A PRIMER ON SQL

3rd Edition

By Rahul Batra

A Primer on SQL

Third Edition

Rahul Batra

This book is for sale at http://leanpub.com/aprimeronsql

This version was published on 2015-02-25

Also By Rahul Batra

A Primer on Java

To Mum and Dad

Contents

1. An Introduction to SQL

A **database** is nothing but a collection of organized data. It doesn't have to be in a digital format to be called a database. A telephone directory is a good example, which stores data about people and organizations with a contact number. Software which is used to manage a digital database is called a **Database Management System (DBMS)**.

The most prevalent database organizational model is the **Relational Model**, developed by Dr. E F Codd in his groundbreaking research paper - *A Relational Model of Data for Large Shared Data Banks*.In this model, data to be stored is organized as rows inside a table with the column headings specifying the corresponding type of data stored. This is not unlike a spreadsheet where the first row can be thought of as column headings and the subsequent rows storing the actual data.

What does the word *relational* in relational database mean?

It is a common misconception that the word relational implies relationship between the tables. A relation is a mathematical term that is roughly equivalent to a table itself. When used in conjunction with the word database, we mean to say that this particular system arranges data in a tabular fashion.

SQL stands for **Structured Query Language** and it is the de-facto standard for interacting with relational databases. Almost all database management systems you'll come across will have a SQL implementation. SQL was standardized by the American National Standards Institute (ANSI) in 1986 and has undergone many revisions, most notably in 1992 and 1999. However, all DBMS's do not strictly adhere to the standard defined but rather remove some features and add others to provide a unique feature set. Nonetheless, the standardization process has been helpful in giving a uniform direction to the vendors in terms of their database interaction language.

1.1 SQL Commands Classification

SQL is a language for interacting with databases. It consists of a number of commands with further options to allow you to carry out your operations with a database. While DBMS's differ in the command subset they provide, usually you would find the classifications below.

- **Data Definition Language (DDL)** : *CREATE TABLE, ALTER TABLE, DROP TABLE etc.*

  These commands allow you to create or modify your database structure.

- **Data Manipulation Language (DML)** : *INSERT, UPDATE, DELETE*

  These commands are used to manipulate data stored inside your database.
- **Data Query Language (DQL)** : *SELECT*

  Used for querying or selecting a subset of data from a database.
- **Data Control Language (DCL)** : *GRANT, REVOKE etc.*

  Used for controlling access to data within a database, commonly used for granting user privileges.
- **Transaction Control Commands** : *COMMIT, ROLLBACK etc.*

  Used for managing groups of statements as a unit of work.

Besides these, your database management system may give you other sets of commands to work more efficiently or to provide extra features. But it is safe to say that the ones above would be present in almost all DBMS's you encounter.

1.2 Explaining Tables

A ***table*** in a relational database is nothing but a matrix of data where the columns describe the type of data and the row contains the actual data to be stored. Have a look at the figure below to get a sense of the visualization of a table in a database.

Figure: a table describing Programming Languages

id	language	author	year
1	Fortran	Backus	1955
2	Lisp	McCarthy	1958
3	Cobol	Hopper	1959

The above table stores data about programming languages. It consists of 4 columns (id, language, author and year) and 3 rows. The formal term for a column in a database is a **field** and a row is known as a **record**.

There are two things of note in the figure above. The first one is that, the *id* field effectively tells you nothing about the programming language by itself, other than its sequential position in the table. The second is that though we can understand the fields by looking at their names, we have not formally assigned a data type to them i.e. we have not restricted (not yet anyways) whether a field should contain alphabets or numbers or a combination of both.

The *id* field here serves the purpose of a **primary key** in the table. It makes each record in the table unique and its advantages will become clearer in chapters to come. But for now consider this, what if a language creator made two languages in the same year; we would have a difficult time narrowing down on the records. An *id* field usually serves as a good primary key since it's guaranteed to be

unique, but usage of other fields for this purpose is not restricted.

Just like programming languages, SQL also has **data types** to define the kind of data that will be stored in its fields. In the table given above, we can see that the fields *language* and *author* must store English language characters. Thus their data type during table creation should be specified as **varchar** which stands for *variable number of characters*.

