SQL Server 2005 Beta II
Snapshot Isolation

Author: Kimberly L. Tripp, SQLskills.com

Summary: In many systems today, significant read activity is isolated from write activity in the form of a data warehouse or separated system. There are numerous advantages to this approach; read-intensive applications tend to want more index structures, data redundancies and even alternate views of data. Transaction processing systems want throughput; only the most minimal overhead to allow the best write throughput. The access patterns of readers and writers typically differs; readers are more prone to larger analysis types of queries and writers are more prone to singleton inserts, updates and deletes. When these activities are separated the administrator can focus on recovery strategies for a smaller more manageable transaction processing system; OLTP databases tend to be only a fraction of their data redundant Decision Support/Analysis Database cousins. Having said this; however, it is not always possible to clearly make this distinction. Once data is copied/transformed/archived to an analysis-oriented database it must be maintained and/or rebuilt periodically. Users certainly benefit from looking at a transactionally consistent version of the database; however that version is no longer current data, may take many hours to build and index and might not be what the user really needs. Enter snapshot isolation.

The primary focus of this paper will be to discuss when this isolation level is appropriate, what possible trade-offs exist and what are the best practices for use. Before reading this document you should consider reading these topics from the SQL Server 2005 Beta II documentation:

"Concurrency Problems"

Scripts from this Whitepaper:

The scripts are in the zip file named SQLServer2005SnapshotIsolation.zip. This file is here:
http://www.SQLskills.com/Resources/Whitepapers/SQLServer2005SnapshotIsolation.zip.

Table of Contents

SQL Server 2005 Beta II Snapshot Isolation **1**
Table of Contents i
Data Access Patterns and Usage **1**
Usage Scenarios 2
Application in Online Transaction Processing 2
Adhoc Reporting against Live Data 4
Adhoc reporting against a copy-managed database 7
Overnight reporting against live data 9
Transaction Processing System Workload 10
Datawarehousing System Workload 11
Migration to a common database technology 12
SQL Server and Oracle Differences in Snapshot 13
SQL Server and Oracle Similarities in Snapshot 16
Understanding Concurrency Control 17
Understanding Isolation 20
Isolation Levels Offered in SQL Server 2005 20
Isolation Level and Application Best Suited 21
Snapshot Isolation Considerations 22
Definitions, Terminology and Syntax **22**
Read Committed with snapshots (Statement-level Snapshot) 23
Snapshot Isolation (Transaction-level Snapshot) 24
Allowing Snapshot Isolation 24
Requesting Snapshot Transactions 26
Understanding the “Beginning” of a Transaction 26
Combining Read-Committed and Snapshot Isolation 27
Understanding Row Versioning 29
Row Versioning in Read Committed with snapshots 29
Row Versioning in Snapshot Isolation 29
DDL Statements within Snapshot Isolation 30
Development Best Practices **34**
Read Committed Snapshot 34
Snapshot Isolation 35
Minimizing Update Conflicts 39
Illustrating Optimistic Concurrency Behavior 39
Administrative Best Practices **49**
Database-level Settings 49
Upgrade Issues 49

Version Store Usage of TempDB 51
Sizing TempDB 52
Monitoring Version Store Activity 53
Function: dm_tran_active_snapshot_database_transactions (): 53
Function: dm_tran_transactions_snapshot(): 54
Performance Monitor Counters 56
For more information 58
Books Online Topics 58
Knowledge Base Articles of Interest 58
Additional Reading 58
Newsgroups of Interest 59

Data Access Patterns and Usage

Production databases are growing rapidly in size and data retention periods are increasing with changing business requirements as well as regulatory changes. Additionally, with drive capacity doubling every 12-18 months and storage costs falling, the amount of desired data to keep "online" is increasing. One solution is to separate analysis from transaction processing and while that may have many benefits for complex detailed analysis and business intelligence probing; it does not always work in terms of disk space and manageability. With the need for more data to be online with more active queries executing, the need for more current and real-time analysis contention for data exists.

In SQL Server 2000 contention can be minimized under Read Committed transaction isolation as the Select statement processing releases read locks after a resource is read. The default environment follows the standard SQL-92 definition in that ONLY committed data is read and uncommitted changes are not visible. However, while only committed data can be read, the standard does not guaranteed read consistency – even within the life of a statement. The resource lock (a shared lock) is released immediately after processing the row and that data row can be immediately (even while the read is still processing other rows) modified.

[*NOTE: If data movement is not likely (i.e. splitting is reduced through thorough proper index creation and maintenance) the chance of re-reading a row within a single statement drops so significantly it becomes hard to produce this anomaly.*]

In many situations this is the correct and performant choice. Only committed changes are visible and they are visible quickly with minimal resources locking. For example, if looking for the current total of sales – as an estimate from the currently processing system – only an estimate may be desired as the value will become "stale" only moments after being accessed (because transactions continue to be processed). In fact, in many environments an even less restrictive transaction isolation level called READ UNCOMMITTED this is often specified with a lock hint, by using either the WITH (NOLOCK) or WITH (READUNCOMMITTED) hints - these are synonyms. This environment allows for uncommitted data to be read; however, when the count of sales and/or total sales is only an estimate then seeing data which is "in progress" may be acceptable. When this is not acceptable then a change in isolation level – made by the programmer to ensure consistency through read repeatability of the data - must be used.

So where do you draw the line? Is it possible to return statement-level or transaction-level read consistent data while a system is actively processing? Can you write a long-running query in a production environment, ask for consistency and not block writers? In SQL Server 2005, the ability to offer this to your users can be done through an optional database-level setting which automatically changes the behavior of READ COMMITTED; this new behavior offers non-locking, non-blocking, statement-level read-consistency. In this whitepaper the traditional READ COMMITTED will be referred to as "Read Committed using locking"; and the optional new behavior as "Read Committed using snapshots".

For transaction-level consistency a new Isolation Level has been added: SNAPSHOT; changing to this Isolation Level will make transaction level consistency a controllable setting. Without any of these new options set, SQL Server 2005 databases' default behavior work as they do in previous releases; this default will continue to be desired in many systems where transaction processing throughput and performance are the highest goals. If a form of non-locking snapshot is desired (either statement level or transaction level) row versioning will be used to track row modifications. To enable this, data writers will pay the cost when an update is made, **even if there is no reader at the time**.

The version store retains version records until all active transactions commit (assuming that the Update/Delete statement has itselfalready committed). Or more accurately: the version store needs to retain specific version records until the transactions explicitly running under Snapshot Isolation or Read Committed with snapshots that started before the commit time of the transaction that made the change themselves commit or end. However, in both cases the version will still need to be made. While this cost of taking a version is minimal, choosing to implement this should not be taken without careful consideration and many best practices in place.

Usage Scenarios

This section explores how the SQL Server 2005 Snapshot Isolation Level and the new form of the Read Committed Isolation Level (Read Committed with snapshots) can help deliver improved performance, reduced latency and greater developer and database administrator productivity in your organization.

The following common business scenarios are discussed:

- Application in Online Transaction Processing
- Adhoc reporting against live data
- Adhoc reporting against a copy-managed database
- Overnight reporting against live data
- Migration to a common database technology

Application in Online Transaction Processing

At first glance the primary use of Snapshot technology might seem to be in read-intensive workloads such as data warehousing and operational reporting systems where there might be a concurrency impact caused by the table-level read locks of complex, long running queries (especially aggregations) against large tables that require a transactionally consistent view of the database which can effectively lock out transactions that need to update the data. However this is not the only application of Snapshot technology – the optional new behavior of the Read Committed isolation level, which works with a snapshot at the statement level, can significantly improve the throughput of mixed-workload systems while offering transactionally consistent data – for large joins and aggregations. Since the snapshot guarantees the consistency of the read – for only the statement – long running conflicts cannot occur. Additionally, in this environment application changes are not necessary; the change is made at the database options level.

When pessimistic locking (the way most database vendors traditionally implement the full ANSI standard for levels of transaction isolation) is used, applications typically exhibit blocking. Simultaneous data access requests from readers and writers within transactions request conflicting locks; this is entirely normal and provided the blocking is short lived, not a significant performance bottleneck. This can change on systems under stress, as any increase in the time taken to process a transaction (for example delays caused by over-utilized system resources such as disk I/O, RAM or CPU as well as delays caused by poorly written transaction such as those that hold locks across user interaction) can have a disproportional impact on blocking – the longer the transaction takes to execute, the longer locks are held and the greater the likelihood of blocking.

An example of this might be a car rental company which uses both an internal and web-based reservation application to book cars on behalf of its customers. Systems such as these have transactions that are contending for the same data (i.e. cars). The system will offer short-running queries that allow the customer service representative to check availability of cars in certain locations prior to booking them for the customer – this is an area where programming techniques such as disconnected datasets are often used to provide optimistic concurrency control, specifically:

1. The application queries for all available cars of a certain class, in a specific date range at a rental location. This query is likely to be a join of at least a few tables such as Car, Class, Reservation and Location. Additionally, this query will run under the Read Committed isolation level to ensure that only committed data is returned to the user.

2. The recordset or dataset obtained by the query will be "disconnected" from the database so as to remove any locks held on the data while the data is displayed in the caller's application. This is often called "batch optimistic" as it emulates the optimistic forms of database concurrency control. It is optimistic in that although the data is active; the likelihood for conflict should be low. The use of row-level timestamps enables the programmer to identify data change and manage conflicting updates with appropriate messages to the user interface.

3. The caller will select a specific car and the dataset will be edited to reflect the reservation.

4. The application will then reconnect and attempt to synchronize the change to the database, using the row-level SQL Server timestamp column to ensure that the data has not been changed by other callers while the data was disconnected.

5. The application then reports back to the caller to either report success (the reservation was taken) or to indicate a conflict (the car was taken by another caller) and to offer the chance to try to book another car.

Note that the above technique is not truly optimistic. In this design pattern a significant amount of contention can take place while the query in step 1 is running to find candidate cars. With SQL Server 2005 Read Committed with snapshots these requests are given a non-locking, non-blocking, transactionally consistent version of the data – while the query runs. With this isolation, the locking/blocking load on the server can be

reduced and the live data is not blocked for other customers who want to reserve cars. While this can improve the end-to-end performance for the transactions booking cars be eliminating lock waits, it does not necessarily improve the chances that a car viewed by the long running query will be available. However, this is an acceptable trade-off. The reservations occur faster and are not blocked by simultaneous requests for car rental data. This leads to increased throughput of transactions, especially under peak workloads, such as those caused by holiday bookings and business travel peak times.

Once the new Read Committed with snapshots behavior has been enabled by the database administrator at the database level, the programming logic that was used in steps 1-5 above can take immediate benefit of this new behavior without changing a line of code. In fact, once the database setting has been set, all queries will default to this form of statement-level read consistency.

Adhoc Reporting against Live Data

All companies are continually striving to reduce costs while expanding the capabilities of their information systems. One of the guiding targets for SQL Server 2005 is the elimination of the latency between data being captured within a database and it being available for use for reporting by the organization – this reduction in latency enables developers to build systems that provide data outside of the traditional batch reporting schedule.

Consider the scenario of a food retailer who is trying to balance the need to minimize the stock of fast moving consumer goods such as sandwiches, milk and other perishables which are held at each store; with the need to ensure that the shelves in the supermarket are stocked with items that their customers want to buy. Many of these kinds of items are very sensitive to the weather, for example barbeque items and ice cream sell more on sunny days; comfort foods sell more on rainy days.

Previous to the introduction of the new snapshot-based isolation levels the developers of the supermarket application might have avoided long blocks on the live data by using the Read Uncommitted Isolation Level, This can be difficult to use, especially when joining across multiple tables because the Read Uncommitted Isolation Level provides non-blocking access to a statement-level transactionally inconsistent view of the database, a view where the data related to a business transaction may have only partially arrived in the database.

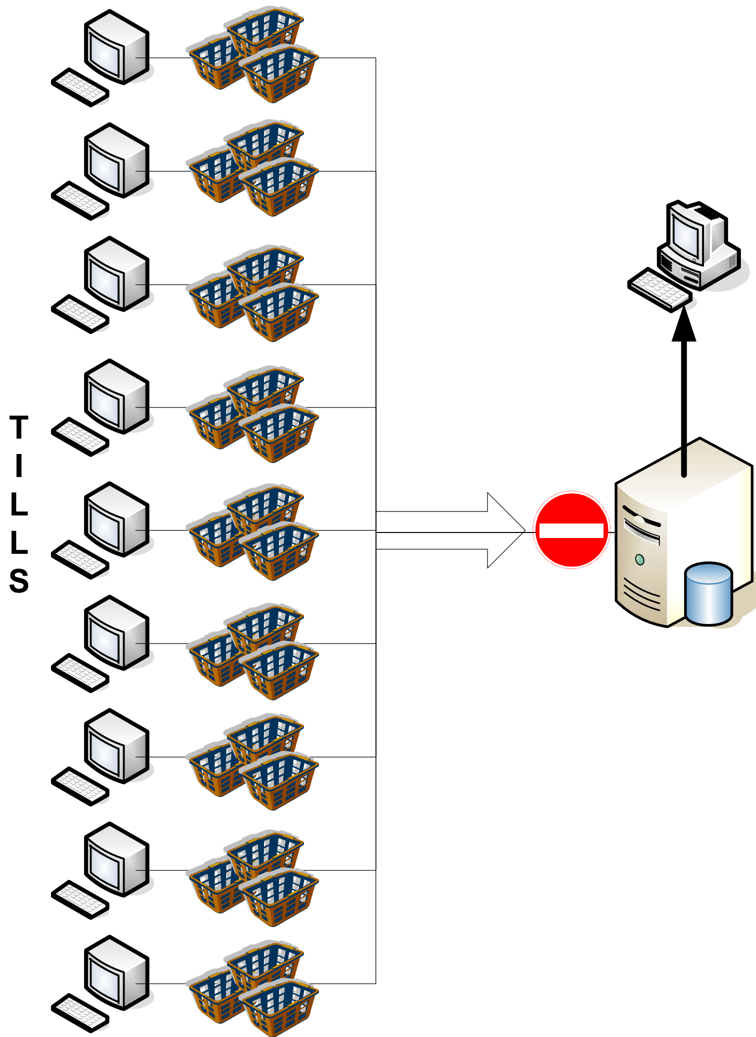


Figure 1: Tills blocked by a reporting user

Furthermore the practice of analyzing sales to look at the mix of other items sold together with the fastest selling items (also known as basket analysis) can be very data intensive and hence long-running, thus leading to an even greater chance of inconsistencies as data continues to arrive in the database.

