INDEX

Syllabus

Unit III : Implementing Indexes, Views and procedures (15 Lectures)
(a) File Organization and Indexing: Cluster, Primary and secondary indexing, Index data structure: hash and Tree based indexing, Comparison of file organization: cost model, Heap files, sorted files, clustered files. Creating, dropping and maintaining indexes using SQL.
(b) Views: Meaning of view, Data independence provided by views, creating, altering dropping, renaming and manipulating views using SQL.
(c) Stored Procedures: Types and benefits of stored procedures, creating stored procedures using SQL, executing stored procedures: Automatically executing stored procedures, altering stored procedures, viewing stored procedures.
(d) Triggers: Concept of triggers, Implementing triggers in SQL: creating triggers, Insert, delete, and update triggers, nested triggers, viewing, deleting and modifying triggers, and enforcing data integrity through triggers.

CHAPTER 1: FILE ORGANIZATION AND INDEXING

Topic covered:

- Cluster, Primary and secondary indexing, Index data structure: hash and Tree based indexing,
- Comparison of file organization: cost model, Heap files, sorted files, clustered files. Creating, dropping and maintaining indexes using SQL.

Q. Write a short note on Clustered Indexing?

- A clustered index determines the physical order of data in a table. A clustered index is analogous to a telephone directory, which arranges data by last name. Because the clustered index dictates the physical storage order of the data in the table, a table can contain only one clustered index. However, the index can comprise multiple columns (a composite index), like the way a telephone directory is organized by last name and first name.
- A clustered index is particularly efficient on columns that are often searched for ranges of values. After the row with the first value is found using the clustered index, rows with subsequent indexed values are guaranteed to be physically adjacent. For example, if an application frequently executes a query to retrieve records between a range of dates, a clustered index can quickly locate the row containing the beginning date, and then retrieve all adjacent rows in the table until the last date is reached. This can help increase the performance of this type of query. Also, if there is a column(s) that is used frequently to sort the data retrieved from a table, it can be advantageous to cluster (physically sort) the table on that column(s) to save the cost of a sort each time the column(s) is queried.
- Clustered indexes are also efficient for finding a specific row when the indexed value is unique. For example, the fastest way to find a particular employee using the unique employee ID column **emp_id** is to create a clustered index or PRIMARY KEY constraint on the **emp_id** column.
  - **Note**: PRIMARY KEY constraints create clustered indexes automatically if no clustered index already exists on the table and a nonclustered index is not specified when you create the PRIMARY KEY constraint.

Alternatively, a clustered index could be created on **lname**, **fname** (last name, first name), because employee records are often grouped and queried in this way rather than by employee ID.

- **Considerations**
  - It is important to define the clustered index key with as few columns as possible. If a large clustered index key is defined, any nonclustered indexes that are defined on the same table will be significantly larger because the nonclustered index entries contain the clustering key. The Index Tuning Wizard does not return an error when saving an SQL script to a disk with insufficient available space. For more information about how nonclustered indexes are implemented in Microsoft® SQL Server™ 2000,
  - The Index Tuning Wizard can consume significant CPU and memory resources during analysis. It is recommended that tuning should be performed against a test version of the production server rather than the production server. Additionally, the wizard should be run on a separate computer from the computer running SQL Server. The wizard cannot be used to select or create indexes and statistics in databases on SQL Server version 6.5 or earlier.

Before creating clustered indexes, understand how your data will be accessed. Consider using a clustered index for:

- Columns that contain a large number of distinct values.
- Queries that return a range of values using operators such as BETWEEN, >, >=, <, and <=.

- Columns that are accessed sequentially.
- Queries that return large result sets.
- Columns that are frequently accessed by queries involving join or GROUP BY clauses; typically these are foreign key columns. An index on the column(s) specified in the ORDER BY or GROUP BY clause eliminates the need for SQL Server to sort the data because the rows are already sorted. This improves query performance.
- OLTP-type applications where very fast single row lookup is required, typically by means of the primary key. Create a clustered index on the primary key.

Clustered indexes are not a good choice for:

- Columns that undergo frequent changes

  This results in the entire row moving (because SQL Server must keep the data values of a row in physical order). This is an important consideration in high-volume transaction processing systems where data tends to be volatile.

- Wide keys

  The key values from the clustered index are used by all nonclustered indexes as lookup keys and therefore are stored in each nonclustered index leaf entry.

Q. Explain Primary Indexing?

➢ **Primary Index**

In this section, we assume that all files are ordered sequentially on some search key.
Such files, with a primary index on the search key, are called **index-sequential files**. They represent one of the oldest index schemes used in database systems. They are designed for applications that require both sequential processing of the entire file and random access to individual records.
Figure 12.1 shows a sequential file of *account* records taken from our banking example.
In the example of Figure 12.1, the records are stored in search-key order, with *branch-name* used as the search key.

Dense and Sparse Indices

An **index record**, or **index entry**, consists of a search-key value, and pointers to one or more records with that value as their search-key value. The pointer to a record consists of the identifier of a disk block and an offset within the disk block to identify the record within the block.
There are two types of ordered indices that we can use:
• **Dense index**: An index record appears for every search-key value in the file.
In a dense primary index, the index record contains the search-key value and a pointer to the first data record with that search-key value. The rest of the records with the same search key-value would be stored sequentially after the first record, since, because the index is a primary one, records are sorted on the same search key.
Dense index implementations may store a list of pointers to all records with the same search-key value; doing so is not essential for primary indices.
• **Sparse index**: An index record appears for only some of the search-key values.
As is true in dense indices, each index record contains a search-key value and a pointer to the first data record with that search-key value. To locate a record, we find the index entry with the largest search-key value that is less than or equal to the search-key value for which we are looking. We start at the record pointed to by that index entry, and follow the pointers in the file until we find the desired record.

Figures 12.2 and 12.3 show dense and sparse indices, respectively, for the *account* file. Suppose that we are looking up records for the Perryridge branch. Using the dense index of Figure 12.2, we follow the pointer directly to the first Perryridge record. We process this record, and follow the pointer in that record to locate the next record in search-key (*branch-name*) order. We continue processing records until we encounter a record for a branch other than Perryridge. If we are using the sparse index (Figure 12.3), we do not find an index entry for "Perryridge." Since the last entry (in alphabetic order) before "Perryridge" is "Mianus," we follow that pointer. We then read the *account* file in sequential order until we find the first Perryridge record, and begin processing at that point.

As we have seen, it is generally faster to locate a record if we have a dense index rather than a sparse index. However, sparse indices have advantages over dense indices in that they require less space and they impose less maintenance overhead for insertions and deletions.

There is a trade-off that the system designer must make between access time and space overhead. Although the decision regarding this trade-off depends on the specific application, a good compromise is to have a sparse index with one index entry per block.

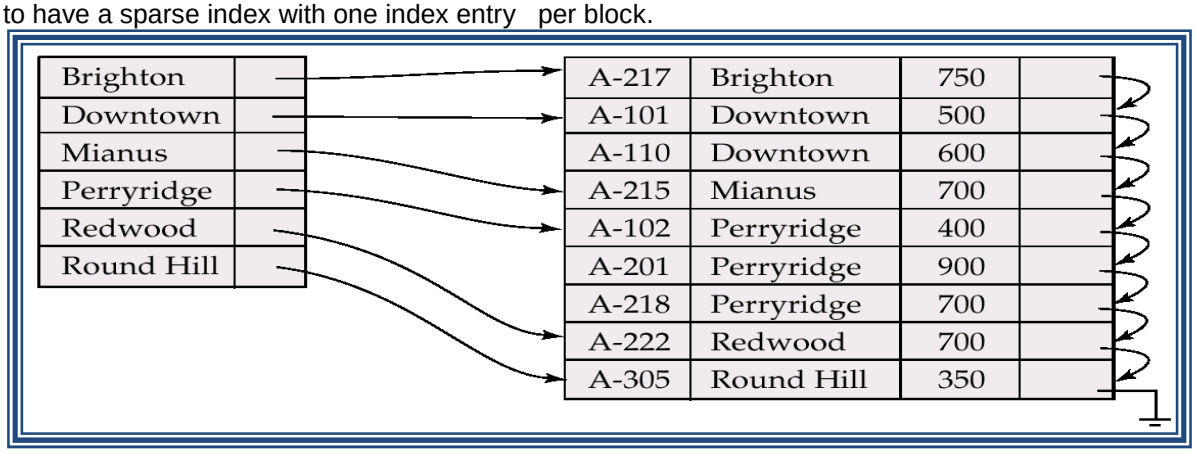


Fig 12.2 **Dense index**

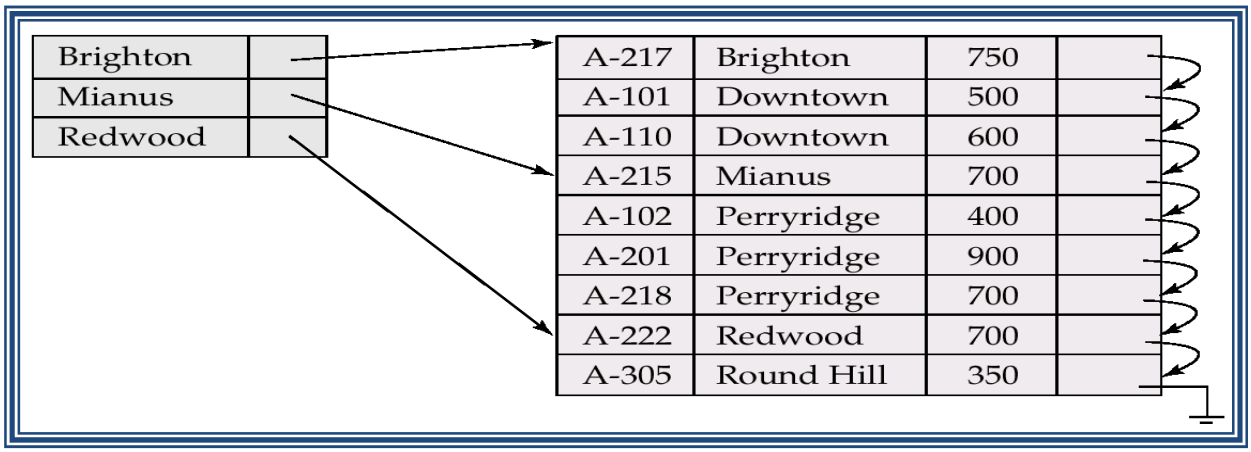


Fig 12.3 Sparse Index

The reason this design is a good trade-off is that the dominant cost in pro-cessing a database request is the time that it takes to bring a block from disk into main memory. Once we have brought in the block, the time to scan the entire block is negligible. Using this sparse index, we locate the block containing the record that we are seeking. Thus, unless the record is on an overflow block (see Section 11.7.1), we minimize block accesses while keeping the size of the index (and thus, our space overhead) as small as possible. For the preceding technique to be fully general, we must consider the case where records for one search-key value occupy several blocks. It is easy to modify our scheme to handle this situation.

