***Performance Tuning – SQL Server Part – 1***

**Q. What are the bottlenecks that effects the performance of a Database / Application**

**Ans:**

The top performance bottlenecks for OLTP applications are outlined as:

* Database Design \ Database Code
* Application Design \ Application Code
* CPU bottleneck
* Memory bottleneck
* IO bottleneck
* Blocking bottleneck
* Network bottleneck
* Server Hardware

**Database Design \ Database Code**

* **Too many indexes on frequently updated (inclusive of inserts, updates and deletes):**

Tables incur extra index maintenance overhead. Generally, OLTP database designs should keep the number of indexes to a *functional minimum*, again due to the high volumes of similar transactions combined with the cost of index maintenance

* Statistics may not be updated or missing statistics
* Excess use of cursors and temporary tables
* Too much of normalization
* Do not use the conversion/system/user defined functions in where clause
* **Unused indexes incur the cost of index maintenance for inserts, updates, and deletes without benefiting any users:**

Unused indexes should be eliminated. Any index that has been used (by select, update or delete operations) will appear in sys.dm\_db\_index\_usage\_stats. Thus, any defined index not included in this DMV has not been used since the last re-start of SQL Server

* **Choose the Appropriate Data Types**

Always choose the smallest appropriate data type. Avoid NCHAR/NVARCHAR unless there is a need of storing Unicode.

* **Use Triggers Cautiously**

Keep the code in your triggers to the very minimum to reduce overhead. The more code that runs in the trigger, the slower each INSERT, UPDATE, and DELETE that fires it will be

* **Don’t Access More Data Than You Need**

Don’t return more columns or rows of data to the client than absolutely necessary. This just increases disk I/O on the server

* **Avoid Using Cursors**
* Wherever possible Try to use alternative solutions includes Temp-Tables, Derived tables, Table Variables or Recursive CTE’s etc
* Always select the cursor with the least amount of overhead. The most efficient cursor you can choose is the fast forward-only cursor.
* When you are done using a cursor, don’t just CLOSE it, DEALLOCATE
* **Use Joins Appropriately**
* If you have two or more tables that are frequently joined together, then the columns used for the joins should have an appropriate index. If the columns used for the joins are not naturally compact, then considering adding surrogate keys to the tables that are compact in order to reduce the size of the keys, thus decreasing read I/O during the join process, and increasing overall performance. You will learn more about indexing in the next section of this article.
* For best performance, the columns used in joins should be of the same data types. And if possible, they should be numeric data types rather than character types.
* Avoid joining tables based on columns with few unique values. If columns used for joining aren’t mostly unique, then the SQL Server optimizer will perform a table scan for the join, even if an index exists on the columns. For best performance, joins should be done on columns that have unique indexes.
* If you have to regularly join four or more tables to get the recordset you need, consider denormalizing the tables so that the number of joined tables is reduced. Often, by adding one or two columns from one table to another, joins can be reduced.
* Generally, frequent operations requiring 5 or more table joins should be avoided by redesigning the database
* **Encapsulate Your Code in Stored Procedures**
* Try to put all your T-SQL code in stored procedures which reduces the network traffic by just calling the proc from application and reduces the I/O overhead by using the compiled execution plan
* Always use the option “SET NOCOUNT ON”
* Design the proc’s to avoid the deadlocks
* Collect all inputs before the transaction begins
* Keep transaction short with in a batch
* Use the correct isolation levels
* Try to use with no lock option

**Application Design / Application code:**

**Application Design issues:**

* Perform as many data-centered tasks as possible on SQL Server in the form of stored procedures. Avoid manipulating data at the presentation and business services tiers.
* Don’t maintain state (don’t store data from the database) in the business services tier. Maintain state in the database as much as possible
* Don’t create complex or deep object hierarchies. The creation and use of complex classes or a large number of objects used to model complex business rules can be resource intensive and reduce the performance and scalability of your application. This is because the memory allocation when creating and freeing these objects is costly.
* Consider designing the application to take advantage of database connection pooling and object pooling using Microsoft Transaction Server (MTS). MTS allows both database connections and objects to be pooled, greatly increasing the overall performance and scalability of your application.
* If your application runs queries against SQL Server that by nature are long, design the application to be able to run queries asynchronously. This way, one query does not have to wait for the next before it can run. One way to build in this functionality into your n-tier application is to use the Microsoft Message Queue Server (MSMQ).

**Application Code:**

* **Use OLE DB to Access SQL Server:** 
  + You can access SQL Server data using either ODBC or OLE DB. For best performance, always select OLE DB. OLE DB is used natively by SQL Server, and is the most effective way to access any SQL Server data.
* **Use DSN-less in Connection String:**
  + While creating an ADO connection to SQL Server, you can either use a DSN in the connection string, or you can use a DSN-less connection. For optimal performance, use DSN-less connections. Using them prevents the need for the OLE DB driver to look up connection string information in the registry of the client the application code is running on, saving some overhead.
* **Encapsulate your DML (Data Manipulation Language) in Stored Procedures**
  + ADO allows you three different ways to SELECT, INSERT, UPDATE, or DELETE data in a SQL Server database. You can use ADO’s methods, you can use dynamic SQL, or you can use stored procedures. For better performance prefer Stored Procedures
* **Encapsulate Your ADO Code in COM Components**
  + Put the ADO code that accesses SQL Server data into COM components. This gives you all the standard benefits of COM components, such as object pooling using MTS. And for ASP-based applications, it provides greater speed because the ADO code in COM objects is already compiled, unlike ADO code found in ASP pages. How you implement your data manipulation code in COM components should be considered when the application is first designed.
  + For optimum performance, COM objects should be compiled as in-process DLLs (which is required if they are to run under MTS). You should always employ early binding when referencing COM objects, and create them explicitly, not implicitly.

**CPU bottlenecks:**

* Signal waits > 25% of total waits.

(See sys.dm\_os\_wait\_stats for Signal waits and Total waits. Signal waits measure the time spent in the runnable queue waiting for CPU. High signal waits indicate a CPU bottleneck.)

* Plan re-use < 90%.

(A query plan is used to execute a query. Plan re-use is desirable for OLTP workloads because re-creating the same plan (for similar or identical transactions) is a waste of CPU resources. Compare SQL Server SQL Statistics: batch requests/sec to SQL compilations/sec. Compute plan re-use as follows: Plan re-use = (Batch requests – SQL compilations) / Batch requests. Special exception to the plan re-use rule: Zero cost plans will not be cached (not re-used) in SQL 2005 SP2. Applications that use zero cost plans will have a lower plan re-use but this is not a performance issue.)

