- Performance tuning is not a silver bullet. Simply put, good system performance depends on: good design, good implementation, defined performance objectives, and performance tuning.
- Performance tuning is ongoing process. Implement mechanisms that provide performance metrics which you can compare against your performance objectives, allowing you to schedule a tuning phase before your system fails.
- The object is to meet your performance objectives, not eliminate all bottlenecks. Resources within a system are finite. By definition, at least one resource (CPU, memory, or I/O) will be a bottleneck in the system. Tuning allows you minimize the impact of bottlenecks on your performance objectives.
- Design your applications with performance in mind:
  - Keep things simple - avoid inappropriate use of published patterns.
  - Apply Java EE performance patterns.
  - Optimize your Java code.

Performance tuning is the improvement of system performance. Typically, in computer systems, the motivation for such activity is called a performance problem, which can be either real or anticipated. Most systems will respond to increased load with some degree of decreasing performance. A system's ability to accept higher load is called scalability, and modifying a system to handle a higher load is synonymous to performance tuning.

Systematic tuning follows these steps:

1. Assess the problem and establish numeric values that categorize acceptable behavior.
2. Measure the performance of the system before modification.
3. Identify the part of the system that is critical for improving the performance. This is called the bottleneck.
4. Modify that part of the system to remove the bottleneck.
5. Measure the performance of the system after modification.
6. If the modification makes the performance better, adopt it. If the modification makes the performance worse, put it back the way it was.

8.1 Client Performance

- Performance of client/server can be improved in many ways. This section of client performance mainly focuses on the attributes that we can examine in order to improve the performance of client machine.
- They can be maintainability, dependability, efficient, usability.
- It mainly includes two types of performance. They are :
  *Hardware performance
  *Software performance

8.1.1 Hardware and Software.

Hardware Performance

✓ The performance of client is to certain extent dictated by a particular hardware within the client. Client performance can be improved by improving any of the subsystems.

✓ **Note**:: When purchasing a client machine the best way is to purchase the fastest, most reliable, accurate, machine available .And it also should have the properties of safety and security.

Software Performance : the software of the client workstation can be broken down into two performance reasons:-

✓ Operating system

- The capability to be simultaneously involved in multiple process is an essential for client/server system.
- Independent tasks can be activated to manage communication processes.
- Multiple personal productivity application such as word processor, spreadsheets and presentation graphics can be active.
- Most multitasking operating system today are thirty two bits

✓ Application
- The client application is normally used where largest improvements can be made.
- Performance of client level is very difficult to judge because each user perception of response is different.
- The best way of determining problem areas is to ask users what areas of application they consider now.

8.2 Server Performance

This section focusses on performance gains and improvements you can make at the server

8.2.1 Hardware and Software.

Hardware Performance

✓ Upgrading server hardware just like upgrading client hardware can improve the performance of the client/server
✓ Using multiple network interface cards within a server can also improve performance by moving the network loads
✓ *Within file server and PC-based database server, high- performance file system using technology such as SCSI(* **Small Computer System Interface)** *and RAID(***redundant array of independent disks***) offer dramatic performance improvements over older ISA and EISA driver technology*

Software Performance

✓ Data base and communication processing should be offloaded to a server processor
✓ Several servers can be used together so that performance of the individual components can be improved

8.2 Database Performance

You can fine-tune your application to gain maximum performance by speeding up forms and queries and increasing data throughput. The goal of database performance tuning is to minimize the response time of your queries and to make the best use of your system's resources by minimizing network traffic, disk I/O, and CPU time. This goal can only be achieved by *understanding the logical and physical structure of your data, understanding the applications used on your system, and understanding how the many conflicting* uses of your database may impact database performance.

For Optimum Performance We Need:

✓ Efficient index design
✓ Efficient Query design
✓ Efficient database design

8.3.1 Index design

An index is a pointer to the data in a table.

