Preventing Deadlocks in SQL Server:

Deadlocks in **SQL Server** can cause serious performance issues if not understood and prevented properly.
Let us see how to prevent them.

1. Understand Deadlocks

- A **deadlock** occurs when **two or more sessions block each other**, each holding a lock and waiting for the other to release a resource.
- SQL Server automatically detects deadlocks and **chooses a victim** (terminates one session).
- Common causes:
    - Accessing resources in different orders.
    - Long transactions with multiple locks.
    - Missing indexes → table scans & unnecessary locks.
    - Poor query design.

2. Best Practices to Prevent Deadlocks

✅ A. Query & Transaction Design

1. **Keep transactions short and simple**
    - The longer a transaction runs, the higher the chance of deadlocks.
    - Avoid holding transactions open while waiting for user input.
2. **Access objects in a consistent order**
    - If all applications access tables in the **same sequence**, circular locking is avoided.
    - Example: Always update TableA before TableB.
3. **Use proper isolation levels**
    - Default READ COMMITTED is fine for many workloads.
    - Use **READ COMMITTED SNAPSHOT (RCSI)** or **SNAPSHOT isolation** to reduce shared locks.
    - Be cautious: SNAPSHOT uses versioning in tempdb.
4. **Avoid unnecessary locking hints**
    - Don't force HOLDLOCK or TABLOCKX unless absolutely needed.

✅ B. Indexing Strategy

1. **Create appropriate indexes**
    - Missing indexes → SQL Server does table scans → more locks.
    - Well-designed indexes minimize locking scope.
2. **Covering indexes**
    - Queries that hit covering indexes avoid touching the base table → fewer lock conflicts.
3. **Filtered indexes**
    - Reduce lock contention by narrowing focus to selective rows.

✅ C. Code & Application Layer

1. **Break large transactions**
    - Commit in batches instead of one big transaction (e.g., 10,000 rows at a time).
2. **Use retry logic in applications**
    - If deadlock happens, catch error 1205 and retry automatically.
3. **Avoid user interaction inside transactions**
    - Don't pause a transaction to wait for user confirmation.
4. **Access fewer rows**
    - Use WHERE clauses properly.
    - Avoid SELECT * → reduce lock footprint.

✅ D. Locking & Concurrency Control

1. **Row versioning**
   - Enable **RCSI** (Read Committed Snapshot Isolation) at the database level:
   - ALTER DATABASE MyDB SET READ_COMMITTED_SNAPSHOT ON;
   - This allows readers to use row versions instead of blocking writers.
2. **Use NOLOCK carefully**
   - WITH (NOLOCK) avoids shared locks but allows dirty reads.
   - Better approach: RCSI instead of NOLOCK.
3. **Apply UPDLOCK + HOLDLOCK cautiously**
   - Can prevent deadlocks by serializing access — but may hurt concurrency.

3. Monitoring & Troubleshooting Deadlocks

1. **Enable Deadlock Trace Flags**
   - DBCC TRACEON (1222, -1); logs deadlock graphs to SQL Server error log.
2. **Use Extended Events / Profiler**
   - Capture xml_deadlock_report event.
   - Deadlock graphs show victim, resource, and processes involved.
3. **Query DMVs**
   - Recent deadlocks:
   - SELECT * FROM sys.dm_tran_locks;
   - SELECT * FROM sys.dm_exec_requests;
4. **Analyze blocking chains**
   - Use sp_whoisactive or sys.dm_exec_requests to see blockers.

4. Example of Preventing Deadlocks

Problem:

Two transactions:

- Transaction 1 updates TableA then TableB.
- Transaction 2 updates TableB then TableA.
  This creates a cycle → deadlock.

Fix:

- Enforce consistent order: Always update TableA first, then TableB.

5. Summary of Deadlock Prevention

- ✅ Keep transactions short.
- ✅ Access resources in consistent order.
- ✅ Use proper indexing.
- ✅ Enable RCSI for readers.
- ✅ Avoid unnecessary locks.
- ✅ Monitor deadlocks with Extended Events.
- ✅ Implement retry logic at the application level.

⚡ In short:

Deadlocks can't be completely eliminated in high-concurrency systems, but with **query design, indexing, and isolation level tuning**, you can reduce their occurrence significantly.

