SQL basics, Normalization & how to use clustered and non-clustered keys

Basics

A database is one or more large structured sets of persistent data, usually associated with software to update and query the data (Database Management System – DBMS). A Relational Database has its data organized and linked (related) to each other.

SQL is declarative language, instead of telling it what to do you simply tell it what you want.

An SQL or Database Object can be a database, a table, a column, a stored procedure etc.

Databases and SQL support multiple data types for storing numbers, text, guids and so on. You may ask "Why have different data types? Why not just treat everything as text?" The main reason is efficiency. The amount of storage space and speed of access improve when the DataBase knows what sort of data it's dealing with.

Character Data Types
1) Fixed Length (char, nchar)
2) Variable Length (varchar, nvarchar)
3) Text (unbounded alphanumeric data)

char(5) → DBMS allocates enough memory for 5 characters
varchar(5) → DBMS doesn't allocate memory, it just tells DBMS that it may need up to 5 characters

Inserting & Updating fixed character data is quicker.
nchar, nvarchar → Support for Unicode character set. Used for internationalization.

Use the decimal type for financial or other calculations.
decimal(38,12) → allocates space for 38 digits, with space reserved for 12 digits after the decimal point.
decimal(3, 2) → 2.45 (3 digits in total, 2 after decimal point)

Example of Inserting Data
INSERT INTO Category(categoryId, Category)
VALUES (1, "Thriller")

Character and Date data must be wrapped up inside single quotes (Sql Server). Delimiters are not necessary around numerical data.

DELETE FROM MEMBERS WHERE MEMBERID = 3
DELETE FROM MEMBERS (Deletes all members NOT the table)

Retrieving all the unique values in a record
SELECT DISTINCT City FROM MemberAddresses
SELECT DISTINCT City, Region FROM MemberAddresses

DISTINCT works on ***ALL columns*** in combination

SELECT FirstName + ' ' + LastName AS [FullName]
FROM MemberDetails
WHERE City = 'London'

SELECT FilmName FROM Films
WHERE (YearReleased NOT BETWEEN 2000 AND 2010)

AND
(FilmName BETWEEN 'A' AND 'D')

LIKE operator
% matches zero or more characters
_ matches one character

SELECT * FROM Members WHERE Name LIKE '%Smith'

will return "John Smith", "Mary Smith" and so on.

NULL = Unknown!
SELECT * FROM MEMBERS WHERE RegistrationDate > '2012-01-01'

It will not return records that have a RegistrationDate of NULL, the comparison
"Unknown" > '2012-01-01' does not make sense to the DBMS.

ORDER BY considers NULLs as equals and groups them together. Use "IS NULL" or "IS NOT NULL" to check for NULL. The "=" operator is not used again because semantically "something = Unknown" does not make sense.

Constraints

When creating database tables and fields you can specify constraints that limit what data can go in a field.

Constraints → NOT NULL, UNIQUE, CHECK, PRIMARY KEY, FOREIGN KEY

PRIMARY KEY Constraint → A combination of the UNIQUE and NOT NULL constraints.
An example

```
CREATE TABLE NamesAndAges
(
    Name varchar(50),
    Age int NOT NULL CHECK (Age >= 0)
)
```

Foreign Keys are columns that refer to primary keys in another table. There's nothing stopping someone from entering a value in a foreign key table that doesn't have a matching value in the primary key table. That's why the foreign key constraint is provided.

ALTER TABLE Members ADD CONSTRAINT locations_fk FOREIGN KEY(locationId) REFERENCES Locations(locationId)

Sql Functions

Sql Common Math Functions → ABS, POWER, SQRT, RAND, ROUND, CEILING, FLOOR
String Functions → SUBSTRING, UPPER, LOWER, REVERSE, TRIM, LENGTH/LEN
Date Functions → DAY, MONTH, YEAR

If NULL appears in any of your Sql Math, then the result is always NULL. ***NULL represents the unknown so any operation with the unknown such as x + "Unknown" can only result in "Unknown", so x + NULL will always be NULL***. Concatenating text values with NULL will also produce NULL. You can use the COALESCE function to handle that.

SELECT City, COALESCE(City, 'Not Known') FROM Members

If City is NULL then 'Not Known' will be returned.

