The how and why of locking in DB2 is one of its least understood features. This article walks through an example of a common locking scenario and explains what locks are held and why.

Introduction

One of the hot topics we see in customer support is locking. "Why does DB2 lock this table, row, or object?" "How long is the lock held, and why?" "Why did I get a deadlock?" "What is my lock request waiting for?" Looking in closer detail at some common locking examples can help illustrate the principles behind DB2's locking strategy.

The following is a common misunderstanding of how DB2 accesses and locks rows. The problem arose when two users inserted into the same table, then the first user attempted to select the row he had just inserted. He ended up in a lock wait. This was unexpected -- either the user was waiting on his own lock, which is impossible, or he was waiting on the unrelated insert, which didn't seem to make sense. Either way, the lock wait was a surprise, and the user did not know how to figure out what was going on.

Let's walk through how you would analyze the locking in this situation.

The problem

Imagine that you have the following two tables in your database named LOCKTEST:
CREATE TABLE MAIN_TABLE(
    MAIN_JOIN_COLUMN VARCHAR(10) NOT NULL ,
    MAIN_DATA_COLUMN VARCHAR(20) NOT NULL )
IN USERSPACE1 ;

ALTER TABLE MAIN_TABLE
    ADD CONSTRAINT MAINPKEY PRIMARY KEY
    (MAIN_JOIN_COLUMN,
    MAIN_DATA_COLUMN);

CREATE TABLE CHILD_TABLE ( 
    CHILD_JOIN_COLUMN VARCHAR(10) NOT NULL ,
    CHILD_DATA_COLUMN VARCHAR(20) NOT NULL )
IN USERSPACE1 ;

ALTER TABLE CHILD_TABLE
    ADD CONSTRAINT CHILDPKEY PRIMARY KEY
    (CHILD_JOIN_COLUMN);

And that these tables are initialized with these data:

<table>
<thead>
<tr>
<th>MAIN_JOIN_COLUMN</th>
<th>MAIN_DATA_COLUMN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>some existing data 1</td>
</tr>
<tr>
<td>2</td>
<td>some existing data 2</td>
</tr>
<tr>
<td>3</td>
<td>some existing data 3</td>
</tr>
<tr>
<td>4</td>
<td>some existing data 4</td>
</tr>
<tr>
<td>5</td>
<td>some existing data 5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHILD_JOIN_COLUMN</th>
<th>CHILD_DATA_COLUMN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ONE</td>
</tr>
<tr>
<td>2</td>
<td>TWO</td>
</tr>
<tr>
<td>3</td>
<td>THREE</td>
</tr>
<tr>
<td>4</td>
<td>FOUR</td>
</tr>
<tr>
<td>5</td>
<td>FIVE</td>
</tr>
</tbody>
</table>

The following series of operations produces a lock wait that seems surprising:

1. From one session, without committing:
   ```sql
   insert into main_table values ('1', 'deadlock 1')
   ```
2. From a second session, without committing:
   ```sql
   insert into main_table values ('1', 'deadlock 2')
   ```
3. From the first session, the following select statement:
   ```sql
   select main_table.main_join_column, child_table.child_data_column
   from main_table, child_table
   where main_table.main_join_column = child_table.child_join_column
   and main_table.main_data_column = 'deadlock 1'
   ```

   The select does not complete; it appears to be waiting on a lock. So it must be waiting for the insert from the second session to commit. But you're not selecting that row. What's going on?

Since there is a primary key on both columns of the main_table, and the select joins on the first column and has a predicate on the second, you might expect that DB2 would fetch the one row
that qualifies from the main_table and would therefore not be held up by the lock from the second session. But DB2 doesn't necessarily work that way.

The answer

To understand what's going on, let's look at the query and the access plan. To find out what locks are held when the lock wait occurs, use the db2pd utility. For easier reading, the following db2pd output has been modified to remove all but the row locks.

```
/home/lorysa $db2pd -db locktest -locks show detail

Locks:
Address TranHdl Lockname Type Mode Sts
Owner Dur HldCnt Att Rlse
0x402C07E0 3 000200020000000A0000000052 Row ..X G
0x402C02E0 2 00020002000000090000000052 Row .NS W
0x402C03A8 2 00020002000000090000000052 Row ..X G
0x402C07E0 3 000200020000000A0000000052 Row .NS W
```

