Indices and partitions are two of the most powerful tools a db admin can use to improve the performance of a database.

Index

-An index is a database object that improves the access of data by “indexing” the tuples stored on disc. Maintaining an index is fundamental for good performance and requires performance tuning by the db admin.

-In the presence of the correct type of index, Accessing data becomes better, but DML operations become worse, and do not benefit from indices.

-An index can be placed on a single column or multiple columns in the same table.

-Indices will be used by the query optimizer and SQL processor to produce the execution plan for the SQL query. The optimizer will first check if the column being filtered has an index or not, if it has an index, it will decide to go inside the index file, and not do a sequential search on disk.

Indexes can have multiple classifications:

Implicit or Explicit

1-Implicit: the user does not explicitly create the index object using a SQL command, e.g. Primary key Index, or UNIQUE column index.

2-explicit: the user explicitly writes a SQL command to create the index on explicitly specified columns.

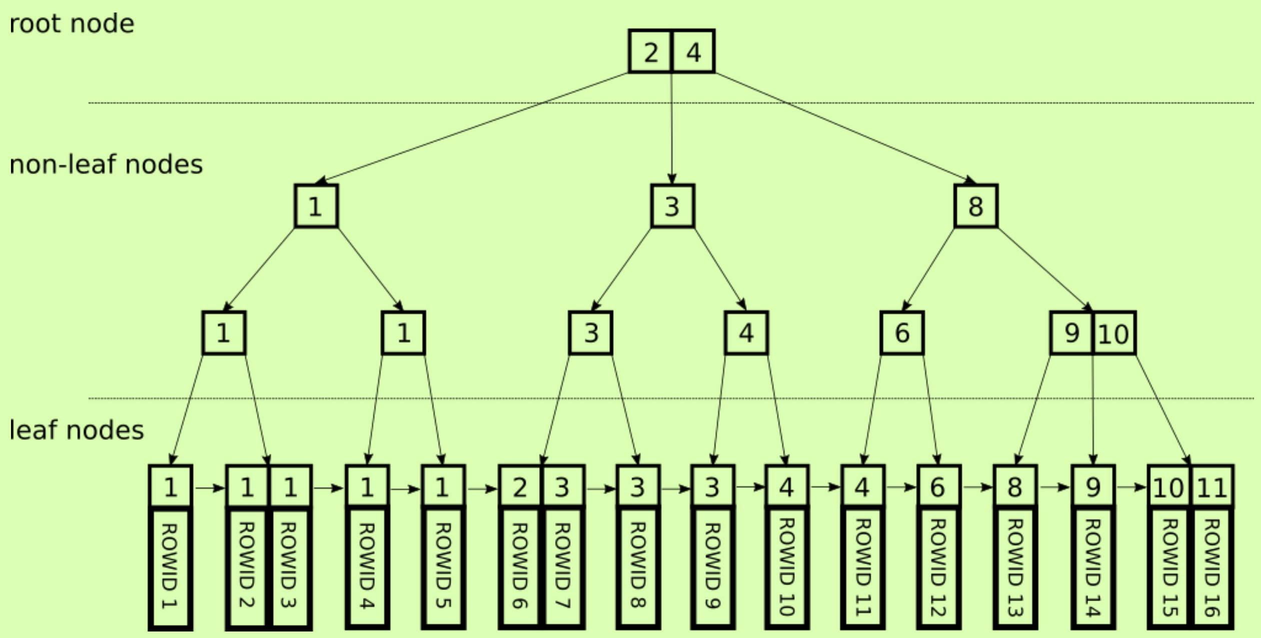
Clustered or Non-Clustered

1-Non-clustered: an index file is created out of the specified columns, keeping pointers to where each combination of values in those columns are located on disk. The data on disk however, is not sorted

2-Clustered: No index file is created, but the data is actually sorted on disk. faster access, because the executor does not need to go inside an index file

