Physical Design
Week 4

Improving Database Performance

Indexes

Files

- Page or block is OK when doing I/O, but higher levels of DBMS operate on records, and files of records
- For persistent storage databases are mapped into a number of files
  - Located in specially protected parts of the file system (tablespaces, etc.)
  - Actually maintained by the operating system

- FILE: A collection of pages, each containing a collection of records
- Must support:
  - insert/delete/modify record
  - read a particular record (specified using record id)
  - scan all records (possibly with some conditions on the records to be retrieved)

Files

- File organization: Method of arranging a file of records on external storage
  - One file can have multiple pages
  - Record id (rid) is sufficient to physically locate the page containing the record on disk
  - Indexes are data structures that allow us to find the record ids

Files

- Lots of alternatives, some work well in some situations but not in others
  - Heap (random order) files: Suitable when typical access is a file scan retrieving all records (linear search to locate)
  - Sorted Files: Best if records must be retrieved in some order, or only a "range" of records is needed.
    - Sequential – store records in sequential order, based on the value of the search key of each record
    - Hashed – a hash function is computed on some attribute of each record. The result specifies in which block of the file the record should be placed
  - Indexes: Data structures to organize records via trees or hashing
    - Like sorted files, they speed up searches for a subset of records, based on values in certain ("search key") fields
    - Updates are much faster than in sorted files

Indexes

Indexes

- Indexes help to locate records in a DB file
- Indexes are additional auxiliary access structures with typically provide either faster access to data or secondary access paths without effecting the physical storage of the data
- They are based on indexing field(s) that are used to construct the index.
- Indexes can be sparse or dense.
  - A dense index has an entry for each record.

Indexes

- Creation of indexes is part of the physical tuning task of database administrators
  - Based on understanding of common user queries
- Indexes often influence the actual location of storage for a record
- Example:
  - Sequential storage, storage via a hash function
  - If the location is determined by the index
- Not all attributes can be directly indexed (but secondary access paths may be used)

Indexes

MovieStar(Name,Address,Gender,Birthdate)

SELECT *

FROM MovieStar

WHERE Name = 'Daniel Craig';

- All the blocks for the MovieStar relation will be inspected if there is no index on Name

Indexes

- An Index is a data structure that facilitates the query answering process by minimizing the number of disk accesses.
- An index structure is usually defined on a single Attribute of a Relation, called the **Search Key.**
- An Index takes as input a Search Key value and returns the address of the record(s) (block physical address offset of the record) holding that value.
- Index structure: Search Key-Pointer p

Search Key	Pointer to a data-file record

Indexes

- The Search Key values stored in the Index are **Sorted** and a binary search can be done on the Index.
- Only a small part of the records of a relation have to be inspected
  - Appropriate indexes can speed up query processing passing from minutes to seconds

Search Key	Pointer to a data-file record

Indexes

- Single-Level Indexes
  - There is only 1 level of indirection.
  - The entries in the index are pointing to the data in the table
- Multi-Level Indexes
  - Entries in the index might point to other entries in the index that eventually point to the data

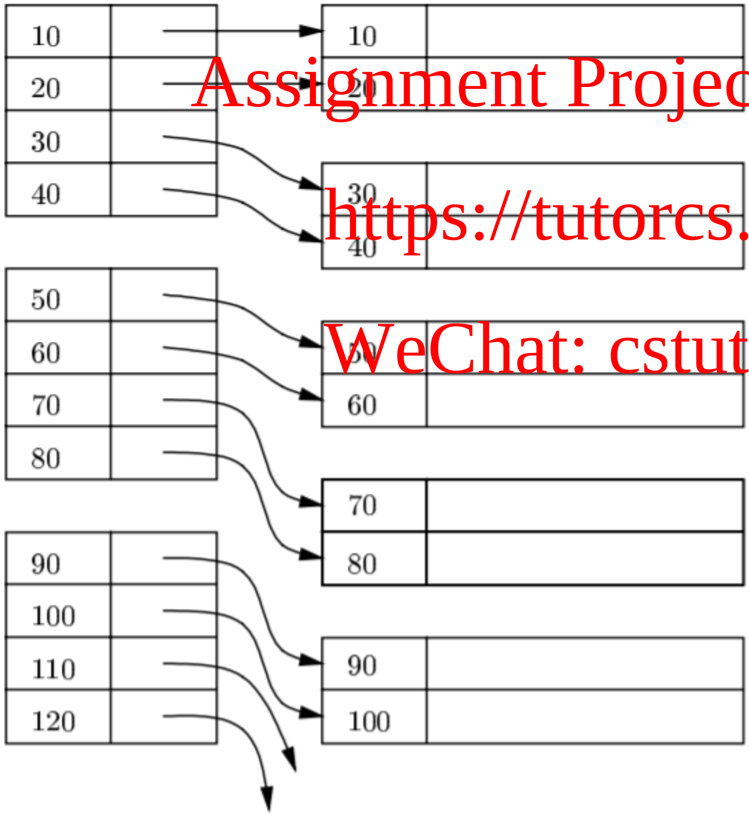
Primary Index

- A ***Primary Index*** is specified on the ***ordering key field*** where each tuple has a ***unique*** value.
  - Order data by some usually unique attribute as indexing field (primary key), store database records in this order
  - Index record contains pointer to the respective storage place (block address)
  - To save entries usually there is only a single index entry for each block (block anchor)

Primary Index

- Sequential
- There can be just one Primary Index for Data File.
- Usually used when the search key is also the primary key of the relation.
- Usually, these indexes fit in main memory.
- Indexes can be:
  - 1. Dense: One entry in the index file for every record in the data
  - 2. Sparse: One entry in the index file for each block of the data

Primary Index



Every value of the search key has a representative in a Dense Index.
The index maintains the keys in the same order as in the data file.

Database System Implementation, H. Garcia-Molina, J. Ullman, and J. Widom, Prentice-Hall, 2000.