You can see that DB2 is holding locks on the table with tableid 2 in the tablespace with tbspaceid 2. Now to find out which table that is:

```
/home/lorysa $db2 "select substr(tabschema,1,9) as tabschema,
    substr(tabname,1,12) as tabname, tableid, tbspaceid
    from syscat.tables
    where tbspaceid = 2 and tableid = 2"

<table>
<thead>
<tr>
<th>TABSCHEMA</th>
<th>TABNAME</th>
<th>TABLEID</th>
<th>TBSPACEID</th>
</tr>
</thead>
<tbody>
<tr>
<td>LORYSA</td>
<td>MAIN_TABLE</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
```

The db2pd output above provides the record identifier (RID) of the rows that are locked. The value 0xA really means 0x0000000A; an RID is a four-character field made up of a three-character page number (0 here) and a one-character slot identifier (0xA or 10 here). This tells us the row that we are interested in is on page 0 of the table, in slot 10. Each data page has up to 255 "slots" that hold the offset into the page where a given row can be found. RIDs are often depicted as (page number; slot number), or (0;10) in decimal notation, (0;A) in hexadecimal notation. The RID uniquely identifies a row within a table.

The output indicates that you are waiting on the lock on the row (0;A) from the db2pd output because the 'Sts' column, which provides the status of the lock request, shows a W for waiting. The other locks have a status of G for granted, so they are held.

So, summarizing the locks of interest:

- agent with TranHdl 2: X lock GRANTED on main table row (0;9) (from uncommitted insert)
- agent with TranHdl 3: X lock GRANTED on main table row (0;A) (from uncommitted insert)
- agent with TranHdl 2: NS lock WAITING on main table row (0;A) (from select)
The row that the agent running the select is waiting on is the one with value ('1', 'deadlock 2'), not the row it just inserted. You can deduce that because the inserted row is the one with an X lock on it -- newly inserted rows are locked exclusively until the insert is committed. Also, an agent cannot wait on itself; if an agent requests a share (NS) lock on a row it is already holding an exclusive (X) lock on, the request is granted because a lock of sufficient or higher mode is already held. So the question remains: if you are selecting the row with value deadlock 1, why are you waiting on the newly inserted row with value deadlock 2? To understand why, you have to look at the access plan for the select.

```sql
select main_table.main_join_column, child_table.child_data_column
from main_table, child_table
where main_table.main_join_column = child_table.child_join_column
  and main_table.main_data_column = 'deadlock 1'
```

The access plan is basically the roadmap of how DB2 is going to access the data required to satisfy the query. You can generate the access plan using DB2's `explain` facility. Use the following command to create the tables that hold the results of the `explain`:

```
db2 -tvf EXPLAIN.DDL
```

The `EXPLAIN.DDL` file, which is found in the `sqllib` directory under the instance home directory, contains the DDL statements necessary to create the explain tables. Then the statement can be explained by:

```
db2 explain all for select statement
```

and the access plan generated by the `db2exfmt` tool.

For more information about the specifics of `explain` and `db2exfmt`, see the DB2 Command Reference and the DB2 SQL Reference, volume 1.
Aha. The optimizer, in the absence of current statistics, is choosing to do a table scan on both tables involved in the join. So to find the row that satisfies the predicate in the select statement, the optimizer has to lock, read in, and compare all the rows in the main table with the value supplied in the predicate. Table scans always read (and lock) all the rows in the table. In the explain (db2exfmt) output, you can see the sargable predicate on the table scan numbered 4 in the plan:

```
Predicates:  
-------------
2) Sargable Predicate
   Relational Operator:       Equal (=)
   Subquery Input Required:   No
   Filter Factor:          0.04

Predicate Text:  
----------------
(Q2.MAIN_DATA_COLUMN = 'deadlock 1')
```

So all 7 rows are read (and first locked) from the main table, the value in the data_column is compared to deadlock 1, and the rows that match are passed up into the join.

Now let's do a runstats on both tables to see how that affects the plan. And this is where things take an interesting turn. The lock wait still happens, but with a slight twist:

```
/home/lorysa $db2pd -db locktest -locks show detail
Locks:  
Adresse  TranHdl  Lockname                   Type       Mode Sts
Owner     Dur HldCnt     Att Rlse
0x402C07E0 3          000200020000000A0000000052 Row        ..X  G
3          1   0          8   0x40 TbspaceID 2 TableID 2 RecordID 0xA
0x402C02E0 2          000200020000000A0000000052 Row        .NS  W
3          1   0          0   0x0  TbspaceID 2 TableID 2 RecordID 0xA
0x402C06A0 2          00020003000000040000000052 Row        ..X  G
2          1   0          0   0x0  TbspaceID 2 TableID 2 RecordID 0x4
0x402C03A8 2          00020002000000090000000052 Row        ..X  G
2          1   0          8   0x40 TbspaceID 2 TableID 2 RecordID 0x9
```