**Memory bottleneck:**

* Consistently low average page life expectancy. (MSSQL$Instance: Buffer Manager\Page Life Expectancy:)

(See Average Page Life Expectancy Counter which is in the Perfmon object SQL Server Buffer Manager (this represents is the average number of seconds a page stays in cache). For OLTP, an average page life expectancy of 300 is 5 minutes. Anything less could indicate memory pressure, missing indexes, or a cache flush)

* Consistently low SQL Cache hit ratio. (MSSQL$Instance: Plan Cache\Cache Hit Ratio:)

(OLTP applications (e.g. small transactions) should have a high cache hit ratio. Since OLTP transactions are small, there should not be (1) big drops in SQL Cache hit rates or (2) consistently low cache hit rates < 90%. Drops or low cache hit may indicate memory pressure or missing indexes.)

**IO bottleneck:**

* High average disk seconds per read.

(When the IO subsystem is queued, disk seconds per read increases. See Perfmon Logical or Physical disk (disk seconds/read counter). Normally it takes 4-8ms to complete a read when there is no IO pressure. When the IO subsystem is under pressure due to high IO requests, the average time to complete a read increases, showing the effect of disk queues. Periodic higher values for disk seconds/read may be acceptable for many applications. For high performance OLTP applications, sophisticated SAN subsystems provide greater IO scalability and resiliency in handling spikes of IO activity. Sustained high values for disk seconds/read (>15ms) does indicate a disk bottleneck.)

* High average disk seconds per write.

(See Perfmon Logical or Physical disk. The throughput for high volume OLTP applications is dependent on fast sequential transaction log writes. A transaction log write can be as fast as 1ms (or less) for high performance SAN environments. For many applications, a periodic spike in average disk seconds per write is acceptable considering the high cost of sophisticated SAN subsystems. However, sustained high values for average disk seconds/write is a reliable indicator of a disk bottleneck.)

* Big IOs such as table and range scans due to missing indexes.

**Blocking bottleneck:**

* High average row lock or latch waits.

(The average row lock or latch waits are computed by dividing lock and latch wait milliseconds (ms) by lock and latch waits. The average lock wait ms computed from sys.dm\_db\_index\_operational\_stats represents the average time for each block.)

* Top wait statistics
* High number of deadlocks.

(See Profiler “Graphical Deadlock” under Locks event to identify the statements involved in the deadlock.)

**Network bottleneck:**

* High network latency coupled with an application that incurs many round trips to the database.
* Network bandwidth is used up.

(See counters packets/sec and current bandwidth counters in the network interface object of Performance Monitor. For *TCP/IP frames actual bandwidth is computed as packets/sec \* 1500 \* 8 /1000000 Mbps*)

**Server Hardware:**

Most slow applications are slow because of poor up front design, not because of slow hardware. Since the application’s design can’t be changed at the time when deployed to production, about the only thing you can try to help boost performance is to throw hardware at it.

* *CPU*: Always purchase a server with the ability to expand its number of CPUs. Usually it goes for larger servers with four or more CPUs. Always leave room for growth.
* *Memory*: Try to get enough RAM to hold the largest table you expect to have, and if you can afford it, get all the RAM your server can handle, which is often 2GB or more.
* *I/O Subsystem*: At the very minimum, purchase hardware-based RAID for your databases. As a rule of thumb, you will to purchase more – smaller drives, not fewer – larger drives in your array. The more disks that are in an array, the faster I/O will be.
* *Network Connection*: At the server, have at least one 100Mbs network card, and it should be connected to a switch. Ideally, you should have two network cards in the server connected to a switch in full-duplex mode.

For best performance on a server, SQL Server should be the only application running on the server, other than management utilities. Don’t try to save a few bucks by putting your IIS server on the same server as SQL Server.

**Q. What is the process of tuning the Performance?**

**Ans:**

* **Identification** – Use native tools like Profiler, Query Tuning Advisor, Query Execution Plans, Performance Monitor, system stored procedures, dynamic management views, custom stored procedures or third party tools
* **Analysis** – Analyze the data to determine the core problems
* **Providing Solution** –
* Creating new index on appropriate columns
* Altering the complex quires to make them use the existing indexes.
* By Updating Statistics for Tables and Views.
* By Rebuilding and Reorganizing indexes.
* By Resolving blocking problems.
* By removing Deadlocks.
* **Testing** – Test the various options to ensure they perform better and do not cause worse performance in other portions of the application
* **Knowledge sharing** – Share your experience with the team to ensure they understand the problem and solution, so the issue does not occur again

**Q. How to choose the correct (Clustered/ Non- Clustered) index on a column?**

**Ans:**

**Selecting Clustered Index:**

* Clustered indexes are ideal for queries that select by a range of values or where you need sorted results. Examples of this include when you are using BETWEEN, <, >, GROUP BY, ORDER BY, and aggregates such as MAX, MIN, and COUNT in your queries.
* Clustered indexes are good for queries that look up a record with a unique value (such as an employee number) and when you need to retrieve most or all of the data in the record.
* Clustered indexes are good for queries that access columns with a limited number of distinct values, such as columns that holds country or state data. But if column data has little distinctiveness, such as columns with a yes or no, or male or female, then these columns should not be indexed at all.
* Avoid putting a clustered index on columns that increment, such as an identity, date, or similarly incrementing columns, if your table is subject to a high level of INSERTS.

**Selecting Non – Clustered Index:**

* Non-clustered indexes are best for queries that return few rows (including just one row) and where the index has good selectivity (above 95%).
* If a column in a table is not at least 95% unique, then most likely the SQL Server Query Optimizer will not use a non-clustered index based on that column. For example, a column with “yes” or “no” as the data won’t be at least 95% unique.
* Keep the “width” of your indexes as narrow as possible, especially when creating composite (multi-column) indexes. This reduces the size of the index and reduces the number of reads required to read the index, boosting performance.
* If possible, try to create indexes on columns that have integer values instead of characters. Integer values have less overhead than character values.
* If you know that your application will be performing the same query over and over on the same table, consider creating a covering index on the table. A covering index includes all of the columns referenced in the query.
* An index is only useful to a query if the WHERE clause of the query matches the column(s) that are leftmost in the index. So if you create a composite index, such as “City, State”, then a query such as “WHERE City = ‘Houston'” will use the index, but the query “WHERE STATE = ‘TX'” will not use the index.