Primary Index

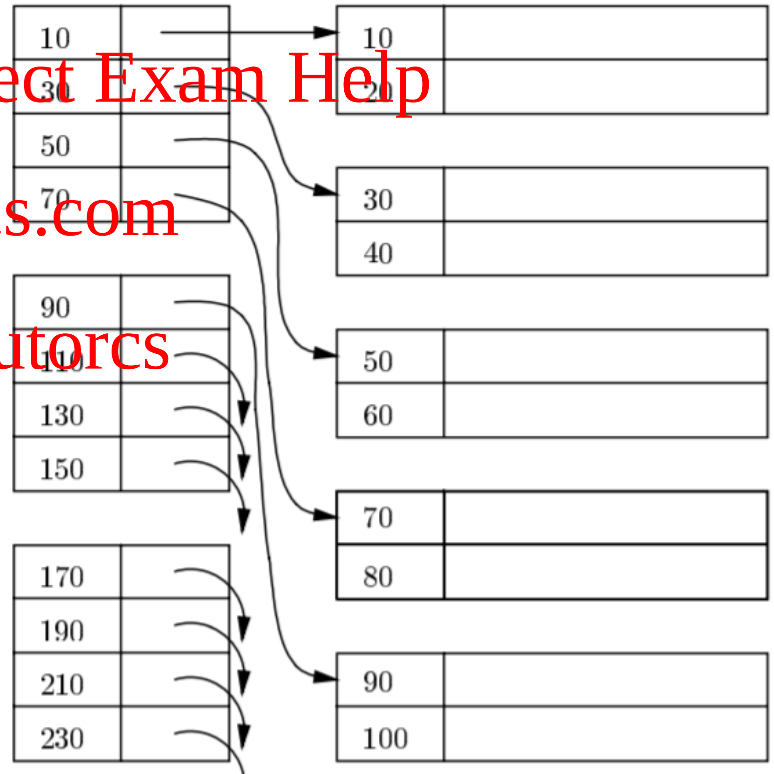
- Advantages
  - Number of blocks needed for storing the index is small compared to data
  - Index entries are smaller than data records
  - Can often be kept in buffer
- Disadvantages
  - Insertions and Deletions need to move data in storage and to update index entries affected by the shifts

Primary Index – Dense Index

- Searching a data record with a given search key value.
  - The index is scanned and when key is found the associated pointer to the data file record is followed and the record (block containing it) is read in main memory.
- Dense indexes support also range queries:
  - The minimum value is located first, if needed, consecutive blocks are loaded in main memory until a search greater than the maximum value is found.
- Query-answering using dense indexes is efficient:
  - 1. Since the index is usually kept in main memory, just 1 disk I/O has to be performed during lookup;
  - 2. Since the index is sorted we can use binary search: If there are n search keys then at most **log2n** steps are required to locate a given search

Primary Index – Sparse Index

- Used when dense indexes are too large
- A Sparse Index uses less space at the expense of more time to find a record given a key.
- A sparse index holds one key-pointer pair per data block, usually the first record on the data block



em Implementation, H. Garcia-Molina, J. Ullman, and J. Widom, Prentic

Primary Index – Sparse Index

- Given a search key K:
  - 1. Search the sparse index for the greatest key <= K using binary search;
  - 2. We retrieve the pointed block to main memory to look for the record with search key K (always using binary search).
- In comparison to a dense index we need to start two different binary searches:
  - The first on the sparse index
  - The second on the retrieved data block.
- A Sparse Index is more efficient in space at the cost of a worst computing time in Main

Primary Index – Dense Index

Example of a **Primary Dense Index** with **Search Key**=`Account#`.

A-101	
A-102	
A-110	
A-201	
A-215	
A-217	
A-218	
A-222	
A-305	

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

Primary Index – Sparse Index

Example of a **Primary Sparse Index** with **Search Key**=`Account#`.

A-101	
A-201	
A-218	

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

Secondary Index

- Specified on a ***NON-ORDERING Field*** of the file.
- Facilitate query-answering on attributes other than primary keys – or, more generally, on non-ordering attributes
- Can have several secondary indexes

Secondary Index

- Let us consider the MovieStar relation:

  `MovieStar(Name,Address,Gender,Birthdate)`

- and a query involving the non-key Birthdate attribute:

```
SELECT Name, Address FROM MovieStar WHERE
Birthdate = '1975-01-01';
```

- A secondary index on the MovieStar relation w.r.t. the Birthdate attribute would reduce the answering time

Secondary Index

- Point to locations of records regarding a non-ordering attribute
  - Indexing does not affect the storage order
  - There can be multiple secondary indexes for the same DB file
- Secondary indexes are usually dense
  - Objects with same or adjacent values are usually not adjacent on disk
  - If the indexing field has unique values (secondary key) all records must be indexed

Secondary Index

- Secondary indexes are sorted w.r.t. the search key => Binary search.
- The Data **IS NOT** sorted w.r.t. the Secondary Index Search Key
- More than one data block may be needed for a given search key so in general more disk I/O is required to answer queries:
  - Secondary Indexes are less efficient than Primary Indexes

Secondary Index

- If the indexing field is not unique there are several possibilities to create a secondary index
  - Create a dense index by including duplicate search keys (one for each record)
  - Use variable-length index entries, where each search key is assigned a list of pointers
  - Keep fixed-length index entries, but point to a block containing (multiple) pointers to the actual records
    - Introduces a level of indirection, but allows for a sparse index
    - Usually used in practice

Secondary Index

- Advantages
  - Speeds up retrieval, because if it does not exist, the entire file would have to be scanned linearly
  - Secondary indexes provides a **logical** ordering
    - Accessing records in that order might not be the most efficient way regarding block accesses
    - Each record access may fetch a new block into the buffer

