# **Database Object Management using DDL**

## **Objectives**

2.1 Construct and analyze queries that create, alter, and drop **tables:**

Create, alter, and drop tables by using proper ANSI SQL syntax; NULL and NOT NULL

2.2 Construct and analyze queries that create, alter, and drop **views:**

Create, alter, and drop views by using proper ANSI SQL syntax; purpose of views

2.3 Construct and analyze **stored procedures and functions:**

Input and output parameters, return values, purpose of stored procedures

2.4 Given a scenario, choose between **clustered and non-clustered indexes:**

When to use clustered vs. non-clustered indexes, syntax for creating indexes

## **Reading Materials**

SQL Primer - An Accelerated Introduction to SQL Basics: Chapters 4, 14 & 15

### Chapter 4: Operations on Tables

#### Overview

Tables are the fundamental storage containers of the relational world. A database will typically contain many tables, each representing a collection of entities. As requirements evolve, so do the tables within a database and database administrators (DBA's) routinely perform administrative operations on individual tables like deleting them or changing their definition. While typical database users are not granted permissions to perform such operations on large production databases, nonetheless it is important to be familiar with them for didactic purposes.

You might have noticed that we keep on making new tables whenever we are introducing a new concept. This has had the not-so-desirable effect of populating our database with many similar tables each holding programming languages data but with slightly varying definitions and constraints. We will now go about dropping unneeded tables and modifying existing ones to suit our needs.

#### Dropping Tables

The deletion of tables in SQL is achieved through the DROP TABLE command. DROP is actually a top-level SQL command, much like CREATE, which performs a deletion operation on many kinds of database objects. To delete a table, we simply append it with the database object type – a TABLE in this case.

We will now drop any superfluous tables we have created during the previous lessons ([Listing 4-1](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=141650109&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)). Note that dropping a table means deleting a table and any data inside it without a chance of recovery. So be careful while writing DROP commands.

DROP TABLE proglang\_tbl;

DROP TABLE proglang\_tblcopy;

DROP TABLE proglang\_constraints;

DROP TABLE proglang\_tbltmp;

If you get no errors returned, it means that the tables have been deleted. DROP TABLE only supports dropping a single table at a time conventionally, though there are clever ways to go about deleting multiple tables with a single statement.

To verify whether the tables have actually been dropped, you have two choices. A simplistic one is to write any query for the table, and you would get an error back similar to **Error: no such table: proglang\_tbl.** The other way is to get the listing of currently existing tables from the database *catalog*, which is a database that the DBMS internally uses to keep a track of databases, tables, and other objects that users create. Querying the catalog in SQLite for a listing of tables is extremely simple ([Listing 4-2](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=141650109&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

sqlite> .tables

proglang\_tbluk

Doing the same thing in PostgreSQL is slightly longer, but it is a SELECT query on the catalog database ([Listing 4-3](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=141650109&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

testdb=# SELECT table\_name

FROM information\_schema.tables

WHERE table\_schema = 'public'

AND table\_type = 'BASE TABLE';

table\_name

----------------------

proglang\_tbluk

(1 row)

If the query seems complicated to you, it is because it contains parts and syntax that we have not covered yet. But rest assured, the syntax will start making perfect sense by the end of the chapter on queries. For now we can infer that the table data in PostgreSQL is stored in the information\_schema.tables catalog table.

#### Creating New Tables from Existing Tables

You might have noticed that we have dropped the *proglang\_tbl* table, and we now have with us only the *proglang\_tbluk* table that has all the necessary constraints and fields. The latter's name was chosen when we were discussing the unique key constraint, but it now seems logical to migrate this table structure (and any corresponding data) back to the name *proglang\_tbl*. We achieve this by creating a copy of the table using a combination of both CREATE TABLE and SELECT commands and learn a new clause in the process – AS ([Listing 4-4](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=954073795&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)). This combination has a particularly catchy name – *CTAS* and was introduced in the SQL:2003 standard but not all DBMS systems implement it yet, notably Microsoft SQL Server.

CREATE TABLE <New Table>

AS

SELECT <Selection> FROM <Old Table>;

Since our *proglang\_tbluk* contains only one record, we will push some more sample data in it so that we can later verify whether the records themselves got copied or not. Notice that we give the field names explicitly, or else the second row (which contains no *standard* field value) would give an error similar to:

sqlite> INSERT INTO proglang\_tbluk

VALUES

(2, 'Perl', 'Wall', 1987);

Error: table proglang\_tbluk has 5 columns but 4 values were supplied

in SQLite. A lot of other DBMS's like *Ingres* would also not accept such a cavalier approach to inserting data. PostgreSQL, however, would accept such a statement provided it could unambiguously insert the data that in this case it can due to the omitted value being the last nullable field only. I would advise writing the column names explicitly wherever possible. We will follow this sage advice in [Listing 4-5](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=954073795&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#).

INSERT INTO proglang\_tbluk (id, language, author, year)

VALUES (2, 'Perl', 'Wall', '1987');

INSERT INTO proglang\_tbluk (id, year, standard, language, author)

VALUES (3, '1964', 'ANSI', 'APL', 'Iverson');

To create an exact copy of the existing table, we use the same selection criteria as we have seen before – \* (star). This will select all the fields from the existing table and create the new table with them along with any records ([Listing 4-6](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=954073795&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)). It is possible to use only a subset of fields from the old table by modifying the selection criteria and we will see this later.

CREATE TABLE proglang\_tbl

AS

SELECT \* FROM proglang\_tbluk;

If you are using *psql*, you would see the prompt displaying SELECT 3, which gives an indicator of how many rows were selected and inserted into the new table. We now run a simple SELECT query to see whether our objective was achieved or not ([Table 4-1](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=954073795&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT \* FROM proglang\_tbl;

| id | language | author | year | standard |
| --- | --- | --- | --- | --- |
| 1 | Prolog | Colmerauer | 1972 | ISO |
| 2 | Perl | Wall | 1987 |  |
| 3 | APL | Iverson | 1964 | ANSI |

The two tables are now exactly identical but are not linked to each other in any way. If you drop any of the tables, the other one will not be affected. Similarly, inserting new data in one of them will not insert the data in the other one from now on.

#### Modifying Tables

After a table has been created, you can still modify its structure using the ALTER TABLE command ([Listing 4-7](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=101445577&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)). What we mean by modify is that you can change field types, sizes, even add or delete columns. Not all database management systems support all operations of ALTER TABLE. To get around such limitations, people frequently copy the data over to a new table that has the newly required structure. While altering a table is not an SQL command you'd use very often (hopefully!), you should be familiar with it.

There are some rules you have to abide by while altering a table and these are usually spelled out in detail by your particular DBMS manual. For now, we will see a simple example to modify the field *author* for the *proglang\_tbl* table.

ALTER TABLE <Table name> <Operation> <Field with clauses>;

To keep our *proglang\_tbl* intact, we are going to be making our changes to the old *proglang\_tbluk* table. We want to make the *author* field hold a tad bit more maximum data length of 30 characters instead of 25. The operation to choose in this case is ALTER COLUMN, which would modify our existing field ([Listing 4-8](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=101445577&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

ALTER TABLE proglang\_tbluk

ALTER COLUMN author TYPE varchar(30);

If you are not using SQLite, the above query should execute quietly in PostgreSQL. SQLite unfortunately does not support altering a column size but happily supports addition of new columns. So let's add another requirement of adding a nullable column *current\_status* to our table fields ([Listing 4-9](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=101445577&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

ALTER TABLE proglang\_tbluk

ADD COLUMN current\_status VARCHAR(32) NULL;

We have used the ADD COLUMN operation in this case for the ALTER TABLE command. Unsurprisingly we hope to add this new 32-character length column to our *proglang\_tbluk* with this statement.

#### The Many Faces of ALTER TABLE

Altering a table is one of those commands where even after three decades, there are discrepancies. For example, you already saw that the ALTER COLUMN doesn't work in SQLite. *Ingres;* another, DBMS, expects you to write ALTER only, which coincidentally also works fine for postgreSQL. We choose the former to be explicit. *Oracle* on the other hand has gone a completely different way, and it uses MODIFY instead of ALTER COLUMN.

Similarly, while adding a column, you could write only ADD <column name> in postgreSQL or SQLite and expect it to work.

However, while altering data types, postgreSQL expects you to write TYPE between the column name and the new data type specification whereas Ingres wouldn't expect it.

Always keep the manual of your DBMS handy!

#### Showing Table Information in PostgreSQL

If you are thinking about using the database system catalog to get table definition information to verify your ALTER TABLE results, congratulations! You are indeed correct in thinking that. As before, the query might seem a little more than we can handle correctly at this point in the text, but its output is highly readable ([Listing 4-10](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=313141694&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

testdb=# SELECT column\_name,

data\_type,

character\_maximum\_length

FROM INFORMATION\_SCHEMA.COLUMNS

WHERE table\_name = 'proglang\_tbluk';

column\_name | data\_type | character\_maximum\_length

----------------+-------------------+--------------------------

id | integer |

language | character varying | 20

author | character varying | 30

year | integer |

standard | character varying | 10

current\_status | character varying | 32

(6 rows)

Both our changes to the fields *author* and *current\_status* seem to be reflected correctly. There are a few other databases where such a query would work, but unfortunately this is another area where a lot of DBMS implementations differ widely.

A PostgreSQL *psql-specific* method is to use \d+ <table name>, which gives almost the same information along with some other values by default ([Listing 4-11](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=313141694&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)). I personally prefer the [Listing 4-10](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=313141694&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#) version that queries the catalog.

testdb=# \d+ proglang\_tbluk;

Table "public.proglang\_tbluk"

Column | Type | Modifiers | Storage |

---------------+-----------------------+-----------+----------+

id | integer | | plain |

language | character varying(20) | | extended |

author | character varying(30) | | extended |

year | integer | | plain |

standard | character varying(10) | | extended |

current\_status | character varying(32) | | extended |

Indexes:

"proglang\_tbluk\_pkey" PRIMARY KEY, btree (id)

"proglang\_tbluk\_language\_key" UNIQUE CONSTRAINT,

btree (language)

#### Showing Table Information in SQLite

As we have already discussed SQLite, as of the writing of this text, does not support modification to column sizes in a table using ALTER TABLE. It does however allow you to add a new column, and we added the *current\_status* field like with PostgreSQL. Let's now verify this by looking at the table information inside the SQLite shell.

