VEEAM
IT JUST WORKS!™

SQL Server 101 – What is SQL Server and how does it work?

Tibor Karaszi
Consultant, SQL Server expert

AVAILABILITY™
for the Modern Data Center

The purpose of this paper is to provide you background on SQL Server. We will explain what SQL Server is, where it came from and its architecture and building blocks.

Relational Database Management System, RDBMS

An RDBMS represents data in what we usually call tables. There are formal terms as well, but we won't use those terms in this paper. A table has a name, and is a structure where you have one or more columns – each column also has a name. The data is represented as rows. A row has a value for each column (or NULL meaning that the value is missing). A row is supposed to represent an entity (an order, a customer, etc.). While a table represents a set of entities (orders or customers). A database is a collection of tables. When you operate on the data (retrieve data, for instance), you work with sets. For instance with the SQL language, a SELECT statement returns a number of rows (0, 1 or more).

SQL, the language

In the 1970s, a language was developed for a prototype of a Relational Database Management System. This language was later called SQL. Some say SQL stands for Structured Query Language, while others think it is simply three letters put together. The SQL language consists of many commands (loosely speaking) that allow us to create, secure, manage and operate over the data in our tables. We sometimes see a categorization of commands in the SQL language:

DDL, Data Definition Language

This category was originally meant to include the commands with which we build the database. For instance, the CREATE TABLE command. And also DROP TABLE and ALTER TABLE (allow us to add a column, for example). Over time, the DDL category came to include all types of CREATE, ALTER and DROP commands (DATABASE, LOGIN, USER, VIEW, PRODECURE, etc.).

DCL, Data Control Language

Here we have the commands with which we define permissions. Who is allowed to do what, optionally with the data. We have the GRANT, DENY and REVOKE commands in this category. A couple of examples:

```
GRANT CREATE TABLE TO Sue

GRANT SELECT ON Customers TO Joe
```

DML, Data Manipulation Language

These are the commands we use to add, remove, modify and read rows in our tables. The commands (also known as statements) are INSERT, DELETE, UPDATE and SELECT. These are the commands sent by your applications to SQL Server when you use your application to add orders, look at customers, etc.

There are other commands as well in the SQL language which do not fit in above categorization. For instance, there is a command to start a transaction or to check that a database is free from corruptions.

SQL was originally an industry standard but it was quickly standardized formally by the likes of ANSI, ISO, etc. The standard is revised now and then, where new language elements can be added and even old elements removed. Each product has its own dialect of the SQL language, where it implements parts of the SQL standard (which is huge) and adds its own language elements. Additions might be added in order to be competitive, or even just useable. For instance, the standard doesn't say a word about indexes, which is an important performance structure.

There is a fair amount of critique of the SQL language where some people mean it was a quick and dirty implementation of a language for a relational database system. Many feel that SQL isn't even a relational language. It is safe to say that the SQL language, in some aspects, doesn't do the relational model justice, as it doesn't allow us to explore the full potential of a relational database system. But SQL, "flawed" or not, has certainly proven itself to be useful.

Background and history of SQL Server

In the 1980s, Sybase had an RDBMS called SQL Server for the Unix environment. "PCs" were growing in popularity and they wanted to get into the PC market. So they worked with Ashton-Tate (Dbase) and Microsoft to port and release the product for the PC architecture. At this point in time, the operating system for PC servers was OS/2. Version 1.0, 1.1 and 4.2 were released for OS/2.

Then came Windows NT (what we call today "Windows") so the product was ported to this new and hot operating system. In 1992, shortly after Windows NT was released, Microsoft also released SQL Server 4.2 for Windows NT. Microsoft then separated from Sybase and developed SQL Server on their own. Later, 6.0 and 6.5 was released with no major changes or additions in these versions.

Then, Microsoft re-wrote the whole product, which we might refer to as "the new architecture." This was a while ago — SQL Server 7.0 was released 1998. The goal was a storage structure that could last two or three decades and considering that we are still using this architecture, it is safe to say it was a successful re-architecture of the product. In 7.0, we also had additions outside the database engine. Microsoft added OLAP Server for cubes and DTS for data export and import. With SQL Server 2000, we also got Reporting Services.

SQL Server 2005 had some pretty significant changes to the product although not as major as 7.0. For instance, 2005 was when SQL Server Management Studio was released. Those of you who have ever worked with SQL Server over the last decade have probably been using this tool. Another significant change was a whole new meta-data structure where Microsoft changed the system table structure totally, from for instance sysobjects (which was a physical table) to sys.tables (which is a view, the physical table is hidden from us). As for the other components, OLAP Server was renamed to Analysis Services and DTS was renamed to SQL Server Integration Services (SSIS) — both re-written. This paper focuses on the database engine, or "the SQL engine."

