VISVESVARAYA TECHNOLOGICAL UNIVERSITY

Jnana Sangama, Belagavi-590014

Fundamentals of

Operating System Report

ON

The Deadly Embrace in Databases – The Microsoft SQL Server Deadlock Case

Submitted in Partial fulfillment of the Requirements for 6th Semester

Bachelor of Engineering

in

Electronics and Communication Engineering

By

AASHISH ARYAN N S	**1KG22EC002**
ABHINANDAN S	**1KG22EC003**
ANVITHA M SHETTY	**1KG22EC009**
ASFIYA KOUSAR B I	**1KG22EC010**
GOWTHAMI B	**1KG22EC011**
BEZAWADA SUCHARITHA	**1KG22EC012**

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING K.S. SCHOOL OF ENGINEERING AND MANAGEMENT BENGALURU-560109 2024-25

The Deadly Embrace in Databases – The Microsoft SQL Server Deadlock Case

Introduction

In database systems, the concept of deadlock is a critical issue in concurrent transaction processing. A deadlock, also known as a deadly embrace, occurs when two or more transactions are each waiting for the other to release a resource, resulting in a cycle of dependencies and indefinite waiting. This phenomenon is analogous to a traffic gridlock, where each vehicle blocks the path of another, and none can proceed.

Microsoft SQL Server, a widely used relational database management system (RDBMS), has robust transaction management and concurrency control mechanisms. However, even in such systems, deadlocks can occur, especially under high concurrency with complex transactions accessing multiple shared resources.

Understanding Deadlocks in Databases

A deadlock arises in SQL Server when the following four conditions hold simultaneously:

1. Mutual Exclusion: At least one resource must be held in a non-shareable mode. Only one transaction can use the resource at any one time.
2. Hold and Wait: A transaction holding at least one resource is waiting to acquire additional resources held by other transactions.
3. No Preemption: A resource cannot be forcibly removed from a transaction holding it. The transaction must release it voluntarily.
4. Circular Wait: A set of transactions exist such that each transaction is waiting for a resource held by the next transaction in the set, forming a cycle.

These conditions create a scenario where progress halts unless the system intervenes.

The Deadly Embrace in Databases – The Microsoft SQL Server Deadlock Case

Scenario:

During a high-traffic sales event, a mid-sized e-commerce platform using Microsoft SQL Server encountered a critical deadlock involving two transactional modules — Order Processing and Inventory Management. These modules executed concurrently to ensure real-time updates of customer orders and product stock.

Transactional Conflict

- Transaction A, initiated by the Order Processing system, locked a row in the Orders table to record a new purchase. It then attempted to update the Inventory table to decrease the stock count.
- Transaction B, launched by the Inventory Replenishment system, locked a row in the Inventory table to register incoming stock and then attempted to update the Orders table to fulfill pending orders.

This created a circular wait: each transaction held a lock the other required. Neither could proceed, resulting in a deadlock. SQL Server's deadlock detection mechanism constructed a wait-for graph, identified the cyclic dependency, and selected Transaction B as the victim due to its lower rollback cost. It was terminated and rolled back, while Transaction A completed successfully. Although recovery was automatic, the incident caused temporary delays and highlighted risks to data consistency during peak operations.

C++ Code: Simulating a Deadlock with SQL Server

SQL SETUP

```
CREATE TABLE TableA (id INT PRIMARY KEY, value VARCHAR(50));
CREATE TABLE TableB (id INT PRIMARY KEY, value VARCHAR(50));

INSERT INTO TableA VALUES (1, 'A1');
INSERT INTO TableB VALUES (1, 'B1');
```