Using Insert with Select

INSERT INTO OtherMembers(MemberId, Name)
SELECT MemberId, Name
FROM Members
WHERE Name LIKE 'Peter%'

Members Table

Name	Location
John	Dublin
Mary	London
Peter	Dublin
Abbie	London

SELECT Location FROM Members GROUP BY Location

"groups all identical locations into one row"

Result

Dublin
London

Most DBMSs don't allow columns that appear in the results to differ from the columns in GROUP BY clause. The DBMS wouldn't know which value to include for a particular group.

COUNT() → counts all records excluding NULLs
COUNT(*) → counts all

SELECT Location, COUNT(Name) FROM Members
NOT allowed since Location can return more than one row while COUNT(Name) will return one.

SELECT Location, COUNT(Name) FROM Members
GROUP BY Location

Result

Dublin	2
London	2

Useful functions to use with group by → SUM, AVG, MIN, MAX

HAVING clause enables you to specify conditions that filter which group results appear in final results.

SELECT Location FROM Members
GROUP BY Location
HAVING COUNT(Name) >= 2

Joins
Using tables Customers & Orders

```
create table Customers
(
        CustomerId int not null,
         CustomerName varchar(200) not null,
        DateOfBirth datetime,
        constraint customers_pk primary key(CustomerId)
);

create table Orders
(
        OrderId int not null,
        CustomerId int not null,
        Price decimal(10,2) not null,
        Constraint customers_fk foreign key(customerId)
        References customers(customerId)
)
```

Customers

1	Peter	16/05/1982
2	Abbie	12/01/1983
3	Mark	21/02/1978

Orders

1	1	100.00
2	4	24.00
3	1	60.00

An INNER JOIN requires each result in one table to match a result in the other table (ON clause must evaluate to true)
Operators other than "=" are allowed for ON clause.

```
SELECT Customers.CustomerName, Orders.Price
FROM Customers INNER JOIN Orders ON
Customers.CustomerId = Orders.CustomerId
```

Result

Peter	100.00
Mark	24.00
Peter	60.00

```
SELECT Customers.CustomerName, SUM(Orders.Price) as Total FROM Customers
INNER JOIN Orders ON Customers.CustomerId = Orders.CustomerId
GROUP BY Customers.CustomerName
```

Peter	160.00
Mark	24.00

ON statement can contain more than one condition e.g.
ON Table1.Column1 = Table2.Column1 AND Table1.Column2 = Table2.Column2

A CROSS JOIN returns all rows from all tables

SELECT CustomerName, Price
FROM Customers, Orders

This will return 3 x 3 = 9 records

SELF JOIN → When a table is joined to itself, useful for finding pairs of records in the same table.

Which customers have the same date of birth?

SELECT Customers.CustomerName, Customers.DateOfBirth
FROM Customers
INNER JOIN Customers as Cust
ON Customers.DateOfBirth = Cust.DateOfBirth AND
Customers. CustomerId != Cust.CustomerId

LEFT OUTER JOIN → All records from table named on the left are returned. If there is no match on the table on the right NULL is returned.

SELECT Customers.CustomerName, Orders.Price
FROM Customers LEFT OUTER JOIN Orders
ON Customers.CustomerId = Orders.CustomerId
ORDER BY Price

Abbie	NULL
Mark	24.00
Dimos	60.00
Dimos	100.00

RIGHT OUTER JOIN → Simply the reverse of a left outer join

SubQueries

A subquery example
SELECT FirstName, LastName, YEAR(DateOfBirth)
FROM MemberDetails
WHERE YEAR(DateOfBirth) IN
(SELECT YearManufactured FROM Cars)

Useful operators for subqueries → ANY, SOME, ALL, EXISTS

Views
A View is a "virtual" table, a saved query. Views can be used to provide security and simplify data extraction. However if you use views built on views you may start having performance problems.

Transactions
A Transaction is a group of SQL statements taken together as a logical unit of work. All statements must execute successfully. If one fails then all operations must be undone.

Transact SQL or T-SQL is an extension to SQL syntax that some DBMSs use to allow a finer degree of control over processing transactions.

A transaction must pass the "ACID" test
1) Atomic. All statements are executed or none
2) Consistent. The database must be in a consistent state at the end of the transaction
3) Isolated. Data that transactions change must not be visible during the changes being applied.

4) Durable. At then end of the transaction, the database must save the data correctly.