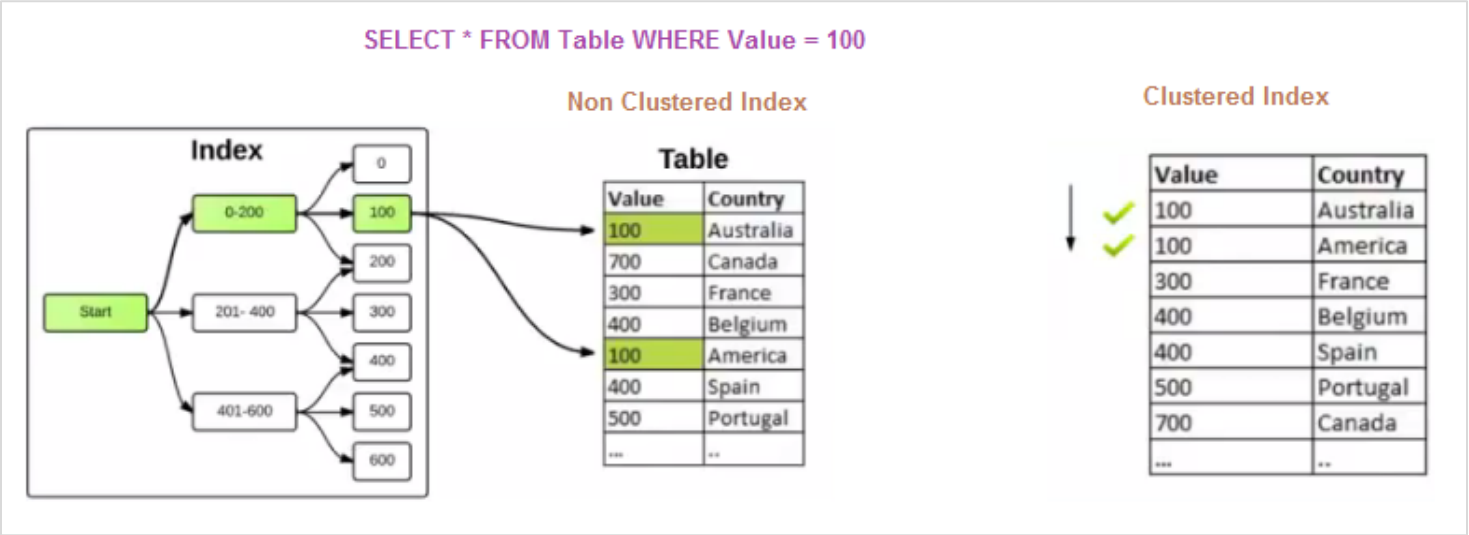
Example : index in the back of a book : book is printed in page order, index represents the book in keywords or subject order
Books index makes finding data faster, more efficient : adding or changing book data means the index must be updated.

Types of index

- **Single index**
- **Composite index** : more than one column is used for index
- **Unique** : does not allow duplicate values

Methods to access data in relational database

- Full scan : searches all rows in table
- Index access : searches from indexes



✓ An index for a table is a data Organization that enables certain queries to access one or more records of that table fast.

- ✓ Proper tuning of index design is essential to high performance of the database.
- ✓ Index can be created by using one or more columns of a database table.

Database software would literally have to look at every single row in the Employee table to see if the Employee_Name for that row is 'Abc'. And, because we want every row with the name 'Abc' inside it, we can not just stop looking once we find just one row with the name 'Abc', because there could be other rows with the name **Abc**. So, every row up until the last row must be searched – which means thousands of rows in this scenario will have to be examined by the database to find the rows with the name 'Abc'. This is what is called a **full table scan**

How does a database know when to use an index? When a query like "SELECT * FROM Employee WHERE Employee_Name = 'Abc' " is run, the database will check to see if there is an index on the column(s) being queried. Assuming the Employee_Name column does have an index created on it, the database will have to decide whether it actually makes sense to use the index to find the values being searched – because there are some scenarios where it is actually less efficient to use the database index, and more efficient just to scan the entire table.

The whole point of having an index is to speed up search queries by essentially cutting down the number of records/rows in a table that need to be examined. An index is a data structure (most commonly a B- tree) that stores the values for a specific column in a table.

Designing an appropriate set of indexes can be one of the more troubling aspects of developing efficient relational database applications. Perhaps, the most important thing you can do to assure optimal application performance when accessing data in a relational/SQL database is to create the correct indexes for your tables based on the queries your applications use. Of course, this is easier said than done.

But we can start with some basics. For example, consider this SQL statement:

```
SELECT LASTNAME, SALARY FROM EMP WHERE EMPNO = '000010' AND DEPTNO = 'D01';
```

What index or indexes would make sense for this simple query? First, think about all the possible indexes that you could create. Your first short list probably looks something like this:

- Index1 on EMPNO
- Index2 on DEPTNO
- Index3 on EMPNO and DEPTNO

This is a good start, and Index3 is probably the best of the lot. It enables the DBMS to use the index to immediately look up the row or rows that satisfy the two simple predicates in the WHERE clause. Of course, if you already have a lot of indexes on the EMP table, you might want to examine the impact of creating yet another index on the table.

With the caveat that appropriate index creation can be complicated, let's look at the steps you you can take to build the right indexes on your tables.

TOP 10 STEPS TO BUILDING USEFUL DATABASE INDEXES

1. INDEX BY WORKLOAD, NOT BY TABLE

Many people make the mistake of just guessing at some indexes to create when they are creating database tables. Without an idea of how the tables are going to be accessed, though, these guesses are often wrong – at least some of them.

Indexes should be built to optimize the access of your SQL queries. To properly create an optimal set of indexes requires a list of the SQL to be used, an estimate of the frequency that each SQL statement will be executed, and the importance of each query. Only then can the delicate balancing act of creating the right indexes to optimize the right queries most of the time be made.