Q. Explain Secondary indexes?

➢ **Secondary Indexes**

- A **secondary index** is also an ordered file with two fields. The first field is of the same data type as some *nonordering field* of the data file that is an **indexing field.** The second field is either a *block* pointer or a *record* pointer. There can be *many* secondary indexes (and hence, indexing fields) for the same file.
- We first consider a secondary index access structure on a key field that has a *distinct value* for every record. Such a field is sometimes called a **secondary key.** In this case there is one index entry for *each record* in the data file, which contains the value of the secondary key for the record and a pointer either to the block in which the record is stored or to the record itself. Hence, such an index is **dense.**
- We again refer to the two field values of index entry i as <K(i), P(i)>. The entries are ordered by value of K(i), so we can perform a binary search. Because the records of the data file are not physically ordered by values of the secondary key field, we cannot use block anchors. That is why an index entry is created for each record in the data file, rather than for each block, as in the case of a primary index. Figure 06.04 illustrates a secondary index in which the pointers P(i) in the index entries are block pointers, not record pointers. Once the appropriate block is transferred to main memory, a search for the desired record within the block can be carried out.
- A secondary index usually needs more storage space and longer search time than does a primary index, because of its larger number of entries. However, the improvement in search time for an arbitrary record is much greater for a secondary index than for a primary index, since we would have to do a linear search on the data file if the secondary index did not exist. For a primary index, we could still use a binary search on the main file, even if the index did not exist. Example 2 illustrates the improvement in number of blocks accessed.
- **EXAMPLE 2**: Consider the file of Example 1 with r = 30,000 fixed-length records of size R = 100 bytes stored on a disk with block size B = 1024 bytes. The file has b = 3000 blocks, as calculated in Example 1. To do a linear search on the file, we would require b/2 = 3000/2 = 1500 block accesses on the average. Suppose that we construct a secondary index on a nonordering key field of the file that is V = 9 bytes long. As in Example 1, a block pointer is P = 6 bytes long, so each index entry is Ri = (9 + 6) = 15 bytes, and the blocking factor for the index is bfri = (B/Ri) = (1024/15) = 68 entries per block. In a dense secondary index such as this, the total number of index entries ri is equal to the number of records in the data file, which is 30,000. The number of blocks needed for the index is hence bi = (ri/bfri) = (30,000/68) = 442 blocks.
- A binary search on this secondary index needs (log2bi) = (log2442) = 9 block accesses. To search for a record using the index, we need an additional block access to the data file for a total of 9 + 1 = 10 block accesses—a vast improvement over the 1500 block accesses needed on the average for a linear search, but slightly worse than the seven block accesses required for the primary index.
- We can also create a secondary index on a nonkey field of a file. In this case, numerous records in the data file can have the same value for the indexing field. There are several options for implementing such an index:
- Option 1 is to include several index entries with the same K(i) value—one for each record. This would be a dense index.
- Option 2 is to have variable-length records for the index entries, with a repeating field for the pointer. We keep a list of pointers <P(i,1), ..., P(i,k)> in the index entry for K(i)—one pointer to each block that contains a record whose indexing field value equals K(i). In either option 1 or option 2, the binary search algorithm on the index must be modified appropriately.
- Option 3, which is more commonly used, is to keep the index entries themselves at a fixed length and have a single entry for each index field value but to create an extra level of indirection to handle the multiple pointers. In this nondense scheme, the pointer P(i) in index entry <K(i), P(i)> points to a block

of record pointers; each record pointer in that block points to one of the data file records with value K(i) for the indexing field. If some value K(i) occurs in too many records, so that their record pointers cannot fit in a single disk block, a cluster or linked list of blocks is used. This technique is illustrated in Figure 06.05. Retrieval via the index requires one or more additional block access because of the extra level, but the algorithms for searching the index and (more importantly) for inserting of new records in the data file are straightforward. In addition, retrievals on complex selection conditions may be handled by referring to the record pointers, without having to retrieve many unnecessary file records .

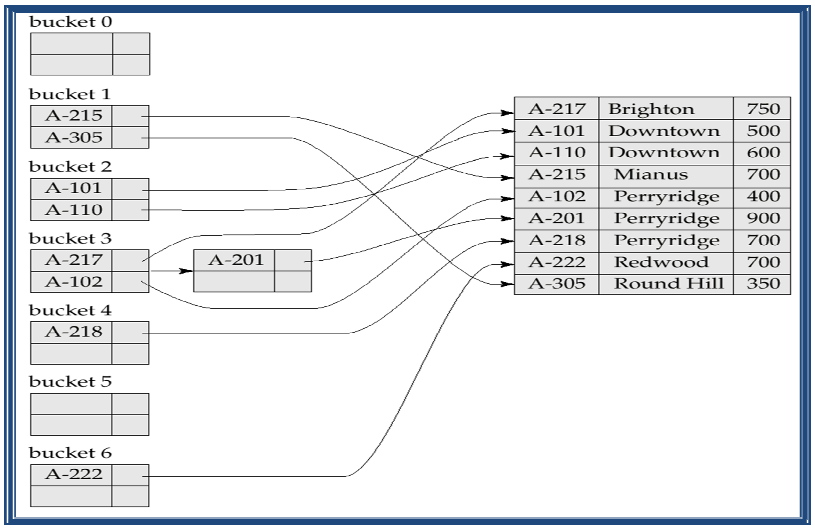
- Notice that a secondary index provides a logical ordering on the records by the indexing field. If we access the records in order of the entries in the secondary index, we get them in order of the indexing field.

Q. Explain hash based indexing?

Hash Indices

- Hashing can be used not only for file organization, but also for index-structure creation.
- A **hash index** organizes the search keys, with their associated record pointers, into a hash file structure.
- Strictly speaking, hash indices are always secondary indices
  - if the file itself is organized using hashing, a separate primary hash index on it using the same search-key is unnecessary.
  - However, we use the term hash index to refer to both secondary index structures and hash organized files.

Example of Hash Index



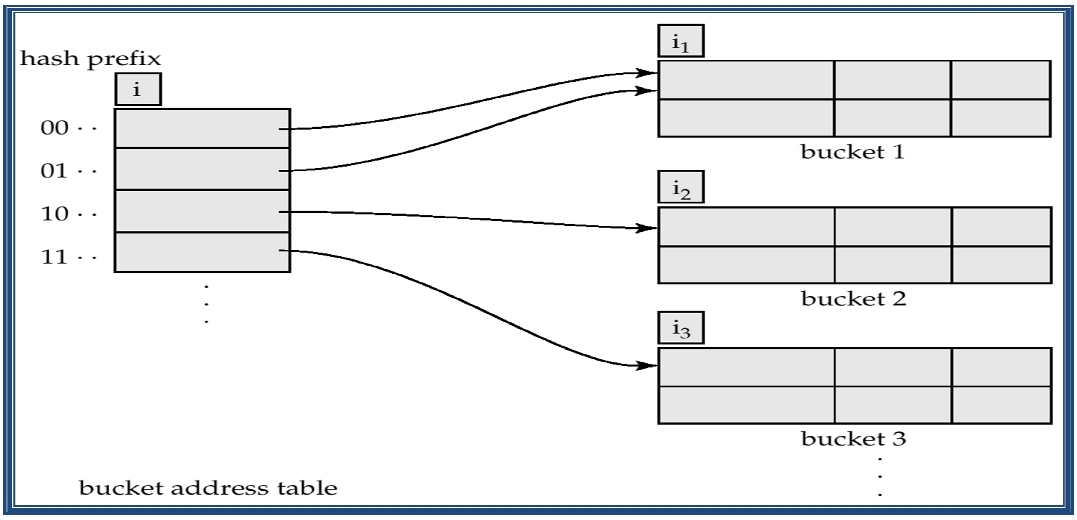
Deficiencies of Static Hashing

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

Dynamic Hashing

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

General Extendable Hash Structure



In this structure, $i_2 = i_3 = i$, whereas $i_1 = i - 1$ (see next slide for details)

Q. Explain Tree Based indexing?

Search Trees

A search tree is slightly different from a multilevel index. A search tree of order p is a tree such that each node contains at most p - 1 search values and p pointers in the order < P1, K1, P2, K2, ..., Pq-1, Kq-1, Pq >, where q 1 p; each Pi is a pointer to a child node (or a null pointer); and each Ki is a search value from some ordered set of values. All search values are assumed to be unique (Note 6). Figure 06.08 illustrates a node in a search tree. Two constraints must hold at all times on the search tree:

1. Within each node, K1 < K2 < ... < Kq-1.

2. For all values X in the subtree pointed at by Pi, we have Ki-1 < X < Ki for 1 < i < q; X < Ki for i = 1; and Ki-1 < X for i = q (see Figure 06.08).

Whenever we search for a value X, we follow the appropriate pointer Pi according to the formulas in condition 2 above. Figure 06.09 illustrates a search tree of order p = 3 and integer search values. Notice that some of the pointers Pi in a node may be null pointers.