🟢 SQL Server Deadlock – Real-time Scenarios (Q&A)

Scenario 1: Update Deadlock
Q: Two transactions update two tables but in different orders. Why does a deadlock occur, and how do you fix it?
A:

- **Cause:**
    - Transaction 1: Updates **TableA → TableB**
    - Transaction 2: Updates **TableB → TableA**
    - Both transactions wait on each other → deadlock.
- **Fix:**
    - Access resources in the **same order** across all transactions.
    - Example: Always update TableA before TableB.

Scenario 2: Reader vs Writer Deadlock
Q: A SELECT query (with shared locks) conflicts with an UPDATE query (exclusive lock). How to avoid deadlock?
A:

- **Cause:** A reader holds a shared lock while waiting for another row, and the writer waits for exclusive access.
- **Fix:**
    - Enable **READ COMMITTED SNAPSHOT ISOLATION (RCSI)** to use row versioning:
    - ALTER DATABASE MyDB SET READ_COMMITTED_SNAPSHOT ON;
    - Or use NOLOCK hints (with caution).

Scenario 3: Long Transaction Deadlock
Q: A large transaction updating thousands of rows causes frequent deadlocks. How do you fix it?
A:

- **Fix:**
    - Break large updates into **smaller batches** (e.g., 5000 rows at a time).
    - Commit frequently to release locks.
    - Ensure proper indexes to minimize locking.

Scenario 4: Missing Index Deadlock
Q: Why can missing indexes increase deadlocks?
A:

- **Cause:** SQL Server scans entire tables → acquires more locks → higher contention.
- **Fix:**
    - Create appropriate **covering indexes**.
    - Example: For frequent WHERE OrderDate > '2023-01-01', index on OrderDate.

Scenario 5: Deadlock in Stored Procedures
Q: Two stored procedures deadlock frequently. What's the best practice to avoid this?
A:

- **Fix:**
    - Ensure **procedures access objects in the same order**.
    - Apply consistent locking hints if necessary (UPDLOCK).
    - Keep transactions inside procedures **short** and **predictable**.

Scenario 6: Application-Level Prevention
Q: How should applications handle deadlock errors?
A:

- **Answer:**
    - Always catch SQL error **1205 (deadlock victim)**.
    - Implement **retry logic** (re-execute the transaction after a short delay).
    - Keep retries capped (e.g., 3 attempts).

Scenario 7: Reporting Query Causes Deadlock
Q: A heavy reporting query locks rows and blocks OLTP transactions, causing deadlocks. How do you fix it?
A:

- **Fix:**
  - Move reporting queries to a **read replica** or **read-only secondary** (AlwaysOn AG).
  - Or run reports under **SNAPSHOT isolation**.
  - Optimize reports with **indexed views** or **materialized aggregates**.

Scenario 8: Detecting Deadlocks
Q: How do you capture and analyze deadlocks in SQL Server?
A:

- **Options:**
  1. **Extended Events** → Capture xml_deadlock_report.
  2. **SQL Profiler** (older method).
  3. **Trace Flags**:
  4. DBCC TRACEON (1222, -1);
  5. Use **system_health session** (enabled by default).

Scenario 9: Deadlock Between Update & Select with Hints
Q: An UPDATE uses WITH (UPDLOCK), while a SELECT uses WITH (HOLDLOCK). They deadlock. Why?
A:

- **Cause:** Conflicting lock hints escalate contention.
- **Fix:**
  - Remove unnecessary lock hints.
  - Use proper isolation levels instead of forcing locks.

Scenario 10: Deadlocks in High-Concurrency System
Q: In a high-traffic OLTP system, deadlocks are frequent. What are global strategies?
A:

- Use **RCSI** to minimize read locks.
- Keep **transactions short**.
- Optimize schema with **proper indexing**.
- Ensure **consistent access patterns**.
- Add **retry logic** in the application.
- Monitor with **Extended Events** and fix hotspots.

✅ **Summary:**
Deadlocks are not always avoidable, but they can be **minimized** by:

- Designing transactions and queries carefully.
- Using proper indexes and isolation levels.
- Monitoring deadlock graphs.
- Adding retry logic at the application layer.