Database index

https://en.wikipedia.org/wiki/Database_index#Index_implementations
(https://en.wikipedia.org/wiki/Database_index#Index_implementations)
https://docs.microsoft.com/en-us/sql/relational-databases/indexes/clustered-and-nonclustered-indexes-described?view=sql-server-2017
(https://docs.microsoft.com/en-us/sql/relational-databases/indexes/clustered-and-nonclustered-indexes-described?view=sql-server-2017)

Definition

A database index is a **data structure** that improves the speed of data retrieval operations on a database table at the cost of additional writes and storage space to maintain the index data structure. Indexes are used to quickly locate data without having to search every row in a database table every time a database table is accessed. Indexes can be created using one or more columns of a database table, providing the basis for both rapid random lookups and efficient access of ordered records.

Usage

Support for fast lookup

Most database software includes indexing technology that enables **sub-linear** time lookup to improve performance, as linear search is inefficient for large databases.
An index is any **data structure** that improves the performance of lookup. There are many different data structures used for this purpose. There are complex design trade-offs involving lookup performance, index size, and index-update performance. Many index designs exhibit logarithmic (O(log(N))) lookup performance and in some applications it is possible to achieve flat (O(1)) performance.

Policing the database constraints

Indexes are used to **police database constraints**, such as UNIQUE, EXCLUSION, PRIMARY KEY and FOREIGN KEY. An index may be declared as UNIQUE (**this means index can be not unique**), which creates an implicit constraint on the underlying table. Database systems usually implicitly create an index on a set of columns declared PRIMARY KEY, and some are capable of using an already-existing index to police this constraint. Many database systems require that both referencing and referenced sets of columns in a FOREIGN KEY constraint are indexed, thus improving performance of inserts, updates and deletes to the tables participating in the constraint.

Advantage and Disadvantages of Index

- If we have a lot of retrieval actions such as finding an entry by '...where lastname = ...', then lastname column should bettered be defined with an index for fast searching.
- For other actions such as joining where searching action might also be involved, it is also better to have an index.
- For group by, if we group by index levels (can be single or multiple columns, see Pandas course), it should also improve the efficiency.
- For very small data set, or if there involved a lot of insertion or deletion, then using index is not a good idea as we have do a lot of work to maintaining and changing index structures accordingly after every insertion or deletion.

Intuitive pictures

https://www.youtube.com/watch?v=EZ3jBam2IEA (https://www.youtube.com/watch?v=EZ3jBam2IEA)
https://www.youtube.com/watch?v=ITcOiLSfVJQ (https://www.youtube.com/watch?v=ITcOiLSfVJQ)

- When we create a table with:

  ```
   Create table phonebook
  {
    lastname varchar(50) NOT NULL
    firstname varchar(50) NOT NULL
    phone number varch(50) NOT NULL
  }
  ```

  The data will be stored in heap without particular order. When we search one records with `where lastname = ...` , then we need to search the whole table all over the total space, which is very inefficient.
- If we store the above phone records with, e.g., alphabetical order, then we will still need search the whole space to find a particular record. However, one we find the record, we know when to stop.
- We store the name and phone number as in a phone book with alphabetical order. Also in the book define some extra stuff such as Aa-Af, Ag-Ba,....These extra stuff (usually B-tree in database) with the ordered phone records are collectively called clustered index. Here index and data are bound together.
- For the same ordering records, we can only have one clustered index. When we store the records not according to the previously alphabetical order, but according to an, e.g., numbering order of phone number, then we can define this storing together with the newly created extra stuff as another clustered index. This indicates that for a single table, only one clustered index can be defined.
- As in many books, there are index in the end of book. For an entry, we may find the relevant page number. This is similar to non-clustered index where index and data are stored in different places.
- For the same ordered or non-ordered results, we may have more than one indices. In the phone book example, we may use alphabetical order index of name, or use the number index of phone number, as long as both pointing to the correct page number.

Index architecture/Indexing Methods

- Non-clustered
  The physical order of the rows is not the same as the index order.
  The indexed columns are typically non-primary key columns used in JOIN, WHERE, and ORDER BY clauses.
  There can be more than one non-clustered index on a database table.