SQLite has its own special *dot syntax* commands that allow certain useful database management tasks. We have already seen the .open command used to create and open a database and .tables to list the table names. Similarly we can use the .schema command to get table information ([Listing 4-12](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=945656396&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

sqlite> .schema proglang\_tbluk

CREATE TABLE proglang\_tbluk (

id INTEGER NOT NULL PRIMARY KEY,

language VARCHAR(20) NOT NULL UNIQUE,

author VARCHAR(25) NOT NULL,

year INTEGER NOT NULL,

standard VARCHAR(10) NULL,

current\_status VARCHAR(32) NULL);

Notice how the new column is added at the end while the length of the *author* field remains 25 characters.

#### Showing Table Information in Other DBMS’s

If you are not practicing on either of the DBMS implementations mentioned above, there might be other ways to verify table field-level information. For example, Ingres utilizes the HELP TABLE <table name> command, which can be run on its *isql* shell.

A lot of other DBMS's like Oracle use the DESCRIBE command to view a table definition. While the information this command shows may vary from one DBMS to another, they at least show the field name, its data type, and whether or not NULL values are allowed for the particular field. The general syntax of the command is given below ([Listing 4-13](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=592270127&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

DESCRIBE <table name>;

### Chapter 14: Views

#### Overview

One of the beautiful aspects of the relational data model and SQL is that the output of a query is also a table, a relation to be precise. It may consist of a single column or a single row, but it is a table nonetheless. A *view* is a query that can be used like a table.

Think of it as a *virtual table* that stores for the viewer's convenience a pre-computed resultset. It does not truly exist like a *base table* but provides a different angle to view the data without the tedium of details.

#### Why Are Views Needed?

Most production database systems would contain a lot of tables. It is also possible that some of these tables consist of a lot of fields because of the complexity of the domain. Views would come to the rescue of the casual database user, people who are not experts in all parts of the database system. They have a specific, repetitive need, and views provide them with a simpler interface to the data they need.

Another advantage that views bring to the table is security. We can restrict access to base tables and provide views containing only the data a particular group of users is allowed to see. Good database design rules often force sensitive columns to be lumped together with oft-accessed fields. Views come to the rescue in such cases by effectively hiding the sensitive columns if you so choose.

For the database designers, views provide independence. To a reasonable degree, views allow the underlying base tables to change their structure to cater to evolving needs and yet views can remain the same. In other cases, views can be re-created with a different query underlying it but will contain the same data in the same format, providing a continuity to the user.

#### Creating a View

The general syntax of creating a view is pretty straightforward ([Listing 14-1](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=362182466&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)). In fact, it probably is as minimal and natural as you can get.

CREATE VIEW <view name> AS <query>

Now let us create a view for ourselves – *language\_chronology* that will have only two fields, namely, languages and their years of creation ([Listing 14-2](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=362182466&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

CREATE VIEW language\_chronology AS

SELECT language, year

FROM proglang\_tbl

ORDER BY year ASC;

Notice how we have explicitly added the ordering clause to the view creation. There are very few restrictions on what is allowed in the query part of CREATE VIEW. Let us now verify the results by running a query on the view exactly the same way as we would on a table ([Listing 14-3](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=362182466&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT \* FROM language\_chronology;

| language | year |
| --- | --- |
| Fortran | 1957 |
| JOVIAL | 1959 |
| APT | 1959 |
| RPG | 1964 |
| APL | 1964 |
| PL/I | 1964 |
| prolog | 1972 |
| perl | 1987 |
| Tcl | 1988 |

We can also include calculated fields in the query part of view creation. The only thing we must keep in mind is how we rename the calculated field column, failing which would undoubtedly result in a loss of clarity. Let's re-create our decade query from [Chapter 9](http://viewer.books24x7.com/assetviewer.aspx?bkid=142634&destid=263#263), this time as a view ([Listing 14-4](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=362182466&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

CREATE VIEW language\_decade AS

SELECT language,

'The '||((year/10)\*10)||'s' decade

FROM proglang\_tbl;

| language | decade |
| --- | --- |
| prolog | The 1970s |
| perl | The 1980s |
| APL | The 1960s |
| JOVIAL | The 1950s |
| APT | The 1950s |
| TcI | The 1980s |
| PL/I | The 1960s |
| Fortran | The 1950s |
| RPG | The 1960s |

If we had failed to rename the column as *decade*, our DBMS systems would execute the view creation but the resultant view would be practically unusable. PostgreSQL would have renamed the column to a mysterious ?column? whereas SQLite would have put the entire expression as the name of the second field 'The '||((year/10)\*10)||'s'. Needless to say, we are better off renaming the fields of the view.

Another way to rename the fields is to specify it in the view definition clause rather than the query that populates it ([Listing 14-5](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=362182466&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)). This works just as well and is arguably clearer because it lists the fields upfront.

CREATE VIEW language\_era (lang, era) AS

SELECT language,

'The '||((year/10)\*10)||'s'

FROM proglang\_tbl

WHERE year < 1971;

| lang | era |
| --- | --- |
| APL | The 1960s |
| JOVIAL | The 1950s |
| APT | The 1950s |
| PL/I | The 1960s |
| Fortran | The 1950s |
| RPG | The 1960s |

If you choose this method of renaming columns, you must specify the names of all the columns in the view. Note that renaming a field has no effect on its data type or null status.

#### Modifying Data Through Views

There are widely ranging opinions on whether data modification through views is a good idea. Some people prefer to treat views as a read-only listing of contents, but most DBMS systems provide some data modification ability through views.

Let us first try a simple update of the *year* column through our *language\_chronology* table ([Listing 14-6](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=293042737&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)). Remember we had pulled the *year* field into the view along with the language name.

UPDATE language\_chronology

SET year=1977

WHERE language='Fortran';

The statement executes fine in PostgreSQL. Now to verify whether it actually made a difference, let us verify the contents of the view first ([Listing 14-7](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=293042737&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT \* FROM language\_chronology

WHERE language='Fortran';

| language | year |
| --- | --- |
| Fortran | 1977 |

All seems to be as expected in the view. While we have an inkling that the base table would also have been updated, let's verify this too ([Listing 14-8](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=293042737&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT \* FROM proglang\_tbl

WHERE language='Fortran';

| id | language | author | year | standard |
| --- | --- | --- | --- | --- |
| 8 | Fortran | Backus | 1977 | ANSI |

This also means that the other view *language\_era* that was dependent on *proglang\_tbl* would not contain the row for Fortran since its creation involved the use of the condition WHERE year < 1971. Records move in and out of views as the underlying base table contents change over time.

#### View Modification in SQLite

If you attempted to execute this UPDATE command in SQLite, it would throw you an error like:

Error: cannot modify language\_chronology because it is a view

SQLite has clearly stated that it would not stand for data modification through a view. A design choice I happen to agree with.

Let's attempt another data modification, but this time we will try to update the calculated field inside the view *language\_era* ([Listing 14-9](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=293042737&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)). We know that JOVIAL was made in the year 1959, so we wish to round off the value and make its era to The 1960s.

UPDATE language\_era SET era='The 1960s'

WHERE lang='JOVIAL';

> ERROR: cannot update column "era" of view "language\_era"

> DETAIL: View columns that are not columns of their base relation are not updatable.

The DBMS has rejected our request to update a calculated field because *era* does not exist in the base table *proglang\_tbl*. If we think about it, this makes sense because the SQL interpreter would not know what *year* value to put in the base table. A value of 1960, a value of 1969, and everything in between would make the *era* value as The 1960s. The DBMS would not attempt to choose any random value because its reasoning would be ambiguous.

Changing the *lang* field of the same view is perfectly unambiguous and hence allowed ([Listing 14-10](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=293042737&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

UPDATE language\_era SET lang='Jovial'

WHERE lang='JOVIAL';

> UPDATE 1

SELECT \* FROM proglang\_tbl

WHERE id=4;

| id | language | author | year | standard |
| --- | --- | --- | --- | --- |
| 4 | Jovial | schwartz | 1959 | US-DOD |

Similarly, we can create a view with aggregated columns using the GROUP BY clause, but modifying the contents of such a view is not allowed ([Listing 14-11](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=293042737&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

CREATE VIEW standards AS

SELECT standard, count(\*)

FROM proglang\_tbl

GROUP BY standard;

| standard | count |
| --- | --- |
|  | 2 |
| ECMA | 1 |
| ANSI | 2 |
| ISO | 3 |
| US-DOD | 1 |

We know from previous experience that adding a new row or modifying the aggregated column would be ambiguous and thus not allowed. But what if we attempted to just update the *standard* field value just like we did with JOVIAL ([Listing 14-12](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=293042737&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#))? Would the update be reflected in all rows containing the field value?

UPDATE standards SET standard='IS'

WHERE standard='ISO';

> ERROR: cannot update view "standards"

> DETAIL: Views containing GROUP BY are not automatically

updatable.

> HINT: To enable updating the view, provide an INSTEAD OF

UPDATE trigger or an unconditional ON UPDATE DO INSTEAD rule.

The operation was disallowed but PostgreSQL gave us a hint on how to go about achieving this. While we won't cover that technique, it's good to know that in the rare case you do need it, it's available in some database systems.

#### Deleting Views

To delete or remove a view in its entirety, we use the DROP VIEW command. It is very similar to the DROP TABLE command we saw in [Chapter 4](http://viewer.books24x7.com/assetviewer.aspx?bkid=142634&destid=114#114) ([Listing 14-13](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=478867400&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

DROP VIEW standards;

Note that you cannot accidentally drop a table using DROP VIEW, which is a relief ([Listing 14-14](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=478867400&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

DROP VIEW proglang\_tbl;

> ERROR: "proglang\_tbl" is not a view

> HINT: Use DROP TABLE to remove a table.

