ARTICLE

SQL Server Deadlocks: How to Detect and Resolve Blocking Issues

Deadlocks are not the same as blocking, and treating them as one problem leads to the wrong fix. This article shows how to tell them apart, capture evidence, and decide on safe remediation in SQL Server.

Databases - July 06, 2026

Key takeaways

SQL Server deadlocks happen when two or more sessions each hold locks the others need, and SQL Server resolves the cycle by killing one victim. Blocking is different: one session waits behind another, but no circular dependency exists. If you confuse the two, you can end up optimizing the wrong query, changing the wrong index, or escalating a problem that could have been fixed with a small transaction or lock-order change.

The practical goal is not just to identify that a deadlock occurred, but to determine why the lock cycle formed, which statements were involved, and whether the safest fix is query tuning, transaction redesign, isolation changes, or a data access pattern change. You should also know what evidence to capture before production changes and what to verify after rollout.

A reliable diagnosis usually combines deadlock graphs, wait evidence, query text, and transaction scope. That evidence tells you whether the issue is a genuine deadlock, long blocking chains that only look similar, or a workload pattern that creates repeated contention under peak load.

Why this matters operationally

Deadlocks are operationally expensive because they interrupt user transactions, trigger retries, and often create noisy alerting without making the root cause obvious. In production, the visible symptom is frequently an application error or a failed job, not a database message that clearly explains what happened.

This matters most in systems with concurrent writes, mixed read-write workloads, or application code that touches multiple tables in variable order. In those environments, a small code path change can create a repeatable lock cycle under load, especially when transactions are too large or when queries take different access paths depending on parameter values.

If you can separate deadlocks from blocking early, you can avoid overcorrecting. Blocking often requires reducing lock duration, improving indexes, or changing scheduling. Deadlocks require breaking the cycle itself, usually by aligning access order, shrinking the lock footprint, or reducing the chance that two sessions need incompatible locks at the same time. For a broader diagnostic baseline, this pairs well with the workflow in SQL Server FAQ: How to Diagnose Blocking and Deadlocks.

How deadlocks and blocking differ

Blocking is a wait relationship. One session holds a lock, another session needs a conflicting lock, and the second session waits until the first session finishes or releases the resource. The chain may be long, but the direction is still one-way.

A deadlock is a cycle. Session A holds a resource Session B needs, while Session B holds a resource Session A needs. Neither can continue. SQL Server detects the cycle and chooses a victim to terminate so the other session can proceed.

In practice, the distinction is visible in the evidence:

- Blocking usually shows a waiting session and a head blocker.
- Deadlocks usually produce a deadlock graph, error 1205, or XE/event evidence showing the victim and participating resources.
- Blocking can persist and grow over time.

- Deadlocks are typically brief, repeated events that resolve by force, but their workload impact can still be significant.

This distinction matters because the fix differs. A blocking chain may be acceptable in short bursts if it is bounded and predictable. A deadlock often signals a design or concurrency mismatch that will recur until the workload pattern changes.

What a deadlock looks like in production

A common production scenario is an order-processing system. One request updates an order header and then inserts line items. Another request updates a line item and then reads or updates the header for a status check. Under light load, both transactions complete. Under peak load, each session acquires one lock and then waits forever for the other resource, creating a cycle.

The application may only see a retryable failure. Operations may see intermittent 1205 errors, a sudden rise in failed writes, or latency spikes around the same code path. If the deadlock involves an index lookup followed by a lookup on another table, the issue may appear only when the optimizer chooses a different plan or when the workload reaches a concurrency threshold.

That is why deadlock investigation should focus on the exact statements and access order, not just the tables involved. Two queries can touch the same objects for months without conflict, then start deadlocking after a release changes the transaction scope or access pattern.

Compact workflow block

A practical workflow for investigation is:

This is intentionally compact because the goal is to preserve the evidence before the next occurrence, then use that evidence to decide whether the problem is caused by code, schema, or operational behavior. If query shape or access path seems relevant, the workflow often overlaps with Troubleshooting SQL Server Slow Query Performance and Index Scans, because a slow or scan-heavy statement can hold locks long enough to make a conflict more likely.

How to detect deadlocks reliably

The most useful signal is a deadlock graph or equivalent event data showing the victim, the competing sessions, and the locked resources. Error 1205 confirms that a deadlock occurred, but it rarely explains why.

In production, capture enough context to answer four questions: which session was chosen as the victim, what statement each session was executing, which resources were contended, and whether the transaction was part of a larger unit of work. If you only capture the application error, you will know that a deadlock happened but not how to prevent the next one.

Useful evidence usually includes:

- Deadlock graph or event payload
- Query text for each participant
- Session identifiers and transaction scope
- Application timestamp, request path, or job name
- Wait and lock context around the event window

If your platform already captures deadlock events, validate that the capture mechanism stores enough detail to reconstruct the sequence. If it does not, add a lightweight capture path before making changes to the workload. Evidence quality matters more than event volume.

