What’s the point of an index and what’s the trade-off to achieve it?

Indexes are used to increase the performance of a database by increasing that of SELECT commands run against a certain table. Index is a data structure that speed ups the location of a required data in the table by creating a sorted data structure. That structure might exist on a table itself, usually in the form of a Primary Key (known as Clustered index) or on a separate table (known as Non-Clustered) index.

The problem that and Index is intended to solve is an amount of disk reads performed by the database engine when looking for data.

What an Index does is basically sort the data so that a Binary Search can be run against it, which significantly decreases an amount of reads required to locate the data. A database engine might apply additional optimization, further decreasing the amount of required reads.

There are two trade-offs, however:

- UPDATE, INSERT and DELETE require more disk writes if an indexed column is affected, resulting in decreased DB performance when altering the data, as an index should be rebuilt each time such change occurs.
- Non-clustered indexes (the ones that exist separately from the table) require additional disk space and might consume a significant amount of it, if the DB contains big volumes of indexed data.

With those trade-offs in mind, a user should consider whether indexation of a certain column will increase the performance of a DB or decrease it.

What makes index good and what makes it bad?

A good index is an index that increases the performance of a DB and has some of the following features:

- Data in indexed columns is changed rarely – Index gets rebuilt less frequently
- Index is based either on a Primary Key, or a Foreign Key – that speed ups the data lookup
- Indexed columns allow quick identification of the data – for example, if the application often needs to find a client’s order by the order number, it would make sense to create a composite index (the one that contains several columns) on the customer_id and order_no columns instead of two separate indexes.
- Helps to find a range of information – for example, if we often need to find all orders placed since June to November, indexing an order_date column should help.
- Keeps data ordered – that point mainly relates to the clustered index which keeps data physically sorted in the table.
- Covering index – an index that contains all the data required to process a certain query therefore allowing the DB engine to complete it without accessing the main table. However, inconsiderate use of a covering index might actually decrease the performance of a database so it should be used with caution.

A bad index is an index that either slows down the database or doesn’t affect its performance positively in a significant way:

- Indexed data is rarely queried – keeping an unused index only creates the additional performance overhead and additional disk space consumption without any benefits.
- Indexed data is changed often – such changes can result in an increased resource overhead as well as performance decrease so a DBA should consider if indexing a certain column will be beneficial to the DB.
- Indexed data has many identical records – many identical records will decrease the performance of a Binary Search as there’s a high chance that DB engine would still have

to look through the identical records one-by-one to find the required value, which negates the benefits of an index.
- Too many columns were indexed – having too many columns in an index slows down its performance as well as increasing an overhead while rebuilding it and might be an indicator that there's something wrong with that particular table's design. As a rule of a thumb, an index should not contain more than 4 or 5 columns.
- And index is created on a table with few values – in that case a DB engine can decide that full table scan would be more fast than querying an index, resulting in an unused index.

Name two basic index types and explain the difference between them.

There are two basic index types – clustered and non-clustered. The main difference between them is location of an index – a clustered index is located in the table itself and physically sorts the data inside it; a non-clustered index is a separate table (that actually has clustered index inside) that stores indexed column's data in a sorted manner and each such record has a pointer to the actual row in the main table.

A table can contain several non-clustered indexes but only one clustered, as its impossible to have data sorted in two different ways at the same time.

Describe additional index types.

There are two (three for some databases) additional index types and one property.

- Composite – an index that contains several columns, for example first and last names. Columns are sorted in sequential order.
- Unique – an index that may not contain repeating values. If combined with clustered, the uniqueness is enforced across all columns.
- Covering – it's not really a type but rather a property of an index. A covering index is an index that contains in itself all the data required to respond to a certain query, without accessing the main table. Usually a covering index is a property of a clustered index but it's not necessarily the case all the time.

- Clustered – some databases, namely MSSQL Server, allow defining a clustered index – an index that defined the physical order of the data on disk.

Replication

Questions

What's the point of replication?