The more recent versions of SQL Server are 2008, 2008 R2, 2012 and 2014. Each of these of course can have added potentially significant additions to the product – but I like to think of them more as evolutions of the product. The fact is, if you worked with 2005, you will more than likely feel at home with 2014.

Architecture of SQL Server, the database engine

Note that for the remainder of this paper, when we say "SQL Server," we refer to the database engine (if not otherwise specified).

SQL Server is a process, viewed from an operating system perspective. Memory allocated from the operating system is owned by the process and so are files that are opened, and the process has a number of threads that the operating system can schedule for execution. SQL Server was started as a service. If you check "Services" in your operating system, you may see several services where the friendly name starts with "SQL." The following is a breakdown of the most common SQL Server services. Note that the explanations are not meant to be exhaustive but more of an idea of what each service entails.

SQL Server

This is the database engine.

SQL Server Agent

We will refer to this simply as "Agent." Agent allows you to schedule jobs for execution at specified times or intervals. A job consists of one or more commands to be executed, and these commands can be SQL commands, operating system commands, a PowerShell or VB script, etc. Agent also has alerting capabilities. You can define that Agent will send an email or execute a job when certain conditions are met. These conditions for instance can be an error written by the database engine to the windows event log, or a SQL Server performance monitor counter reaching some specified value.

SQL Server Analysis Services

SQL Server Analysis Services is also known as SSAS and allows you to build cubes over data in a data warehouse, or as of SQL Server 2012, use the tabular model instead of cubes. SSAS also has data mining functionality, allowing you to work over large amounts of data, using various algorithms to find patterns and trends that exist in data.

SQL Server Browser

This will be defined and discussed later in the text.

SQL Server Integration Services

This service is involved when executing SSIS packages, i.e. performing data transfer and such operations.

SQL Server Reporting Services

Reporting Service allows you to use a tool to design reports and upload such reports to the Reporting Services service. This service stores the reports in a database and publishes them either through a website or a SharePoint site.

SQL Server VSS Writer

This component allows some software to "freeze" the I/O over one or more databases so the software can grab the data from the database file in a consistent state and produce a "snapshot" over the data in your SQL Server. Veeam is an example of a software that can use VSS to produce such a snapshot.

Instances

For some of the above components (services), you can have more than one installed in an operating system. We call each installed component an *instance*. The components in question are the database engine, Analysis Services and Reporting Services. You can see by the service name whether it is a component that allows us to have several instances — there is a parenthesis and an instance name after the friendly service name. One instance can be a default instance (where you see MSSQLSERVER in parenthesis), and you can have several named instances for which you specify the instance name when you install that instance.

Each instance is separated from each other, except that they of course live in the same operating system. Each instance also has its own folder structure, for instance, where the exe file is stored. This allows them to be different versions and build numbers. You can have totally different configurations at the instance level and also different security settings (including who are "super administrators," sysadmin, etc.).

Database and storage structures

Each SQL Server instance has several databases. A database is created using the CREATE DATABASE command (or using "New Database" in SSMS, which in turn uses the CREATE DATABASE command). However, there are five databases in SQL Server from the beginning:

master

The master database is the "root" database, so to speak. There are certain configuration information which is common to the whole instance which is stored in the master database. Examples are logins (sys.logins), configuration settings (sys.configurations) and linked servers (sys.servers). Master is special from another perspective as well — the startup, or *boot*, perspective. When you start your SQL Server it reads command-line parameters from the registry. Some of those parameters are the full path and name to the two files that make your master database (master.mdf and mastlog.ldf).

msdb

The msdb database is mainly used by SQL Server Agent. There are tables in this database designed to store your scheduled jobs, alerts, operators and anything else Agent works with. It is, however, a common misconception that msdb is *only* used by Agent. The database engine also uses msdb. For instance, each time you perform a backup or restore, information about that operation is written to a number of tables in the msdb database.

model

When you create a database, SQL Server uses a template for that database. This template includes items such as database option settings. And yes, you have guessed it – this template database is of course the *model* database.

tempdb

As the name implies, this database is used for temporary storage. There are a number of things that uses tempdb, including:

- Worktables used during execution of a SQL query
- Explicitly created temporary tables (tables beginning with #) and table variables
- Row versioning, where a transaction can get to the prior value for a row which is being modified by somebody else, instead of being blocked or doing a dirty read
- The inserted and deleted tables that you have available in a trigger that fires for INSERT, UPDATE or DELETE

The tempdb database is re-created every time you start your SQL Server. The database file structure (number of database files, size and path) is not picked from the model database, however – it is based on a system table – visible using master.sys.master_files.

