DB2 Data Management Software

IBM

e business software

DB2 UDB Universal Database Version 7

Concurrency and Locking

Sven Noltemeier
IBM Deutschland Entwicklung GmbH

IBM Software Group

Agenda

- Concurrency problems
- Isolation with locking as the solution
- Isolation levels in DB2
- Locking in DB2
- Deadlock detection and time-outs
- Summary

Concurrency

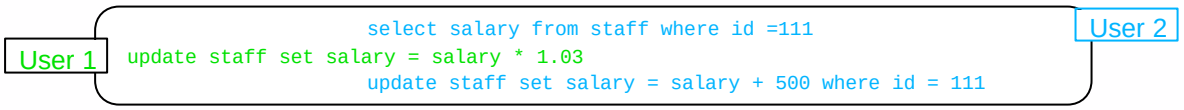
- Many applications (users) work on the same database at the same time ("concurrently")
- They perform logical units of work (transactions) with atomicity ("all or nothing" behavior): COMMIT or ROLLBACK
- Typically, two conflicting objectives:
  - As much concurrent access to data as possible
  - As much data integrity as necessary for the application.
- Problems with applications concurrently working on the same data ...

Concurrency Problems

- Lost update
- Uncommitted read
- Non-repeatable read
- Phantom read

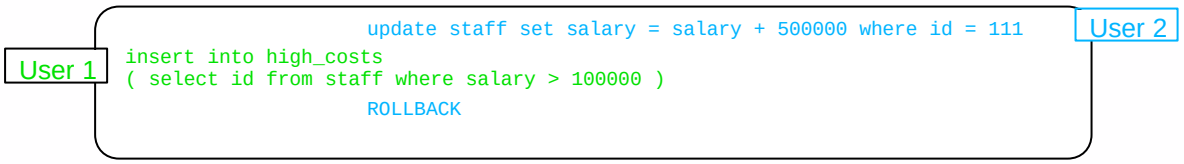
Lost Update

- Occurs when:
  - The same data is retrieved by two applications (users)
  - Both work with the data concurrently
  - Both change and save the data
- The last successful change to the data will be kept
- The first change will be overwritten
- Example:



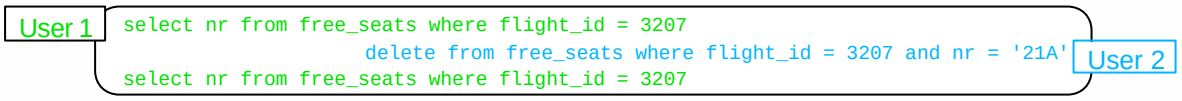
Uncommitted Read ("Dirty Read")

- Occurs when:
  - Uncommitted changes to data are read by an application (user)
  - The changes are rolled back
- The reading application gets invalid data
- Example:



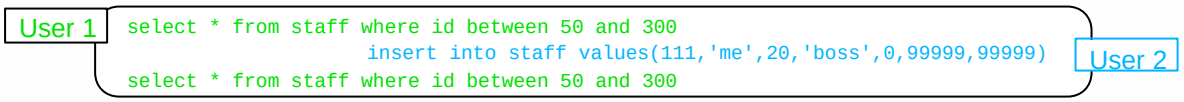
Non-Repeatable Read

- Occurs when:
  - An application reads a query result
  - Later on, in the same transaction, the same query yields a different result (some rows have vanished)
- Query results are not repeatable, one can't rely on them
- Reason: rows in the result set were updated or deleted by someone else
- Example:



Phantom Read

- Occurs when:
  - An application reads a query result
  - Later in the same transaction, the same query yields a different result (some additional rows appear like a phantom)
- Reason: rows qualifying for the result set were inserted or updated by someone else
- Acceptable for many applications
- Example:



Solution to Concurrency Problems

- Isolate your application from concurrent application by locking the data your application reads or modifies
- DB2 uses row-level locking by default.
- Rationale of locking:
  - Allow as much concurrent access to data as possible and at the same time
  - Guarantee as much data integrity as necessary.

Isolating Concurrent Applications

- Through explicit locking
  - Only two possibilities:
    - SELECT ... FOR UPDATE
    - LOCK TABLE ...
  - The user or application programmer is responsible
- Through implicit locking with isolation levels
  - More possibilities (4 isolation levels)
  - DB2 is responsible

Isolation Level

- Relevant when an application reads data.
- Isolation protects the data you read from someone else updating it.
- Isolation level decides:
  - **How much** of the data you read is protected
  - **How long** the data you read is protected
- No impact on the locks and the lock duration required for changing data through UPDATEs, INSERTs, DELETEs.