If you are doing it any other way, you are doing it wrong.

2. BUILD INDEXES BASED ON PREDICATES

3. INDEX MOST-HEAVILY USED QUERIES

Numbers 2 and 3 can be thought of as corollaries to Number 1… that is, these are the aspects of application workload that need to be examined to produce appropriate and effective indexes.

4. INDEX IMPORTANT QUERIES

The more important the query, the more you might want to tune by index creation. If you are coding a query that the CIO will run every day, you want to make sure it delivers optimal performance. So building indexes for that particular query is important. On the other hand, a query for a clerk might not necessarily be weighted as high, so that query might have to make do with the indexes that already exist. Of course, the decision depends on the application's importance to the business-not just on the user's importance.

5. INDEX TO AVOID SORTING (GROUP BY, ORDER BY)

In addition to building indexes to optimize data access, indexes can be used to avoid sorting. The GROUP BY and ORDER BY clauses tend to invoke sorts, which can cause performance slowdowns. By indexing on the columns specified in these clauses the relational optimizer can use an index to avoid a sort, and thereby potentially improve performance.

6. CREATE INDEXES FOR UNIQUENESS (PK, U)

Some indexes are required in order to make the database schema valid. Most database systems require that unique indexes be created when unique contrainst and primary key constraints exist.

7. CREATE INDEXES FOR FOREIGN KEYS

Creating indexes for each foreign key can optimize the performance when accessing and enforcing referential constraints (RI – referential integrity). Most database systems do not require such indexes, but they can improve performance.

8. CONSIDER ADDING COLUMNS FOR INDEX ONLY ACCESS

Sometimes it can be advantageous to include additional columns in an index to increase the chances of index-only access. With index-only access all of the data needed to satisfy the query can be found in the index alone — without having to read data from the table space. For example, suppose that there is an index on the DEPTNO column of the DEPT table. The following query may use this index:

```
SELECT DEPTNAME FROM DEPT WHERE DEPTNO = 'D01';
```

The index could be used to access only those columns with a DEPTNO greater than D00, but then the DBMS would need to access the data in the table space to return the DEPTNAME. If you added DEPTNAME to the index, that is, create the index on (DEPTNO, DEPTNAME) then all of the data needed for this query exists in the index and additional I/O to the table space would not need be needed. This technique is sometimes referred to as *index overloading*.

Of course, this is not always a good idea. You have to take into account whether other queries use the index and how it might negatively impact their performance.

9. DON'T ARBITRARILY LIMIT NUMBER OF INDEXES

There should be no *arbitrary limit* on the number of indexes that you can create for any database table. Relational optimizers rely on indexes to build fast access paths to data. Without indexes data must be scanned – and that can be a long, inefficient means by which to retrieve your data.

Sometimes organizations develop database standards with rules that inhibit the number of indexes that can be created. When a standard such as this exists, it usually is stated as something like "Each table can have at most five indexes created for it" — or — "Do not create more than three indexes for any single table in the database." ***These are bad standards.*** If you already have three indexes, or five indexes, or even 32 indexes, and another index will improve performance why would you arbitrarily want to avoid creating that index?

Anyway, a good indexing standard, if you choose to have one, should read something like this: "Create indexes as necessary to support your database queries. Limitations on creating new indexes should only be entertained when they begin significantly to impede the efficiency of data modification."

Which brings us to…

10. BE AWARE OF DATA MODIFICATION IMPLICATIONS

The DBMS must automatically maintain every index you create. This means every INSERT and every DELETE to an indexed table will insert and delete not just from the table, but also from its indexes.

Additionally, when you UPDATE the value of a column that has been defined in an index, the DBMS must also update the index. So, indexes speed the process of retrieval but slow down modification.

8.3.2 Query design

- ✓ Describes how the correct design of the query used by an application can significantly improves the performance.
- ✓ Efficient SQL code is primarily about efficient queries using the SELECT command.
- ✓ The SELECT command allows use of a WHERE clause, reducing the amount of data read.
- ✓ The WHERE clause is used to return (or not return) specific records.
- ✓ The UPDATE and DELETE commands can also have a WHERE clause and, thus, they can also be performance-tuned with respect to WHERE clause use, reducing the amount of data accessed.

Improve Indexes

Creating useful indexes is one of the most important ways to achieve better query performance. Useful indexes help you find data with fewer disk I/O operations and less system resource usage.

To create useful indexes, you much understand how the data is used, the types of queries and the frequencies they are run, and how the query processor can use indexes to find your data quickly.

When you choose what indexes to create, examine your critical queries, the performance of which will affect user experience most. Create indexes to specifically aid these queries. After adding an index, rerun the query to see if performance is improved. If it is not, remove the index.

