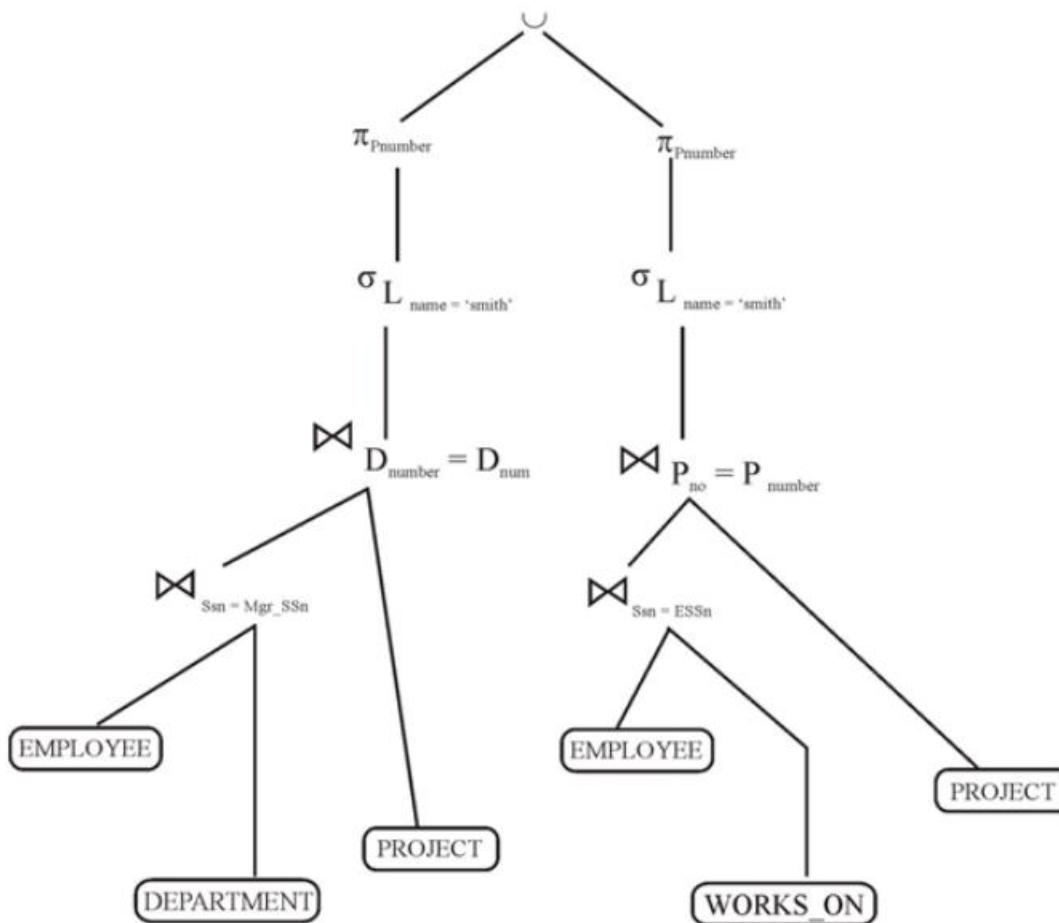
$$Q4 \leftarrow \pi_{\text{pnumber}} (\text{A}) \cup \pi_{\text{P}_{number}} (\text{B})$$

(a)



The above relational algebra expression shows the several optimization steps.

It the data base is not big and the numbers of attributes in the table are small then the above expression is used. In this time, applying the PROJECT operations before joining may add more over head to the execution time.

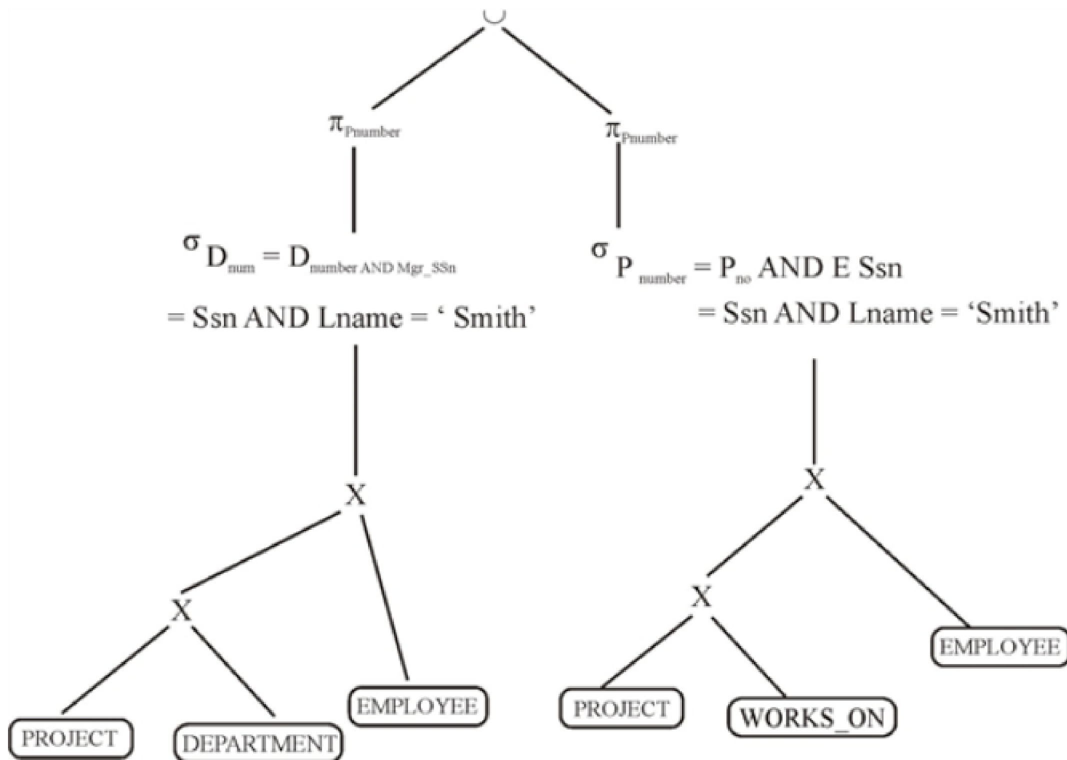


The above tree is another query tree.

In this tree, the SELECT operations are executed right before the UNION.

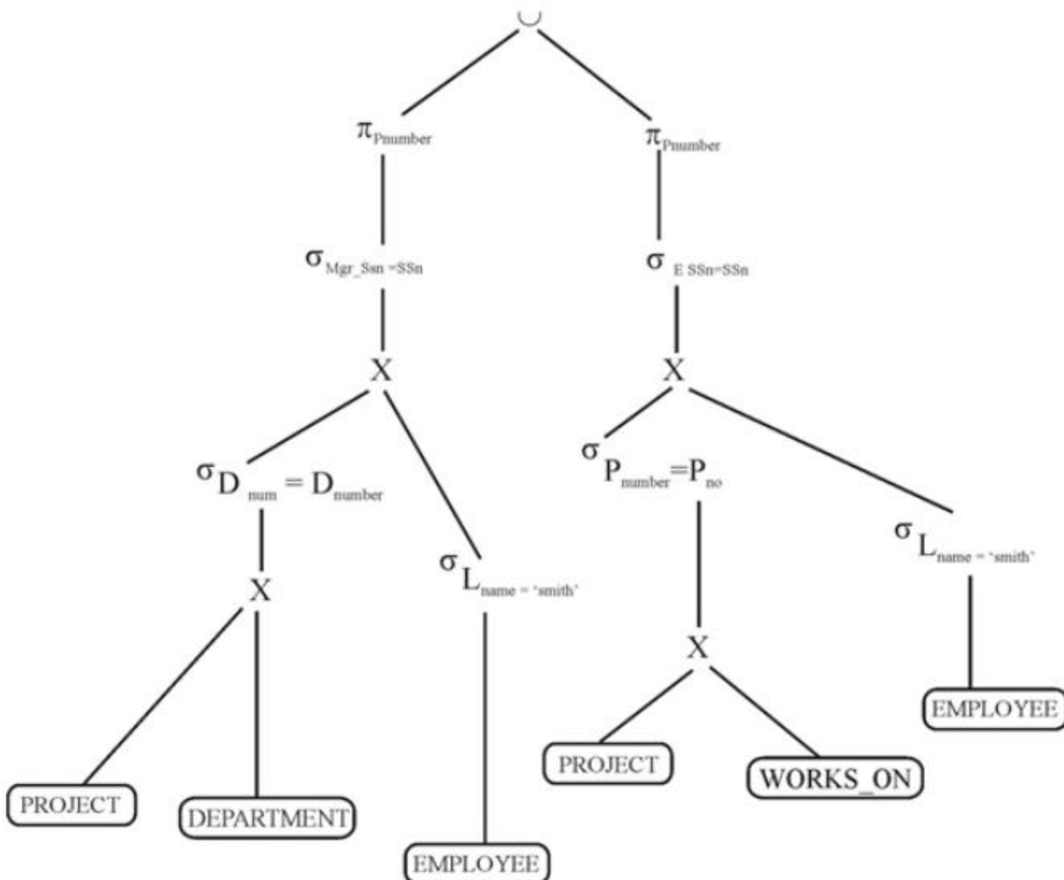
This tree can be tolerated in the circumstance that the number of employees is small.