Secondary Index

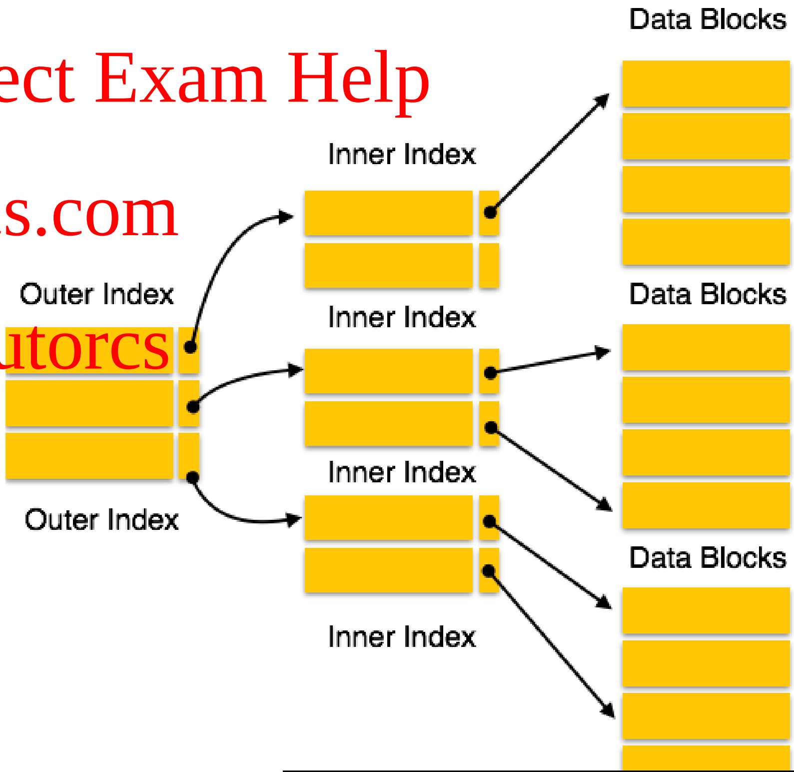
- You can use a separate index for every attribute you wish to use in the WHERE clause of your select query
- But there is the overhead of maintaining a large number of these indexes

Clustered Index

- **Clustering indexes** store data records in the order of a non-unique indexing field not pointers
- Reorders the way records are stored on disk based on key value as included in the index definition
- Sorts the data rows in the table on their key values. In the Database, there is only one clustered index per table.
- Can be for primary or secondary indexes
- Sometimes the Index is created on non-primary key columns which might not be unique for each record.
  - In this situation, you can group two or more columns to get the unique values and create an index which is called clustered Index. This also helps you to identify the record faster.

Multilevel Indexes

- Created when a primary index does not fit in memory
- A ***Multilevel Index*** is where you construct a ***Second- Level*** index on a ***First-Level*** Index.
- This allows much faster access than binary search because at each level the size of the index is reduced by the fan out factor. Rather just by 2 as in binary search.
- In databases, multi-level indexes are implemented using a tree structure
- A tree is a data structure with specific rules

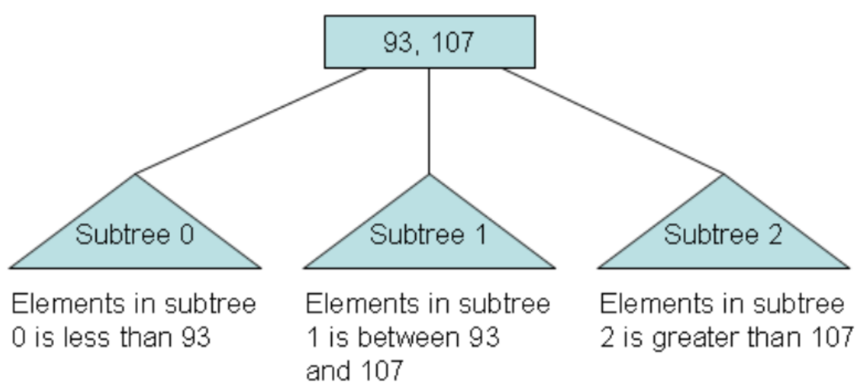


B-Trees - Terminology

- Tree is formed of nodes
- Each node (except root) has one parent and zero or more child nodes
- Leaf node has no child nodes
  - Tree is unbalanced if leaf nodes occur at different levels
- Non-leaf node is called an internal node
- Subtree of node consists of the node and all its descendant nodes
- Most common type is B* tree which stores only data pointers at the leaf nodes
- For each node x, the keys are stored in increasing order.
- If n is the order of the tree, each internal node can contain at most n - 1 keys along with a pointer to each child.

B-Trees - Terminology

- Every B-tree depends on a positive constant integer *minimum degree 't'.* (depends on block size for indexes)
  - Used to determine how many elements are held in a single node.
- The root can have as few as one element (or even no elements if it also has no children).
  - Every other node has at least t-1 elements.
- All nodes (including root) may contain *at most* 2*t – 1 keys
- Number of children of a node is equal to the number of keys in it plus 1.
- All keys of a node are sorted in increasing order. The child between two keys k1 and k2 contains all keys in the range from k1 and k2.
- Insertion of a Node in B-Tree happens only at Leaf Node.