Access Methods

1-B-Tree



Typically a self balancing B-Tree, where a single node can have n values, and n+1 children. This is in the order of O(log n) for searching

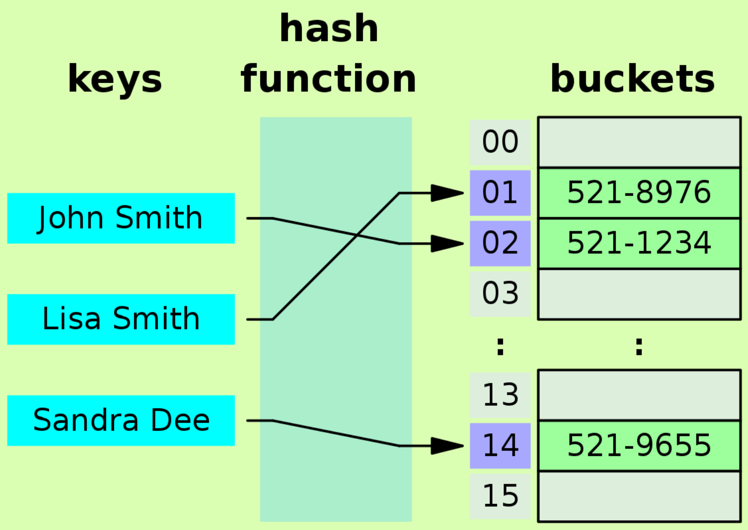
-supports unique index, and prefers them. because by nature, Tree data structures do not allow duplicate values.

-Because tree data structures are always sorted, they have a sense of order, and thus support comparison operators in the WHERE clause filters. i.e. **>, >=, <, <=, =, BETWEEN, IS,** and **IN**

-can also be used for pattern matching.

-In fact, this is the best for the above operators

2-Hash



-Uses a hash table/map data structure. A hash function will be applied to the key and used to find a place to store in the index file. and thus when accessing, the filter value will be hashed, and the hash value will be used to immediately access the place in memory without searching. O(1) (neglecting the hash function itself)

-Because this hashes the value, it can only be used for the **=** operator in the WHERE clause, and has no sense of order.