DBMSs support transactions by using transaction logs and locks.
1) The Transaction Log is a separate file to which a record is written to include data before a modification and data after a modification. The data is made permanent on the database files after the transaction ends. If the transaction fails data in the transaction log will be used to undo any changes and restore unmodified state. If you need to roll back changes, everything required to perform this function is available in the transaction log. Every change to data (insert, update, delete) is logged to the transaction log.
2) Locks make a data structure unavailable to other valid users of that data for some period of time. In general, you don't apply locks yourself, DBMS applies the locks although there is a LOCK TABLE syntax.

Locking granularity → Database, Table, Page, Row

Locking Levels
1) Shared – no one can modify data but can read it
2) Exclusive – no one can see data because data is about to change

Deadlocks are possible with DBMSs when multiple transactions are being executed concurrently. If you ever face this problem your transaction may be rolled back, consider resubmitting it.

SQL Security

SQL security is centered around the following concepts
1) User IDs → Each user is assigned a user Id. A user can be a person or a program. The user may need to provide a password to connect to DBMS or DBMS can accept his Operating System password.
2) Objects → Initially these were just tables and views. Modern DBMSs have extended this to include various other objects like stored procedures etc.
3) Privileges → Privileges allow users to manipulate objects. Basic privileges are SELET, UPDATE, DELETE, INSERT. Again modern DBMSs have added more privileges for various objects.

Each action a DBMS performs is performed on behalf of a User ID. If the User ID has the appropriate privileges for the object he wants to manipulate, the action is performed.

Groups/Roles → User IDs can belong to a Role and the Role is assigned certain privileges.

Privileges are assigned or unassigned using GRANT/REVOKE statements but most DBMSs provide GUIs for performing these actions.

Many "physical" users of a database can be assigned the same User ID. At least one user is created by DBMS during installation (SA – System Admin) to get things rolling, creating the first tables and other user IDs.

SQL Batches
Simply a group of statements executed at the same time. If there is an error in any statement in the batch the entire batch is rejected.

Batches are separated by the keyword GO.

```
select * from authors   }
select * from titles    }   1st Batch

GO

select * from books     }   2nd Batch
```

Sql Server Comments

-- this line is a comment
/* these 2 lines
are a comment */

Sql Server supports global and local variables. You can only use local variables that are confined to the batch, stored procedure or trigger they are created in. Global variables are used by Sql Server itself.

Examples of declaration and use of local variables
declare @myvar int -- null
declare @myvar = 42 -- 42

declare @sales decimal(10,2)
select @sales = price * quantity from titles where titleId = 123

select @sales -- 298.45 let's say!

Global variables are preceded by @@

@@error	@@version
@@identity	@@max_connections
@@rowcount	@@servername

T-SQL Basic Statements & Examples

If/Else

```
if @myvar < 15
        update titles set title = "xx" where titleId = @myvar
else
        update titles set title = "aa" where titleId = @myvar
```

Begin/End (Equivalent to a block of code in {})

```
if @myvar < 15
    begin
        update titles set title = "xx" where titleId = @myvar
        update titles set title = "aa" where titleId = @myvar
     end
```

if [not] exists

```
declare @name varchar(30)
select @name = 'Smith'
if exists (select * from authors where lastName = @name)
    update authors set firstName = 'john' where lastName = @name
```

Return
Return is used to exit a batch, trigger or stored procedure unconditionally, any statements following return are not executed.

While

```
while (select avg(price) from titles < 40)
```

```
    begin
        update titles set price = price – 0.1
        update titles set lastModified = GETDATE()
    end
```

break & continue functionality is also available

goto

It will branch to a user defined label

```
if @@error != 0
    begin
        select @errno = 300
        goto error
    end

….

error:
    … more sql …
```

The best way to avoid deadlocks is to write transactions in the same order. Avoid the following:

begin tran	begin tran
update table A	update table B
update table B	update table A
commit tran	commit tran

Transaction Control Statements
begin tran, rollback tran, commit tran.
You can also name a transaction if you wish to.