Summarizing as before:

- agent 2: X lock GRANTED on main table row ID (0;9) (from uncommitted insert)
- agent 3: X lock GRANTED on main table row ID (0;A) (from uncommitted insert)
- agent 2: NS lock WAITING on child table row ID (0.4) (from select)
- agent 2: NS lock WAITING on main table row ID (0;A) (from select)

The access plan here is:

```
| Total Cost:       | 17.0258 |
| Query Degree:     | 1      |
| Rows RETURN       | (1)    |
```
Here we are using the two indexes, and we have the same predicate, this time on the index scan numbered 3 in the plan:

```
Predicates:
-------------
2) Sargable Predicate
   Relational Operator:       Equal (=)
   Subquery Input Required:   No
   Filter Factor:         0.2

Predicate Text:
---------------
(Q2.MAIN_DATA_COLUMN = 'deadlock 1')
```

Note the exact same predicate. That's the key. This is a sargable predicate, but not an index-sargable predicate (that is, it cannot be applied as a start/stop key on the index scan). This is because it is on the second column of the index.

An index sarg, or start/stop key, will limit the number of rows returned by an index scan. It means: Start the scan at value X and stop it at value Y, returning only the keys with values between X and Y inclusive. Without an index sarg, an index scan will use the index to access the data, but will return every key in the index.

To illustrate this, compare it to the predicate that is applied to the index scan numbered 5 in the plan, which is on the one and only column of the index on the child table:
Because the predicate on the main table cannot be applied to the index, we have to read all the rows from the table, using the index, and then apply the predicate before passing the qualifying rows up to the join. And as before, we have to lock the rows before we can evaluate the predicate.

**Ways to avoid the lock wait**

Now that you understand what is going on, there are steps you can take to improve the concurrency:

1. Reverse the order of columns in the primary key on the main table (so MAIN_DATA_COLUMN first, then MAIN_JOIN_COLUMN). This will allow the predicate to be applied to the index.
2. Check out the DB2_EVALUNCOMMITTED registry variable. This is a setting that allows DB2 to evaluate sargable predicates without first locking the row under the CS or RS isolation levels, so the row will not be locked until after we determine it satisfies the predicate. However, accessing unlocked data can have side effects which may not be acceptable to everyone (for example, this registry setting alters the isolation level) so it is important to understand this feature before using it.
3. Check out the DB2_SKIPINSERTED registry variable. This variable controls whether uncommitted inserts can be ignored for cursors under the CS or RS isolation levels. Enabling this variable has the effect of treating uncommitted inserts as if they haven’t been inserted at all. Once again, this may or may not be acceptable behavior, so it is important to understand the implications.

**Conclusion**

In walking through this example, you have seen how different tools -- db2pd, SQL, explain, db2exfmt -- can shed some light on the situation. The information collected from these tools allows you to understand why the locks are being held, and this in turn can help you determine strategies and techniques to avoid undesirable locking. Armed with this information, anyone who has a need to understand locking can apply the techniques and principles used here to their own scenario and produce a similar analysis and recommendations.
Related topics

- **DB2 Command Reference**: Learn more about the db2exfmt tool.
- **DB2 SQL Reference, volume 1**: See appendix I to learn more about the DB2 explain facility.
- **DB2 V8 Information Center**: Read about the DB2_EVALUNCOMMITTED feature of DB2.
- **DB2 V8 Information Center**: Read about the DB2_SKIPINSERTED feature of DB2.
- **"The db2pd tool"** (developerWorks, April 2005) explains how to use the db2pd utility for monitoring DB2 databases and instances.
- **"Tips for improving INSERT performance in DB2 Universal Database"** (developerWorks, March 2004) talks about locking along with other factors that affect insert performance.
- The article **"Lock avoidance in DB2 UDB V8"** (developerWorks, September 2005) explains DB2 concurrency basics and details how you can reduce locking in some situations using new registry variables.
- Visit the [developerWorks DB2 UDB for Linux, UNIX, and Windows page](http://www.ibm.com/developerWorks/db2) for resources to expand your DB2 UDB skills.
- Download a free trial version of **DB2 Universal Database Enterprise Server Edition**.
- Build your next development project with [IBM trial software](http://www.ibm.com/developerWorks.ibm.com/trial/software/), available for download directly from developerWorks.

© Copyright IBM Corporation 2005

[Trademarks](http://www.ibm.com/developerworks/ibm/trademarks/)