In situations where a transactionally consistent view of the data is a necessity the system designers would typically design for these types of reports to run out of hours to avoid impacting the concurrency of the live system (where a long running, read-only report that was launched during peak usage could end up blocking all writers from updating the system, as depicted in Figure 1, above).

Having an IT infrastructure that only provides for pre-planned overnight reporting could hinder the ability of the Supermarket Manager to react to unexpected demands and review which products are at risk of selling out, and thus potentially miss an opportunity to order a second delivery to restock from the warehouse to meet demand, leading to loss of potential revenue, or even worse, loss of customers to competitors.

The new isolation levels provide applications with non-locking access to a transaction or statement level transactionally consistent view of the whole database, thus making the report writer's job much easier and also much more rewarding. In SQL Server 2005 the power of the database engine has been even more tightly integrated with the advanced aggregation and analytical capabilities of the Analysis Services component, which introduces the Universal Data Model and enables full analytical reporting without the need to extract and transform data into star schema. Snapshot Isolation technology has a major role to play in making data more accessible to this kind of application – being able to combine the power of the cross-selling reports with live data has the potential to change the way these business processes work.

- The new Read Committed with snapshots Isolation Level is best used for existing report systems (or systems purchased from third parties where the isolation level cannot be changed) as it is likely that no application change will be required to take advantage of non-locking reads, especially as the majority of these applications populate reports with the results of a single query. In this scenario the row version (or snapshot) need only be retained for the length of each query. (Note that in Beta 2 of SQL Server 2005 the versions will remain until the end of the transaction).

- The new Snapshot Isolation Level is well suited to more complex requirements such as running a series of reports that must run within the same transaction in order to all see the same transactionally consistent view of the data – this is more likely in complex financial reporting systems where it is not desirable that data changes are picked up while the report suite is running as it could easily cause anomalies in totals and checksums between reports. In this scenario the row version would be retained for the length of the transaction.

SQL Server 2005 makes it simple to enable these new isolation levels for a database. Once the new form of Read Committed behavior is configured, SQL Server automatically uses it without requiring any application or transactional code changes. To make use of the transaction-level snapshot isolation, you must configure the isolation level for the connection before any snapshot transactions can begin.

Once either of these capabilities is enabled, it is then safe to provide the supermarket manager with a series of parameterized reports that can be run when unexpected demand takes place in the store – without blocking the data coming in from the store's tills, thus helping the manager ensure that the needs of the store's customers are anticipated and met – leading to high customer satisfaction.

Enabling Snapshot Isolation makes additional demands on the database server. In the scenario discussed above it is assumed that the back office server used to collect data arriving from the supermarket's tills had enough spare capacity to support the occasional requirement to run adhoc reports against the live data. The use of snapshot isolation imposes added load on a server running update transactions, both for data writers and data readers. For data writers, their changes must be versioned. For data readers, their reads must traverse the version chain to obtain the version appropriate to the time of their transaction start.

The additional load can apply to TempDB (as TempDB is where SQL Server stores the version store - used to provide a transactionally consistent view of the changing data, especially for long running transactions) and so it is recommended that the DBA test this new technology on preproduction systems under simulated load prior to production deployment.

Note that simple measures such as providing more I/O bandwidth for TempDB, together with the scalability improvements made to TempDB in SQL Server 2005, may more than offset the impact of enabling the versioning-based isolation levels. However, if the

system is already heavily loaded with a mixed update and read workload then configurations discussed in other scenarios (see below) may be more appropriate. This can be especially true when transaction-wide Snapshot Isolation is required rather than statement-level Read Committed with snapshots.

Adhoc reporting against a copy-managed database

In systems with a high percentage of data changing, enabling the use of the new Snapshot Isolation Level may have a negative impact on overall performance as the overhead of creating and managing the previous versions of a row can slow down transactions, particularly if TempDB or the disk subsystem is already close to being a system bottleneck. In this situation the performance cost of enabling the new infrastructure may not worth the value of reporting against the real time data, especially as any reporting will likely add even more load to an already busy system.

This scenario is typical of reservation systems (such as airline and hotel reservation systems) as well as order entry systems including online systems such as web shopping sites. The performance of updates during peak periods of load is critical – a slow update can cause a consumer to give up their purchase and head to another site; conversely the customer service departments and demand forecasting staff need access to reports containing live data to help in their interaction with customers and to perform planning.

These conflicting requirements may best be served by creating a copy managed database – a near real time replica of the data that lags behind the live system, but that is "live enough" for reporting to take place. The goal of this replica is to offload the reporting users to another server (or even a set of servers) so that they do not add to the workload of the live system.

SQL Server 2005 provides two options for automating the maintenance of a replica database, both of which operate within the transaction logging mechanism and hence on committed data:

1. Database Mirroring (a.k.a. Synchronous Log-Shipping) – this technology is primarily designed to provide a hot-standby of the live system: data is sent to the replica during each transaction commit process – the commit does not complete until the data is in both the live and replica database logs. Performance on the live system is thus sensitive to the ability of the replica standby system to commit. For this reason Data Mirroring is less suited to offloading reporting workload as any spikes in reporting workload can directly impact live system performance. For systems that must sustain continuous high rates of update transactions database mirroring should be seen as more an availability feature rather than as enabling a secondary reporting database.

   Data Mirroring has the advantages that it is extremely easy to set up and to manage, and once established all data is transferred without the database administrator having to select specific tables, in fact as changes are made to the

live system they automatically occur on the replica. Furthermore the use of SQL Server 2005 Database Snapshots on the replica server can be used to provide transactionally consistent point in time reporting. However database snapshots must be created manually and it may be unrealistic to maintain a viewpoint for each report (assuming each report requires access to the most current data).

Other disadvantages include the fact that it is not possible to make changes solely to the replica such as filtering a subset of the data; adding reporting-only users with read-only privileges; and adding additional table indexes and indexed views designed to aid reporting performance. These changes can only be made to the live system, which may see degradation in update performance as a result.

2. Replication (specifically Transactional Replication) – this technology imposes only a light overhead on the live system, which can be mitigated by improving database log file I/O bandwidth. Committed transactions are read asynchronously from the database transaction log file and the data moved to the distribution database from where it can be fanned-out to multiple subscribers.

   This technology can be harder to manage yet is often well known by database

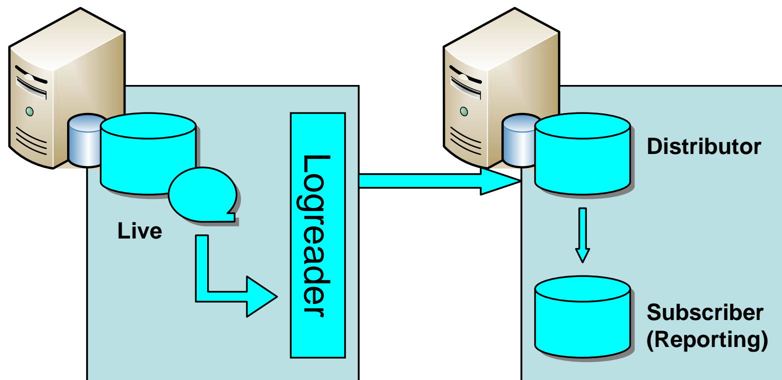


Figure 2: Replicating Data

administrators as it has been a core component of SQL Server over many years. The data that is replicated can be a subset of the live system (by table and both by row and by column) and has the advantage of allowing different users, indexes and views to be present in the subscriber (reporting) database.

Disadvantages are few – there is always a lag between a committed record arriving in each subscriber replica system; and an additional copy of data must be maintained in the distributor database until it is propagated to all subscribers.

SQL Server 2005 schema changes on replicated objects rarely require the sort of reworking (and no longer require use of replication-specific stored procedures to add/remove columns) that previous versions required. Previously, these commands restricted the use of replication from third party applications when the applications did not use replication commands to change schema across

releases. Figure 2 illustrates a typical transactional replication setup across two servers.

The main issue with the use of transactional replication in the past was that the distribution/subscription database link suffered from the same issues seen in the live system. When long running reports run in the subscriber database they block the replicated data from arriving in the subscriber system. This blocking can cause the replica to get increasingly out of synch and behind the live system. In turn, this frustrates the call center employees who are trying to help customers with recent purchases or reservations that have not yet arrived in their system. This problem is fully addressed by the new Snapshot Isolation Level and the new Read Committed with snapshots.

The subscriber database(s) can be set to use Snapshot Isolation and the reports (and read-only applications) that rely on that data can use either Snapshot Isolation (for a consistent view across a suite of reports/dialogs) or Read Committed with snapshots for individual reports. Neither of these applications will require shared locks, thus removing blocking caused by database readers and hence preventing the mostly-read database from getting significantly behind the live system, The incoming data will not be blocked behind long running read transactions and queries will execute against a transactionally consistent view of the database. Additionally, replication can maintain better transactional fidelity as it moves data around the system.

This is also a very scalable solution. As the reporting workload grows, it is possible to add a second (or more) subscriber databases on new servers so as to cope with the additional load without any further impact on the live system as data is fanned out from the distributor database rather than the live system. Now the customer service representative should be able to see the transactions made by the customer (in the live system) while accessing the replica to assist the customer as necessary.

Overnight reporting against live data

This scenario considers the classic data-processing model of an “Online Day” and an “Overnight Batch.” The Online Day matches a defined set of typical in-office “business hours” where data is entered into the system with a workload exclusively composed of short transactions. Overnight batch is where long-running reports move and report against the data that arrived during the day. This scenario is very typical in mainframe applications, with TP monitors running in the day, and batch jobs running at night.

The growth of customer-facing, internet-aware applications as well as increasing globalization of companies (with some offices coming online as others go offline) has meant that this model is less relevant to modern datacenters. However there are still some lessons that can be learned by looking at this old technology:

1. User-centric workloads tend to have peaks and troughs.
2. Reports tend to be run at specific times so as to allow comparison with reports run previously at that time.

3. The workload of most databases has peaks of updates (such as data loading) and peaks of reads (such as reporting).

Consider "Gadget.com" a fictitious internet-facing company that supports and sells personal audio technology; it has a datacenter in New York that services its US business as well as several smaller offices across 7 countries. Like most companies, its online systems have definite usage patterns. Here, the peak load coincides with US patterns with the arrival of staff in offices and the arrival of their primary customers at the web site:

Transaction Processing System Workload

Time (Eastern)	Business Events	Datacenter Events
08:00 AM	US Offices coming online	Any remaining reporting is halted
12:00 PM	All US offices online, European offices closing	Peak office load
06:00 PM	US offices start closing, Smaller Asian & Australian offices coming online	Peak online load
10:00 PM		Lowest office & online load Snapshot Isolation enabled, data extracts begin followed by main operational reporting suite
02:00 AM	European offices coming online	Snapshot Isolation disabled, some US-specific operational reporting continues

In the above scenario the datacenter can manage the state of Snapshot Isolation while the database is online; there is no need to restart the database to pick up different settings. By only activating Snapshot Isolation in a narrow window Gadget.com continued to offer service to online and global office users, but also ensured that long-running reports that require a transactionally-consistent view of the database across queries would not block those users. By disabling Snapshot Isolation during peak usage Gadget.com also ensured that maximum throughput is available to its primary users and customers.

Gadget.com also runs a complex data warehouse that is used to provide information about customer and stock trends and to run larger reports that look for other patterns in the data over time. This system is primarily read-only; however there are a limited number of users who need update access to the database so as to perform accounting-style journal adjustments, as well as stock adjustments following audits. This does not present a problem for Gadget.com as they have adapted their strategy for using Snapshot Isolation to the needs of this system.

Datawarehousing System Workload

Gadget.com runs their data warehouse system 24x7; they use Snapshot Isolation to provide their report consumers with high-performance access to transactionally consistent data.

Time (Eastern)	Business Events	Datacenter Events
08:00 AM	US Offices coming online	Peak online reporting load
12:00 PM	All US offices online, European offices closing	Peak adjustment load (but minor)
06:00 PM	US offices start closing, Smaller Asian & Australian offices coming online	Reporting still online
10:00 PM		Lowest reporting load Database placed into the bulk-logged recovery model. The reporting application queues incoming report requests, the data loads and then data transformations begin.
02:00 AM	European offices coming online	System placed into the full recovery model. A log backup begins, long-running and queued reports start and then finally, the ad-hoc workload begins.