Nonclustered indexes have a structure separate from the data rows. A nonclustered index contains the nonclustered index key values and each key value entry has a pointer to the data row that contains the key value.

The pointer from an index row in a nonclustered index to a data row is called a row locator. The structure of the row locator depends on whether the data pages are stored in a heap or a clustered table. For a heap, a row locator is a pointer to the row. For a clustered table, the row locator is the clustered index key.

You can add nonkey columns to the leaf level of the nonclustered index to by-pass existing index key limits, and execute fully covered, indexed, queries.

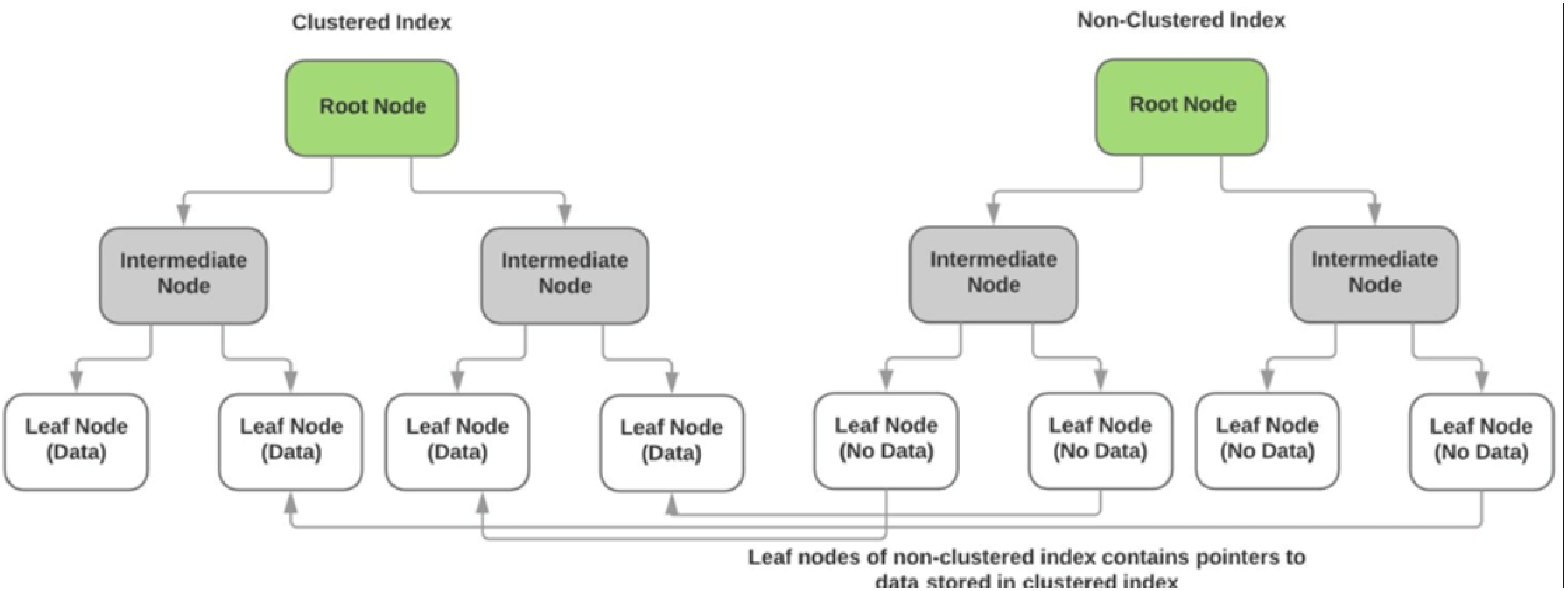
- Clustered
  Clustering alters the data block into a certain distinct order to match the index, resulting in the row data being stored in order. Therefore, only one clustered index can be created on a given database table. Clustered indices can greatly increase overall speed of retrieval, but usually only where the data is accessed sequentially in the same or reverse order of the clustered index, or when a range of items is selected.

Clustered indexes sort and store the data rows in the table or view based on their key values. These are the columns included in the index definition. There can be only one clustered index per table, because the data rows themselves can be stored in only one order.