We can use a search tree as a mechanism to search for records stored in a disk file. The values in the tree can be the values of one of the fields of the file, called the search field (which is the same as the index field if a multilevel index guides the search). Each key value in the tree is associated with a pointer to the record in the data file having that value. Alternatively, the pointer could be to the disk block containing that record. The search tree itself can be stored on disk by assigning each tree node to a disk block. When a new record is inserted, we must update the search tree by inserting an entry in the tree containing the search field value of the new record and a pointer to the new record.

Algorithms are necessary for inserting and deleting search values into and from the search tree while maintaining the preceding two constraints. In general, these algorithms do not guarantee that a search tree is balanced, meaning that all of its leaf nodes are at the same level (Note 7). The tree in Figure 06.07

is not balanced because it has leaf nodes at levels 1, 2, and 3. Keeping a search tree balanced is important because it guarantees that no nodes will be at very high levels and hence require many block accesses during a tree search. Another problem with search trees is that record deletion may leave some nodes in the tree nearly empty, thus wasting storage space and increasing the number of levels. The B-tree addresses both of these problems by specifying additional constraints on the search tree.

B-Trees

The B-tree has additional constraints that ensure that the tree is always balanced and that the space wasted by deletion, if any, never becomes excessive. The algorithms for insertion and deletion, though, become more complex in order to maintain these constraints. Nonetheless, most insertions and deletions are simple processes; they become complicated only under special circumstances—namely, whenever we attempt an insertion into a node that is already full or a deletion from a node that makes it less than half full. More formally, a B-tree of order p, when used as an access structure on a key field to search for records in a data file, can be defined as follows:

1. Each internal node in the B-tree (Figure 06.10a) is of the form

<P1, <K1, Pr1> , P2, <K2, Pr2> , ..., <Kq-1,Prq-1> , Pq>

where q 1 p. Each Pi is a tree pointer—a pointer to another node in the B-tree. Each Pri is a data pointer (Note 8)—a pointer to the record whose search key field value is equal to Ki (or to the data file block containing that record).

2. Within each node, K1 <K2 < ... < Kq-1.

3. For all search key field values X in the subtree pointed at by Pi (the ith subtree, see Figure 06.10a), we have:

Ki-1 < X < Ki for 1 < i < q; X < Ki for i = 1; and Ki-1 < X for i = q.

4. Each node has at most p tree pointers.

5. Each node, except the root and leaf nodes, has at least (p/2) tree pointers. The root node has at least two tree pointers unless it is the only node in the tree.

6. A node with q tree pointers, q 1 p, has q - 1 search key field values (and hence has q - 1 data pointers).

7. All leaf nodes are at the same level. Leaf nodes have the same structure as internal nodes except that all of their tree pointers Pi are null.

Figure 06.10(b) illustrates a B-tree of order p = 3. Notice that all search values K in the B-tree are unique because we assumed that the tree is used as an access structure on a key field. If we use a B-tree on a nonkey field, we must change the definition of the file pointers Pri to point to a block—or cluster of blocks—that contain the pointers to the file records. This extra level of indirection is similar to Option 3, discussed in Section 6.1.3, for secondary indexes.

A B-tree starts with a single root node (which is also a leaf node) at level 0 (zero). Once the root node is full with p - 1 search key values and we attempt to insert another entry in the tree, the root node splits into two nodes at level 1. Only the middle value is kept in the root node, and the rest of the values are split evenly between the other two nodes. When a nonroot node is full and a new entry is inserted into it, that node is split into two nodes at the same level, and the middle entry is moved to the parent node along

with two pointers to the new split nodes. If the parent node is full, it is also split. Splitting can propagate all the way to the root node, creating a new level if the root is split. We do not discuss algorithms for B-trees in detail here; rather, we outline search and insertion procedures for B+-trees in the next section.

If deletion of a value causes a node to be less than half full, it is combined with its neighboring nodes, and this can also propagate all the way to the root. Hence, deletion can reduce the number of tree levels. It has been shown by analysis and simulation that, after numerous random insertions and 1 deletions on a B-tree, the nodes are approximately 69 percent full when the number of values in the tree stabilizes. This is also true of B+-trees. If this happens, node splitting and combining will occur only rarely, so insertion and deletion become quite efficient. If the number of values grows, the tree will expand without a problem—although splitting of nodes may occur, so some insertions will take more time. Example 4 illustrates how we calculate the order p of a B-tree stored on disk.

EXAMPLE 4: Suppose the search field is V = 9 bytes long, the disk block size is B = 512 bytes, a record (data) pointer is Pr = 7 bytes, and a block pointer is P = 6 bytes. Each B-tree node can have at most p tree pointers, p - 1 data pointers, and p - 1 search key field values (see Figure 06.10a). These must fit into a single disk block if each B-tree node is to correspond to a disk block. Hence, we must have:

(p * P) + ((p - 1) * (Pr + V)) 1 B

(p * 6) + ((p - 1) * (7 + 9)) 1 512

(22 * p) 1 528

We can choose p to be a large value that satisfies the above inequality, which gives p = 23 (p = 24 is not chosen because of the reasons given next).

In general, a B-tree node may contain additional information needed by the algorithms that manipulate the tree, such as the number of entries q in the node and a pointer to the parent node. Hence, before we do the preceding calculation for p, we should reduce the block size by the amount of space needed for all such information. Next, we illustrate how to calculate the number of blocks and levels for a B-tree.

EXAMPLE 5: Suppose that the search field of Example 4 is a nonordering key field, and we construct a B-tree on this field. Assume that each node of the B-tree is 69 percent full. Each node, on the average, will have p * 0.69 = 23 * 0.69 or approximately 16 pointers and, hence, 15 search key field values. The average fan-out fo =16. We can start at the root and see how many values and pointers can exist, on the average, at each subsequent level:

Root:	1 node	15 entries	16 pointers
Level 1:	16 nodes	240 entries	256 pointers
Level 2:	256 nodes	3840 entries	4096 pointers
Level 3:	4096 nodes	61,440 entries	

At each level, we calculated the number of entries by multiplying the total number of pointers at the previous level by 15, the average number of entries in each node. Hence, for the given block size,

pointer size, and search key field size, a two-level B-tree holds 3840 + 240 + 15 = 4095 entries on the average; a three-level B-tree holds 65,535 entries on the average.

B-trees are sometimes used as primary file organizations. In this case, whole records are stored within the B-tree nodes rather than just the <search key, record pointer> entries. This works well for files with a relatively small number of records, and a small record size. Otherwise, the fan-out and the number of levels become too great to permit efficient access.

In summary, B-trees provide a multilevel access structure that is a balanced tree structure in which each node is at least half full. Each node in a B-tree of order p can have at most p-1 search values.

Q. Write a short note on cost model?

- **COST MODEL**
  - In this section we introduce a cost model that allows us to estimate the cost (in terms of execution time) of different database operations. We will use the following notation and assumptions in our analysis. There are *B* data pages with *R* records per page.
  - The average time to read or write a disk page is *D*, and the average time to process a record (e.g., to compare a field value to a selection constant) is *C*. In the hashed file organization, we will use a function, called a *hash function*, to map a record into a range of numbers; the time required to apply the hash function to a record is *H*.
  - Typical values today are *D* = 15 milliseconds, *C* and *H* = 100 nanoseconds; we therefore expect the cost of I/O to dominate. This conclusion is supported by current hardware trends, in which CPU speeds are steadily rising, whereas disk speeds are not increasing at a similar pace. On the other hand, as main memory sizes increase, a much larger fraction of the needed pages are likely to fit in memory, leading to fewer I/O requests.
  - We therefore use the number of disk page I/Os as our cost metric in this book. We emphasize that real systems must consider other aspects of cost, such as CPU costs (and transmission costs in a distributed database). However, our goal is primarily to present the underlying algorithms and to illustrate how costs can be estimated. Therefore, for simplicity, we have chosen to concentrate on only the I/O component of cost. Given the fact that I/O is often (even typically) the dominant component of the cost of database operations, considering I/O costs gives us a good first approximation to the true costs.
  - Even with our decision to focus on I/O costs, an accurate model would be too complex for our purposes of conveying the essential ideas in a simple way. We have therefore chosen to use a simplistic model in which we just count the number of pages that are read from or written to disk as a measure of I/O. We have ignored the important issue of **blocked access** typically, disk systems allow us to read a block of contiguous pages in a single I/O request.
  - The cost is equal to the time required to **seek** the first page in the block and to **transfer** all pages in the block. Such blocked access can be much cheaper than issuing one I/O request per page in the block, especially if these requests do not follow consecutively: We would have an additional seek cost for each page in the block.
  - This discussion of the cost metric we have chosen must be kept in mind when we discuss the cost of various algorithms in this chapter and in later chapters. We discuss the implications of the cost model whenever our simplifying assumptions are likely to affect the conclusions drawn from our analysis in an important way.

COMPARISON OF THREE FILE ORGANIZATIONS

We now compare the costs of some simple operations for three basic _le organizations: *files of randomly ordered records, or heap files*; *files sorted on a sequence of fields*; and *files that are hashed on a sequence of fields*. For sorted and hashed files, the sequence of fields (e.g., *salary*, *age*) on which the file is sorted or hashed is called the **search key**.

Note that the search key for an index can be any sequence of one or more fields; it need not uniquely identify records. We observe that there is an unfortunate overloading of the term *key* in the database literature. A *primary key* or *candidate key* (fields that uniquely identify a record; see Chapter 3) is unrelated to the concept of a search key. Our goal is to emphasize how important the choice of an appropriate file organization can be. The operations that we consider are described below.

Scan: Fetch all records in the file. The pages in the file must be fetched from disk into the buffer pool. There is also a CPU overhead per record for locating the record on the page (in the pool).

Search with equality selection: Fetch all records that satisfy an equality selection, for example, \Find the Students record for the student with *sid* 23." Pages that contain qualifying records must be fetched from disk, and qualifying records must be located within retrieved pages.