### Chapter 15: Indexing

#### Overview

Databases have long been the primary data storage components from which insights are derived. As businesses increasingly adopt technology-enabled workflows, the rate of data generation has grown substantially. This trend has accelerated with the adoption of the Internet and mobile computing.

A well-sized relational database used to run into hundreds of megabytes in the 1990s. It is not uncommon to hear of database systems running into hundreds of gigabytes or even a few terabytes nowadays. As a professional, you will frequently encounter tables with million rows in them.

Until now we have seen the parts of SQL that allow you to perform operations and run queries, but we haven't touched anything close to performance tuning. When you want to run your queries on multimillion record tables, taking a hard look at performance optimization is not optional.

One of the most common performance optimization techniques is indexing. An *index* allows the SQL engine to quickly *look up* specific records in a table. This is not unlike jumping directly to the letter W in a dictionary when you wish to know the meaning of *wistfUl*. It would be quite tedious to go through all the pages sequentially starting with A until we reach our desired word.

A lot of commands in this chapter are specific to the DBMS at hand. While the general concept of indexes and basic commands to create and delete indexes remain similar across products, there is no getting around the fact that as we get deeper into our SQL journey, vendor-based differences become more visible.

#### Creating an Index

As with most statements in SQL, index creation is pretty straightforward. You use the CREATE INDEX command to achieve this ([Listing 15-1](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=520851399&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

CREATE INDEX <index name> ON <table name> (<column list>);

Let's create a simple index on the *language* column of *proglang\_tbl* ([Listing 15-2](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=520851399&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)). We make a reasonable assumption that there are going to be a lot of queries using the *language* field in the WHERE clause, and creating an index on it would increase performance.

CREATE INDEX language\_idx ON proglang\_tbl(language);

If this command succeeds in *psql*, you would not get an error back but no other visual indication. Let's verify our index creation attempt by listing the table description like we did in [Chapter 4](http://viewer.books24x7.com/assetviewer.aspx?bkid=142634&destid=114#114). Instead of the detailed \d+ <table name> option, we will use the slightly compact \d <table name> command ([Listing 15-3](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=520851399&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

\d proglang\_tbl;

Table "public.proglang\_tbl"

Column | Type | Modifiers

----------+-----------------------+-----------

id | integer |

language | character varying(20) |

author | character varying(25) |

year | integer |

standard | character varying(10) |

Indexes:

"language\_idx" btree (language)

We can see at the very end of the output, there is an entry for our index *language\_idx*. Running the same CREATE INDEX on SQLite also succeeds, and there are two primary ways to verify the creation. The first is the .schema command we had seen in [Chapter 4](http://viewer.books24x7.com/assetviewer.aspx?bkid=142634&destid=114#114) ([Listing 15-4](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=520851399&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

sqlite> .schema proglang\_tbl

CREATE TABLE proglang\_tbl (

id INTEGER NOT NULL,

language VARCHAR(20) NOT NULL,

author VARCHAR(25) NOT NULL,

year INTEGER NOT NULL,

standard VARCHAR(10) NULL);

CREATE INDEX language\_idx ON proglang\_tbl (language);

This showed us the DDL commands that were used to create the table and its related entities like indexes. Another approach is to use an SQLite *pragma* to list all indexes on a table ([Listing 15-5](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=520851399&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)). Pragmas are statements provided by SQLite to query its own metadata such as index information.

sqlite> PRAGMA index\_list(proglang\_tbl);

seq name unique origin partial

---------- ------------ ---------- ---------- ----------

0 language\_idx 0 c 0

#### Using EXPLAIN to See Indexes at Work

We have now seen that the index we created actually exists, but how do we see it in action? We should be able to get a measurable speed up on a large table. The EXPLAIN command would come to our rescue here. But first let's go about creating a large table to run our index-enabled queries on.

A quick and dirty way to get a large table would be to use a cartesian product or cross joins. We already have our *proglang\_tbl* with 5 columns and 9 rows in it. Doing a cartesian product on each of the fields with each other should yield us 9 to the power 5 = 59049 rows ([Listing 15-6](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=747144215&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)). This is not a huge table by any means but it would allow us to see the effect an index has on query execution.

SELECT a.language,

b.author,

c.year,

d.standard,

e.id

INTO biglang\_tbl

FROM proglang\_tbl a, proglang\_tbl b, proglang\_tbl c, proglang\_tbl d, proglang\_tbl e;

SELECT count(\*) FROM biglang\_tbl;

count

-------

59049

Now that we have a populated table, let us try to analyze the query time for finding all Fortran rows using EXPLAIN ([Listing 15-7](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=747144215&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)). While this command is not in the SQL standard, most relational database systems implement it.

EXPLAIN SELECT \* FROM biglang\_tbl WHERE language="Fortran";

QUERY PLAN

---------------------------------------------------------------

Seq Scan on biglang\_tbl (cost=0.00..1150.11 rows=6486

width=24)

Filter: ((language)::text = 'Fortran'::text)

(2 rows)

Well there is some output even though it is not entirely evident yet what we are seeing, A *query plan*, is a term for how the SQL engine is going to execute your query. Let us now contrast this output with the one after creating an index on the *language* field ([Listing 15-8](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=747144215&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

CREATE INDEX biglang\_idx ON biglang\_tbl (language);

EXPLAIN SELECT \* FROM biglang\_tbl WHERE language='Fortran';

QUERY PLAN

---------------------------------------------------------------

----------------

Bitmap Heap Scan on biglang\_tbl (cost=126.56..619.63

rows=6486 width=24)

Recheck Cond: ((language)::text = 'Fortran'::text)

-> Bitmap Index Scan on biglang\_idx (cost=0.00..124.94

rows=6486 width=0)

Index Cond: ((language)::text = 'Fortran'::text)

(4 rows)

We immediately see that this output is different from the previous one, and it involves the use of our recently created index. The previous output mentioned a Seq Scan or a sequential scan, which as the name suggests, involves going through our records sequentially. The current output, however, mentions Bitmap Heap Scan and Bitmap Index Scan, which sound way faster than a full simple scan. The details of how these particular scans work is out of the scope of this text, but we can look at another parameter in the output to get a relative sense of the efficiency of our index in this case.

Both plans mention a parameter-like cost=<1st value>..<2nd value>. The second value is the estimated total cost of query execution. The smaller this value is, the greater is the efficiency of query execution. In the first output without the index, this value is estimated as 1150.11 while after index creation, we reduce it down to 619.63, a big win for us.

While index creation on SQLite is similar to other databases, if you had tried to execute [Listing 15-6](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=747144215&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#) in it, you would have gotten an error saying something along the lines of Error: near "INTO": syntax error. The supported way to create the *biglang\_tbl* in SQLite would be to use the CREATE TABLE .. AS .. <query> ([Listing 15-9](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=747144215&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

CREATE TABLE biglang\_tbl AS

SELECT a.language,

b.author,

c.year,

d.standard,

e.id

FROM proglang\_tbl a, proglang\_tbl b, proglang\_tbl c,

proglang\_tbl d, proglang\_tbl e;

Also instead of using the simple EXPLAIN, which gives a huge and pretty incomprehensible output at first glance, we use the more succinct EXPLAIN QUERY PLAN statement like below ([Listing 15-10](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=747144215&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

EXPLAIN QUERY PLAN SELECT \* FROM biglang\_tbl WHERE

language="Fortran";

selectid order from detail

---------- ---------- ---------- ---------------------------

0 0 0 SEARCH TABLE biglang\_tbl

USING INDEX biglang\_idx

(language=?)

While the output is pretty small as compared to the one from PostgreSQL, we can clearly see that it is going to use our index to search our table for Fortran rows.

#### Unique Indexes

We can optionally specify the keyword UNIQUE during index creation to make an index that only allows non-duplicate values ([Listing 15-11](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=126868748&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)). This makes the index have a dual responsibility of data integrity along with performance enhancement.

CREATE UNIQUE INDEX <index name> ON <table name> (<column list>)

However, since it has an implied data integrity meaning, we cannot use this kind of index on a field that is already known to have duplicate values. In our newly created *biglang\_tbl*, the ID field is actually duplicated many times due to our cross join conditions. Creating a unique index on this field would result in an error ([Listing 15-12](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=126868748&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

CREATE UNIQUE INDEX id\_idx ON biglang\_tbl (id);

ERROR: could not create unique index "id\_idx"

DETAIL: Key (id)=(4) is duplicated.

Similarly adding a duplicate value into a field that has a unique index would result in an error along the lines of ERROR: duplicate key value violates unique constraint "<index name>".

If you have an extremely sharp memory, you'd recall that this is the same error we saw in [Listing 3-12](http://viewer.books24x7.com/assetviewer.aspx?bkid=142634&destid=111#111) back in [Chapter 3](http://viewer.books24x7.com/assetviewer.aspx?bkid=142634&destid=80#80) when we were discussing unique constraints. If we now try to describe the schema of the involved table *proglang\_tbluk*, we would see how PostgreSQL defined the constraints in terms of indexes ([Listing 15-13](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=126868748&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

\d proglang\_tbluk;

Table "public.proglang\_tbluk"

Column | Type | Modifiers

----------------+------------------------+-----------

id | integer | not null

language | character varying(20) | not null

author | character varying(30) | not null

year | integer | not null

standard | character varying(10) |

current\_status | character varying(32) |

Indexes:

"proglang\_tbluk\_pkey" PRIMARY KEY, btree (id)

"proglang\_tbluk\_language\_key" UNIQUE CONSTRAINT, btree

(language)

Actually, this is also something we had tried in [Chapter 4](http://viewer.books24x7.com/assetviewer.aspx?bkid=142634&destid=114#114) in [Listing 4-11](http://viewer.books24x7.com/assetviewer.aspx?bkid=142634&destid=140#140), but we were not paying very close attention to the INDEXES section of the output back then.

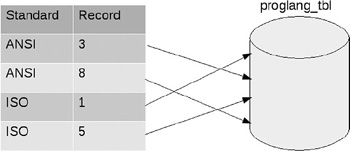
#### How Do Indexes Work?

Having a high-level overview of how indexes work can help the user write effective, fast-executing queries. Most SQL users ignore the conceptual understanding of indexes, but they are not really hard to grasp.