(b) Initial tree

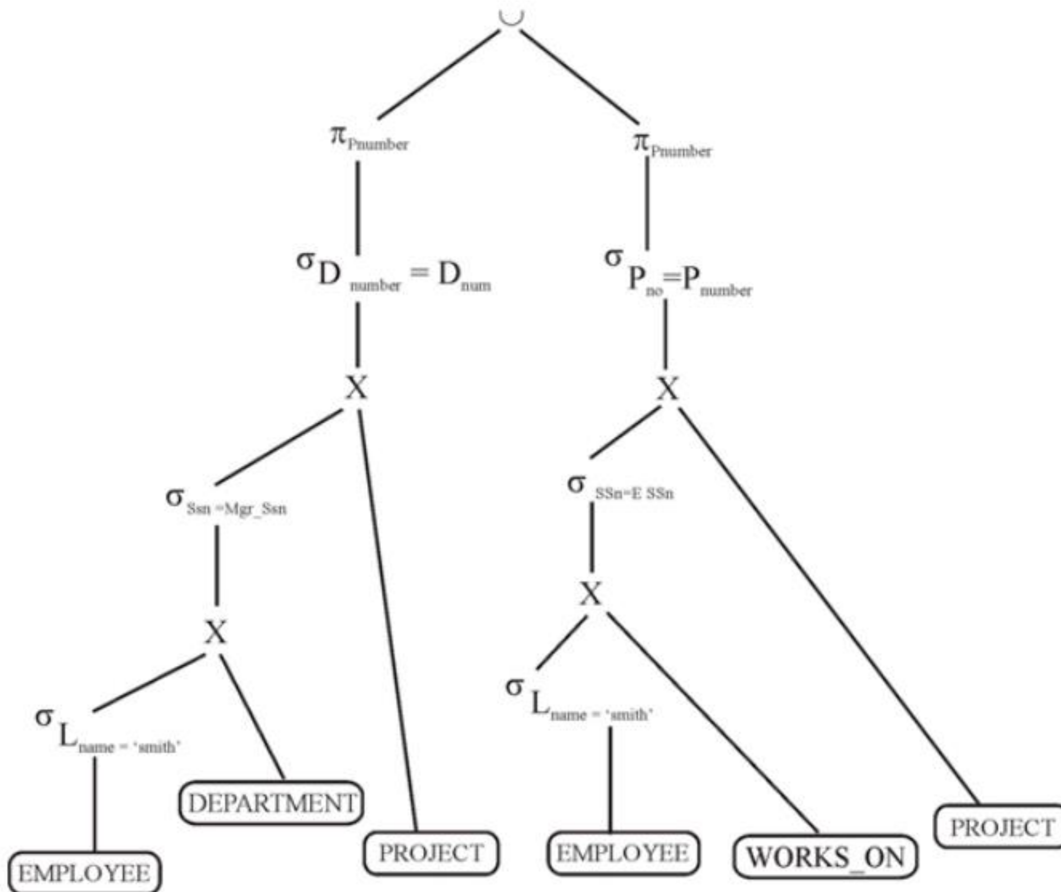


Now, from section 15.7, we may take the query tree and optimized by the algorithm.

Then the SELECT operations are moving down. So, the query tree is shown like this.

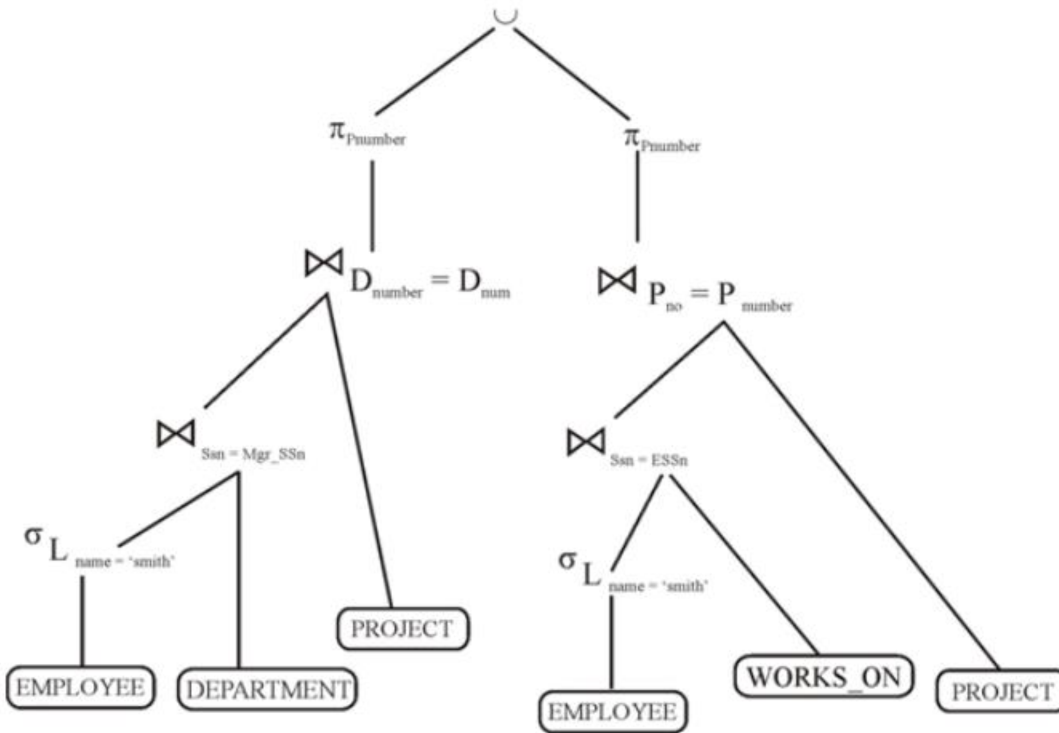


Now applying the more SELECT operations first.

