Improving Database Performance Indexes

Advanced Databases

Indexes

- Without indexes, every request to a table would lead to a full scan of the entirety of a table to find the relevant results.
  - With a large data set, this can be extremely slow
- Indexes provide information to your database engine about roughly where in the system the data you're looking for lives.

Indexes

- In order to generate efficient indexes we need an understanding of our data and how we’re trying to access it.
- We also need to be aware that indexes don’t come for free – every time you update the table you need also to update the index

Indexes

- Help to locate records in a DB file
- Are additional auxiliary access structures
- Typically provide either faster access to data or secondary access paths without effecting the physical storage of the data
- Are based on indexing field(s)

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
- Not all attributes can be directly indexed (but secondary access paths may be used)

Indexes

- 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

Indexes

- 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.

Indexes

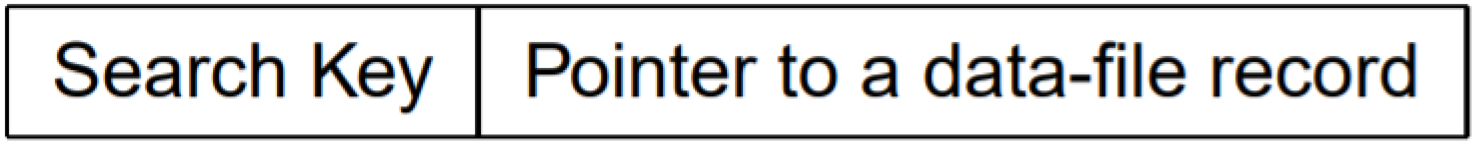
```
MovieStar(Name,Address,Gender,Birthdate)

SELECT *
FROM MovieStar
WHERE Name = ’Daniel Craig’;
```

- If there is no index on Name, all the data blocks for the MovieStar relation will be inspected

Indexes

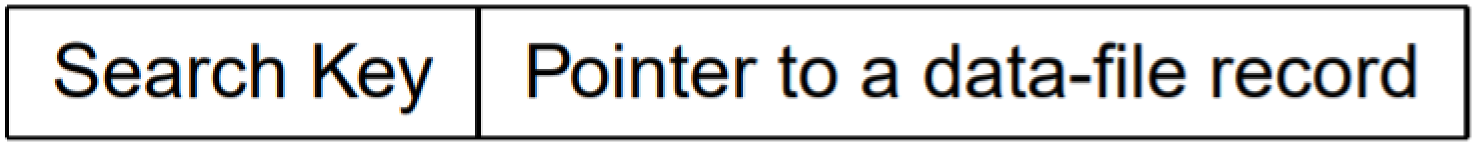
- An Index is a data structure that facilitates the query answering process by minimizing the number of disk accesses.
- An index structure is usually defined on a single Attribute of a Relation, called the **Search Key.**
- An Index takes as input a Search Key value and returns the address of the record(s)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



Search Key	Pointer to a data-file record

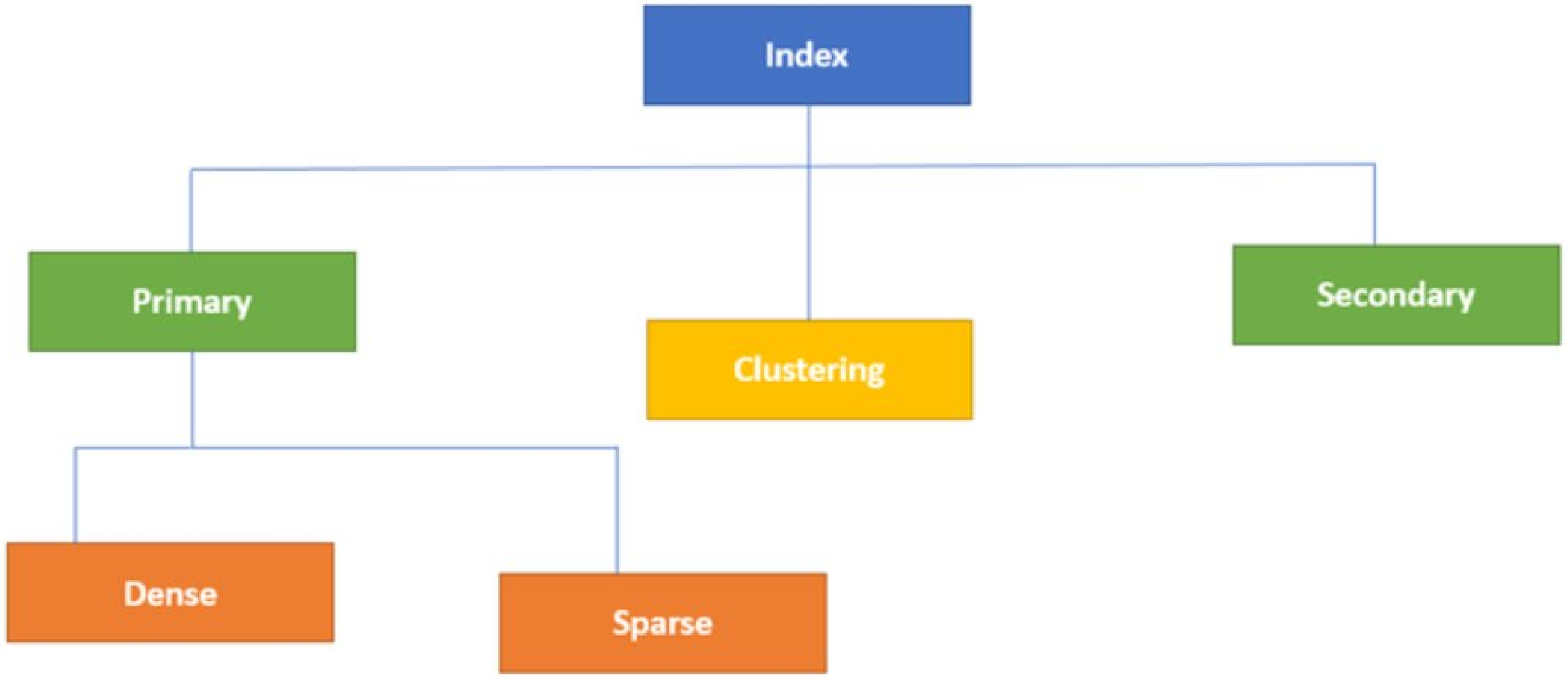
Indexes

- 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

Types of Index



Primary Index

- In the primary Index, there is always one to one relationship between the entries in the index table.
- 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

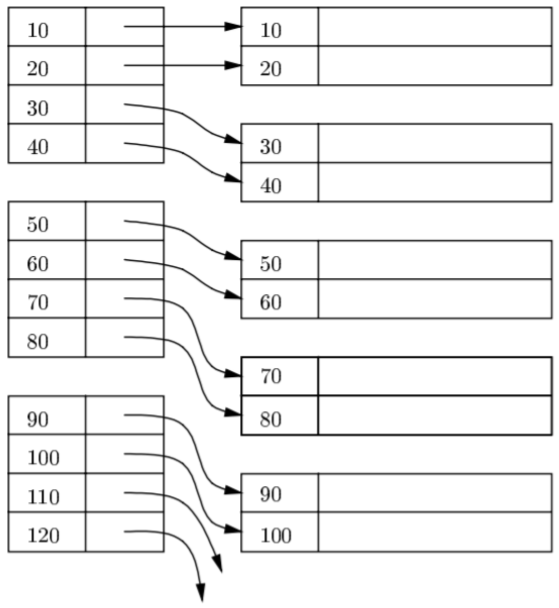
- Advantages
  - Number of blocks needed for storing the index is small compared to data
  - Index entries are smaller than data records
  - Can often be kept in buffer
- Disadvantages
  - Insertions and Deletions need to move data in storage and to update index entries affected by the shifts