As with most performance optimization techniques, there are tradeoffs. For example, with more indexes, **SELECT** queries will potentially run faster. However, DML (**INSERT**, **UPDATE**, and **DELETE**) operations will slow down significantly because more indexes must be maintained with each operation. Therefore, if your queries are mostly **SELECT** statements, more indexes can be helpful. If your application performs many DML operations, you should be conservative with the number of indexes you create.

SQL Server Compact includes support for showplans, which help assess and optimize queries. SQL Server Compact uses the same showplan schema as SQL Server 2008 R2 except SQL Server Compact uses a subset of the operators. For more information, see the Microsoft Showplan Schema at http://schemas.microsoft.com/sqlserver/2004/07/showplan/.

The next few sections provide additional information about creating useful indexes.

Create Highly-Selective Indexes

Indexing on columns used in the WHERE clause of your critical queries frequently improves performance. However, this depends on how selective the index is. Selectivity is the ratio of qualifying rows to total rows. If the ratio is low, the index is highly selective. It can get rid of most of the rows and greatly reduce the size of the result set. It is therefore a useful index to create. By contrast, an index that is not selective is not as useful.

A unique index has the greatest selectivity. Only one row can match, which makes it most helpful for queries that intend to return exactly one row. For example, an index on a unique ID column will help you find a particular row quickly.

You can evaluate the selectivity of an index by running the sp_show_statistics stored procedures on SQL Server Compact tables. For example, if you are evaluating the selectivity of two columns, "Customer ID" and "Ship Via", you can run the following stored procedures:

```
sp_show_statistics_steps 'orders', 'customer id';
RANGE_HI_KEY RANGE_ROWS EQ_ROWS DISTINCT_RANGE_ROWS
-----------------------------------------------------------
ALFKI          0         7             0
ANATR          0         4             0
ANTON          0         13            0
AROUT          0         14            0
BERGS          0         23            0
BLAUS          0         8             0
BLONP          0         14            0
BOLID          0         7             0
BONAP          0         19            0
BOTTM          0         20            0
BSBEV          0         12            0
CACTU          0         6             0
CENTC          0         3             0
CHOPS          0         12            0
COMMI          0         5             0
CONSH          0         4             0
DRACD          0         9             0
DUMON          0         8             0
EASTC          0         13            0
ERNSH          0         33            0
(90 rows affected)
```

And

```
sp_show_statistics_steps 'orders', 'reference3';
RANGE_HI_KEY RANGE_ROWS EQ_ROWS DISTINCT_RANGE_ROWS
-----------------------------------------------------------
1            0          320             0
2            0          425             0
3            0          333             0
(3 rows affected)
```

The results show that the "Customer ID" column has a much lower degree of duplication. This means an index on it will be more selective than an index on the "Ship Via" column.

For more information about using these stored procedures, see sp_show_statistics (SQL Server Compact), sp_show_statistics_steps (SQL Server Compact), and sp_show_statistics_columns (SQL Server Compact).

Create Multiple-Column Indexes

Multiple-column indexes are natural extensions of single-column indexes. Multiple-column indexes are useful for evaluating filter expressions that match a prefix set of key columns. For example, the composite index CREATE INDEX Idx_Emp_Name ON Employees ("Last Name" ASC, "First Name" ASC) helps evaluate the following queries:

- ... WHERE "Last Name" = 'Doe'
- ... WHERE "Last Name" = 'Doe' AND "First Name" = 'John'
- ... WHERE "First Name" = 'John' AND "Last Name" = 'Doe'

However, it is not useful for this query:

- ... WHERE "First Name" = 'John'

When you create a multiple-column index, you should put the most selective columns leftmost in the key. This makes the index more selective when matching several expressions.

Avoid Indexing Small Tables

A small table is one whose contents fit in one or just a few data pages. Avoid indexing very small tables because it is typically more efficient to do a table scan. This saves the cost of loading and processing index pages. By not creating an index on very small tables, you remove the chance of the optimizer selecting one.

SQL Server Compact stores data in 4 Kb pages. The page count can be approximated by using the following formula, although the actual count might be slightly larger because of the storage engine overhead.

<sum of sizes of columns in bytes> * <# of rows>

<# of pages> = ----------

4096

For example, suppose a table has the following schema:

Column Name	Type (size)
Order ID	INTEGER (4 bytes)
Product ID	INTEGER (4 bytes)
Unit Price	MONEY (8 bytes)
Quantity	SMALLINT (2 bytes)
Discount	REAL (4 bytes)

The table has 2820 rows. According to the formula, it takes about 16 pages to store its data:

<# of pages> = ((4 + 4 + 8 + 2 + 4) * 2820) / 4096 = 15.15 pages

Choose What to Index

We recommend that you always create indexes on primary keys. It is frequently useful to also create indexes on foreign keys. This is because primary keys and foreign keys are frequently used to join tables. Indexes on these keys lets the optimizer consider more efficient index join algorithms. If your query joins tables by using other columns, it is frequently helpful to create indexes on those columns for the same reason.