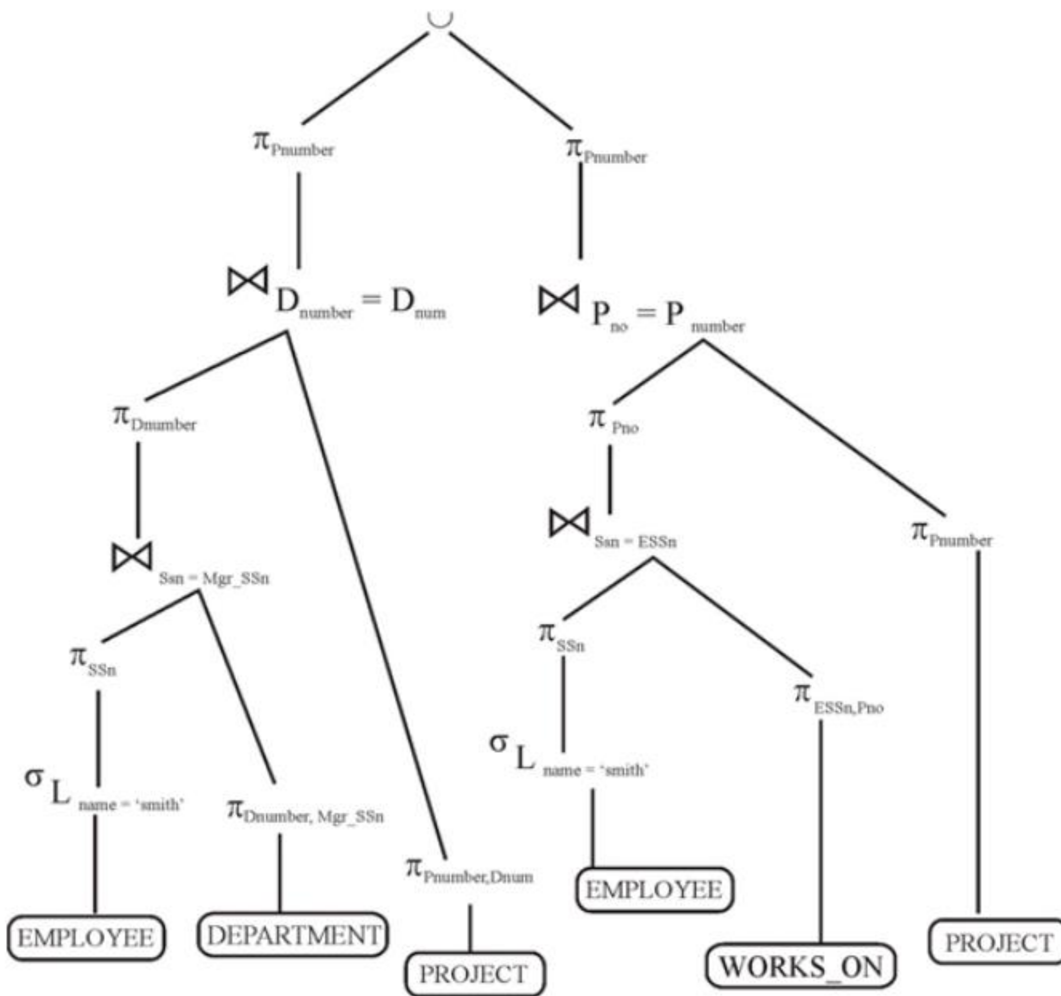


Now replace the CARTESIAN PRODUCT and SELECT operations with JOIN operations.

Then the query tree is generated like this.



So, final tree moving PROJECT operations down the query tree. Them the final tree is optioned like this.



(c)

In the above query trees, the fist tree in part (a) is not the final tree.

So, we need to move the PROJECT operations down and the query tree to optimize it to the final tree.

In the second tree in part (a) is not the final tree.

So, we need to move the SELECT operations down the query tree to optimize it to the final tree.

18.14. A file of 4,096 blocks is to be sorted with an available buffer space of 64 blocks. How many passes will be needed in the merge phase of the external sort-merge algorithm?

Sort – merge strategy:-

Stats by sorting small sub files (runs) of the main file and then merges the sorted runs. Creating larger sorted sub files that are merged in turn.

Sorting phase $(nR) = ?(b/nB)$

Merging phase $(dm) = \min(nB - 1, nB)$

$(np) = ?(\log dM(nB))$

Here

nR: number of initial runs

b: number of file blocks.

nB: available buffer space

dM: degree of merging.

Now, take the data given in the solution, and we need to find out the merge phase

So, given data is

Number of file blocks b = 4096

Available buffer space nB = 64.

So,

Sorting phase nR = b/nB

$$= \frac{4096}{64} = 64$$

Now we need to find out the dM value.

$$dM = \min(nB - 1, nB)$$

$$\Rightarrow dM = \min(64 - 1, 64)$$

$$= \min(63, 64)$$

dM is the smaller of $(nB - 1)$ and nR.

So, we take the dM value as 63.

By this we need to find out the number of passes.

So, $nP = \log_{dM} nR$

$$= \log_{63} 64$$

$$= \frac{\log_{10} 64}{\log_{10} 63}$$

$$= 1.004$$

18.15. Can a nondense index be used in the implementation of an aggregate operator? Why or why not? Illustrate with an example.

Implementation of aggregate operators using non-dense index:

• Non-dense index has the pointer that points to entire block of records instead of each record. The aggregate operators MAX, AVERAGE, SUM, and COUNT cannot be implemented with non-dense index, because all the data values are not visible in the index.

• The non-dense index can be used to implement MIN function, if in the data block, index key pointer point to smallest key value.

• The dense index can be used to implement AVG and SUM aggregate functions, because in dense index, every record has an associated index entry.

• In non-dense index when GROUP BY clause is used in query the aggregate operator must be applied separately on every group of records, and sorting or hashing is used on group attribute to partition the file into appropriate groups, then aggregate operator is applied to each group. This computation is complex.

• Consider the following query:

SELECT Branch, AVG(cost)

FROM PRODUCT;

WHERE category="Clothing"

Non-dense index cannot be used to implement MIN in the above query. Queries like above involve partitioning the file by using hashing or sorting on the grouping attribute.

In the above query, all the **PRODUCT** tuples corresponding to the category C**lothing** would be aggregated in several groups and average cost would be calculated for each group.

18.16. Extend the sort-merge join algorithm to implement the LEFT OUTER JOIN operation.

Sort – merge join algorithm that implements the LEFT OUTER JOIN operation.

Let the relation $R \rhd\lhd_{A=B} S$

Assume R has n records and

S has m records

Algorithm:-

Sort the records in R on attribute A;

Sort the records in S on attribute B;

Set $i = 1$,

$j = 1$;

While $(i <= n)$ and $(j <= m)$

do {

if $R(i)[A] > S(j)[B]$ then

set $j = j + 1$;

else if $R(i)[A] < S(j)[B]$ then

{

//* records on R does not have an associated records on S. *//

Out put the record < $R(i)$, null for all attributes of

$S>$ to T;

Set $i=i+1;$

}

//* output other records that match $S(j)$. If any *//

Set $k=i+1;$

While $(k<=n)$ and $(R(K)[A]=S(j)[B])$

do

{

Output the combined record $<R(K),S(j)>$ to T ;

Set $K=K+1;$

}

Set $i=K, j=1;$

}

}

//* add remaining records of R when there is no more records on s *//

While $(i<=n)$

do

{

Output the record $<R(i)$, null for all attributes of

$S>$ to T;

Set $i=i+1;$

}.

Review Questions

19.1. What is a query execution plan?

Query execution plane:-

It is a special case of the relational model concept of access plan.

Query execution plan is a set of steps, that are used to access or modify the information in a SQL relational database management system.

An query execution plan is a combination of

→ Relational algebra query representation, and

→ The information about the access methods to be used for each relation.

In the relational algebra, the query representing is in the form of tree.

Here the tree nodes are in the order.

In a each relation, the information about the access methods that are computing and

relational operators are stored in the tree.

In this execution plans, many algorithms are to do joins, sorting, selection etc.

→ Query execution plan follows the searching algorithms.

That are linear (or) binary.

Joins that are used nested loop join.

Let R JOLNS S

For each t_1 in R do

For each t_s in S do

Test pair $(t \text{ is } t_s)$ to see it they can be joined.

If so, add to result

.

Let an example for query execution plan.

Select salary

From employee

Where salary < 25000.

$\Rightarrow \pi$ salary

$\downarrow$

$\sigma_{\text{salary} < 25000}$

$\downarrow$ herre use secondary index on salary ; use binary search.

EMPLOYEE

19.2. What is meant by the term *heuristic optimization*? Discuss the main heuristics that are applied during query optimization.

Heuristic optimization:

Heuristic optimization is a rules – based method of producing. It is an efficient query execution plan. Because the query output of the standardization. Process is Q represented as a canonical query tree, in this each node o the tree maps directly to a relational algebraic expression.

The function of heuristic query optimizer is to apply algebraic rules of equivalence. To this expression tree and transform it into a more efficient representation.

During the optimization the main heuristic is to

→ First apply the operations that reduce the size of intermediate results.

Ex:

(1) Apply SELECT and PROJECT operations before applying the JOIN or other binary operations.

→ We use the algorithm for general heuristic optimization that

1 – Push selection down.

2 – Apply more restrictive selections first

3 – Combine cross products and selections to become join.

4 – Push projections down.

19.3. How does a query tree represent a relational algebra expression? What is meant by an execution of a query tree? Discuss the rules for transformation of query trees, and identify when each rule should be applied during optimization.

Execution of a query tree:-

A query tree is a tree data structure that represents the input relations of the query as leaf node and the relational algebra operations as internal nodes.

Execution may follow below steps.

Step (1)

Execute an internal node operation when ever its operands are available and then replace the internal node by the resulting operation.

Step (2)

Repeat step 1 as long as there are leaves in the tree that is thej execution terminates the root node is executed and produce the result relation for the query.

Transformation rules of a query tree:-

Using the transformation rules we may restructure the tree to improve performance transformation rules. That are useful in query optimization.