Primary Index

- 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 – Dense Index

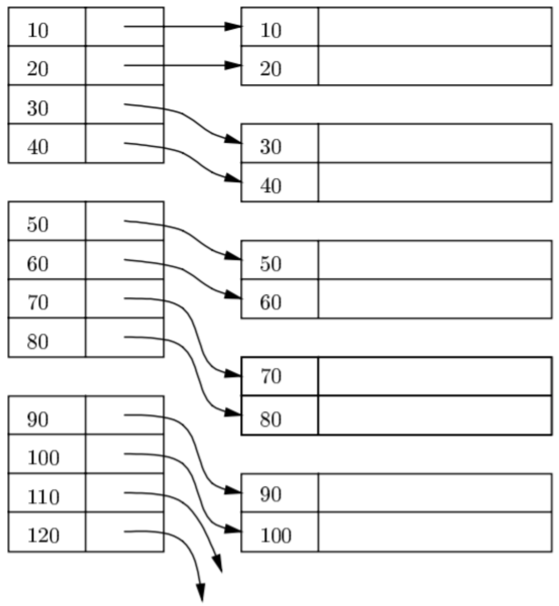
- Every value of the search key has a representative in a Dense Index.
- The index maintains the keys in the same order as in the data file.



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

Primary Index – 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.

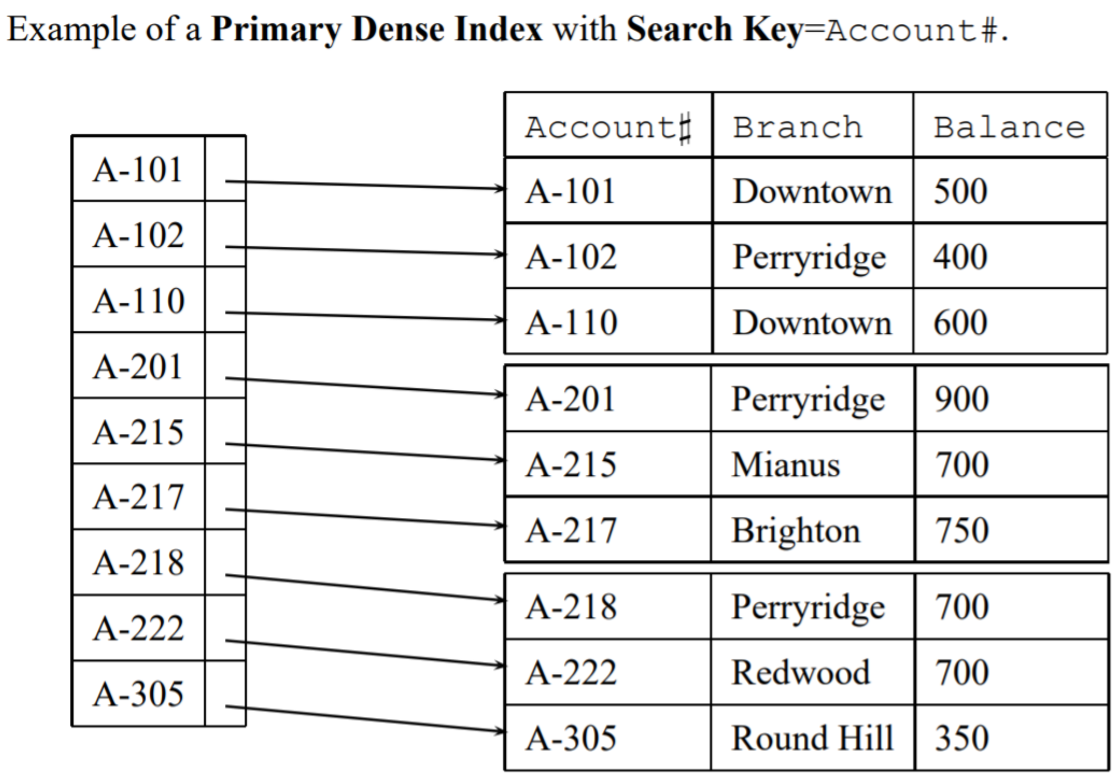


Database System Implementation, H. Garcia-Molina, J. Ullman, and J. Widom, Prentice-Hall, 2000.

Primary Index – Dense Index

- 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.

Primary Index – Dense Index


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

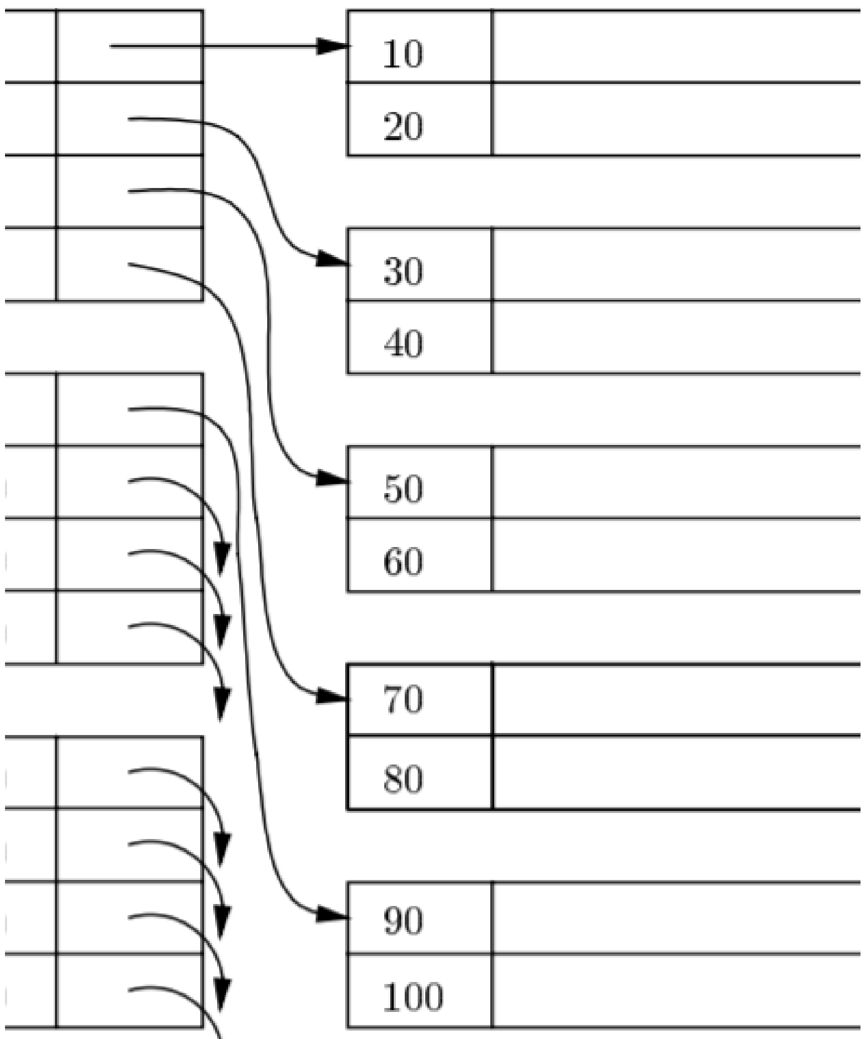
Account#	Branch	Balance
A-101	Downtown	500
A-102	Perryridge	400
A-110	Downtown	600
A-201	Perryridge	900
A-215	Mianus	700
A-217	Brighton	750
A-218	Perryridge	700
A-222	Redwood	700
A-305	Round Hill	350

Primary Index – Dense Index

- 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

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



lementation, H. Garcia-Molina, J. Ullman, and J. Widom,

Primary Index – Sparse Index

- Given a search key K:
  - 1. Search the sparse index for the greatest key <= K using binary search;
  - 2.Retrieve the pointed block to main memory to look for the record with search key K (always using binary search).

