Q) How does database indexing work?

A) **Why is it needed?**

When data is stored on disk based storage devices, it is stored as blocks of data. These blocks are accessed in their entirety, making them the atomic disk access operation. Disk blocks are structured in much the same way as linked lists; both contain a section for data, a pointer to the location of the next node (or block), and both need not be stored contiguously.

Due to the fact that a number of records can only be sorted on one field, we can state that searching on a field that isn’t sorted requires a Linear Search which requires N/2 block accesses (on average), where N is the number of blocks that the table spans. If that field is a non-key field (i.e. doesn’t contain unique entries) then the entire table space must be searched at N block accesses.

Whereas with a sorted field, a Binary Search may be used, this has log2 N block accesses. Also since the data is sorted given a non-key field, the rest of the table doesn’t need to be searched for duplicate values, once a higher value is found. Thus the performance increase is substantial.

**What is indexing?**

Indexing is a way of sorting a number of records on multiple fields. Creating an index on a field in a table creates another data structure which holds the field value, and pointer to the record it relates to. This index structure is then sorted, allowing Binary Searches to be performed on it.

The downside to indexing is that these indexes require additional space on the disk, since the indexes are stored together in a table using the MyISAM engine, this file can quickly reach the size limits of the underlying file system if many fields within the same table are indexed.

**How does it work?**

Firstly, let’s outline a sample database table schema;

Field name Data type Size on disk

id (Primary key) Unsigned INT 4 bytes

firstName Char(50) 50 bytes

lastName Char(50) 50 bytes

emailAddress Char(100) 100 bytes

**Note**: char was used in place of varchar to allow for an accurate size on disk value. This sample database contains five million rows, and is unindexed. The performance of several queries will now be analyzed. These are a query using the *id* (a sorted key field) and one using the *firstName* (a non-key unsorted field).

***Example 1***

Given our sample database of r = 5,000,000 records of a fixed size giving a record length of R = 204 bytes and they are stored in a table using the MyISAM engine which is using the default block size B = 1,024 bytes. The blocking factor of the table would be bfr = (B/R) = 1024/204 = 5 records per disk block. The total number of blocks required to hold the table is N = (r/bfr) = 5000000/5 = 1,000,000 blocks.

A linear search on the id field would require an average of N/2 = 500,000 block accesses to find a value given that the id field is a key field. But since the id field is also sorted a binary search can be conducted requiring an average of log2 1000000 = 19.93 = 20 block accesses. Instantly we can see this is a drastic improvement.

Now the *firstName* field is neither sorted, so a binary search is impossible, nor are the values unique, and thus the table will require searching to the end for an exact N = 1,000,000 block accesses. It is this situation that indexing aims to correct.

Given that an index record contains only the indexed field and a pointer to the original record, it stands to reason that it will be smaller than the multi-field record that it points to. So the index itself requires fewer disk blocks that the original table, which therefore requires fewer block accesses to iterate through. The schema for an index on the *firstName* field is outlined below;

Field name Data type Size on disk

firstName Char(50) 50 bytes

(record pointer) Special 4 bytes

**Note**: Pointers in MySQL are 2, 3, 4 or 5 bytes in length depending on the size of the table.

***Example 2***

Given our sample database of r = 5,000,000 records with an index record length of R = 54 bytes and using the default block size B = 1,024 bytes. The blocking factor of the index would be bfr = (B/R) = 1024/54 = 18 records per disk block. The total number of blocks required to hold the table is N = (r/bfr) = 5000000/18 = 277,778 blocks.

Now a search using the *firstName* field can utilise the index to increase performance. This allows for a binary search of the index with an average of log2 277778 = 18.08 = 19 block accesses. To find the address of the actual record, which requires a further block access to read, bringing the total to 19 + 1 = 20 block accesses, a far cry from the 277,778 block accesses required by the non-indexed table.

**When should it be used?**

Given that creating an index requires additional disk space (277,778 blocks extra from the above example), and that too many indexes can cause issues arising from the file systems size limits, careful thought must be used to select the correct fields to index.

Since indexes are only used to speed up the searching for a matching field within the records, it stands to reason that indexing fields used only for output would be simply a waste of disk space and processing time when doing an insert or delete operation, and thus should be avoided. Also given the nature of a binary search, the cardinality or uniqueness of the data is important. Indexing on a field with a cardinality of 2 would split the data in half, whereas a cardinality of 1,000 would return approximately 1,000 records. With such a low cardinality the effectiveness is reduced to a linear sort, and the query optimizer will avoid using the index if the cardinality is less than 30% of the record number, effectively making the index a waste of space.

Q) How to choose my primary key?

A) I believe that in practice using a [natural key](http://en.wikipedia.org/wiki/Natural_key) is rarely better than a [surrogate key](http://en.wikipedia.org/wiki/Surrogate_key).

The following are the main disadvantages of using a natural key as the primary key:

* You might have an incorrect key value, or you may simply want to rename a key value. To edit it, you would have to update all the tables that would be using it as a foreign key.
* It is often difficult to have a truly [unique](http://en.wikipedia.org/wiki/Unique_key) natural key.
* Natural keys are often strings. An index on an numeric field will be much more compact than one on a string field.