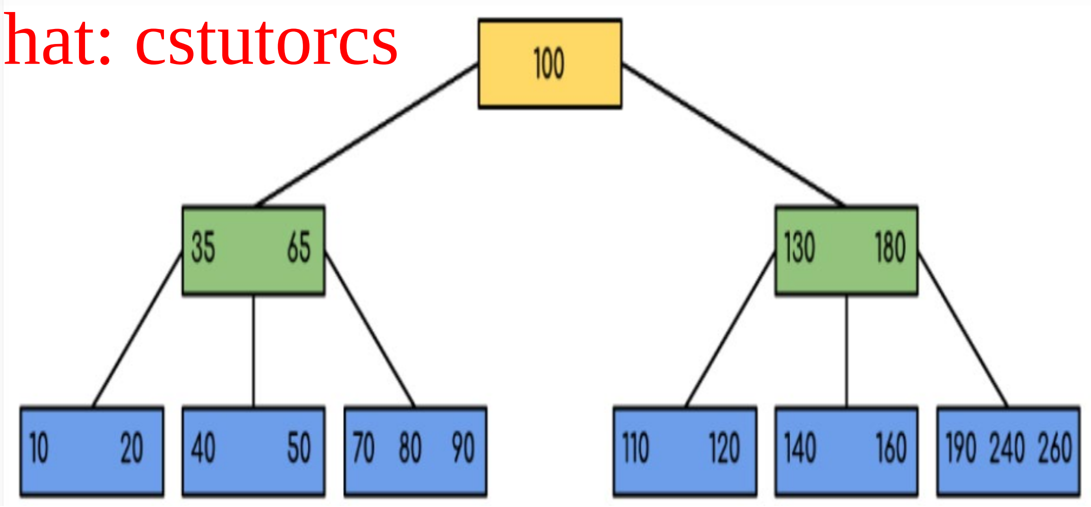


B-Trees

- Problem: Given a B-tree (dense) index, find a record with search key K.
- Recursive search, starting at the root and ending at a leaf:
  - 1. If we are at a leaf then if K is among the keys of the leaf follow the associated pointer to the data, else fail.
  - 2. If we are at an interior node (including the root) with keys K1, K2… Kn, then if K <K1 then go to the first child, if K1 <= K < K2 then go to the second child, and so on.
- Note: B-Trees are useful for queries in which a range of values are asked for: Range Queries

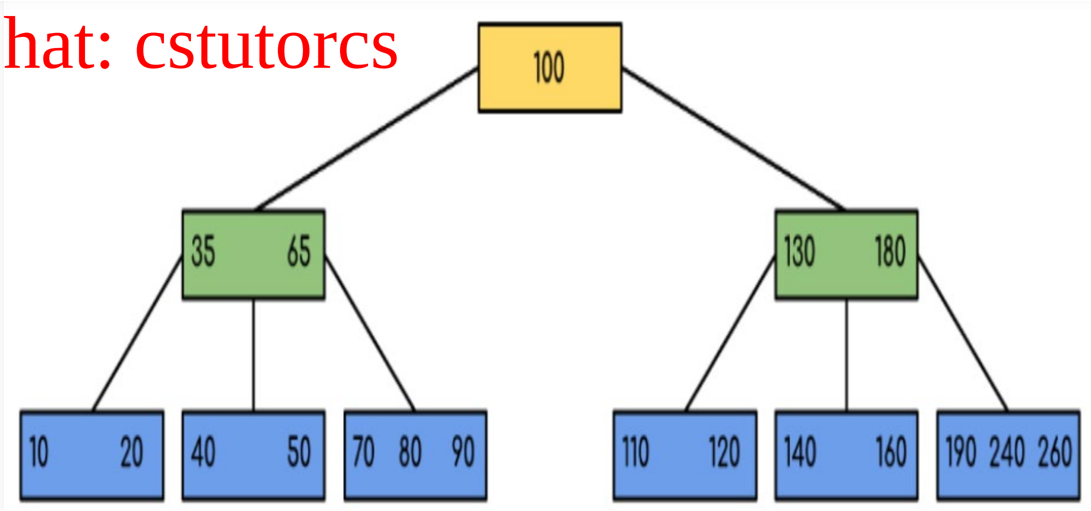
B-Trees - Terminology

- B-Tree of minimum order 5
- All the leaf nodes are at the same level and all non-leaf have no empty sub-tree and have keys one less than the number of their children.



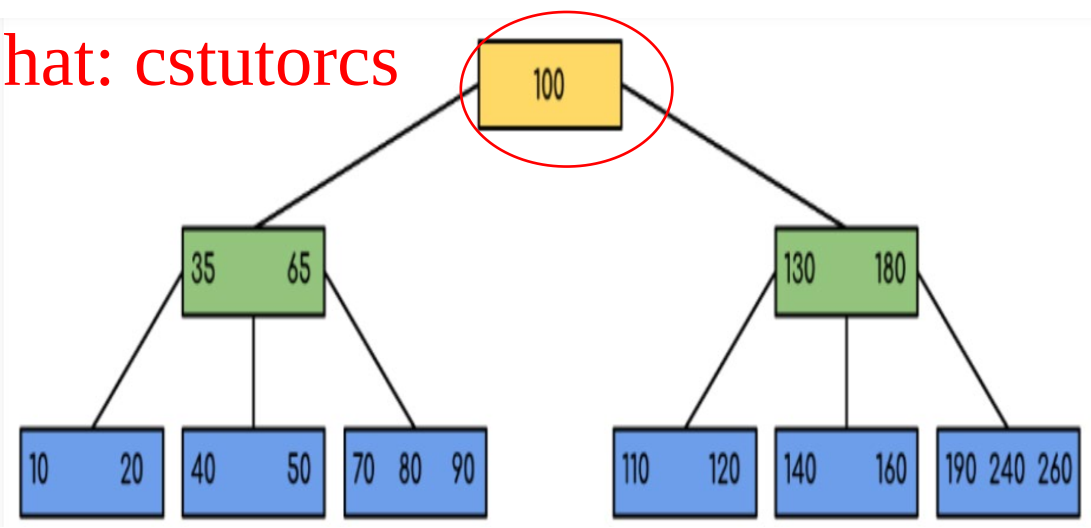
B-Trees - Terminology

- Search:
  - Let the key to be searched be k.
  - Start from the root and recursively traverse down.
  - For every visited non-leaf node, if the node has the key, return the node.
  - Otherwise, recur down to the appropriate child (The child which is just before the first greater key) of the node.
  - If a leaf node is reached and k is not found in the leaf node, return NULL.