Search with range selection: Fetch all records that satisfy a range selection, for example, \Find all Students records with *name* alphabetically after `Smith.' "

Insert: Insert a given record into the file. We must identify the page in the file into which the new record must be inserted, fetch that page from disk, modify it to include the new record, and then write back the modified page. Depending on the file organization, we may have to fetch, modify, and write back other pages as well.

Delete: Delete a record that is specified using its rid. We must identify the page that contains the record, fetch it from disk, modify it, and write it back. Depending on the file organization, we may have to fetch, modify, and write back other pages as well.

Q. Explain Heap files and Sorted files?

- **Heap Files**
  - **Scan:** The cost is $B(D + RC)$ because we must retrieve each of B pages taking time
  - D per page, and for each page, process R records taking time C per record.
  - **Search with equality selection:** Suppose that we know in advance that exactly one record matches the desired equality selection, that is, the selection is specified on a candidate key. On average, we must scan half the file, assuming that the record exists and the distribution of values in the search field is uniform. For each retrieved data page, we must check all records on the page to see if it is the desired record. The cost is $0{:}5B(D + RC)$. If there is no record that satisfies the selection, however, we must
  - scan the entire file to verify this.
  - If the selection is not on a candidate key field (e.g., \Find students aged 18"), we always have to scan the entire file because several records with *age* = 18 could be dispersed all over the file, and we have no idea how many such records exist.
  - **Search with range selection:** The entire file must be scanned because qualifying records could appear anywhere in the file, and we do not know how many qualifying records exist. The cost is $B(D + RC)$.
  - **Insert:** We assume that records are always inserted at the end of the file. We must fetch the last page in the file, add the record, and write the page back. The cost is $2D + C$.
  - **Delete:** We must find the record, remove the record from the page, and write the modified page back. We assume that no attempt is made to compact the file to reclaim the free space created by deletions, for simplicity.1 The cost is the cost of searching plus $C + D$.
  - We assume that the record to be deleted is specified using the record id. Since the page id can easily be obtained from the record id, we can directly read in the page.
  - The cost of searching is therefore D.
  - If the record to be deleted is specified using an equality or range condition on some fields, the cost of searching is given in our discussion of equality and range selections.
  - The cost of deletion is also affected by the number of qualifying records, since all pages containing such records must be modified.
- **Sorted Files**
  - **Scan:** The cost is $B(D + RC)$ because all pages must be examined. Note that this case is no better or worse than the case of unordered files. However, the order in which records are retrieved corresponds to the sort order.

- **Search with equality selection:** We assume that the equality selection is specified on the field by which the file is sorted; if not, the cost is identical to that for a heap file. We can locate the first page containing the desired record or records, should any qualifying records exist, with a binary search in *log*2*B* steps. (This analysis assumes that the pages in the sorted file are stored sequentially, and we can retrieve the ith page on the file directly in one disk I/O. This assumption is not valid if, for example, the sorted file is implemented as a heap file using the linked-list organization, with pages in the appropriate sorted order.) Each step requires a disk I/O and two comparisons.
- Once the page is known, the first qualifying record can again be located by a binary search of the page at a cost of *Clog*2*R*. The cost is *Dlog*2*B* + *Clog*2*R*, which is a significant improvement over searching heap files.If there are several qualifying records (e.g., \Find all students aged 18"), they are guaranteed to be adjacent to each other due to the sorting on *age*, and so the cost of retrieving all such records is the cost of locating the first such record (*Dlog*2*B*+*Clog*2*R*) plus the cost of reading all the qualifying records in sequential order. Typically, all qualifying records fit on a single page. If there are no qualifying records, this is established by the search for the first qualifying record, which finds the page that would have contained a qualifying record, had one existed, and searches that page.
- **Search with range selection:** Again assuming that the range selection is on the sort field, the first record that satisfies the selection is located as it is for search with equality. Subsequently, data pages are sequentially retrieved until a record is found that does not satisfy the range selection; this is similar to an equality search with many qualifying records.
- The cost is the cost of search plus the cost of retrieving the set of records that satisfy the search. The cost of the search includes the cost of fetching the first page containing qualifying, or matching, records. For small range selections, all qualifying records appear on this page. For larger range selections, we have to fetch additional pages containing matching records.
- **Insert:** To insert a record while preserving the sort order, we must first find the correct position in the file, add the record, and then fetch and rewrite all subsequent pages (because all the old records will be shifted by one slot, assuming that the file has no empty slots). On average, we can assume that the inserted record belongs in the middle of the file. Thus, we must read the latter half of the file and then write it back after adding the new record. The cost is therefore the cost of searching to find the position of the new record plus 2 *fi* (0:5*B*(*D* + *RC*)), that is, search cost plus *B*(*D* + *RC*).
- **Delete:** We must search for the record, remove the record from the page, and write the modified page back. We must also read and write all subsequent pages because all records that follow the deleted record must be moved up to compact the free space.2 The cost is the same as for an insert, that is, search cost plus *B*(*D* + *RC*). Given the rid of the record to delete, we can fetch the page containing the record directly. If records to be deleted are specified by an equality or range condition, the cost of
- deletion depends on the number of qualifying records. If the condition is specified on the sort field, qualifying records are guaranteed to be contiguous due to the sorting, and the first qualifying record can be located using binary search.

Q. Write a short note on clustered files?

- A clustered file stores records in a tree structure. Clustered files are useful for applications that need to access records based on the value of a specific field or fields, and not on physical placement of the records. The primary index of a clustered file is based on any field or combination of fields specified by the user when the file is created. For example, the fields chosen for the primary index can be fields most frequently accessed by an application.
- The records in clustered files can have fixed-length or variable-length fields. When a record in a clustered file is updated, the new record can be any length, as long as it does not exceed the maximum record size specified when the file was created. The disk space allocated to a record is freed for use when the record is deleted or when the record update makes the record smaller. The SFS automatically reclaims the freed space.
- You can create a clustered file by using the **sfsadmin create clusteredfile** command.

- You can also create a clustered file by using the **create** command of CICSSDT.
- When you create a clustered file, you must specify the following:
  - The name of the new file
  - The number of fields per record and a description of each field
  - The name of the primary index and its key field or fields.
  - The name of the volume on which the file will be stored
- Note that you cannot change any of these characteristics except the name of the file after you have created it. See Specifying common required and optional arguments for SFS file types for details on specifying record fields.
- By default, the SFS organizes keys in ascending order and allows duplicate keys in the primary index of a clustered file. When you create a clustered file, you can optionally
- Specify that the key values are sorted in descending order.
- Specify that key values in the index are unique.

Index Definition in SQL

The SQL standard does not provide any way for the database user or administrator to control what indices are created and maintained in the database system. Indices are not required for correctness, since they are redundant data structures. However, indices are important for efficient processing of transactions, including both update transactions and queries. Indices are also important for efficient enforcement of integrity constraints. For example, typical implementations enforce a key declaration by creating an index with the declared key as the search key of the index.

In principle, a database system can decide automatically what indices to create.

However, because of the space cost of indices, as well as the effect of indices on update processing, it is not easy to automatically make the right choices about what indices to maintain. Therefore, most SQL implementations provide the programmer control over creation and removal of indices via data-definition-language commands.

We illustrate the syntax of these commands next. Although the syntax that we show is widely used and supported by many database systems, it is not part of the SQL:1999 standard. The SQL standards (up to SQL:1999, at least) do not support control of the physical database schema, and have restricted themselves to the logical database schema.

We create an index by the **create index** command, which takes the form

create index <index-name> **on** <relation-name> (<attribute-list>)

The *attribute-list* is the list of attributes of the relations that form the search key for the index.

To define an index name *b-index* on the *branch* relation with *branch-name* as the search key, we write

create index *b-index* **on** *branch* (*branch-name*)

If we wish to declare that the search key is a candidate key, we add the attribute **unique** to the index definition. Thus, the command

create unique index *b-index* **on** *branch* (*branch-name*)

declares *branch-name* to be a candidate key for *branch*. If, at the time we enter the **create unique index** command, *branch-name* is not a candidate key, the system will display an error message, and the attempt to create the index will fail. If the index creation attempt succeeds, any subsequent attempt to insert a tuple that violates the key declaration will fail. Note that the **unique** feature is redundant if the database system supports the **unique** declaration of the SQL standard.

Many database systems also provide a way to specify the type of index to be used (such as B+-tree or hashing). Some database systems also permit one of the indices on a relation to be declared to be clustered; the system then stores the relation sorted by the search-key of the clustered index.

The index name we specified for an index is required to drop an index. The **drop index** command takes the form:

drop index <index-name>

Quick Revision

- A *file organization* is a way of arranging records in a file. In our discussion of different file organizations, we use a simple cost model that uses the number of disk page I/Os as the cost metric.
- We compare three basic file organizations (*heap files, sorted file*s, and *hashed files*) using the following operations: scan, equality search, range search, insert, and delete. The choice of file organization can have a significant impact on performance.
- An *index* is a data structure that speeds up certain operations on a file.
- The operations involve a *search key*, which is a set of record fields (in most cases a single field).
- The elements of an index are called *data entries*. Data entries can be actual data records, *hsearch-key, ridi* pairs, or *hsearch-key, rid-listi* pairs.
- A given file of data records can have several indexes, each with a different search key.
- In a *clustered* index, the order of records in the file matches the order of data entries in the index. An index is called *dense* if there is at least one data entry per search key that appears in the file; otherwise the index is called *sparse*.
- An index is called a *primary index* if the search key includes the primary key; otherwise it is called a *secondary index*.
    - If a search key contains several fields it is called a *composite key*. SQL-92 does not include statements for management of index structures, and so there some variation in index-related commands across different DBMSs.