Savepoints are intermediate points in a transaction and they allow you to only rollback a portion of the work you have done. You can monitor a transaction through 2 global variables
@@error → detects errors during or after statements execute
@@transtate → monitors current state of the transaction

Some statement errors such as trying to insert a NULL in a column that does not allow nulls are not fatal and will not cause a rollback. To ensure all errors are caught use the following:

```
begin tran
    insert into …
    update …
    if @@error != 0
        begin
            rollback tran
            return
        end
commit tran
```

OR

```
begin tran UpdateTran
    begin try
        update …
```

```
        update …
    end try
    begin catch
        rollback tran UpdateTran
        return
    end catch
commit tran UpdateTran
```

Stored Procedures

A Stored Procedure is a collection of SQL statements stored under a name and executed.
Stored procedures have 2 performance benefits over directly passing Sql to Server.
1) The steps that the Server does when executing SQL like parsing, checking syntax etc. are done when you create the procedure
2) Network traffic is reduced since you are only sending the procedure name and parameters

With Stored Procedures it is also easier to defend against SQL injection attacks.

The advent of Linq to Sql, Entity Framework and other ORMs, faster computers and improved Sql Server caching make some of the benefits of stored procedures questionable compared to the simplicity of Linq & ORMs. As usual in these situations there is no right or wrong way, a decision needs to be made based on the needs, requirements and context of the specific application you are trying to build.

Every stored procedure that has a name starting with “sp_” is treated as a system procedure and can be executed with a DB different than the one created in.

To execute a stored proc

```
    execute proc_name
    OR
    exec proc_name
```

```
create procedure myproc
@name varchar(40),
@age int = 50
as
select * from authors where lastName = @name
return

exec myproc ‘Smith’
OR
exec myproc @name = ‘Smith’
```

@age has a default value so it does not need to be passed a value.

To return values from a stored procedure.

@parameter datatype output

```
create procedure myproc
@bookId char(6),
@totalSales int output
as
select @totalSales = sum(quantity) from salesdetail
where titleId = @bookId
```

You can also nest procedures if you need to.

```
create procedure myproc
    exec proc1
    exec proc2
```

Triggers are a special type of stored procedure that are executed when a user issues an insert, update or delete on a table “firing the trigger”. Triggers are useful for enforcing referential integrity, custom business rules and admin functions.

Referential Integrity → The process whereby relationships are maintained within the data.
Cascading → The process whereby changes to a parent are propagated to the child.

Normalization

Normalization consists of a series of guidelines for creating a proper database structure. Normal Forms apply to tables.

1st Normal Form
Place related data items in a table, ensure there are no repeating groups of data, ensure there is a primary key.

2nd Normal Form
There must be no partial dependencies of any of the columns on the primary key.

Example table not in 2nd NF (which authors have written which books)

BookId
BookName
AuthorId
AuthorName

BookId and AuthorId could be the primary key (to cover cases where a book has multiple authors) but AuthorName only depends on AuthorId.

To bring this to 2nd Normal Form we need to use 3 tables.

Books

BookId
BookName

Authors

AuthorId
AuthorName

BookAuthors

BookId
AuthorId

2nd Normal Form is usually enough for Database design. You may want to avoid the 3rd normal form to avoid multiple joins.

3rd Normal Form
The table is in 2nd Normal Form and all non-primary fields are dependent on the primary key.

Table in 2nd Normal Form

Members

MemberId
Name
CountryCode
Country

You can think of Country as not really dependent on the MemberId, it depends on the CountryCode value so it may be better off being in a separate table.

Tables in 3rd Normal Form

Members

MemberId
Name

Countries

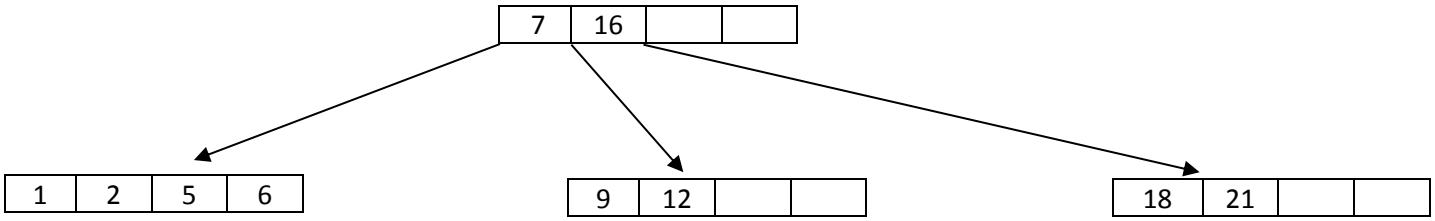
CountryCode
Country

Database Internals & Indexes

B-Tree

A tree data structure that keeps data sorted and allows fast searches, insertions and deletions. It is optimized for systems that read/write large blocks of data and is commonly used in databases and file systems.