C++ CODE

```
#include <windows.h>
#include <sql.h>
#include <sqlext.h>
#include <iostream>
#include <thread>

void printSQLError(SQLSMALLINT handleType, SQLHANDLE handle) {
   SQLCHAR sqlState[1024];
   SQLCHAR message[1024];
   SQLINTEGER nativeError;
   SQLSMALLINT textLength;

   if (SQLGetDiagRecA(handleType, handle, 1, sqlState, &nativeError, message,
sizeof(message), &textLength) == SQL_SUCCESS) {
      std::cerr << "[SQL ERROR] State: " << sqlState << ", Message: " << message << ", Native
Error: " << nativeError << std::endl;
   }
}

void executeTransaction(const char* connStr, bool firstLocksA, int retryCount = 1) {
   int attempts = 0;
   bool success = false;
 while (attempts <= retryCount && !success) {
 attempts++;
 SQLHENV hEnv;
 SQLHDBC hDbc;
 SQLHSTMT hStmt;
 SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &hEnv);
 SQLSetEnvAttr(hEnv, SQL_ATTR_ODBC_VERSION, (void*)SQL_OV_ODBC3, 0);
 SQLAllocHandle(SQL_HANDLE_DBC, hEnv, &hDbc);
```

```
      if (SQLDriverConnectA(hDbc, NULL, (SQLCHAR*)connStr, SQL_NTS, NULL, 0,
NULL, SQL_DRIVER_COMPLETE) == SQL_SUCCESS) {
          SQLAllocHandle(SQL_HANDLE_STMT, hDbc, &hStmt);
          SQLExecDirectA(hStmt, (SQLCHAR*)"BEGIN TRAN", SQL_NTS);

          SQLRETURN ret1, ret2;
          if (firstLocksA) {
              ret1 = SQLExecDirectA(hStmt, (SQLCHAR*)"UPDATE TableA SET value =
'ThreadA' WHERE id = 1", SQL_NTS);
              Sleep(1000); // simulate timing gap
              ret2 = SQLExecDirectA(hStmt, (SQLCHAR*)"UPDATE TableB SET value =
'ThreadA' WHERE id = 1", SQL_NTS);
          } else {
              ret1 = SQLExecDirectA(hStmt, (SQLCHAR*)"UPDATE TableB SET value =
'ThreadB' WHERE id = 1", SQL_NTS);
              Sleep(1000); // simulate timing gap
              ret2 = SQLExecDirectA(hStmt, (SQLCHAR*)"UPDATE TableA SET value =
'ThreadB' WHERE id = 1", SQL_NTS);
          }

          if (SQL_SUCCEEDED(ret1) && SQL_SUCCEEDED(ret2)) {
              SQLExecDirectA(hStmt, (SQLCHAR*)"COMMIT TRAN", SQL_NTS);
              std::cout << "[INFO] Transaction completed successfully on attempt " << attempts
<< std::endl;
              success = true;
          } else {
              SQLExecDirectA(hStmt, (SQLCHAR*)"ROLLBACK TRAN", SQL_NTS);
              std::cerr << "[WARNING] Transaction rolled back due to error on attempt " <<
attempts << std::endl;
              printSQLError(SQL_HANDLE_STMT, hStmt);
          }
SQLFreeHandle(SQL_HANDLE_STMT, hStmt);
 SQLDisconnect(hDbc);
 } else {
 std::cerr << "[ERROR] Database connection failed on attempt " << attempts << std::endl;
 printSQLError(SQL_HANDLE_DBC, hDbc);
 }

 SQLFreeHandle(SQL_HANDLE_DBC, hDbc);
 SQLFreeHandle(SQL_HANDLE_ENV, hEnv);
```

```
        if (!success && attempts <= retryCount) {
            std::cout << "[RETRY] Retrying transaction..." << std::endl;
            Sleep(500); // Wait before retrying
        }
    }}

    if (!success) {
        std::cerr << "[FATAL] Transaction failed after " << attempts << " attempts." << std::endl;
    }
}

int main() {
    const char* connStr = "DRIVER={SQL
Server};SERVER=localhost;DATABASE=TestDB;Trusted_Connection=yes;";
    std::thread t1(executeTransaction, connStr, true, 1);
    std::thread t2(executeTransaction, connStr, false, 1);

    t1.join();
    t2.join();

    std::cout << "Execution complete. Check SQL Server logs if deadlock occurred." <<
std::endl;
    return 0;
}
```

OUTPUT :

[INFO] Transaction completed successfully on attempt 1
[WARNING] Transaction rolled back due to error on attempt 1
[SQL ERROR] State: 40001, Message: Transaction (Process ID 55) was deadlocked..., Native Error: 1205
[RETRY] Retrying transaction...
[INFO] Transaction completed successfully on attempt 2
Execution complete. Check SQL Server logs if deadlock occurred.

Conditions for Deadlock (The Coffman Conditions)

Deadlocks arise when four conditions occur simultaneously:

a) Mutual Exclusion

Each resource can be assigned to only one transaction at a time. For example, if Transaction A locks Row X, other transactions must wait.

b) Hold and Wait

A transaction is holding at least one resource and waiting to acquire additional resources held by others. This is a common occurrence in complex SQL queries.

c) No Preemption

Resources cannot be forcibly taken from a transaction; they must be released voluntarily. SQL Server does not preemptively unlock resources.

d) Circular Wait

A closed chain of transactions exists, where each transaction holds a resource the next one needs. This forms a cycle, the fundamental structure of a deadlock.

These conditions define the theoretical model under which deadlocks occur. If one of these conditions can be broken, a deadlock can be prevented.