IMP Question Set

SR No.	Question	Reference page No.
1	Q. Write a short note on Clustered Indexing?	3
2	Q. Explain Primary Indexing?	4
3	Q. Explain Secondary indexes?	6
4	Q. Explain hash based indexing?	7
5	Q. Explain Tree Based indexing?	9
6	Q. Write a short note on cost model?	12
7	Q. Explain Heap files and Sorted files?	13
8	Q. Write a short note on clustered files?	14

CHAPTER 2: VIEWS

Topic Covered:

- Meaning of view, Data independence provided by views, creating, altering dropping, renaming and manipulating views using SQL.

Q. Write a short note on Views?

- **VIEWS**

- A **view** is a table whose rows are not explicitly stored in the database but are computed as needed from a **view definition**. Consider the Students and Enrolled relations.
- Suppose that we are often interested in finding the names and student identifiers of students who got a grade of B in some course, together with the cid for the course.
- We can define a view for this purpose. Using SQL-92 notation:

```
CREATE VIEW B-Students (name, sid, course)
AS SELECT S.sname, S.sid, E.cid
FROM Students S, Enrolled E
WHERE S.sid = E.sid AND E.grade = `B'
```

- The view B-Students has three fields called *name*, *sid*, and *course* with the same domains as the fields *sname* and *sid* in *Students* and *cid* in Enrolled. (If the optional arguments *name*, *sid*, and *course* are omitted from the CREATE VIEW statement, the column names *sname*, *sid*, and *cid* are inherited.)
- This view can be used just like a **base table**, or explicitly stored table, in defining new queries or views. Given the instances of Enrolled and Students shown in Figure 3.4, BStudents contains the tuples shown in Figure 3.18. Conceptually, whenever B-Students is used in a query, the view definition is first evaluated to obtain the corresponding instance of B-Students, and then the rest of the query is evaluated treating B-Students like any other relation referred to in the query. (We will discuss how queries on views are evaluated in practice in Chapter 23.)

name	*sid*	*course*
Jones	53666	History105
Guldu	53832	Reggae203

Figure 3.18 An Instance of the B-Students View

Q. Explain Data independence provided by views?

- **Data Independence**

- Consider the levels of abstraction that we discussed in Section 1.5.2. The *physical* schema for a relational database describes how the relations in the conceptual schema are stored, in terms of the file organizations and indexes used. The *conceptual* schema is the collection of schemas of the relations stored in the database. While some relations in the conceptual schema can also be exposed to applications, i.e., be part of the *external* schema of the database, additional relations in the *external* schema can be defined using the view mechanism.
- The view mechanism thus provides the support for *logical data independence* in the relational model. That is, it can be used to define relations in the external schema that mask changes in the conceptual schema of the database from applications. For example, if the schema of a stored relation is changed, we can define a view with the old schema, and applications that expect to see the old schema can now use this view.
- Views are also valuable in the context of *security*: We can define views that give a group of users access to just the information they are allowed to see. For example, we can define a view that allows students to see other students' name and age but not their gpa, and allow all students to access this view, but not the underlying Students table.

Q. Explain creating, altering dropping, renaming and manipulating views using SQL?

- a view is a **virtual** table

- how a view is defined:
- **CREATE VIEW** ATL-FLT
- **AS SELECT** FLT#, AIRLINE, PRICE
- **FROM** FLT-SCHEDULE
- **WHERE** FROM-AIRPORTCODE = “ATL”**;**

- how a query on a view is written:
  SELECT *
  FROM ATL-FLT
  WHERE PRICE <= 00200.00;
- how a query on a view is computed:
  SELECT FLT#, AIRLINE, PRICE
  FROM FLT-SCHEDULE
  WHERE FROM-AIRPORTCODE="ATL"
  AND PRICE<00200.00;
- how a view definition is dropped:
- **DROP VIEW** ATL-FLT [**RESTRICT**|**CASCADE**
- views inherit column names of the base tables they are defined from columns may be explicitly named in the view definition
- column names must be named if inheriting them causes ambiguity
- views may have computed columns, e.g. from applying built-in-functions; these must be named in the view definition
- a view is updatable if and only if:
  - it does not contain any of the keywords JOIN, UNION, INTERSECT, EXCEPT
  - it does not contain the keyword DISTINCT every column in the view corresponds to a uniquely identifiable base table column the FROM clause references exactly one table which must be a base table or an updatable view.
- the table referenced in the FROM clause cannot be referenced in the FROM clause of a nested WHERE clause
- it does not have a GROUP BY clause
- it does not have a HAVING clause
- updatable means insert, delete, update all ok

```
CREATE VIEW LOW-ATL-FARES  /*updatable view*/
AS SELECT *
FROM FLT-SCHEDULE
WHERE FROM-AIRPORTCODE="ATL"
AND PRICE<00200.00;
UPDATE          LOW-ATL-FARES     /*moves row        */
SET      PRICE = 00250.00              /* outside the view*/
WHERE TO-AIRPORTCODE = "BOS";
INSERT INTO LOW-ATL-FARES     /*creates row       */
VALUES ("DL222", "DELTA",          /*outside the view*/
"BIR", 11-15-00, "CHI", 13-05-00, 00180.00);
CREATE VIEW LOW-ATL-FARES
AS SELECT *
FROM FLT-SCHEDULE
WHERE FROM-AIRPORTCODE="ATL"
AND PRICE<00200.00
WITH CHECK OPTION;     /*prevents updates*/  /*outside the view
```

- **DESTROYING/ALTERING TABLES AND VIEWS**
- If we decide that we no longer need a base table and want to destroy it (i.e., delete all the rows *and* remove the table definition information), we can use the DROP TABLE command. For example, DROP TABLE Students RESTRICT destroys the Students table unless some view or integrity constraint refers to Students; if so, the command fails.
- If the keyword RESTRICT is replaced by CASCADE, Students is dropped and any referencing views or integrity constraints are (recursively) dropped as well; one of these two keywords must always be specified. A view can be dropped using the DROP VIEW command, which is just like DROP TABLE.
- ALTER TABLE modifies the structure of an existing table. To add a column called
- *Maiden-name* to Students, for example, we would use the following command:
- ALTER TABLE Students
- ADD COLUMN maiden-name CHAR(10)

- The definition of Students is modified to add this column, and all existing rows are padded with *null* values in this column. ALTER TABLE can also be used to delete columns and to add or drop integrity constraints on a table; we will not discuss these aspects of the command beyond remarking that dropping columns is treated very similarly to dropping tables or views.

- A *view* is a relation whose instance is not explicitly stored but is computed as needed. In addition to enabling logical data independence by defining the external schema through views, views play an important role in restricting access to data for security reasons. Since views might be defined through complex queries, handling updates specified on views is complicated, and SQL-92 has very stringent rules on when a view is updatable.

- SQL provides language constructs to modify the structure of tables (ALTER TABLE) and to destroy tables and views (DROP TABLE).

- A **view** is a table whose rows are not explicitly stored in the database but are computed as needed from a **view definition**. Consider the Students and Enrolled relations.
- The *physical* schema for a relational database describes how the relations in the conceptual schema are stored, in terms of the file organizations and indexes used.
- The *conceptual* schema is the collection of schemas of the relations stored in the database.
- The view mechanism thus provides the support for *logical data independence* in the relational model. That is, it can be used to define relations in the external schema that mask changes in the conceptual schema of the database from applications.

IMP Question Set

SR No.	Question	Reference page No.
1	Q. Write a short note on Views?	17
2	Q. Explain Data independence provided by views?	17
3	Q. Explain creating, altering dropping, renaming and manipulating views using SQL?	17

CHAPTER 3: STORED PROCEDURES

Topic covered:

- Types and benefits of stored procedures, creating stored procedures using SQL,
- executing stored procedures: Automatically executing stored procedures, altering stored procedures, viewing stored procedures.

Q. Explain Advantages of stored procedure?

- **A dvantages of Stored Procedures**

- Stored procedures offer several advantages, both for database users and database administrators, including:
- ***Run-time performance***. Many DBMS brands compile stored procedures (either automatically or at the user's request) into an internal representation that can be executed very efficiently by the DBMS at run-time. Executing a precompiled stored procedure can be much faster than running the equivalent SQL statements through the PREPARE/EXECUTE process.
- ***Reusability***. Once a stored procedure has been defined for a specific function, that procedure may be called from many different application programs that need to perform the function, permitting very easy reuse of application logic and reducing the risk of application programmer error.
- ***Reduced network traffic***. In a client/server configuration, sending a stored procedure call across the network and receiving the results in a reply message generates much less network traffic than using a network round-trip for each individual SQL statement. This can improve overall system performance considerably in a network with heavy traffic or one that has lower speed connections.
- ***Security***. In most DBMS brands, the stored procedure is treated as a "trusted" entity within the database and executes with its own privileges. The user executing the stored procedure needs to have only permission to execute it, not permission on the underlying tables that the stored procedure may access or modify. Thus, the stored procedure allows the database administrator to maintain tighter security on the underlying data, while still giving individual users the specific data update or data access capabilities they require.
- ***Encapsulation***. Stored procedures are a way to achieve one of the core objectives of object-oriented programming—the "encapsulation" of data values, structures, and access within a set of very limited, well-defined external interfaces. In object terminology, stored procedures can be the "methods" through which the objects in the underlying RDBMS are exclusively manipulated. To fully attain the object-oriented approach, all direct access to the underlying data via SQL must be disallowed through the RDBMS security system, leaving *only* the stored procedures for database access. In practice, few if any production relational databases operate in this restricted way.

Q. What are the types of stored procedures?

- The classification of stored procedures is depends on the Where it is Stored. Based on this you can divide it in 4 sections.
  1. System stored procedures
  2. Local stored procedures
  3. Temporary stored procedures
  4. Extended stored procedures

System Stored Procedures:

System stored procedures are stored in the Master database and are typically named with a sp_ prefix. They can be used to perform variety of tasks to support SQL Server functions that support external application calls for data in the system tables, general system procedures for database administration, and security management functions.

For example, you can view the contents of the stored procedure by calling sp_helptext [StoredProcedure_Name].