(1) Case code of σ:-

If we want to break the individual σ operation the operation then we use the case code of σ it can be broken up into a sequence.

$$\sigma_{C_1} \text{ AND } C_2 \text{AND} \ldots\ldots\ldots\ldots\text{AND} C_n(R) = \sigma_{C_1}\left(\sigma_{C_2}\left(\ldots\ldots\left(\sigma_{C_n}(R)\right)\ldots\ldots\ldots\right)\right)$$

(2) Commutatively of σ:-

The σ operation is commutative

$$\sigma_{C_1}\left(\sigma_{C_2}(R)\right) \equiv \sigma_{C_2}\left(\sigma_{C_1}(R)\right)$$

(3) Cascade of π

In a sequence of π operations, if we want to ignore the last one then we use this rule.

$$\pi_{list}\left(\pi_{list}\left(\ldots\ldots\left(\pi_{list}(R)\right)\right)\right) \equiv \pi_{list}(R)$$

(4) Commuting σ **with** π

If the selection condition 'C' involves only those attributes $A_1, \ldots\ldots\ldots\ldots, A_n$ in the projection list, then the two operations can be commuted

$$\pi_{A_1, A_2, \ldots\ldots\ldots\ldots A_n}\left(\sigma_C(R)\right) \equiv \sigma_C\left(\pi_{A_1, A_2 \ldots\ldots\ldots\ldots A_n(R)}\right)$$

(5) Commutatively of $\triangleright\triangleleft\left(\textbf{and } X\right)$

It is used for commutative as is the jX operation

$$R \triangleright\triangleleft_C S \equiv S \triangleright\triangleleft_C R$$
$$R \times S \equiv S \times R$$

(6) Commuting $\sigma\,\textbf{with} \triangleright\triangleleft \text{ or}\left(X\right)$

The attributes in the selection condition C involve only the attributes of one of the relations being joined – say, R – the two operations can be commuted as follows.

$$\sigma_C\left(R \triangleright\triangleleft S\right) \equiv \left(\sigma_C\left(R\right)\right) \triangleright\triangleleft S$$

(7) Commuting π **with** $\triangleright\triangleleft\left(\text{or } X\right)$

Let the projection list

$$L = \left\{A_1, \ldots\ldots A_n, B_1 \ldots\ldots B_m\right\}$$

Where $A_1, \ldots\ldots A_n$ are attributes of R and $B_1 \ldots\ldots B_m$ are attributes of S.

Here the join condition 'C' involves only attributes in L and the two operations can be commuted as

$$\pi_1\left(R \triangleright\triangleleft_C S\right) \equiv \left(\pi_{A_1 \ldots\ldots\ldots A_n}\left(R\right)\right) \triangleright\triangleleft_C \left(\pi_{B_1 \ldots\ldots B_m}\left(S\right)\right)$$

(8) Commutative of set operation:

In this true the set operations $\bigcup$ **and** $\bigcap$ that are used for commutative.

(9) Associatively of $\bowtie, X, \cup$ and $\cap$

There operations are individually associative means

$$(R\theta S)\theta T \equiv R\theta(S\theta T)$$

Here θ stans for any one of the above four operations.

(10) Commuting σ with set operations:-

$\sigma_{\text{operation commutes with } \cup, \cap \text{ and}}$ - means

$$\sigma_C(R\theta S) \equiv (\sigma_C(R))\theta(\sigma_C(S))$$

Here θ stans for any one of the above operations.

(11) π operation commutes with $\cup$

Means

$$\pi_1(R \cup S) \equiv (\pi_L(R)) \cup \pi_L(S)$$

(12) Converting $a(\sigma, X)$ sequence into $\bowtie$

The condition C of a σ that follows a X corresponds to a join condition convert the (σ, X)

Sequence into a $\bowtie$ as

$$(\sigma_C(R \times S)) \equiv (R \bowtie_C S)$$

Using the rules:-

$\rightarrow$ If we want to separate out unary operations to simplify tree

$\rightarrow$ Group unary operators on same relation

$\rightarrow$ Commute unary operators with binary operators and order binary operators

the we use the transformation rules.

19.4. How many different join orders are there for a query that joins 10 relations? How many left-deep trees are possible?

Join order:

n relations in a query block that are to be joined will have $n - 1$ join operations, and the corresponding join orders will be $n!$. This is the Cartesian product of all the numbers obtained through $n!$.

If there are 10 relations in a query block, then numbers of join orders are:

$$\begin{aligned} n! &= 10! \\ &= 10\times 9\times 8\times 7\times 6\times 5\times 4\times 3\times 2\times 1 \\ &= 3,628,800 \end{aligned}$$

Hence, the number of join orders for 10 relations are $\boxed{3,628,800}$.

Left-deep tree:

The left-deep tree is a binary tree such that every non-leaf node comprises of a right child in the form of base relation.

For $n = 10$ relations, the possible number of left-deep trees are

$$\begin{aligned} 10! &= 10\times 9\times 8\times 7\times 6\times 5\times 4\times 3\times 2\times 1 \\ &= 3,628,800 \end{aligned}$$

Hence, the number of left-deep trees for 10 relations are $\boxed{3,628,800}$.

19.5. What is meant by *cost-based query optimization*?

Cost – based query optimization.

Cost – based query optimization is a good declarative query optimizer. It does not depend solely on heuristic rules.

It should estimate the query execution plan which has the lowest cost estimate and heuristic rules that are applied to a query on number of alternative ways to execute it. These alter native ways relay on the execution of different auxiliary data structures and algorithms.

Query optimizer estimates the cost of executing each of alternative ways and choose the cheapest one.

Cost – components of query execution:

The cost of executing a query includes

→ Secondary storage access cost

In this data blocks are reading during the data searching.

Writing data block and cost of storing intermediate files are done.

→ Computation cost means – CPU cost

→ Memory cost – buffer cost.

→ Communication cost.

19.6. What is the optimization approach based on dynamic programming? How is it used during query optimization?

The optimization approach based on dynamic programming is greedy heuristic approach. In this approach problems are divided into subproblems that further can be divided into subproblems and these subproblems are solved only once.

Using heuristic approach during query optimization:

• Heuristic approach is an optimization technique that breaks queries into subqueries and further these subqueries are broken into other sub queries to get an optimal solution.

• Consider the relations *r*1, *r*2, *r*3, *r*4, *r*5 on which 5-way join order needs to be performed.

• The 5-way join can be split as:

*r*1 ? *r*2 ? *r*3 ? *r*4 ? *r*5 = (*r*1 ? *r*2 ? *r*3) ? *r*4 ? *r*5

There are 6 options for evaluating r1? r2 ? r3. These 6 options can be combined with temp1, such that temp1 represents the 6 possible options obtained by the result of first join, and then considering the next join as follows:

(temp1 ? *r*4 ? *r*5)

• There would be 6 options for evaluating temp1 and corresponding to each option, there would be 6 options for evaluating the join (temp1 ? *r*4 ? *r*5). Hence, there would be $6*6=36$ possible solutions.

• Heuristic approach based on dynamic programming comes handy here, as it provides the optimal solution, by using the optimal plan for determining temp1. Hence, the possible number of options reduces to $6+6=12$, instead of $5!=120$ options in the nonheuristic approach.

19.7. What are the problems associated with keeping views materialized?

Problems associated with keeping the view materialized are as follows:

• Updating any materialized view involves three updates. Updating a view through immediate update involves huge overhead. Such change involves changing the base relations through insert, delete and modify operation.

Therefore, updating a view involves creating the entire view corresponding to any change in base table and is thus, costly.

• Refer to the view V2 in section 7.3. For example, hiring an employee or deleting an employee affects the rows corresponding to that department in the view. These updates are handled manually by the programs that are responsible for updating the views. However, there is no guarantee that all the views would be updated.

• When materialized view is used with pre-joined tables, then it requires keeping the tables current according to the view. As the materialized view constitutes many tables, so any changes in source tables requires to be updated in materialized view too.

• When materialized view is used as a multiple refresh group, then single refresh group can cause referential integrity problem.

19.8. What is the difference between *pipelining* and *materialization*?

Difference between pipelining and materialization

Materialization:

The output of one operation is stored in a temporary relation for processing by next.

Pipelining:

Pipeline results of one operation to another with out creating temporary relation is know as pipelining or on the fly processing.

In general, the pipeline is implemented as separate process of thread.

Pipelining can save on cost of creating temporary relations and reading results back in again.

19.9. Discuss the cost components for a cost function that is used to estimate query execution cost. Which cost components are used most often as the basis for cost functions?

