Chapter 37: Indexes

Indexes are a data structure that contains pointers to the contents of a table arranged in a specific order, to help the database optimize queries. They are similar to the index of book, where the pages (rows of the table) are indexed by their page number.

Several types of indexes exist, and can be created on a table. When an index exists on the columns used in a query's WHERE clause, JOIN clause, or ORDER BY clause, it can substantially improve query performance.

Section 37.1: Sorted Index

If you use an index that is sorted the way you would retrieve it, the `SELECT` statement would not do additional sorting when in retrieval.

```
CREATE INDEX ix_scoreboard_score ON scoreboard (score DESC);
```

When you execute the query

```
SELECT * FROM scoreboard ORDER BY score DESC;
```

The database system would not do additional sorting, since it can do an index-lookup in that order.

Section 37.2: Partial or Filtered Index

SQL Server and SQLite allow to create indexes that contain not only a subset of columns, but also a subset of rows.

Consider a constant growing amount of orders with `order_state_id` equal to finished (2), and a stable amount of orders with `order_state_id equal` to started (1).

If your business make use of queries like this:

```
SELECT id, comment
  FROM orders
 WHERE order_state_id =  1
   AND product_id = @some_value;
```

Partial indexing allows you to limit the index, including only the unfinished orders:

```
CREATE INDEX Started_Orders
          ON orders(product_id)
       WHERE order_state_id = 1;
```

This index will be smaller than an unfiltered index, which saves space and reduces the cost of updating the index.

Section 37.3: Creating an Index

```
CREATE INDEX ix_cars_employee_id ON Cars (EmployeeId);
```

This will create an index for the column *EmployeeId* in the table *Cars*. This index will improve the speed of queries asking the server to sort or select by values in *EmployeeId*, such as the following:

```
SELECT * FROM Cars WHERE EmployeeId = 1
```

The index can contain more than 1 column, as in the following;

```
CREATE INDEX ix_cars_e_c_o_ids ON Cars (EmployeeId, CarId, OwnerId);
```

In this case, the index would be useful for queries asking to sort or select by all included columns, if the set of conditions is ordered in the same way. That means that when retrieving the data, it can find the rows to retrieve using the index, instead of looking through the full table.

For example, the following case would utilize the second index;

```
SELECT * FROM Cars WHERE EmployeeId = 1 Order by CarId DESC
```

If the order differs, however, the index does not have the same advantages, as in the following;

```
SELECT * FROM Cars WHERE OwnerId = 17 Order by CarId DESC
```

The index is not as helpful because the database must retrieve the entire index, across all values of EmployeeId and CarID, in order to find which items have `OwnerId = 17`.

(The index may still be used; it may be the case that the query optimizer finds that retrieving the index and filtering on the `OwnerId`, then retrieving only the needed rows is faster than retrieving the full table, especially if the table is large.)

Section 37.4: Dropping an Index, or Disabling and Rebuilding it

```
DROP INDEX ix_cars_employee_id ON Cars;
```

We can use command `DROP` to delete our index. In this example we will `DROP` the index called *ix_cars_employee_id* on the table *Cars*.

This deletes the index entirely, and if the index is clustered, will remove any clustering. It cannot be rebuilt without recreating the index, which can be slow and computationally expensive. As an alternative, the index can be disabled:

```
ALTER INDEX ix_cars_employee_id ON Cars DISABLE;
```

This allows the table to retain the structure, along with the metadata about the index.

Critically, this retains the index statistics, so that it is possible to easily evaluate the change. If warranted, the index can then later be rebuilt, instead of being recreated completely;

```
ALTER INDEX ix_cars_employee_id ON Cars REBUILD;
```

Section 37.5: Clustered, Unique, and Sorted Indexes

Indexes can have several characteristics that can be set either at creation, or by altering existing indexes.

```
CREATE CLUSTERED INDEX ix_clust_employee_id ON Employees(EmployeeId, Email);
```

The above SQL statement creates a new clustered index on Employees. Clustered indexes are indexes that dictate the actual structure of the table; the table itself is sorted to match the structure of the index. That means there can be at most one clustered index on a table. If a clustered index already exists on the table, the above statement will

fail. (Tables with no clustered indexes are also called heaps.)

```
CREATE UNIQUE INDEX uq_customers_email ON Customers(Email);
```

This will create an unique index for the column *Email* in the table *Customers*. This index, along with speeding up queries like a normal index, will also force every email address in that column to be unique. If a row is inserted or updated with a non-unique *Email* value, the insertion or update will, by default, fail.

```
CREATE UNIQUE INDEX ix_eid_desc ON Customers(EmployeeID);
```

This creates an index on Customers which also creates a table constraint that the EmployeeID must be unique. (This will fail if the column is not currently unique - in this case, if there are employees who share an ID.)

```
CREATE INDEX ix_eid_desc ON Customers(EmployeeID Desc);
```

This creates an index that is sorted in descending order. By default, indexes (in MSSQL server, at least) are ascending, but that can be changed.

Section 37.6: Rebuild index

Over the course of time B-Tree indexes may become fragmented because of updating/deleting/inserting data. In SQLServer terminology we can have internal (index page which is half empty) and external (logical page order doesn't correspond physical order). Rebuilding index is very similar to dropping and re-creating it.

We can re-build an index with

```
ALTER INDEX index_name REBUILD;
```

By default rebuilding index is offline operation which locks the table and prevents DML against it , but many RDBMS allow online rebuilding. Also, some DB vendors offer alternatives to index rebuilding such as `REORGANIZE` (SQLServer) or `COALESCE`/`SHRINK SPACE`(Oracle).

Section 37.7: Inserting with a Unique Index

```
UPDATE Customers SET Email = "richard0123@example.com" WHERE id = 1;
```

This will fail if an unique index is set on the *Email* column of *Customers*. However, alternate behavior can be defined for this case:

```
UPDATE Customers SET Email = "richard0123@example.com" WHERE id = 1 ON DUPLICATE KEY;
```