The other commonly used data types you will encounter in subsequent chapters are:

Fixed length characters	*char*
Integer values	*int*
Decimal numbers	*decimal*
Date data type	*date*

2. Getting your database ready

2.1 Using Ingres

The best way to learn SQL is to practice writing commands on a real relational database. In this book SQL is taught using a product called **Ingres**. The reasons for choosing Ingres are simple - it comes in a free and open source edition, it's available on most major platforms and it's a full-fledged enterprise class database with many features. However, any relational database product that you can get your hands on should serve you just fine. There might be minor incompatibilities between different vendors, so if you choose something else to practice on while reading this book, it would be a good idea to keep the database vendor's user manual handy.

Since this text deals largely with teaching SQL in a product independent manner, rather than the teaching of Ingres per se, details with respect to installation and specific operations of the product will be kept to a minimum. Emphasis is instead placed on a few specific steps that will help you to get working on Ingres as fast as possible.

The current version of Ingres during the writing of the book was **10.1** and the **Community Edition** has been used on a Windows box for the chapters to follow. The installation itself is straightforward like any other Windows software. However if you are unsure on any option, ask your DBA (database administrator, in case one is available) or if you are practicing on a home box - select the 'Traditional Ingres' mode and install the Demo database when it asks you these questions. Feel free to refer to the Ingres installation guide that is available on the web at the following location. Ingres Installation Guide[1]

If your installation is successful, you should be able to start the *Ingres Visual DBA* from the Start Menu. This utility is a graphical user interface to manage your Ingres databases, but we will keep the usage of this to a minimum since our interest lies in learning SQL rather than database administration.

2.2 Using SQLite

If installing Ingres seems like a daunting task, you are in luck. There is a very credible, free alternative database for you to practice on. It is called **SQLite** and it's creator D. Richard Hipp has generously licensed it in the public domain. You can download it from the SQLite Download page[2].

If you are using Microsoft Windows, you are looking for the section titled *Precompiled Binaries for Windows*. Download the SQLite DLL zip archive, named like *sqlite-dll-win32-x86-xxxxxxx.zip*,

[1] http://docs.actian.com/ingres/10.0/installation-guide
[2] http://sqlite.org/download.html

which contains SQLite but not a way to interact with it. For that you must download the SQLite shell, named like *sqlite-shell-win32-x86-xxxxxxx.zip*, which will allow us to create and query SQLite databases through the command line.

Extract both these archives into the same directory and you are done installing SQLite. Your folder should now contain atleast three files - *sqlite3.dll, sqlite3.def, sqlite3.exe*. The last one launches the command shell used to interact with SQLite databases.

2.3 Creating your own database

Most database management systems, including Ingres, allow you to create multiple databases. For practice purposes it's advisable to create your own database, so that you are free to perform any operations on it.

Most database systems differ in the way they provide database creation facilities. Ingres achieves the same by providing you multiple ways to do this, including through the Visual DBA utility. However for didactic purposes, we will instead use a command operation to create our database. Open up the *Ingres Command Prompt* from the program menu (usually found inside Start Menu->Programs->Ingres for Microsoft Windows systems), and enter the command as below.

Listing: using createdb and its sample output

```
C:\\Documents and Settings\\rahulb>createdb testdb
Creating database 'testdb' . . .
  Creating DBMS System Catalogs . . .
  Modifying DBMS System Catalogs . . .
  Creating Standard Catalog Interface . . .
  Creating Front-end System Catalogs . . .
Creation of database 'testdb' completed successfully.
```

The command ***createdb*** is used to create a database which will serve as a holding envelope for your tables. In the example and output shown above, we created a database called *testdb* for our use. You (or more specifically your system login) are now the owner of this database and have full control of entities within it. This is analogous to creating a file in an operating system where the creator gets full access control rights and may choose to give other users and groups specific rights.

If you are using SQLite, fire up the command shell and you will be greeted with a window with the text displayed below.

```
SQLite version 3.8.2 2013-12-06 14:53:30
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
```

Here we enter our *.open* command to both create a SQLite database or open it in case it already exists.

```
.open testdb
```

If you are using Linux, SQLite does not come with the *.open* command on it. Instead you directly write the database name on the terminal immediately after the interactive SQL shell program name like below.

```
sqlite3 testdb
```

Turning on column headers in SQLite

SQLite, by default, does not display column headers in the output of a query. This being a very useful visual helper, I usually turn it on using by two commands below (to be executed inside the SQLite shell).

```
sqlite> .mode column sqlite> .headers on
```

2.4 Table Creation

We have already explored the concept of a table in a relational model. It is now time to create one using a standard SQL command - *CREATE TABLE*.