The resource database

This resource database is in fact hidden from us. We cannot see this database in the Object Explorer in SQL Server Management Studio nor does it show in sys.databases. It is hidden because it is not supposed to be altered. I once heard someone from Microsoft suggest to "think of it as a DLL file." It only contains code, not data. But what code you may ask? The code that is your system stored procedure (like sp_detach_db, sp_help, etc) and also the system views (sys.databases, sys.tables, etc) are included in the resource database. This allow Microsoft to replace these files when you patch your SQL Server instead of running script files to re-create these objects in the master database (like we had in SQL Server 2000 and earlier).

Database files

Each database is stored over a set of files. Files are internally numbered from file one on up (see for instance sys.database_files). We have at least two files for each database; one data file (.mdf) and one transaction log file (.ldf). We can have more than one data file (.ndf) and also more than one transaction log file (.ldf).

A data file belongs to a filegroup. We have a minimum of one filegroup for each database – the PRIMARY filegroup. When you create a table or an index, you can specify what filegroup will be used in the CREATE command. There's always a default filegroup, which will be PRIMARY unless you change the default filegroup. If you do not specify what filegroup a table or index should be created on, then it will be created on the default filegroup.

You can have more than one data file for each database. The primary (.mdf) always belongs to the PRIMARY filegroup. Each filegroup can have more than one data file. If you have more than one file in the same filegroup then the files will be filled up proportionally to their sizes.

In the end, the filegroup concept allows you to specify on what file - or files - a table will be stored, such as if you have one table for which you know you will have heavy access. The table is of moderate size, large enough so it won't fit in RAM by the standard caching mechanism in SQL Server. But you also have lots of other data in the database. Also, let's say you have a fast disk (SSD) of moderate size that you can utilize on this server. Create a filegroup, add one file on this SSD disk and create your heavily accessed table on this SSD disk (the filegroup with the file which is on this SSD disk). This is only one example of what you can use several filegroups for, and another is backup at the filegroup level, or performing database consistency checks at the filegroup level (DBCC CHECKFILEGROUP instead of DBCC CHECKDB).

However, the vast majority of databases have only one filegroup and even only one data file (the .mdf file). And, in most cases, this is perfectly sufficient. I'm a strong believer in the KISS concept (keep it simple) and if more than one filegroup fails to give you advantages, then why should you have it?

Data file architecture

Each data file is divided into a set of pages, 8 kB in size. Pages are internally numbered from page 0 and up. A page can be unused, i.e. a free page. Pages are grouped into eight consecutive pages referred to as an "extent." An extent can also be unused if none of the pages on the extent are in use. We sometimes refer to this type of extent as "unallocated" (as opposed to allocated).

An allocated extent can be allocated in two ways. Either as a shared (or mixed) extent. On a shared extent you find pages from different tables and indexes, hence the name shared (or mixed). Also, an extent can essentially be owned by a certain table or index, what we call a "uniform extent." The first eight pages for each table or index comes from shared extents and subsequent pages come from uniform extents.

We have assumed two types of allocations in this discussion, data and indexes. There are other types as well, such as LOB allocations for instance, but data and index pages are enough to illustrate the page and extent allocation principle in SQL Server.

Table and index architecture

Logically, a table is defined over a number of columns. Each column has a name and is a certain data type.

From a physical perspective, a table has rows stored in pages. However, the table (the data) can be structured in two different ways:

A Heap table

A heap table is a table for which we have not created a clustered index. SQL Server doesn't store the rows in any particular order. Rows and pages are essentially stored where there is free space. Needless to say, SQL Server tries to be conservative and not use more pages than necessary and will try to have as many rows on each page, within reasonable limits.

A clustered table

A clustered table is a table over which you have created a clustered index. This index might have been created automatically since SQL Server by default will create a clustered index on the Primary Key (column), however you can override this. Or, you can create the clustered index explicitly using the CREATE CLUSTERED INDEX command.

Now, this begs the question: What is an index?