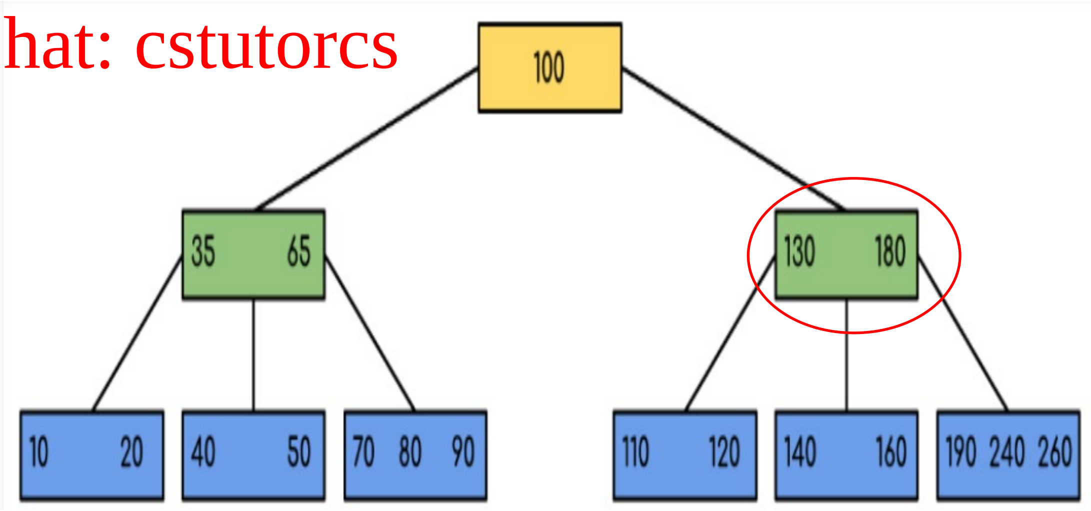
B-Trees - Terminology

- Search:
  - Let the key to be searched be 120.
  - First search will start with the root node.
  - Not found



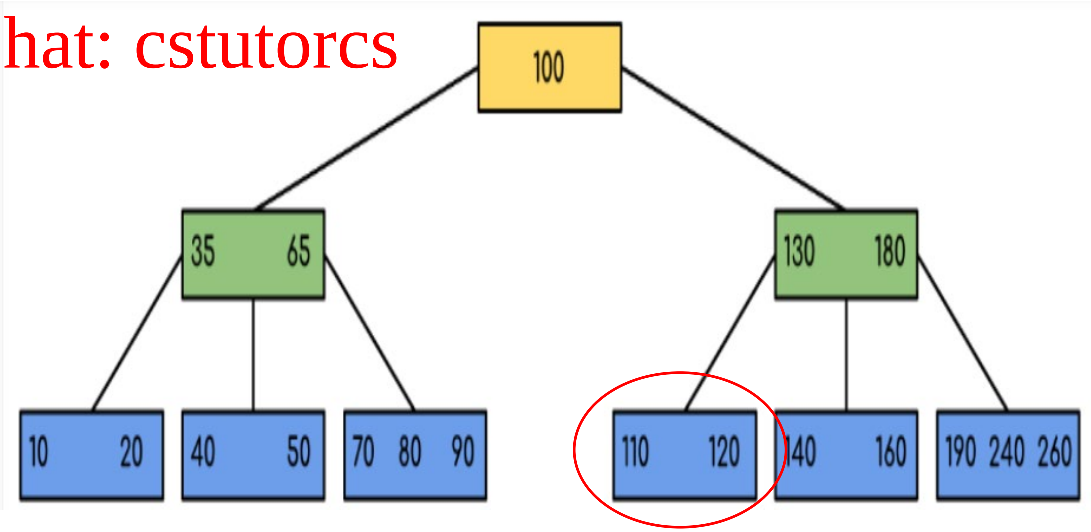
B-Trees - Terminology

- Search:
  - Next search will jump to child 130 180 (as 120 > 100)
  - Will check the range the key falls into
    - As it is less than 130 it will be in the left branch if it is there at all