**Q. How to read the graphical execution plan?**

**Ans:**

The plan should be read from right to left

* Check the Graphical execution plan of a stored procedure / Query
* Table Scan – Index is missing
* Index Scan – Proper indexes are not using
* BookMark Lookup – Limit the number of columns in the select list
* Filter – Remove any functions from where clause, May require additional indexes
* Sort – Does the data really need to be sorted? Can an index be used to avoid sorting? Can sorting be done at the client more efficiently?
* DataFlow Arrow – High density: Sometimes you find few rows as outcome but the arrow line density indicates the query/proc processing huge number of rows
* Cost – Can easily find out which table / operation taking much time
* From the execution plan we can find out the bottleneck and give the possible solution to avoid the latency

**Q. Why the Actual and Estimated Execution Plans Might Differ**

**Ans**

* **When Statistics are Stale:**

The main cause of a difference between the plans is differences between the *statistics* and the actual data. This generally occurs over time as data is added and deleted.

* **When the Estimated plan is invalid:**

When the batch contains temporary tables or the T-SQL statements which refers some of the objects that are not currently existed in the database, but will be created once the batch is run. (Create table is there in batch)

**Q. What are the permissions required to view execution plans?**

**Ans:**

Either the user must be mapped to sysadmin, db\_owner, db\_creator or he/she will be granted the permission “Show Plan”.

GRANT SHOWPLAN TO [username]

**Q. What are the tools available for performance tuning/monitoring?**

**Ans:**

* **Performance Studio:** Act as a Central Data Repository, Collect Selected SQL Server Performance Data and Display Performance Reports
* **Activity Monitor:** It displays graphically about Processes, Resource Waits, Datafile I/O, Recent expensive Quires.
* **Database Tuning Advisor (DTA):** Recommend indexes
* **Profiler:** Can run traces and find out the expensive/long running quires/transactions
* **Execution Plans:** There are three types Graphical, Text and XML.
* **DMV:** Dynamic management views shows the current state of the sql server
* **PerfMon:** Windows native tool to view / monitor the performance of both sql and windows servers
* **Third Party:** Redgate products

**Q. How to identify the CPU bottlenecks and how to resolve it?**

**Ans:**

**Identifying CPU Bottlenecks:**

Firstly we have to confirm that SQL Server – CPU utilization is high. Run the below query

SELECT Timestamp, CONVERT(XML, record) AS XmlRecord

FROM SYS.DM\_OS\_RING\_BUFFERS

WHERE ring\_buffer\_type = N’RING\_BUFFER\_SCHEDULER\_MONITOR’

AND record like ‘%<SystemHealth>%’

ORDER BY timestamp DESC

One record is stored every minute up to a maximum of 256 records. Clicking on any of the XML links will take you to the XML editor and will show an entry similar to below

<Record id=”434″ type=”RING\_BUFFER\_SCHEDULER\_MONITOR” time=”22398046″>

<SchedulerMonitorEvent>

<SystemHealth>

<ProcessUtilization>**55**</ProcessUtilization>

<SystemIdle>**35**</SystemIdle>

<UserModeTime>228180000</UserModeTime>

<KernelModeTime>251812000</KernelModeTime>

<PageFaults>64252</PageFaults>

<WorkingSetDelta>21770240</WorkingSetDelta>

<MemoryUtilization>100</MemoryUtilization>

</SystemHealth>

</SchedulerMonitorEvent>

</Record>

Information from above XML:

ProcessUtilization: Percentage of CPU utilized by SQL Server – 55%

SystemIdle: Percentage of Idle CPU – 35%

Other processes using CPU: 100- (55+35) = 10 %

**Now find out the query/proc/process that is making CPU utilization High:**

SELECT TOP 20

qst.sql\_handle,

qst.execution\_count,

qst.total\_worker\_time AS Total\_CPU,

total\_CPU\_inSeconds = –Converted from microseconds

qst.total\_worker\_time/1000000,

average\_CPU\_inSeconds = –Converted from microseconds

(qst.total\_worker\_time/1000000) / qst.execution\_count,

qst.total\_elapsed\_time,

total\_elapsed\_time\_inSeconds = –Converting from microseconds

qst.total\_elapsed\_time/1000000,

st.text AS ‘Query’,

qp.query\_plan

from

sys.dm\_exec\_query\_stats as qst

CROSS APPLY sys.dm\_exec\_sql\_text(qst.sql\_handle) as st

cross apply sys.dm\_exec\_query\_plan (qst.plan\_handle) as qp

ORDER BY qst.total\_worker\_time DESC

From the above script we can find the commands which are taking the most CPU time along with the execution plan. By reviewing the execution plan you can see what additional indexes need to be added to the database which will improve database performance and decrease the CPU load time.

By adding missing indexes or by using the proper indexes we can decrease the load on CPU.

**Other options:**

* **Sp\_monitor:** Displays statistics, including CPU usage, I/O usage, and the amount of time idle since sp\_monitor was last executed. We can get the information about the “CPU Time (Sec)”, “I/O Time (Sec)”, “Count of Input\Output Packets”, “No of logins attempted”, “Errors in reading/writing network packets” etc.
* **@@CPU\_BUSY / @@IO\_BUSY:** Returns the time that SQL Server has spent working since it was last started. Result is in CPU time increments, or “ticks,” and is cumulative for all CPUs, so it may exceed the actual elapsed time. Multiply by @@TIMETICKS to convert to microseconds. But it may not the accurate value to be considered.
* **PerfMon**
* **Profiler**

**Q. Can you tell me what the Wait Type “LAZY WRITTER” is?**

**Ans:**

The job of the lazy writer is to find dirty pages in the buffer pool and write them out to disk and drop those pages from cache.

**Q. Can we find performance bottleneck from sysprocesses?**

**Ans:**

Yes. We may not confirm that it is the only bottleneck but at least we can find the bottleneck. **Lastwaittype** column with **waittime** plays a vital role in identifying the issue. This is a very interesting column because it can tell you what the offending query is waiting for to complete.