Cost components:

For the query execution, the query includes the below cost components. That are

(1) Access cost to secondary storage:

This is the cost of searching for, reading and writing data blocks that reside on secondary storage, mainly on disk.

Cost of searching for records in a file depends on the type of access structures on that file

(2) Memory usage cost:

Number of memory buffers needed for the equerry.

(3) Storage cost:

Sorting any intermediate files that are generated by an execution strategy for the query.

(4) Communication cost:

Shipping the results from the data base site to the user's site.

(5) Computation cost:

Of performing in – memory operations on the date buffer's during the execution plan

In the large database, the main aim is, to minimizing the access cost to secondary storage and in terms of the number of block transfers between disk and main memory simply cost functions are ignore and

For the small databases, most of the data in the files involves in the query can be completely stored in memory.

It influence on minimizing the computation cost.

In this time, communication cost also minimized.

So, access cost to secondary storage,

Computation cost.

Communication cost.

These three cost components are used for most often as the basis for cost functions.

19.10. Discuss the different types of parameters that are used in cost functions. Where is this information kept?

<u>Cost function – parameters:</u>

Parameters that are include in the cost function is

(1) Number of records (tuples) (r)

(2) Record size (R)

(3) Number of blocks (b)

(4) Blocking factor (b r)

Based on these parameters, we may know the information about the size of the file.

The information may be stored in the place of DBMS catalog. Where it is accessed by the query optimization.

19.11. What are semi-join and anti-join? What are the join selectivity and join cardinality parameters associated with them? Provide appropriate formulas.

Semi join: These joins are used in un-nesting the query in sub queries. Theses join operations are used to simplify and optimize the nested query.

Semi join is used to avoid unnecessary matching of two tables on the join condition.

Consider the following syntax of Join selectivity and cardinality:

Syntax of Semi joins

SELECT COUNT (*) FROM table T1 WHERE T1.M IN (SELECT T2.N FROM table T2);

This is the nested query with connector IN. To simplify the operation, nested query will be removed and the result will be semi join statement:

SELECT COUNT (*) FROM table T1, T2 where T1.M S=T2.N;

Where notation "S=" represent the non-standard semi- join

Anti-join: when nested query is connected with NOT IN, NIT EXIT, ALL clauses. Anti-join is used for un-nesting of these queries.

Syntax of anti joins

SELECT COUNT (*) FROM table T1 WHERE T1.M IN (SELECT T2.N FROM table T2);

After un-nesting the anti-join statement is:

SELECT COUNT (*) FROM table T1, T2 where T1.M A=T2.N;

Where notation "A=" represent the non-standard anti- join

Consider the following Selectivity and cardinality formulas:

The semi-join selectivity formula is as follows.

Js=MIN (1, NDV (N, T2)/NDV (M, T1))

In the above formula Js is the semi-join selectivity and NDV is the number of distinct value.

The semi-join cardinality formula is as follows.

Jc=|T1|*js

Where T1, T2 are left and right relations

The anti-join selectivity formula is as follows:

js= 1-MIN (1, NDV (T2.y)/NDV(T1.x))

The anti-join cardinality formula is as follows:

jc=|T1|*js

19.12. List the cost functions for the SELECT and JOIN methods discussed in Sections19.4 and 19.5.

Cost functions for various SELECT methods:

The parameters used in cost function formulas for various SELECT algorithms are as follows:

C_{Si} : The cost for method Si in block access

r_X : Number of records in a relation *X*

b_X : Number of blocks present in a relation *X*

bfr_X : Blocking factor in a relation *X*

sl_A : Selectivity of an attribute *A* for a particular condition

sA : Selection cardinality of the selected attribute $(= sl_A * r)$

xA : Number of the index levels corresponding to attribute *A*

$b_{I1}A$: Number of first-level blocks of the index on attribute *A*

$\text{NDV}(A, X)$: Number of distinct values of attribute A in relation X

- **S1—Linear search or brute force approach:**

When the entire file blocks are to be searched for fetching all the records that satisfies the selection condition, the cost function is $C_{S1a} = b$.

When only half of the files blocks are to be searched, the cost function is $C_{S1b} = (b/2)$ in case the record is found.

If the record is not found, the cost function is $C_{S1b} = b$.

• **S2—Binary search:**

The cost function for this method is $C_{S2} = \log_2 b + \lceil (s / bfr) \rceil - 1$.

• **S3a—Using a primary index to retrieve a single record:**

The cost function for this method is one disk block extra than the number of index levels. Hence, the cost is $C_{S3a} = x + 1$.

• **S3b—Using a hash key to retrieve a single record:**

The cost function is $C_{S3b} = 1$, in the case of linear hashing and $C_{S3b} = 2$ for extendible hashing.

• **S4—Using an ordering index to retrieve multiple records:**

In case comparison condition, such as $>, <, >=$ or $<=$, are applicable on a key field and half the file records satisfies the condition, then the cost function is $C_{S4} = x + (b/2)$.

• **S5—Using a clustering index to retrieve a multiple record:**

If *s* represents the selection cardinality corresponding to the indexing attribute, then *s* records will satisfy the equality condition on indexing attribute. This implies that cost function is $C_{S5} = x + \lceil (s / bfr) \rceil$, as cluster of file blocks will contain $\lceil (s / bfr) \rceil$ file blocks.

• **S6—Using a Secondary (B^+ tree) index:**

• Provided an equality selection condition, for a secondary index present on a unique attribute, the cost is $x + 1$ disk block accesses.

• If *s* represents the selection cardinality corresponding to the indexing attribute, then *s* records will satisfy the equality condition for a secondary index present on a nonunique attribute. However, due to the possibility of the records being residing on a different disk block, the cost function is $C_{S6a} = x + 1 + s$.

• In case, half the file records satisfies the condition such as $>, <, >=$ or $<=$, then half the files records plus half of the first level index blocks are accessed. The cost function for this case is thus $C_{S6b} = x + (b_{I1} / 2) + (r / 2)$

• **S7— Conjunctive selection:**

• Either S1 or one of the methods from S2 to S6 can be used for determining the cost functions for this method.

• While using the latter methods, the records are fetched using one condition and then each record is verified in the main memory to see if it satisfies the rest of the conditions.

• **S8—Conjunctive selection using a composite index:**

Depending upon index type, the cost functions of this method is same as S3a, S5, or S6a.

Cost functions for various JOIN methods:

The parameters used in cost function formulas for various JOIN algorithms are as follows:

Join selectivity (*js*):

• Size of the file obtained after join operation is required for estimating the cost functions for JOIN methods. Join Selectivity is the ratio of the size of the file obtained after join operation to the size of Cartesian Product file.

js = |(*R* ?$_C$ *S*) | / | *R* × S | = | (*R* ?$_C$ *S*) | / (|*R*| * |*S* |)

Where |*R*| denotes the number of tuples of a relation *R*

• If join condition *c* is not present, then *js* = 1. If join condition is not satisfied by any tuple, then *js* = 0.

Join cardinality (*jc*):

The size of the resultant file obtained after join operation is known as join cardinality and is determined through query optimizer. Join cardinality is given as:

jc = | (*R*$_C$ *S*) | = *js* * |*R*| * |*S*|

• J1—Nested-loop join:

In order to determine the number of block accesses for this method, the cost function is as follows:

$$C_{J1} = b_R + (b_R * b_S) + ((js * |R| * |S|) / bfr_{RS})$$

In case, n_B memory buffer blocks are available for performing the join, the cost function becomes:

$$C_{J1} = b_R + (\lceil b_R / (n_B - 2) \rceil * b_S) + ((js * |R| * |S|) / bfr_{RS})$$

• J3-—Sort-merge join:

When the files are sorted on the basis of join attributes, then the cost function for this method is:

$$C_{J3a} = b_R + b_S + ((js * |R| * |S|) / bfr_{RS})$$

• J4—Partition-hash join:

The same hashing function *h* can be used for partitioning the files *R* and *S* into smaller files. Hence the cost function for this method becomes:

$$C_{J4} = 3 * (b_R + b_S) + ((js * |R| * |S|) / bfr_{RS})$$

19.13. What are the special features of query optimization in Oracle that we did not discuss in the chapter?

The special features of query optimization in Oracle are as follows:

• **Parallel serialization:** Multiple processors work in synchronization, when Oracle executes SQL statements in parallel. Oracle divides the necessary work, required to execute the SQL statements among several processors. By doing so Oracle is able to execute the statements quickly and this process is known as parallel serialization.

• The number of parallel processors used for executing multiple SQL statements depends on the hints used in the SQL statement and the default parallel degree.