Root Causes of SQL Server Deadlocks

Deadlocks in SQL Server are often the result of systemic inefficiencies or design flaws in transaction management and resource allocation. Understanding the underlying causes is essential for designing robust, deadlock-resistant applications. Below are the primary root causes:

a) Inconsistent Lock Ordering

A major contributor to deadlocks is the inconsistent ordering in which transactions acquire locks on shared resources. When transactions access tables or rows in different sequences, they create the possibility for circular waits.

Conditions for Deadlock (The Coffman Conditions)

Deadlocks arise when four conditions occur simultaneously:

a) Mutual Exclusion

Each resource can be assigned to only one transaction at a time. For example, if Transaction A locks Row X, other transactions must wait.

b) Hold and Wait

A transaction is holding at least one resource and waiting to acquire additional resources held by others. This is a common occurrence in complex SQL queries.

c) No Preemption

Resources cannot be forcibly taken from a transaction; they must be released voluntarily. SQL Server does not preemptively unlock resources.

d) Circular Wait

A closed chain of transactions exists, where each transaction holds a resource the next one needs. This forms a cycle, the fundamental structure of a deadlock.

These conditions define the theoretical model under which deadlocks occur. If one of these conditions can be broken, a deadlock can be prevented.

Root Causes of SQL Server Deadlocks

Deadlocks in SQL Server are often the result of systemic inefficiencies or design flaws in transaction management and resource allocation. Understanding the underlying causes is essential for designing robust, deadlock-resistant applications. Below are the primary root causes:

a) Inconsistent Lock Ordering

A major contributor to deadlocks is the inconsistent ordering in which transactions acquire locks on shared resources. When transactions access tables or rows in different sequences, they create the possibility for circular waits.

b) Long-Running Transactions

Transactions that perform many operations or involve user interaction (e.g., waiting for input) tend to hold locks for extended durations. The longer a transaction holds a lock, the more likely it is to block others, increasing the chance of deadlock.

Long-running transactions may also cause lock escalation—from row/page level to table-level—further amplifying contention. Best practice is to keep transactions short and efficient, avoiding unnecessary delays between BEGIN and COMMIT.

c) Poor Indexing

Poorly indexed tables often require full table scans to satisfy queries, especially for JOIN or WHERE clauses. These scans lock more rows (or even entire tables), which significantly broadens the locking footprint.

This increases the probability of two transactions locking overlapping data ranges. Proper index design can reduce lock contention by narrowing scan ranges and enabling efficient lookups.

d) High Isolation Levels

SQL Server supports various isolation levels to manage concurrency. While high levels like SERIALIZABLE offer maximum consistency, they also retain locks for the full duration of a transaction, including read locks.

For example, SERIALIZABLE prevents phantom reads by locking entire ranges of data. This prolonged and expanded locking behavior amplifies contention, increasing the risk of deadlocks.

e) Blocked Resources

A blocking chain is a sequence of transactions where each one waits for the resource held by the next. When such a chain forms a cycle—where the last transaction waits for the first—a deadlock occurs.

Resolutions and Mitigation Strategies

Short-Term (Reactive) Strategies

- Deadlock Retry Logic: Automatically retrying the transaction after a rollback.
- Transaction Scope Minimization: Keeping transactions as short as possible.
- Monitoring Tools: Use Extended Events, SQL Profiler, and system health sessions to capture deadlock graphs.

Long-Term (Preventive) Strategies

1. Consistent Resource Access Order:
2. Ensuring all transactions access tables and rows in the same sequence.
3. Index Optimization:
4. Reducing full-table scans and narrowing lock scopes using selective indexes.
5. Appropriate Isolation Levels:
6. Using READ COMMITTED SNAPSHOT isolation to enable optimistic concurrency.
7. Partitioning and Row-Level Locking:
8. Reducing contention by segmenting large tables and minimizing lock scope.
9. Application Logic Design:
10. Avoid nested transactions, ensure fast commit, and limit resource locks in critical code paths.

Key Learnings

The SQL Server deadlock scenario serves as a microcosm of how resource management and scheduling in operating systems are critical for stability and performance.

- Deadlocks in databases are analogous to deadlocks in OS where threads compete for CPU, memory, or I/O devices.
- Just as an OS uses a resource allocation graph to detect cycles, SQL Server uses a wait-for graph.
- SQL Server's automatic deadlock resolution is akin to an OS killing or rolling back a process to break the deadlock.

Conclusion

Deadlocks, or deadly embraces, pose a significant challenge in high-performance database systems. The SQL Server deadlock case illustrates how even advanced systems can suffer performance degradation and data processing issues without careful design.

Understanding the conditions and causes of deadlocks, coupled with real-time detection and robust prevention strategies, is critical for developers, DBAs, and system architects. This case reinforces the importance of concurrency control, efficient transaction design, and resource-aware programming—foundational principles in both database and operating system domains