Example B-Tree



Each internal node's elements act as separation values which divide its subtrees. If an internal node has 3 child nodes then it must have 2 separation values.

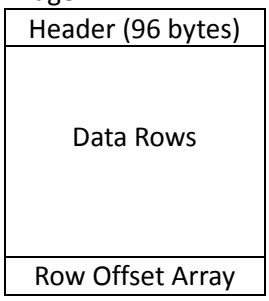
A search is performed starting at the root and choosing the child pointer whose separation values are on either side of the value that is being searched. Binary search is typically used within nodes to find separation values and child tree of interest.

Data Page Format

The fundamental unit of data storage in SQL Server is the page. The disk space allocated to a data file (.mdf or .ndf) in a database is logically divided into pages numbered continuously from 0 to n. Disk I/O operations are performed at the page level. That is, SQL Server reads or writes whole data pages.

All data is SQL Server is stored in pages including data, indexes and various other page types to support the operations of SQL Server. Each page is 8kb and contains various kinds of data depending on page type.

Page



Each page begins with a 96-byte header which stores system information about the page. This information includes the page number, page type, the amount of free space on the page, and the allocation unit ID of the object that owns the page.

The row offset array is an integer array pointing out the locations in the page of the individual rows.

The first entry in the rows offset array will be 96 since this is the first byte after header that occupies bytes 0 to 95. If for example you have a table with an Integer column, the second row of the table will be 117. 11 bytes are used in total, 4 bytes to store the actual value and 7 bytes which hold various data that Sql Server needs.

Assume that you have a table with EmployeeId, Age and various other information and you have not created any indexes. Let's say that each row is around 2kb so a data page can hold 4 rows and you added 12 rows