At the beginning of the chapter, we compared the database index to looking up a word in a dictionary. That lookup process was made easier by the alphabetical ordering nature of a dictionary. Similarly a book index allows you to look up concepts discussed in the book by listing them alphabetically with a page number where the concept is discussed.

This is very similar to an actual database index. While the underlying details vary from implementation to implementation, it is helpful to think of it as an ordered lookup table. The values of the field being indexed are sorted and stored along with the pointers to the locations of the actual record in the base table.

If we created an index on the *standard* field of *proglang\_tbl*, a simplified representation of the index would look like [Figure 15-1](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=560778068&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#). The SQL interpreter would not have to traverse through the whole of the table to find the two rows with ANSI as the standard field. The inefficient whole table traversal is what is sometimes referred to as full table scan or sequential scan. The point of an index is to avoid this kind of scan.



When someone writes a query with a WHERE clause finding the specific value of a *standard*, this index would come into effect automatically without the user having to specify using it. Adding or deleting more rows with differing values of this field would automatically update the index too, so that the index always refers to the latest data in the table.

#### Index Overheads

With all the niceties that indexes bring to the user without much effort in terms of arcane commands, sometimes users want to create indexes for every column possible. After all, there doesn't seem to be a downside to index creation yet.

Well as it turns out, like with everything else in the world, there is no free lunch. If there is an index on every column for a table with *N* fields, then for every DML statement like INSERT, UPDATE, or DELETE, the *N* indexes have to be kept in sync. This makes changing data slow for large tables, sometimes annoyingly and sometimes worryingly.

Another serious overhead that too many indexes bring is their storage requirements. Indexes occupy physical space on the disk just like a table. While storage has become cheap in recent times, database administrators are not known for their cavalier attitude toward server free space.

Let's check how the disk is impacted by index creation. We will keep our focus on the *biglang\_tbl* and its index *biglang\_idx*. First let's find the total space occupied by the table and its related objects ([Listing 15-14](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=186419168&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT pg\_size\_pretty(pg\_total\_relation\_size('biglang\_tbl'));

pg\_size\_pretty

----------------

4608 kB

(1 row)

The function pg\_total\_relation\_size would return the disk space occupied by the table, its indexes, and a few other things. The pg\_size\_pretty is for prettifying the output to a more human-readable unit of kB rather than the number of bytes. As is evident from these function names, they are specific to PostgreSQL. Check your DBMS manual for commands to query the database catalog in case you are using a different product.

Now let's find how much space the table and index take out of this figure ([Listing 15-15](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=186419168&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)). That should give us a relative idea about how big indexes get.

testdb=# SELECT pg\_size\_pretty(pg\_relation\_size('biglang\_tbl'));

pg\_size\_pretty

----------------

3296 kB

(1 row)

testdb=# SELECT pg\_size\_pretty(pg\_relation\_size('biglang\_idx'));

pg\_size\_pretty

----------------

1312 kB

(1 row)

Our index is roughly 39% of the size of our table! Not to mention it occupies 28% of the total relation size of *biglang\_tbl*. Keep in mind that we are talking about a single index on one column here. Clearly we need to be parsimonious with our index creation.

A good rule of thumb is to rely on the primary key and unique indexes a lot during your queries. Over time you will start seeing patterns of slow- running queries. If these queries are not run often, perhaps we can live with the extra time taken. But if the slow-running queries are run regularly and often contain the same field in the WHERE clause that is not indexed, it is a perfect candidate for index creation.

#### Index Size in SQLite

You might have noticed that we discussed index sizes in postgreSQL only. The helper functions like pg\_size\_ pretty, pg\_total\_relation\_size, and pg\_relation\_size are specific to postgreSQL and are not a part of the SQL standard.

I haven't found a good way to calculate index sizes in SQLite. Since it is a self-contained single file database, you could verify the file size before and after index creation to get a rough estimate.

#### Deleting an Index

If you no longer need an index, SQL gives you the DROP INDEX command to delete an index. The general syntax of this command is simple enough ([Listing 15-16](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=864863462&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

DROP INDEX <index name>

Note that this does not change the data of the underlying table in any way ([Listing 15-17](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=864863462&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)). All you are doing is dropping the index, so the query time may become slower.

DROP INDEX biglang\_idx;

SELECT COUNT(\*) FROM biglang\_tbl;

count

-------

59049

(1 row)

SQL for Dummies, 9th Edition: Chapter 21

### Chapter 21: Adding Procedural Capabilities with Persistent Stored Modules

#### Overview

* Tooling up compound statements with atomicity, cursors, variables, and conditions
* Regulating the flow of control statements
* Doing loops that do loops that do loops
* Retrieving and using stored procedures and stored functions
* Assigning privileges, creating stored modules, and putting stored modules to good use

Some of the leading practitioners of database technology have been working on the standards process for years. Even after a standard has been issued and accepted by the worldwide database community, progress toward the next standard doesn’t slow down. A seven-year gap separated the issuance of SQL-92 and the release of the first component of SQL:1999. During the intervening years, ANSI and ISO issued an addendum to SQL-92, called SQL-92/PSM (Persistent Stored Modules). This addendum formed the basis for a part of SQL:1999 with the same name. SQL/PSM defines a number of statements that give SQL flow of control structures comparable to the flow of control structures available in full-featured programming languages. It enables you to use SQL to perform tasks that programmers previously were forced to use other tools for. Can you imagine what your life would have been like in the caveman times of 1992, when you’d have to repeatedly swap between SQL and its procedural host language just to do your work?

#### Compound Statements

Throughout this book, SQL is represented as a nonprocedural language that deals with data a set at a time rather than a record at a time. With the addition of the facilities covered in this chapter, however, this statement is not as true as it used to be. Although SQL still deals with data a set at a time, it is becoming more procedural.

Archaic SQL (defined by SQL-92) doesn’t follow the procedural model — where one instruction follows another in a sequence to produce a desired result — so early SQL statements were standalone entities, perhaps embedded in a C++ or Visual Basic program. With these early versions of SQL, posing a query or performing other operations by executing a series of SQL statements was discouraged because these complicated activities resulted in a performance penalty in the form of network traffic. SQL:1999 and all following versions allow compound statements, made up of individual SQL statements that execute as a unit, easing network congestion.

All the statements included in a compound statement are enclosed between a BEGIN keyword at the beginning of the statement and an END keyword at the end of the statement. For example, to insert data into multiple related tables, you use syntax similar to the following:

void main {

EXEC SQL

BEGIN

INSERT INTO students (StudentID, Fname, Lname)

VALUES (:sid, :sfname, :slname) ;

INSERT INTO roster (ClassID, Class, StudentID)

VALUES (:cid, :cname, :sid) ;

INSERT INTO receivable (StudentID, Class, Fee)

VALUES (:sid, :cname, :cfee)

END ;

/\* Check SQLSTATE for errors \*/

}

This little fragment from a C program includes an embedded compound SQL statement. The comment about SQLSTATE deals with error handling. If the compound statement doesn't execute successfully, an error code is placed in the status parameter SQLSTATE. Of course, placing a comment after the END keyword doesn't correct the error. The comment is placed there simply to remind you that in a real program, error-handling code belongs in that spot. (I discuss error handling in detail in [Chapter 22](http://viewer.books24x7.com/assetviewer.aspx?bkid=145080&destid=1142#1142).)

#### Atomicity

Compound statements introduce a possibility for error that you don’t face when you construct simple SQL statements. A simple SQL statement either completes successfully or doesn’t, and if it doesn’t complete successfully, the database is unchanged. This is not necessarily the case when a compound statement creates an error.

Consider the example in the preceding section. What if the INSERT to the STUDENTS table and the INSERT to the ROSTER table both took place, but because of interference from another user, the INSERT to the RECEIVABLE table failed? A student would be registered for a class but would not be billed. This kind of error can be hard on a university's finances.

The concept that is missing in this scenario is *atomicity.* An atomic statement is indivisible — it either executes completely or not at all. Simple SQL statements are atomic by nature, but compound SQL statements are not. However, you can make a compound SQL statement atomic by specifying it as such. In the following example, the compound SQL statement is safe by introducing atomicity:

void main {

EXEC SQL

BEGIN ATOMIC

INSERT INTO students (StudentID, Fname, Lname)

VALUES (:sid, :sfname, :slname) ;

INSERT INTO roster (ClassID, Class, StudentID)

VALUES (:cid, :cname, :sid) ;

INSERT INTO receivable (StudentID, Class, Fee)

VALUES (:sid, :cname, :cfee)

END ;

/\* Check SQLSTATE for errors \*/

}

By adding the keyword ATOMIC after the keyword BEGIN, you ensure that either the entire statement executes, or — if an error occurs — the entire statement rolls back, leaving the database in the state it was in before the statement began executing. Atomicity is discussed in detail in [Chapter 15](http://viewer.books24x7.com/assetviewer.aspx?bkid=145080&destid=841#841) in the course of the discussion of transactions.

You can find out whether a statement executed successfully. Read the section "[Conditions](https://viewer.books24x7.com/assetviewer.aspx?bookid=145080&chunkid=241542640&resumebookmarkid=7f36470f-ed52-ee11-aa73-005056b543a4#)," later in this chapter, for more information.

#### Variables

Full computer languages such as C and Java have always offered *variables,* but SQL didn't offer them until the introduction of SQL/PSM. A variable is a symbol that takes on a value of any given data type. Within a compound statement, you can declare a variable, assign it a value, and use it in a compound statement.

After you exit a compound statement, all the variables declared within it are destroyed. Thus, variables in SQL are local to the compound statement within which they are declared.