When primary key and foreign key constraints are created, SQL Server Compact automatically creates indexes for them and takes advantage of them when optimizing queries. Remember to keep primary keys and foreign keys small. Joins run faster this way.

Use Indexes with Filter Clauses

Indexes can be used to speed up the evaluation of certain types of filter clauses. Although all filter clauses reduce the final result set of a query, some can also help reduce the amount of data that must be scanned.

A search argument (SARG) limits a search because it specifies an exact match, a range of values, or a conjunction of two or more items joined by AND. It has one of the following forms:

- Column operator <constant or variable>
- <constant or variable> operator Column

SARG operators include =, >, <, >=, <=, IN, BETWEEN, and sometimes LIKE (in cases of prefix matching, such as LIKE 'John%'). A SARG can include multiple conditions joined with an AND. SARGs can be queries that match a specific value, such as:

- "Customer ID" = 'ANTON'
- 'Doe' = "Last Name"

SARGs can also be queries that match a range of values, such as:

- "Order Date" > '1/1/2002'
- "Customer ID" > 'ABCDE' AND "Customer ID" < 'EDCBA'
- "Customer ID" IN ('ANTON', 'AROUT')

An expression that does not use SARG operators does not improve performance, because the SQL Server Compact query processor has to evaluate every row to determine whether it meets the filter clause. Therefore, an index is not useful on expressions that do not use SARG operators. Non-SARG operators include NOT, <>, NOT EXISTS, NOT IN, NOT LIKE, and intrinsic functions.

Use the Query Optimizer

When determining the access methods for base tables, the SQL Server Compact optimizer determines whether an index exists for a SARG clause. If an index exists, the optimizer evaluates the index by calculating how many rows are returned. It then estimates the cost of finding the qualifying rows by using the index. It will choose indexed access if it has lower cost than table scan. An index is potentially useful if its first column or prefix set of columns are used in the SARG, and the SARG establishes a lower bound, upper bound, or both, to limit the search.

Understand Response Time vs. Total Time

Response time is the time it takes for a query to return the first record. **Total time** is the time it takes for the query to return all records. For an interactive application, response time is important because it is the perceived time for the user to receive visual affirmation that a query is being processed. For a batch application, total time reflects the overall throughput. You have to determine what the performance criteria are for your application and queries, and then design accordingly.

For example, suppose the query returns 100 records and is used to populate a list with the first five records. In this case, you are not concerned with how long it takes to return all 100 records. Instead, you want the query to return the first few records quickly, so that you can populate the list.

Many query operations can be performed without having to store intermediate results. These operations are said to be pipelined. Examples of pipelined operations are projections, selections, and joins. Queries implemented with these operations can return results immediately. Other operations, such as **SORT** and **GROUP-BY**, require using all their input before returning results to their parent operations. These operations are said to require materialization. Queries implemented with these operations typically have an initial delay because of materialization. After this initial delay, they typically return records very quickly.

Queries with response time requirements should avoid materialization. For example, using an index to implement **ORDER-BY** yields better response time than does using sorting. The following section describes this in more detail.

Index the ORDER-BY / GROUP-BY / DISTINCT Columns for Better Response Time

The **ORDER-BY**, **GROUP-BY**, and **DISTINCT** operations are all types of sorting. The SQL Server Compact query processor implements sorting in two ways. If records are already sorted by an index, the processor needs to use only the index. Otherwise, the processor has to use a temporary work table to sort the records first. Such preliminary sorting can cause significant initial delays on devices with lower power CPUs and limited memory, and should be avoided if response time is important.

In the context of multiple-column indexes, for **ORDER-BY** or **GROUP-BY** to consider a particular index, the **ORDER-BY** or **GROUP-BY** columns must match the prefix set of index columns with the exact order. For example, the index CREATE INDEX Emp_Name ON Employees ("Last Name" ASC, "First Name" ASC) can help optimize the following queries:

- ... ORDER BY / GROUP BY "Last Name" ...
- ... ORDER BY / GROUP BY "Last Name", "First Name" ...

It will not help optimize:

- ... ORDER BY / GROUP BY "First Name" ...
- ... ORDER BY / GROUP BY "First Name", "Last Name" ...

For a DISTINCT operation to consider a multiple-column index, the projection list must match all index columns, although they do not have to be in the exact order. The previous index can help optimize the following queries:

- ... DISTINCT "Last Name", "First Name" ...
- ... DISTINCT "First Name", "Last Name" ...

It will not help optimize:

- ... DISTINCT "First Name" ...
- ... DISTINCT "Last Name" ...