Page 1

ID	Age
4	25
12	22
10	30
1	50

Page 2

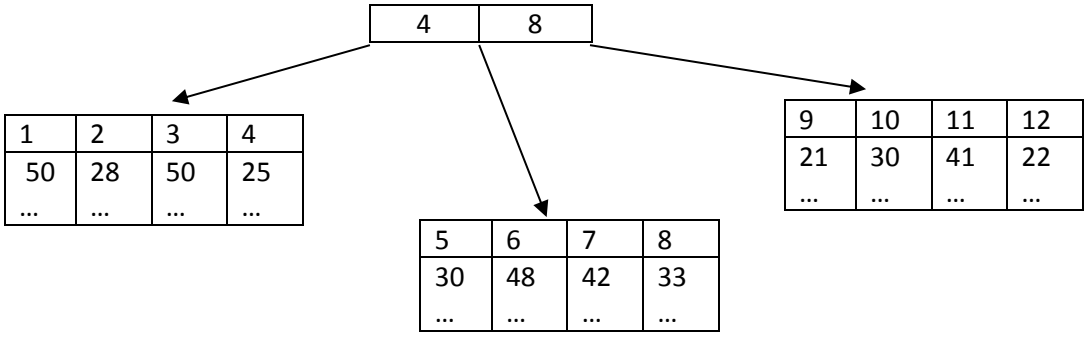
ID	Age
2	28
3	50
5	30
6	48

Page 3

ID	Age
9	21
8	33
11	41
7	42

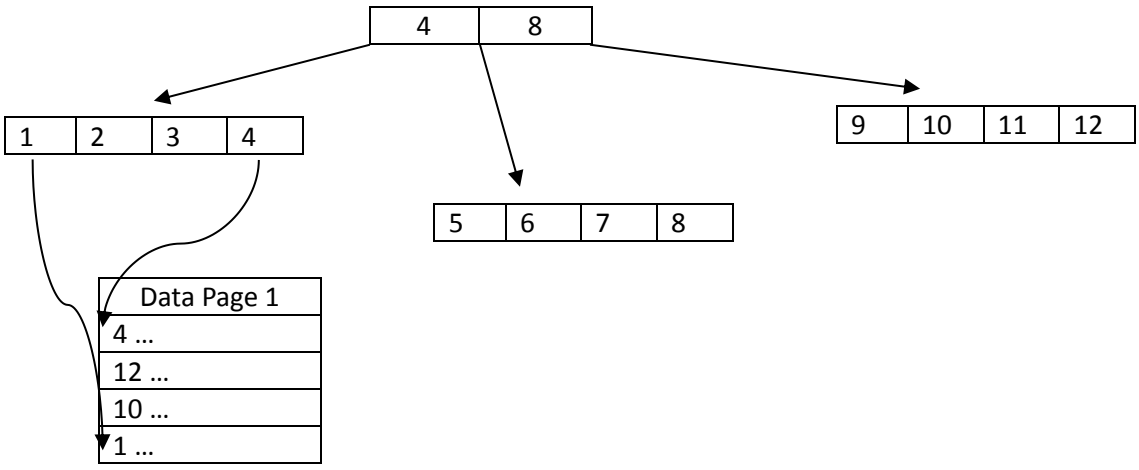
In this case there is no ordering of data pages or rows within the pages. If you add a clustered key on employeeId however the data will be sorted by employeeId. This means that you can have only 1 clustered key per table, since the physical order of the data depends on it.

Indexes are stored in Index pages in B-Trees. A possible B-Tree could be the following:



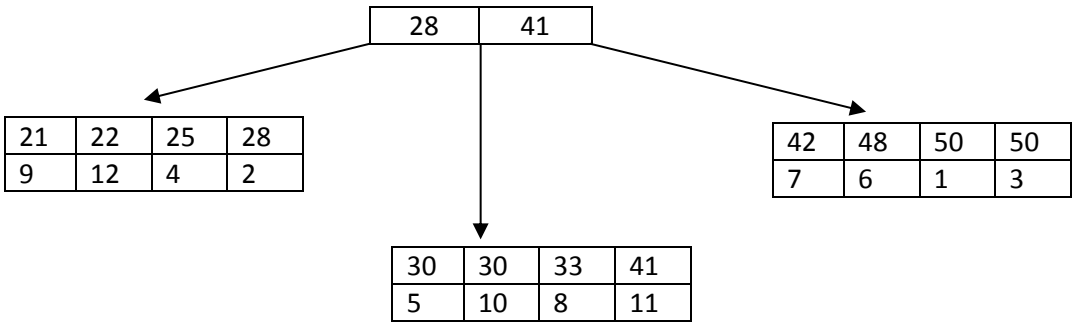
The Leaf Nodes of the B-Tree include the actual data rows. Clustered keys are great for range searches because of the ordering provided and for searching a particular row using a unique index like employeeId in this case. You should avoid long composite clustered keys because they are being used by non-clustered keys and take too much space. Also avoid them on columns that are often modified because of the performance penalty for maintaining order. Clustered indexes should be used for queries that return large result sets and with columns that contain a large number of distinct values.

If you were about to have a non-clustered key on employeeId the following B-Tree could be created.



With a non-clustered key the actual data is not ordered in any way, the leaf nodes of the B-Tree include a pointer to the location on the data page.

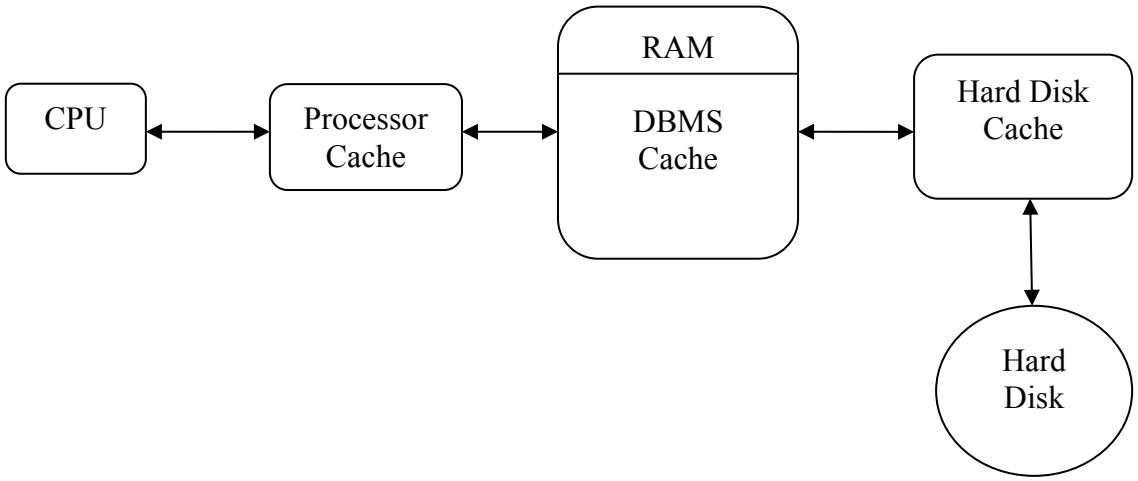
If you create a non-clustered key on age and you have a clustered key for employeedId.



