***Case-Insensitive Search Using UPPER or LOWER:***

SELECT first\_name, last\_name, phone\_number

FROM employees

WHERE **UPPER(last\_name) = UPPER('winand')**

It is a return of our old friend the full table scan. Although there is an index on **LAST\_NAME**, it is unusable—because the search is *not* on **LAST\_NAME** but on **UPPER(LAST\_NAME)**. From the database’s perspective, that’s something *entirely different*.

This is a trap we all might fall into. We recognize the relation between **LAST\_NAME** and **UPPER(LAST\_NAME)** instantly and expect the database to “see” it as well. In reality the optimizer’s view is more like this:

SELECT first\_name, last\_name, phone\_number

FROM employees

WHERE **BLACKBOX(...) = 'WINAND'**

The **UPPER** function is just a black box. The parameters to the function are not relevant because there is no general relationship between the function’s parameters and the result.

To support that query, we need an index that covers the actual search term. That means we do not need an index on **LAST\_NAME** but on **UPPER(LAST\_NAME)**:

CREATE INDEX emp\_up\_name

ON employees (**UPPER(last\_name)**)

An index whose definition contains functions or expressions is a so-called function-based index

SQL Server and MySQL do not support function-based indexes as described but both offer a workaround via computed or generated columns. To make use of this, you have to first add a generated column to the table that can be indexed afterwards:

[MySQL](https://use-the-index-luke.com/sql/where-clause/functions/case-insensitive-search)

Since MySQL 5.7 you [can index a generated columns](https://dev.mysql.com/doc/refman/5.7/en/create-table.html#create-table-secondary-indexes-virtual-columns) as follows:

ALTER TABLE employees

ADD COLUMN last\_name\_up VARCHAR(255) AS (UPPER(last\_name));

CREATE INDEX emp\_up\_name ON employees (last\_name\_up);

***User-Defined Functions***

Function-based indexing is a very generic approach. Besides functions like **UPPER** you can also index expressions like **A + B** and even use user-defined functions in the index definition.

There is one important exception. It is, for example, not possible to refer to the current time in an index definition, neither directly nor indirectly, as in the following example.

CREATE FUNCTION **get\_age**(date\_of\_birth DATE)

RETURN NUMBER

AS

BEGIN

RETURN

**TRUNC(MONTHS\_BETWEEN(SYSDATE, date\_of\_birth)/12)**;

END

The function **GET\_AGE** uses the current date (**SYSDATE**) to calculate the age based on the supplied date of birth. You can use this function in all parts of an SQL query, for example in select and the where clauses:

SELECT first\_name, last\_name, **get\_age(date\_of\_birth)**

FROM employees

WHERE **get\_age(date\_of\_birth) = 42**

The query lists all 42-year-old employees. Using a function-based index is an obvious idea for optimizing this query, but you cannot use the function **GET\_AGE** in an index definition because it is not *deterministic*. That means the result of the function call is not fully determined by its parameters. Only functions that always return the same result for the same parameters—functions that are deterministic—can be indexed.

The reason behind this limitation is simple. When inserting a new row, the database calls the function and stores the result in the index and there it stays, unchanged. There is no periodic process that updates the index. The database updates the indexed age only when the date of birth is changed by an update statement. After the next birthday, the age that is stored in the index will be wrong.

Besides *being* deterministic, PostgreSQL and the Oracle database require functions to be *declared* to be deterministic when used in an index so you have to use the keyword **DETERMINISTIC** (Oracle) or **IMMUTABLE** (PostgreSQL).

***Indexing LIKE Filters***

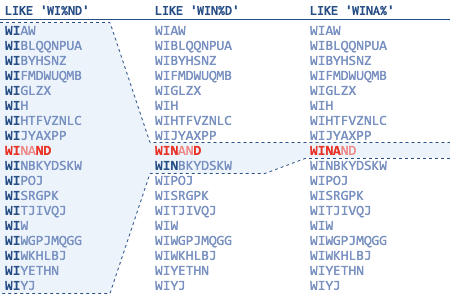
The SQL **LIKE** operator very often causes unexpected performance behavior because some search terms prevent efficient index usage. That means that there are search terms that can be indexed very well, but others can not. It is the position of the wild card characters that makes all the difference.

The following example uses the **%** wild card in the middle of the search term:

SELECT first\_name, last\_name, date\_of\_birth

FROM employees

WHERE **UPPER(last\_name) LIKE 'WIN%D'**



Only the part before the first wild card serves as an access predicate.