Note
If your query always returns unique rows on its own, avoid specifying the **DISTINCT** keyword, because it only adds overhead

Rewrite Subqueries to Use JOIN

Sometimes you can rewrite a subquery to use JOIN and achieve better performance. The advantage of creating a JOIN is that you can evaluate tables in a different order from that defined by the query. The advantage of using a subquery is that it is frequently not necessary to scan all rows from the subquery to evaluate the subquery expression. For example, an EXISTS subquery can return TRUE upon seeing the first qualifying row.

Note
The SQL Server Compact query processor always rewrites the IN subquery to use JOIN. You do not have to try this approach with queries that contain the IN subquery clause.

For example, to determine all the orders that have at least one item with a 25 percent discount or more, you can use the following EXISTS subquery:

```
SELECT "Order ID" FROM Orders O
WHERE EXISTS (SELECT "Order ID"
FROM "Order Details" OD
WHERE O."Order ID" = OD."Order ID"
AND Discount >= 0.25)
```

You can also rewrite this by using JOIN:

```
SELECT DISTINCT O."Order ID" FROM Orders O INNER JOIN "Order Details"
OD ON O."Order ID" = OD."Order ID" WHERE Discount >= 0.25
```

Limit Using Outer JOINs

OUTER JOINs are treated differently from INNER JOINs in that the optimizer does not try to rearrange the join order of OUTER JOIN tables as it does to INNER JOIN tables. The outer table (the left table in LEFT OUTER JOIN and the right table in RIGHT OUTER JOIN) is accessed first, followed by the inner table. This fixed join order could lead to execution plans that are less than optimal.

For more information about a query that contains INNER JOIN, see Microsoft Knowledge Base.

Use Parameterized Queries

If your application runs a series of queries that are only different in some constants, you can improve performance by using a parameterized query. For example, to return orders by different customers, you can run the following query:

```
SELECT "Customer ID" FROM Orders WHERE "Order ID" = ?
```

Parameterized queries yield better performance by compiling the query only once and executing the compiled plan multiple times. Programmatically, you must hold on to the command object that contains the cached query plan. Destroying the previous command object and creating a new one destroys the cached plan. This requires the query to be re-compiled. If you must run several parameterized queries in interleaved manner, you can create several command objects, each caching the execution plan for a parameterized query. This way, you effectively avoid re-compilations for all of them.

Query Only When You Must

The SQL Server Compact query processor is a powerful tool for querying data stored in your relational database. However, there is an intrinsic cost associated with any query processor. It must compile, optimize, and generate an execution plan before it starts doing the real work of performing the plan. This is particularly true with simple queries that finish quickly. Therefore, implementing the query yourself can sometimes provide vast performance improvement. If every millisecond counts in your critical component, we recommend that you consider the alternative of implementing the simple queries yourself. For large and complex queries, the job is still best left to the query processor.

For example, suppose you want to look up the customer ID for a series of orders arranged by their order IDs. There are two ways to accomplish this. First, you could follow these steps for each lookup:

1. Open the Orders base table
2. Find the row, using the specific "Order ID"
3. Retrieve the "Customer ID"

Or you could issue the following query for each lookup:

```
SELECT "Customer ID" FROM Orders WHERE "Order ID" = <the specific order id>
```

The query-based solution is simpler but slower than the manual solution, because the SQL Server Compact query processor translates the declarative SQL statement into the same three operations that you could implement manually. Those three steps are then performed in sequence. Your choice of which method to use will depend on whether simplicity or performance is more important in your application.

8.3.3 Database design

- ✓ Proper tuning of database design is essential to high performance of the database.
- ✓ Normalization of logical database design yields the best performance improvement of database.
- ✓ Normalization is the process of breaking down a single table into many small tables with few fields(columns).

Database design and implementation is the cornerstone of any data centric project (read 99.9% of business applications) and should be treated as such when you are developing. This article, while probably a bit preachy, is as much a reminder to me as it is to anyone else who reads it. Some of the tips, like planning properly, using proper normalization, using a strong naming standards and documenting your work- these are things that even the best DBAs and data architects have to fight to make happen. In the heat of battle, when your manager's manager's manager is being berated for things taking too long to get started, it is not easy to push back and remind them that they pay you now, or they pay you later. These tasks pay dividends that are very difficult to quantify, because to quantify success you must fail first. And even when you succeed in one area, all too often other minor failures crop up in other parts of the project so that some of your successes don't even get noticed.