The leaf nodes include the clustered key value which is being used to find the row. As you can see, in all cases the Index nodes are sorted by the Key. Non-clustered keys should be used on columns that contain a large number of distinct values, queries that do not return large result sets (probably because of all the jumping to different data pages)

Primary Keys are by default added as clustered unique indexes.

Database Design & Indexing Tips.
The most important consideration is ensuring that you have the right hardware. DBMSs are designed to use multiprocessor systems effectively so having multiple CPUs will help. Having 4 GB or RAM or even more if possible is also necessary as SQL Server uses RAM quite aggressively.



A cache is a location where data is stored for fast access after it is retrieved from a location that is significantly slower to access. Database Indexes and frequently accessed data would be stored on DBMS cache on RAM if enough space is available.

In general, you should use Indexes on any column used is a table join or in any where clause. The join requirement is the most important. Also, you should avoid using indexes on columns that may be frequently updated or deleted.

A practical example of using Indexes.

You have a table that holds “Points of Interest”, various venues/attractions around the world that may be of interest to tourists. Each Point of Interest has coordinates (latitude & longitude). All latitude & longitude values will be unique with very few exceptions. Adding a clustered key on both latitude and longitude will not add any value for longitude, the rows will be sorted by latitude.

A range check (latitude > x and latitude < y) can be faster with a clustered key. However (longitude > x and longitude < y) will not really benefit from a composite clustered key. It will be better to use a clustered key for longitude and then use a non-clustered key for longitude. This way the data rows will be sorted by latitude which will allow you to perform the first range check fast. The longitude index will be using longitudes as the key so Index pages will be sorted by longitude. The second range check can be performed faster by using this Index.

Another Example
A table with addresses that contains Region and City columns (as part of address). In this case you will have many rows with the same state and quite a few with the same city. It will be beneficial to have a clustered index on region, city so that your data is sorted like

Scotland, Edinburgh
Scotland, Edinburgh
Scotland, Glasgow

However a non-clustered index can also be beneficial for queries like
"where region = 'Scotland' and city = 'Edinburgh' "

Adding those indexes is obviously something that you would want to do if you know that columns region and city will be heavily used in searches.

Selecting appropriate primary keys

The possible options that you have for primary keys are the following:
- Composite (consisting of more than one column)
- Surrogate (a key produced by the database, it can be numeric and auto generated or you can use a GUID)
- Natural (Key already exists in the real world)

A Natural Key is a Key that already exists, it is a data attribute that uniquely identifies an object and has some sort of meaning in the real world such as an Employee Number or a Social Security Number. If a Key like that already exists then you should use it in your tables instead of introducing an "unnatural" key. Remember the principle of DDD (Domain Driven Development) of always trying to use the appropriate business terminology in your code when present. By using a social security number you can have meaningful searches by this number something that you cannot have with a surrogate key. On the other hand, natural keys that are too closely tied to business requirements may need to change in the future as the business changes. Always plan for future growth and future changes when making a decision.

A Surrogate Key is a key produced by the DBMS and it is the best option for large tables that do not have a Natural Key.
For cross reference tables that contain IDs or other keys from 2 or more other tables you should use composite keys. For example remember our example with Books and Authors

We have a table called BookAuthors that helps us determine which authors have written which books. This table has 2 columns, BookId & AuthorId which should together from the primary key.

Numeric keys can be a bit more efficient however in small tables that store "static" data which are not likely to change like a table of countries and codes, consider using character keys.

Code	Country
AT	Austria
DE	Germany
UK	United Kingdom

In this table there's nothing wrong with having the Code as the primary key.