Let's first look at this from the clustered index perspective. An index sorts the data. For example, let's say you represent people in a table, with columns such as first name, last name, city, etc. Also, let's say you define the clustered index over the last name column (last name being the index key). The rows are now stored in order of last name. Imagine a set of pages (for this index), and in the header of the first page you have the address (page number) to the next page. This then repeats until the last page. A page also points back to the previous page. This is, what we call a doubly linked list. The row with the lowest last name will be the first row on the first page, and vice versa. In other words, you can follow the linked list from the beginning to the end and you will have read the person in order of the last name. What we have described here is the leaf level of the index.

We also have a tree structure above this level. Take the first value of the index key (last name) for the first page, and the same for the second page, etc. Store these on another set of pages, along with the page number they point to. You now have the level above the leaf level. If one page isn't sufficient for this level then you keep building higher levels until you have exactly one root page. This is how SQL Server builds the index tree.

Non-clustered indexes

The description above describes a clustered index. As we know, a table can have a clustered index, or it might not have a clustered index (in which case it is a heap table). Regardless of which, we can also have a number of non-clustered indexes for the table. A non-clustered index is also defined over one or more columns (say first name, for our previous example). SQL Server builds the non-clustered index pretty much the same way as the clustered index, but in the leaf level of the index we don't have all columns for the row (the data). Instead, we only have the index key column(s), which in our example is the first name column along with a "pointer" to the data row. If the table is a heap table, we will have the file, page and row number for the row. If the table is a clustered table, we will instead have the clustering key column values.

You might realize that there is plenty more we could say about these things, digging deeper into index structures and going deeper into index options. However, the above description is sufficient to understand what data in tables are used and the fact that we can create an index that SQL Server can use when finding data. Imagine if SQL Server had to read all the pages that the table uses, just to find the rows your query is looking for!

The optimizer and execution plans

The SQL language, a SELECT statement for instance, operates at a logical level. You specify **what** you want, not **how** it should be performed. For instance, loosely speaking, "give me all rows with the last name Johnson." We want this to be executed as quickly and efficiently as possible by SQL Server. For this, we have an optimizer.

The optimizer is responsible for creating an execution plan. The execution plan defines **how** the query should be executed, like for instance, what index should be used to find the rows for which we are searching. It also helps determine what order to join tables, if the query combines data from more than one table. Or, how join(s) should be performed; there are various ways that joins can be performed, such as hash join, loop join and merge join. We can go on and on. The execution plan consists of a series of operators. Imagine a rather simple query where the most inner operator reads data from the table or index and passes the rows out the next outer operator - which might for instance sort the data if you imagine an ORDER BY in the query.

There are various ways to get a representation of this execution plan. For instance, there are options to see a graphical representation of the execution plan in SQL Server Management Studio ("Show estimated execution plan" and "Show actual execution plan"). When you work with performance tuning, you are interested in execution plans, how the query is executed – what indexes are used etc., and whether we can somehow make SQL Server run a query faster, for instance using less system resources.

Keeping track of used and free space

SQL Server keeps track of used versus free extents in each data file.

The third page, or page two (page numbers starts with 0), in the file is a GAM (Global Allocation Map) page and this has a bit for each extent in the following approximate 4 GB of the database file. GAM keeps track of whether the extent is in use (as shared or uniform) with 0 for the extent, or if it is free (unallocated) with the value 1.

The fourth page (page number three) is the SGAM (Secondary Global Allocation Map) page. The SGAM comes into play when we think about shared extents. An extent allocated as a mixed extent with free pages has 1 in the SGAM, else it has 0.

This makes it easy for SQL Server to find free storage. Imagine that SQL Server needs a free (unallocated) extent. SQL Server scans the GAM page and finds an extent having 1. Or, imagine that SQL Server needs a page from a shared extent. SQL Server scans the SGAM page and finds an extent having 1.

In other words, an extent can be represented in the GAM and SGAM combinations (in that order) by 1, 0 meaning this is a free, unallocated extent. Or, we can have 0, 0 meaning either a uniform extent or a mixed extent with no free pages. Finally, we can have 0, 1 meaning a shared extent with free pages. The combination 1, 1 is not used.

As mentioned, the GAM and SGAM pages represent approximately 4 GB in the database file, and then we have another set of GAM and SGAM pages, and so on. The GAM and SGAM pages are always at fixed positions in the database files. How many pairs of GAMs and SGAMs there are depends on the size of the database file.

But how can we know what pages a table or index are using, or more precisely, a heap or an index? Every heap and every index has an IAM (Index Allocation Map) page. The IAM page also maps approximately 4 GB in the database file and we have one for each extent that this heap or index is using. IAM pages are of course not at fixed positions, so SQL Server keeps track of the address (page number) of the first IAM page at the heap or index level. If the heap or index uses extents across a larger area than 4 GB in the database files, then we have more IAM pages and pointers from the first IAM page to the next one.