Isolation Levels

How much data integrity do you need?
Choose the appropriate isolation level...

Isolation Level Comparison

- Repeatable Read (RR):
  - No change of rows retrieved until the LUW ends.
    - This can be much more than what is returned to your application!
  - No change of answer set until the LUW ends.
  - This means that no rows can be added to the answer set while the LUW is still active.
- Read Stability (RS):
  - No change of rows which have been read until the LUW ends.
  - The answer set can change (grow)!
- Cursor Stability (CS):
  - The row to which the cursor points cannot be changed.
    - Careful: Do you really know to which row your cursor points?
- Uncommitted Read (UR):
  - Committed and uncommitted data can be read with read-only cursors. Other cursors: same as CS!
  - Data which has been read can be changed.

Repeatable Read vs. Read Stability

Do you need repeatable results?

User 1

```
select * from staff where id between 50 and 300
                    insert into staff values(111,'me',20,'boss',0,99999,99999)
select * from staff where id between 50 and 300
```

User 2

RR for User 1: **User 2 waits until User 1 commits**, User 1 get same result for both SELECTs

RS for User 1: **User 2 completes,** User 1 sees inserted row with 2nd SELECT

Do you use column functions?

User 1

```
insert into dept_size
(select dept, count(id) from staff
  where id between 50 and 300 group by dept)
                    insert into staff values(111,'me',20,'boss',0,99999,99999)
insert into  people_costs (
  select dept, sum(salary), sum(comm) from staff
   where id between 50 and 300 group by dept)
```

User 2

RR for User 1: **User 2 waits until User 1 commits**, for all depts: avg = sum/count

RS for User 1: **User 2 completes,** for dept 20: avg IS NOT equal to sum/count

Do you need all these locks?

User 1

```
select * from staff where id between 50 and 300 and name like '%il%'
                                        update staff set years=11 where id =160
```

User 2

RR for User 1: **User 2 waits until User 1 commits although id=160 is not in the result set**

RS for User 1: **User 2 completes**

RR - **RS - CS** - UR

Read Stability vs. Cursor Stability

What is locked?

User 1

```
Declare c1 cursor for select * from staff where id between 50 and 300
open c1, fetch c1, fetch c1, fetch cl
```

```
update staff set years = 11 where id = xxx
```

User 2

RS for User 1: **User 2 waits if xxx <= ID retrieved by last fetch**

CS for User 1: **User 2 only waits if xxx = ID retrieved by last fetch**

Do you always see the cursor?

User 1

```
insert into people_costs (
  select dept, sum(salary), sum(comm)from staff
    where id between 50 and 300 group by dept)

insert into avg_costs (
  select dept, avg(salary), avg(comm)from staff
    where id between 50 and 300 group by dept)
```

```
update staff set salary=99999 where id=200
```

User 2

RS for User 1: **User 2 waits until User 1 commits**, for all depts: avg = sum/count

CS for User 1: **User 2 completes,** for dept containing id=200 avg IS NOT equal to sum/count

Do you know where your cursor is?

User 1

```
declare c1 cursor for select * from staff where id between 50 and 300
  order by salary
open c1, fetch c1, fetch c1, fetch c1
```

```
update staff set years = 11 where id = xxx
```

User 2

RS for User 1: **User 2 waits if xxx between 50 and 300**

CS for User 1: **User 2 completes independent of xxx**

Cursor Stability vs. Uncommitted Read

Don't save uncommitted data!

User 1

```
                                update staff set salary = 22222 where id=200
declare c1 cursor for select * from staff
where id between 50 and 300 order by salary desc
open c1
fetch c1
insert into high_costs (select id, current date, salary, comm from staff
where id=xxx)                                                    rollback
fetch
insert
fetch
insert
...
commit
```

User 2

User 2

If ID = 200 is amongst fetched rows, then...