Here’s an example:

BEGIN

DECLARE prezpay NUMERIC ;

SELECT salary

INTO prezpay

FROM EMPLOYEE

WHERE jobtitle = 'president' ;

END;

#### Cursors

You can declare a *cursor* within a compound statement. You use cursors to process a table’s data one row at a time. (See [Chapter 20](http://viewer.books24x7.com/assetviewer.aspx?bkid=145080&destid=1067#1067) for details.) Within a compound statement, you can declare a cursor, use it, and then forget it because the cursor is destroyed when you exit the compound statement. Here’s an example of this usage:

BEGIN

DECLARE ipocandidate CHARACTER(30) ;

DECLARE cursor1 CURSOR FOR

SELECT company

FROM biotech ;

OPEN CURSOR1 ;

FETCH cursor1 INTO ipocandidate ;

CLOSE cursor1 ;

END;

#### Conditions

When people say that a person has a condition, they usually mean that something is wrong with that person — he or she is sick or injured. People usually don’t bother to mention that a person is in *good* condition; rather, they talk about people who are in serious condition or, even worse, in critical condition. This idea is similar to the way programmers talk about the condition of an SQL statement. The execution of an SQL statement leads to a successful result, a questionable result, or an outright erroneous result. Each of these possible results corresponds to a *condition.*

Every time an SQL statement executes, the database server places a value into the status parameter SQLSTATE. SQLSTATE is a five-character field. The value that is placed into SQLSTATE indicates whether the preceding SQL statement executed successfully. If it did not execute successfully, the value of SQLSTATE provides some information about the error.

The first two of the five characters of SQLSTATE (the class value) give you the major news as to whether the preceding SQL statement executed successfully, returned a result that may or may not have been successful, or produced an error. [Table 21-1](https://viewer.books24x7.com/assetviewer.aspx?bookid=145080&chunkid=241542640&resumebookmarkid=7f36470f-ed52-ee11-aa73-005056b543a4#) shows the four possible results.

| Class | Description | Details |
| --- | --- | --- |
| 00 | Successful completion | The statement executed successfully. |
| 01 | Warning | Something unusual happened during the execution of the statement, but the DBMS can't tell whether there was an error. Check the preceding SQL statement carefully to ensure that it is operating correctly. |
| 02 | Not found | No data was returned as a result of the execution of the statement. This may or may not be good news, depending on what you were trying to do with the statement. You may be hoping for an empty result table. |
| Other | Exception | The two characters of the class code, plus the three characters of the subclass code, comprise the five characters of SQLSTATE. They also give you an inkling about the nature of the error. |

#### Handling Conditions

You can have your program look at SQLSTATE after the execution of every SQL statement. What do you do with the knowledge that you gain?

* **If you find a class code of 00, you probably don't want to do anything.** You want execution to proceed as you originally planned.
* **If you find a class code of 01 or 02, you may want to take special action.** If you expected the "Warning" or "Not Found" indication, then you probably want to let execution proceed. If you didn't expect either of these class codes, then you probably want to have execution branch to a procedure that is specifically designed to handle the unexpected, but not totally unanticipated, warning or not found result.
* **If you receive any other class code, something is wrong.** You should branch to an exception-handling procedure. Which procedure you choose to branch to depends on the contents of the three subclass characters, as well as the two class characters of SQLSTATE. If multiple different exceptions are possible, there should be an exception-handling procedure for each one because different exceptions often require different responses. You may be able to correct some errors or find workarounds. Other errors may be fatal; no one will die, but you may end up having to terminate the application.

#### Handler Declarations

You can put a *condition handler* within a compound statement. To create a condition handler, you must first declare the condition that it will handle. The condition declared can be some sort of exception, or it can just be something that’s true. [Table 21-2](https://viewer.books24x7.com/assetviewer.aspx?bookid=145080&chunkid=241542640&resumebookmarkid=7f36470f-ed52-ee11-aa73-005056b543a4#) lists the possible conditions and includes a brief description of what causes each type of condition.

| Condition | Description |
| --- | --- |
| SQLSTATE VALUE ‘xxyyy’ | Specific SQLSTATE value |
| SQLEXCEPTION | SQLSTATE class other than 00, 01, or 02 |
| SQLWARNING | SQLSTATE class 01 |
| NOT FOUND | SQLSTATE class 02 |

The following is an example of a condition declaration:

BEGIN

DECLARE constraint\_violation CONDITION

FOR SQLSTATE VALUE '23000' ;

END ;

This example is not realistic, because typically the SQL statement that may cause the condition to occur — as well as the handler that would be invoked if the condition did occur — would also be enclosed within the BEGIN…END structure.

#### Handler Actions & Handler Effects

If a condition occurs that invokes a handler, the action specified by the handler executes. This action is an SQL statement, which can be a compound statement. If the handler action completes successfully, then the handler effect executes. The following is a list of the three possible handler effects:

* CONTINUE: Continue execution immediately after the statement that caused the handler to be invoked.
* EXIT: Continue execution after the compound statement that contains the handler.
* UNDO: Undo the work of the previous statements in the compound statement and then continue execution after the statement that contains the handler.

If the handler can correct whatever problem invoked the handler, then the CONTINUE effect may be appropriate. The EXIT effect may be appropriate if the handler didn't fix the problem, but the changes made to the compound statement do not need to be undone. The UNDO effect is appropriate if you want to return the database to the state it was in before the compound statement started execution. Consider the following example:

BEGIN ATOMIC

DECLARE constraint\_violation CONDITION

FOR SQLSTATE VALUE '23000' ;

DECLARE UNDO HANDLER

FOR constraint\_violation

RESIGNAL ;

INSERT INTO students (StudentID, Fname, Lname)

VALUES (:sid, :sfname, :slname) ;

INSERT INTO roster (ClassID, Class, StudentID)

VALUES (:cid, :cname, :sid) ;

END ;

If either of the INSERT statements causes a constraint violation, such as trying to add a record with a primary key that duplicates a primary key already in the table, SQLSTATE assumes a value of '23000', thus setting the constraint\_violation condition to a true value. This action causes the handler to UNDO any changes that have been made to any tables by either INSERT command. The RESIGNAL statement transfers control back to the procedure that called the currently executing procedure.

If both INSERT statements execute successfully, execution continues with the statement following the END keyword.

The ATOMIC keyword is mandatory whenever a handler's effect is UNDO. This is not the case for handlers whose effect is either CONTINUE or EXIT.

#### Conditions That Aren’t Handled

In the example in the preceding section, consider this possibility: What if an exception occurred that returned an SQLSTATE value other than '23000'? Something is definitely wrong, but the exception handler that you coded can't handle it. What happens now?

Because the current procedure doesn’t know what to do, a RESIGNAL occurs. This bumps the problem up to the next higher level of control. If the problem isn’t handled there, it continues to be elevated to higher levels until either it is handled or it causes an error condition in the main application.

**WARNING** The idea that I want to emphasize here is that if you write an SQL statement that may cause exceptions, then you must write exception handlers for all such possible exceptions. If you don’t, you will have more difficulty isolating the source of a problem when it inevitably occurs.

#### Assignment

With SQL/PSM, SQL gained a function that even the lowliest procedural languages have had since their inception: the ability to assign a value to a variable. Essentially, an assignment statement takes the following form:

SET target = source ;

In this usage, target is a variable name, and source is an expression. Several examples include the following:

SET vfname = 'Joss' ;

SET varea = 3.1416 \* :radius \* :radius ;

SET vWIMPmass = NULL ;

#### Flow of Control Statements

Since its original formulation in the SQL-86 standard, one of the main drawbacks that has prevented people from using SQL in a procedural manner has been its lack of flow of control statements. Until SQL/PSM was included in the SQL standard, you couldn't branch out of a strict sequential order of execution without reverting to a host language like C or Java. SQL/PSM introduces the traditional flow of control structures that other languages provide, thus allowing SQL programs to perform needed functions without switching back and forth between languages.

#### IF…THEN…ELSE…END IF

The most basic flow of control statement is the IF…THEN…ELSE…END IF statement. This statement, roughly translated from computerese, means IF a condition is true, then execute the statements following the THEN keyword. Otherwise, execute the statements following the ELSE keyword. For example:

IF

vfname = 'Joss'

THEN

UPDATE students

SET Fname = 'Joss'

WHERE StudentID = 314159 ;

ELSE

DELETE FROM students

WHERE StudentID = 314159 ;

END IF

In this example, if the variable vfname contains the value 'Joss', then the record for student 314159 is updated with 'Joss' in the Fname field. If the variable vfname contains any value other than 'Joss', then the record for student 314159 is deleted from the STUDENTS table.

The IF…THEN…ELSE…END IF statement is great if you want to choose one of two actions based on the value of a condition. Often, however, you want to make a selection from more than two choices. At such times, you should probably use a CASE statement.

#### CASE…END CASE

CASE statements come in two forms: the simple CASE statement and the searched CASE statement. Both kinds allow you to take different execution paths based on the values of conditions.

#### Simple CASE Statement

A simple CASE statement evaluates a single condition. Based on the value of that condition, execution may take one of several branches. For example:

CASE vmajor

WHEN 'Computer Science'

THEN INSERT INTO geeks (StudentID, Fname, Lname)

VALUES (:sid, :sfname, :slname) ;

WHEN 'Sports Medicine'

THEN INSERT INTO jocks (StudentID, Fname, Lname)

VALUES (:sid, :sfname, :slname) ;

WHEN 'Philosophy'

THEN INSERT INTO skeptics (StudentID, Fname, Lname)

VALUES (:sid, :sfname, :slname) ;

ELSE INSERT INTO undeclared (StudentID, Fname, Lname)

VALUES (:sid, :sfname, :slname) ;

END CASE

The ELSE clause handles everything that doesn't fall into the explicitly named categories in the THEN clauses.

You don’t need to use the ELSE clause — it's optional. However, if you don’t include it, and the CASE statement’s condition is not handled by any of the THEN clauses, SQL returns an exception.

#### Searched CASE Statement

A searched CASE statement is similar to a simple CASE statement, but it evaluates multiple conditions rather than just one. For example:

CASE

WHEN vmajor

IN ('Computer Science', 'Electrical Engineering')

THEN INSERT INTO geeks (StudentID, Fname, Lname)

VALUES (:sid, :sfname, :slname) ;

WHEN vclub

IN ('Amateur Radio', 'Rocket', 'Computer')

THEN INSERT INTO geeks (StudentID, Fname, Lname)

VALUES (:sid, :sfname, :slname) ;

WHEN vmajor

IN ('Sports Medicine', 'Physical Education')

THEN INSERT into jocks (StudentID, Fname, Lname)

VALUES (:sid, :sfname, :slname) ;

ELSE

INSERT INTO skeptics (StudentID, Fname, Lname)

VALUES (:sid, :sfname, :slname) ;

END CASE

You avoid an exception by putting all students who are not geeks or jocks into the SKEPTICS table. Because not all nongeeks and nonjocks are skeptics, this may not be strictly accurate in all cases. If it isn't, you can always add a few more WHEN clauses.