Replication is a scaling technique that's used to increase the following properties of a DB:

- Speed – replication allows to keep data physically close to its users, reducing the time a request and a response should travel each time a query is run against an instance of a DB.
- Performance – replication allows multiple instances of a database handle requests simultaneously, decreasing the workload of each particular instance.
- Stability – replication results in several databases having exactly the same data, so in case of a main database's failure, one of its replicas can step up and handle the requests.

However, according to CAP theorem, only two of the following properties: Consistence, Availability and Partition tolerance can exist on a distributed system at the same time so replication can't fully provide all its benefits completely at the same time – the DBA will have to consider the needs of an application and decide the most suitable way to implement replication.

What types of replication exist?

There are three replication types:

8. Snapshot replication – creates a single snapshot of a DB and distributes it among the replicas. It doesn't tack any further changes. This replication type also severs as a starting point for other two types.
9. Transactional replication – also known as statement-based replication, it applies the changes from the Master to the Followers as they occur incrementally, statement-by-statement. It preservers the transactional bounds of each change among all the Followers in the same way as those changes occurred on Master.
10. Merge replication – also known as row-based replication, it synchronizes individual rows between two or more databases and resolves conflicts between them, should they occur. That replication is used in server-client environment where client might go offline for some time and change the data inside its DB during that period. Once he connects to the network, the Merge Agent will take care of synchronizing his changes with the changes that happened on the Master DB and other Followers.

What replication topologies exist?
There are three main replication topologies:

11. Single-Leader – only one DB instance handles all writes and then replicates them across the Followers. Clients can't write into the Followers directly but they can directly read data from them. This topology is rather simple as it excludes the possibility of conflicts but the Leader must be able to handle all the requests and it somewhat diminishes the benefits of geo-replication as the data should still travel to the location of a main DB.
12. Multi-Leader – several Leader DBs are handling the requests and replicate them among the followers. This topology provides more than one DB instance clients can send their writes to and also allows the Leader to be located close to clients. If some DB instances have to work offline, they can have their own Leader that synchronizes with the rest of DB later. The main problem of this setup is the possibility of conflicts between two or more Leaders.
13. Leader-less – this topology doesn't have a Leader, rather all the DB instances are Leaders. Those instances don't communicate with each-other and a client has to send requests to all of them at once. The main benefit is high availability without the price tag of Multi-Leader topology, the main minus is that the data isn't replicated automatically and it's up to the client to ensure that all DBs stay synchronized.

What's the difference between asynchronous and synchronous replication?
An asynchronous replication happens in the background, once the Leader processed the request and sent the response to the client. Synchronous replication happens after a Leader received and processed a request but before it sends a response.

Asynchronous replication offers zero client performance impact at a cost of lower durability and replication lag.

Synchronous replication offers higher durability at a cost of both performance and availability, as the Leader can't send the response until a change has been replicated among all its Followers. To solve this problem, a workaround called semi-asynchronous replication exists – the Leader only have to replicate data synchronously to some of the Followers before sending the response; the rest of replication will happen asynchronously.

Partitioning and Sharding

Questions

What's a partitioning? What's the point of partitioning?
Partitioning is a scaling technique, division of a certain database object to several smaller ones for the sake of maintainability or performance. The performance and maintainability is achieved by spreading those smaller object across different machines or file groups.

What’s vertical partitioning?

Vertical partitioning is a data division strategy where each partition holds a subset of the fields for the items in the data store. The fields are divided according to their pattern of use. For example, frequently accessed fields can be placed into one vertical partition and less frequently accessed fields in another. The vertical partitioning is somewhat similar to the normalization and an added benefit of performance can be achieved by moving the new partitions to the different file groups.

What’s horizontal partitioning?

Horizontal partitioning – also called sharding - is a partitioning strategy where each partition is a data store in its own right but all partitions have the same schema. Each partition is known as a shard and hold a specific subset of the data, such as all the orders for a specific set of customers. Sharding mainly relies on shards being distributed across several physical machines, although it can be implemented within a single DB instance by employing different file groups.

What entities can be partitioned?

Both single tables and databases can be partitioned.