Using a SQL Server database for consistent OLTP without blocking reads

Eliminating write skew errors in SQL Server's implementation of Serializable Snapshot Isolation through the use of application locks.

by Eric Driggs
2013-06-26
ericdriggs@hotmail.com

Use Case

Many systems, especially online transaction processing (OLTP) have a functional requirement for their database that 1) all data always be readable within a short time and that 2) writes be fully serialized without consistency errors. Using only isolation levels provided by SQL Server, one can fulfill either of these requirements, but not both at the same time.

Isolation levels provided by SQL Server:

As seen in the below table, the only isolation levels free of read anomalies are snapshot and serializable. Serializable blocks reads for the transaction duration, so it does not meet the read availability requirement specified. Snapshot isolation, however, suffers from a write skew anomalies resulting from race conditions between different transactions reading and updating the same data.

	Dirty Reads	Non-Repeatable Reads	Phantom Reads	Read Lock Duration	Write Skew anomalies
READ UNCOMMITTED	X	X	X		
READ COMMITTED,		X	X	Short	
REPEATABLE READ			X	Short	
SNAPSHOT				Short	X
SERIALIZABLE				Transaction	

[1, 2, 3, 4, 5]

For example, simultaneous credit card charges on an account could result an incorrect balance as the balance was updated by transaction B after the balance was initially read by transaction A. Since the database is using multi-version concurrency, even if A attempted to read the value again after B's update, transaction A is still blind to the balance update which occured in transaction B. In this example, if the initial balance were $0 and two charges for $5 occurred simultaneously, it would be possible for the ending balance to be only $5 instead of $10.

<u>**Enforcing Consistent Write Serializability:**</u>

Several mechanisms can be used to elminate write skew anomalies. All of them share a common strategy of tracking which data is currently being modified and then causing additional transactions to fail if they attempt to modify the same data after it was updated.

Regardless of the mechanism chosen, write skew can only be eliminated by requiring that all methods which write to the database use the same mechanism to generate transaction connections supporting the same type of serialization. Methods which only perform reads can access the databse using any type of connection.

Two common approaches to fix write skew are:

> “**Materialize the conflict:** Add a special conflict table, which both transactions update in order to create a direct write-write conflict.
>
> **Promotion**: Have one transaction "update" a read-only location (replacing a value with the same value) in order to create a direct write-write conflict (or use an equivalent promotion, e.g. Oracle's SELECT FOR UPDATE).” [3]

Materializing the conflict has an advantage in that it allows for fine grained control on preventing updates for a particular row or field being updated. This is similar to the approach which is used internally by PostgreSQL in their implementation of the serializable isolation level, allowing both short read locks and serializable write isolation. [7] The question then is how best to materialize the conflict in SQL server? One could create a special table for materialization, with fields for table and row. The downside of this approach is it that the materialization is persisted, which adds state complexity and potential I/O bottlenecks.

Promotion is more problematic in that TSQL does not support SELECT FOR UPDATE and any row-level promotion technique would require either DDL or logic based on the table definitions of the modified table.

Another alternative is to use a TSQL application lock as a promotion mechanism. Application locks are a light-weight mutex, global to a database and referenced by a 255 character unique identifier. [6] TSQL applications locks have the following advantages: 1) They can be create to automatically expire at the end of a transaction. 2) They do not require a DDL statement (e.g. create table) in order to be used against an existing database. 3) the algorithm for implementing a critical section through two mutexes is straightforward [8]. The main downside of application locks is the length limitation with auto-truncation, which means that a truly general purpose solution would have to hash the application lock name.

A pseudo-code implementation of this algorithm for obtaining a serialized, transactional connection may be implemented as follows:

```
begin transaction connection 1
validate first mutex available using connection 1
take first mutex on connection 1

begin transaction connection 2
validate first mutex available using connection 2
take first mutex on connection 2

rollback connection 1
return connection 2
```

References:

1 - Set transaction isolation level (Transact-SQL)
http://msdn.microsoft.com/en-us/library/ms173763.aspx

2 - A Critique of ANSI SQL Isolation Levels
http://www.cs.umb.edu/~poneil/iso.pdf

3 - https://en.wikipedia.org/wiki/Isolation_(database_systems)

4 - Weak Isolation in Relational Databases
http://www.evanjones.ca/db-isolation-semantics.html

5 - http://en.wikipedia.org/wiki/Snapshot_isolation

6 - http://en.wikipedia.org/wiki/Multiversion_concurrency_control

7 - PostgreSQL - Transaction Isolation
http://www.postgresql.org/docs/9.1/static/transaction-iso.html

8 - sp_getapplock (Transact-SQL)
http://msdn.microsoft.com/en-us/library/ms189823.aspx

9 - Solution of a Problem in Concurrent Programming Control (E. W. DIJXSTRA)
http://www.di.ens.fr/~pouzet/cours/systeme/bib/dijkstra.pdf