Finally, we have PFS (Page Free Space) pages. A PFS keeps track of how full a page is, approximately. The first PFS is the second page (page number 1) in each data file. The PFS map approximate 64 MB in the data file. And they are repeated for each additional 64 MB portion. Each byte (not bit) in the PFS represents one page in that 64 MB area, with roughly how much free space is included on this page. The PFS is not maintained for index pages, since that information is not of interest in the first place – when you insert a new row in an index, we always know where to put this row, in the right position according to the index key..

So, there you have it — this is a look at the allocation architecture for SQL Server. Now, you might wonder whether or not we have to know or care about these things? No, in most cases you can happily administer and program your SQL Server with no knowledge about GAM, SGAM, IAM and PFS's. However, an understanding can help you understand some error messages more effectively, index fragmentation, or just to de-mystify the engine that is "under the hood" of SQL Server.

If you were to dig further you will discover more details. For example:

- A heap or index can be partitioned (having more than one partition) and you will see that IAMs are actually at the partition level.
- There are more allocation types (again, IAMs) than heap and index. We have LOB pages, pages for data types such as varchar(max), nvarchar(max), varbinary(max), text, ntext and image. And there are also allocations for row-overflow pages – we can have a combination of variable length data types for columns so that a row no longer fits on one column and this is when SQL Server stores some of the column values on such row-overflow pages.

If you want to dig deeper into this area, I recommend the SQL Server documentation (SQL Server Books Online, BOL) as a good starting point. Unfortunately, Microsoft has decided to no longer maintain some of the architectural sections of the product documentation, but we can use the SQL Server 2008 R2 version of BOL which is still accurate:

https://technet.microsoft.com/en-us/library/cc280361(v=sql.105).aspx

Transaction logging

As you know, each database has at least one file for its transaction log, the ldf file. You can have more than one, but this won't give you any performance impact since they will be used one after the other serially. Here's a brief description of what happens when you modify data in SQL Server:

Every modification is always done in a transaction. Among other things, a transaction is defined as a number of modifications that should either be formed all or none — an *atomic* operation. By default, a single modification statement, such as an INSERT, UPDATE or DELETE, will be performed within its own transaction – meanwhile, if anything fails, while the modification command is being executed, then everything performed until that point, for that modification command, will be rolled back.

You can also group several modification commands inside the same transaction, using commands such as BEGIN TRANSACTION, COMMIT TRANSACTION and ROLLBACK TRANSACTION.

When a transaction is started, either implicitly using a modification command, or explicitly by a BEGIN TRANSACTION command, SQL Server will record in the transaction log that this session has started a transaction.

For each modification (a row is inserted, updated, deleted, similar for index modifications, etc.), SQL Server will make sure the modified page is in cache. Every read and modification is served from cache. If it isn't in cache already then the page will be read from disk and be brought into cache. A log record is constructed to reflect the modification, and written to the transaction log (not necessarily physically to the file yet). Now, the page can be modified in cache. This happens for each modification within this transaction. And in regards to an end of transaction, such as a commit, SQL Server will reflect the commit in the transaction log and also make sure all log records produced up until the specific point in time are physically written to the disk ("force log write at commit"). This is why you want to have good write performance where the ldf file is located, since the writing to the ldf file is synchronous writes – the application will wait until the operating system and SQL Server has acknowledged that the write operation has been performed.

The pages that have been modified are dirty at this point. They have been modified since brought into cache and don't look the same in cache as on disk. SQL Server performs checkpoints now and then where it writes all dirty pages (for the database) to disk, and also reflects that in the transaction log This gives SQL Server a starting point for recovery, for instance when you start SQL Server. It will find the most recent checkpoint in the transaction log and use the transaction log to make sure that all modifications recorded in the log have actually been made to the data pages, but also roll back any transactions that were incomplete when you stopped your SQL Server. We sometimes refer to this as REDO and UNDO. You can see information about this recovery process reflected in the SQL Server errorlog file, from when you started your SQL Server.

Tools

You get several tools to help you manage SQL Server. The following is a list of the most commonly used and most important tools included.

SQL Server Management Studio (SSMS)

SSMS is quite simply a graphical user interface allowing you to manage your SQL Server. You can connect to a SQL Server (or some of the other components) and use Object Explorer to view your databases, tables, views etc. You can right-click on an object and get a context-menu allowing you to perform various tasks against that object. And you can open query windows, allowing you to type SQL queries and commands and execute them — and of course also save these as files, "script files."

