FAQ

SQL Server FAQ: How to Diagnose Blocking and Deadlocks

Blocking and deadlocks in SQL Server often look similar from the outside, but they require different evidence and different fixes. This FAQ explains how to tell them apart, what to capture first, and how to validate a safe change before production rollout.

Databases - July 05, 2026

What is the fastest way to tell blocking from a deadlock?

Blocking means one session is waiting for another session to release a lock; a deadlock means two or more sessions are waiting on each other in a cycle, and SQL Server breaks the cycle by terminating one victim. The fastest clue is time: blocking often persists until the blocking transaction finishes, while a deadlock appears as an immediate error to one session.

A practical example is a report query waiting on a row locked by an update transaction. If the report eventually runs after the update commits, that was blocking. If one of the sessions returns error 1205 and the transaction is rolled back, that was a deadlock. As a decision rule, use the error message and wait behavior first; do not assume all long waits are deadlocks.

What evidence should you capture first when users report slow queries?

Capture the blocking chain, the exact wait type, the SQL text, and the session IDs involved. That gives you enough context to determine whether the problem is lock contention, a long-running transaction, or something unrelated such as I/O or CPU pressure.

If you need a quick operational workflow, start with the current waiter and blocker, then check the active request and transaction scope:

Validate the result by confirming that the blocker has an open transaction or is running a statement that holds locks longer than expected. If the waiting session shows a lock-related wait and the blocker is active, you have a concrete blocking chain; if not, keep investigating the execution plan or infrastructure path. For broader performance isolation, the workflow in Troubleshooting SQL Server Slow Query Performance and Index Scans can help separate locking symptoms from scan-driven slowness.

Which wait types matter most for blocking diagnosis?

Lock waits such as LCK_M_* are the primary indicators of blocking. They usually mean a session is waiting for a lock in a specific mode, such as shared, update, or exclusive, and the exact suffix helps narrow the contention pattern.

The nuance is that not every lock wait is equally important. A brief LCK_M_S wait during a busy workload may be normal, while sustained LCK_M_X or LCK_M_U waits often point to write contention or a transaction that is holding locks too long. If the wait type changes rapidly, sample more than once before concluding that the issue is stable.

A simple example is a queue table where many workers update adjacent rows. If sessions pile up on LCK_M_U and LCK_M_X, the likely fix is not to kill sessions but to shorten the transaction, narrow the rows touched, or change the access pattern. The validation point is whether the wait disappears after the transaction scope is reduced.

How do I identify the blocking session safely?

Identify the blocker by tracing the session that is being waited on, not by guessing which query is ?the problem.? The true blocker may be a legitimate business transaction, a trigger, an open transaction from a previous step, or even an idle session that never committed.

In practice, check the blocker?s last executed command, transaction state, and how long it has been open. If the blocker is sleeping but still owns locks, inspect the application path that left the transaction open. If the blocker is active, review whether it is touching too many rows or scanning more data than needed.

A useful caveat is that killing a blocker may free users quickly, but it can also roll back a large transaction and make the situation worse. Before taking that action, estimate the rollback cost and verify whether a less disruptive option exists, such as letting a long commit finish or moving the workload to a quieter window.

What causes deadlocks most often in SQL Server?

Deadlocks usually happen when two sessions access the same resources in different orders or when a transaction holds locks longer than necessary. The classic pattern is session A locking row 1 and requesting row 2 while session B locks row 2 and requests row 1.

A practical example is an application that reads a row, updates a related row in another table, and then returns to update the first row later in the same transaction. That mixed order creates a cycle risk under load. Another common cause is missing indexes that force broader scans, which expands the lock footprint and increases the chance of conflict.

The decision rule is straightforward: if the deadlock graph shows a repeatable access pattern, fix the code path or indexing pattern rather than retrying the same workload. If the deadlock appears only under unusual concurrency, validate whether isolation level, batching, or transaction length is the real trigger.

How do I capture a deadlock graph without guessing?

Use the built-in deadlock reporting path available in your SQL Server environment and collect the deadlock graph from an Extended Events session or the system health session if it is already enabled. The point is to capture the exact resources, lock modes, and victim selection, not just the 1205 error.

When you read the graph, look for the involved objects, the statements in each process, and the order of resource acquisition. That tells you whether the issue is code order, index design, or transaction scope. If you only capture the error number, you will know that a deadlock happened, but not why it happened.

A validation point is whether the same pair of statements appears across multiple deadlocks. If the pattern repeats, you have evidence for a durable fix. If the victim changes but the resource pair is stable, the root cause is still likely the same.