Gadget.com decided to continually operate this database with Snapshot Isolation enabled. In addition they looked at the amount of data they had to take on into the system each day and decided to maximize the performance of the load by using a mix of the Full and Bulk-logged recovery models. The Full recovery model is used to protect the adhoc adjustments made to data by the warehouse administrators. The Bulk-logged recovery model is used to reduce the logging when loading data.

In the data loading case, Snapshot Isolation settings did not need to be disabled because row insertion does not generate a version chain entry (there is no older data to version). Running with 24x7 Snapshot Isolation allows Gadget.com to enjoy both unimpacted fast data loading while allowing adhoc data adjustments to continue without being impacted by long running reports that could otherwise block the data load process. The only operational adjustment made was to adjust the recovery models to reduce logging and improve the data load:

- Bulk-logged Recovery model during dataload (as the system is fully recoverable using the previous full backup, its associated logs and the incoming data extract files. The load is followed by a changing the recovery model to Full and then performing a log backup.

- Full Recovery model the rest of the time, allowing log backups to be taken so that accountant adjustments are not lost by hardware failure or media corruption.

The above scenarios illustrate how Snapshot Isolation can be deployed in a system with variable transaction workloads, both online transaction processing and data warehousing. Snapshot Isolation can be left active so that it's benefits can be realized without significant impact on any of the key activities of the systems underlines the utility of this technology.

Migration to a common database technology

In the commercial relational database management system world, prior to SQL Server 2005, there were two camps. The first were the systems that implemented a pessimistic concurrency based on locking schemes that enable support for the four ANSI-standard isolation levels as defined in the SQL-92 standard (ANSI X3.135-1992, American National Standard for Information Systems — Database Language — SQL, November, 1992) – these systems include Microsoft SQL Server, IBM DB2 (all of its many code bases/platforms and variants) as well as Sybase Adaptive Server. The second camp implemented a non-standard transaction isolation model with optimistic concurrency based on retaining a view of the data as of the start of the transaction – the only commercial system in this camp was Oracle.

This division has led to three types of software developer:

1. Develops on Oracle, ports to Microsoft SQL Server
2. Develops on Microsoft SQL Server, ports to Oracle
3. Develops and optimizes for both camps.

Generally only the largest software companies can afford to be "type 3" – companies such as SAP, Siebel and Peoplesoft. Most developers must pick between type 1 or type 2, their choice normally being predicated by the degree to which the datacenter Unix market matters to their sales.

With SQL Server 2005 and the introduction of optimistic concurrency control with Snapshot Isolation it is now much easier for a type 1 application vendor to make a direct port to SQL Server and extend their market beyond the confines of the Oracle/Unix platform. IT Departments who want to drive down the complexities associated with supporting multiple database platforms and to avoid costs such as:

- multiple database teams
- increased training costs
- reduced volume software licensing costs
- management time interacting with multiple vendors
- matching differing supplier service levels

SQL Server 2005 offers the opportunity for customers to eliminate these additional costs, without having to change application vendor or suffer a drop in optimal performance caused by the paradigm shift in transaction isolation model.

The implementation of optimistic concurrency differs widely between SQL Server 2005 and Oracle – the SQL Server implementation is designed to be more controllable by the database administrator (it can be enabled & disabled on command, as illustrated in the previous scenario) and also to be more manageable – there are a wealth of Windows System Monitor performance counters and virtual tables accessible through system functions which can help detect and decipher what is happening with the database.

SQL Server and Oracle Differences in Snapshot

Microsoft SQL Server 2005	Oracle
No table modifications required, the snapshot version store and the version chain of changed records is completely independent of your table definition – this is something the system manages on your behalf	Requires use of INITRANS >= 3 & MAXTRANS on CREATE/ALTER TABLE DDL to enable space for on-page transaction information before SERIALIZABLE can be used – you have to get it right before you start using the table or face a costly DDL change after your users start complaining about ORA-08177: "Can't serialize access for this transaction."
Version store is held in memory and TempDB. The dba must ensure that TempDB is optimized for increased i/o bandwidth based on the version store workload – TempDB database size must also be monitored (especially if the application has long running transactions), SQL Server has supported dba-friendly percentage and absolute database and log autogrow settings for many releases, but these are obviously constrained by the physical availability of disk space. This can lead to long version chains in SQL Server. The version store keeps a full copy of the data row which saves the expense of reconstructing the row when it is accessed by another transaction. The integrated SQL Server Agent event management and job scheduling sub-system can be programmed to react automatically to an out of physical space condition and corrective action (such as forcing rollback of any transaction contributing to version store space usage)	Can require complex configuration of ROLLBACK SEGMENTS (creation & on/offline status) & defining transaction level USE ROLLBACK SEGMENT statements to avoid ORA-01555: "Snapshot too old." caused by "long running" transactions overwriting their versioned pages in the rollback segment. Note: Oracle does not have a definition for "long running" transactions. In recent versions (starting with Oracle 9i) Oracle have introduced a technology similar to that used in SQL Server 2005 called "Automatic Undo Management Mode" – this new method is incompatible with the previous manual method and will require code changes if USE ROLLBACK SEGMENT statements have been used The rollback segment/undo tablespace only stores the 'changed' value of the row (thus saving space) at the expense of reconstructing the versioned row at

Microsoft SQL Server 2005	Oracle
	run-time. Oracle 9i users still experienced the "dreaded" ORA-01555[1] and in response Oracle 10i introduced the new "UNDO_RETENTION_PERIOD" initialization setting, used in conjunction with the RETENTION GUARANTEE property of undo tablespaces to allow the Oracle DBA to specify how long undo data is retained. This is not an automatic setting and changes are picked up by stopping/starting the Oracle instance – to tune this setting the Oracle DBA can monitor V$ROLLSTAT to track "wrapping" (the reuse of undo tablespace storage) and the application can detect and report either ORA-01555 (no retention guarantee); or out of space conditions.
TempDB can autogrow as a % of current size (to elastically reduce the number of autogrow attempts) or as an absolute value	ROLLBACK SEGMENTS don't support PCTINCREASE and hence don't "autogrow" so you must get the size right when they are created. If using the automatic mode then the Undo Tablespace behaviour is very similar to that of SQL Server 2005 TempDB
Row Based data versioning – smaller amounts of data are written to/read from the version store, row level versioning means true row level serialization of transacted data access	The INITTRANS setting determines how many changes can be tracked in any one data block – exceed this value and the next transaction using SERIALIZABLE access to access other rows on an updated data block by other transactions causes ORA-08177: "Can't serialize access for this transaction." This is why Oracle recommends the use of SERIALIZABLE for systems with few, short update transactions (where the chance of filling the INITTRANS area is low)
Snapshot Isolation & Read Committed with snapshots are enabled at the database level. Only	Data versioning is not optional; it is always enabled.

[1] From: http://www.oracle.com/technology/pub/articles/10gdba/week20_10gdba.html

Microsoft SQL Server 2005	Oracle
databases which require this option need to enable it and incur the overhead associated with it. You must enable it across all databases which will participate in a cross-database transactions using snapshot isolation.	
Extensive operational Performance Counters, especially the SQLServer:Transactions set of counters that allow the dba to monitor the state of the version store, including: • Free Space in TempDB • Size of Version Store • Rate of growth • Number of Conflicts • Longest running active transaction (excluding non-version consuming or generating transactions)	Oracle has chosen to implement a platform-independent approach – this has benefits to customers who are very familiar with the Oracle toolset but excludes customers from easily integrating Oracle performance counters with the wealth of systems management products and expertise that is available on the Windows Platform.
SQL Server 2005 implements Virtual tables which allow the dba to see whether or not snapshot transactions have occurred as well as the size of the version store and earliest record in the version store: • sys.dm_tran_active_snapshot_database_transactions() • sys.dm_tran_active_transactions • sys.dm_tran_current_snapshot() • sys.dm_tran_current_transaction() • sys.dm_tran_database_transactions • sys.dm_tran_locks • sys.dm_tran_session_transactions • sys.dm_tran_top_version_generators() • sys.dm_tran_transactions_snapshot() • sys.dm_tran_version_store() These virtual system tables & functions are also called Data Management Views and can be used to monitor & report on the state of active transactions and their version & lock usage.	Oracle implements Data Management Views (Virtual Tables and functions) – typically V$UNDOSTAT (a histogram like record of undo statistics); V$WAITSTAT (with the Undo class); and V$TRANSACTION for per-transaction undo space usage

As well as the differences outline above, which are designed to ease the job of the database administrator, there are also similarities that are designed to make the developers life easier when porting an application from Oracle to Microsoft SQL Server 2005.

SQL Server and Oracle Similarities in Snapshot

Microsoft SQL Server 2005	Oracle
SELECT ... WITH (UPDLOCK) Equivalent, performs conflict checks immediately	SELECT... FOR UPDATE Lock a record within a transaction to prevent conflicts
READ COMMITTED with snapshots	READ COMMITTED
SNAPSHOT	SERIALIZABLE
SNAPSHOT	READ ONLY
READ UNCOMMITTED (access to uncommitted data)	No Equivalent
READ COMMITTED with locking	No Equivalent
REPEATABLE READ	No Equivalent
SERIALIZABLE	No Equivalent – the lack of read locking can cause design challenges for the developer, as outlined in **Oracle9*i* Application Developer's Guide - Fundamentals Release 2 (9.2)** Part Number A96590-01 : "Because Oracle does not use read locks, even in SERIALIZABLE transactions, data read by one transaction can be overwritten by another. Transactions that perform database consistency checks at the application level should not assume that the data they read will not change during the execution of the transaction (even though such changes are not visible to the transaction). Database inconsistencies can result unless such application-level consistency checks are coded carefully, even when using SERIALIZABLE transactions."
Can use blocking in pessimistic isolation levels or must handle conflicts (data row updated outside of the transaction) & retry failed transactions. Row level versioning reduces chances of conflict.	Must handle conflicts (ORA-08177: data page updated outside of the transaction) & retry failed transactions.
The application can choose an appropriate concurrency model.	Application always sees potentially stale data unless using manual table locking or SELECT...FOR UPDATE as there is no choice between concurrency models.
Transact-SQL TRY/CATCH logic handles conflict errors but doesn't handle out of space issues with TempDB.	PL/SQL has error handling that enables ORA-08177 (conflict) error handling, but doesn't handle ORA-01555 (rollback segment space issue). With Undo Tablespaces a similar out

	of space issue can arise.

Based on these similarities, SQL Server 2005 makes the porting of applications built to run against databases that support optimistic concurrency significantly easier than in past releases. Additionally, SQL Server 2005 introduces a programming model which allows the choice between pessimistic and optimistic concurrency control – as well as numerous mechanisms for implementation. The database administrator's life is eased by having a simple, easily configured version store that is online, enabled at the database level, and the developer's task in porting code is simpler because of the close functional match between the Oracle and the SQL Server 2005 schemes (*although the SQL Server 2005 exhibits more granular consistency behavior in that it manages versions at the row-level rather than the data block level*).

Understanding Concurrency Control

As seen within the usage scenarios, there are two primary models used in controlling concurrency: pessimistic concurrency and optimistic concurrency.

Under a pessimistic concurrency control-based system locks are used to prevent users from modifying data in a way that affects other users. Once a lock has been applied, other users cannot perform actions that would conflict with the lock until the owner releases it. This level of control is used in environments where there is high contention for data and where the cost of protecting the data using locks is less than the cost of rolling back transactions if/when concurrency conflicts occur.

Conversely, under an optimistic concurrency control-based system, users do not lock data when they read it. Instead, when an update is performed the system checks to see if another user changed the data after it was read. If another user updated the data, an error is raised. Typically, the user receiving the error rolls back the transaction, resubmits (application/environment dependant) and/or starts over. This is called optimistic concurrency because it is mainly used in environments where there is low contention for data, and where the cost of occasionally rolling back a transaction outweighs the costs of locking data when read.

Note that updates performed under Read Committed with snapshots do not conflict and hence will never incur the cost of rollback.

Prior to SQL Server 2005, transactions are always controlled in a pessimistic manner – meaning all transactions acquire locks. While locking can be the best concurrency control choice for applications requiring data consistency and inter-transaction protection, it can cause writers to block readers. If a transaction changes a row, then another transaction cannot read the row until the writer commits. There are cases where waiting for the change to complete is the correct response; however there are cases where the previous transactionally consistent state of the row is sufficient.

Snapshot-based isolation levels allow the reader to get the previously committed value of the row at the cost of having to keep this version when the row is modified – even if no one is "currently" accessing the data. This means that all select, update and delete statements (but not inserts – unless re-inserting over a recently deleted record) may have to pay the cost of versioning with additional I/O into/from the versioning store.

You must decide to make this trade in improved concurrency at the cost of overhead (and therefore performance). It is important to state that while each query may cost more to execute (because of versioning) the end result may be that you are able to support more throughput because of reduced contention. For this reason it is important that snapshot-based isolation levels be enabled where contention was costing you throughput; if you use this as a solution to a performance problem which is not caused by contention then you might be solving the wrong problem and in fact, degrading your system throughput – this is akin to throwing hardware at a problem where the performance issue is caused by poor application design and locking conflicts.