The remaining characters do not narrow the scanned index range—non-matching entries are just left out of the result.

Avoid **LIKE** expressions with leading wildcards (e.g., **'%TERM'**).

***One index scan is faster than two.***

***Partial Indexes***

A partial index is useful for commonly used where conditions that use constant values—like the status code in the following example:

SELECT message

FROM messages

WHERE **processed = 'N'**

AND receiver = ?

Queries like this are very common in queuing systems. The query fetches all unprocessed messages for a specific recipient. Messages that were already processed are rarely needed. If they are needed, they are usually accessed by a more specific criteria like the primary key.

We can optimize this query with a two-column index. Considering this query only, the column order does not matter because there is no range condition.

CREATE INDEX messages\_todo

ON messages (receiver, processed)

The index fulfills its purpose, but it includes many rows that are never searched, namely all the messages that were already processed. Due to the logarithmic scalability the index nevertheless makes the query very fast even though it wastes a lot of disk space.

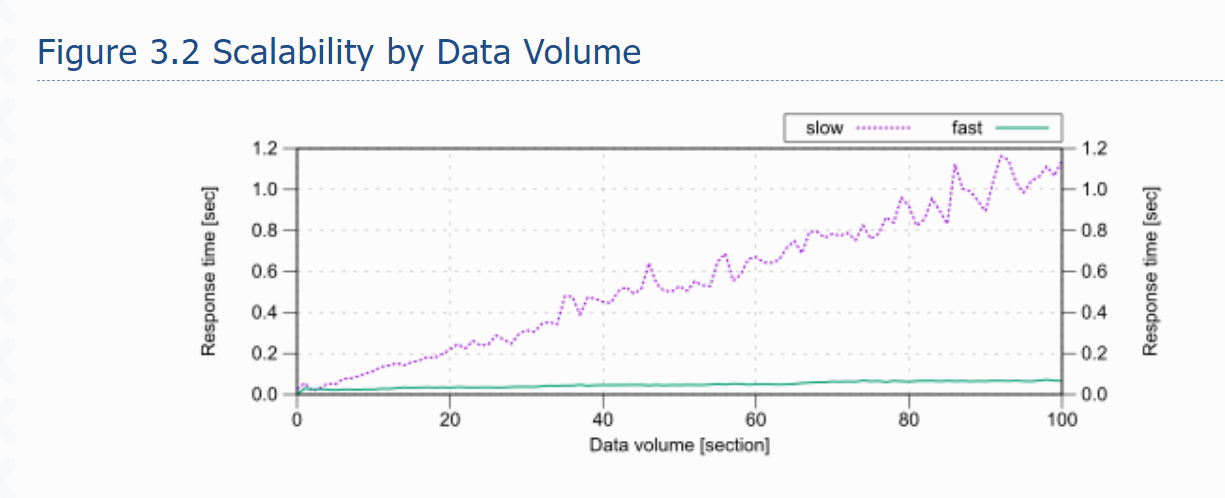
With partial indexing you can limit the index to include only the unprocessed messages. The syntax for this is surprisingly simple: a where clause.

CREATE INDEX messages\_todo

ON messages (receiver)

**WHERE processed = 'N'**

The index only contains the rows that satisfy the where clause. In this particular case, we can even remove the **PROCESSED** column because it is always **'N'** anyway. That means the index reduces its size in two dimensions: vertically, because it contains fewer rows; horizontally, due to the removed column.



SELECT count(\*)

FROM scale\_data

WHERE section = ?

AND id2 = ?

The definition of the **SCALE\_SLOW** index must start with the column **SECTION**—otherwise it could not be used as access predicate. The condition on **ID2** is not an access predicate—so it cannot follow **SECTION** in the index definition. That means the **SCALE\_SLOW** index must have minimally three columns where **SECTION** is the first and **ID2** not the second. That is exactly how it is in the index definition used for this test:

CREATE INDEX **scale\_slow** ON scale\_data (**section**, id1, **id2**)

The database cannot use **ID2** as access predicate due to column **ID1** in the second position.

The definition of the **SCALE\_FAST** index must have columns **SECTION** and **ID2** in the first two positions because both are used for access predicates. We can nonetheless not say anything about their order. The index that was used for the test starts with the **SECTION** column and has the extra column **ID1** in the third position:

CREATE INDEX **scale\_fast** ON scale\_data (**section**, **id2**, id1)

The column **ID1** was just added so this index has the same size as **SCALE\_SLOW**—otherwise you might get the impression the size causes the difference.