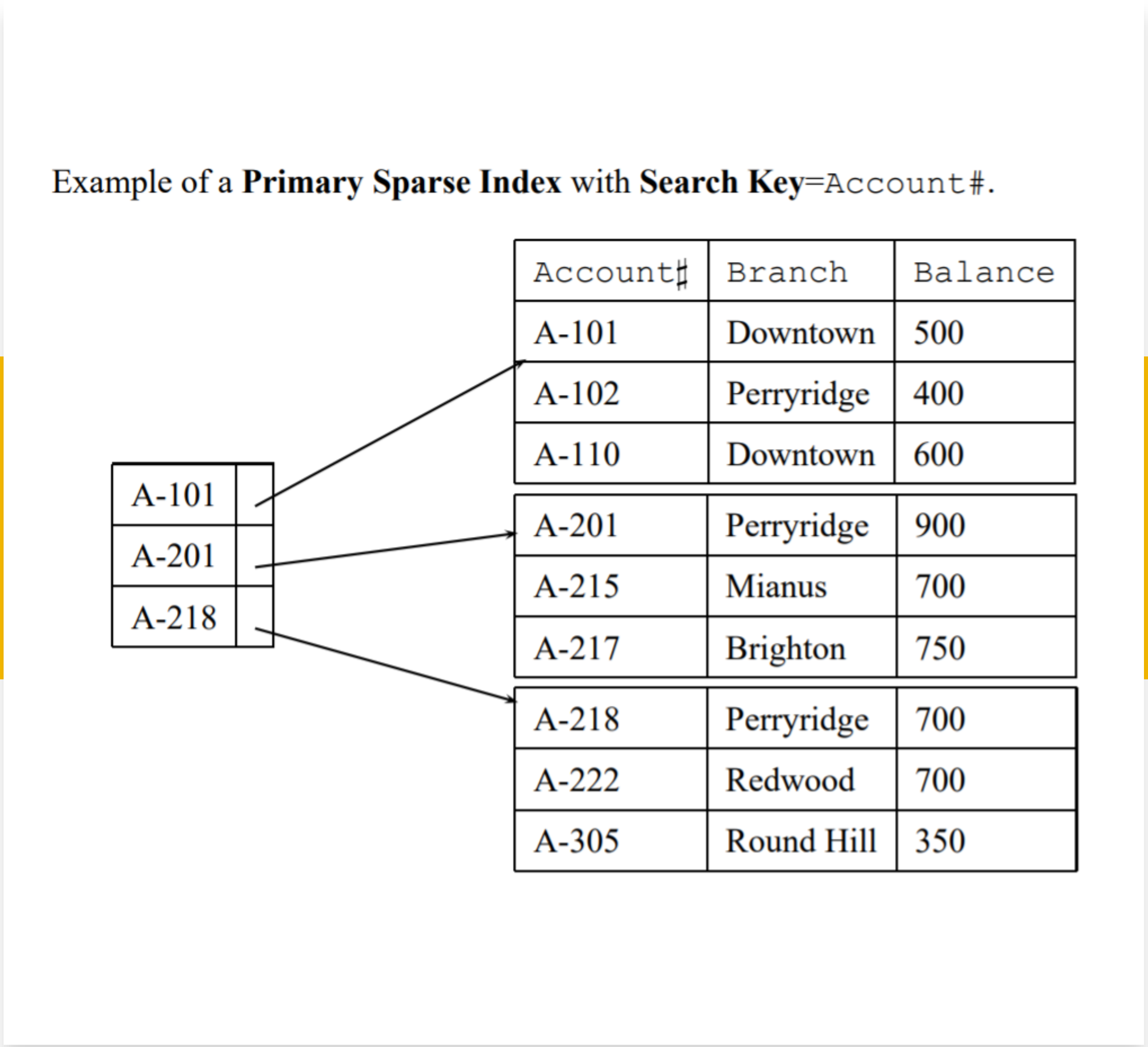
Primary Index – Sparse Index

- In comparison to a dense index two different binary searches are needed:
  - The first on the sparse index
  - The second on the retrieved data block.

Primary Index – Sparse Index

- A Sparse Index is more efficient in space than a dense index
- But it is at the cost of a poorer computing time in Main Memory

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.
- To 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 for 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 Index

- 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

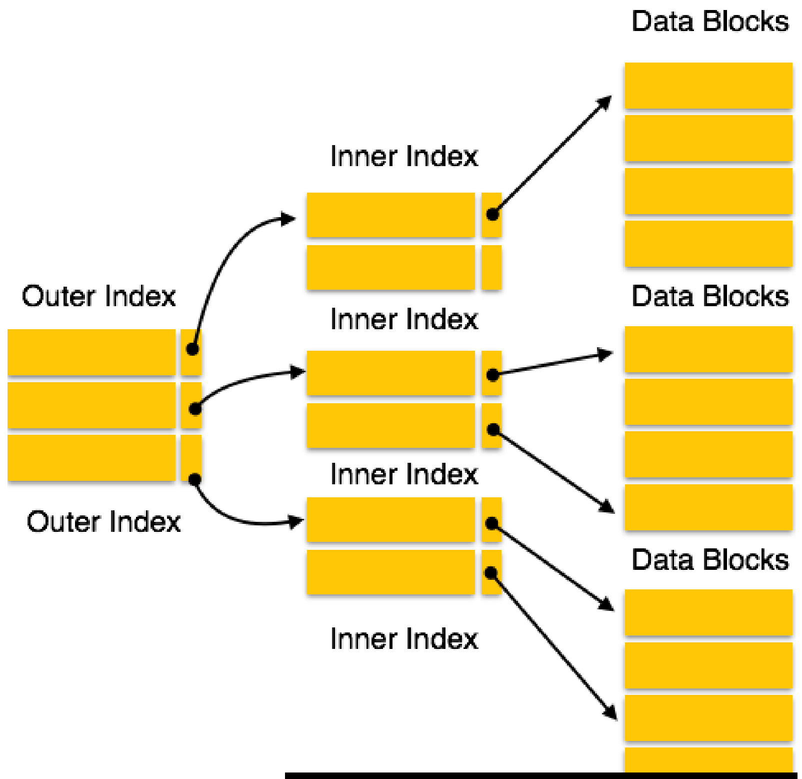
- **Clustering indexes** store data records in the order of a non-unique indexing field not pointers
- Reorders the way records are stored on disk based on key value as included in the index definition
- Sorts the data rows in the table on their key values. In the Database, there is only one clustered 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.