3-Block Range Index **brin**

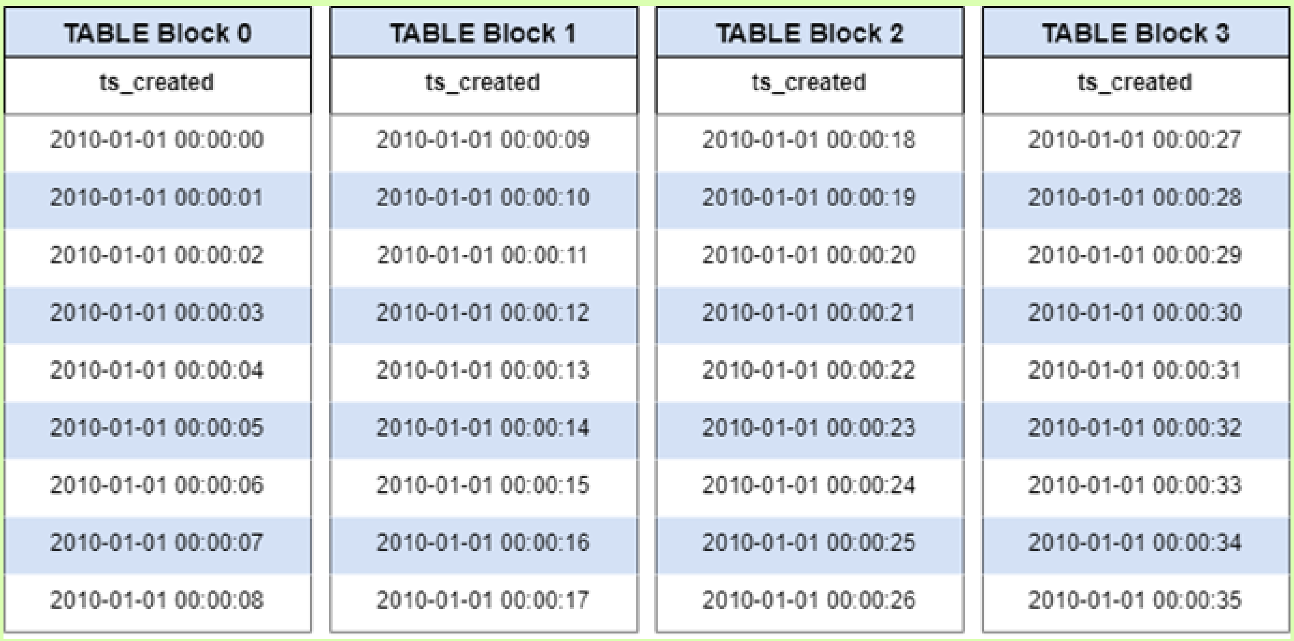


TABLE Block 0	TABLE Block 1	TABLE Block 2	TABLE Block 3
ts_created	ts_created	ts_created	ts_created
2010-01-01 00:00:00	2010-01-01 00:00:09	2010-01-01 00:00:18	2010-01-01 00:00:27
2010-01-01 00:00:01	2010-01-01 00:00:10	2010-01-01 00:00:19	2010-01-01 00:00:28
2010-01-01 00:00:02	2010-01-01 00:00:11	2010-01-01 00:00:20	2010-01-01 00:00:29
2010-01-01 00:00:03	2010-01-01 00:00:12	2010-01-01 00:00:21	2010-01-01 00:00:30
2010-01-01 00:00:04	2010-01-01 00:00:13	2010-01-01 00:00:22	2010-01-01 00:00:31
2010-01-01 00:00:05	2010-01-01 00:00:14	2010-01-01 00:00:23	2010-01-01 00:00:32
2010-01-01 00:00:06	2010-01-01 00:00:15	2010-01-01 00:00:24	2010-01-01 00:00:33
2010-01-01 00:00:07	2010-01-01 00:00:16	2010-01-01 00:00:25	2010-01-01 00:00:34
2010-01-01 00:00:08	2010-01-01 00:00:17	2010-01-01 00:00:26	2010-01-01 00:00:35

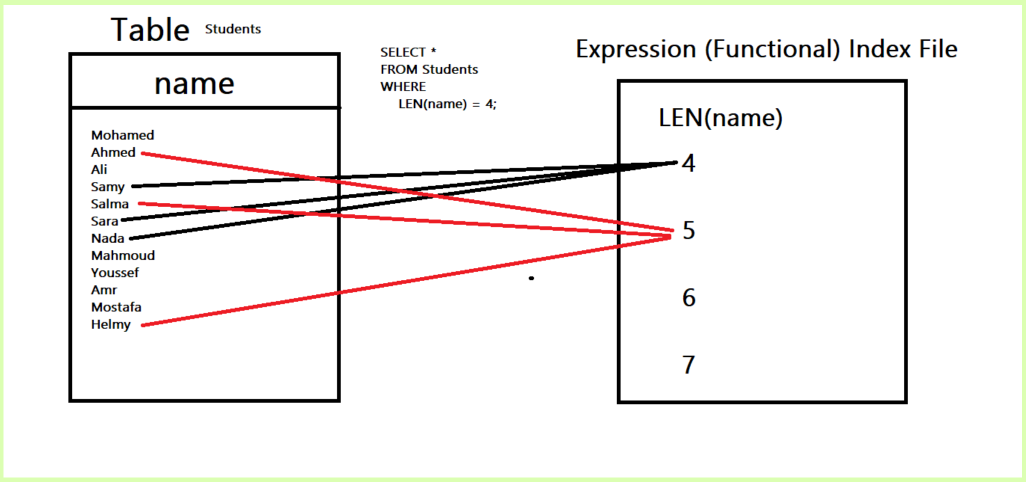
-This stores the minimum and maximum values located in each block and disk, and thus when a value is queried, can easily check the index file for the block on disk to search in, instead of searching sequentially in all blocks.