#### LOOP…ENDLOOP

The LOOP statement allows you to execute a sequence of SQL statements multiple times. After the last SQL statement enclosed within the LOOP…ENDLOOP statement executes, control loops back to the first such statement and makes another pass through the enclosed statements. The syntax is as follows:

SET vcount = 0 ;

LOOP

SET vcount = vcount + 1 ;

INSERT INTO asteroid (AsteroidID)

VALUES (vcount) ;

END LOOP

This code fragment preloads your ASTEROID table with unique identifiers. You can fill in other details about the asteroids as you find them, based on what you see through your telescope when you discover them.

Notice the one little problem with the code fragment in the preceding example: It is an infinite loop. No provision is made for leaving the loop, so it will continue inserting rows into the ASTEROID table until the DBMS fills all available storage with ASTEROID table records. If you're lucky, the DBMS will raise an exception at that time. If you’re unlucky, the system will merely crash.

For the LOOP statement to be useful, you need a way to exit loops before you raise an exception. That way is the LEAVE statement.

#### LEAVE

The LEAVE statement works just like you might expect it to work. When execution encounters a LEAVE statement embedded within a labeled statement, it proceeds to the next statement beyond the labeled statement. For example:

AsteroidPreload:

SET vcount = 0 ;

LOOP

SET vcount = vcount + 1 ;

IF vcount > 10000

THEN

LEAVE AsteroidPreload ;

END IF ;

INSERT INTO asteroid (AsteroidID)

VALUES (vcount) ;

END LOOP AsteroidPreload

The preceding code inserts 10,000 sequentially numbered records into the ASTEROID table and then passes out of the loop.

#### WHILE…DO…END WHILE

The WHILE statement provides another method of executing a series of SQL statements multiple times. While a designated condition is true, the WHILE loop continues to execute. When the condition becomes false, looping stops. For example:

AsteroidPreload2:

SET vcount = 0 ;

WHILE

vcount< 10000 DO

SET vcount = vcount + 1 ;

INSERT INTO asteroid (AsteroidID)

VALUES (vcount) ;

END WHILE AsteroidPreload2

This code does exactly the same thing that AsteroidPreload did in the preceding section. This is just another example of the often-cited fact that with SQL, you usually have multiple ways to accomplish any given task. Use whichever method you feel most comfortable with, assuming your implementation allows both.

#### REPEAT…UNTIL…END REPEAT

The REPEAT loop is very much like the WHILE loop, except that the condition is checked after the embedded statements execute rather than before. For example:

AsteroidPreload3:

SET vcount = 0 ;

REPEAT

SET vcount = vcount + 1 ;

INSERT INTO asteroid (AsteroidID)

VALUES (vcount) ;

UNTIL vcount = 10000

END REPEAT AsteroidPreload3

Although you can perform the same operation three different ways (with LOOP, WHILE, and REPEAT), you will encounter some instances when one of these structures is clearly better than the other two. Have all three methods in your bag of tricks so that when a situation like this arises, you can decide which one is the best tool available for the situation.

#### FOR…DO…END FOR

The SQL FOR loop declares and opens a cursor, fetches the rows of the cursor, executes the body of the FOR statement once for each row, and then closes the cursor. This loop makes processing possible entirely within SQL, instead of switching out to a host language. If your implementation supports SQL FOR loops, you can use them as a simple alternative to the cursor processing described in [Chapter 20](http://viewer.books24x7.com/assetviewer.aspx?bkid=145080&destid=1067#1067). Here's an example:

FOR vcount AS Curs1 CURSOR FOR

SELECT AsteroidID FROM asteroid

DO

UPDATE asteroid SET Description = 'stony iron'

WHERE CURRENT OF Curs1 ;

END FOR

In this example, you update every row in the ASTEROID table by putting 'stony iron' into the Description field. This is a fast way to identify the compositions of asteroids, but the table may suffer some in the accuracy department. Perhaps you'd be better off checking the spectral signatures of the asteroids and then entering their types individually.

#### ITERATE

The ITERATE statement provides a way to change the flow of execution within an iterated SQL statement. The iterated SQL statements are LOOP, WHILE, REPEAT, and FOR. If the iteration condition of the iterated SQL statement is true or not specified, then the next iteration of the loop commences immediately after the ITERATE statement executes. If the iteration condition of the iterated SQL statement is false or unknown, then iteration ceases after the ITERATE statement executes. For example:

AsteroidPreload4:

SET vcount = 0 ;

WHILE

vcount< 10000 DO

SET vcount = vcount + 1 ;

INSERT INTO asteroid (AsteroidID)

VALUES (vcount) ;

ITERATE AsteroidPreload4 ;

SET vpreload = 'DONE' ;

END WHILE AsteroidPreload4

Execution loops back to the top of the WHILE statement immediately after the ITERATE statement each time through the loop until vcount equals 9999. On that iteration, vcount increments to 10000, the INSERT performs, the ITERATE statement ceases iteration, vpreload is set to 'DONE', and execution proceeds to the next statement after the loop.

#### Stored Procedures

Stored procedures reside in the database on the server rather than execute on the client — where all procedures were located before SQL/PSM. After you define a stored procedure, you can invoke it with a CALL statement. Keeping the procedure located on the server rather than on the client reduces network traffic, thus speeding performance. The only traffic that needs to pass from the client to the server is the CALL statement. You can create this procedure in the following manner:

EXEC SQL

CREATE PROCEDURE ChessMatchScore

( IN score CHAR (3),

OUT result CHAR (10) )

BEGIN ATOMIC

CASE score

WHEN '1-0' THEN

SET result = 'whitewins' ;

WHEN '0-1' THEN

SET result = 'blackwins' ;

ELSE

SET result = 'draw' ;

END CASE

END ;

After you have created a stored procedure like the one in this example, you can invoke it with a CALL statement similar to the following statement:

CALL ChessMatchScore ('1-0', :Outcome) ;

The first argument is an input parameter that is fed to the ChessMatchScore procedure. The second argument is an embedded variable that accepts the value assigned to the output parameter that the ChessMatchScore procedure uses to return its result to the calling routine. In this case, it returns 'white wins'.

SQL:2011 added a couple of enhancements to stored procedures. The first of these is the introduction of named arguments. Here's the equivalent of the preceding call, with named arguments:

CALL ChessMatchScore (result => :Outcome,score =>'1-0');

Because the arguments are named, they can be written in any order without a danger of them being confused.

The second enhancement added in SQL:2011 is the addition of default input arguments. You can specify a default argument for the input parameter. After you do that, you don’t need to specify an input value in the CALL statement; the default value is assumed. (Of course, you would want to do this only if the default value were in fact the value you wanted to send to the procedure.)

Here’s an example of that usage:

EXEC SQL

CREATE PROCEDURE ChessMatchScore

( IN score CHAR (3) DEFAULT '1-0',

OUT result CHAR (10) )

BEGIN ATOMIC

CASE score

WHEN '1-0' THEN

SET result = 'whitewins' ;

WHEN '0-1' THEN

SET result = 'blackwins' ;

ELSE

SET result = 'draw' ;

END CASE

END ;

You can now call this procedure thusly with the default value:

CALL ChessMatchScore (:Outcome) ;

#### Stored Functions

A stored function is similar in many ways to a stored procedure. Collectively, the two are referred to as *stored routines.* They are different in several ways, including the way in which they are invoked. A stored procedure is invoked with a CALL statement, and a stored function is invoked with a *function call,* which can replace an argument of an SQL statement. The following is an example of a function definition, followed by an example of a call to that function:

CREATE FUNCTION PurchaseHistory (CustID)

RETURNS CHAR VARYING (200)

BEGIN

DECLARE purch CHAR VARYING (200)

DEFAULT '' ;

FOR x AS SELECT \*

FROM transactions t

WHERE t.customerID = CustID

DO

IF a <>''

THEN SET purch = purch || ', ' ;

END IF ;

SET purch = purch || t.description ;

END FOR

RETURN purch ;

END ;

This function definition creates a comma-delimited list of purchases made by a customer that has a specified customer number, taken from the TRANSACTIONS table. The following UPDATE statement contains a function call to PurchaseHistory that inserts the latest purchase history for customer number 314259 into her record in the CUSTOMER table:

SET customerID = 314259 ;

UPDATE customer

SET history = PurchaseHistory (customerID)

WHERE customerID = 314259 ;

#### Privileges

I discuss the various privileges that you can grant to users in [Chapter 14](http://viewer.books24x7.com/assetviewer.aspx?bkid=145080&destid=803#803). The database owner can grant the following privileges to other users:

* The right to DELETE rows from a table
* The right to INSERT rows into a table
* The right to UPDATE rows in a table
* The right to create a table that REFERENCES another table
* The right of USAGE on a domain

SQL/PSM adds one more privilege that can be granted to a user — the EXECUTE privilege. Here are two examples:

GRANT EXECUTE on ChessMatchScore to TournamentDirector ;

GRANT EXECUTE on PurchaseHistory to SalesManager ;

These statements allow the tournament director of the chess tournament to execute the ChessMatchScore procedure, and the sales manager of the company to execute the PurchaseHistory function. People lacking the EXECUTE privilege for a routine aren't able to use it.