What usually causes the cycle

Most deadlocks are created by a combination of access order and lock duration. If one transaction updates table A and then table B while another does the reverse, a cycle can form even if both queries are individually efficient.

Other recurring causes include:

- Large transactions that hold locks longer than necessary
- Different code paths touching the same tables in different orders
- Missing or inefficient indexes that force broader locking or longer execution times

- Read-modify-write patterns that overlap under concurrency
- Application retries that reintroduce the same collision immediately

Sometimes the root cause is not the query text alone but the surrounding unit of work. A transaction that includes external calls, long-running validation, or multiple data changes is much more likely to deadlock than one that acquires locks, updates rows, and commits quickly.

What this means in practice

In production, the correct response depends on what the evidence shows. If the deadlock is caused by two code paths updating the same objects in opposite order, the safest fix is usually to standardize the order. If the deadlock arises because one statement scans too many rows, indexing or query shape changes may reduce lock time. If a transaction is too broad, the fix may be to split it into smaller atomic units.

This is also where retry logic needs care. Retrying a deadlocked request is often appropriate, but retries are not a root-cause fix. If the same workload pattern still exists, retries can amplify load during peak periods and create a storm of repeated conflicts.

A useful rule is simple: if the event is rare and caused by unavoidable concurrency, controlled retries may be enough. If the same pair of statements deadlocks repeatedly, you need to change access order, reduce transaction scope, or change the data access path.

Safe resolution options and their trade-offs

The best fix depends on what you can change without creating more risk than the deadlock itself. Query tuning can reduce lock duration, but it may not eliminate the cycle if the access order remains inconsistent. Transaction redesign can be very effective, but it may require application changes and broader testing. Index changes can reduce contention, but they also introduce write overhead and maintenance cost.

A few common options and their trade-offs:

- Standardize object access order: strong fix for deterministic cycles, but requires code discipline across all paths.

- Reduce transaction scope: lowers lock duration and contention, but may require refactoring or additional consistency checks.
- Improve indexes: can shorten execution and reduce lock footprints, but every new index adds write overhead.
- Use row-versioning features where appropriate: can reduce reader-writer blocking, but must be validated carefully because it changes concurrency behavior and storage usage.
- Add application retry handling: useful as a resilience layer, but not a substitute for correcting the deadlock pattern.

The wrong fix is often the one that reduces symptoms while leaving the cycle intact. For example, forcing a plan or adding a broad index may change the timing enough to hide the deadlock temporarily, but the same pattern can reappear later under a different load shape.

Decision guidance

Choose the fix based on repeatability and scope.

If the deadlock is frequent and involves the same statements, treat it as a design issue rather than a random production event. Focus first on consistent access order and smaller transactions. If the deadlock is tied to one query that touches too many rows, examine the plan shape and supporting indexes. If the workload is mostly read-heavy and deadlocks arise between readers and writers, confirm whether row-versioning is available and acceptable for that database and application behavior.

When the evidence is ambiguous, do not change multiple variables at once. A single well-scoped change is easier to validate and rollback than a bundle of indexing, code, and isolation changes. The objective is to prove that the deadlock cycle is broken, not just to reduce incident noise for one release.

Common mistakes

One frequent mistake is assuming every timeout is a deadlock. Timeouts and deadlocks are different failure modes, and each one requires different evidence. Another common error is tuning only the victim statement while ignoring the other participant. In a cycle, both sides matter.

Other mistakes include:

- Capturing too little evidence to reconstruct the event
- Fixing the symptom by adding retries without changing contention behavior
- Assuming an index change is safe without checking write workload impact
- Ignoring transaction boundaries because the statements themselves look fast
- Testing in low-concurrency environments that never reproduce the production cycle

A related mistake is applying a fix that improves one transaction path while worsening another. In SQL Server environments, concurrency behavior often depends on the exact mix of reads, writes, and access paths. That is why validation must include representative concurrency, not just single-session testing.

Production readiness checklist

Before moving a deadlock fix into production, verify the following:

- You have a deadlock graph or equivalent event evidence for the recurring incident
- You know which statements, sessions, and resources formed the cycle
- The proposed change addresses the root cause, not just the symptom
- Any indexing change has been checked for write overhead and maintenance impact
- Any transaction change preserves business consistency and rollback behavior
- Application retry behavior is bounded and does not create retry storms
- The change has been validated under realistic concurrency, not only single-user testing
- A rollback plan exists if lock behavior or performance worsens after deployment

Final takeaway

SQL Server deadlocks are best handled as a concurrency design problem, not just a runtime error. If you can tell deadlocks from blocking, capture the right evidence, and verify the exact access pattern involved, you can choose a fix that actually breaks the cycle. In most environments, the safest resolution is the one that shortens transactions, aligns lock order, and preserves enough evidence to confirm the issue stays fixed after production rollout.