• The actual degree of parallelism used for processing a query, not only depends on the default setting but also depends on other factors—such as system capabilities.

For some SQL statements, the default degree might be too small resulting in suboptimal result. But for other statements, it might be too large leading to the underperformance of over parallelized queries.

• The exact number of parallel processes required to execute SQL statements, can be determined using the initialization parameter called parallel_max_servers.

If this value is very high, then it might result in the poor performance of the queries and consequently, in the over allocation of parallel processes by the system.

• **Adaptive Plans:** Oracle uses a key driver component called optimizer, for the execution of SQL queries. The optimizer provides an execution plan after working with the available statistics.

The new optimizer comprises of **statistics collector** which determines if the current estimates vary by a large margin. The adaptive plan works in the following cases:

• **Join method:** During runtime, the optimizer in Oracle is able to flip the join methods on the basis of information provided by the **statistics collector**.

The collector suggests a sub-plan to the optimizer, after determining the appropriate join method. Now, the optimizer selects the best plan after reviewing both the plans.

• **Parallel distribution method:** Generally, the optimizer uses broadcast method for distributing the data among parallel processors. In Oracle 12c, the optimizer makes use of Hybrid Hash to distribute the data among parallel processors.

Optimizer selects the best method on the basis of buffering rows.

• **Adaptive Statistics:** When the object's statistics are not sufficient to select optimal execution plan, then adaptive statistics are used. In case, the query have complex predicates or stale stats, then the optimizer makes use of following approaches:

• **Dynamic statistics:** The dynamic sampling feature is now known as dynamic statistics in Oracle 12c. In the parameter OPTIMIZER_DYNAMIC_SAMPLING, optimizer is intelligent enough to select dynamic sampling and is also aware of the query complexity.

• **Automatic Re-optimization:** Automatic Re-optimization is the optimizer's capability to change the execution plans. In case the optimizer finds any cardinality mismatch, it can decide to re-optimize the plan.

19.14. What is meant by *semantic query optimization*? How does it differ from other query optimization techniques?

Semantic query optimization:

It is the one of approach to query optimization. It uses the constraints specified on the database schema in order to modify one query into another query. That is more efficient to execute.

We can show this by below example.

Let SQL query

SELECT. L NAME , M.L NAME

FROM EMPLOYEE EM

WHERE E.SUPERSSN = M. SSN AND E.SALARY > M. SALARY

Description:

We have the constraint on the database schema that stated that no employee can earn more than his or her direct supervisor. If the statement query optimizer checks for the existence of this constraint. It need not execute the query at all because it knows that the result of the query will be empty.

Differ from other query optimization techniques:

Other query optimization, that provides two different approaches to query optimization.

Rule – based query optimization:-

Cost – based query optimization:-

In the rule – based query optimization, the optimizer choose the execution plans based on heuristically ranked operations.

And

In the cost – based query optimization, the optimizer examines alternative access paths and operator algorithms and chooses the execution plan with lowest estimate cost.

The query cost is calculated based on the estimated usage of resources such as I/O CPU and memory needed.

Exercises

19.15. Develop cost functions for the PROJECT, UNION, INTERSECTION, SET DIFFERENCE, and CARTESIAN PRODUCT algorithms discussed in Section 19.4.

Take R and S are two relations that are stored in bR and bS disk blocks, respectively, and file resulting from the operation is stored in bRESULT disk blocks. So,

PROJECT operation:

if includes a key of R, then the cost is 2*bR since theread–in and write–out files have the same size, which is the size of R itself;

if does not include a key of R, then we must sort the intermediate result file before eliminating duplicates, which costs another scan.

So, the latter cost is 3*b R + k*b R *log 2 b R.

SET OPERATIONS (, DIFFERENCE, INTERSECTION):

According to the algorithms

where the usual sort and scan/merge of each table is done, the cost of any of these is:

k*[(bR *log 2 bR) + (bS *log 2 bS)] + bR + bS + bRESULT

CARTESIAN PRODUCT:

The join selectivity of Cartesian product is js = 1, and the typical way of doing it is the nested loop method, since there are no conditions to match.

First assume two memory buffers and nB memory buffers (file R is smaller and is used in the outer loop).

J1: C R X S = b R + (b R *b S) + (|R|*|S|)/bfr RS

J1': C R X S = b R + ceiling(b R /(n B - 1)) * b S)+ (|R|*|S|)/bfr RS ,

which is better than J1, per its middle term.

19.16. Develop cost functions for an algorithm that consists of two SELECTs, a JOIN, and a final PROJECT, in terms of the cost functions for the individual operations.

Cost functions:

Cost functions, are helpful because it ignores the computation time, storage cost and other factors.

Cost functions for SELECT :

The following cost functions are considerable in the SELECT operation.

S1. Linear search (brute force) approach:

The following selection condition must be used to retrieve the records by searching all the file blocks is:

$C_{S1a} = b$;

For the equality condition, if the record is found only the half of the blocks is searched on average. Hence, the calculation is $C_{S1a} = (b/2)$. Otherwise use the condition $C_{S1a} = b$.

S2. Binary search:

It uses the following condition for search access .$C_{s2} = \log_2 b + (s/\mathrm{bfr}) - 1$. In this condition, if the equality condition is a unique key attribute then it reduces to $\log_2 b$ because s = 1.

S3. Using the primary index (S3a) to retrieve single record:

It retrieves one disk block from each index level and additionally one disk block from data file. Thus the cost is additionally one than the number of index levels

$C_{s3a} = X + 1$;

S3b. Using the hash key (S3b) to retrieve the "single record":

It requires only one disk block to be accessed in most of the cases. Hence, the cost function is $C_{s3b} = 1$ for static or linear hashing; but two disk blocks for extendible hashing $C_{s3b} = 2$.

S4. Using an ordering index to retrieve multiple records:

This uses the key field with an ordering index, and calculates roughly the half of the file records. Thus cost function is:

$$C_{s4} = X + (b/2)$$

Although, it is a rough estimate, it produces the correct result on the average. But on some rare cases, the result is inaccurate. The most accurate result is possible only if distributed records are provided in the histogram.

S5. Using clustering index to retrieve multiple records:

This accesses one disk block at each index level . The equality condition for records with indexing attribute is:

$$C_{s5} = X + \left[(s/\mathrm{bfr}) \right]$$

S6. Using a secondary $(\mathrm{B}^{+} - \mathrm{tree})$ index:

If the secondary index on key with unique attribute then the cost is $C_{s6a} = X + 1;$ for disk block accesses. But for the non key attribute the records "s" must satisfy equality condition, where "s" is the selection cardinality. Thus cost function is $C_{s6} = X + 1 + s;$

For the comparison condition, such as $>, <, >=,$ or $<=,$ it takes half of the file records to satisfy condition and half of the first-level index blocks plus half the file records via index. Thus cost function is:

$$C_{s6} = X + (b_{l1}/2) + (r/2)$$

S7. Conjunctive selection:

Use either S_1 or one of above methods from S_2 to S_6 to solve conjunctive selection.

In the latter case, we use the condition to retrieve records and then check in memory buffer whether each retrieved record satisfies the condition in the conjunction.

If the multiple indexes present, then the search index produces a set of record pointers in main memory buffer.

S7. Conjunctive selection using composite index:

It uses the same function of S3a, S5, and S6a based on the type of index.

JOIN operation is time consuming operations while processing a query.

Cost functions for JOIN :-

The cost function can be estimated by the size (number of tuples) of the file that results after JOIN operation. The ratio between the size of resulting join file to size of Cartesian product file are applied with same input files then it is referred as join selectivity.

The algorithm that consider for join operation is $R \bowtie_{A=B} S$ .Here A,B are domain compatible attributes and R,S are general forms of join. Thus, the Cost functions for JOIN:

Join selectively (js)

$$js = |R \bowtie_c S| / |R \times S|$$

$$= |R \bowtie_c S| / (|R| \times |S|)$$

In the above function, no join condition so js = 1 ; but if the join is as same as Cartesian product and no tuples from the relations satisfy the condition C, then $Js = 0;$

Usually, $0 <= js <= 1;$ so total size of the result file after join operation is

$$|(R \bowtie_c S)| = js * |R| * |S|$$

Cost functions for JOIN operation:

J1. Nested – loop join:

Consider that "R" is used for outer loop. So, cost function evaluates the number of block accesses with the assumption of three memory buffers.