**Network\_io:** There is too much of traffic in Network

**Cxpacket:** Your process is waiting on other parallel processes to complete.

**SOS\_SCHEDULER\_YIELD:** CPU bound. We may not have enough CPU in your box

**IO\_Completion:** Disk issue. We may not have enough disk space or running on corrupted disk array.

**Q. What Are SQL Server Waits?**

**Ans:**

Instead of measuring activity of CPU, storage, or memory, why not ask what SQL Server has been waiting on when executing queries?

In general there are three categories of waits that could affect any given request:

* **Resource waits** are caused by a particular resource, perhaps a specific lock that is unavailable when the requested is submitted.
* **External waits** occur when SQL Server worker thread is waiting on an external process
* **Queue waits** normally apply to internal background tasks, such as ghost cleanup, which physically removes records that have been previously deleted.

**Q. How could you know the statistics are outdated?**

**Ans:**

If old statistics is your problem, you will likely experience this as a gradual decline of SQL Server slowing down over many days or weeks, or you may have just upgraded your platform (from 2000 to 2008) and forgot to update the statistics. Out of date statistics cause inaccurate execution plans.

**Q. What are the main parameters we need to check when you are dealing with memory performance?**

**Ans:**

There are four significant properties of sql server.

**Max server memory** and **Min server memory:**

Use the two server memory options, min server memory and max server memory, to reconfigure the amount of memory (in megabytes) that is managed by the SQL Server Memory Manager for a SQL Server process used by an instance of SQL Server. By default Min Memory is set to be 0 and Max Memory is set to be 2147483647 MB (21 GB). Never leave these two settings as default. Depends on the memory available and other applications running on windows Server, change these two settings.

For example we have 24 GB available and the settings can be like this:

Min Memory: 1 GB

Max Memory: 16 GB

Remember total max memory of all instances should not exceeds the actual physical memory available

**Priority boost:** By default, the priority boost setting is 0, which causes SQL Server to run at a normal priority. If you set priority boost to 1, the SQL Server process runs at a high priority.

**Lightweight pooling:** Switch on this parameter when you want to make sql server use the fiber mode facility. Unless there is a real need and environment (Large multi-processor servers) available we should not use this option at production servers.

***Performance Tuning – SQL Server Part -2 – Indexes***

**Q. What are the primary differences between an index reorganization and an index rebuild?**

**Ans:**

* Reorganization is an “online” operation by default; a rebuild is an “offline” operation by default
* Reorganization only affects the leaf level of an index
* Reorganization swaps data pages in-place by using only the pages already allocated to the index; a rebuild uses new pages/allocations
* Reorganization is always a fully-logged operation; a rebuild can be a minimally-logged operation
* Reorganization can be stopped mid-process and all completed work is retained; a rebuild is transactional and must be completed in entirety to keep changes

**Q. During an index reorganization operation, if the index spans multiple files, will pages be allowed to migrate between files**

**Ans:**

No – pages will not migrate between files during an index reorganization.

**Q. If you need to REBUILD a non-clustered index that is 10GB in size and have 5GB of free data-file space available with no room to grow the data file(s), how can you accomplish the task?**

**Ans:**

When rebuilding an existing non-clustered index, you typically require free space that is approximately equivalent to 2.2 times the size of the existing index, since during a rebuild operation, the existing index is kept until the rebuilt structure is complete and an additional approximately 20% free space for temporary sorting structures used during the rebuild operation

* In this case, you would require at least an additional 10+ GB of free space for the rebuild operation to succeed, since the index itself is 10GB in size.
* Using SORT\_IN\_TEMPDB would not suffice in this case, since only the temporary sort tables are stored in tempdb in this case, and at least 10 GB of free space would still be required in the database data files.

Your possibilities are different for SQL Server2000 vs. SQL Server 2005/2008:

* **In SQL Server 2000**, you only have 1 option in this case. Drop the index and recreate it.
* **In SQL Server 2005/2008**, you can do the same as you did with SQL Server 2000 (drop, recreate), but you also have another option. If you first disable the index (via the ALTER INDEX…DISABLE statement) the existing space consumed by the index will be freed. Then running a simple ALTER INDEX…REBUILD command will allow the build operation to use the now 15gb of free space to build the index.

**Q. What is the Covering Index?**

**Ans:**

If you know that your application will be performing the same query over and over on the same table, consider creating a covering index on the table. A covering index, which is a form of a composite non clustered index, includes all of the columns referenced in SELECT, JOIN, and WHERE clauses of a query. Because of this, the index contains the data you are looking for and SQL Server doesn’t have to look up the actual data in the table, reducing logical and/or physical I/O, and boosting performance.

On the other hand, if the covering index gets too big (has too many columns), this could actually increase I/O and degrade performance.

**Q. What is Online Indexing?**

**Ans:**

Online Indexing is a new feature available in SQL Server 2005. In SQL Server 2005, DBAs can create, rebuild, or drop indexes online. The index operations on the underlying table can be performed concurrently with update or query operations. This was not possible in previous versions of SQL Server. In the past, indexing operations (reorganizing or rebuilding) were usually performed as a part of other maintenance tasks running during off-peak hours. During these offline operations, the indexing operations hold exclusive locks on the underlying table and associated indexes. During online index operations, SQL Server 2005 eliminates the need of exclusive locks

The Online indexing feature is very helpful for environments that run 24 hours a day, seven days a week. The Online Indexing feature is available only in the Enterprise Edition of SQL Server 2005/2008.

**Q. How Online Indexing works?**

**Ans:**

The online index operation can be divided into three phases:

* Preparation
* Build
* Final

The Build phase is a longest phase of all. It is in this phase where the creation, dropping, or rebuilding of indexes take place. The duration of the Build phase depends on the size of the data and the speed of the hardware. Exclusive locks are not held in this phase, so concurrent DML operations can be performed during this phase. The Preparation and Final phases are for shorter durations. They are independent of the size factor of the data. During these two short phases, the table or the indexed data is not available for concurrent DML operations.

**Q. How to know unused indexes in a table.**

**Ans:**

By using a DMV we‘ll find the unused indexes details. We have a DMV called “sys.dm\_db\_index\_usage\_stats” which retrieves the statistics of indexes , if an index id is not in that dmv then we can say that index is not been using from a long time.

