View

A database view is a searchable object in a database that is defined by a query. Though a view doesn't store data, some refer to a views as "virtual tables," you can query a view like you can a table. A view can combine data from two or more table, using joins, and also just contain a subset of information. This makes them convenient to abstract, or hide, complicated queries.

Stored procedure

A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again.

So if you have an SQL query that you write over and over again, save it as a stored procedure, and then just call it to execute it.

You can also pass parameters to a stored procedure, so that the stored procedure can act based on the parameter value(s) that is passed.

Material View

A materialized view is a pre-computed data set derived from a query specification (the SELECT in the view definition) and stored for later use.

Because the data is pre-computed, querying a materialized view is faster than executing a query against the base table of the view. This performance difference can be significant when a query is run frequently or is sufficiently complex. As a result, materialized views can speed up expensive aggregation, projection, and selection operations, especially those that run frequently and that run on large data sets.

Issues:

How and when the view will be updated. Ideally it'll regenerate in response to an event indicating a change to the source data, although this can lead to excessive overhead if the source data changes rapidly. Alternatively, consider using a scheduled task, an external trigger, or a manual action to regenerate the view.

Trigger

A trigger is a stored procedure in database which automatically invokes whenever a special event in the database occurs. For example, a trigger can be invoked when a row is inserted into a specified table or when certain table columns are being updated.

Syntax:

```
create trigger [trigger_name]
[before | after]
{insert | update | delete}
on [table_name]
[for each row]
[trigger_body]
```

B Tree And B+ Tree

They are Used to Store Data in Disks When the Entire Data Cannot be Stored in the Main Memory

When the amount of data to be stored is very high, we cannot store the entire data in the main memory. Hence we store data in the disk. Data access from the disk takes more time when compared to the main memory access.

When the number of keys of the data stored in disks is very high, the data is usually accessed in the form of blocks. The time to access these blocks is directly proportional to the height of the tree.

B-Tree:

B-Tree is known as a self-balancing tree as its nodes are sorted in the inorder traversal. In B-tree, a node can have more than two children. B-tree has a height of logM N (Where 'M' is the order of tree and N is the number of nodes). And the height is adjusted automatically at each update. In the B-tree data is sorted in a specific order, with the lowest value on the left and the highest value on the right. To insert the data or key in B-tree is more complicated than a binary tree.

There are some conditions that must be hold by the B-Tree:

1. All the leaf nodes of the B-tree must be at the same level.
2. Above the leaf nodes of the B-tree, there should be no empty sub-trees.
3. B- tree's height should lie as low as possible.

B+ Tree

B+ tree eliminates the drawback B-tree used for indexing by storing data pointers only at the leaf nodes of the tree. Thus, the structure of leaf nodes of a B+ tree is quite different from the structure of internal nodes of the B tree. It may be noted here that, since data pointers are present only at the leaf nodes, the leaf nodes must necessarily store all the key values along with their corresponding data pointers to the disk file block, in order to access them. Moreover, the leaf nodes are linked to providing ordered access to the records. The leaf nodes, therefore form the first level of the index, with the internal nodes forming the other levels of a multilevel index. Some of the key values of the leaf nodes also appear in the internal nodes, to simply act as a medium to control the searching of a record.

S.NO	B tree	B+ tree
1.	All internal and leaf nodes have data pointers	Only leaf nodes have data pointers
2.	Since all keys are not available at leaf, search often takes more time.	All keys are at leaf nodes, hence search is faster and accurate..
3.	No duplicate of keys is maintained in the tree.	Duplicate of keys are maintained and all nodes are present at leaf.
4.	Insertion takes more time and it is not predictable sometimes.	Insertion is easier and the results are always the same.
5.	Deletion of internal node is very complex and tree has to undergo lot of transformations.	Deletion of any node is easy because all node are found at leaf.
6.	Leaf nodes are not stored as structural linked list.	Leaf nodes are stored as structural linked list.
7.	No redundant search keys are present..	Redundant search keys may be present..