There are hundreds of system stored procedures included with SQL Server. For a complete list of system stored procedures, refer to "System Stored Procedures" in SQL Server Books Online.

Local stored procedures

Local stored procedures are usually stored in a user database and are typically designed to complete tasks in the database in which they reside. While coding these procedures don't use sp_ prefix to you stored procedure it will create a performance bottleneck. The reason is when you can any procedure that is prefixed with sp_ it will first look at in the mater database then comes to the user local database.

Temporary stored procedures

A temporary stored procedure is all most equivalent to a local stored procedure, but it exists only as long as SQL Server is running or until the connection that created it is not closed. The stored procedure is deleted at connection termination or at server shutdown. This is because temporary stored procedures are stored in the TempDB database. TempDB is re-created when the server is restarted.

There are three types of temporary stored procedures: local , global, and stored procedures created directly in TempDB.
A local temporary stored procedure always begins with #, and a global temporary stored procedure always begins with ##. The execution scope of a local temporary procedure is limited to the connection that created it. All users who have connections to the database, however, can see the stored procedure in Query Analyzer. There is no chance of name collision between other connections that are creating temporary stored procedures. To ensure uniqueness, SQL Server appends the name of a local temporary stored procedure with a series of underscore characters and a connection number unique to the connection. Privileges cannot be granted to other users for the local temporary stored procedure. When the connection that created the temporary stored procedure is closed, the procedure is deleted from TempDB.

Any connection to the database can execute a global temporary stored procedure. This type of procedure must have a unique name, because all connections can execute the procedure and, like all temporary stored procedures, it is created in TempDB. Permission to execute a global temporary stored procedure is automatically granted to the public role and cannot be changed. A global temporary stored procedure is almost as volatile as a local temporary stored procedure. This procedure type is removed when the connection used to create the procedure is closed and any connections currently executing the procedure have completed.

Temporary stored procedures created directly in TempDB are different than local and global temporary stored procedures in the following ways:

You can configure permissions for them.
They exist even after the connection used to create them is terminated.
They aren't removed until SQL Server is shut down.
Because this procedure type is created directly in TempDB, it is important to fully qualify the database objects referenced by Transact-SQL commands in the code. For example, you must reference the Authors table, which is owned by dbo in the Pubs database, as pubs.dbo.authors.

```
--create a local temporary stored procedure.
CREATE PROCEDURE #tempShashi
AS
SELECT * from [pubs].[dbo].[shashi]

--create a global temporary stored procedure.
```

```
CREATE PROCEDURE ##tempShashi
AS
SELECT * from [pubs].[dbo].[shasi]

--create a temporary stored procedure that is local to tempdb.
CREATE PROCEDURE directtemp
AS
SELECT * from [pubs].[dbo].[shashi]
```

Extended Stored Procedures

An extended stored procedure uses an external program, compiled as a 32-bit dynamic link library (DLL), to expand the capabilities of a stored procedure. A number of system stored procedures are also classified as extended stored procedures. For example, the xp_sendmail program, which sends a message and a query result set attachment to the specified e-mail recipients, is both a system stored procedure and an extended stored procedure. Most extended stored procedures use the xp_ prefix as a naming convention. However, there are some extended stored procedures that use the sp_ prefix, and there are some system stored procedures that are not extended and use the xp_ prefix. Therefore, you cannot depend on naming conventions to identify system stored procedures and extended stored procedures.
Use the OBJECTPROPERTY function to determine whether a stored procedure is extended or not. OBJECTPROPERTY returns a value of 1 for IsExtendedProc, indicating an extended stored procedure, or returns a value of 0, indicating a stored procedure that is not extended.

```
USE Master
SELECT OBJECTPROPERTY(object_id('xp_sendmail'), 'IsExtendedProc')
```

Q. Explain How to create stored procedure in SQL?

- **Creating a Stored Procedure**
  - In many common SPL dialects, the CREATE PROCEDURE statement is used to create a stored procedure and specify how it operates. The CREATE PROCEDURE statement assigns the newly defined procedure a name, which is used to call it. The name must typically follow the rules for SQL identifiers (the procedure in Figure 20-1 is named ADD_CUST). A stored procedure accepts zero or more parameters as its arguments (this one has six parameters, C_NAME, C_NUM, CRED_LIMI, TGT_SLS, C_REP, and C_OFFC).
  - In all of the common SPL dialects, the values for the parameters appear in a comma separated list, enclosed in parentheses, following the procedure name when the procedure is called. The header of the stored procedure definition specifies the names of the parameters and their data types. The same SQL data types supported by the DBMS for columns within the database can be used as parameter data types.
  - In Figure 20-1, all of the parameters are *input* parameters (signified by the IN keyword in the procedure header in the Oracle PL/SQL dialect). When the procedure is called, the parameters are assigned the values specified in the procedure call, and the statements in the procedure body begin to execute.
  - The parameter names may appear within the procedure body (and particularly within standard SQL statements in the procedure body) anywhere that a constant may appear. When a parameter name appears, the DBMS uses its current value. In Figure 20-1, the parameters are used in the INSERT statement and the UPDATE statement, both as data values to be used in column calculations and search conditions.
  - In addition to input parameters, some SPL dialects also support *output* parameters.
  - These allow a stored procedure to "pass back" values that it calculates during its execution. Output parameters aren't useful for stored procedures invoked from interactive SQL, but they provide an important capability for passing back information from one stored procedure to another stored procedure that calls it. Some SPL dialects support parameters that operate as *both* input

and output parameters. In this case, the parameter passes a value to the stored procedure, and any changes to the value during the procedure execution are reflected in the calling procedure.

- Figure 20-2 shows the same ADD_CUST procedure definition, expressed in the Sybase Transact-SQL dialect. (The Transact-SQL dialect is also used by Microsoft SQL Server; its basics are largely unchanged since the original Sybase SQL Server version, which was the foundation for both the Microsoft and Sybase product lines.) Note the differences from the Oracle dialect:

```
/* Add a customer procedure */
create proc add_cust
@c_name varchar(20), /* input customer name */
@c_num integer, /* input customer number
*/
@cred_lim money, /* input credit limit */
@tgt_sls money, /* input target sales */
@c_rep integer, /* input salesrep emp # */
@c_offc varchar(15) /* input office city */
as
begin
/* Insert new row of CUSTOMERS table */
insert into customers (cust_num, company, cust_rep,
credit_limit)
values (@c_num, @c_name, @c_rep, @cred_lim)
/* Update row of SALESREPS table */
update salesreps
set quota = quota + quota + @tgt_sls
where empl_num = @c_rep
/* Update row of OFFICES table */
update offices
set target = target + @tgt_sls
where city = @c_offc
/* Commit transaction and we are done */
commit trans
end
```

- **Figure 20-2:** The ADD_CUST procedure in Transact-SQL
- The keyword PROCEDURE can be abbreviated to PROC. No parenthesized list of parameters follow the procedure name. Instead, the parameter declarations immediately follow the name of the stored procedure.

- The parameter names all begin with an at sign (@), both when they are declared at the beginning of the procedure and when they appear within SQL statements in the procedure body.

- There is no formal "end of procedure body" marker. Instead, the procedure body is a single Transact-SQL statement. If more than one statement is needed, the Transact-SQL block structure is used to group the statements.
- Figure 20-3 shows the ADD_CUST procedure again, this time expressed in the Informix stored procedure dialect. The declaration of the procedure head itself and the parameters more closely follows the Oracle dialect. Unlike the Transact-SQL example, the local variables and parameters use ordinary SQL identifiers as their names, without any special identifying symbols. The procedure definition is formally ended with an END PROCEDURE clause, which makes the syntax less error-prone.

```
/* Add a customer procedure */
create procedure add_cust (
c_name varchar(20), /* input customer name */
c_num integer, /* input customer number
*/
```

```
cred_lim money(16,2), /* input credit limit */
tgt_sls money(16,2), /* input target sales */
c_rep integer, /* input salesrep emp # */
c_offc varchar(15)) /* input office city */
/* Insert new row of CUSTOMERS table */
insert into customers (cust_num, company, cust_rep,
credit_limit)
values (c_num, c_name, c_rep, cred_lim);
/* Update row of SALESREPS table */
update salesreps
set quota = quota + quota + tgt_sls
where empl_num = c_rep;
/* Update row of OFFICES table */
update offices
set target = target + tgt_sls
where city = c_offc;
/* Commit transaction and we are done */
commit transaction;
end procedure;
```

Figure 20-3: The ADD_CUST procedure in Informix SPL

- In all dialects that use the CREATE PROCEDURE statement, the procedure can be dropped when no longer needed by a corresponding DROP PROCEDURE statement:
  DROP PROCEDURE ADD_CUST

Q. Write a short note on Executing Stored Procedure?

➢ **Executing a Stored Procedure**

- The next piece of code demonstrates the method of calling a stored procedure from our trigger:

```
DECLARE @errorHold int, --used to hold the error after modification
--statement holds return value from stored
@returnValue int, --procedure for formatting messages
@msg varchar(8000)
EXEC @retval = <procedureName> <parameters>
SET @errorHold = @@error --get error value after update
IF @errorHold <> 0 or @retval < 0 --something went awry
Ensuring Data Integrity
BEGIN
SET @msg = CASE WHEN @errorHold <> 0
THEN 'Error ' + CAST(@errorHold) + 'occurred cascading update to table'
ELSE 'Return value ' + CAST(@retval as varchar(10))
+ ' returned from procedure'
END
--always raise another error to tell the caller where they are
RAISERROR 50000 @msg
ROLLBACK TRANSACTION
RETURN --note the trigger doesn't end until you stop it, as
--the batch is canceled AFTER the trigger
END
```

- As we have seen, the error messages that we get back from constraints are neither very readable nor very meaningful.
- However, when we build triggers, we write our own error messages that can be made as readable and meaningful as we want. We could go further and put every database predicate into our triggers (including unique constraints) but this would require more coding and would not be that advantageous to us; triggers are inherently slower than constraints and are not the best method of protecting our data except in circumstances when constraints cannot be used.