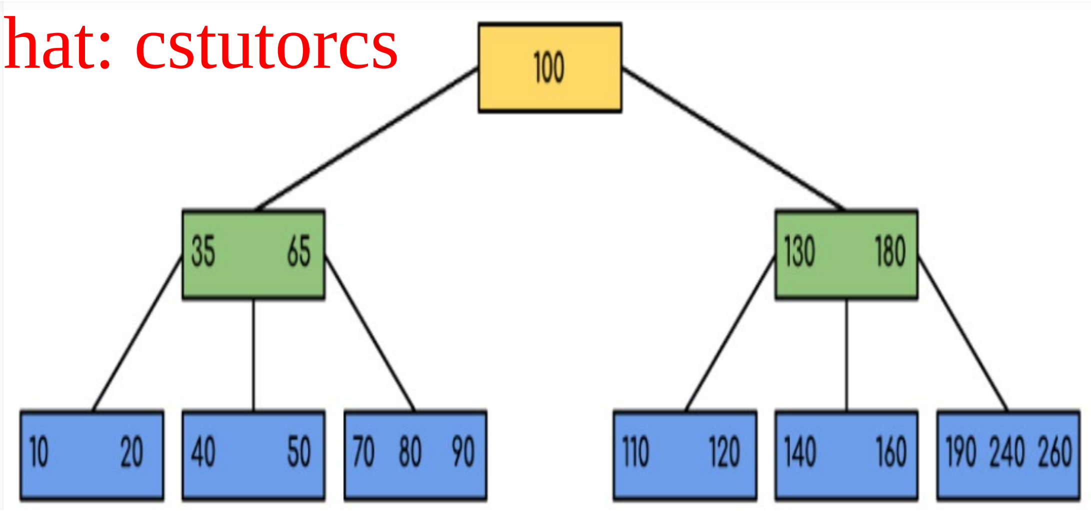
B-Trees - Terminology

- Search:
  - As 120 is less than 130 it will be in the right most node



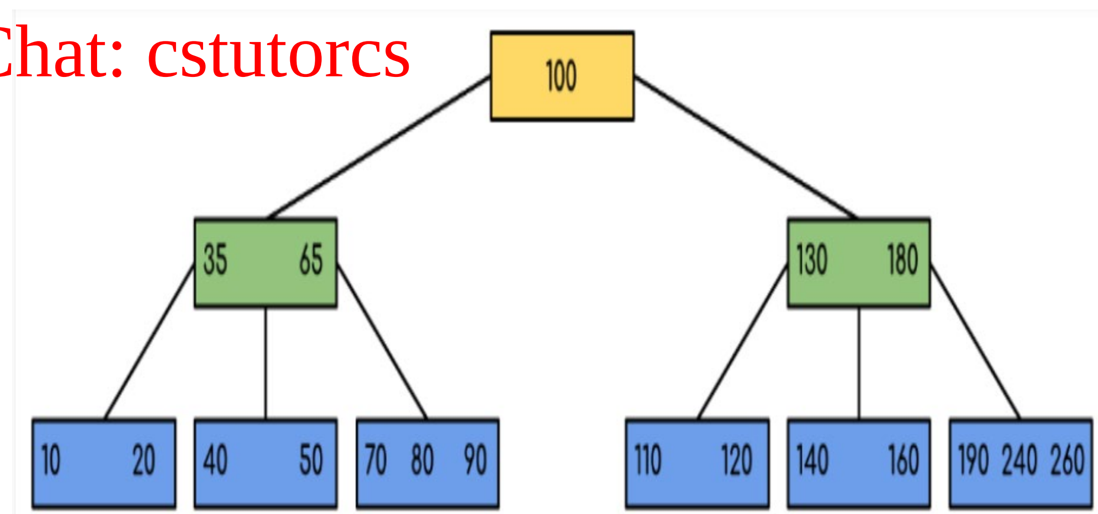
B-Tree Terminology

- Search:
  - If we are looking for 180, then the control will stop at step 2 because the program will find that the key 180 is present within the current node.



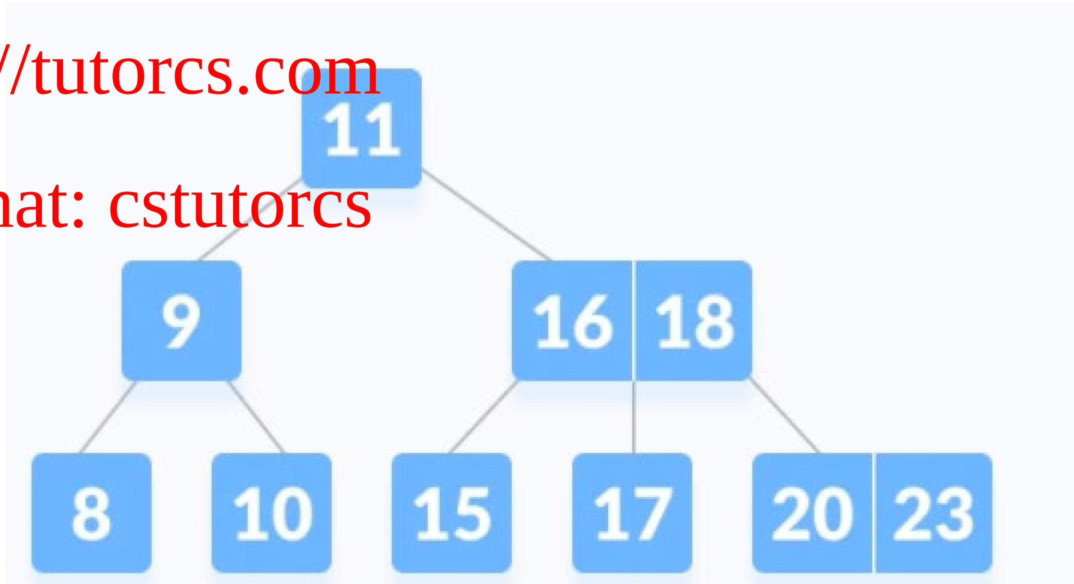
B-Tree Terminology

- Search:
  - If we are looking for 90 then as 90 < 100 search will go to the left subtree automatically
    - Will then go to right most node as 90 > 65



B-Tree Terminology

- Search for key term 17



B-Tree

- Commercial systems (including ORACLE) implement indexes with B-Tree
- Have the additional constraint that they be ***balanced***
  - Height of the left and right subtree of any node differ by not more than 1.
- They contain pointers to data records.

Indexes in Action

Oracle

Index in Action

- Suppose we are storing information about employees in a table:
  - We want to retrieve a list based on the EmployeeID column.
  - Without having an index on the EmployeeID column, the database will scan all the table rows to retrieve the requested data.

Index in Action

- Suppose we are storing information about employees in a table:
  - If we create an index on the EmployeeID column on the Employees table and perform a search based on the EmployeeID
    - The database will seek for the requested EmployeeID values in the index and use that index to locate the rest of the employees' information from the related rows in the source table
  - Provides a significant performance enhancement and reducing the effort required to locate the requested data

Index in Action

- Indexes are schema objects that are logically and physically separate from the data in objects with which they are associated
  - Therefore we can delete an index without impacting the stored data

Index in Action

- Indexes do not affect how you write your SQL statements
- An index is a fast access path to a single row of data. It affects only the speed of execution.
  - Given a data value that has been indexed, the index points directly to the location of the rows containing that value.

Index in Action

- The database automatically maintains and uses indexes after they are created.
- The database also automatically reflects changes to data, such as adding, updating, and deleting rows, in all relevant indexes with no additional actions required by users.
- Retrieval performance of indexed data remains almost constant, even as rows are inserted.
- However, the presence of many indexes on a table degrades DML performance because the database must also update the indexes.

Index in Action

- **Note**: Primary and unique keys automatically have indexes, but you might want to create an index on a foreign key or some non-key attributes.

Index in Action

- A key is a set of columns or expressions on which you can build an index.
- Although the terms are often used interchangeably, indexes and keys are different.
- Indexes are structures stored in the database that users manage using SQL statements.
- Keys are strictly a logical concept.

- The following statement creates an index on the customer_name column of the table orders:


```
CREATE INDEX ord_customer_ix ON
orders (customer_id);
```

- customer_id is a foreign key in the orders table

Index in Action

- A **composite index**, also called a concatenated index, is an index on multiple columns in a table.
- Columns in a composite index should appear in the order that makes the most sense for the queries that will retrieve data and need not be adjacent in the table.

- Composite indexes can speed retrieval of data for SELECT statements in which the WHERE (predicate) clause references all or the leading portion of the columns in the composite index.

- Therefore, the order of the columns used in the definition is important.
  - In general, the most commonly accessed columns go first.