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

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



Indexes in Postgresql

Six Types of Index in PostgreSQL

- B-Tree
- Hash
- Generalized Inverted Index (GIN)
- Generalized Search Tree (GiST)
- General Index Framework for Space Partitioning Trees (SP-GiST)
- Block Range Index (BRIN)

- Called methods of indexing because they define the way each particular index handles its task (and each type requires a particular syntax)

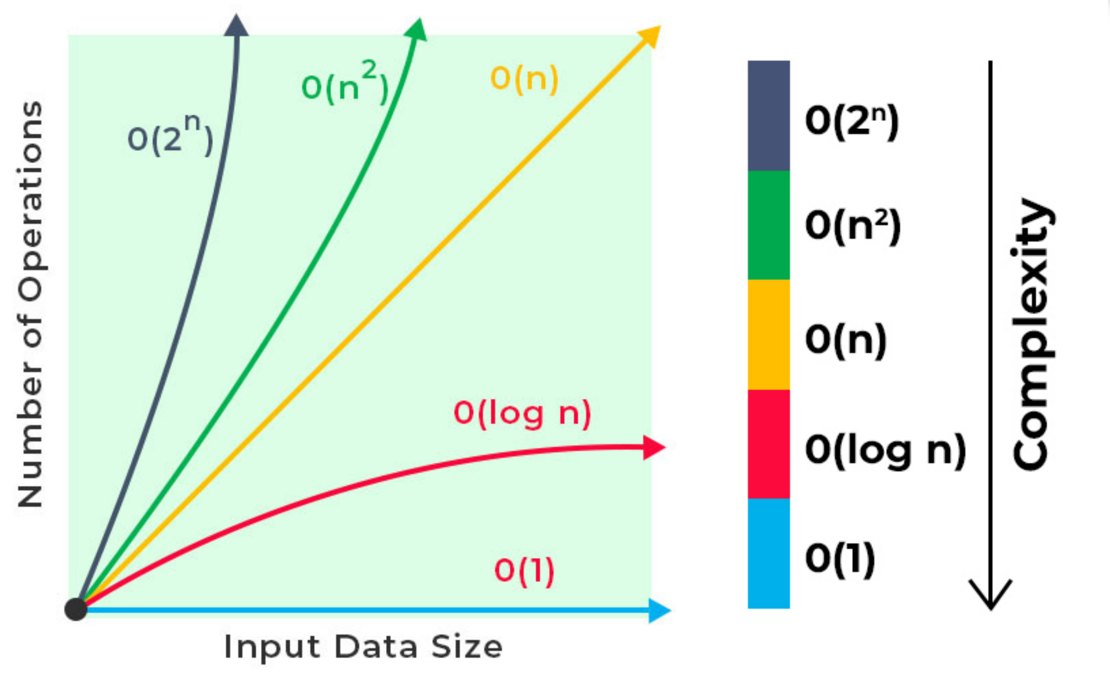
B-Tree (Balanced Tree)

- A B-tree is a data structure
- It keeps data sorted
- It is optimized for applications that read and write large blocks of data

B-Tree (Balanced Tree)

- It allows searches, insertions, and deletions in **logarithmic amortized time**
  - What does this mean?
  - It’s a measure of algorithmic time complexity
  - Amortized means average time taken per operation – used when the performance is very slow only very occasionally
  - Why logarithmic?
    - when the algorithm runtime increases very slowly compared to an increase in input size (the logarithm of input size) then the algorithm is said to exhibit logarithmic time complexity
  - *Remember Big O notation?*

B-Tree (Balanced Tree)



O(1)	Constant
O(log N)	Logarithmic
O(N)	Linear time
O(N^2)	Quadratic
O(2^N)	Exponential

B-Tree (Balanced Tree)

- B-Tree is the default index type for the CREATE INDEX command in PostgreSQL.
- It is compatible with all data types, and it can be used, for instance, to retrieve NULL values and work with caching.
- B-Tree is the most common index type, suitable for most cases.

B-Trees - Terminology

- Tree is formed of nodes
- Each node (except root) has one parent and zero or more child nodes
  - B-tree nodes have many more than two children.
  - A B-tree node may contain more than just a single element
- Leaf node has no child nodes
  - Tree is unbalanced if leaf nodes occur at different levels
- Non-leaf node is called an internal node

B-Trees - Terminology

- 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 - Rules

B-Tree is a self-balanced search tree in which every node contains multiple keys and has more than two children.

B-Trees - Rules

Each node in a B-Tree can have more than 2 children.

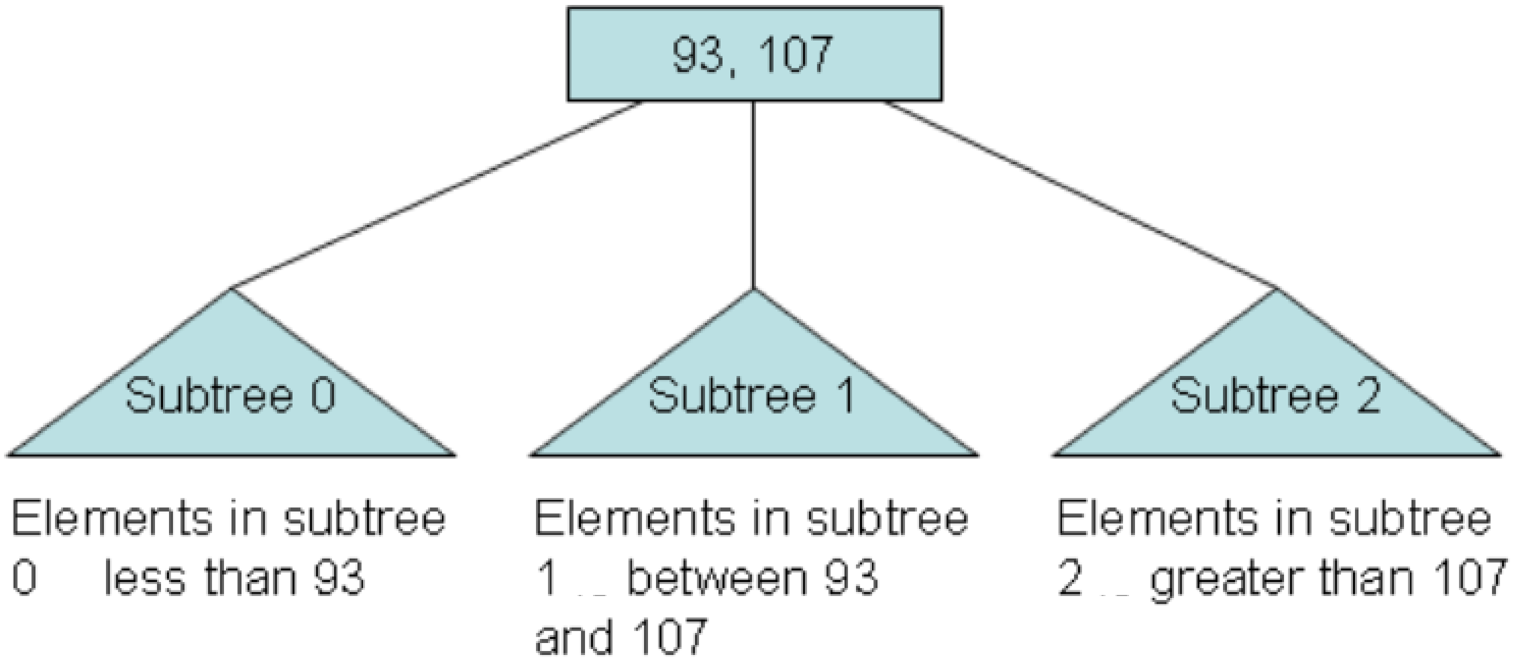
Each node can have up to m children, where m is called the tree’s “**order**”.