In general, application programming is easier when the database automatically controls your view of the data through versioning-based isolation. In this environment, you worry less about deadlocks and blocking and pay a slight additional cost in administrative management overhead and performance. In many cases, paying a cost in administrative overhead and in providing more disk throughput for TempDB can be an easier choice; this is often known as "killing it with iron" and has the benefit that programmers do not have to worry about complex programming logic. If all snapshot based queries are solely for read consistency and not the basis for later modifications; no application retry logic is necessary. However, you may end up with conflicts in transactions which use the Snapshot isolation level and later perform updates; if the version is "stale" then it is likely you will need to use transaction retry logic for updates.

In situations where blocking is being used to control access to resources (such as queues implemented in tables) then if you have enabled Read Committed with snapshots then you must use the WITH (READCOMMITTEDLOCK) locking hint to get the expected "classic" behavior as the snapshot-based read will never block.

The programmer now has the option of using SQL Server 2005 conflict resolution in conjunction with application and/or Transact-SQL transaction error handling in place of previous timestamp management techniques. Additionally, when your workload consists of batch style updates where many rows are modified, snapshot isolation is not recommended as the chance of conflict can be significantly larger. In that case you should choose a lock based isolation level (READ COMMITTED, REPEATABLE READ, OR SERIALIZABLE), keep your transaction short, and carefully design your transactions to minimize resource conflicts so that you minimize deadlocks.

Note that in the Beta 2 Release of SQL Server 2005 indexes were required to avoid update conflicts when two transactions update different rows in the table.

(Beta 2 Only) This is because snapshot isolation introduces a further consideration for the database developer/designer in that data access as part of an update must be through an index otherwise an update conflict will be returned even if no actual conflict occurs as, without an index, SQL Server has no mechanism to determine the range of data changed by the update command.

In the simple case of two snapshot transactions that update data in a table with no indexes, then the first transaction to commit will cause the second transaction to receive an update conflict:

```
Msg 3960, Level 16, State 1, Line 1
Cannot use snapshot isolation to access table 'TableWithNoIndexes' in database
'Optimistic'. Snapshot transaction aborted due to update conflict. Retry
transaction.
```

This can also occur in the less obvious situation where un-indexed data is updated:

(Beta 2 Only)

Transaction 1	Transaction 2

-- Create & Populate a Table
create table UnIndexedData
(SurrogateKey int identity(1,1)
not null primary key clustered,
 NoIndex int not null)
go
declare @cnt int
set @cnt = 0
while @cnt < 1000
 begin
   insert UnIndexedData
(NoIndex) values(@cnt)
   set @cnt = @cnt + 1
 end
go	

	-- Begin a transaction set transaction isolation level snapshot go begin transaction select top 5 * from UnIndexedData
-- Begin a transaction set transaction isolation level snapshot go begin transaction select top 5 * from UnIndexedData	
	-- Update some unindexed data update UnIndexedData set NoIndex = NoIndex +1 where NoIndex <100
-- Update some unindexed data -- This will block on Transaction 2 -- even though they don't overlap update UnIndexedData set NoIndex = NoIndex + 1 where NoIndex > 950	
	-- Commit the update

(Beta 2 Only)

Transaction 1	Transaction 2
	`Commit Transaction`
`-- Transaction 1 will abort with update conflict (Beta 2 only)`	

Note that it would have been possible to avoid the above conflict in Beta 2 had the query included the (indexed) SurrogateKey column. This will not be a restriction in SQL Server 2005 Beta 3 onwards; furthermore the blocking behavior will only be seen when using Read Committed with Snapshots.

Understanding Isolation

Because Isolation Level is completely controllable in SQL Server 2005, understanding the most appropriate isolation for your application is important – for both concurrency and performance while still maintaining the appropriate level of accuracy. The concept of Isolation Level is not new – in fact, details regarding the ANSI specifications for Isolation can be found on: www.ansi.org and the current specification to review is ANSI INCITS 135-1992 (R1998). However, the standard is intended to be implementation-independent and doing so is somewhat ambiguous in what the exact trade-offs are in consistency and performance as well as how to achieve these goals and standards. As a result numerous papers have been written to further clarify the standards: *Generalized Isolation Level Definitions* or even critique them – as in *The Critique of ANSI Isolation Levels*. Based on the philosophies these works represent and the ambiguity in the ANSI standard, SQL Server 2005 offers many of the possible combinations typically desired.

Isolation Levels Offered in SQL Server 2005

	Possible Phenomena (as defined in ANSI SQL Standard)			**Concurrency Control**
Isolation level	Dirty read	Non-repeatable read	Phantom	
Read Uncommitted	Yes	Yes	Yes	(None)
Read Committed with locks	No	Yes	Yes	Pessimistic
Read Committed with snapshots	No	Yes	Yes	Optimistic

Repeatable Read	No	No	Yes	Pessimistic
Snapshot	No	No	No	Optimistic
Serializable	No	No	No	Pessimistic

The application usage for each of the above varies based on the desired level of "correctness" and the trade-off chosen in performance and administrative overhead.

Isolation Level and Application Best Suited

Isolation level	Best Suited for an Application when:
Read Uncommitted	The application does not require absolute accuracy of data (and could get a larger/smaller number than the final value) and wants performance of OLTP operations above all else. No version store, no locks acquired, no locks are honored when reading data. Data accuracy of queries in this isolation may see uncommitted changes.
Read Committed (with locks)	The application does not require point-in-time consistency for long running aggregations or long-running queries yet wants data values which are read to be only transactionally consistent. The application does not want the overhead of the version store when reading data with the trade-off of potential incorrectness for long running queries because of non-repeatable reads. This isolation level is ideally suited to transactions that rely upon the blocking behavior of locks to implement queuing applications and/or other ordered access to data.
Read Committed (with snapshots)	The application requires absolute point-in-time consistency for long running aggregations and/or long-running queries. All data values must be transactionally consistent at the point in time where the query begins. The database administrator chooses the overhead of the version store for the application for the benefit of increased throughput due to reduced lock contention. Additionally, the application wants transactional consistency for large queries not transactions.
Repeatable Read	The application requires absolute accuracy for long running multi-statement transactions and must hold all requested data from other modifications until the transaction completes. The application requires consistency for all data which is read repeatedly within this transaction and requires that no other modifications are allowed – this can impact concurrency in a multi-user system if other transactions are attempting to update data that has been locked by the reader. This is best when the application is relying on consistent data and plans to modify it later within the same transaction.
Snapshot	The application requires absolute accuracy for long running multi-statement transactions but does not plan to modify the data. The

	application requires consistency for all data which is read repeatedly within this transaction but plans to only read data. Read Locks are not necessary to prevent modifications by other transactions as the changes will not be seen until after the data modification transactions commit or rollback and the snapshot transaction completes. Data can be modified within this transaction level at the risk of conflicts with transactions that have updated the same data after the snapshot transaction started. This conflict must be handled by each updating transaction. A system with multiple readers and a single writer (such as the replicated reporting system in the scenario section above) will not encounter conflicts.
Serializable	The application requires absolute accuracy for long running multi-statement transactions and must hold all requested data from other modifications until the transaction completes. Additionally, the transactions are requesting sets of data and not just singleton rows. Each of the sets must produce the same output at each request within the transaction and with modifications expected no other users can modify not only the data which has been read but must prevent new rows from entering the set. This is best when the application is relying on consistent data, plans to modify it later within the same transaction, requires absolute accuracy and data consistency – even at the end of the transaction (within the active data).

Snapshot Isolation Considerations

While the change to Read Committed with snapshots does not require application changes (unless the application is reliant on the underlying locking behavior); it does require administration changes – the option must be activated per database. Enabling a database to allow snapshot isolation requires both administrative planning and possibly application planning. In both cases, the snapshot option is enabled at the database level and in all cases row versioning data is stored within memory (for short lived versions) and TempDB.

Note that as with other database-level settings both snapshot isolation settings can be made on the Model system database; these settings are then propagated when databases are created – Model acts as a template that is applied at create time – this is useful if you wish to create a set of standard database settings at it saves you from the administration task of connecting & updating each database in turn.

Definitions, Terminology and Syntax

To implement snapshot isolation in SQL Server 2005 you must be familiar with a few new concepts, terms and syntax. In previous releases Isolation Level was controlled solely by a session setting (SET TRANSACTION ISOLATION LEVEL, or the equivalent settings on the ADO or ADO.NET call) or by a query hint (FROM *tablename* WITH (isolation hint)).

In SQL Server 2005, one of the two supported database options must already be set (and not pending) in order to use the snapshot isolation. If snapshot isolation is

requested and the database is not yet ready to handle snapshots (i.e. still pending) then the statement requesting snapshot will fail. It is important to make changes at the appropriate time when changing back and forth – as well as understand the state of the database and client requests at the time of the change.

In order to use row versioning, you must first determine which isolation level is required for your application. SQL Server 2005 supports two types of snapshot isolation: statement-level snapshot and transaction-level snapshot.

Read Committed with snapshots (Statement-level Snapshot)

When set, statement-level snapshot guarantees that each statement under read-committed isolation sees only committed changes which occurred before the start of the statement. Each new statement within the transaction picks up the most recent committed changes. The version "refresh" occurs at the beginning of each SELECT statement. In other words, this version of read committed is semantically similar in that only committed changes are visible but the timing of when those changes committed differs. Each statement sees the changes that were committed before the statement began instead of when the resource is read. In other words, this is solely a new flavor of read committed that is non-locking, non-blocking and creates a solid point in time for which the data is accurate – accurate as of the start of the statement.

Statement-level snapshot is allowed by turning on the READ_COMMITTED_SNAPSHOT database option. Once turned on no other application changes are necessary.

Syntax:

```
      ALTER DATABASE <databasename>
            SET READ_COMMITTED_SNAPSHOT ON
            WITH <termination>
```

Executing this statement requires single user session access to the database. Use the ALTER DATABASE WITH <termination> options to end other user sessions in the database and to rollback their incomplete transactions. Ideally, this change should be made off hours and it is likely that this will be a permanent change. To see if a database has this option set use the sys.databases system view.

Syntax:

```
SELECT sd.is_read_committed_snapshot_on
FROM sys.databases AS sd
WHERE sd.[name] = <databasename>
```

The value returned for is_read_committed_snapshot_on will return either true (1) or false (0). When Read_committed_snapshot option is turned ON, read operations under the read committed isolation level are based on snapshot scans and are executed in a

non-locking, mode. When Read_committed_snapshot is turned OFF, the scans under read committed isolation are executed in a short-term locking mode where locks are only held for the life of the read request.

Snapshot Isolation (Transaction-level Snapshot)

When set, transaction-level snapshot isolation guarantees – by default – that every statement within a snapshot isolation transaction sees only committed changes which occurred before the start of the transaction. Effectively, each statement in the transaction sees the same set of data; while the data is available for modifications outside of this transaction. The modifications are not prevented and this "snapshot" transaction is unaware of the changes. The version "refresh" occurs only in the beginning of each transaction as long as you run under transaction-level snapshot semantics. If you override the transaction with the lock-based READ COMMITTED hint then locks (and potentially blocking) will occur unless you also have the new Read Committed Isolation turned on. If the new Read Committed with snapshots Isolation is in effect you will use row-versioning to return data to the query – unless you override that with the new READCOMMITTEDLOCK lock hint.

Note that DDL (Data Definition) changes to the database catalog can have immediate impact on transactions running under Snapshot isolation.

To achieve transaction-level snapshots there are two changes required. First the database must allow it by turning on the ALLOW_SNAPSHOT_ISOLATION database option. Second, the application/user must explicitly request a snapshot transaction.

Allowing Snapshot Isolation

Administrators must set a database option to allow snapshot isolation. This database option may not take effect immediately; however, it can be changed while users are connected to the database. If users are currently processing transactions when the state change is made, all transactions must complete before snapshot transactions can occur (because row versions have not been maintained for those transactions currently executing).

If changing the state is taking a lot of time and transactions attempt a snapshot transaction while the database is still "pending" then they will receive an error. If there are long running transactions executing at the time of the change, then the change to a versioning state can take a long time to complete. The DBA can cancel the request and if cancelled, the versioning state is rolled back to the prior versioned (or non-versioned) state. To request snapshot isolation for the database, change the database state using ALTER DATABASE:

Syntax:

```
ALTER DATABASE <databasename>
      SET ALLOW_SNAPSHOT_ISOLATION ON
```

To see if the option has taken effect you can check the sys.databases system view. There are two columns which may be of interest: snapshot_isolation_state and sd.snapshot_isolation_state_desc. The snapshot_isolation_state returns a tinyint value between 0 and 3:

0 = Snapshot-Isolation is Off

1 = Snapshot-Isolation is On

2 = Snapshot-Isolation state is in transition to Off state

3 = Snapshot-Isolation state is in transition to On state

The snapshot_isolation_state_desc returns an nvarchar(60) which is a character description of this pending state:

OFF = Snapshot-Isolation is Off

ON = Snapshot-Isolation is On

IN_TRANSITION_TO_OFF = Snapshot-Isolation state is in transition to Off state

IN_TRANSITION_TO_ON = Snapshot-Isolation state is in transition to On state

Syntax:

```
SELECT sd.snapshot_isolation_state
    , sd.snapshot_isolation_state_desc
FROM sys.databases AS sd
WHERE sd.[name] = <databasename>
```

Snapshot Isolation state	Description
OFF	Snapshot isolation state is disabled in the database. In other words, transactions with Snapshot based Isolations levels are not allowed. Database versioning state is initially set to OFF during restart recovery (a new SQL Server 2005 feature is that the database is made available after the REDO phase of recovery). If versioning is enabled, then after recovery completes the versioning state is set to ON.
PENDING_ON	In the process of enabling snapshot isolation state. It waits for the completion of all update transactions which are active when ALTER DATABASE command was issued. New update transactions in this database start paying the cost of versioning by generating record versions. Transactions under snapshot isolation can not start.
ON	Snapshot isolation state is enabled. New snapshot transactions can start in this database. Existing snapshot transactions (in another snapshot enabled database) which start before versioning state is turned ON can not do a snapshot scan in this database, because the snapshot those transactions are interested in

	cannot be properly generated by the update transactions.
PENDING_OFF	In the process of disabling snapshot isolation state. Unable to start new snapshot transactions. Update transactions still pay the cost of versioning in this database. Existing snapshot transactions can still do snapshot scans. PENDING_OFF does not become OFF until all existing transactions finish.