Automatic Execution of Stored Procedures

Stored procedures marked for automatic execution are executed every time SQL Server starts. This is useful if you have operations that you want to perform regularly, or if you have a stored procedure that runs as a background process and is expected to be running at all times. Another use for automatic execution of stored procedures is to have the stored procedure perform system or maintenance tasks in **tempdb**, such as creating a global temporary table. This ensures that such a temporary table will always exist when **tempdb** is re-created as SQL Server starts.

A stored procedure that is automatically executed operates with the same permissions as members of the **sysadmin** fixed server role. Any error messages generated by the stored procedure are written to the SQL Server error log. Do not return any result sets from a stored procedure that is executed automatically. Because the stored procedure is being executed by SQL Server rather than a user, there is nowhere for the result sets to go.

Execution of the stored procedures starts when the **master** database is recovered at startup.

Modifying and Compiling Stored Procecdure

Once a stored procedure is resided in the database server catalog, we can modify it by using the ALTER PROCEDURE statement as follows:

```
1 ALTER PROCEDURE spName(parameter_list)

2 AS

3 BEGIN

4     -- stored procedure body here

5 END
```

To remove a stored procedure from database catalog, we can use DROP PROCEDURE.

```
1 DROP PROCEDURE spName
```

When you remove stored procedure, be noted that some database products have a feature that allows you to remove the dependency database objects like table, view, index or other stored procedures also.

Stored procedure is compiled before it executes. Each database server has its own compiling strategy. You can specify the compiling option when you with WITH RECOMPILE statement as follows:

```
1 CREATE PROCEDURE spName

2     (parameter_list) AS
```

```
3   WITH RECOMPILE

4 BEGIN

5     -- stored procedure body here

6  END
```

The WITH RECOMPILE statement guarantees that each time the stored procedure is invoked, the compiler is called and compile the stored procedure again. With WITH RECOMPILER the stored procedure is recompiled and adjusted to the current situation of the database which brings some advantages to the processing strategy. But be noted that the compilation process takes time and also decreases the performance of the database server. Therefore for each stored procedure ,we can specify which is the best to use WITH RECOMPILE.

Alters a previously created procedure, created by executing the CREATE PROCEDURE statement, without changing permissions and without affecting any dependent stored procedures or triggers. For more information about the parameters used in the ALTER PROCEDURE statement, see CREATE PROCEDURE.

Syntax

```
ALTER PROC [ EDURE ] procedure_name [ ; number ]
   [ { @parameter data_type }
      [ VARYING ] [ = default ] [ OUTPUT ]
   ] [ ,...n ]
[ WITH
   { RECOMPILE | ENCRYPTION
      | RECOMPILE , ENCRYPTION
   }
]
[ FOR REPLICATION ]
AS
   sql_statement [ ...n ]
```

Examples

This example creates a procedure called **Oakland_authors** that, by default, contains all authors from the city of Oakland, California. Permissions are granted. Then, when the procedure must be changed to retrieve all authors from California, ALTER PROCEDURE is used to redefine the stored procedure.

```
USE pubs
GO
IF EXISTS(SELECT name FROM sysobjects WHERE name = 'Oakland_authors' AND type = 'P')
  DROP PROCEDURE Oakland_authors
GO
-- Create a procedure from the authors table that contains author
-- information for those authors who live in Oakland, California.
USE pubs
GO
CREATE PROCEDURE Oakland_authors
AS
```

```
SELECT au_fname, au_lname, address, city, zip
FROM pubs..authors
WHERE city = 'Oakland'
and state = 'CA'
ORDER BY au_lname, au_fname
GO
-- Here is the statement to actually see the text of the procedure.
SELECT o.id, c.text
FROM sysobjects o INNER JOIN syscomments c ON o.id = c.id
WHERE o.type = 'P' and o.name = 'Oakland_authors'
-- Here, EXECUTE permissions are granted on the procedure to public.
GRANT EXECUTE ON Oakland_authors TO public
GO
-- The procedure must be changed to include all
-- authors from California, regardless of what city they live in.
-- If ALTER PROCEDURE is not used but the procedure is dropped
-- and then re-created, the above GRANT statement and any
-- other statements dealing with permissions that pertain to this
-- procedure must be re-entered.
ALTER PROCEDURE Oakland_authors
WITH ENCRYPTION
AS
SELECT au_fname, au_lname, address, city, zip
FROM pubs..authors
WHERE state = 'CA'
ORDER BY au_lname, au_fname
GO
-- Here is the statement to actually see the text of the procedure.
SELECT o.id, c.text
FROM sysobjects o INNER JOIN syscomments c ON o.id = c.id
WHERE o.type = 'P' and o.name = 'Oakland_authors'
GO
```

Viewing Stored Procedures

Several system stored procedures and catalog views provide information about stored procedures. Using these, you can:

- See the definition of the stored procedure. That is, the Transact-SQL statements used to create a stored procedure. This can be useful if you do not have the Transact-SQL script files used to create the stored procedure.
- Get information about a stored procedure such as its schema, when it was created, and its parameters.
- List the objects used by the specified stored procedure, and the procedures that use the specified stored procedure. This information can be used to identify the procedures affected by the changing or removal of an object in the database.

To view the definition of a stored procedure

- sys.sql_modules (Transact-SQL)
- OBJECT_DEFINITION (Transact-SQL)
- sp_helptext (Transact-SQL)

To view information about a stored procedure

- sys.objects (Transact-SQL)
- sys.procedures (Transact-SQL)
- sys.parameters (Transact-SQL)
- sys.numbered_procedures (Transact-SQL)
- sys.numbered_procedure_parameters (Transact-SQL)
- sp_help (Transact-SQL)

To view the dependencies of a stored procedure

- sys.sql_expression_dependencies (Transact-SQL)
- sys.dm_sql_referenced_entities (Transact-SQL)
- sys.dm_sql_referencing_entities (Transact-SQL)

To view information about an extended stored procedure

- sp_helpextendedproc (Transact-SQL)

Quick Revision

- Executing a precompiled stored procedure can be much faster than running the equivalent SQL statements through the PREPARE/EXECUTE process.
- Once a stored procedure has been defined for a specific function, that procedure may be called from many different application programs that need to perform the function, permitting very easy reuse of application logic and reducing the risk of application programmer error.
- In a client/server configuration, sending a stored procedure call across the network and receiving the results in a reply message generates much less network traffic than using a network round-trip for each individual SQL statement.
- The user executing the stored procedure needs to have only permission to execute it, not permission on the underlying tables that the stored procedure may access or modify.
- Stored procedures are a way to achieve one of the core objectives of object-oriented programming—the "encapsulation" of data values, structures, and access within a set of very limited, well-defined external interfaces.
- System stored procedures are stored in the Master database and are typically named with a sp_ prefix.
- Local stored procedures are usually stored in a user database and are typically designed to complete tasks in the database in which they reside.
- A temporary stored procedure is all most equivalent to a local stored procedure, but it exists only as long as SQL Server is running or until the connection that created it is not closed.
- An extended stored procedure uses an external program, compiled as a 32-bit dynamic link library (DLL), to expand the capabilities of a stored procedure. A number of system stored procedures are also classified as extended stored procedures.
- The CREATE PROCEDURE statement assigns the newly defined procedure a name, which is used to call it.

IMP Question Set

SR No.	Question	Reference page No.
1	Q. Explain Advantages of stored procedure?	20
2	Q. What are the types of stored procedures?	20
3	Q. Explain How to create stored procedure in SQL?	22
4	Q. Write a short note on Executing Stored Procedure?	24

CHAPTER 4: TRIGGERS

Topic covered:

- Concept of triggers, Implementing triggers in SQL: creating triggers, Insert, delete, and update triggers, nested triggers, viewing, deleting and modifying triggers, and enforcing data integrity through triggers.

Q. Explain triggers?

A special kind of "stored procedure" that goes into effect when you modify data.
- In a specified table using one or more data modification operations: UPDATE, DELETE, INSERT.
- Can query other tables and can include complex SQL statements

They are primarily useful for enforcing complex business rules or requirements
- you could control whether to allow an order to be inserted based on a customer's current account status useful for enforcing referential integrity, which preserves the defined relationships between tables when you add, update, or delete the rows in those tables
- A state in which all foreign key values in a database are valid

Triggers are automatic
- they are activated immediately after any modification to the table's data, such as a manual entry or an application action

Can cascade change trough related tables in a database
- you can write a delete trigger on the title_id column of the titles table to cause a deletion of matching rows in other tables
- can enforce restrictions that are more complex than those defined with check constraints

Defines which data value are acceptable in a column

❖ **INSERT Trigger**

❖ When a new row is inserted in the requisition table, the value of tax column should be less then salary column
❖ Ensure that this user-defined data integrity requirement is implemented
❖ Task List
- Identify the object that can maintain user defined data integrity
- Draft statement to create an INSERT trigger
- Create trigger in database
- Check the existence of trigger in the database
- Insert a row in the requisition table and verify that trigger is working

❖ Identify the object that can maintain user defined-data integrity
➢ A trigger is a block of code that constitutes a set of T-SQL statements that are activated in response to certain actions
➢ Characteristic trigger :
- It is fired automatically by DBMS when any data modification statement is issued
- It cannot be explicitly invoked or executed, as in the case of the stored procedures

❖ Identify …
➢ It prevents incorrect, unauthorized or inconsistent changes in data
➢ It cannot return data to the user
❖ Result
➢ A trigger can be used to maintain data integrity

Q. Which statement is used to create triggers?

- **Creating trigger**
  - Syntax

```
CREATE TRIGGER trigger_name
ON table_name
[WITH ENCRYPTION]
FOR [INSERT | DELETE | UPDATE]
AS sql_statements
```