Let us assumed that the blocking factor is "bfr_{RS}" and hence, the join selectivity is:

$$Cj1 = bR + (bR * bS) + (CjS * |R| * |S|) / \text{bfrRS}$$

J2. Single – loop join:

The index exists for the join attribute "B" of "S" with index level $x_{B.}$ so, retrieve all records in "R" and match records to t from s that satisfy condition $t[B] = s[A]$.

In secondary index s_B is the selection cardinality. The cost function is

$$C_{J2a} = b_R + (|R| * (x_B + 1 + s_B)) + ((js * |R| * |S|)/bfr_{RS})$$

For a clustering index, S_B is the selection cardinality of B,

$$C_{J2b} = b_R + |(R| * (x_B + (s_B/bfr_B))) + ((js * |R| * |S|)/bfr_{RS})$$

For a primary index, the cost function is:

$$C_{J2c} = b_R + |(R| * (x_B + 1)) + ((js * |R| * |S|)/bfr_{RS}).$$

Cost function for PROJECT :

The cost of join order is evaluated as follows, the first join between PROJECT and DEPARTMENT. Both the join method and the input relations must be determined.

The PROJECT relation will have the selection operation before performing the join, so two options exist: table scan (linear search) or utilizing its PROJ_PLOC index.

19.17. Develop a pseudo-language-style algorithm for describing the dynamic programming procedure for join-order selection.

An algorithm that describes the dynamic programming procedure for join-order selection is as follows:

1. Procedure findoptimalsolution(A)

2. if (optimalsolution(A).cost)

3. return optimalsolution(A)

4. else findoptimalsolution(A)

5. for each non-empty subset A1 of A such that A1 A

6. Q1= findoptimalsolution(A1)

7. Q2= findoptimalsolution(A - A1)

8. B = optimal algorithm for joining the results of Q1 and Q2

9. cost = Q1.cost + Q2.cost + cost of B

10. if cost < optimalsolution(A).cost

11. optimalsolution(A).cost = cost

12. optimalsolution(A).plan = execute Q1.plan;

execute Q2.plan; "join results of Q1 and Q2 using B"

13. return optimalsolution(A)

Explanation:

• Initialize the algorithm, for finding the optimal cost out of the costs of the *n* relations using the Procedure findoptimalsolution(A), where A is the set of *n* relations.

• If the optimal cost is not infinity then it is returned as the optimal cost, otherwise optimal solution, which involves finding the optimal cost out of the all the costs of *n* relations, is computed.

• In the step 5, A1 is the subset of A.

• In the steps 6 and 7, Q1 and Q2 represent the query execution plans.

• In step 9, the cost of executing the relations present in the execution plan and cost of executing the algorithm B is determined.

• In step 10, if the cost determined in the step 9 is less than the optimal cost then it becomes the new optimal cost.

• The step 13 returns the optimal cost, after comparing and selecting the best cost from the cost of each of the *n* relations.

19.18. Calculate the cost functions for different options of executing the JOIN operation OP7 discussed in Section 19.4.

Cost functions for different options of exacting the JOIN operation.

Let the relation

DEPARTMENT

Let us consider the nested – loop approach

The number of buffer's available in main memory of implementing the join is 'nB'.

Take

The DEPARTMENT file consists of records.

Stored in disk blocks and that EMPLOYEE block consists of

recors. Stored in disk block.

In the nested – loop join, each block of EMPLOYEE is read once and the entire DEPARTMENT file is read once for each time

We read in blocks of employee file.

Total number of blocks accessed for

Outer file = bE

Number of times block of outer file are loaded

Total number of blocks accessed for inner file

So, from these two outer block and inner block, we get total number of block access.

i.e

$b_E + \left(\lceil b_E/(nB-2) * b_D \rceil\right)$

$= 2000 + \left(\lceil (2000/5) \rceil * 10\right)$

$= 6000$ block access.

In the other hand DEPARTMENT records are in the outer loop, then the total number of accent block is $b_D + \left(\lceil b_D/(n_B-2) \rceil\right) * b_E$

$= 10 + \left(\lceil 10/5 \rceil\right) * 2000$

$= 4010$ block accesses.

19.19. Develop formulas for the hybrid hash-join algorithm for calculating the size of the buffer for the first bucket. Develop more accurate cost estimation formulas for the algorithm.

Cost for Hybrid Hash Join will be 3* (bR + bS) + bRES- bR1S , bR is the number of blocks for file R and bS is blocks for file S. Cost of Partition Join - cost of accessing data of first partition from m partition for R and S is the cost of Hybrid hash join.

19.20. Estimate the cost of operations OP6 and OP7 using the formulas developed in Exercise 19.19.

Hybrid hash joining

• Hybrid hash joining is a combination of more than one techniques of cost estimation. It consists of two relations *R* and *S*.

• Relation *R* is partitioned into *k* buckets. Let *t* buckets $(R_1, R_2, R_3, \ldots, R_t)$ reside in the memory and (*k*-*t*) buckets $(R_{t+1}, R_{t+2}, R_{t+3}, \ldots, R_k)$ reside in disk. Similarly relation *S* is partitioned into *k* bucket. Let *t* buckets $(S_1, S_2, S_3, \ldots, S_t)$ reside in the memory and

(*k*-*t*) buckets $(S_{t+1}, S_{t+2}, S_{t+3}, \ldots, S_k)$ reside in disk.

• First *t* buckets of relations *R* and *S* are joined. Now join (*k*-*t*) pair of buckets as follows:

$(R_{t+1}, S_{t+1}), (R_{t+2}, S_{t+2}), (R_{t+3}, S_{t+3}), \ldots, (R_k, S_k)$

• Choosing values of *t* and *k* plays a very important role in the cost estimation. Value of *k* must be as large as possible but smaller than or equal to *M*, that is $(k \leq M)$, where $M^2 \geq \min(b_R, b_S)$. Values b_R and b_S are the number of blocks in the relations *R* and *S* respectively. Value of t can calculated from the following equation:

$$\frac{t}{k} * b_S + k - t \leq M$$

Since $\frac{t}{k} * b_S >> (k - t)$ ignore $(k - t)$. Hence $\frac{t}{k} = \frac{M}{b_S}$

• Cost of partitioning hash join is:

$$C_{RS} = 3(b_R + b_S) + \frac{(js * |R| * |S|)}{bfr_{RS}}$$

Where $\frac{(js * |R| * |S|)}{bfr_{RS}}$ is the cost of writing to the disk.

The variable *js* represents joining selectivity, |*R*| represents number of tuples in the relation *R*, |S| represents number of tuples in the relation *S* and bfr_{RS} represents blocking the factor of joining.

• Hybrid hash joining algorithm uses more than one technique. The *t* pairs are joined using partition hash joining and remaining (*k*-*t*) pairs are joined using other techniques like sort merge joining. It saves the I/O cost that is approximately,

$$\frac{2t}{k} * (b_R + b_S)$$

• So the final cost for hybrid hash join algorithm is as follows:

$$C_{RS} = 3(b_R + b_S) + \frac{(js * |R| * |S|)}{bfr_{RS}} - \frac{2t}{k} * (b_R + b_S)$$

$$C_{RS} = \frac{(3k - 2t)}{k} * (b_R + b_S) + \frac{(js * |R| * |S|)}{bfr_{RS}}$$

Refer article 19.5.2 for OP6.

OP6 : EMPLOYEE $\bowtie_{\text{Dno=Dnumber}}$ DEPARTMENT

• Assume a primary index on Dnumber of DEPARTMENT with level 1, having $x_{\text{Dnumber}} = 1$. Assume joining selectivity for OP6 is $js_{\text{OP6}} = 1/|\text{DEPARTMENT}| = 1/125$ as Dnumber is a key of DEPARTMENT. Assume that OP6 has a blocking factor of $bfr_{ED} = 4$ records per block.

Also, OP6 has $b_E = 2000 \text{ disk blocks}, b_D = 13 \text{ disk blocks}, E = 10,000 \text{ records}$ and $D = 125 \text{ records}$.

• Hence the worst case costs for OP6 using hybrid hash join is as follows:

$$C_{ED} = \frac{(3k-2t)}{k} * (b_E + b_D) + \frac{(js * |E| * |D|)}{bfr_{ED}}$$

$$C_{ED} = \left(3 - \frac{2t}{k}\right) * (b_E + b_D) + \frac{(js * |E| * |D|)}{bfr_{ED}}$$

Also $M^2 \geq \min(b_E, b_D) \Rightarrow M \geq \min(2000, 13) \Rightarrow M = 4$ and thus, $\frac{t}{k} = \frac{M}{b_D} = \frac{4}{13}$.