**Q. What are the index limitations?**

**Ans:**

INDEXTYPE – 32 BIT / 64 BIT

NON CLUSTERED – 999 / 999

XML indexes 249 / 249

Columns per index key – 16 / 16 – Only 900 Byte

**Q. What are the different indexes available?**

**Ans:**

**Index Types**

* **Clustered Index:** A clustered index sorts and stores the data rows of the table or view in order based on the clustered index key. The clustered index is implemented as a B-tree index structure that supports fast retrieval of the rows, based on their clustered index key values.
* **Non Clustered Index:** A nonclustered index can be defined on a table or view with a clustered index or on a heap. Each index row in the nonclustered index contains the nonclustered key value and a row locator. This locator points to the data row in the clustered index or heap having the key value. The rows in the index are stored in the order of the index key values, but the data rows are not guaranteed to be in any particular order unless a clustered index is created on the table.
* **Unique Index:** An index that ensures the uniqueness of each value in the indexed column. If the index is a composite, the uniqueness is enforced across the columns as a whole, not on the individual columns. For example, if you were to create an index on the FirstName and LastName columns in a table, the names together must be unique, but the individual names can be duplicated. A unique index is automatically created when you define a primary key or unique constraint:
  + **Primary key:** When you define a primary key constraint on one or more columns, SQL Server automatically creates a unique, clustered index if a clustered index does not already exist on the table or view. However, you can override the default behavior and define a unique, nonclustered index on the primary key.
  + **Unique:** When you define a unique constraint, SQL Server automatically creates a unique, nonclustered index. You can specify that a unique clustered index be created if a clustered index does not already exist on the table.
* **Index with Included Columns:** A nonclustered index that is extended to include nonkey columns in addition to the key columns. They can be data types not allowed as index key columns. They are not considered by the Database Engine when calculating the number of index key columns or index key size. Need to use INCLUDE clause while creating index.
* **Full Text Indexes:** A special type of token-based functional index that is built and maintained by the Microsoft Full-Text Engine for SQL Server. It provides efficient support for sophisticated word searches in character string data.
* **Spatial Indexes:** A spatial index provides the ability to perform certain operations more efficiently on spatial objects (spatial data) in a column of the **geometry** data type. The spatial index reduces the number of objects on which relatively costly spatial operations need to be applied.
* **Filtered Index:** An optimized nonclustered index, especially suited to cover queries that select from a well-defined subset of data. It uses a filter predicate to index a portion of rows in the table. A well-designed filtered index can improve query performance, reduce index maintenance costs, and reduce index storage costs compared with full-table indexes. Use where clause in create index statement.
* **Composite index:** An index that contains more than one column. In both SQL Server 2005 and 2008, you can include up to 16 columns in an index, as long as the index doesn’t exceed the 900-byte limit. Both clustered and nonclustered indexes can be composite indexes.
* **XML Indexes:** A shredded, and persisted, representation of the XML binary large objects (BLOBs) in the xml data type column.

**Q. I have a situation where I need to create an index on 20 columns. Is it practically possible? If yes justify?**

**Ans:**

Yes ! We can create an index on 20 columns using INCLUDE clause.

You can include nonkey columns in a nonclustered index to avoid the current index size limitations of a maximum of 16 key columns and a maximum index key size of 900 bytes. The SQL Server Database Engine does not consider nonkey columns when calculating the number of index key columns or the total size of the index key columns. In a nonclustered index with included columns, the total size of the index key columns is restricted to 900 bytes. The total size of all nonkey columns is limited only by the size of the columns specified in the INCLUDE clause; for example, varchar(max) columns are limited to 2 GB. The columns in the INCLUDE clause can be of all data types, except text, ntext, and image.

**Q. On which basis we will create computed indexes?**

**Ans:**

For composite indexes, take into consideration the order of the columns in the index definition. Columns that will be used in comparison expressions in the WHERE clause (such as WHERE FirstName = ‘Charlie’) should be listed first. Subsequent columns should be listed based on the uniqueness of their values, with the most unique listed first.

**Q. How to design indexes for a database? Or**

**What are the index creation best practices?**

**Ans:**

* Understand the characteristics of the database (OLTP / OLAP)
* Understand the characteristics of the most frequently used queries.
* Understand the characteristics of the columns used in the queries.
* Choose the right index at the right place. For example, creating a clustered index on an existing large table would benefit from the ONLINE index option.
* Determine the optimal storage location for the index. A nonclustered index can be stored in the same filegroup as the underlying table, or on a different filegroup. If those filegroups are in different physical drives it will improve the performance.
* **Database Considerations:**
* Avoid over indexing
* Use many indexes to improve query performance on tables with low update requirements, but large volumes of data.
* Indexing small tables may not be optimal
* Indexes on views can provide significant performance gains when the view contains aggregations and/or table joins. The view does not have to be explicitly referenced in the query for the query optimizer to use it
* **Column considerations:**
* Keep the length of the index key short for clustered indexes. Additionally, clustered indexes benefit from being created on unique or nonnull columns.
* Columns that are of the ntext, text, image, varchar(max), nvarchar(max), and varbinary(max) data types cannot be specified as index key columns. However, varchar(max), nvarchar(max), varbinary(max), and xml data types can participate in a nonclustered index as nonkey index columns
* Columns that are of the ntext, text, image, varchar(max), nvarchar(max), and varbinary(max) data types cannot be specified as index key columns. However, varchar(max), nvarchar(max), varbinary(max), and xml data types can participate in a nonclustered index as nonkey index columns
* Consider using filtered indexes on columns that have well-defined subsets
* Consider the order of the columns if the index will contain multiple columns. The column that is used in the WHERE clause in an equal to (=), greater than (>), less than (<), or BETWEEN search condition, or participates in a join, should be placed first.
* **Index Characteristics:** Determine the right index depends on the business need
* Clustered versus nonclustered
* Unique versus nonunique
* Single column versus multicolumn
* Ascending or descending order on the columns in the index
* Full-table versus filtered for nonclustered indexes
* Determine the **Fill Factor**

**Q. Can we create index on table variables?**

**Yes, Implicitly**

**Create Index on Table Variable**

Creating an index on a table variable can be done implicitly within the declaration of the table variable by defining a primary key and creating unique constraints. The primary key will represent a clustered index, while the unique constraint a non clustered index.

DECLARE @Users TABLE

(

UserID INT PRIMARY KEY,

UserName varchar(50),

UNIQUE (UserName)

)

The drawback is that the indexes (or constraints) need to be unique. One potential way to circumvent this however, is to create a composite unique constraint:

DECLARE @Users TABLE

(

UserID INT PRIMARY KEY,

UserName varchar(50),

FirstName varchar(50),

UNIQUE (UserName,UserID)

)

You can also create the equivalent of a clustered index. To do so, just add the clustered reserved word.