#### Stored Modules

A stored module can contain multiple routines (procedures and/or functions) that can be invoked by SQL. Anyone who has the EXECUTE privilege for a module has access to all the routines in the module. Privileges on routines within a module can’t be granted individually. The following is an example of a stored module:

CREATE MODULE mod1

PROCEDURE MatchScore

( INscore CHAR (3),

OUT result CHAR (10) )

BEGIN ATOMIC

CASE result

WHEN '1-0' THEN

SET result = 'whitewins' ;

WHEN '0-1' THEN

SET result = 'blackwins' ;

ELSE

SET result = 'draw' ;

END CASE

END ;

FUNCTION PurchaseHistory (CustID)

RETURNS CHAR VARYING (200)

BEGIN

DECLARE purch CHAR VARYING (200)

DEFAULT '' ;

FOR x AS SELECT \*

FROM transactions t

WHERE t.customerID = CustID

DO

IF a <>''

THEN SET purch = purch || ', ' ;

END IF ;

SET purch = purch || t.description ;

END FOR

RETURN purch ;

END ;

END MODULE ;

The two routines in this module (a procedure and a function) don’t have much in common, but they don’t need to. You can gather related routines into a single module, or you can stick all the routines you’re likely to use into a single module, regardless of whether they have anything in common.

-----------------------------------------------------------------------------------------

## Domain

### Pre-Assessment

Q1: Object Names (ID 27833) = V

What is not allowed in object names in SQL?

Symbols, Numbers, Spaces, Special Characters

Spaces

Q2: Removing a Table (ID 27834) = V

Which command removes a table from a database?

CLEAR TABLE, DROP TABLE, ELIMINATE TABLE, DELETE TABLE

DROP TABLE

Q3: View (ID 27835) = V

Which statement best defines a view?

A stored statement that leads one to specific information

A location within a database that stores specific information

A stored statement that leads one to general information

A location within a database that stores general information

**A stored statement that leads one to specific information**

Q4: Specifying Parameters (ID 27836) = X

What type of parameter must be specified when creating a stored procedure?

Boolean, Number, Input, Output

Boolean

Q5: Index Fields (ID 27837) = V

While a clustered index is based on a \_\_\_, a nonclustered index can be based on any other field.

Foreign Key Field, Primary Key Field, Column, Row

Primary Key Field

### Post-Assessment

Q1: Creating a Table (ID 27838) = V

Which three pieces of information should be specified when creating a table?

Rows, Columns, Data Types for each Column, Table Name, Data Types for each Row

**Columns, Data Types for each Column, Table Name**

Q2: Changing a Table (ID 27839) = V

What is the only way to make changes to a table structure after a table has been created?

UPDATE, Make changes & then CREATE, CREATE NEW TABLE, ALTER

**ALTER**

Q3: Column Values (ID 27840) = V

By default, \_\_\_ values are allowed in columns when tables are created.

date, decimal, not null, null

**null**

Q4: Allowing Null Values (ID 27841) = V

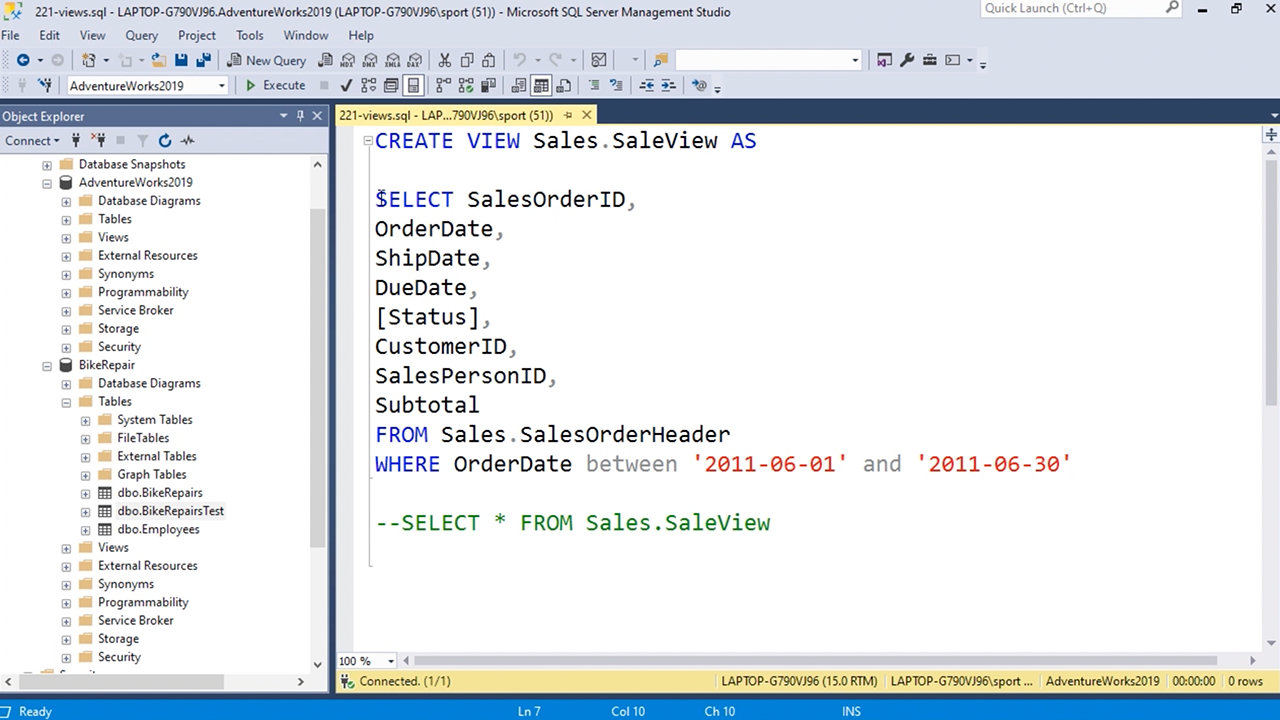
How should one change a column from not allowing null values to allowing null values?

New Table via CREATE, New Column via CREATE, ALTER, Restore Column to Default Settings via DEFAULT

**ALTER**

Q5: Creating a View (ID 27842) = V

Refer to the image. What will happen when this code is executed?



The entire code runs, the code will not run due to a syntax error, the first SELECT will not run, the second SELECT will not run

**The second SELECT will not run.**

Q6: Changing a View (ID 27843) = V

To change a view, with what keyword should CREATE in the CREATE VIEW command be replaced?

UPDATE, ALTER, CHANGE, REPLACE

**REPLACE**

**CHANGE**

**UPDATE**

**ALTER**

Q7: Returning Values in Apps (ID 27845) = V

What type of parameter is used in apps to return a value that the app will use later?

Boolean, Number, Input, Output

**INPUT**

**OUTPUT**

Q8: Seeing Results of a Query (ID 27846) = V

What function must be included in a stored procedure to see the results of a function that runs inside of the stored procedure?

RETURN, REVIEW, VIEW, RESULT

**RETURN**

Q9: Stored Procedure Purpose (ID 27847) = V

What is one main security benefit of stored procedures being used within app code?

Stored procedure hides SQL keywords from being exposed in app code

Stored procedure accepts parameters

Stored procedure allows for filtering of data

Stored procedure has permissions

**Stored procedure accepts parameters**

**Stored procedure hides SQL keywords from being exposed in app code**

Q10: Creating an Index (ID 27849) = V

Create a clustered index for the dbo.BikeRepairs table, and generate the code used to create the index in a new window:

**Right-click Indexes in the dbo**

**Click New Index, then Clustered Index**

**Click Unique checkbox**

**Click Add & click First Column from Selection**

**Click Script**

**Click New Query Editor Window**

**Click OK**

Q11: Index Syntax (ID 27851) = V

Which piece of information goes in parentheses when creating a nonclustered index in SQL?

The NON-CLUSTERED keyword

The table on which the index is being created

The columns being indexed

The number of total columns

**The table on which the index is being created**

**The number of total columns**

**The columns being indexed**

Q12: Index (ID 27848) = V

Which best defines the purpose of an index?

Helps speed up queries within a table as long as the indexes are created with strategic purpose

Ensures that fewer people are needed to manage a database

Creates new tables automatically

Stores SQL code within a database so it can be quickly accessed when creating new tables

**Helps speed up queries within a table as long as the indexes are created with strategic purpose**

Q13: Purpose of a View (ID 33419) = V

The main purpose of a view is that it can accept parameters.

**False**

## Videos

### How to Create a Table (9m)

**Table Basics:**

A table is a two-dimensional array consisting of rows and columns:

Create a table using the SQL CREATE TABLE command

Parameters include the name and data type of each column

Use the ALTER TABLE command to change the structure of the table

The DROP command can delete the table upon request

**Example of Command to Create a Table called CUSTOMER:**

CREATE TABLE CUSTOMER

(

CustomerID INTEGER NOT NULL,

FirstName VARCHAR(15),

LastName VARCHAR(20) NOT NULL,

Street VARCHAR(25),

City VARCHAR(20),

State CHAR(2),

Zipcode INTEGER,

Phone CHAR(13)

)

**Key Points when Creating Tables:**

Ensure that every table in the database has at least one common column:

This helps to relate one table to another with a common identifier

Identify all tables that will be used before creating them

Define the columns that each table will need

Give each table a primary key

Refine each table to Third Normal Form (3NF)

**Single and Multi-Table Views:**

**Single-Table View:**

Useful when information exists in a single table

Example would be a list of names and phone numbers of persons from one city:

**CREATE VIEW CITYNAME\_CUST AS**

**SELECT CUSTOMER.FirstName,**

**CUSTOMER.LastName,**

**CUSTOMER.Phone**

**FROM CUSTOMER**

**WHERE CUSTOMER.City = ‘CITYNAME’**

**Multi-Table View:**

This requires more than one table and is useful when requiring relation between tables and retrieving information from tables that have commonalities.