There is no hard rule on what the data type of the primary key should be. A numeric key normally performs better, but you could use a string, especially if the table is not big, and the tables that reference it are not big either.

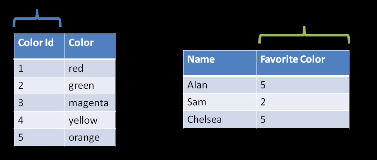
In [relational model](http://en.wikipedia.org/wiki/Relational_model) [database](http://en.wikipedia.org/wiki/Database) design, a **natural key** is a key that is formed of attributes that already exist in the real world. For example, a USA citizen's social security number could be used as a natural key. In other words, a natural key is a [candidate key](http://en.wikipedia.org/wiki/Candidate_key) that has a logical relationship to the attributes within that [row](http://en.wikipedia.org/wiki/Row_(database)). A natural key is sometimes called *domain key*.

In a [current database](http://en.wikipedia.org/wiki/Current_database), the surrogate key can be the [primary key](http://en.wikipedia.org/wiki/Primary_key), generated by the [database management system](http://en.wikipedia.org/wiki/Database_management_system) and *not* derived from any application data in the database. The only significance of the surrogate key is to act as the primary key. It is also possible that the surrogate key exists in addition to the database-generated [UUID](http://en.wikipedia.org/wiki/Universally_unique_identifier) (for example, an HR number for each employee other than the UUID of each employee).

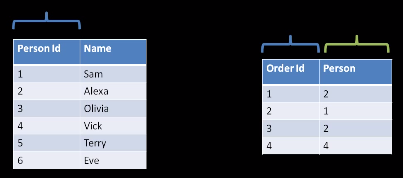
-----------------------------------------------------------------------------------------------------------------------------------

One – Many relationship

One color belongs to multiple persons

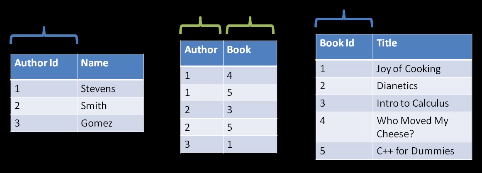


One person might make multiple orders.



Many-Many relationship

To describe this relationship, we need a third table. Each author might have multiple books like wise each book might have multiple authors.



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

Q) Why do you create a View in a database?

A) A view provides several benefits.

**1. Views can hide complexity**

If you have a query that requires joining several tables, or has complex logic or calculations, you can code all that logic into a view, then select from the view just like you would a table.

**2. Views can be used as a security mechanism**

A view can select certain columns and/or rows from a table, and permissions set on the view instead of the underlying tables. This allows surfacing only the data that a user needs to see.

**3. Views can simplify supporting legacy code**

If you need to refactor a table that would break a lot of code, you can replace the table with a view of the same name. The view provides the exact same schema as the original table, while the actual schema has changed. This keeps the legacy code that references the table from breaking, allowing you to change the legacy code at your leisure.

These are just some of the many examples of how views can be useful.

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

Q) When should you use stored procedures? When should you use a stored procedure (such as in MySQL) instead of writing OO or procedural code (such as in PHP, Ruby or Python) that may execute simple SQL queries and does other processing but performs the same task?

A) Stored procedures are generally non-portable, meaning they are specific to a particular RDBMS. As a matter of fact, stored procedures tend to be specific to a particular VERSION of a particular RDBMS.

The development tools for the lifecycle of stored procedures tend to be very limited compared to the tools available for general programming languages/platforms. The tools are lacking in contextual help, in storage of the code, in debugging, in refactoring, etc.

The languages for writing stored procedures tend to be very limited compared to general programming languages/platforms. They tend to be procedural, lack many operations, lack most common APIs, and lack many syntax advances (classes, scope, etc.). This has changed somewhat with the introduction of Java into Oracle and .NET into SQL Server.

So, as a general rule, avoid writing stored procedures; writing your code in a general programming environment is more desirable. Use stored procedures when you need their particular advantages, which mainly means high-performance and/or tightly-isolated data processing. A typical system will then have maybe a stored procedure or two, but definitely not dozens to hundreds.

Best wishes.

EDIT: Clarification...

Please note that I am addressing enterprise-class development **in-the-large**. If you have a tiny application and a few toy stored procedures, then you can probably ignore everyone's advice. I am assuming that the question is being asked for non-trivial scenarios.

I have dealt with every significant RDBMS over a period of nearly twenty years. I have dealt with databases upto 138 TBs, and individual tables of 8 TBs. I have worked with systems exceeding one thousand SPs. I have converted such databases across major versions and across major vendors. I am an architect, DBA, and just a programmer. If you want the benefit of such experience, then here it is. If not, fair enough.

EDIT: Expounding...

Nearly everything done in a stored procedure can be done by issuing comparable SQL statements from an application, particularly including anonymous procedure blocks (the guts of an SP without the name and permanence). Doing it well can avoid the problems and limitations of stored procedures while still retaining most of the benefits.