DECLARE @Users TABLE

(

UserID INT PRIMARY KEY,

UserName varchar(50),

FirstName varchar(50),

UNIQUE CLUSTERED (UserName,UserID)

)

**Q. What is the Heap Table?**

**Ans:**

HEAP

* Data is not stored in any particular order
* Specific data cannot be retrieved quickly, unless there are also non-clustered indexes
* Data pages are not linked, so sequential access needs to refer back to the index allocation map (IAM) pages
* Since there is no clustered index, additional time is not needed to maintain the index
* Since there is no clustered index, there is not the need for additional space to store the clustered index tree
* These tables have a index\_id value of 0 in the sys.indexes catalog view

**Q. How to get the index usage information?**

**Q. Query to retrieve the indexes that are being used?**

**Ans:**

**sys.dm\_db\_index\_usage\_stats – bind with sysobjects**

This view gives you information about overall access methods to your indexes. There are several columns that are returned from this DMV, but here are some helpful columns about index usage:

* user\_seeks – number of index seeks
* user\_scans- number of index scans
* user\_lookups – number of index lookups
* user\_updates – number of insert, update or delete operations

**sys.dm\_db\_index\_operational\_stats – bind with sysindexes**

This view gives you information about insert, update and delete operations that occur on a particular index. In addition, this view also offers data about locking, latching and access methods. There are several columns that are returned from this view, but these are some of the more interesting columns:

* leaf\_insert\_count – total count of leaf level inserts
* leaf\_delete\_count – total count of leaf level inserts
* leaf\_update\_count – total count of leaf level updates

SELECT DB\_NAME(DATABASE\_ID) AS DATABASENAME,  
OBJECT\_NAME(B.OBJECT\_ID) AS TABLENAME,  
INDEX\_NAME = (SELECT NAME  
FROM SYS.INDEXES A  
WHERE A.OBJECT\_ID = B.OBJECT\_ID  
AND A.INDEX\_ID = B.INDEX\_ID),  
USER\_SEEKS,  
USER\_SCANS,  
USER\_LOOKUPS,  
USER\_UPDATES  
FROM SYS.DM\_DB\_INDEX\_USAGE\_STATS B  
INNER JOIN SYS.OBJECTS C  
ON B.OBJECT\_ID = C.OBJECT\_ID  
WHERE DATABASE\_ID = DB\_ID(DB\_NAME())  
AND C.TYPE <> ‘S’

**Q. What is fill factor? How to choose the fill factor while creating an index?**

**Ans:**

The Fill Factor specifies the % of fullness of the leaf level pages of an index. When an index is created or rebuilt the leaf level pages are written to the level where the pages are filled up to the fill factor value and the remainder of the page is left blank for future usage. This is the case when a value other than 0 or 100 is specified. For example, if a fill factor value of 70 is chosen, the index pages are all written with the pages being 70 % full, leaving 30 % of space for future usage.

**Q. When to choose High or Low Fillfactor Value?**

**Ans:**

You might choose a high fill factor value when there is very little or no change in the underlying table’s data, such as a decision support system where data modification is not frequent, but on a regular and scheduled basis. Such a fill factor value would be better, since it creates an index smaller in size and hence queries can retrieve the required data with less disk I/O operations since it has to read less pages.

On the other hand if you have an index that is constantly changing you would want to have a lower value to keep some free space available for new index entries. Otherwise SQL Server would have to constantly do page splits to fit the new values into the index pages.

**Q. What methods are available for removing fragmentation of any kind on an index in SQL Server?**

**Ans:**

**SQL Server 2000:**

* + DBCC INDEXDEFRAG
  + DBCC DBREINDEX
  + CREATE INDEX…DROP EXISTING (cluster)
  + DROP INDEX; CREATE INDEX

**SQL Server 2005 and above:** The same processes as SQL Server 2000, only different syntax

* + ALTER INDEX…REORGANIZE
  + ALTER INDEX…REBUILD
  + CREATE INDEX…DROP EXISTING (cluster)
  + DROP INDEX; CREATE INDEX

**Q. What page verification options are available in SQL Server and how do they work?**

**Ans:**

**SQL Server 2000:** Only “Torn Page Detection” is available in SQL Server 2000.

**SQL Server 2005:** Both “Torn Page Detection” and “Checksum” page verification options exist in SQL Server 2005. Page verification checks help to discover damaged database pages caused by disk I/O path errors. Disk I/O path errors can be the cause of database corruption problems and are generally caused by power failures or disk hardware failures that occur at the time the page is being written to disk.

**TORN PAGE DETECTION:**

Works by saving a specific bit for each 512-byte sector in the 8-kilobyte (KB) database page and is stored in the database page header when the page is written to disk. When the page is read from disk, the torn bits stored in the page header are compared to the actual page sector information. Unmatched values indicate that only part of the page was written to disk. In this situation, error message 824 (indicating a torn page error) is reported to both the SQL Server error log and the Windows event log. Torn pages are typically detected by database recovery if it is truly an incomplete write of a page. However, other I/O path failures can cause a torn page at any time.

**CHECKSUM:**

Works by calculating a checksum over the contents of the whole page and stores the value in the page header when a page is written to disk. When the page is read from disk, the checksum is recomputed and compared to the checksum value stored in the page header. If the values do not match, error message 824 (indicating a checksum failure) is reported to both the SQL Server error log and the Windows event log. A checksum failure indicates an I/O path problem. To determine the root cause requires investigation of the hardware, firmware drivers, BIOS, filter drivers (such as virus software), and other I/O path components.

**Q. What are the primary differences between an index reorganization and an index rebuild?**

**Ans:**

* Reorganization is an “online” operation by default; a rebuild is an “offline” operation by default
* Reorganization only affects the leaf level of an index
* Reorganization swaps data pages in-place by using only the pages already allocated to the index; a rebuild uses new pages/allocations
* Reorganization is always a fully-logged operation; a rebuild can be a minimally-logged operation
* Reorganization can be stopped mid-process and all completed work is retained; a rebuild is transactional and must be completed in entirety to keep changes

**Q. During an index reorganization operation, if the index spans multiple files, will pages be allowed to migrate between files**

**Ans:**

No – pages will not migrate between files during an index reorganization.

**Q. How to make forcefully use an index in a query? Or What table hint needs to be specified to forcefully use an index in a query?**

**Ans:**

We can specify “Index” table hint in a query to forcefully use an index.