-By nature of storing min, and max, can support the comparison operators as well.

-This is not as good as B-tree when it comes to performance, but has a smaller size on disk, because it doesn't store all the values, just the min/max in each block. and has much smaller cost to maintain.

-best for dates.

4-Expression Index



-This is used when the values being filtered are not the actual values in the column, but the result of a function/expression using the columns as arguments. e.g. in the example above, storing a regular b-tree index pointing to the actual values on disk will not help when the required are names with a length < 4 characters.

-instead, the expression index will store the results of the expressions/functions in the index file, and point to the values in the column that led to this result.

→Although these seem incredibly useful, they are some of the most expensive indexes to maintain, because unlike a normal index, ALL values in the index file are going to be updated with each DML.

Indexes are useful, and can produce an incredible improvement in query performance. But they should not be used for all columns in all tables.

When do we decide to create an index?

- When a frequently executed query has a WHERE, GROUP BY, or JOIN clause on particular columns. to speed up this repetitive operation, we can index those columns.
- When the table has little NULL values.
- When the frequently executed query returns ~<10% of the table's tuples

When to not create an index?

- When the query returns >20% of the table's tuples.
- When the column has a lot of NULL values
- When the table is DML intensive.
- When the space on disk is limited because index files take up a lot of space on disk, especially if the table is large in size.

<u>Partitions</u>

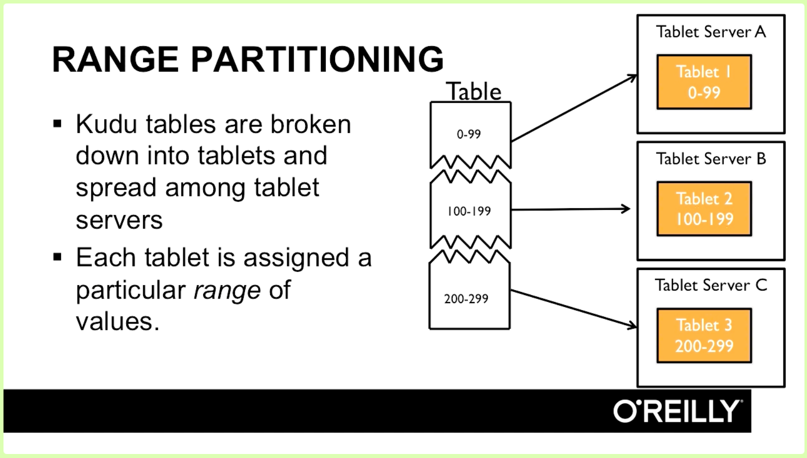
-As more rows are inserted into a table, a filter has to go through more rows in each query or DML, whether that is in a sequential or index search.

-And thus making the table smaller would improve the performance. How to make the table smaller? by splitting into multiple smaller tables.

-Most dbms support table inheritance, which gives the users an abstracted interface to deal with partitions. i.e. the user will interact with the main original table, and the query executor and optimizer will deal with the smaller partitions. the user does not need to interact with the partitions directly.

There are multiple basis to partitioning or "splitting" the data. The most commonly used are:

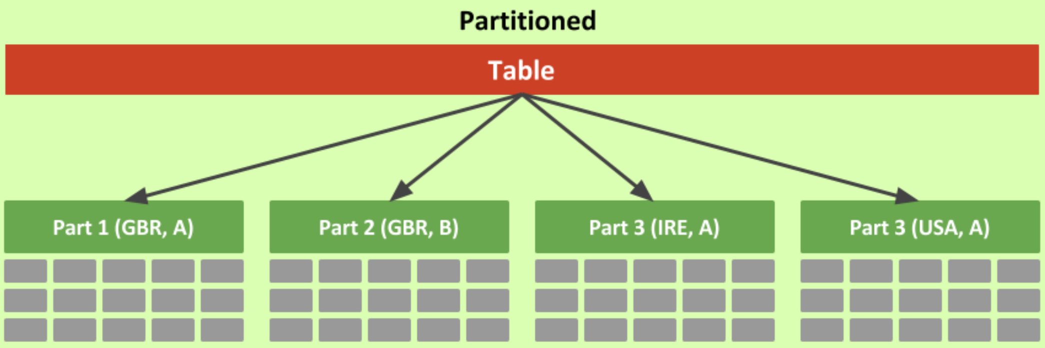
1. Range: commonly used for dates and time series data
    - The table could be divided into multiple tables, one for each day (daily partitioning), or a coarser partitioning would be monthly, quarterly, or even yearly.
    - It is called a range because for example the Quarter 1 partition would keep data between '1/1/2022' to '31/3/2022'



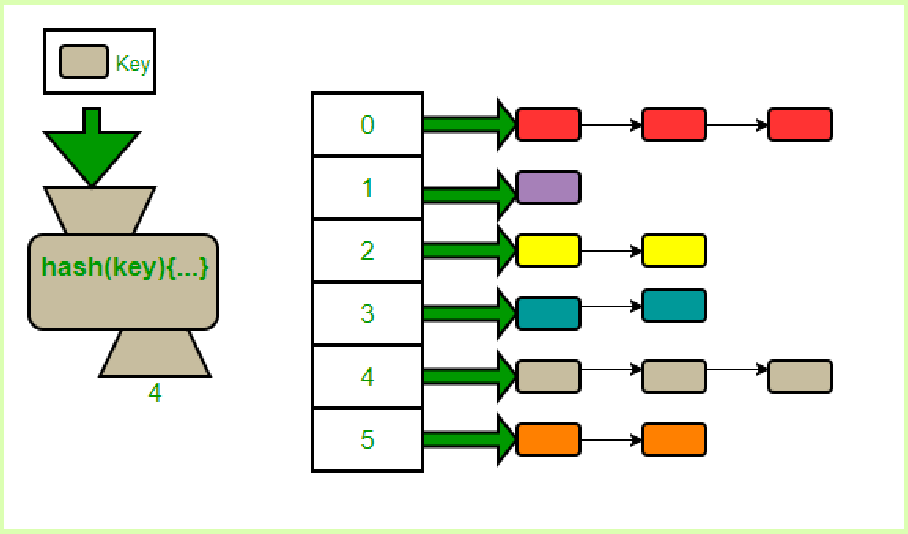
RANGE PARTITIONING

- Kudu tables are broken down into tablets and spread among tablet servers
- Each tablet is assigned a particular *range* of values.

2. List: commonly for used when there are a limited number of values in a column. e.g. Students in the ITI.
    - e.g. the "Departments" table would be divided into "Data Science" which only has data science students, "Data Management" which only have data management students and etc.



3. Hash: Implemented similar to separate chaining of a typical hash table/map data structure.
   - The values kept in each partition are not logically related like Ranges or Lists, but based on the output of the hash function. In this case, the user is free to specify the desired number of partitions.
   - e.g. assuming the user wants to use 10 partitions, A hash function that produces values between 1 and 10 will be used.
   - Values that produce a hash value of 3 will be stored in the 3rd partition, 8 in the 8th partition and etc.



When to Partition?

- When the table has MILLIONS of rows. not less than that. because partitions add complexity. they need to be worth it.
- When the table is *expected* to eventually reach millions of rows. plan ahead. e.g. in data warehouses where the data is never deleted.
- Indexes are preferred as the first solution over partitioning. if an index was added, but the query is still small, resort to partitioning.
- Tables have reached their maximum allowed size on disk

When not to partition?

- Avoid multi-level partitioning. this can explode the number of tables exponentially
- When you have too many partitions e.g. daily partition.
- When the number of rows in a table is not too much.