Optimized Locking

Last updated by | Peter Hewitt | Feb 2, 2023 at 3:41 AM PST

Contents

Introduction

The optimized locking feature is a new SQL Server Database Engine capability introduced in February 2023 for Azure SQL Database and is now GA in several regions.

Refer to optimized locking available regions for the list of currently supported regions.

What is optimized locking?

Optimized Locking (OL) reduces lock memory consumption, reduces lock escalations, and improves concurrency. It helps to reduce lock memory as very few locks are held for large transactions, avoiding lock escalations and allowing more concurrent access to the table.

Optimized locking is composed of two primary components: **Transaction ID (TID) locking** and **lock after qualification (LAQ)**.

1. Transaction ID (TID) locking: SQL Server traditionally used locks on key or row identifiers to synchronize across concurrent transactions. TID locking uses locks on the Transaction ID label present on the rows. This new scheme replaces potentially many key or row identifier locks with a single TID lock. A transaction updating one million rows previously held 1 million row locks. With OL, the transaction will hold only 1 TID lock to protect all those rows.

2. Lock after qualification (LAQ): SQL Server traditionally acquired U row locks to filter out rows to be updated. LAQ filters out rows without acquiring any locks using just a version of the row. This significantly improves concurrency. If there are two concurrent queries which update two different rows, they still get blocked today as two U locks are incompatible. With OL, these two queries will not block each other.

For more detailed information, see Optimized locking.

How to check if optimized locking is enabled for a given database

Database check: Optimized locking is enabled per user database. Connect to your database, then use the following query to check if optimized locking is enabled on your database:

```
SELECT IsOptimizedLockingOn = DATABASEPROPERTYEX('testdb', 'IsOptimizedLockingOn');
```

If you are not connected to the database specified in `DATABASEPROPERTYEX`, the result will be `NULL`. You should receive `0` (optimized locking is disabled) or `1` (enabled).

Optimized locking builds on other database features:

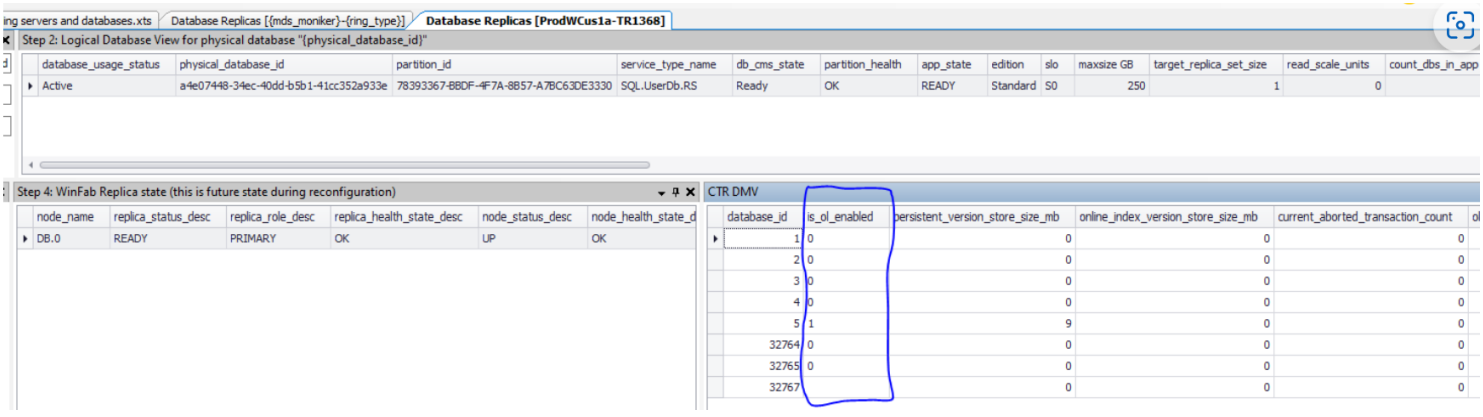
- Optimized locking requires accelerated database recovery (ADR) to be enabled on the database.
- For the most benefit from optimized locking, read committed snapshot isolation (RCSI) should be enabled for the database.

Both ADR and RCSI are enabled by default in Azure SQL Database. To verify that these options are enabled for your current database, use the following T-SQL query:

```
SELECT name
, is_read_committed_snapshot_on
, is_accelerated_database_recovery_on
FROM  sys.databases
WHERE name = db_name();
```

From XTS: open *sterling\database replicas.xts* view and input physical database id/partition id for the database. You can also open this view directly by double clicking the partition corresponding to this database in the *Partition info* tab from the *sterling\sterling servers and databases.xts* view.

The *is_ol_enabled* column in the *CTR DMV* tab of the *database replicas* view will have information on whether optimized locking is enabled or not.



From Kusto:

The following kusto query can be used to check if optimized locking is enabled for a given database at a given time. This query needs 'AppName' and 'database_id'. However, logical database id of the database can also be used in the predicate during the column 'logical_database_guid'.

```
MonSqlLocking
| where TIMESTAMP > ago(6h)
| where AppName == 'appName' and database_id == 5
| where isnotempty(is_tid_locking_enabled)
| project TIMESTAMP, is_tid_locking_enabled
```

Check if queries are getting retried because of lock after qualification

With optimized locking using read committed snapshot isolation (RCSI) the rows in an update query are locked only after they are qualified for the update. We call this as lock after qualification. By locking only the qualified rows, queries only lock the rows that they intend to update and this reduces blocking compared to lock before qualification where the rows are locked first and then qualified later for the update. The side effect of using lock after qualification is that rows may have to be requalified if the row has changed after it is read for qualification. We try to do the requalification at row level, but sometimes for queries with complex plan, requalification at row level may not be possible. In those cases, we simply abort the current statement, and retry the whole statement with lock before qualification. Aborting and retrying the whole statement can cause latency increase for the customers. We have a feedback mechanism to stop using lock after qualification if the number of such retries are frequent. The following kusto query can be used to check queries that got retried because of using lock after qualification and the corresponding logical reads done before doing the retry.

```
MonSqlLocking
| where TIMESTAMP > ago(6h)
| where AppName == 'e004f284b52c' and database_id == 32
| where isnotempty(logical_reads_before_retry)
| project TIMESTAMP, query_plan_hash_binary, logical_reads_before_retry
```

Higher value of *logical_reads_before_retry* indicates that query did a lot of work before retry, and all this work is aborted and retried resulting in latency increase. From production telemetry, very few databases have queries that have such retires and the feedback mechanism we have to stop using lock after qualification is helping those databases. The *query_plan_hash_binary* from the above query can be used to check query execution statistics from QDS telemetry using the kusto query below.

```
MonWiQdsExecStats
| where TIMESTAMP > ago(6d)
| where AppName == 'e004f284b52c' and logical_database_guid == 'D705AAEA-0073-44BE-B9E2-D0E2BFE3B071'
| where query_plan_hash == '0xB4CDA82B08189BC2'
```

IcM required

If an IcM is required it should be created with the *SQL DB Perf : Query Processing* owning team.

Public Doc Reference

Optimized locking

Internal Reference

CSS Mentoring Series - Azure SQL DB - Optimized Locking session

Root Cause Classification

Cases resolved by this TSG should be coded to the following root cause:
Azure SQL v3\Performance\Waits\Locking/Blocking

How good have you found this content?