Ex: SELECT Emp\_ID, Name FROM Emp WITH(INDEX(NIX\_NAME\_EMP))

**Q. What are the index statistics?**

**Ans:**

Index statistics contain information about the distribution of index key values. By distribution, I mean the number of rows associated with each key value. SQL Server uses this information to determine what kind of execution plan to use when processing a query.

**Q. When are index statistics updated?**

**Ans:**

The AUTO\_UPDATE\_STATISTICS database setting controls when statistics are automatically updated. Once statistics have been created, SQL Server then determines when to update those statistics based on how out-of-date the statistics might be. SQL Server identifies out of date statistics based on the number of inserts, updates, and deletes that have occurred since the last time statistics were updated, and then recreates the statistics based on a threshold. The threshold is relative to the number of records in the table. (Enable the properties – “auto create statistics” and “auto update statistics” for OLTP)

**Q. Explain database options “Auto Update Statistics” and “Auto Update Statistics Asynchronous”?**

**Ans:**

**Auto Update Statistics:** If there is an incoming query but statistics are stale then sql server first update the statistics before building the execution plan.

**Auto Update Statistics Asynchronous:** If there is an incoming query but statistics are stale then sql servers uses the stale statistics, builds the execution plan and then update the statistics.

**Q. How to update statistics manually?**

**Ans:**

If you want to manually update statistics, you can use either sp\_updatestats or UPDATE STATISTICS <statistics name>

**Q. What are the various types of statistics available?**

**Ans:**

There are three different types of statistics available.

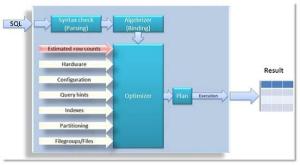
* Statistics created due to index creation.
* Statistics created by optimizer.
* User defined statistics created from “CREATE STATISTICS”

**Q. What are the various options to be considered while designing a new execution plan?**

**Ans:**

There are a number of factors that influence the decision in choosing the proper execution plan. One of the most important ones are cardinality estimations, the process of calculating the number of qualifying rows that are likely after filtering operations are applied. A query execution plan selected with inaccurate cardinality estimates can perform several orders of magnitude slower than one selected with accurate estimates. These cardinality estimations also influence plan design options, such as join-order and parallelism. Even memory allocation, the amount of memory that a query requests, is guided by cardinality estimations.

Other factors that influence the optimizer in choosing the execution plan are:

[](http://udayarumilli.com/wp-content/uploads/2014/12/index.jpg)

**Q. How histogram built?**

**Ans:**

As we all know that when we index a column, SQL Server does two things:

1. Sorts the values of the column in an internal data structure called “Index” This data structure contains the sorted value and a pointer to its respective row in the table.
2. Generates a histogram of values.

The histogram shows the distribution of values for the sorted column. For example:

if we have following values in an integer column 1,2,2,3,4,1,4,5,4 then a typical histogram will be similar to this

|  |  |
| --- | --- |
| **Value** | **Rows matched** |
| **1** | 2 |
| **2** | 2 |
| **3** | 1 |
| **4** | 3 |
| **5** | 1 |

So suppose if you are searching for all rows having column values between **3 and 5** then by looking at the above histogram table you can estimate that you will get total **1+3+1=5** rows.Â If your table contains just 9 rows in total then 5 rows means approximately 55% of total rows will match the criteria. Based on this calculation you may elect to directly scan the table instead of first reading the index, fetching the pointers of the matching rows and then read the actual rows. Although, it is a very simple example and there are other factors involve in deciding whether to use index or not but this example demonstrates the concept in very simple terms.

**Q. How to find out when statistics updated last time?**

**Ans:**

A simple logic is, run the query and observe the values for both “estimated rows” and “actual rows”, if they both are close to each other you need not worried about the statistics. If you find big difference between them then you need to think about updating statistics.

In general we can find out last statistics updated info from below query

select object\_name(object\_id) as table\_name

,name as stats\_name

,stats\_date(object\_id, stats\_id) as last\_update

from sys.stats

where objectproperty(object\_id, ‘IsUserTable’) = 1

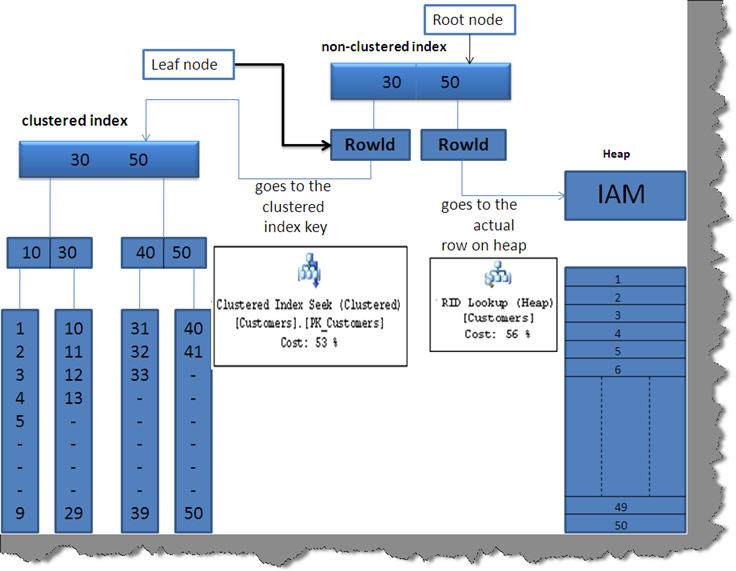
order by last\_update

**Q. What is RID Lookup \ Bookmark Lookup?**

**Ans:**

RID lookup will be seen when you use non-clustered indexes with join queries.

In order to understand the RID look up we need to first understand how non-clustered indexes work with clustered indexes and heap tables. Below is a simple figure which describes the working and their relationships.



Non-clustered indexes also use the B-tree structure fundamental to search data. In non-clustered indexes the leaf node is a ‘Rowid’ which points to different things depending on two scenarios:-

**Scenario 1:-** If the table which is having the primary key in the join has a clustered index on the join key then the leaf nodes i.e. ‘rowid’ will point to the index key of clustered index hence a clustered index seek happens

**Scenario 2 :-** if the table which is having the primary does not have a clustered index then the non-clustered index leaf node ‘rowid’ will point to actual row on the heap table. As the data is stored in a different heap table, it uses the lookup (i.e. RID lookup) to get to the actual row hence an Index seek (Non clustered) with RID lookup happens

***Performance Tuning – SQL Server Part – 3 – TempDB***

**Q. Tempdb is filling up drastically, what might be the activity that filling up the tempdb?**

**Ans:**

Usually, tempdb fills up when you are low on disk space, or when you have set an unreasonably low maximum size for database growth.