However, don't forget that [bad code can be written in any language](http://c2.com/cgi/wiki?BadCodeCanBeWrittenInAnyLanguage), so it is just as possible to write bad SP code as it is to write bad application code. Indeed, based on history and reports of observations in the wild, it seems even more likely to write bad SP code.

EDIT: @Chris Lively: regarding putting database code where the DBA can apply his tools...

Crippling your application development by using the DBA's limited tools is not an advantage or a step forward, nor is it even necessary.

Besides that, having been a senior DBA/architect for about twenty years, I am not generally impressed with what most DBAs do with database code in the applications that they support. I have mentored a lot of DBAs and programmers regarding database code, so please let me describe what I encourage them to do.

Every DBA should know how to make the database engine show them every SQL statement that is executed, regardless of source (inside or outside the engine), and they should know how to analyze that SQL's performance characteristics. I recommend that every programmer learn to do the same. If you can do this, then it no longer matters where the SQL originated, so Chris' recommendation to put the SQL in a SP is null and void.

If the performance of your system matters, such as when several million customers depend on it every day, then you should be checking the performance of every piece of SQL before it gets deployed to production. I recommend doing so as part of the automated tests that can be run as a part of the automated build for the system.

For example, it is very easy to configure an Ant build script to issue each piece of SQL to the database engine for an execution plan analysis. I like to save each execution plan to a text file and commit it to source control, where I can readily see a history of changes. I also make the build script check the execution plan against some simple criteria to ensure that SQL changes have not altered or compromised the performance.

Likewise, I check all my SQL into source control, and I make it easily available both to my application (for execution) and to my build script (for verification). At a minimum, my build script for the database can recreate the entire structure from scratch, and I often make it capable of loading or transferring data as well.

Obviously, I can handle stored procedures, but they are just one tool among many. It is a mistake (an antipattern) to treat SPs as a [Golden Hammer](http://c2.com/cgi/wiki?GoldenHammer).

On the other hand, when the performance really matters, a stored procedure can often be the best and even the only option. For example, when I redesigned a database recently for a major telecommunications provider, a stored procedure was an essential part of the strategy. I was loading forty thousand data files per day, totaling forty million rows, into a single database table (8 TB) that was growing past two billion rows of current data. A public-facing web site accessed that data via a web service, which required pulling a handful of rows from those two billion within just a few seconds. This was done using Oracle 10g, a custom C application, external tables, some bulk data loading, and a stored procedure. However, most of the database code was still in the C application and the stored procedure handled just one specific, performance-intensive piece.

Finally, please allow me to add that I think that everyone needs to get over the idea that programmers and DBAs have to be separated, or that they do radically different jobs, or that one is inherently more difficult or superior than the other. A database, even a big RDBMS, is just another bit of functionality captured in a component with a published API. It is not rocket science, and it does not have to take long for anyone to learn and master, especially at a minimal level necessary to support most applications. I see no reason why most programmers cannot become decent DBAs, or vice-versa. Perhaps not everyone is willing or able to aspire to become an expert in either general programming or databases, but it certainly seems feasible for most everyone to be able to support themselves in both areas. That is especially true with recently-available tools like Python and SQLite.

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

Q) In which scenario we can use in stored procedure?

A) Cursors are a mechanism to explicitly enumerate through the rows of a result set, rather than retrieving it as such.

However, while they may be more comfortable to use for programmers accustomed to writing While Not RS.EOF Do ..., they are typically a thing to be avoided within SQL Server stored procedures if at all possible -- if you can write a query without the use of cursors, you give the optimizer a much better chance to find a fast way to implement it.

In all honesty, I've never found a realistic use case for a cursor that couldn't be avoided, with the exception of a few administrative tasks such as looping over all indexes in the catalog and rebuilding them. I suppose they might have some uses in report generation or mail merges, but it's probably more efficient to do the cursor-like work in an application that talks to the database, letting the database engine do what it does best -- set manipulation.

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

Q) Why is it considered bad practice to use cursors in SQL Server stored procedure?

A) Because cursors take up memory and create locks.

What you are really doing is attempting to force set-based technology into non-set based functionality. And, in all fairness, I should point out that cursors *do* have a use, but they are frowned upon because many folks who are not used to using set-based solutions use cursors instead of figuring out the set-based solution.

But, when you open a cursor, you are basically loading those rows into memory and locking them, creating potential blocks. Then, as you cycle through the cursor, you are making changes to other tables and still keeping all of the memory and locks of the cursor open.

All of which has the potential to cause performance issues for other users.

So, as a general rule, cursors are frowned upon. Especially if that's the first solution arrived at in solving a problem.

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

Q) Why Database Design & Impl is required?

A) We are doing this to serve a need within the organization. By enlarge that need is to provide information that organization needs to do their job. To a gooder degree we can do the following, the better job that we can do, which gives potential for the organization which are being more successful. At next level mgmt. is going to take information and go for the decisions. As IT folks it is our job to get to head this information, then it is up to the mgmt. folks to deal with it properly.

*Data => Info*

*More flexible the better*

*Timely*

*Accurate data*