To keep the tree mostly balanced, we also say nodes have to have at least m/2 children (rounded up).

B-Trees - Rules

- Exceptions:
  - Leaf nodes have 0 children
  - The root node can have fewer than m children but must have at least 2
  - If the root node is a leaf node (the only node), it has 0 children

B-Trees


93, 107
Subtree 0
Subtree 1
Subtree 2
Elements in subtree 0 less than 93
Elements in subtree 1 between 93 and 107
Elements in subtree 2 greater than 107

B-Trees

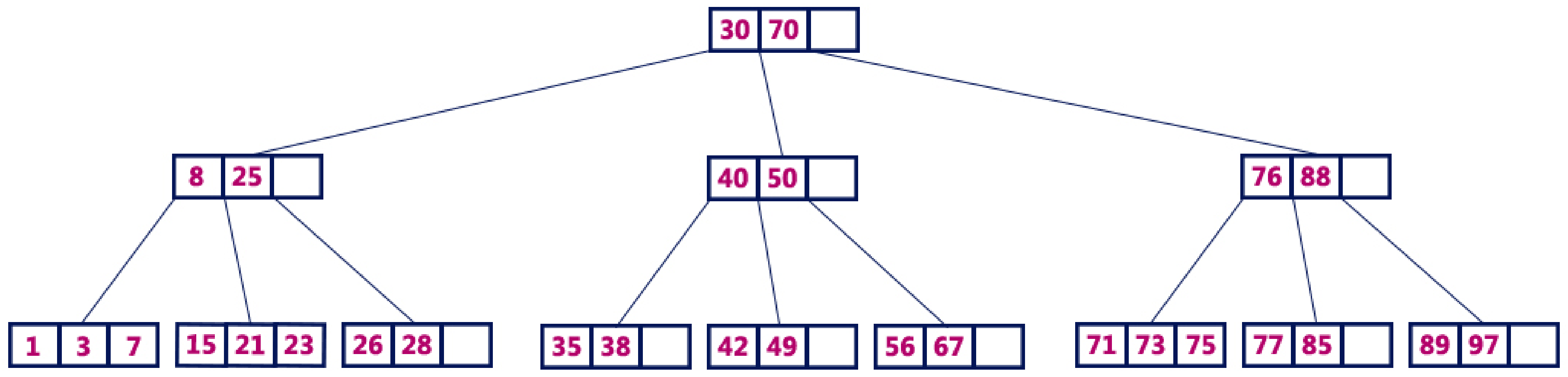
Every child of a node could also be the root of a smaller B-tree

B-Trees Properties

- All leaf nodes must be at same level.
- All nodes except root must have at least [m/2]-1 keys and maximum of m-1 keys.
- All non leaf nodes except root (i.e. all internal nodes) must have at least m/2 children.
- If the root node is a non leaf node, then it must have at least 2 children.
- A non leaf node with n-1 keys must have n number of children.
- All the key values in a node must be in Ascending Order.

B-Tree

B-Tree of Order 4



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

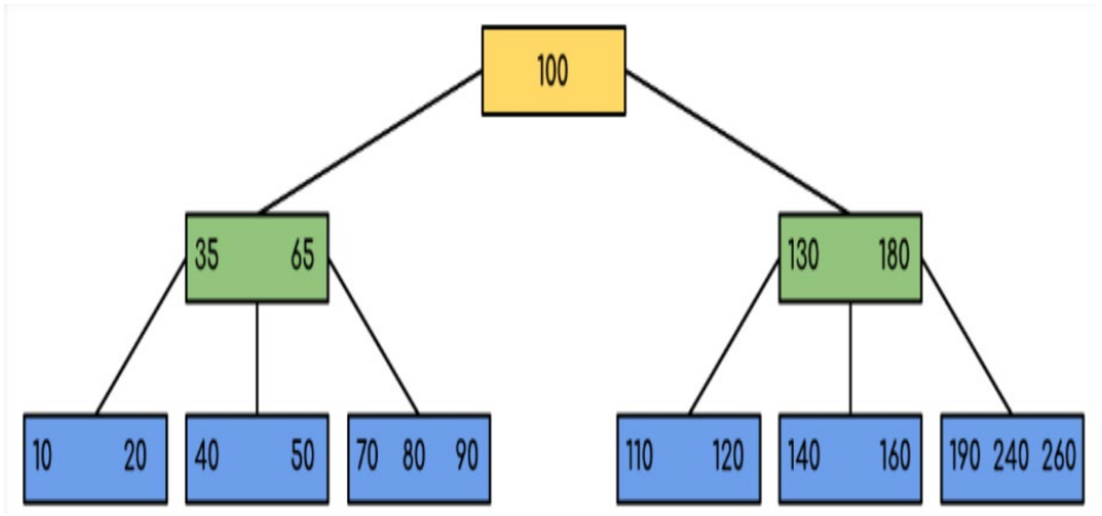
B-Tree - Search

- Recursive search, starting at the root and ending at a leaf
- If we are conducting a search for an element k:

1. Compare the search element with first key value of root node in the tree.
2. If they match we have completed the search and can terminate
3. If not matched, then check whether search element is smaller or larger than that key value.
    1. If search element is smaller, then continue the search process in left subtree.
    2. If search element is larger, then compare the search element with next key value in the same node and repeat until we find the exact match or until the search element is compared with last key value in the leaf node.
4. If the last key value in the leaf node is also not matched then the search has not found the element and can terminate

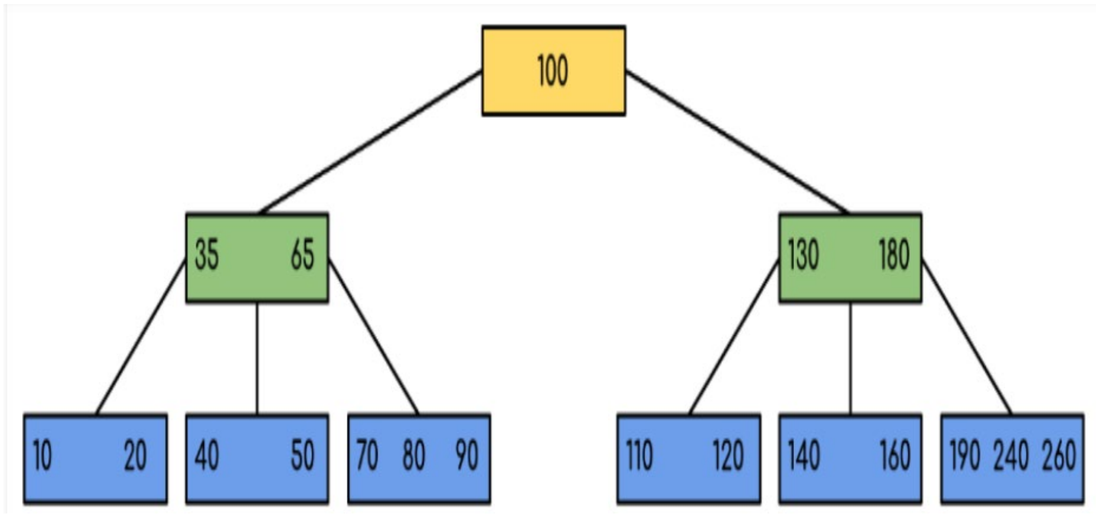
B-Trees - Search

- Example:
- 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.