The only time the data rows in a table are stored in sorted order is when the table contains a clustered index. When a table has a clustered index, the table is called a clustered table. If a table has no clustered index, its data rows are stored in an unordered structure called a heap.

Both clustered and nonclustered indexes can be unique. This means no two rows can have the same value for the index key. Otherwise, the index is not unique and multiple rows can share the same key value.

- Cluster
  When multiple databases and multiple tables are joined, it's referred to as a cluster (not to be confused with clustered index described above). The records for the tables sharing the value of a cluster key shall be stored together in the same or nearby data blocks.



Column order

The order that the index definition defines the columns in is important. It is possible to retrieve a set of row identifiers using only the first indexed column. However, it is not possible or efficient (on most databases) to retrieve the set of row identifiers using only the second or greater indexed column.

For example, imagine a phone book that is organized by city first, then by last name, and then by first name. If you are given the city, you can easily extract the list of all phone numbers for that city. However, in this phone book it would be very tedious to find all the phone numbers for a given last name.

In the phone book example with a composite index created on the columns (city, last_name, first_name), if we search by giving exact values for all the three fields, search time is minimal - but if we provide the values for city and first_name only, the search uses only the city field to retrieve all matched records. Then a sequential lookup checks the matching with first_name. So, to improve the performance, one **must ensure that the index is created on the order of search columns.**

Applications and limitations

Indexes are useful for many applications but come with some limitations. Consider the following SQL statement: SELECT first_name FROM people WHERE last_name = 'Smith';. To process this statement without an index the database software must look at the last_name column on every row in the table (this is known as a full table scan). With an index the database simply follows the B-tree (not confused with binary tree) data structure until the Smith entry has been found; this is much less computationally expensive than a full table scan.

Consider this SQL statement: SELECT email_address FROM customers WHERE email_address LIKE '%@wikipedia.org';. This query would yield an email address for every customer whose email address ends with "@wikipedia.org", but even if the email_address column has been indexed the database must perform a full index scan. This is because the index is built with the assumption that words go from left to right. With a wildcard at the beginning of the search-term, the database software is unable to use the underlying B-tree data structure (in other words, the WHERE-clause is not sargable). This problem can be solved through the addition of another index created on reverse(email_address) and a SQL query like this: SELECT email_address FROM customers WHERE reverse(email_address) LIKE reverse('%@wikipedia.org');. This puts the wild-card at the right-most part of the query (now gro.aidepikiw@%), which the index on reverse(email_address) can satisfy.

When the wildcard characters are used on both sides of the search word as %wikipedia.org%, the index available on this field is not used. Rather only a sequential search is performed, which takes O(N) time.

Types of indexes

- Bitmap index
  A bitmap index is a special kind of indexing that stores the bulk of its data as bit arrays (bitmaps) and answers most queries by performing bitwise logical operations on these bitmaps. **The most commonly used indexes, such as B+ trees, are most efficient if the values they index do not repeat or repeat a small number of times**. In contrast, the bitmap index is designed for cases where the values of a variable repeat very frequently. For example, the sex field in a customer database usually contains at most three distinct values: male, female or unknown (not recorded). For such variables, the bitmap index can have a significant performance advantage over the commonly used trees.
- Dense index
  A dense index in databases is a file with pairs of keys and pointers for every record in the data file. Every key in this file is associated with a particular pointer to a record in the sorted data file. In clustered indices with duplicate keys, the dense index points to the first record with that key.
- Sparse index
  A sparse index in databases is a file with pairs of keys and pointers for every block in the data file. Every key in this file is associated with a particular pointer to the block in the sorted data file. In clustered indices with duplicate keys, the sparse index points to the lowest search key in each block.
- Reverse index
  A reverse-key index reverses the key value before entering it in the index. E.g., the value 24538 becomes 83542 in the index. Reversing the key value is particularly useful for indexing data such as sequence numbers, where new key values monotonically

increase.

Implementation

Indices can be implemented using a variety of data structures. Popular indices include balanced trees, B+ trees and hashes. **Note B tree or B+ tree is not the usual binary tree.** If a tree is not balance, then an ordinary binary tree might be inefficient.