The SQL standard by definition allows commands and keywords to be written in a case insensitive manner. In this book we would use uppercase letters while writing them in statements, which is a widely accepted practice.

Listing: General Syntax of a CREATE TABLE statement

```
CREATE TABLE <Table_Name>
(<Field 1> <Data Type>,
 <Field 2> <Data Type>,
 \. \. \.
 <Field N> <Data Type>);
```

This is the simplest valid statement that will create a table for you, devoid of any extra options. We'll further this with clauses and constraints as we go along, but for now let us use this general syntax to actually create the table of programming languages we introduced in Chapter 1.

The easiest way to get started with writing SQL statements in Ingres is to use their **Visual SQL** application which gives you a graphical interface to write statements and view output. The usual place to find it on a Windows system is Start -> Programs -> Ingres -> Ingres II -> Other Utilities.

When you open it up, it gives you a set of dropdown boxes on the top half of the window where you can select the database you wish to work upon and other such options. Since we'll be using the same database we created previously (testdb), go ahead and select the options as specified below.

Default User	
Default Server	INGRES
Database	testdb

The actual SQL statement you would be writing to create your table is given below.

Listing: Creating the programming languages table

```
CREATE TABLE proglang_tbl (
id        INTEGER,
language  VARCHAR(20),
author    VARCHAR(25),
year      INTEGER);
```

Press the 'Go' or F5 button when you're done entering the statement in full. If you get no errors back from Visual SQL, then congratulations are in order since you've just created your first table.

The statement by itself is simple enough since it resembles the general syntax of *CREATE TABLE* we discussed beforehand. It is interesting to note the data types chosen for the fields. Both *id* and *year* are specified as integers for simplicity, even though there are better alternatives. The *language* field is given a space of 20 characters to store the name of the programming language while the *author* field can hold 25 characters for the creator's name.

The semicolon at the last position is the delimiter for SQL statements and it marks the end of a statement.

The same CREATE TABLE statement also works fine for SQLite and is written in the SQLite command shell itself.

2.5 Inserting data

The table we have just created is empty so our task now becomes insertion of some sample data inside it. To populate this data in the form of rows we use the DML command INSERT, whose general syntax is given below.

Listing: General syntax of INSERT TABLE

```
INSERT INTO <Table Name>
VALUES ('Value1', 'Value2', ...);
```

Fitting some sample values into this general syntax is simple enough, provided we keep in mind the structure of the table we are trying to insert the row in. For populating the **proglang_tbl** with rows like we saw in chapter 1, we would have to use three *INSERT* statements as below.

Listing: Inserting data into the proglang_tbl table

```
INSERT INTO proglang_tbl VALUES (1, 'Fortran', 'Backus', 1955);
INSERT INTO proglang_tbl VALUES (2, 'Lisp', 'McCarthy', 1958);
INSERT INTO proglang_tbl VALUES (3, 'Cobol', 'Hopper', 1959);
```

If you do not receive any errors from Ingres Visual SQL (or the SQL interface for your chosen DBMS), then you have managed to successfully insert 3 rows of data into your table. Notice how we've carefully kept the ordering of the fields in the same sequence as we used for creating our table. This strict ordering limitation can be removed and we will see how to achieve that in a little while.

2.6 Writing your first query

Let us now turn our attention to writing a simple query to check the results of our previous operations in which we created a table and inserted three rows of data into it. For this, we would use a Data Query Language (DQL) command called *SELECT*.

A *query* is simply a SQL statement that allows you to retrieve a useful subset of data contained within your database. You might have noticed that the *INSERT* and *CREATE TABLE* commands were referred to as statements, but a fetching operation with *SELECT* falls under the query category.

Most of your day to day operations in a SQL environment would involve queries, since you'd be creating the database structure once (modifying it only on a need basis) and inserting rows only when new data is available. While a typical *SELECT* query is fairly complex with many clauses, we will begin our journey by writing down a query just to verify the contents of our table. The general syntax of a simple query is given below.

Listing: General Syntax of a simple SQL query

```
SELECT <Selection> FROM <Table Name>;
```

Transforming this into our result verification query is a simple task. We already know the table we wish to query - **proglang_tbl** and for our selection we would use * (star), which will select all rows and fields from the table.

```
SELECT * FROM proglang_tbl;
```

The output of this query would be all the (3) rows displayed in a matrix format just as we intended. If you are running this through Visual SQL on Ingres, you would get a message at the bottom saying - *Total Fetched Row(s): 3.*