Common Database Design Mistakes

1. Poor design/planning
   it is impossible to predict every need that your design will have to fulfill and every issue that is likely to arise, but it is important to mitigate against potential problems as much as possible, by careful planning.
2. Ignoring normalization
   Normalization defines a set of methods to break down tables to their constituent parts until each table represents one and only one "thing", and its columns serve to fully describe only the one "thing" that the table represents.
3. Poor naming standards
   Use table name as **tblCustomer**, use column name as **part_number**, **partNumber** or **PartNumber**
4. Lack of documentation
   Not only will a well-designed data model adhere to a solid naming standard, it will also contain definitions on its tables, columns, relationships, and even default and check constraints, so that it is clear to everyone how they are intended to be used
5. One table to hold all domain values
   it is better to do the work upfront, making structures solid and maintainable, rather than trying to attempt to do the least amount of work to start out a project. By keeping tables down to representing one "thing" it means that most changes will only affect one table, after which it follows that there will be less rework for you down the road.
6. Using identity/guid columns as your only key
   each of your tables should have a natural key that means something to the user, and can uniquely identify each row in your table. In the very rare event that you cannot find a natural key (perhaps, for example, a table that provides a log of events), then use an artificial/surrogate key.
7. Not using SQL facilities to protect data integrity
   All fundamental, non-changing business rules should be implemented by the relational engine. The **base rules** of nullability, string length, assignment of foreign keys, and so on, should all be defined **in the database**.
8. Not using stored procedures to access data : Maintainability, encapsulation, security, performance,
   Stored procedures are your friend. Use them whenever possible as a method to insulate the database layer from the users of the data.
9. Trying to build generic objects : to reusable
10. Lack of testing :

8.4 Network Performance

Network performance refers to measures of service quality of a network as seen by the customer.

There are many different ways to measure the performance of a network, as each network is different in nature and design. Performance can also be modeled and simulated instead of measured; one example of this is using state transition diagrams to model queuing performance or to use a Network Simulator

Performance Measures

The following measures are often considered important:

- **Bandwidth** commonly measured in bits/second is the maximum rate that information can be transferred
- **Throughput** is the actual rate that information is transferred
- **Latency** the delay between the sender and the receiver decoding it, this is mainly a function of the signals travel time, and processing time at any nodes the information traverses
- **Jitter** variation in packet delay at the receiver of the information

- **Error rate** the number of corrupted bits expressed as a percentage or fraction of the total sent

One of the principle causes of poor data transfer performance is packet loss between the data transfer client and server hosts. There are many possible causes of packet loss, ranging from bad or failing hardware to misconfigured hosts or network equipment. Network design, coupled with proper configuration of hosts, routers and switches, can eliminate packet loss and result in significantly improved data transfer performance.

These days, whenever anyone talks about networking technologies, particularly for home offices or small businesses, the conversation inevitably turns to the latest in wireless technologies. While you can easily see the benefits of implementing a wireless network (no cables to trip over, ease of bringing the network into every room, lower implementation cost and so on), don't forget about the reliability or superior performance afforded by a good old-fashioned wired Ethernet network. However, just because a technology is old, doesn't mean it can't be improved upon.

For instance, if your network continues to operate with hubs, the obvious change to make to immediately boost network performance is to replace those hubs with more efficient switches. The difference is that hubs broadcast traffic to all available ports on the network, switches direct traffic only to their destinations, such as a specific server, router or workstation.

Over the last decade or so the recommended approach has been to move Ethernet networks to Fast Ethernet switching by installing 10/100 dual speed switches in the wiring closet and upgrading key desktop and server connections to 100Mbps.

While these speeds have suited our needs for a long time, many applications and services are available today that has started to push the performance of our networks to their limits. As a result, it isn't uncommon for users to experience slow network response times during peak hours of operation.

Get Giga With It

Fortunately, there are a variety of ways to circumvent data bottlenecks. Yet it doesn't take a rocket scientist to figure out that the most efficient way to help alleviate network congestion is to widen the available network bandwidth. To this end, more and more networking professionals are turning to Gigabit Ethernet [define] technology.

Gigabit Ethernet is capable of increasing data transmission rates up to 1000Mbps, often using the cabling that is already in place. That's 100 times faster than regular 10Mbps Ethernet and 10 times faster than 100Mbps Fast Ethernet. Gigabit switches are quick and easy to install and have the capability to transfer very large files such as CAD or desktop publishing files, stream audio and video, host LAN collaboration software, enhance LAN gaming, and overall improve the performance of speed-intensive applications across your network.

When Gigabit Ethernet was standardized for fiber optic [define] cabling in the late 1990s, IT managers began to see the benefits of Gigabit speeds applied to the network backbone and in the data center. Despite its many benefits, though, bringing Gigabit technology to home offices and small businesses wasn't particularly feasible as its cost per port was prohibitively high.

In recent years, though, Gigabit Ethernet has become a much more practical and economical option for home offices and small businesses alike. Today it now makes sense to consider Gigabit to the desktop as a viable option when upgrading or setting up a new network. IT managers have found that Gigabit Ethernet is simple, easy to use and is the perfect performance enhancement for IT departments on a tight budget.

Able to Scale Ethernet Speeds

Since it is a standards-based technology, an organization can scale from 10 to 100 or 1000Mbps Ethernet, either network-wide or a segment at a time, knowing that the new equipment will be backwards compatible with their existing legacy equipment, such as LAN printers and older servers. This reduces the infrastructure investment that an organization must make, providing you with not just speed, but value as well.