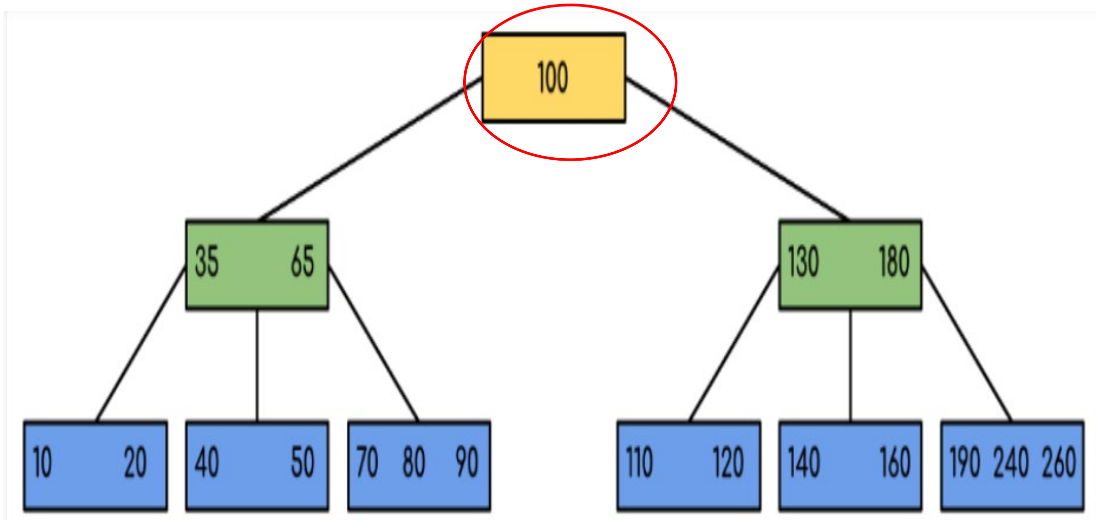
B-Trees - Search

- 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, recurse 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.



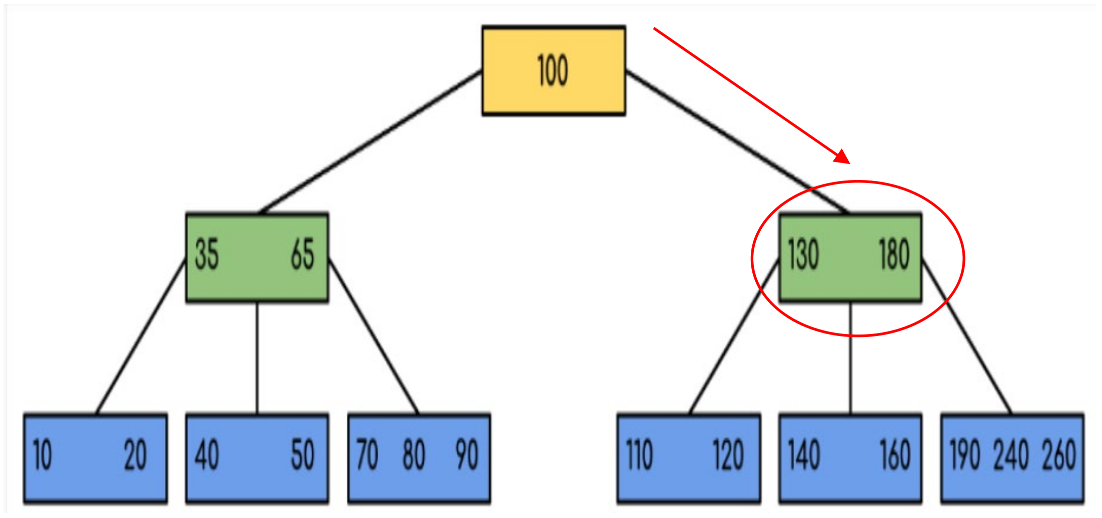
B-Trees - Search

- Let the key to be searched be 120 (k=120)
- Step 1: First search will start with the root node.
    - Not found



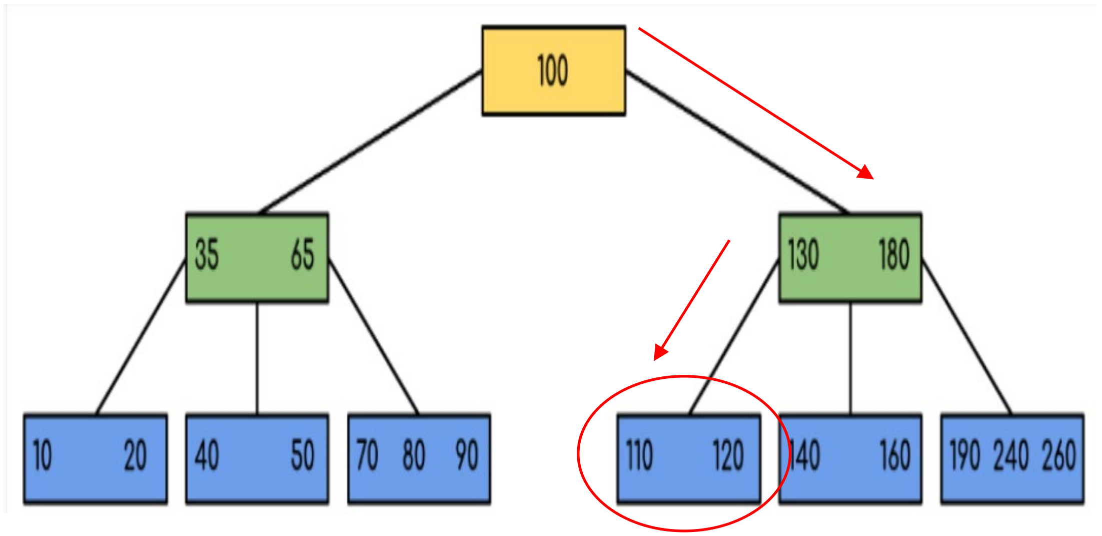
B-Trees - Search

- Step 2: Search will now 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 right branch if it is there at all