Many people think that tempdb is only used for #temp tables. When in fact, you can easily fill up tempdb without ever creating a single temp table. Some other scenarios that can cause tempdb to fill up:

* Any sorting that requires more memory than has been allocated to SQL Server will be forced to do its work in tempdb
* DBCC CheckDB(‘any database’) will perform its work in tempdb — on larger databases, this can consume quite a bit of space
* DBCC DBREINDEX or similar DBCC commands with ‘Sort in tempdb’ option set will also potentially fill up tempdb
* Large resultsets involving unions, order by / group by, cartesian joins, outer joins, cursors, temp tables, table variables, and hashing can often require help from tempdb
* Any transactions left uncommitted and not rolled back can leave objects orphaned in tempdb
* Use of an ODBC DSN with the option ‘create temporary stored procedures’ set can leave objects there for the life of the connection
* Table value functions that pull large data sets hold their data in tempdb
* If you are using Snapshot isolation on a database with a high number of transactions the snapshot data is stored in tempdb

**Q. How to analyze the TempDB contention**

**Ans:**

* Run the below command

USE tempdb

GO  
SELECT table\_name FROM information\_schema.tables

If you have a few tables, then you are most likely dealing with a large set issue. If you have a very large number of tables, then there is a runaway process or an unclosed process.

* Find the size of the tables. Shortlist the biggest tables and depends on the type of table we can find out the source.
* Make sure that disk space is not full and capacity settings for tempdb (low size for max growth)

Also with queries that fail to use an index on large tables can cause this from time to time. Do you have autoshrink turned on for tempdb

* You could check in on your record locking strategies for your executable sessions and/or records might not be getting released properly. Use the query optimizer tools maybe
* Check for index rebuilding jobs, data pulling jobs/triggers blocking sessions, long running transactions
* Use profiler / perfmon / dbcc commands to find the bottlenecks
* You can use Profiler to watch for events like database file auto grow and log file auto grow. If this is happening often, then you know that the space you’ve allocated to tempdb is not sufficient.
* You can also watch performance monitor’s counter for PhysicalDisk: CurrentDiskQueueLength on the drive where tempdb exists. If this number is consistently greater than 2, then there is likely a bottleneck in disk I/O.

**Q. How to fix the TempDB filling issue?**

**Ans:**

**Short-Term Fix:**

* Shrink Tempdb: We can shrink the db using ShrinkDatabase, DBCC ShrinkFile. If you can’t shrink the log, it might be due to an uncommitted transaction.
* See if you have any long-running transactions with the following command “dbcc opentran(tempdb)”
* Check the oldest transaction (if it returns any), and see who the SPID is “DBCC INPUTBUFFER(SPID)”
* Query will help you determine if you want to end this process with “KILL SPID”
* Restarting SQL Server will re-create tempdb from scratch, and it will return to its usually allocated size. In and of itself, this solution is only effective in the very short term; assumedly, the application and/or T-SQL code which caused tempdb to grow once will likely cause it to grow again.

**Long-Term Prevention:**

* Make sure that tempdb is set to autogrow — do \*NOT\* set a maximum size for tempdb. If the current drive is too full to allow autogrow events, then buy a bigger drive, or add files to tempdb on another device (using ALTER DATABASE) and allow those files to autogrow.
* For optimal performance, make sure that its initial size is adequate to handle a typical workload (autogrow events can cause performance to suffer as it allocates new extents).
* If possible, put tempdb on its own physical disk, array or disk subsystem
* To prevent tempdb log file growth, make sure tempdb is in simple recovery mode
* Try to make sure you have covering indexes for all large tables that are used in queries that can’t use a clustered index / index seek.
* In general, try to make your code as efficient as possible… avoid cursors, nested loops, and #temp tables if possible.
* Make sure all the transactions are having the corresponding Commit and Rollback

**Q. The most common argument I heard is table variables are in-memory structures (stored in memory not on tempdb) not like temporary tables. Is that true? How can you justify?**

**Ans:**

It’s actually a wrong assumption; both table variables and temp tables are stored in tempdb. A DMV “sys.dm\_db\_session\_space\_usage” can help us to make sure these objects use Temdb.

On a development machine restart the SQL server and select data from above DMV as below.

SELECT session\_id,

database\_id,

user\_objects\_alloc\_page\_count

FROM sys.dm\_db\_session\_space\_usage

WHERE session\_id > 50 ;

Initially user\_objects\_alloc\_page\_count is shown as 0.

Create a simple temporary table and insert test data and now check the same DMV, we can see the page\_count as 1.

Then create a table variable and insert a row and we can see the page\_count increased to 2.

**Q. What database objects are stored in tempdb?**

**Ans:**

There are three different types of objects stored in tempdb.

**Internal Objects:**

1. Intermediate runs for sort.
2. Intermediate results for hash join and hash aggregates.
3. XML variables or other large object (LOB) data type variables. (text, image, ntext, varchar(max), varbinary(max))
4. Queries that need a spool to store intermediate results.
5. Keyset cursors to store the keys.
6. Static cursors to store a query result.
7. Service Broker to store messages in transit.
8. INSTEAD OF triggers to store data for internal processing.
9. DBCC CHECK internally uses a query that may need to spool intermediate results.
10. Query notification and event notification use Service Broker.

**Note:**

Page allocations on internal objects and Updates to internal objects do not generate log records.

Does not appear in catalog views such as sys.all\_objects

**Version Store:**

1. Snapshot Isolation / Read Committed Snapshot Islotaion
2. Triggers (After Triggers). Instead of triggers doesn’t generate versions.
3. MARS (Multiple Active Result Sets)
4. Index Rebuilds

Note:

Inserts into version stores do not generate log records.

Does not appear in catalog views such as sys.all\_objects

**User Objects:**

1. User defined tables and indexes
2. Local and global temporary tables, bulk insert and BCP intermediate results
3. Index rebuilds with “SORT IN TEMPDB” option.

Note:

Most of the operations under this category are bulk logged.

Appear in catalog views such as sys.all\_objects.