Detecting and Eliminating Deadlocks from a SQL Anywhere Database

Purpose

The purpose of this page is to discuss how to identify and deal with deadlocks in a relational database.

Overview

The term ‘deadlock’ refers to what is known as a ‘cyclical deadlock’: a situation wherein two or more competing transactions are blocking each other on the same resource (usually table or row lock) and thus since neither are able to proceed; one of them must be terminated by the server.

Depending on the situation, there might be a couple different approaches to resolving deadlocks in the database. Each approach is based on the information available, individual’s experience or the preference. Some administrators might be more comfortable modifying database schema, others would rather change SQL code or server/ connection options.

The difficulty with deadlocks is that they rarely occur in test or QA environments, therefore they are harder to discover early in the development cycle. They typically occur under heavy production load, where it’s harder to debug and not always possible to make any code changes.

There are a number of things that need to be done in order to avoid deadlocks on the database:

- Logging deadlock information
- Identifying SQL involved
- Making sure queries are optimized for performance
- Keeping transactions short
- Revising your general performance

It is important to remember that deadlocks are just symptoms of poor database design, poor SQL coding or other problems that are hiding in the system. Addressing deadlock issues will not only make an application more stable, but it may also improve overall database performance.

Note that although in this example SQL Anywhere is used as the database server, in principle similar techniques could be applied on all other relational database systems including SAP HANA, SAP ASE, SAP IQ, Microsoft SQL Server, Oracle, etc.

How to Identify if Deadlocks are Affecting an Application

Deadlocks could be sporadic (happen once and then never again) or repetitive (happening at a certain time of the day or during one specific report/procedure call). A database administrator needs to develop a judgment to know which ones to ignore and which ones to pay attention to. It is important to mention that the sole reason for their occurrence lies in the application and database design.

If your application occasionally behaves abnormally, e.g. most of the time everything works perfectly but sometimes, even though nothing has changed, transactions are rolled back, scripts are failing, or your application is simply returning the following error:

```
SQLCODE=-306, ODBC 3 State="40001"
```

then, it’s time to perform some quick analysis and verify what is causing deadlocks in your database.

Logging Deadlock Information

SAP SQL Anywhere provides deadlock logging capabilities that allows users to captures valuable information needed for deadlock resolution. Deadlock logging is not turned on by default. An administrator needs to enable this either through Sybase Central or Interactive SQL. In addition to deadlock logging, the user should also turn the database option ‘RememberLastStatement’ on in order to track down which SQL statement is being executed during deadlocks.

In order to turn the required options on, run the following from Interactive SQL:

```
// make sure default blocking_timeout has not been modified
SET OPTION PUBLIC.blocking_timeout = 0;
// set server option to remember LastStatement
SET OPTION PUBLIC.rememberlaststatement = 'On';
// remember information about deadlocks
SET OPTION PUBLIC.log_deadlocks = 'On';
```

Once the required options are set, the server will temporarily keep all of the information in memory until a restart, or it is manually cleared.
Deadlock Example

In order to capture a simple deadlock, use the following example. Open two dbisql windows ('Connection 1' and 'Connection 2') and connect to the demo16 database. From 'Connection 1', run the following and it will create a deadlock_example table and two procedures that are updating and selecting from the table:

```sql
// SQL Anywhere 16 create sample table
CREATE TABLE IF NOT EXISTS deadlock_example
(
    pk INT NOT NULL PRIMARY KEY,
    s1 VARCHAR(30) NOT NULL
);

CREATE OR REPLACE PROCEDURE proc1()
BEGIN // make sure isolation level is set to 1
    SET TEMPORARY OPTION isolation_level = 1;
    // clean the table first
    DELETE deadlock_example;
    // populate sample table with two rows
    INSERT INTO deadlock_example (pk, s1) VALUES (1, 'initial 1');
    INSERT INTO deadlock_example (pk, s1) VALUES (2, 'initial 2');
    COMMIT;

    // update the table, which creates a read lock on the row
    UPDATE deadlock_example SET s1 = 'deadlock 1' WHERE pk = 1;
    WAITFOR DELAY '00:00:10';
    SELECT * FROM deadlock_example WHERE pk = 2;
    COMMIT;
END;

CREATE OR REPLACE PROCEDURE proc2()
BEGIN // make sure you are in isolation level 1
    SET TEMPORARY OPTION isolation_level = 1;
    // lock row with pk = 2
    UPDATE deadlock_example SET s1='deadlock 2' WHERE pk=2;
    // try to read row with pk = 1
    SELECT * FROM deadlock_example WHERE pk = 1;
    COMMIT;
END;
```

Just like in a real environment, this example shows how much a deadlock is a timing dependent event. The WAITFOR DELAY call simulates long running transactions on the database, during which deadlock_example rows are locked.

Now, from 'Connection 1', run:

```sql
CALL proc1();
```

and then quickly (< 10 seconds) run the following from 'Connection 2':

```sql
CALL proc2();
```

Shortly after you execute `proc2()`, an error will occur. When you click on 'Show Details', you should see the following message indicate that a deadlock has been detected.
To view deadlock information, SQL Anywhere provides the `sa_report_deadlocks()` system stored procedure. Here is a sample output from `dbisql` call:

```sql
SELECT * FROM sa_report_deadlocks();
```

<table>
<thead>
<tr>
<th>snapshotId</th>
<th>snapshotAt</th>
<th>waiter</th>
<th>who</th>
<th>object_id</th>
<th>record_id</th>
<th>owner</th>
<th>is_victim</th>
<th>rollback_operation_count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2014-06-12 14:56:13.298</td>
<td>2</td>
<td>DBA</td>
<td>null</td>
<td>3591</td>
<td>null</td>
<td>false</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2014-06-12 14:56:13.298</td>
<td>1</td>
<td>DBA</td>
<td>null</td>
<td>3591</td>
<td>null</td>
<td>true</td>
<td>1</td>
</tr>
</tbody>
</table>

Other than the information from `sa_report_deadlocks()`, a visual representation of deadlocks can be found in Sybase Central. Just click on the database icon and choose the Deadlocks tab.

As per the definition of deadlock, there needs to be at least two different transactions involved for a deadlock to happen. The likelihood of a deadlock to appear increases with the duration of a transaction and the number of tables (locks) involved. In other words, in order to eliminate them, it is important to keep transactions short, and the number of tables/locks accessed during the transaction small.
Very often, long-running transactions are caused by poorly written queries or bad indexes. In many cases, speeding them up would not only fix the performance bottleneck but also resolve deadlocks.

The main idea to resolve deadlocks is to change the locking behavior of one of the transactions involved so it still produces the same results, but behaves differently as far as locking is concerned.

Changing locking behavior can be done in a couple different ways. For example:

- Modifying existing SQL code (e.g. changing isolation level NOLOCK hint, reordering statements)
- Modifying existing database schema (modifying, adding, removing indices)

Depending on the environment, different solutions could be applied. In situations where SQL code is embedded within a compiled application, the only option might be to change the schema (e.g. adding a new index to speed up the transaction). On other systems, where changing SQL is not a problem, a simple rewrite of a stored procedure may be the way to go.

In the example above, decreasing the number of seconds in a WAITFOR command, or changing the isolation level to '0' will either make the deadlock less likely to appear or remove it complete, as follows:

```
SELECT * FROM deadlock_example WITH (NOLOCK) WHERE pk =2
```