SQL Server Configuration Manager

This tool allows you to configure your server and perform changes that in general are stored outside of SQL Server (mostly in the Windows Registry) instead of inside SQL Server. These settings include items such as what Windows account each service is started as, startup parameters and network settings for your SQL Server.

SQLCMD.EXE

SQLCMD is a command-line tool for executing queries against a SQL Server. You can do this by either specifying the query as a command-line option, read them from an input-file or use the tool interactively where you get a prompt from which you can type and use GO to send the queries to your SQL Server.

SQL Server Documentation (also known as Books Online)

This is exactly what it sounds like: the documentation for SQL Server. By default, this will take you to a website for the documentation. This has several limitations. The navigation is very slow compared to a local help application and the search functionality is managed by Bing, not limited to only SQL Server.

You can change the documentation to a local application using the tool "Manage Help Settings." Change this to local and then download the parts you are most interested in using the same tool.

APIs, the client and network architecture

In this context, a client application ("client" for short) is an application that logs on to your SQL Server instance and submits SQL queries to the instance. For example, this can be SSMS, SQLCMD, or whatever application that utilizes SQL Server to store and manage its data. Regardless of what this application is, there are some common grounds, regarding to how it communicate with your SQL Server.

The developers who created and programmed this application had to have a way to submit SQL queries from the application to your SQL Server instance. For example, let's say we are using a language like C#. We can't just have SQL queries in our source code and expect our (C#) compiler to understand this. We need to use a library (loosely speaking) with which we can send our SQL to SQL Server and after execution read the results returned from SQL Server. This is commonly known as our database API (Application Programming Interface).

There have been many database APIs over the years including Db Library, ADO, RDO, ODBC, OLEDB, ADO.NET, and JDBC to mention a few. The actual implementation of this API can vary, but it is common to implement it as one or more DLL files. From the programmer's perspective, it can be, for instance, function calls in a DLL, or a class library. Some APIs might be limited in usage from a certain programming environment. ADO.NET, for instance, is only available in the .Net programming environment.

Whichever API is used, typically, the DLL that implements the API needs to be able to send the SQL command from the client application to your SQL Server. This is done using what we call a "Network Library," or Netlib. This needs to be implemented in both the client machine and the server machine. On the server side, we sometimes refer to this as an *Endpoint*. Nowadays, there are three different Netlibs:

Shared Memory

This is only usable when the client application is in the same operating system as the SQL Server instance. Due to its architecture (a shared memory area), it is the fastest Netlib.

Named Pipes

This is pretty much outdated since it uses NetBIOS. NetBIOS is something we have been trying to discard from our environments the last couple of decades.

TCP/IP Sockets

This is what we typically use. This is also the default Netlib, the one that a client will try first – assuming the server isn't on the same machine as the client (where it will try Shared Memory first).

Port numbers

When I say "SQL Server" and "port number" to somebody, they either look at me like I'm from Mars or they say 1433.

A default instance will, by default (it is changeable) listen on port 1433. And a client application, when you address the SQL Server instance using only the machine name, host name or IP address, will try 1433. That is how the client finds the right instance on your server machine. But what if you have several SQL Servers? Remember that we can have only one default instance - only one listening to 1433.

A named instance will pick a free port by asking the operating system when it first starts. This port number is saved in the registry and when you start it next time, it will try to use the same port number. You can see what port an instance is using by looking at the startup information in the SQL Server "errorlog" file, or using the SQL Server Configuration Manager tool – where you also can change the port number if you wish. But we don't connect using the port number, you might say. We connect using the instance name, such as MachineName\InstanceName. Obviously, we have something that translates the instance name to a port number. This is the SQL Server Browser service. When you have the "\InstanceName" part in your connection, the client libraries send a request to the server machine using port 1434 (UDP), and the SQL Server Browser service on the server machine replies to the client with the port number that instance is listening on. We can connect using the port number instead of instance name using MachineName,PortNumer (or host name or IP address instead of machine name).

Security and the logon process

We need a way to identify people using SQL Server and make sure they can perform what they need, but not more. Or at least this is what we should aim for.