Primary key vs Index

Foreign key is just the primary key of other tables. Because most primary key has unique index created one it, and index is usually implemented with B/B+ tree or hashes, so primary key should also be implemented with these data structures.

Part I General comparison

https://itknowledgeexchange.techtarget.com/sql-server/difference-between-an-index-and-a-primary-key/ (https://itknowledgeexchange.techtarget.com/sql-server/difference-between-an-index-and-a-primary-key/)

The Primary Key is a **logical object**. By that I mean that is simply defines a set of properties on one column or a set of columns to require that the columns which make up the primary key are unique and that none of them are null. Because they are unique and not null, these values (or value if your primary key is a single column) can then be used to identify a single row in the table every time. **In most if not all database platforms the Primary Key will have an index created on it.**

An index on the other hand doesn't define uniqueness. An index is used to more quickly find rows in the table based on the values which are part of the index. When you create an index within the database, you are creating a **physical object** which is being saved to disk. Using a table which holds employees as an example:

```
CREATE TABLE dbo.Employee
 (EmployeeId INT PRIMARY KEY,
 LastName VARCHAR(50),
 FirstName VARCHAR(50),
 DepartmentId INT,
 StartDate DATETIME,
 TermDate DATETIME,
 TermReason INT)
```

The EmployeeId is the Primary Key for our table as that is what we will use to uniquely identify an employee. If we were to search the table based on the last name the database would need to read the entire table from the disk into memory so that we can find the few employees that have the correct last name. Now if we create an index on the LastName column when we run the same query, **the database only needs to load the index from the disk into memory**, which will be much quicker, and instead of scanning through the entire table looking for matches, because the values in the index are already sorted the database engine can go to the correct location within the index and find the matching records very quickly.

part II Unique index or primary key

https://stackoverflow.com/questions/487314/primary-key-or-unique-index (https://stackoverflow.com/questions/487314/primary-key-or-unique-index)

- What is a unique index?

A unique index on a column(s) is an index on that column(s) that also enforces the constraint that you cannot have two equal values in that column in two different rows. Example:

` CREATE TABLE table1 (foo int, bar int); CREATE UNIQUE INDEX ux_table1_foo ON table1(foo); -- Create unique index on foo.

INSERT INTO table1 (foo, bar) VALUES (1, 2); -- OK INSERT INTO table1 (foo, bar) VALUES (2, 2); -- OK INSERT INTO table1 (foo, bar) VALUES (3, 1); -- OK INSERT INTO table1 (foo, bar) VALUES (1, 4); -- Fails!`

Duplicate entry '1' for key 'ux_table1_foo'
The last insert fails because it violates the unique index on column foo when it tries to insert the value 1 into this column for a second time.

In MySQL a unique constraint allows multiple NULLs.

- Primary key versus unique index

Things that are the same:

```
* A primary key implies a unique index.
```

Things that are different:

```
* A primary key also implies NOT NULL, but a unique index can be nullable.
* There can be only one primary key, but there can be multiple unique indexes.  **But these multiply unique indices should be non-clustered indices, right?**
* If there is no clustered index defined then the primary key will be the clustered index.
```

My comments: If a primary key is defined, then it indicates we have already a clustered index. Because we can have only one clustered index for one table, we thus cannot define any clustered index once we have a primary key, right? See next section.

Part III Indexes and Constraints

Indexes are automatically created when PRIMARY KEY and UNIQUE constraints are defined on table columns.

- When you create a table with a UNIQUE constraint, Database Engine automatically creates a **non-clustered index**.
- If you configure a PRIMARY KEY, Database Engine automatically creates a **clustered index, unless a clustered index already exists**.
- When you try to enforce a PRIMARY KEY constraint on an existing table and a clustered index already exists on that table, SQL Server enforces the primary key using a nonclustered index.

Summary

- It seems index is more broadly used than primary key, and primary key is also handled and policed by index. Check whether the index in Pandas library of Python is similar to the index in SQL.
- Because we can only have one primary key for a table, it will not be convenient if we want to join or find with other column names. So we introduce other indices which can be more than just a primary key.