Index in Action

- For example, suppose an application frequently queries the last_name, job_id, and salary columns in the employees table.
- Also assume that last_name has high cardinality, which means that the number of distinct values is large compared to the number of table rows.
- You create an index with the following column order:


```
CREATE INDEX employees_ix
   ON employees (last_name,
job_id, salary);
```


- Queries that access all three columns, only the last_name column, or only the last_name and job_id columns use this index.
- Queries that do not access the last_name column do not use the index.

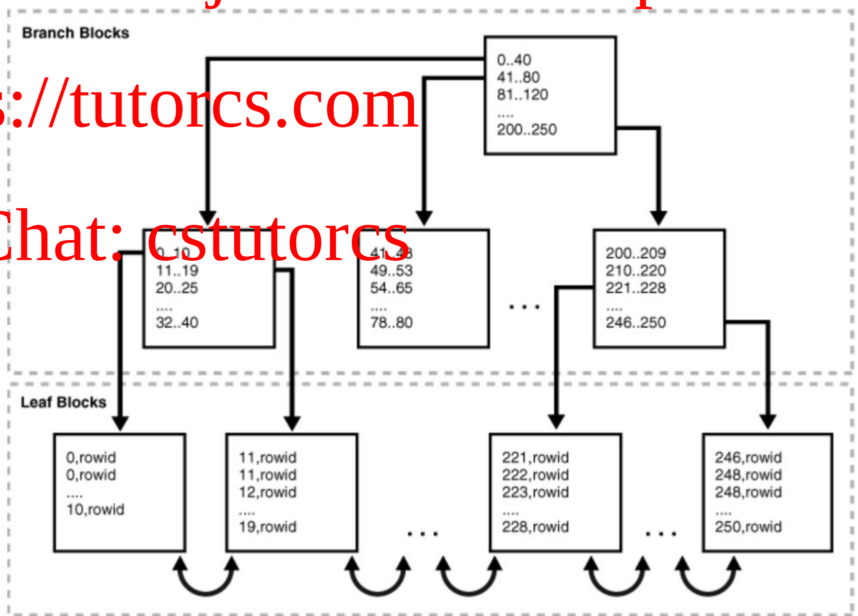
Index in Action

- Multiple indexes can exist for the same table if the permutation of columns differs for each index.
- You can create multiple indexes using the same columns if you specify distinctly different permutations of the columns.
- For example, the following SQL statements are both valid:

```
CREATE INDEX employee_idx1 ON
employees (last_name, job_id);
CREATE INDEX employee_idx2 ON
employees (job_id, last_name);
```

Index in Action – Oracle Indexing

- B-tree indexes
  - Standard index type.
  - Excellent for primary key and highly-selective indexes.
  - Used as concatenated indexes, can retrieve data sorted by the indexed columns.
- Contains **branch blocks** for searching **and leaf blocks** that store values.

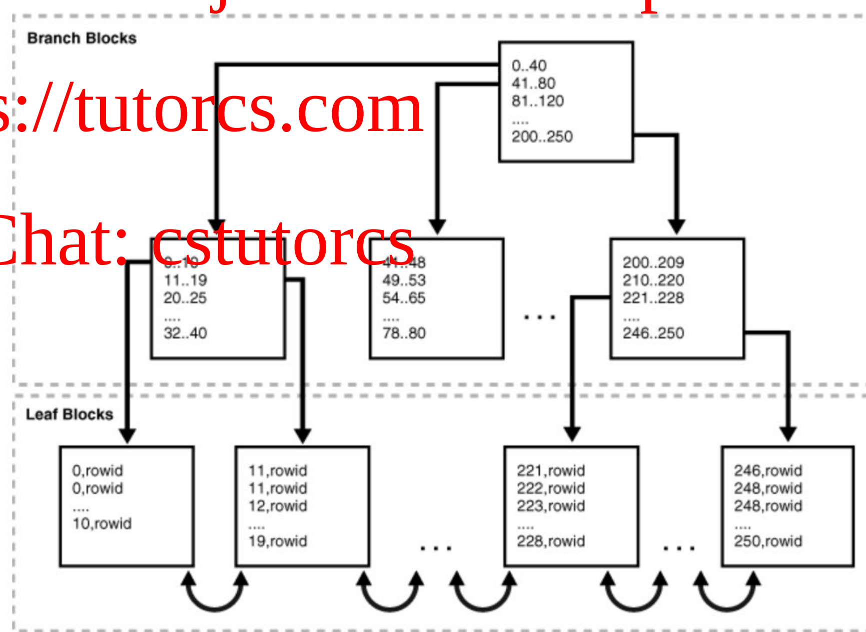


Description of "Figure 3-1 Internal Structure of a B-tree Index"

Source: Oracle Docs (docs.oracle.com)

Index in Action – Oracle Indexing

- The upper-level branch blocks of a B-tree index contain index data that points to lower-level index blocks.
- In Figure 3-1, the root branch block has an entry 0-40, which points to the leftmost block in the next branch level.
- This branch block contains entries such as 0-10 and 11-19.
- Each of these entries points to a leaf block that contains key values that fall in the range.

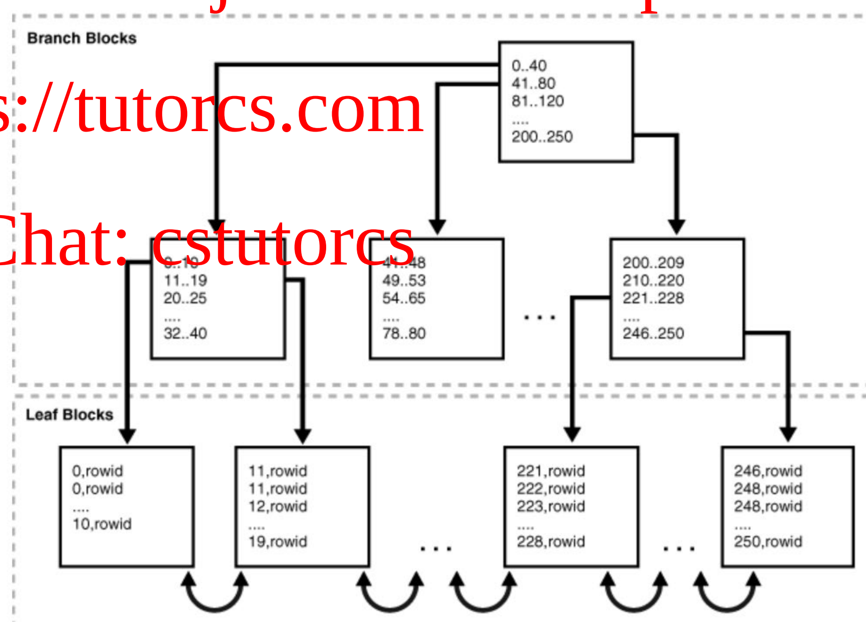


Description of "Figure 3-1 Internal Structure of a B-tree Index"