If the database option is taking a long time to move out of the pending state you can use the following function to return a virtual table showing the transactions that are active and blocking the database state change.

Syntax:

```
SELECT stx.spid
     , atx.[name]
     , stx.transaction_sequence_num
     , stx.first_snapshot_sequence_num
     , stx.commit_sequence_num
FROM sys.dm_tran_active_transactions AS atx
INNER JOIN sys.dm_tran_active_snapshot_database_transactions() AS stx
ON atx.tran_id = stx.transaction_id
```

Requesting Snapshot Transactions

As mentioned above, once the database has been enabled for Snapshot Isolation developers and users must then request that their transactions run in this snapshot mode. This must be done before beginning a transaction, either by a client side directive on the ADO.NET transaction object or within their Transact-SQL query.

Syntax:

```
      SET TRANSACTION ISOLATION LEVEL SNAPSHOT
```

If users execute this session setting change before the database has completed this change, the user's transaction will fail with error 3959: *Transaction failed in database <databasename> because an ALTER DATABASE command which enables snapshot isolation is not finished yet. Wait until the command is finished.*

Understanding the "Beginning" of a Transaction

Versioning is performed for all updates when the database allows snapshot; however, the version which a transaction will use is based on the first statement which accesses

data – not the BEGIN TRAN that creates the transaction. However if the transaction isolation level is being set to use Snapshot Isolation this must be done before any transaction is created (with a BEGIN TRAN or ADO.NET equivalent), otherwise the following error message will result:

```
Msg 3951, Level 16, State 1, Line 7
Transaction failed in database 'Optimistic' because the statement used snapshot isolation
but the transaction did not start in snapshot isolation.
```

To resolve this error message relocate the isolation specification before the transaction is created.

Syntax:

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT
BEGIN TRAN
      SELECT getdate() -- (T1) transaction has not "officially begun"
      SELECT * FROM <tablename> -- (T2) transaction t has begun
      SELECT... -- will see all committed changes as of (t)
      SELECT... -- will see all committed changes as of (t)
COMMIT TRAN
```

Combining Read-Committed and Snapshot Isolation

Transaction	**Without Snapshot based Isolations levels** (locking)	**Read Committed with snapshots** (non-locking)	**Snapshot Isolation**	**Database set to support both Read Committed with snapshots and Snapshot Isolation**
BEGIN TRAN				
SELECT * FROM t1	Lock (Data is locked as it is accessed) Can see committed changes occurring while the	Committed version of the rows – committed before the statement began. Non-locking, non-	Committed version of the rows – committed before the transaction. Non-locking, non-blocking.	Committed version of the rows – committed before the transaction began. Non-locking, non-

	statement is executing. Locks released after the resource is read.	blocking.		blocking.
SELECT * FROM t1 WITH (NOLOCK) OR (READUNCOMMITTED)	Uncommitted data can be accessed.	Uncommitted data can be accessed.	Uncommitted data can be accessed.	Uncommitted data can be accessed.
SELECT * FROM t1 WITH (READCOMMITTED)	Lock (Data is locked as it is accessed. Can see committed changes occurring while the statement is executing. Locks released after the resource is read.	Committed version of the rows – committed before the statement began. Non-locking, non-blocking.	Lock (Data is locked as it is accessed. Can see committed changes occurring while the statement is executing. Locks released after the resource is read.	Committed version of the rows – committed before the statement began. Non-locking, non-blocking.
SELECT * FROM t1 WITH (REPEATABLEREAD)	Data accessed is locked until the end of the transaction. Data cannot be modified by other transactions.	Data accessed is locked until the end of the transaction. Data cannot be modified by other transactions.	Data accessed is locked until the end of the transaction. Data cannot be modified by other transactions.	Data accessed is locked until the end of the transaction. Data cannot be modified by other transactions.
SELECT * FROM t1 WITH (SERIALIZABLE)	Datasets accessed are locked until the end of the transaction. Data cannot be modified or added to the set by other transactions.	Datasets accessed are locked until the end of the transaction. Data cannot be modified or added to the set by other transactions.	Datasets accessed are locked until the end of the transaction. Data cannot be modified or added to the set by other transactions.	Datasets accessed are locked until the end of the transaction. Data cannot be modified or added to the set by other transactions.
COMMIT TRAN				

Understanding Row Versioning

Versioning effectively starts with a copy-on-write mechanism which is invoked when a row is modified or deleted. This requires that while the transaction is running the old version of the row must be available for transactions which require an earlier transactionally consistent state. Snapshot transactions can effectively "view" the consistent version of the data from these previous row-versions. Row versions are stored within the version store which is housed within the TempDB database.

More specifically, when a record in a table or index is modified, the new record is stamped with the "sequence_number" of the transaction that is performing the modification. The old version of the record is copied to the version store, and the new record contains a pointer to the old record in the version store. If multiple long running transactions exist and multiple "versions" are required, records in the version store may contain pointers to even earlier versions of the row. All the earlier versions of a particular record are chained in a linked list, and in the case of long running snapshot transactions, the link will need to be traversed on each access so as to reach the transactionally consistent version of the row. Version records need to be kept in the version store only as long as there are snapshot queries that might be interested in them; this length of time depends primarily on whether or not the snapshot is statement-based or transaction-based.

Row Versioning in Read Committed with snapshots

For selects running under Read Committed with snapshots Isolation, the need of a particular row version ends when there are no queries running which reference the row. In other words the particular row version is not needed once ALL SELECTs that started before or during the transaction that modified the row have completed. Any SELECT which started after or during the row modification's transaction will require the row version to stay active in the version stored. However, once ALL of the SELECTs complete the row version can be removed. The version store under Read Committed with snapshots should not grow as large or be as difficult to predict as the size will be self-maintained by the frequent invalidation of a previous row version. However, this does depend solely upon the execution time of the statement.

Row Versioning in Snapshot Isolation

For queries running under snapshot isolation, the row versions need to be kept until the end of the transaction. Since a transaction may span multiple statements and a potentially longer period of time; the version store will need to potentially accommodate multiple versions of a row for a potentially longer period of time.

In the following figure, the current version of the record is generated by transaction T_3, and it is stored in the normal data page. The previous versions of the record, generated by transaction T_2 and transaction T_1 are stored in pages in the *version store* as there

are still transaction running under Snapshot isolation accessing the previous state of the data.

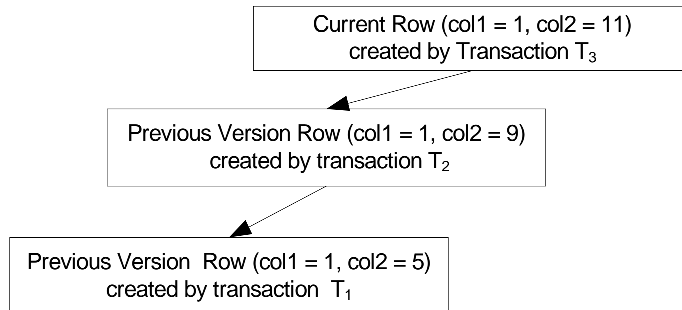


Figure 3: Versions of a Row

Using row-versioning will slow the update performance because of the extra work involved in keeping old versions; however, in the case when contention was costly you may see improved performance in the reduction in contention. Additionally, snapshot statements and transactions (a.k.a. version readers) have the extra cost of traversing the version link pointers. If many snapshot transactions exist and are long running transactions – a larger and faster TempDB may be necessary and performance may degrade if TempDB is not configured properly.

DDL Statements within Snapshot Isolation

Certain data definition language statements which modify the structure of an object will be disallowed as their changes cannot be seen through row versioning. Take for example a snapshot transaction which has read table1 and found 6 rows:

Syntax:

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT
BEGIN TRAN
      SELECT count(*) FROM <tablename> -- (Returns 6 rows)
…
```

A second transaction adds rows to this table – which under snapshot are NOT visible to this transaction. If this transaction were allowed to execute a CREATE INDEX statement then how would this work? Would the index be created on the snapshot view of the data or would it include all rows – as create index normally would? And if they chose the first – how would they reconcile the changes which are occurring simultaneously with the DDL? Even worse, what if multiple snapshots were creating additional indexes… Instead, CREATE INDEX is disallowed within a snapshot transaction. In fact, numerous DDL statements are disallowed as they violate the concept of "snapshot" and must be invoked against the actual base object – not a "version" of it.

Note that these restrictions do not apply to Read Committed with snapshots – DDL statements (or data access queries) will queue and not fail.

DDL Statements NOT Allowed within Snapshot Isolation

Certain statements are not allowed within a transaction running under Snapshot Isolation because of their disruptive potential on the snapshot copies of the data:

- CREATE INDEX
- CREATE XML INDEX
- ALTER INDEX
- ALTER TABLE
- DBCC DBREINDEX
- ALTER PARTITION FUNCTION
- ALTER PARTITION SCHEME
- DROP INDEX
- CLR DDL

An attempt to run one of these commands will result in a severity level 16 message such as:

```
Msg 3964, Level 16, State 1, Line 1
Transaction failed because the statement is not allowed in snapshot
transaction.
```

Other DDL statements not listed here, for example CREATE TABLE, are allowed as other transactions could not have been viewing prior versions of the data as it is a new object. This does not violate the rules listed above.

Other DDL Statements Changes after Snapshot Isolation Started

In most production databases the schema is relatively stable. However, changes may need to be made. If the system is highly available and user activity will occur concurrently with schema changes, programmers should be prepared for errors results from these changes. Since row versioning only exists for data rows – not metadata, one way of keeping the view consistent for the databases is to block all DDL in the server instance while a snapshot transaction is running. This would be too restrictive because a long running snapshot transaction would prevent a DBA from performing any DDL in a database for potentially a lengthy time. Instead, the DDL is supported while snapshot transactions are running yet the snapshot transaction may encounter failure if they try to access the objects changed since the start of the snapshot transaction. A timeline (time is from left to right) is shown below:

T_1 |--- Snapshot transaction ------------------------ Use object (fail) -----------|

T_2 |---DDL, change the object ---commit --|

The application programmer should put in retry logic for snapshot transactions to deal with this kind of error and administrators should attempt to minimize DDL changes during highly active times of day.

Snapshot Transaction Failure due to DDL Changes outside of the transaction

Not all DDL changes will cause a snapshot transaction to fail. Use the table below to see the likely impact of DDL changes taking place while snapshot transactions are running. It is important to realize that stable schemas will avoid transaction failure when using snapshot.

Note that this behavior is expected because the system catalogs are not covered by the snapshot scheme – if they were then for example the system would potentially have to maintain multiple copies of an index (and the accompanying data) – which would incur large overhead, perhaps inadvertently.

DDL Changes outside of the Snapshot Transaction	The Snapshot Transaction will Fail when:
• CREATE TABLE • ALTER TABLE * Including column change, type change, XML type binding change, constraint change, etc. • DROP TABLE	The snapshot transaction attempts to use the table AFTER the modification has occurred. A sample error message: `Msg 3961, Level 16, State 1, Line 1` Transaction failed in database 'Optimistic' because data needed by the statement has been modified by a DDL statement in another transaction since the start of this transaction. It may help to retry the transaction.
• CREATE STATISTICS • UPDATE STATISTICS • DROP STATISTICS	Allowed – these statements are not impacted by the Snapshot Isolation or Read Committed with snapshots settings
• CREATE INDEX • ALTER INDEX • DROP INDEX Includes all index types (clustered, non-clustered, XML Indexes, Fulltext Indexes, etc.)	The snapshot transaction attempts to use the table or view AFTER the modification has occurred to one of the table's associated index(es).
• CREATE TYPE	The snapshot transaction attempts

• DROP TYPE	to use the type AFTER the type modification has occurred.
• CREATE PROC/FUNCTION/VIEW • ALTER PROC/FUNCTION/VIEW • DROP PROC/FUNCTION/VIEW Includes both TSQL and CLR procedures and functions, including User-Defined Aggregate Functions.	The snapshot transaction attempts to use the procedure, function or view AFTER the modification has occurred.
• CREATE TRIGGER • ALTER TRIGGER • DROP TRIGGER Includes both TSQL and CLR triggers	The snapshot transaction attempts to use the table AFTER the change has occurred.
• sp_addextendedproc • sp_dropextendedproc Refers to extended stored procedures	The snapshot transaction attempts to use the procedure AFTER the modification has occurred.
• CREATE DEFAULT/RULE • DROP DEFAULT/RULE • sp_bindefault/sp_bindrule • sp_unbindefault/sp_unbindrule	The snapshot transaction attempts to use the table AFTER the change has occurred.
• CREATE SCHEMA • ALTER SCHEMA • DROP SCHEMA Includes XML Schema commands	The snapshot transaction attempts to use the table AFTER the change has occurred.
• CREATE ASSEMBLY • ALTER ASSEMBLY • DROP ASSEMBLY	The snapshot transaction attempts to use the assembly AFTER the assembly modification has occurred.
• CREATE PARTITION SCHEME/FUNCTION • ALTER PARTITION SCHEME/FUNCTION • DROP PARTITION SCHEME/FUNCTION Includes data spaces	The snapshot transaction attempts to use the partition function or scheme.
Start full text crawl on table	The snapshot transaction attempts to use the table AFTER the change has occurred.
Change Full Text Catalog	The snapshot transaction attempts DDL on the catalog AFTER the change has occurred.
CREATE EXTENDED TRIGGER ON DB FOR DDL	The snapshot transaction DDL which must check if there is extended trigger defined.