The best part is that all you need to do to achieve these performance increases is to substitute this switch with your current workgroup hub or switch. In fact since the majority of new PCs sold within the last two or three years came standard with gigabit Ethernet adapters, these systems will immediately benefit from the new switch, while continuing to service other clients at their current speeds. Allowing you to upgrade your other workstations to full Gigabit speeds as needed.

Even older PCs equipped with only Fast Ethernet adapters will see a performance increase — admittedly not to the same degree as the others, but a noticeable one none the less. Ethernet is a reliable technology and experience shows that it can be deployed with confidence for mission-critical applications. Gigabit Ethernet is also a good choice because it supports Quality of Service (QoS) [define] methods that are increasingly important for avoiding latency problems as voice, video and data share the cable for Next-Generation Networking (NGN) [define] applications.

There are two basic types of switches available: unmanaged and managed. Some networks will make use of both types. Most small businesses in need of a simple networking solution should make out fine with a low-cost unmanaged gigabit switch. However, if you would like to take advantages of some advanced networking features, like remote management and port management, for example, then you might want to take a closer look at managed switches. These are typically much more expensive then the unmanaged variety.

Cost of gigabit switches can vary greatly. They can be had for as little as $45 for a small, unmanaged, 5-port switch to as high as several thousands of dollars for a 24-48 port managed switch/router with all the bells and whistles. Be sure to discuss your needs with a qualified IT professional before making a purchase.

In case you didn't realize it, installing a Gigabit switch on your network will significantly increase the performance of your LAN traffic, with perhaps only the most modest effect on your WAN [define] connection. Your WAN speed will still be dictated by your ISP.

So if you're looking for an inexpensive way to improve your network performance, take a closer look at th at old wired Ethernet connection of yours. Even if the majority of your networks PCs connect over WiFi, it can't hurt to have a nice fat gigabit pipe availabl e when you need it.

Summary of the advantages of Gigabit Technology:

- Increased bandwidth for higher performance and the virtual elimination of bottlenecks.
- Full-duplex capacity, allowing the effective bandwidth to be virtually doubled.
- Quality of Service (QoS) features to help eliminate jittery video or distorted audio.
- Low cost of acquisition and ownership. Prices of Gigabit desktop switches have dropped considerably. For only a bit more than the cost of 10/100, users can get the added performance of 10/100/1000.
- Full compatibility with the large installed base of Ethernet and Fast Ethernet nodes
- Transferring large amounts of data across a network quickly. Gigabit performance allows graphics-intensive processing, high-bandwidth file sharing, centralized backup, and network-based "ghosting" of configurations.
- Increase employee productivity by speeding access to databases and messaging applications.
- Compatible with existing copper, eliminating the expense of running new wiring.
- Leverage networking standards to get more efficiency from a network.
- Add network management to centrally deal with larger network environments.

8.4.1 Data rate

The data rate is a term to denote the transmission speed, or the number of bits per second transferred. The useful data rate for the user is usually less than the actual data rate transported on the network. One reason for this is that additional bits are transferred for e.g signaling, the address, the recovery of timing information at the receiver or error correction to compensate for possible transmission errors.

Data rate is the amount of data in bits per second of footage, *measured per second and expressed as bits per second*.

Bit rate is the number of bits per second of footage, *measured per frame or field*, *also expressed as bits per second*.

8.4.2 Bandwidth :

Boost Your Bandwidth, Boost your Network Performance

commonly measured in bits/second is the maximum rate that information can be transferred. The available channel bandwidth and achievable signal-to-noise ratio determine the maximum possible throughput.

8.4.3 Throughput : is the actual rate that information is transferred. *Throughput* is the number of messages successfully delivered per unit time. Throughput is controlled by available bandwidth, as well as the available signal-to-noise ratio and hardware limitations. Throughput for the purpose of this article will be understood to be measured from the arrival of the first bit of data at the receiver, to decouple the concept of throughput from the concept of latency. For discussions of this type the terms 'throughput' and 'band width' are often used interchangeably.

The *Time Window* is the period over which the throughput is measured. Choice of an appropriate time window will often dominate calculations of throughput, and whether latency is taken into account or not will determine whether the latency affects the t hroughput or not.

8.4.4 Congestion

Refers to a network state where a node or link carries so much data that it may deteriorate network service quality, resulting in queuing delay, frame or data packet loss and the blocking of new connections. In a congested ne twork, response time slows with reduced network throughput. Congestion occurs when bandwidth is insufficient and network data traffic exceeds capacity.

Avoiding network congestion and collapse requires two major components:

- Routers capable of reordering or dropping data packets when received rates reach critical levels
- Flow control mechanisms that respond appropriately when data flow rates reach critical levels