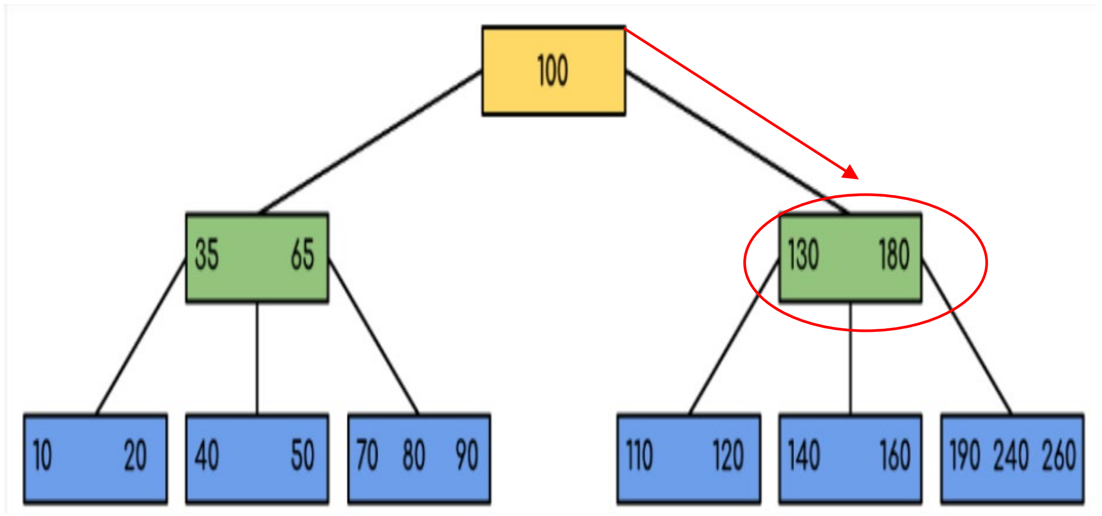
B-Trees - Search

- Step 3: As 120 is less than 130 it will be in the left most node (if it exists at all)



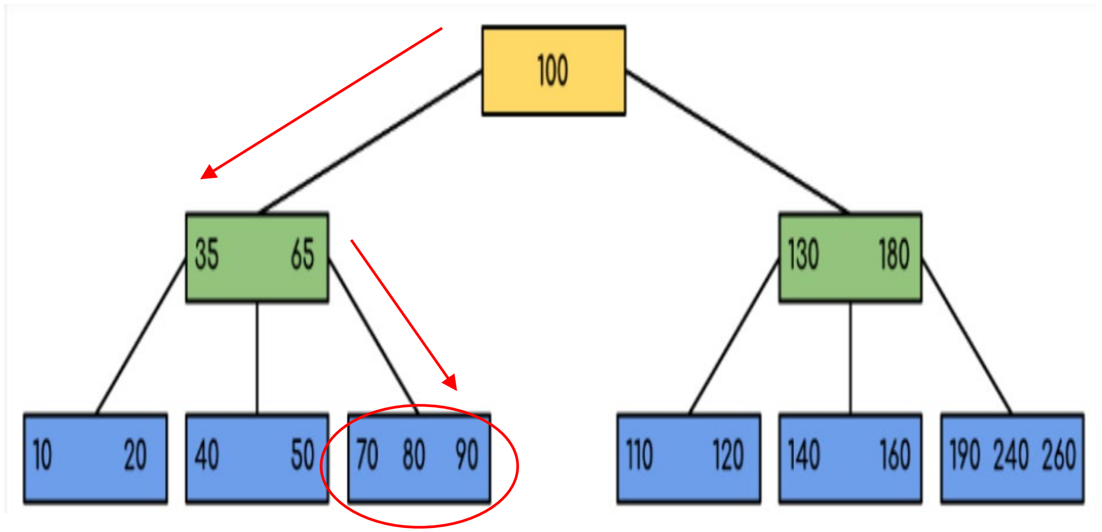
B-Tree Search

- Searching for k=180
- If we were searching 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 Search

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



B-Tree Insert

- Check whether tree is Empty.
- If tree is Empty, then create a new node with new key value and insert it into the tree as a root node.
- If tree is Not Empty, then find the suitable leaf node to which the new key value is added using Binary Search Tree logic.
- If that leaf node has an empty position, add the new key value to that leaf node in ascending order of key value within the node.
- If that leaf node is already full, split that leaf node by sending the middle value to its parent node. Repeat the same until the sending value is fixed into a node.
- If the splitting is performed at the root node then the middle value becomes new root node for the tree and the height of the tree is increased by one.

B-Tree

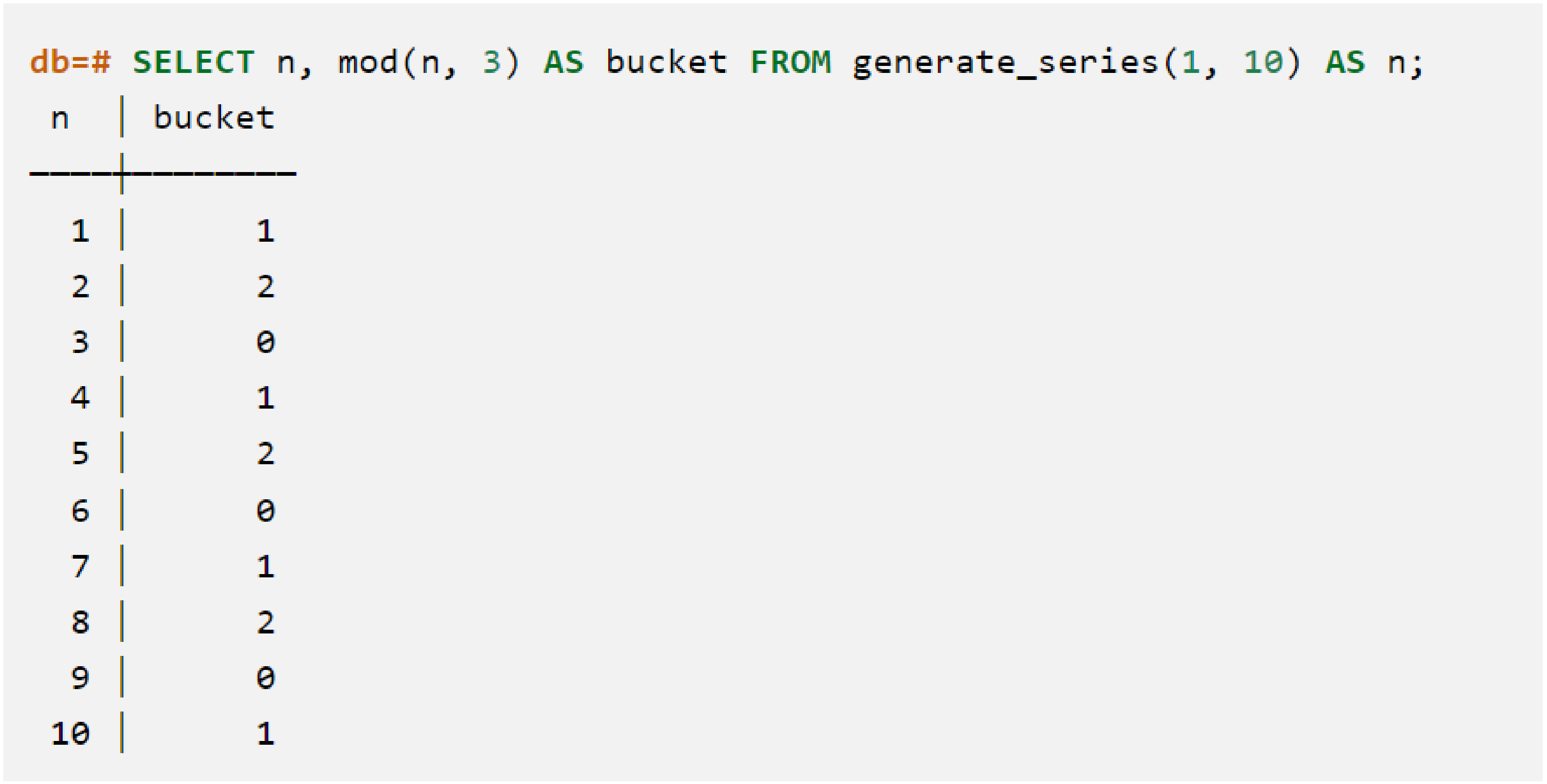
- Commercial systems (including PostgreSQL and POSTGRES) implement indexes with B-Tree
- Have the additional constraint that they be ***balanced***
  - Height of the left and right subtree of any node differ by not more than 1.
- They contain pointers to data records.