- CREATE SERVICE - ALTER SERVICE - DROP SERVICE	The snapshot transaction attempts to use the service AFTER the change has occurred.

Development Best Practices

It is expected that the system's DBA will perform due diligence in terms of system and application impact prior to activating either of the optimistic transaction isolation schemes – this leaves the developer with the responsibility of understanding how to exploit the new isolation level behavior to build better applications. The new capabilities of SQL Server 2005 present the developer with a new toolkit and understanding when and how to use these new tools is important if the resulting application is to perform as expected. The scenarios documented earlier illustrated some configurations where the new isolation levels would make sense, this section looks a bit more deeply at how the developer can make use of the new functionality.

Read Committed Snapshot

SQL Server 2005 provides a non-blocking Read Committed transaction isolation level based on statement level row versioning – this option must be enabled by the DBA and does not require any application level changes to exploit. If this option is enabled then transactions running under the default Read Committed isolation level (referred to as Read Committed with snapshots in this paper) do not acquire read locks as they read data, instead the version store is used to isolate the transaction from changes taking place as the read operation executes. This protection is at the statement level – if the application runs two select statements within the same read committed transaction the results can differ if data changes have been committed between the two statements, as they can with the traditional Read Committed with locks isolation level.

This behavior is a powerful tool for developers as it enables the application to consume more data without leading to increased blocking as readers compete with writers for data. It is common to see statements such as:

SELECT COUNT(Orders) FROM sales.dbo.orders WITH(NOLOCK)

In this query the NOLOCK (Read Uncommitted) hint is required to stop the read locks that would normally be taken by this query from blocking the taking of new orders – however it has the side-effect of returning uncommitted orders, possibly leading to an inaccurate total (as the uncommitted orders may never be committed to the database). Using the new behavior of Read Committed with snapshots the query can be run without the lock hint and an accurate view of committed data can be obtained without blocking online updaters. This is even more advantageous when performing a more complex query involving several joins as a stable view of the database is provided to the statement, avoiding anomalies in the results caused by late arrival of parent or child records, with data picked up by the NOLOCK hint.

Some applications, especially those which implement queues in tables, may require the old blocking behavior – in this case the locking hint READCOMMITTEDLOCK should be used.

```
SELECT TOP 1 NextOrderID FROM sales.dbo.orders
WITH(READCOMMITTEDLOCK) WHERE OrderStatus = N'Unprocessed'
```

In this query the SELECT statement will be blocked until update statements commit or rollback – so the order will not be picked up until it is committed.

As stated earlier in this section, the new Read Committed with snapshots behavior functions only at the statement level even when used within a multi-step transaction – this has another advantage in that the update conflicts that are possible with the Snapshot isolation level cannot happen with Read Committed with snapshots, and hence as a developer, there is no need to add additional logic to handle potential conflicts, allowing the new behavior to be activated without application change.

Read Committed should be the first choice when implementing a transaction that includes multiple select statements that do not require (as a group) a uniform, consistent view of the database.

Snapshot Isolation

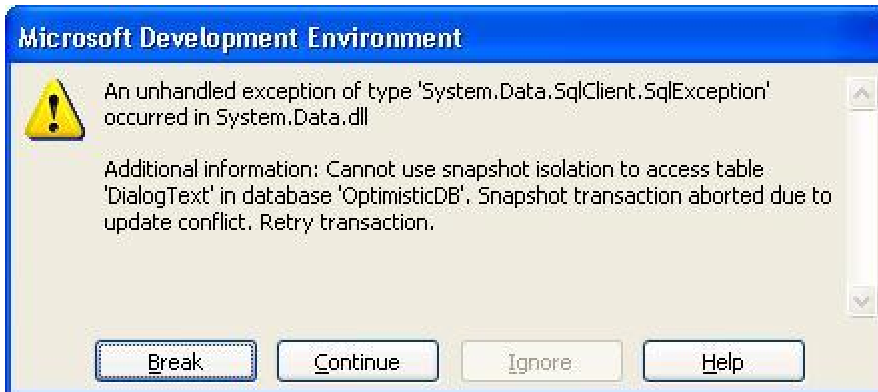
Sometimes having a uniform, consistent view of a database over multiple select statements within a transaction is important. Examples abound in read-only applications such as financial and human resources reporting where it is important that totals, sub-totals and check-sums be consistent despite the fact that they can be calculated over several selects, and sometimes over several minutes – if the system dictates a uniform view for a longer period then another tool, not discussed here, is the database snapshot. If the data changes whilst the transaction is running it is possible that spurious data quality issues will arise as minor discrepancies creep into reports.

Developers who are building read-only systems that run multiple data aggregation and sorting reports against a database that is constantly changing should consider Snapshot Isolation if a consistent view of the data is required and if their DBA has determined that database server capacity allows for the slight increase in database i/o – this class of application can be built either against the primary system or against a replica system built using SQL Server 2005 transactional replication. Snapshot Isolation will prevent the data writers (either other users or the replication distributor process) from being blocked by long-running read locks taken by the reporting application.

Outside of reports that require consistent data across queries in the same transaction there are other examples where the developer might want to use Snapshot Isolation – when filling data-driven dialog elements that are interrelated (again to avoid inconsistencies between pull-down lists, and other array controls); or in DBA-centered live system status dialogs where system statistics are being correlated from data stored across the database.

Snapshot Isolation development gets interesting when the application must perform updates against the data read within the transaction. This is because the transactionally consistent view of the database, as of the start of the transaction, necessarily masks any conflicting updates – they are only discovered when the update is sent to the database and a conflict error is raised.

The screenshot shows the SqlException that results from the conflict – best practice dictates that the application intercept this exception and take corrective action as in any optimistic locking application.



This side-effect of optimistic concurrency control (the application was optimistic that no other application users would update the same data) means that the developer must do a little extra work to ensure that their user's data is not lost. Much of this logic will already exist in systems that "disconnect" their data from the database, probably based on the row-level incrementing timestamp value, to provide optimistic concurrency control. If this logic already exists in your application then Read Committed Snapshot may be a better choice to eliminate the blocking in the data population phase and to maintain conflict detection without the need to maintain an active transaction in the database – it remains best practice to get in and out of the database as quickly as you can to avoid tying up resources, in this case the version store.

Snapshot Isolation provides an automatic mechanism for detecting conflicts within a transaction that avoids the need to add timestamp columns or to make other schema changes – if a conflict is detected when the update is sent to the database then a SqlException is thrown and the current transaction is aborted.

Consider the Visual C# 2005 code fragments below (note that best practice dictates that try/catch logic would normally be wrapped around the (missing) connection Open and around the Fill command, which is where the transaction is initiated):

```
// (Definition of a SqlConnection object skipped)

// Define a transaction object using the Snapshot Isolation Level.
SqlTransaction DT = sqlCon.BeginTransaction(IsolationLevel.Snapshot);

// Hook up Select & Update command handlers to the dataadapter
// Use the Snapshot transaction "DT"
SqlCommand selectCMD = new SqlCommand();
selectCMD.Connection = sqlCon;
selectCMD.Transaction = DT;
selectCMD.CommandText = "select MessageNo, MessageText " +
                        " from dbo.DialogText";
sqlDataAdapter1.SelectCommand = selectCMD;

SqlCommand updateCMD = new SqlCommand();
updateCMD.Connection = sqlCon;
updateCMD.Transaction = DT;
```

```
updateCMD.CommandText = "update dbo.DialogText " +
                        "set MessageText = @MessageText " +
                        "where MessageNo = @MessageNo";
updateCMD.Parameters.Add("@MessageText",
                         SqlDbType.NVarChar,
                         15, "MessageText");
updateCMD.Parameters.Add("@MessageNo",
                         SqlDbType.SmallInt,
                         2, "MessageNo");

sqlDataAdapter1.UpdateCommand = updateCMD;
// Now get the data
sqlDataAdapter1.Fill(dataSet1, "DialogText");
```

The Visual C# code above uses ADO.NET and the Sql Client to populate a dataset with data from a SQL Server 2005 that has had Snapshot Isolation enabled by the DBA. A normal Windows Forms application would now commit the transaction, disconnect from the database and present the data to the user. In the code above we have left the transaction open so as to allow another transaction to change the data read into the **dataset1** dataset.

The code below illustrates the return of the data to the database:

```
// Bind the data to the form's grid control
dataGridView1.DataSource = dataSet1;
dataGridView1.DataMember = "DialogText";
dataGridView1.AutoGenerateColumns = true;

// ...Time passes, conflicting changes take place

// User presses "update now" button:

try
{
      sqlDataAdapter1.Update(dataSet1, "DialogText");
      dialogTrans.Commit();
      dataSet1.AcceptChanges();
}
catch (SqlException h)
{
      string errorMessages = "";
      for (int i = 0; i < h.Errors.Count; i++)
      {
            errorMessages += "Index #" + i + "\n" +
            "Message: " + h.Errors[i].Message + "\n" +
            "ErrorNumber: " + h.Errors[i].Number + "\n" +
            "LineNumber: " + h.Errors[i].LineNumber + "\n" +
            "Source: " + h.Errors[i].Source + "\n" +
            "Procedure: " + h.Errors[i].Procedure + "\n";
      }
      if (dialogTrans.Connection != null)
      {
            dialogTrans.Rollback();
      }
```

```
        MessageBox.Show(errorMessages, "Conflict Errors");
}
catch (Exception i)
{
// for general exceptions make sure the transaction is rolled back
        dialogTrans.Rollback();
}
```

In the second snippet the dataset object is bound to a grid control on the form, where the user is free to make multiple updates to the data – in parallel other transactions have made conflicting changes. When the user requests that their changes be stored to the database the dataset's changes are sent through the sqlDataAdapter as a series of database update statements – the first update statement that detects a conflict will cause an exception to be fired that rolls back the work, if no exception is fired then the transaction is explicitly committed by the dialogTrans.Commit() statement.

The exception handler above catches the SqlException thrown and formats an error message (see dialog screenshot above) which could be sent to an application log - the conflict can be explicitly tested as SqlException.Errors[i].Number, where 3960 is the error number – note that best practice dictates that the whole error collection be checked in case of other, more severe errors.

Note also that the SqlException handler tests the dialogTrans SqlTransaction object to see if it is still active (i.e. it has a connection to the database) – if it is active it is rolled back to ensure transactional consistency. If you attempt to commit/rollback an inactive object you will see a SystemException indicating a COM+ exception code of 0xE0434F49 (-532459699) with text "This SqlTransaction has completed; it is no longer usable"

Once the conflict is detected the application should inform the user that their changes have been rejected and offer them the opportunity to resubmit their changes under a new transaction.

Conflict detection, resultant transaction rollback and then the need to resubmit the work illustrates a decision facing the developer – if the optimistic concurrency control mechanism is too optimistic, and data conflicts occur frequently, then pessimistic concurrency control may be a better choice. You must balance the blocking caused by lock contention versus the additional work caused by a transaction rollback when deciding which method of transaction isolation to deploy within your application.

Minimizing Update Conflicts

Applications designed to work under optimistic concurrency must invest in conflict avoidance techniques – although they can be handled it is bets to avoid the cost of transaction rollback & retry if possible. There are two main methods to reduce the risk of a conflict:

- Application design: when gating access to shared resources it is possible to "reserve" them for the updating application, the concurrency of the individual data item is reduced (others cannot access it whilst it is reserved) but the overall system concurrency is not impacted – examples of this technique are seen in online ticket booking applications.
- Index design: ensure that modified rows are uniquely identifiable by ensuring appropriate indexes are used in the access path of update queries.

Illustrating Optimistic Concurrency Behavior

The table below uses the simple Customers/Orders/Items schema to illustrate the behavior of the two, new optimistic concurrency isolation levels and both the default Read Committed with locking and Read Uncommitted.

Two client sessions are required:

- Updater → sets up the sample schema and inserts/updates data
- Reader → runs under the four isolation levels to illustrate their behavior

The simple schema is recreated at the start of each run by deletion and re-creation of the database note that to drop a database requires that no sessions can be active in the database that is being dropped (including that of the dropping session).