What is the best first fix for repeated blocking?

The best first fix is usually to reduce transaction duration and lock footprint, then verify whether the blocking chain becomes shorter or disappears. Shorter transactions release locks earlier, and narrower queries touch fewer rows, which lowers contention without changing business logic.

For example, if a batch job updates thousands of rows in one transaction, splitting it into smaller commits may eliminate hour-long blocking. If a read query is holding shared locks too long because it scans large tables, improving the access path can help. In some cases, the query shape is the real issue, and the same troubleshooting workflow for slow query performance and index scans also explains why the optimizer may be reading more data than expected.

Do not assume the first fix should be changing isolation level. That may reduce blocking in some systems, but it can also introduce other consistency trade-offs. The validation point is whether the change reduces waits without causing incorrect results or excessive rollback risk.

Can indexes reduce blocking and deadlocks?

Yes, the right indexes can reduce both blocking and deadlocks because they shorten the time a query holds locks and reduce the number of rows touched. Better indexing often changes a scan into a seek, which lowers lock pressure and narrows the conflict window.

The nuance is that an index fix only helps when the query pattern supports it. If the statement still updates many rows or runs inside a large transaction, the benefit may be limited. A useful example is a delete or update that targets rows by a selective predicate; if the supporting index is missing, SQL Server may lock far more data than necessary.

Treat index changes as a measured validation step, not a reflex. Compare the before-and-after execution plan, logical reads, and lock wait behavior under realistic concurrency. If the query becomes faster but the blocker still holds locks for too long, you have only partially solved the problem.

Should I use READ COMMITTED SNAPSHOT or other isolation changes?

Sometimes, but only after you verify the consistency and workload impact requirements. Row-versioning options can reduce reader-writer blocking, yet they do not eliminate writer-writer conflicts and they can increase tempdb pressure.

A practical example is a reporting query that frequently blocks OLTP updates even though it only reads data. In that case, row versioning may be a good fit if the application can tolerate reading committed versions rather than locking the live rows. The caveat is that you must validate tempdb capacity, query semantics, and any application assumptions about immediate visibility.

As a decision rule, consider isolation changes when most contention is read versus write. If the problem is primarily two writers colliding on the same rows, changing reader behavior will not solve it.

How do I decide whether to kill a blocking session?

Kill a session only when the impact of waiting is worse than the rollback cost and you have verified that the session is safe to terminate. That means checking what it is doing, how long it has been open, and whether it is inside a transaction that will take a long time to undo.

In operational terms, an idle blocker with an open transaction is often a stronger candidate than a busy session that is about to commit. If the blocker belongs to a critical batch job, forcing a rollback may create a larger outage than the original block. The safest rule is to estimate both user impact and rollback duration before acting.

A practical validation step is to notify the application owner or check the job history when possible. If the blocker is from an application pool or automated task, confirm whether a retry would immediately recreate the same condition. Killing a session without fixing the root cause usually just moves the pain into a new time window.

What should I verify before calling a fix production-safe?

Verify that the change reduces blocking or deadlocks under realistic concurrency, that it does not break transaction semantics, and that rollback is still acceptable if the change misbehaves. Production-safe means more than ?the query got faster? because lock behavior can change under load.

A concise validation checklist is:

- Re-run the workload with similar concurrency and data volume.
- Confirm the blocking chain or deadlock pattern no longer repeats.
- Check execution plans, waits, and transaction duration before and after.
- Confirm that no required rows are skipped, duplicated, or read inconsistently.

If you change code, index design, or isolation level, capture a baseline before rollout and compare it after rollout. The final check is whether the user-visible symptom disappears and the underlying wait pattern stays stable over time, not just during a single test run.

What is the practical takeaway for diagnosing blocking and deadlocks?

Blocking and deadlocks are related but not interchangeable, so the diagnosis starts with the symptom, the wait type, and the evidence trail. Blocking points to a session waiting on another session; deadlocks point to a cycle that SQL Server resolves by choosing a victim.

The fastest reliable workflow is to capture the waiter and blocker, confirm the wait type, inspect the transaction scope, and use the execution pattern or deadlock graph to find the real cause. From there, decide whether the right fix is shorter transactions, better indexing, a safer access order, or a carefully validated isolation change. If you can prove the pattern repeats and the change removes the wait without introducing new risk, you have a production-worthy answer rather than a one-time workaround.

Use this guidance together with role-based access control for NoSQL databases to connect the workflow with related operational context already available on the site.