**Diagram of Single-Table View:**

| **CUSTOMER TABLE** |  | **CityName\_CUST View** |
| --- | --- | --- |
| CustomerID |  |  |
| FirstName | **=>** | FirstName |
| LastName | **=>** | LastName |
| Street |  |  |
| City | **=>** | City |
| State |  |  |
| Zipcode |  |  |
| Phone | **=>** | Phone |

### Changing an Existing Table (7m)

**– – Use ALTER TABLE to add a new column**

ALTER TABLE Customers ADD

MobilePhone varchar(25) NULL

**– – Use ALTER TABLE to add a new column**

ALTER TABLE Customers

DROP COLUMN MobilePhone

**– – Use ALTER TABLE to remove the primary key constraint**

ALTER TABLE Customers

DROP CONSTRAINT PK\_Customers

### Deleting an Existing Table (2m)

**– – Use DROP TABLE to remove the entire table**

DROP TABLE test\_table\_1

### Creating a View (8m)

Views, Right-click & New View

Add Tables to be Queried

Customers, Order\_Details & Orders

### Updating Views in SQL (9m)

Altering = changing the design for a View

Updating = changing the records

### Dropping Views (4m)

DROP VIEW Customers\_View, Orders\_View

Check for Dependencies prior to dropping

Data remains in tables

### Understanding Stored Procedures (8m)

**Definitions:**

A named collection of T-SQL statements

Ideal for repetitive T-SQL tasks

Encapsulate repetitive program logic

Accept input parameters

Return output parameters

**Advantages:**

**Security**

Encryption

**Performance**

Compilation

(hold code in a central repository)

**Reduce Amount of Data Passing over a Network**

Keep code on server

**Hide Raw Data**

Stored procedures can gain access to data, but the data itself need not be exposed to the user or app.

**System Stored Procedures:**

**Activities:**

Administrative

Informational

**Can enhance basic functionality of SQL Server:**

Extend functionality

Create new functionality

**User Stored Procedures:**

Any procedure that is stored and compiled in SQL Server, but created by a user

**Types:**

**EXEC** - manually executed stored procedures

**Triggers** - automatically executed stored procedures (triggered via modification event)

In conclusion, stored procedures can:

Accept and produce parameters

Call other procedures

Return a status value to a calling procedure or batch

Indicate success or failure

Reason for failure

Return values of parameters

Procedure

Batch

Be executed on remote servers

## Assignment

Please upload screenshots of your solutions for any 2 of 4 Exercises:

**Exercise 1:**

You are the owner of a Software Development company named Quality Software.

You have customers (who are companies) based throughout Europe.

You hire Software Development contractors (self-employed individuals) to work on customer projects.

In this exercise, you will create a database with two tables, one to store details of your

customers, and another table to store details of your contractors.

**Part A:** Create a database named *Quality\_Software*

**Part B:** Create the following tables with the stated fields:

**Customers**

**customerId int NOT NULL PRIMARY KEY,**

**companyName varchar(50) NOT NULL,**

**address1 varchar(50) NOT NULL,**

**address2 varchar(50) NOT NULL,**

**city varchar(50) NOT NULL,**

**country varchar(50) NOT NULL,**

**contactName varchar(50) NOT NULL,**

**contactEmail varchar(50) NOT NULL**

**Contractors**

**contractorId int NOT NULL PRIMARY KEY,**

**firstName varchar(50) NOT NULL,**

**lastName varchar(50) NOT NULL,**

**address1 varchar(50) NOT NULL,**

**address2 varchar(50) NOT NULL,**

**city varchar(50) NOT NULL,**

**country varchar(50) NOT NULL,**

**contactPhone varchar(50) NOT NULL,**

**contactEmail varchar(50) NOT NULL**

**Part C:** Insert the following records:

**Customers**

(1, ‘German Motors’, ‘22 Strabe Munchen’, ‘Munchen Klien’, ‘Munich’, ‘Deutscheland’, ‘Michael Schmidt’, ‘[michaelschmidt@gmmotors.com](mailto:michaelschmidt@gmmotors.com)’)

(2, ‘Das Auto’, ‘100 Munchen Business Park’, ‘Munchen 2323’, ‘Munich’, ‘Deutscheland’, ‘Philip Kroos’, ‘[philipkroos@dasauto.com](mailto:philipkroos@dasauto.com)’)

(3, ‘Danone’, ‘15 Parkway House’, ‘Leicester Road’, ‘Leicester’, ‘England’, ‘Jamie Taylor’, ‘[jamietaylor@danone.com](mailto:jamietaylor@danone.com)’)

(4, ‘Nestle’, ‘13 Rue de Paris’, ‘Paris’, ‘Paris 201212’, ‘France’, ‘Michele Gaulle’, ‘[michelegaulle@nestle.com](mailto:michelegaulle@nestle.com)’)

**Contractors**

(1, ‘Hans’, ‘Christian’, ‘12 Strabe Munchen’, ‘Munchen Klien’, ‘Munich’, ‘Deutscheland’, ‘4987565656’, ‘[hanschristian@hotmail.com](mailto:hanschristian@hotmail.com)’)

(2, ‘Karl Heinz’, ‘Vakkel’, ‘10 New Forrest Strabe’, ‘Munchen Grob’, ‘Munich’, ‘Deutscheland’, ‘4987549543’, ‘[karlheinz@hotmail.com](mailto:karlheinz@hotmail.com)’)

(3, ‘Anne’, ‘Leicester’, ‘54 Emmerdale Road’, ‘Emmerdale Farm’, ‘Emmerdale’, ‘England’, ‘055454545’, ‘[anneleicester@hotmail.com](mailto:anneleicester@hotmail.com)’)

(4, ‘Peter’, ‘Rummeniege’, ‘13 Hamburg Way’, ‘Hamburg’, ‘Hamburg’, ‘Germany’, ‘4987545454’, ‘[peterrummeniege@hotmail.com](mailto:peterrummeniege@hotmail.com)’)

**Part D:** Update the following records:

**Contractors**

For the contractor with the contractorId value of 2, change the email address to read: [karlheinz@gmail.com](mailto:karlheinz@gmail.com)

**Customers**

For the customer with the customerId value of 3, change the company name to *Group Danone*

**Part E:** Delete the following record:

In the Contractors table, delete the record where the field contractorId has a value of 3.

## Quiz

Q1: Fill in the blanks:

A \_\_\_ is created automatically when you create a table with a primary key, whereas a \_\_\_ can be added or removed at a later time.

**clustered index, nonclustered index**

Q2: Fill in the blanks:

In a stored procedure, a value that is passed in is known as an \_\_\_ \_\_\_\_.

**input parameter**

A parameter that is created to return a value from the stored procedure is known as an \_\_\_ \_\_\_.

**output parameter**

The \_\_\_ \_\_\_ is used to immediately end a stored procedure by returning the flow of control to the caller of the stored procedure.

**RETURN statement**

Q3: What is being described below? Use the dropdown menu to fill in the gap:

\_\_\_ are used to speed up database queries. They help to retrieve data from a database more quickly than otherwise. The end user cannot see them.

**Indexes**

Q4: How should one change a column from not allowing null values to allowing null values?

Restore the column to its default setting with the DEFAULT command.

Create a new table with the CREATE command.

Use the ALTER TABLE command followed by the ALTER COLUMN command.

Create a new column with the CREATE command.

**Use the ALTER TABLE command followed by the ALTER COLUMN command.**

Q5: Would it be a good idea to specify a column storing first names as a primary key in a table?

Yes, that would be fine.

No, because a primary key value has to be unique.

**No, because a primary key value has to be unique.**

Q6: What type of index is described below?

The index does not contain a pointer to the data but instead contains the actual data - 'the index is the table'. In the real world, how data is stored in a telephone book would be an analogy for this type of database index.

Nonclustered index

Clustered index

**Clustered index**

Q7: How many clustered indexes can you have in a table?

5

50

More than 50

1

**1**

Q8: True or False; you can have more than one nonclustered index in a table.

**False**

**True**

Q9: If you would like to improve the performance of searches on columns other than the primary key column(s), which type of index would you create?

Nonclustered

Clustered

**Clustered**

**Nonclustered**

Q10: What is the only way to make changes to a table structure after a table has been created?

Use the ALTER command to make changes to the table

Make the changes, then run the CREATE command again

Create a new table to replace the old one

Use the UPDATE command to make changes to the table

**Use the ALTER command to make changes to the table**

Q11: What is one difference between a stored procedure and a view?

A stored procedure can accept parameters

A view is a saved query

A stored procedure is more time consuming to create

A stored procedure does not use SQL

**A stored procedure can accept parameters**

Q12: To edit a view, which of the following keywords should be used?

UPDATE VIEW

ALTER VIEW

RECREATE VIEW

REPLACE VIEW

**ALTER VIEW**

Q13: You wish to create some SQL code that is stored on a database server. When run, the SQL code should query a table named Employees. The SQL code should have one input parameter (employee name) and one output parameter (employee id).

What database feature is being described?

Stored procedure

Referential Integrity

View

ERD

**Stored procedure**

Q14: What database design feature is being described below?

Can represent a subset of the data contained in a table

Can hide the complexity of data

Can join and simplify multiple tables into a single virtual table

Cannot accept input parameters

Appropriate privileges can be granted - the user need not be given privileges on underlying base objects

Stored Procedure

View

**View**

Q15: Assuming that a table called Customers exists in a database, and it stores 5 records. And given the following statement, which of the following answers is correct?

– – SELECT \* FROM Customers;

There is a syntax error in the command & it doesn’t execute

5 records will be returned after the command executes

0 records will be returned after the command executes

As the command is commented out, it will not be executed

**As the command is commented out, it will not be executed**

Q16: What are previously written SQL statements that have been stored within a database?

Data statements

Views

DDL statements

Stored procedures

**Stored procedures**

Q17: What command is used to execute a stored procedure?

USE

START

GO

EXEC

**EXEC**

Q18: What is the best reason to create an index?

To reduce the execution time of queries in small databases

To reduce the execution time of queries in large databases

To enforce referential integrity between two tables in a database

To map out the entities/tables to be included in a database

**To reduce the execution time of queries in large databases**

Q19: Which type of index is described below?

It is like the index found at the back of a textbook where the index contains a pointer to where the data is contained within the book.

Clustered index

Nonclustered index

**Nonclustered index**

Q20: Is the following statement true or false; a nonclustered index will cause overhead, as it has to be maintained for every update, insert and delete made in the associated table.

**True**

Q21: Is the following statement true or false; by default, SQL Server creates a clustered index on the specified primary key when you create a table.

**True**

Q22: Is the following statement true or false; the following CREATE TABLE command fails to execute successfully because columns are not allowed to have null as a value.

CREATE TABLE Customer (

ID int NOT NULL,

LastName varchar(255) NOT NULL,

FirstName varchar(255) NULL,

Age int,

PRIMARY KEY (ID)

);

**False**