- **Magic tables**
  - Whenever a trigger fires in response to the INSERT, DELETE or UPDATE statement, tw special table are created
  - These are the inserted and the deleted tables
  - Inserted table contains a copy of all records that are inserted in the trigger table
  - The delete table contains all records that have been deleted from trigger table
  - Whenever any update takes place, the trigger uses both the inserted and the deleted tables
- An INSERT Trigger is fired whenever an attempt is made to insert a row in the trigger table
- When an INSERT statement is issued, a new row is added to both trigger and the inserted tables
- Action :
  - The table on which the trigger has to be created is requisition
  - The trigger has to be of insert type
  - The name of trigger can be trginsertRequisiton
  - Write the batch statements

```
CREATE TRIGGER trgInsertRequisition
ON Requisition FOR insert AS
DECLARE @vacancyreported int
DECLARE @actualvacancy int
SELECT @actualvacancy = iBudgetedStrength – iCurrentStrength
FROM Position Join Inserted on Position.cPositionCode = Inserted.cPositionCode
SELECT @vacancyreported = inserted.siNoOfVacancy FROM inserted
IF (@vacancyreported > @actualvacancy)
BEGIN
PRINT 'The actual vacancies are less than the vacancies reported. Hence, cannot
    insert.'
ROLLBACK TRANSACTION
END
RETURN
```

- Create Trigger in the database
  - Action :
  - Type the drafted Code in the Query Analyzer window
  - Press F5 to execute the code
    - Check the existence of trigger in the database
    - Action :
    - Sp_help trgInsertRequisition
    - Insert a row in the Requisition table and verify that the trigger is working
    - Action :
    - INSERT Requisition
    - VALUES ('000003','0001',getdate(),getdate() + 7, '0001', 'North', 20)

Q. When is a DELETE Trigger fired?

- **DELETE Trigger**
  - Create trigger to disable deleting rows from the ContractRecruiter table
  - Task List
    - Draft statement to create a delete trigger
    - Create the trigger in the database
    - Check the existence of the trigger in the database
    - Delete a row from the Contractrecruiter table to verify the trigger
  - A DELETE Trigger is fired whenever an attempt is made to delete rows from the trigger table
  - There are three ways of implementing referential integrity using a DELETE trigger. These are :
    - The cascade method
    - The restrict method
    - The nulify method
  - Result :
- The table on which the trigger is to be created is ContractRecruiter
- The trigger is a DELETE Trigger
- The name of the trigger is trgDeleteContractRecruiter
- The batch statements are :
  - CREATE TRIGGER trgDeleteContractRecruiter
  - ON ContractRecruiter FOR delete
  - AS
  - PRINT 'Deletion of Contract Recuiters is not allowed'
  - ROLLBACK TRANSACTION RETURN
  - Create trigger in the database
  - Type drafted code in the Query analyzer window
  - Press F5 to execute the code
- Check the existence of the trigger in the database
- Sp_help trgDeleteContractRecruiter
  - Delete a row from the ContractRecruiter table to verify the trigger
  - Action :
- Execute the following statement :
  - DELETE ContractRecruiter
  - WHERE cContractRecruiterCode = '000001'
- When this command is executed, the trigger is would be fired and it would prevent the deletion of rows from the ContractRecruiter table.

Q. Explain Nested triggers?

- A trigger that contains data modification logic within itself is called a nested trigger.
- Use the nested triggers option to control whether an AFTER trigger can cascade; that is, perform an action that initiates another trigger, which initiates another trigger, and so on. When nested triggers is set to 0, AFTER triggers cannot cascade. When nested triggers is set to 1 (the
- default), AFTER triggers can cascade to as many as 32 levels. INSTEAD OF triggers can be nested regardless of the setting of this option.
- If nested triggers are enabled, a trigger that changes a table on which there is another trigger fires the second trigger, which can in turn fire a third trigger, and so forth. If any trigger in the chain sets off an infinite loop, the nesting level is exceeded and the trigger aborts. You can use nested triggers to perform useful housekeeping functions such as storing a backup copy of rows affected by a previous trigger.

- **UPDATE Trigger**
  - Create a trigger so that the average siPercentageCharge attribut of the ContractRecruiter table should not be more than 11 when the value of siPercentageCharge is increased for any ContractRecruiter
  - Task List
- Draft statements to create an update trigger
- Create trigger in the database
- Check the existence of the trigger in the database
- Update siPercentageCharge of the ContractRecruiter table and verify that the average does not exceed the required value
- Draft statements to create an UPDATE trigger
  - **The update trigger**
- Fired whenever there is a modification to the trigger table
- Result :
- The table on which the trigger is to be created is ContractRecruiter
- The trigger is an UPDATE trigger
- The name of the trigger is trgUpdateContractRecruiter

```
CREATE TRIGGER trgUpdateContractRecruiter
ON ContractRecruiter FOR UPDATE
AS
DECLARE @AvgPercentageCharge int
SELECT @AvgPercentageCharge = avg(siPercentageCharge)
FROM ContractRecruiter
IF (@AvgPercentageCharge > 11)
BEGIN
PRINT 'The average cannot be more than 11'
ROLLBACK TRANSACTION
END
RETURN
```

- Create the trigger in the database
- In the query analyzer window, type the draft code
- Press F5 to execute the code
- Check the existence of the trigger in the database
- Sp_help trgUpdateContractRecruiter
- Update siPercentageCharge of the ContractRecruiter table and verify that the average does not exceed the required value

Q. Which statement is used to recreate a trigger?

- **Modifying the trigger**

- **Task List**
  - Draft the command to modify the trigger
  - Create the trigger in the database
  - Check that the trigger has been modified in the database
  - Insert a row in the requisition table and verify that the trigger is working

- **The ALTER TRIGGER Command**

- The contents of a trigger can be modified by:
  - Dropping the trigger and recreating it
  - Using the ALTER TRIGGER statement
- It is advisable to drop a trigger and recreate it, if the objects being referenced by it are renamed

- **Syntax:**

```
ALTER TRIGGER trigger_name
ON table_name
[WITH ENCRYPTION]
FOR [INSERT | DELETE | UPDATE]
AS sql_statements
```

  - Action :
  - The table on which the trigger had been created is Requisiton
  - Determine the type of the trigger
  - The name of the trigger to be modified is trgInsertreRequisiton

  - Check that the trigger has been modified in the database
  - Sp_helptext trgInsertRequisiton
- Insert a row in Requisition table and verify that the trigger is working

➢ **Dropping the Trigger**

- The DELETE trigger named trgDeleteContractRecruiter needs to be removed, as it is no longer required
- **Task List**
- Draft the command to delete the trigger
- Execute the command
- Verify that the trigger has been removed
- The DROP Trigger command is used to delete a trigger from the database
- DROP TRIGGER trigger_name[…,n]
- Action:
- The command to delete the trigger would be:
  - DROP TRIGGER trgDeleteContractRecruiter
- Execute the command
- Action:
  - In the query……..
  - Verify that the trigger has been removed
  - Sp_help trgDeleteContractRecruiter

➢ The above command will give an error message as the trigger has been removed

Q. Explain Enforcing Data Integrity Through Triggers?

➢ **Enforcing Data Integrity Through Triggers.**

- A trigger can be used to enforce business rules and data integrity in the following ways:
- If changes are made to the master table, then the same changes are cascaded to all the dependent tables
- If some changes violate referential integrity, then all such changes are rejected, thereby canceling any attempt to modify data in the database
- It allows very complex restriction to be enforced
- It can perform a particular action, depending on the outcome of the modifications that have been made to the tables
- **Multiple triggers**
- SQL Server allows multiple triggers to be defined on a given table. This implies that a single DML statement may fire two or more triggers. The triggers are fired in the order of creation

- It is very important for data in a database to be accurate and correct. There are various methods to ensure this, but triggers have their own significance in maintaining the integrity and consistency of data.

A trigger can be used to enforce business rules and data integrity in the following ways:

If changes are made to the master table, then the same changes will be cascaded to all the dependent tables.

Suppose you have to delete a record from the Titles table, then all the records in the dependent table, TitleAuthor should also be deleted to maintain data integrity.
Example

```
CREATE TRIGGER  trgDeleteTitle
ON Titles
FOR DELETE
AS
 DELETE TitleAuthor
FROM TitleAuthor  t  JOIN Delete  d
ON t.Title_ID = D.Title_ID
```

If some changes violate referential integrity, then all such changes are rejected, thereby cancelling any a attempt to modify data in the database.

It allows very complex restrictions to be enforced.

It can perform a particular action, depending on the modification that have been made to the table.

- A trigger can be used to ensure and enforce business rules and data integrity. Business rules refer to the policies of an organization, which ensure that its business runs smoothly. Data integrity refers to the accuracy and reliability of data.

Quick Revision

- A trigger is a stored procedure that goes into effect when you insert, delete, or update data in a table.
- **Triggers are automatic**: they are activated immediately after any modification to the table's data, such as a manual entry or an application action.
- **Trigger** Defines which data value are acceptable in a column
- Whenever a trigger fires in response to the INSERT, DELETE or UPDATE statement, two special table are created These are the inserted and the deleted tables.
- A trigger is a block of code that constitutes a set of T-SQL statements that are activated in response to certain actions.
- A trigger can be used to enforce business rules and data integrity in the following ways:
    - If changes are made to the master table, then the same changes are cascaded to all the dependent tables
    - If some changes violate referential integrity, then all such changes are rejected, thereby canceling any attempt to modify data in the database
    - It allows very complex restriction to be enforced
    - It can perform a particular action, depending on the outcome of the modifications that have been made to the tables
- A DELETE Trigger is fired whenever an attempt is made to delete rows from the trigger table

IMP Question Set

SR No.	Question	Reference page No.
1	Q. Explain triggers?	30
2	Q. Which statement is used to create triggers?	31
3	Q. When is a DELETE Trigger fired?	32
4	Q. Explain Nested triggers?	32
5	Q. Which statement is used to recreate a trigger?	33
6	Q. Explain Enforcing Data Integrity Through Triggers?	34