Source: Oracle Docs (docs.oracle.com)

Index in Action – Oracle Indexing

- The leaf blocks contain every indexed data value and a corresponding rowid used to locate the actual row.
- Each entry is sorted by (key, rowid).
- Within a leaf block, a key and rowid is linked to its left and right sibling entries.
- The leaf blocks themselves are also doubly linked.
- In Figure 3-1 the leftmost leaf block (0-10) is linked to the second leaf block (11-19).



Description of "Figure 3-1 Internal Structure of a B-tree Index"

Source: Oracle Docs (docs.oracle.com)

Index in Action – Oracle Indexing

- In an **index scan**, the database retrieves a row by traversing the index, using the indexed column values specified by the SQL statement.
- If the database scans the index for a value, then it will find this value in *n* I/Os where *n* is the height of the B-tree index.
  - This is the basic principle behind Oracle Database indexes.

Source: Oracle Docs (docs.oracle.com)

Index in Action – Oracle Indexing

- If a SQL statement accesses only indexed columns, then the database reads values directly from the index rather than from the table.
- If a SQL statement accesses columns in addition to the indexed columns, then the database uses rowids to find the rows in the table.
  - Typically, the database retrieves table data by alternately reading an index block and then a table block.

Source: Oracle Docs (docs.oracle.com)

Index in Action – Oracle Indexing

- In a **full index scan**, the database reads the entire index in order.
- A full index scan is available if a WHERE clause in the SQL statement references a column in the index,
  - Also can be done in some circumstances when no WHERE clause is specified.
- A full scan can eliminate sorting because the data is ordered by index key.

Source: Oracle Docs (docs.oracle.com)

Index in Action – Oracle Indexing

```
SELECT department_id, last_name, salary
FROM   employees
WHERE  salary > 5000
ORDER BY department_id, last_name;
```

- Suppose department_id, last_name, and salary are a composite key in an index.
- Oracle performs a full scan of the index, reading it in sorted order (ordered by department ID and last name) and filtering on the salary attribute.
- This means that the database scans a set of data smaller than the employees table, which contains more columns than are included in the query, and avoids sorting the data.
- E.g. may read index entries

- `50,Atkinson,2800,rowid`
- `60,Austin,4800,rowid`
- `70,Baer,10000,rowid`
- `80,Abel,11000,rowid`
- `80,Ande,6400,rowid`
- `110,Austin,7200,rowed`
- …

Source: Oracle Docs (docs.oracle.com)

Index in Action – Oracle Indexing

- A **fast full index scan** is a full index scan in which the database accesses the data in the index itself without accessing the table, and the database reads the index blocks in no particular order.
- Fast full index scans are an alternative to a full table scan when both of the following conditions are met:
  - The index must contain all columns needed for the query.
  - A row containing all nulls must not appear in the query result set.
    - For this result to be guaranteed, at least one column in the index must have either:
      - A NOT NULL constraint
      - A predicate applied to it that prevents nulls from being considered in the query result set

Source: Oracle Docs (docs.oracle.com)

Index in Action – Oracle Indexing

```
SELECT last_name, salary
FROM   employees;
```

Suppose the last_name column has a not null constraint.

If the last name and salary are a composite key in an index, then a fast full index scan can read the index entries to obtain the requested information:


```
Baida,2900,rowid
Zlotkey,10500,rowid
Austin,7200,rowid
Baer,10000,rowid
Atkinson,2800,rowid
Austin,4800,rowid
.
.
.
```

Source: Oracle Docs (docs.oracle.com)

Index in Action – Oracle Indexing

- An index range scan is an ordered scan of an index that has the following characteristics:
  - One or more leading columns of an index are specified in conditions.
    - A condition specifies a combination of one or more expressions and logical (Boolean) operators and returns a value of TRUE, FALSE, or UNKNOWN.
    - 0, 1, or more values are possible for an index key.
- The database commonly uses an index range scan to access selective data.

Source: Oracle Docs (docs.oracle.com)

Index in Action – Oracle Indexing

- The selectivity is the percentage of rows in the table that the query selects, with 0 meaning no rows and 1 meaning all rows.
- Selectivity is tied to a query predicate, such as WHERE last_name LIKE 'A%', or a combination of predicates.
- A predicate becomes more selective as the value approaches 0 and less selective (or more unselective) as the value approaches 1.

Source: Oracle Docs (docs.oracle.com)

Index in Action – Oracle Indexing

- For example, a user queries employees whose last names begin with A.
- Assume that the last_name column is indexed, with entries as follows:

```
Abel,rowid
Ande,rowid
Atkinson,rowid
Austin,rowid
Austin,rowid
Baer,rowid
.
.
.
```


- The database could use a range scan because the last_name column is specified in the predicate and multiples rowids are possible for each index key.
- For example, two employees are named Austin, so two rowids are associated with the key Austin.

Source: Oracle Docs (docs.oracle.com)

Index in Action – Oracle Indexing

- An index range scan can be bounded on both sides, as in a query for departments with IDs between 10 and 40, or bounded on only one side, as in a query for IDs over 40.
- To scan the index, the database moves backward or forward through the leaf blocks.
- For example
  - A scan for IDs between 10 and 40 locates the first index leaf block that contains the lowest key value that is 10 or greater.
  - The scan then proceeds horizontally through the linked list of leaf nodes until it locates a value greater than 40.

Source: Oracle Docs (docs.oracle.com)