READ COMMITTED with locking	
Updater Session	**Reader Session**
- Connect to the SQL Server 2005 Beta2 Instance using SQL Management Studio - Load the PessimisticSetup.sql script - If required drop the [Pessimistic] database - Execute the PessimisticSetup.sql script to create the [Pessimistic] database and populate a small sample schema	<disconnected>

READ COMMITTED with locking	
Updater Session	**Reader Session**
	- Open another query window in SQL Management Studio and connect to the Yukon Beta2 Instance - Load the PessimisticReader.sql script and ensure that the SET TRANSACTION ISOLATION LEVEL READ COMMITTED statement is not commented out, and that the SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED is commented out. - Execute the PessimisticReader script *The script will connect to the [Pessimistic] database and execute the three queries, each returning data as created in the setup script* *Note that the lock_wait session option is set to zero so as to report lock conflicts immediately without blocking*
- Load the PessimisticUpdater.sql script - Select & execute statement block 1 *This will lock records in the OrderHeaders table as the explicit transaction is left active*	

	- Execute the PessimisticReader script
	This time the first & third SELECT queries will fail as their data is locked (without the zero lock_wait the query would have hung until either the update completed or the lock wait timeout was reached. You should see a message:
	`Msg 1222, Level 16, State 51, Line 8`
	`Lock request time out period exceeded.`
	If you select the Results tab you will see that the second query completed successfully.
	Why did the second query work?
	If you look at the execution plans for the three SELECT statements you'll see that the second seeks into the OrderHeaders table based on values in the OrderDetail table, thus avoiding the locked row as it has no corresponding OrderDetails row(s). (To display the execution plans in SQL Management Studio, highlight the three SELECT statements and press <ctrl>L)

READ COMMITTED with locking	
Updater Session	**Reader Session**
- Select & execute statement block 2 *This will rollback the changes made by the insert statement* **-** Select & execute statement block 3 *This will lock rows in both the OrderHeaders & OrderDetails tables*	
	- Execute the PessimisticReader script *This time all three SELECT queries will fail as each one's data is locked*
- Select & execute statement block 4 *This will rollback the changes made by the previous insert statements*	
	- Execute the REPORT 1 Select statement in the PessimisticReader script *This will run successfully and return the base data entered when the schema was created*
- Select & execute statement block 5 *This inserts & commits a new order*	
	- Execute the REPORT 2 & REPORT 3 Select statements in the PessimisticReader script *These will run successfully and will return the base data as well as the new order (because it was committed) – this behavior can cause issues in report suites that expect constant data across multiple select statements* ***How would you resolve this?*** *(see the discussion below for an answer)*
Close this session	**Close this session**

The sessions above illustrate the positive behaviors of Read Committed with locking:

- Committed updates are seen immediately
- Locks can be used to serialize access to data, short lock waits (known as "blocking") are usually acceptable to most systems

It also illustrated the potential negatives:

- Long lock waits can cause command timeout or lock timeout errors and increase the risk of deadlocking (mutually exclusive lock requests)

- Data can change while running a suite of queries that expect consistent data.

There are a number of techniques available for mitigating the negative behavior of Read Committed with locking:

- Make a read-only replica of the data for reporting purposes. In versions of SQL Server prior to SQL Server 2005 this can be done with a one-off backup/restore; and continuously with periodic log shipping or replication (various techniques).
- Use Read Uncommitted to avoid lock waits (see next table)
- Use Repeatable Read/Serializable transaction isolation levels in a single transaction that spans the report queries to avoid data changing in between queries
- In SQL Server 2005 the previous techniques apply and there are new options such as optimistic concurrency with Snapshot Isolation (across several queries), Read Committed with snapshots (for a single query), and database mirroring/viewpoints.

READ UNCOMMITTED	
Updater Session	**Reader Session**
- Connect to the SQL Server 2005 Beta2 Instance using SQL Management Studio - Load the PessimisticSetup.sql script - If required drop the [Pessimistic] database - Execute the PessimisticSetup.sql script to create the [Pessimistic] database and populate a small sample schema	<disconnected>
	- Open another query window in SQL Management Studio and connect to the SQL Server 2005 Beta 2 Instance - Load the PessimisticReader.sql script and ensure that the SET TRANSACTION ISOLATION LEVEL READ COMMITTED statement is commented out, and that the SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED is not commented -Execute the PessimisticReader.sql script, the results should be same as when first executed under Read Committed with locking (above)
- Load the PessimisticUpdater.sql script - Select & execute statement block 1 *This will lock records in the OrderHeaders table as the explicit transaction is left active*	
	-Execute the PessimisticReader.sql script *This time all three queries complete successfully, however notice the discrepancy between the first and second query caused by the first select reading the uncommitted OrderHeader record that is filtered out by the join in the second query* *This sort of anomaly can cause problems, especially for systems that checksum totals to ensure accuracy*

READ UNCOMMITTED	
Updater Session	**Reader Session**
- Select & execute statement block 2 *This will rollback the changes made by the insert statement* **-** Select & execute statement block 3 *This will insert & lock rows in both the OrderHeaders & OrderDetails tables*	
	- Execute the PessimisticReader script *This time no anomaly is obvious, however if the addition of OrderDetail records requires extensive validation, or a large number of OrderDetail rows is being inserted across multiple user dialogs; it is possible that the report could contain transient numbers, caused by partially entered orders – the kind of side-effect that gives DBA's white hair as by the time they investigate the anomaly the transaction is completed and the data is stable.*
- Select & execute statement block 4 *This will rollback the changes made by the previous insert statements*	
	- Execute the REPORT 1 Select statement in the PessimisticReader script *This will run successfully and return the base data entered when the schema was created*
- Select & execute statement block 5 *This inserts & commits a new order*	
	- Execute the REPORT 2 & REPORT 3 Select statements in the PessimisticReader script *These will run successfully and will return the base data as well as the new order (because it was committed) – the Read Uncommitted behavior is the same as Read Committed with locking*
Close this session	**Close this session**

The use of Read Uncommitted is an effective way of obtaining quick results from the database without blocking or being blocked by other users – this technique is

recommended when data accuracy is not paramount and the use of transient data is acceptable.

Where non-blocking access to accurate data within a single query or across several queries is required, and offloading the query to a replica database is not possible or desirable then SQL Server 2005 introduces two optimistic transaction isolation schemes whose behavior is explored below:

READ COMMITTED with snapshots	
Updater Session	**Reader Session**
- Connect to the SQL Server 2005 Beta2 Instance using SQL Management Studio - Load the OptimisticSetup.sql script - If required drop the [Optimistic] database - Execute the OptimisticSetup.sql script to create the [Optimistic] database and populate a small sample schema	<disconnected>
	- Open another query window in SQL Management Studio and connect to the SQL Server 2005 Beta 2 Instance - Load the OptimisticReader.sql script and ensure that the SET TRANSACTION ISOLATION LEVEL READ COMMITTED statement is not commented out, and that the SET TRANSACTION ISOLATION LEVEL SNAPSHOT is commented out. -Execute the OptimisticReader.sql script *The results should be same as those returned by the PessimisticReader.sql script when first executed under Read Committed with locking (as above)*
- Load the OptimisticUpdater.sql script - Select & execute statement block 1 in the OptimisticUpdater.sql script *This will lock records in the OrderHeaders table as the explicit transaction is left active*	

READ COMMITTED with snapshots	
Updater Session	**Reader Session**
	- Execute the OptimisticReader.sql script again *This time the script executes without lock contention with the Updater session resulting in a lock timeout – the snapshot data allows the query to access the original data and to produce a consistent view of committed data*
- Select & Execute statement block 2 in the OptimisticUpdater.sql script *This will rollback the change made by the insert statement*	
	- Execute the REPORT 1 Select statement in the OptimisticReader script *This will run successfully and return the base data entered when the schema was created*
- Select & Execute statement block 3 in the OptimisticUpdater.sql script *This block inserts & commits a new order, simulating online activity whilst a report suite is running*	
	- Execute the REPORT 2 & REPORT 3 Select *As with the classic Read Committed with locking behavior these queries will pick up the data inserted by the Updater Session and hence be out of synchronization with the first report.*
Close this session	**Close this session**

The sessions above illustrate how the new Read Committed with snapshots behavior can help when the system design requires a mix of update/long-running read activity – by obtaining the original data for changes that are uncommitted the report writer can obtain data:

- without blocking/being blocked by other users
- that is consistent within transaction boundaries

Unlike the Read Uncommitted behavior that will return data that may never be committed to the database, and that can give inconsistent views of the database.

The one negative behavior is that data is subject to change whilst the Reader Session runs – this can impact a set of related queries that require a transactionally consistent view of the database across statements. In pre-SQL Server 2005 versions of SQL Server there were two isolation levels that delivered the consistent view:

- Repeatable Read – locks the data read within the transaction
- Serializable – locks the sets read within the transaction

Both of these isolation levels shape application concurrency, and hence are usually unsuitable for scenarios where a mix of multiple, random data changes must be coupled with long-running read transactions, especially when the impact of lock escalation (the term given to the run time escalation of multiple granular locks into fewer, less granular locks to conserve lock space) is taken into consideration.

Prior to SQL Server 2005 application designers would usually deliver the needed consistency and concurrency by taking occasional snapshots of the data for reporting, or by implementing some form of row timestamp/datetime versioning within the application. This design pattern can be avoided by using the new Snapshot Isolation behavior as illustrated below:

SNAPSHOT ISOLATION	
Updater Session	**Reader Session**
- Use SQL Server Management Studio to connect to the SQL Server 2005 Beta2 Instance - Load the OptimisticSetup.sql script - If required drop the [Optimistic] database - Execute the OptimisticSetup.sql script to create the [Optimistic] database and populate a small sample schema	<disconnected>

SNAPSHOT ISOLATION	
Updater Session	**Reader Session**
	- Connect to the SQL Server 2005 Beta 2 Instance - Load the OptimisticReader.sql script and ensure that the SET TRANSACTION ISOLATION LEVEL SNAPSHOT is <u>not commented out</u>, and that the SET TRANSACTION ISOLATION LEVEL READ COMMITTED statement is <u>commented out</u> -Select & execute the statements up to and including the REPORT 1 Select statement *Remember that the Snapshot transaction does not begin until data is accessed, if the Select statement is not executed then the transaction will have access to any data committed between the BEGIN TRANSACTION and the first query to access data.*
- Load the OptimisticUpdater.sql script, connect to the [Optimistic] database and select & execute statement block 3 to insert a new order into the schema. *This new data should not be seen by the Reader Session until its transaction completes*	
	- Select & execute the REPORT 2 & REPORT 3 Select statements in the OptimisticReader.sql script *The output should not show the data inserted by the Updater session – the data is now consistent across the REPORT Select statements, <u>without</u> blocking the Updater session* - Select & execute the ROLLBACK TRANSACTION statement to complete the reporting transaction - Execute the entire OptimisticReader.sql script *This time the results should include the data entered in the Updater session.*
Close this session	**Close this session**

This final session illustrated the multi-statement behavior of a transaction running under Snapshot Isolation – the results were consistent across Select statements even though the Updater session had successfully committed new data. This consistency is achieved without the potentially negative impact of Repeatable Read & Serializable.

Note that the same script was used to demonstrate the behavior of the new Read Committed with snapshots isolation level, and included the same Begin & Rollback Transaction statements, but because Read Committed applies at the statement level there is no "memory" across statements.

The Snapshot Isolation level must be explicitly requested via a SET TRANSACTION ISOLATION LEVEL statement, and then activated by both starting a transaction and accessing data.

Administrative Best Practices

As an administration enabling Read-committed Isolation or snapshot isolation should be decided with care as the impact on performance may be negative when used to solve the wrong problems. If performance problems exist due to lack of proper indexing and query performance suffers changing to row versioning probably won't solve this problem. If query performance suffers due to significant conflicts due to a mixed workload of readers and writers then Read-committed Isolation (with snapshot) may be all that is needed. If transactional consistency is needed for long running transactions then snapshot may be needed however, each of these incrementally puts a heavier load on TempDB.

Database-level Settings

Because snapshot isolation is configured at the database level, administrators need to enable snapshot isolation for each database that requires it. If cross-database transactions are attempted with snapshot isolation and not all databases are configured for them; the transaction will fail unless a locking hint is used to override the default.

If all databases are configured for snapshot isolation, then cross-database transactions will use a consistent snapshot across databases within one server instance. For example, assume you have two tables in two databases that are enabled for snapshot in the same server, and your update transactions make the same changes to these two tables. Your transaction under snapshot isolation never gives you different values for the two tables.

Upgrade Issues

While upgrading to SQL Server 2005 is dynamic and requires only internal changes to support row versioning; however an extra 14 bytes per data row is required to store versioning data irrespective of snapshot/read committed with snapshot being enabled – this data is added when the row is updated and hence can lead to page splits (for tables

with clustered indexes) or row forwarding for heaps; also changes will need to be made to all text/image data to allow row versioning.

None of these changes are made during upgrade but are instead made during later data row and text/image data modifications. It is important that Database Administrators who manage systems with large LOB data columns are aware that, for upgraded databases, the text/image columns will be modified dynamically to include versioning changes when any part of the LOB data is changed. All the text/image pages belonging to that particular text/image value will be changed. This operation can potentially be very expensive for large values which extend over many pages (due to page allocations, copying and logging). You will only pay this overhead when you modify the text/image column value; there is no overhead if you only modify the parent data row.

Because text/image data modifications can be run in a minimally logged mode, DBAs should determine if performing a separate and manual step as part of the upgrade to SQL Server 2005 would be beneficial. The change of fragment size could cause a lot of fragmentation to existing blobs when there are lots of random, small updates to only pieces of the text/image values. While random, small, partial updates to blobs are not the common type of text/image manipulation performed this overhead could be potentially expensive (both in terms of time and logging) in a live system. DBAs need to consider adding a step (during upgrade) which will modify all text/image data to have this new format before going live with SQL Server 2005.