As a starter you need some identity in SQL Server and a way to authenticate yourself as that identity. In short, we are referring to some logon or login process. There are two types of logins in SQL Server:

SQL Server Logins

When you create a SQL Server Login, you specify a name and a password. SQL Server will store the name and a hash of the password. Since SQL Server only stores a hashed version of the password, there is no way to "decrypt" the password, even if you find the stored bits on disk. When a client connects to a SQL Server instance using a SQL Server login, the login name and password is always sent encrypted from the client to the server, encrypted by the client libraries. The server libraries decrypts them, hashes the password, makes sure the name exists and the hashed password sent by the client matches with the stored hash. This is sometimes known as a *non-trusted connection*. The most known SQL login is *sa*. This exists from the beginning and is member of the sysadmin role. In my opinion, we should never use the *sa* login — it should have a complex password, possibly renamed and definitely disabled. The employee that needs system administrator privileges on your SQL Server should have their own login for such purposes instead of using the *sa* login.

Windows Logins

When you create a Windows Login, you specify a user, or a group, in your Windows environment (in your domain, most likely). SQL Server will grab the SID (Security Identifier), from Windows and store the name and SID. When you connect using a Windows Login, through a *trusted connection*, SQL Server verifies that the SID the user is represented as, exists as a login in SQL Server, or any of the groups that the user is a member. If you are allowed to connect to SQL Server using a Windows login which is a group, we can still identify the Windows user name inside SQL Server. This is important because we don't lose the ability to track who is connected or who did what.

Server roles

At the server level, there are roles you can add a login as member to. There's the *public* role which every login is a member. This can be used to grant permission that should apply to everyone. Then there are eight *fixed* server roles. You cannot change the permissions that come with a fixed server role, but you can of course decide who (if anyone) should be a member of that role. Probably, the most known server role is *sysadmin*. As a sysadmin, you can do anything everywhere in your SQL Server instance. This should of course be used very carefully. Think of it as SQL Server's equivalent to *Domain Admin*. Examples of other fixed server roles are *dbcreator* and *processadmin*.

As of SQL Server 2012, we can also create our own server roles as well as add members and grant permissions to the role instead of to individual logins.

Gaining access to a database – the user

A login only allows you to connect to the instance. This is meaningless unless you can also access the database(s) you need. This is the user concept. Sometimes we think of this like *mapping a login to a database*, or granting access for the login to the database. But what we are really doing is creating a user (with a name) in the database to, "*point to,*" the login. This allows the login to access the database. You can then grant permission to this user, so they can, for instance, SELECT from a table. In most cases, we of course have the same name for the user as we have for the login. The connection for the user to the login is made, though, using the SID for the login. A Windows' login SID comes from Windows and a SQL Server's login SID is generated by SQL Server when you create that login. There is always a user named *dbo* in each database. The login who owns a database has the *dbo* user in that database.

Database roles

Just like the server roles, we also have database roles. We can assign a database role to a user and the permissions granted to this database roles are now also available for that user. There's a public database role and every user always has this role – you can grant permissions to this if you want the permissions to apply to all users in the databases. And there are also fixed database rows. *Db_owner* gives the same privileges as the dbo. Examples of other fixed database roles are *db_datareader* and *db_backupoperator*. We can also create our own database roles, assign permissions to them and add users to such a role.

Permissions

Some permissions can be inherited from some of the fixed server or database roles. Other permissions you will grant specifically, either to a role (server- or database-, depending on type of permission) or directly to a login or user. In general, you can perform an operation if you have been granted privileges, unless there is a "deny" for this operation. Privileges granted accumulate and the following paragraph is an example of this:

Sue connects to SQL Server using Windows authentication. The Sue Windows account exists as a login in SQL Server. Sue is also a member of the Accounting Windows group, and that group also exist as a Windows login on your SQL Server. The Sue login has a user in database A and the Accounting login has access to database B. Sue will be able to access both databases A and B. In database A, the "Sue" user is also a member of a database role named, X. In this database, Sue will be able to perform the operations that have been granted to her, and also the operations that have been granted to the X database role. The same principle goes for the B database, of course. The exception to this is DENY, which overrides GRANT. With DENY, you **know** that a login or user cannot perform the operation in question. But there is an exception to this as well: a sysadmin can do everything in the instance. SQL Server doesn't even check DENY for somebody who is sysadmin.

Data transfer, exporting and importing data

It seems like the need for import and export has exploded over the last decade or so. We exchange data for business purposes, export for reporting, perform migrations, build data warehouses, etc. Years ago, the tools for export and import were very simple. As soon as you needed something more advanced, like understating various file formats, you needed to go outside of what came with the product. Then version 7.0 was introduced and it popularized "Data Warehousing." We needed a tool to export data from our production systems into our data warehouse. Version 7.0 introduced DTS (Data Transformation Services). Since then, we have seen various new tools, commands and improvements in this area.