Hash

- Use a form of the hash table data structure
  - Similar to Dict in Python, Hashmap in Java
- Use a hash function.
  - In PostgreSQL the **hash function** maps any database value to a 32-bit integer, the hash code (about 4 billion possible hash codes).
- A good hash function can be computed quickly and "jumbles" the input uniformly across its entire range.
- The hash codes are divided to a limited number of **buckets**.
  - Buckets map the hash codes to the actual table rows

Hash

- Example
- For integers, a simple hash function is modulo – the remainder is the hash code
- To divide values across 3 buckets you can use the hash function mod(3):



```
db=# SELECT n, mod(n, 3) AS bucket FROM generate_series(1, 10) AS n;
 n  | bucket
----+--------
  1 |      1
  2 |      2
  3 |      0
  4 |      1
  5 |      2
  6 |      0
  7 |      1
  8 |      2
  9 |      0
 10 |      1
```

Hash

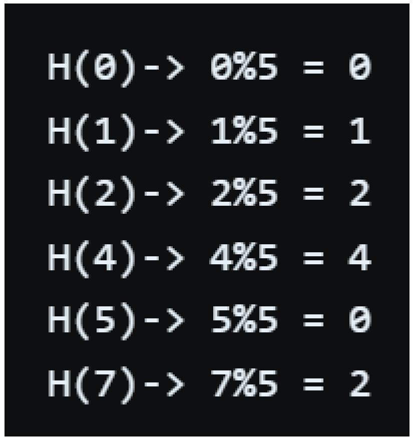
- To add a new value to the index
  - PostgreSQL applies the hash function to the value
  - PostgreSQL puts the hash code and a pointer to the tuple in the appropriate bucket.
- For our integers using the hash function mod(3), if you insert the value 5 the index entry will be added to bucket 2, because 5 % 3 = 2.

Hash Collision

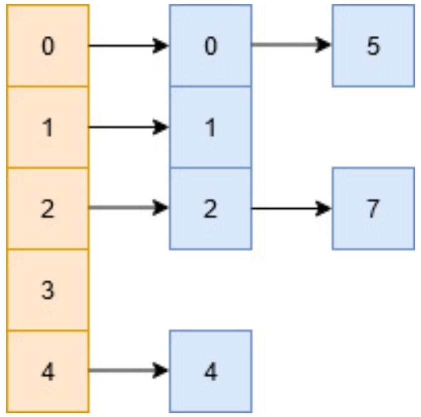
- Multiple values can map to the same bucket
- This is called a **collision**
- For the integers hash function mod(3) returns the hash code 2 for the integers 2, 5 and 8
- The hash algorithm must be chosen carefully
- A poor algorithm can be a security risk

Hash Collision - Solutions

- Open Hashing (Separate Chaining)
  - Collisions are resolved using a list of elements to store objects with the same key together



```
H(0)-> 0%5 = 0
H(1)-> 1%5 = 1
H(2)-> 2%5 = 2
H(4)-> 4%5 = 4
H(5)-> 5%5 = 0
H(7)-> 7%5 = 2
```



Hash Collision - Solutions

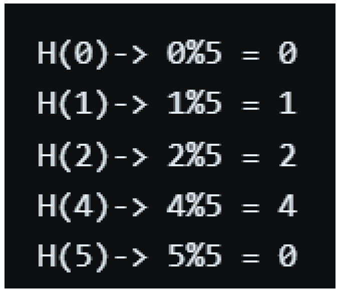
- Open Hashing (Separate Chaining)
  - Table size is obviously an issue
  - Lookups/inserts/updates can become linear [O(N)] instead of constant time [O(1)] if the hash function has too many collisions
  - It doesn't account for any empty slots which can be leveraged for more efficient storage and lookups.
  - Ideally we require a good hash function to guarantee even distribution of the values.

Hash Collision - Solutions

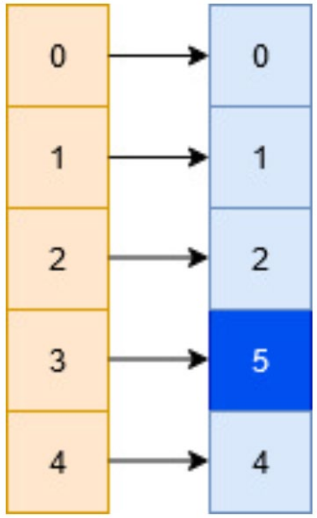
- Closed Hashing (Open Addressing)
  - Requires a hash table with fixed and known size
  - Approaches:
    - Linear Probing
    - Quadratic probing
    - Double hashing

Hash Collision - Solutions

- Linear Probing
- Take a fixed sized hash table and every time a hash collision occurs, linearly traverse the table in a cyclic manner to find the next empty slot



```
H(0)-> 0%5 = 0
H(1)-> 1%5 = 1
H(2)-> 2%5 = 2
H(4)-> 4%5 = 4
H(5)-> 5%5 = 0
```



Hash Collision - Solutions

- Quadratic probing
- Look at the Q(i) increment at each iteration when looking for an empty bucket, where Q(i) is some quadratic expression of i

Hash Collision - Solutions

- Double hashing
  - In the event of a collision use an another hashing function (H2) with the key value as an input to find where in the open addressing scheme the data should actually be placed at.
  - A good H2 is a function which never evaluates to zero and ensures that all the cells of a table are effectively traversed.

Hash – When to use

- When querying specific inputs with specific attributes.
- Best optimized for SELECT and UPDATE-heavy workloads that use equality scans on larger tables.

GIN Index

- Generalized Inverted Index
- Designed for
  - Handling cases where the items to be indexed are composite values
  - And
  - The queries to be handled by the index need to search for element values that appear within the composite items.
- Preferred index for JSONB, array and text search
- Focus is on lexical units (lexemes)
- As inverted indexes, they contain an index entry for each word (lexeme), with a compressed list of matching locations.
- Multi-word searches can find the first match, then use the index to remove rows that are lacking additional words

GiST Index

- Generalized Search Tree
- Build a search tree inside your database and are most often used for spatial databases and full-text search use cases

SP GiST Index

- General Index Framework for Space Partitioning Tree
- Again suitable for spatial databases

BRIN Index

- Block range index
- Specially targeted at very large datasets where the data you're searching is in blocks, like timestamps and date ranges.
- They are known to be very performance and space efficient.

Indexes – When NOT to use

You should not use indexes on small tables.

You should not use indexes on tables that face large and frequent batch UPDATE and INSERT operations.

You should not use indexes on columns that have many NULL values.

You should not use indexes on columns that are frequently edited.

Indexes – What are we going to do with them?

You are going to use EXPLAIN ANALYSE to examine some queries and read through the query plan

You are then going to implement some indexes to try to improve performance (examining query plans that result)