To perform this modification the general process of steps would be:

1. Upgrade the database to SQL Server 2005
    - In place – In place upgrades are the easiest to perform as all components are updated in one upgrade process
    - Install SQL Server 2005 on a new server and then use backup/restore to upgrade. SQL Server 2005 supports restoring SQL Server 2000 databases.

   See the "Preparing to Upgrade to SQL Server 2005" topic in the BOL for complete details about how to successfully upgrade from SQL Server 2000.
2. Verify and/or change the Recovery Model to either SIMPLE or BULK_LOGGED. Simple is preferred as a full database backup will be performed upon successfully completing this process.
3. Perform an update to all text/image data values (note that executing a command such as:

   UPDATE ... SET a=a

   will not actually update the text data, instead use a command sequence such as:

   DECLARE @ptrval binary(16)
   DECLARE @dataval CHAR(1)
   SELECT @ptrval = TEXTPTR(anytab.a) ,
   @dataval = SUBSTRING(anytab.a,1,1)

```
  FROM dbo.anytable AS anytab
  WHERE anytab.primarykey = 'unique value'
UPDATETEXT anytab.a @ptrval 0 0 @dataval
GO
```

To do this for all records in a table wrap the above set of commands in cursor loop that iterates through all values in the table.

4. Change the Recovery Model back to the desired recovery model (i.e. Full Recovery Model).

Version Store Usage of TempDB

The version store is maintained in TempDB (and memory - versions associated with short running transactions, such as those found in OLTP workloads such as defined by the TPC-C™ benchmarks may never be written to disk). Because of this sizing TempDB is critical to the overall performance of the system and whether or not row versioning will even be possible for some long-running transactions. For example, if the TempDB runs low on space, performance will degrade as the version store attempts to cleanup. The regular cleanup function is performed every minute in the background attempting to reclaim all reusable space from the version store. When TempDB runs out of free space, the regular cleanup function is called before auto-growth occurs. When the disk is full and auto-grow cannot increase the file sizes, the version store is first truncated to return space and then if space pressure continues row versioning is stopped. If a snapshot query later encounters a record and wants to read an older version of the record which was not generated due to space constraints, the query fails. Updates and deletes don't fail, only queries requesting their row versions fail because once the version store fills updates/delete no longer generate row versions.

One alternative is to detect a long running snapshot query/transaction and terminate it. By canceling the query you can help reduce the size of the version store. This can be automated by associating a script with the event (Error number 3958) in TempDB. This is a more desirable error behavior for most applications. Otherwise, users might have many more transactions that fail due to out of space issues in the version store.

To ensure smooth running of a production system using snapshot isolation, the DBA must allocate enough disk space for TempDB such that there is always roughly 10% free space. When free space falls below 10%, system throughput will degrade as the version cleanup process will spend more time trying to reclaim space in the version store.

It is recommended that if IO performance in TempDB becomes an issue, the DBA should create more than one file for TempDB on different disks to increase IO bandwidth. In fact, on multiproc machines increasing the number of files to equal the number of processors can often yield even greater gains. See Q328551: Concurrency Enhancements for the Tempdb Database for more information.

If any one application on a server creates unexpectedly large numbers of version store entries it can impact other applications by physically filling the shared TempDB database. Large numbers of versions, or long running transactions (not necessarily running under Snapshot Isolation) that prevent version cleanup can lead to out of space related problems.

Sizing TempDB

If only read committed isolation is required, sizing TempDB is not as critical as the row versions are not likely to be held for long periods of time. However, long running transactions – both readers as well as any writers – can cause problems when the transactions are excessively long. However, if you are running in Snapshot Isolation mode, the need for space in TempDB is increased. It is recommended that you use the following formula to estimate the amount of space needed in TempDB for running Snapshot Isolation queries.

To estimate how much space you need to have in TempDB, you need to first consider that an active transaction must keep all its changes in the version store, so that a snapshot transaction that starts later will be able to get to the old versions. In addition, if there is an active snapshot transaction, then all the version store data generated by previous transactions that are active when the snapshot starts must also be maintained until the last snapshot transaction using them completes.

Size of Version Store = 2(Version store data generated per minute *
Longest running time (minutes) of your transaction)

(Note that the 2* multiplier reflects the possibility of two long running transaction with a slight overlap, thus leading to twice the longest running transaction time before the snapshot records can be released)

Version store data generated per minute for the system on behalf of active transactions can be compared to the log rows generated per minute – when sizing remember that a log record will contain data changes and the snapshot the entire row. Using Performance Monitor counters you can see the amount of version store data generated per second. In your production system, you should consider monitoring these counters in order to fine tune the size of TempDB.

Note that SQL Server 2005 online index build transactions are excluded from this calculation. They do not directly affect the overall version store cleanup because their processing is handled differently than user transaction version management.

If you have enough disk space, always allocate more than your estimate to prevent potential space problems. When estimating size of TempDB, the DBA must also consider the space requirements of DBCC CHECKDB, DBCC CHECKTABLE, index building, query, and other activities.

Monitoring Version Store Activity

There are a variety of ways to monitor version store activity – from functions which access virtual tables to performance monitor counters to Profiler Events. Each one offers a different perspective on the activity currently occurring on the system.

Function: dm_tran_active_snapshot_database_transactions ():

This function returns a virtual table for all active transactions with a row version-related sequence_number. Only transactions which are running under snapshot isolation will include a sequence number. Read-only transactions in auto-commit mode and system transactions will not appear in this virtual table.

The function returns these columns:

Column Name	Type	Description
transaction_id	bigint	A unique number given for each transaction started in the system. Every transaction has id.
transaction_sequence_num	bigint	A unique sequence number indicating when the transaction starts. Transactions that do not generate version records, and do not use snapshot scans do not need transaction sequence_number.
commit_sequence_num	bigint	A sequence number indicating when the transaction finishes (commits or aborts). For active transaction, the value is NULL.
is_snapshot	bit	If the transaction is snapshot transaction
Spid	Int	Process id of the connection that started this transaction.
first_snapshot_sequence_num	Bigint	When a snapshot

		transaction starts, it takes a snapshot of all active transactions at the time. This is the lowest sequence_number of the transactions in the snapshot.
max_version_chain_traversed	int	Max length of version chain traversed
average_version_chain_traversed	int	Average length of version chain traversed
elapsed_time_seconds	bigint	The elapsed time in seconds since the transaction obtained the sequence_number.

The table outputs data in the sequence of the "transaction_sequence_number" column. This shows transactions based on start time and therefore also "elapsed_time (seconds)" to help you determine transactions which are long running.

To find the 10 longest (i.e. earliest) transactions:

```
SELECT TOP 10 atx.transaction_id, atxs.[name]
FROM sys.dm_tran_active_snapshot_database_transactions() AS atx
INNER JOIN sys.dm_tran_active_transactions as atxs
  ON atx.transaction_id = atxs.tran_id
```

To find out the transaction that has traversed the longest version chains:

```
SELECT TOP 1 atx.*
FROM sys.dm_tran_active_snapshot_database_transactions() AS atx
ORDER BY atx.max_version_chain_traversed
```

Function: dm_tran_transactions_snapshot():

This function returns a virtual table for all active transactions with a row version-related sequence_number. This function returns a virtual table for the sequence_number of transactions that are active when each snapshot transaction starts.

The function returns these columns:

Column Name	Type	Description

transaction_sequence_num	BIGINT	sequence_number of a transaction, e.g. 'X'
snapshot_id	BIGINT	Statement Id for each statement started under read committed snapshot
snapshot_sequence_num	BIGINT	sequence_number of a transaction that is active when transaction 'X' starts.

Example:

T_1: BEGIN TRAN T1
T_1: SELECT … FROM …
T_2: BEGIN TRAN T2
T_2: SELECT … FROM …
T_3: BEGIN TRAN T3

First, find out the transaction sequence information:

```
SELECT stx.transaction_sequence_num as N'sequence_number'
    , atx.[name]
    , stx.first_snapshot_sequence_num
    , stx.commit_sequence_num
FROM sys.dm_tran_active_snapshot_database_transactions() AS stx
INNER JOIN sys.dm_tran_active_transactions AS atx
ON stx.transaction_id=atx.tran_id
```

For T3 this returns:

sequence_number	name	first_snapshot_sequence_num	commit_sequence_num
50	T1	0	NULL
52	T2	50	NULL
53	T3	50	NULL

Second, find out the snapshot transactions running:

```
SELECT txs.*
FROM sys.dm_tran_transactions_snapshot() AS txs
```

For T3 this returns:

transaction_sequence_number	snapshot_sequence_number	Snapshot_id
50	0	0
52	50	0

53	50	0
53	52	0

What the result shows are that there are multiple snapshot transactions running and the first one has a transaction sequence number of 50. Any updates at this sequence number must stay in the version store until the transaction completes. Transaction T2 picks up where this one leaves off with a sequence number of 52. An additional transaction starts (T3) and because there are two transactions with row versions already T3 will have a dependency (in terms of order) on BOTH T1 and T2.

Performance Monitor Counters

The Windows 2003 Performance tool (System Monitor in Windows 2000) allows the DBA to monitor a variety of system and SQL Server counters in a graphic interface, log the performance counters in a performance log, analyze the performance log, and define actions based on these events. There is also API so that DBA can develop their own programs to access these counters and take proper actions.

The various counters present are as follows:

Counter	Explanation
(1) Free Space in tempdb (KB)	The free space in tempdb in KB. The Version store is in tempdb, so the DBA has to make sure that the tempdb has enough free space. This is implemented by having a running count of free extents in tempdb.
(2) Version Store Size(KB)	The size of the version store in KB. The DBA knows how much space in tempdb is being utilized for the version store.
(3) Version Generation rate(KB/s)	The version generation rate in KB per seconds.
(4) Version Cleanup rate(KB/s)	The version cleanup rate in KB per seconds.
With the information from the counters 3 and 4, the DBA can predict the size requirement of TempDB and make space for it.	
(5) Version Store unit count	Number of version store units used in the Version Store. This counter reflects the currently active version unit count.
(6) Version Store unit creation	Creation of new version store units in Version Store. This counter represents the count since the instance was started.
(7) Version Store unit truncation	Truncation of version store units in Version Store. This counter represents the count since the instance was started.

From counters 5, 6 and 7, the DBA would know from the active count & creation count when a system has reached steady state. A high truncation rate can indicate that TempDB is/was under space stress from other applications who are also using TempDB, and could be a cue to the DBA to increase TempDB size.	
(8) Update conflict ratio	The fraction of update snapshot transactions that have update conflicts to the total number of update snapshot transactions. The DBA would know how appropriate the snapshot isolation transaction level is based on this percentage. We note that a transaction can have multiple updates. Here the measure is the number of transactions that do updates and not the number of updates themselves. The reason for not taking the number of updates as the measure is that this would give a deceivingly low figure. This is so because for an update conflict, the numerator count gets incremented by one only with other earlier updates in the transaction getting rolled back; whereas in the case of a successful transaction, the denominator count is incremented by the number of updates in the transaction. Note: This is a rate counter and gives the Update conflict ratio for the last second.
(9) Longest Transaction Running Time	The longest Running time of any transaction in seconds. The DBA can look at this and find out if any transaction is running for unreasonably long time. To get more information, the DBA can query the virtual table *dm_tran_active_transactions()* to get the transaction_id and spid. This table is sorted on the column *elapsed_time* also giving the DBA the top-*n* longest running time transactions with their information.
(10) Transactions	The total number of active transactions. The number gives all the transactions that are active in the system. It includes the background internal transactions in SQL Server, but it does not include the system transactions.
(11) Snapshot Transactions	The total number of active snapshot transactions.
(12) Update Snapshot Transactions	The total number of active snapshot transactions that also include update statements.
(13) NonSnapshot Version Transactions	The total number of active non-snapshot transactions that generate version records. This is from updates that have not requested SNAPSHOT ISOLATION.
Since all snapshot transactions that do updates result in version generation, the	

total number of transactions that cause version generation is the sum of the counters 12 and 13. Also from the counters 11 and 12 the DBA can figure out the number of snapshot transactions that are read-only.

Thus from these counters, the DBA knows to what extent the versioning feature is being used and also how it is being used.

All the above counters are server-wide and have been grouped together in a new Performance Monitor Object called as "SQLServer: Transactions."

For more information

Since snapshot isolation impacts both administration and development aspects of a system it is important to make sure that all aspects are understood. If DBAs unnecessarily allow snapshot isolation where long running transactions occur and modifications are constant; users can experience problems committing changes if TempDB is not sized appropriately. Additionally, if developers expect READ COMMITTED Isolation to be set and it's not then data inconsistencies may occur undetected. Make sure to review all of the associate resources and participate in the beta newsgroups for more information.

Books Online Topics

Understanding Snapshot Isolation

Adjusting Transaction Isolation Levels

Using Snapshot

Preparing to Upgrade to SQL Server 2005

Knowledge Base Articles of Interest

Q328551: Concurrency Enhancements for the TempDB Database

Additional Reading

Generalized Isolation Level Definitions:
http://research.microsoft.com/~adya/pubs/icde00.pdf

A Critique of ANSI SQL Isolation Levels:
http://research.microsoft.com/research/pubs/view.aspx?tr_id=5

Newsgroups of Interest

Newsgroups for SQL Server 2005 can be found on the betanews.microsoft.com news server. Information about your account login and password are associated with how you received the beta of SQL Server 2005.

For specific questions and to previous customer questions, comments, etc. please see the Microsoft.beta.yukon.relationalserver.general forum.