Importing data, bulk loading

Importing data into a table is also known as *loading* data. There are certain code paths in SQL Server to facilitate loading data as quickly and efficiently as possible. This is different from performing an INSERT. There are various types of optimization taking place in this scenario, like potentially minimizing transaction logging, optimizations in how storage is allocated, potentially bypassing database constraints and triggers, etc. Some tools and commands designed for data loading always use the bulk loading functionality in SQL Server, while others allow you to choose whether to load the data using regular INSERTs or as bulk loading. Tools and functionality for bulk loading data include:

BCP.EXE, which is a console command (a command-line exe file). It allows you to export from a table to a file, or vice versa. It uses command-line switches to specify the table, file, direction, etc.

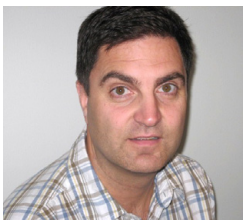
BULK INSERT is pretty much the same as BCP, but as a T-SQL command and only for import. An architectural difference between the two is BULK INSERT which is the database engine that reads the file to import. With BCP, it is the client (BCP.EXE) that reads from (or write to) the file.

SQL Server Integration Services (SSIS) is a more advanced tool for export and import. We sometimes refer to these types of tools as ETL tools, as in Extract, Transform and Load. You define the transfer using a design tool. This came with the product and was named "SQL Server Business Intelligence Development Studio" in SQL Server 2008 and "SQL Server Data Tools" in SQL Server 2012. As of SQL Server 2014, this no longer comes with the product and is a separate download. Regardless of version, the design tool is in the end a plug-in to the Visual Studio development environment.

When you create an SSIS package, you add and configure Tasks. A task performs an action, such as running an EXE file, creating a folder, executing a SQL command or sending an email. There are a number of tasks available, and you connect these using "Precedence Constraints," which basically define the sequence to execute your tasks. A special task type is the Data Flow Task, where you add data sources and destinations. And in between, you can have transformations — which can do things such as lookups, aggregations, and calculations, etc.

The package can then be executed using various methods. You can use the Execute Package Utility GUI program, DTEXEC.EXE command-line program or a SQL Server Agent job step. There is much more functionality in SSIS. You can also create SSIS packages using the easier to use but less powerful Import and Export Wizards.

About the Author



Tibor Karaszi

Tibor lives in Stockholm, Sweden. He has worked with Microsoft SQL Server since 1988. In 1997, Tibor earned the title Most Valuable Professional (MVP), as the 7th SQL Server MVP in the world. He has been honored with an MVP award each year since.

Tibor's SQL Server training experience dates back to 1991. Since then, he has delivered internal SQL Server training for Microsoft in Sweden and internationally.

He is the co-author of several books on SQL Server and was a regular contributor and editor for SQL Server Magazine. Tibor also speaks at conferences, seminars and workshops, including events organized by Microsoft. He co-manages the Swedish SQL Server user group, SQLUG.se, since its inception.

Tibor currently works as a consultant, focusing solely on Microsoft SQL Server. Assignments include training, reviewing both code and design, elaborating on issues as a discussion partner and other more hands-on consulting assignments

About Veeam Software

Veeam® recognizes the new challenges companies across the globe face in enabling the Always-On Business™, a business that must operate 24/7/365. To address this, Veeam has pioneered a new market of *Availability for the Modern Data Center*™ by helping organizations meet recovery time and point objectives (RTPO™) of less than 15 minutes for all applications and data, through a fundamentally new kind of solution that delivers high-speed recovery, data loss avoidance, verified protection, leveraged data and complete visibility Veeam Availability Suite™, which includes Veeam Backup & Replication™, leverages virtualization, storage, and cloud technologies that enable the modern data center to help organizations save time, mitigate risks, and dramatically reduce capital and operational costs.

Founded in 2006, Veeam currently has 29,000 ProPartners and more than 135,000 customers worldwide. Veeam's global headquarters are located in Baar, Switzerland, and the company has offices throughout the world. To learn more, visit http://www.veeam.com.

VEEAM

IT JUST WORKS!™

AVAILABILITY™

for the Modern Data Center

RTPO <15 min for **ALL** applications and data

Veeam Availability Suite v8

High-Speed Recovery

Data Loss Avoidance

Verified Protection

Leveraged Data

Complete Visibility

To learn more, visit **www.veeam.com**