Improve Indexes

Creating useful indexes is one of the most important ways to achieve better query performance. Useful indexes help you find data with fewer disk I/O operations and less system resource usage.

To create useful indexes, you much understand how the data is used, the types of queries and the frequencies they are run, and how the query processor can use indexes to find your data quickly.

When you choose what indexes to create, examine your critical queries, the performance of which will affect user experience most. Create indexes to specifically aid these queries. After adding an index, rerun the query to see if performance is improved. If it is not, remove the index.

As with most performance optimization techniques, there are tradeoffs. For example, with more indexes, **SELECT** queries will potentially run faster. However, DML (**INSERT**, **UPDATE**, and **DELETE**) operations will slow down significantly because more indexes must be maintained with each operation. Therefore, if your queries are mostly **SELECT** statements, more indexes can be helpful. If your application performs many DML operations, you should be conservative with the number of indexes you create.

### Create Highly-Selective Indexes

Indexing on columns used in the WHERE clause of your critical queries frequently improves performance. However, this depends on how selective the index is. Selectivity is the ratio of qualifying rows to total rows. If the ratio is low, the index is highly selective. It can get rid of most of the rows and greatly reduce the size of the result set. It is therefore a useful index to create. By contrast, an index that is not selective is not as useful.

A unique index has the greatest selectivity. Only one row can match, which makes it most helpful for queries that intend to return exactly one row. For example, an index on a unique ID column will help you find a particular row quickly.

You can evaluate the selectivity of an index by running the sp\_show\_statistics stored procedures on SQL Server Compact tables. For example, if you are evaluating the selectivity of two columns, "Customer ID" and "Ship Via", you can run the following stored procedures:

sp\_show\_statistics\_steps 'orders', 'customer id';

### Create Multiple-Column Indexes

Multiple-column indexes are natural extensions of single-column indexes. Multiple-column indexes are useful for evaluating filter expressions that match a prefix set of key columns. For example, the composite index CREATE INDEX Idx\_Emp\_Name ON Employees ("Last Name" ASC, "First Name" ASC) helps evaluate the following queries:

* ... WHERE "Last Name" = 'Doe'
* ... WHERE "Last Name" = 'Doe' AND "First Name" = 'John'
* ... WHERE "First Name" = 'John' AND "Last Name" = 'Doe'

However, it is not useful for this query:

* ... WHERE "First Name" = 'John'

### Create Multiple-Column Indexes

Multiple-column indexes are natural extensions of single-column indexes. Multiple-column indexes are useful for evaluating filter expressions that match a prefix set of key columns. For example, the composite index CREATE INDEX Idx\_Emp\_Name ON Employees ("Last Name" ASC, "First Name" ASC) helps evaluate the following queries:

* ... WHERE "Last Name" = 'Doe'
* ... WHERE "Last Name" = 'Doe' AND "First Name" = 'John'
* ... WHERE "First Name" = 'John' AND "Last Name" = 'Doe'

However, it is not useful for this query:

* ... WHERE "First Name" = 'John'

### SQL Tuning: Execution Plans in SQL Server

By the way: the Execution Plan tool in SQL Server can be useful for creating indexes.

## Rewrite Subqueries to Use JOIN

Sometimes you can rewrite a subquery to use JOIN and achieve better performance. The advantage of creating a JOIN is that you can evaluate tables in a different order from that defined by the query. The advantage of using a subquery is that it is frequently not necessary to scan all rows from the subquery to evaluate the subquery expression. For example, an EXISTS subquery can return TRUE upon seeing the first qualifying row.

## Use Parameterized Queries

If your application runs a series of queries that are only different in some constants, you can improve performance by using a parameterized query. For example, to return orders by different customers, you can run the following query:

SELECT "Customer ID" FROM Orders WHERE "Order ID" = ?

Parameterized queries yield better performance by compiling the query only once and executing the compiled plan multiple times. Programmatically, you must hold on to the command object that contains the cached query plan. Destroying the previous command object and creating a new one destroys the cached plan. This requires the query to be re-compiled. If you must run several parameterized queries in interleaved manner, you can create several command objects, each caching the execution plan for a parameterized query. This way, you effectively avoid re-compilations for all of them.