• Substitute the values in the above formula and calculate the cost of OP6 as follows:

$$C_{ED} = \left(3 - \frac{2t}{k}\right) * (b_E + b_D) + \frac{(js_{\text{OP6}} * |E| * |D|)}{bfr_{ED}}$$

$$C_{ED} = \left(3 - \left(2 * \frac{4}{13}\right)\right) * (2000 + 13) + \frac{\left(\frac{1}{125} * 10000 * 125\right)}{4}$$

$$C_{ED} = 7300$$

Hence, the cost of OP6 is $\boxed{C_{ED} = 7300}$.

Refer article 19.5.2 for OP7.

$$\textbf{OP7 : DEPARTMENT} \bowtie_{\textbf{Mge_ssn=Ssn}} \textbf{EMPLOYEE}$$

• Assume a primary index on Ssn of EMPLOYEE with level 2, having $x_{Mgr_ssn} = 2$ and a secondary index on Mgr_ssn of DEPARTMENT having selection cardinality $(s_{\text{Mgr_ssn}} = 1)$. Assume join selectivity for OP7 is $js_{\text{OP7}} = 1/|\text{EMPLOYEE}|$ because Ssn is a key of EMPLOYEE.

• Assume that OP7 has a blocking factor of $bfr_{ED} = 4$ records per block.

Also, OP7 has $b_E = 2000 \text{ disk blocks}, b_D = 13 \text{ disk blocks}, E = 10,000 \text{ records}$ and

$D = 125 \text{ records}$.

• Hence the worst case costs for OP7 using hybrid hash join is as follows:

$$C_{ED} = \left(3 - \frac{2t}{k}\right) * (b_E + b_D) + \frac{(js * |E| * |D|)}{bfr_{ED}}$$

Also $M^2 \geq \min(b_E, b_D) \Rightarrow M \geq \min(2000, 13) \Rightarrow M = 4$ and thus,

$$\frac{t}{k} = \frac{M}{b_E} = \frac{4}{2000} = \frac{1}{500}.$$

• Substitute the values in the above formula and calculate the cost of OP7 as follows:

$$C_{ED} = \left(3 - \frac{2t}{k}\right) * (b_E + b_D) + \frac{(js_{\text{OP7}} * |E| * |D|)}{bfr_{ED}}$$

$$C_{DE} = \left(3 - \left(2 * \frac{1}{500}\right)\right) * (13 + 2000) + \frac{\left(\frac{1}{10000} * 10000 * 125\right)}{4}$$

$$C_{DE} = 4831$$

Hence, the cost of OP7 is $\boxed{C_{ED} = 4831}$.

19.21. Compare the cost of two different query plans for the following query:

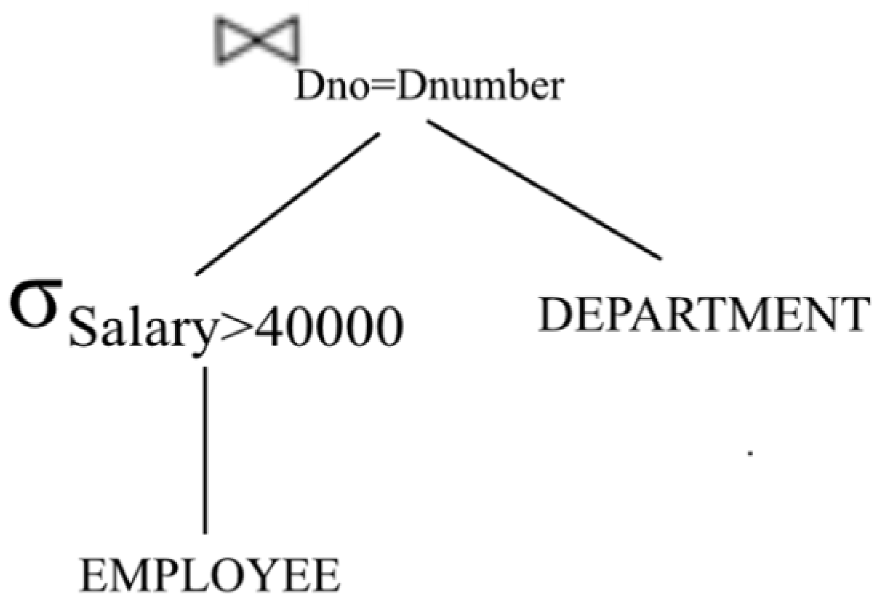
$$\sigma_{Salary < 40000}(\text{EMPLOYEE} \bowtie_{Dno=Dnumber} \text{DEPARTMENT})$$

Use the database statistics shown in Figure 19.6.

The given query is $\sigma_{Salary>40000}(\text{EMPLOYEE} \bowtie_{Dno=Dnumber} \text{DEPARTMENT})$.

Consider the data in figure 19.8 to calculate the cost of the query plan.

Query plan 1:



EMPLOYEE

In table EMPLOYEE, there are 500 unique values for Salary column with a low value of 1 and a high value of 500.

The low and high values are expressed in units. 1 unit=100 dollars.

$\sigma_{Salary>40000}$ can be estimated as $\frac{500-400}{500} = \frac{1}{5}$

With the value obtained, it can be considered that the salaries are spread evenly.

Cost of accessing the index $= \text{Blevel} + \frac{1}{5} \times (\text{LEAF BLOCKS})$

$= 1 + \frac{1}{5} \times 50$

$= 11$

Number of data blocks to be accessed = $\frac{1}{5}\times(NUM_ROWS)$

$= \frac{1}{5}\times 10000$

$= 2000$

10000 rows are stored in 2000 blocks. It means that 500 rows can be stored in each block.

The TEMPORARY table, which contains the result of selection operation, will require 400 blocks to store the resultant.

Hence, the cost of writing the TEMPORARY table to disk would be the cost of writing 400 blocks.

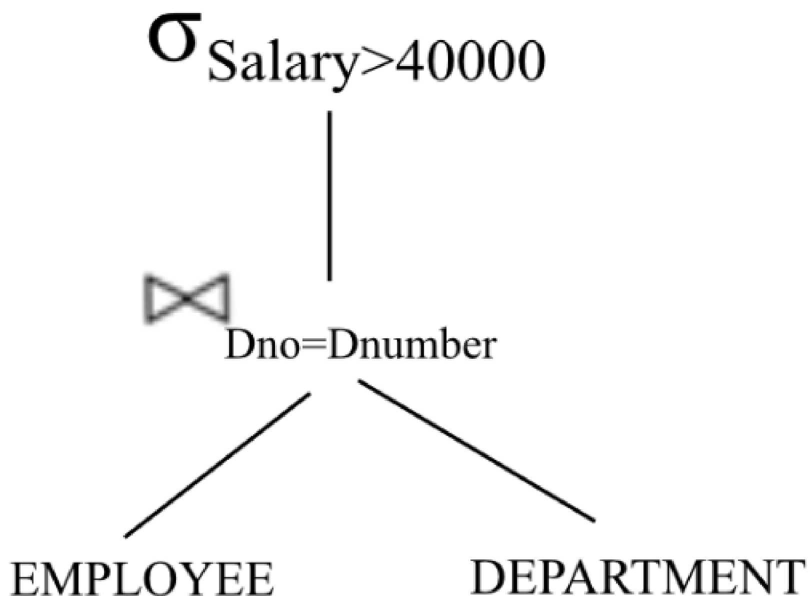
The cost of performing a nested loop joins of the temporary table and the DEPARTMENT table would be b + (b * b).

Hence, 5 + (5 * 400) = 2005 block accesses.

Therefore,

$$\begin{aligned}\text{Total cost} &= \text{Cost of accessing the index} + \text{Number of data blocks to be accessed} \\ &\quad + \text{Number of blocks for TEMPORARY table} + \text{Cost of performing a nested loop} \\ &= 11 + 2000 + 400 + 2005 \\ &= 4416 \text{ block accesses}\end{aligned}$$

Query plan 2:



It is possible to use nested loop for the join, which creates a temporary table. However, the better approach is to use a pipelining approach to generate the rows and use them as input to the select operation to get the result.

Using a nested loop join algorithm the query would yield the following:

Number of data blocks to be accessed=2000

Number of rows in DEPARTMENT=50

Number of leaf blocks=50

Hence, the total cost can be calculated as shown below:

$50 + (50\times 2000) = 100050$ block accesses