... CS for User 1: **User 1 waits until rollback complete, inserted data is consistent**
... UR for User 1: **User 1 does not wait, inserted data is inconsistent**

Isolation Level Summary

Phenomenon / Isolation Level	Access to Uncommitted Data (Dirty Read)	Nonrepeatable Reads	Phantom Read Phenomenon
Repeatable Read (RR)	Not Possible	Not Possible	Not Possible
Read Stability (RS)	Not Possible	Not Possible	Possible
Cursor Stability (CS)	Not Possible	Possible	Possible
Uncommitted Read (UR)	Possible	Possible	Possible

How to Set the Isolation Level

- Compiled language:
  - ISOLATION option of PREP or BIND commands
  - PREP or BIND APIs
- Call Level Interface
  - db2cli.ini, TXNISOLATION
  - SQLSetConnectAttr, SQL_ATTR_TXN_ISOLATION
  - SQLSetStmtAttr, SQL_ATTR_TXN_ISOLATION
- Command line processor:
  - CHANGE ISOLATION LEVEL command.
- Default is cursor stability.

Attributes of Locks

- Object
  - The resource being locked
  - Explicitly lockable: tables
  - Implicitly by database manager: rows, tables, table spaces, internal objects
- Duration
  - Length of time a lock is held
  - Affected by isolation levels
  - Affected by DML
- Mode
  - Type of access allowed for the lock owner as well as the type of access permitted for concurrent users of the locked object

A Definition

- UDB using "next key locking".
- So let's define:
  - Given two rows A and B.
  - Assume there is an index where the key of row B is the next key for the key of row A.
  - Then A is called preceding B
  - and B is called adjacent to A.

Exclusive locks

- **X (Exclusive)**
  - obtained on a row when row is updated or deleted
  - obtained when a row is inserted, converted to W, when index entries have been created
  - lock owner can read and change locked row
  - ncurrent LUWs can not change locked row
  - UWs can not insert preceding rows
  - concurr read locked row except with isolation UR
- **W (Weak Exclusive)**
  - obtained when a row is inserted ies have been created and X lock is released
  - same as X locks except that concurrent LU ceding rows
- **NX (Next Key Exclusive)**
  - obtained on adjacent rows during index update when a row is del
  - lock owner can read locked row
  - lock owner can only change the row after the NX lock has been converted to a higher lock
  - same as X lock except that it the row can be read by concurrent LUWs using RS, CS, UR
- **NW (Next Key Weak Exclusive)**
  - obtained on adjacent rows during index update when a row is inserted into a table
  - lock owner can read locked row
  - lock owner can only change the row after the NW lock has been converted to a higher lock
  - same as NX except that the insert is not put in wait by adjacent rows inserted by other LUWs

Exclusive locks are normally held until LUW ends!

Update Row Locks

- U (Update)
  - Obtained on a cursor with FOR UPDATE clause when it moves to a row
  - Lock owner can read locked row
  - Lock owner can get lock upgraded if he wants to update the row
  - ...nt LUWs can not change locked row
  - Con... ...not delete or insert preceding rows
  - Concurrent LUW... ...ed but not with a cursor with FOR UPDATE

Lock duration depends on isolation level !

Shared Row Locks

- **S (Share)**
  - obtained by SELECTs
  - concurrent LUWs can read, but not change, the locked data
  - concurrent LUWs can not delete or insert rows which precede the locked row
  - owner can only change the data after the S lock has been converted to a higher lock
- **NS (**
  - similar to S lock; concurrent LUWs can delete and insert rows which precede the locked row
  - obtained by SELECTs in p… for RS and CS isolation

Lock duration depends on isolation level !

A Bit on Cursors

S locks

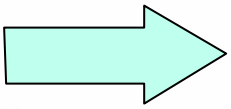
- Cursors can be read-only. This is the case if the cursor
  - is declared with FOR READ ONLY clause
  - contains ORDER BY, GROUP BY, HAVING, DISTINCT clauses
  - includes UNION, EXCEPT, INTERSECT operators
  - is based on a join, nested or common table expression
  - contains derived columns in the select list

  You can not use UPDATE and DELETE ... WHERE CURRENT OF.....

S locks,
U locks,
X locks

- Cursors can be deletable (updateable). This is the case if the cursor
  - is declared with FOR UPDATE CLAUSE
  - is not read-only because of one of the above conditions

  You can use UPDATE and DELETE ... WHERE CURRENT OF.....



- Cursors can be ambiguous. This is the case if the cursor
  - has no FOR READ ONLY and no FOR UPDATE clause
  - is not read-only because of one of the above conditions

  Considered read-only if BLOCKING bind option is ALL

Row Locks: Compatibility

Lock requested \ Lock Held	NS	S	U	NX	X	NW	W
NS	yes	yes	yes	yes	no	yes	no
S	yes	yes	yes	no	no	no	no
U	yes	yes	no	no	no	no	no
NX	yes	no	no	no	no	no	no
X	no	no	no	no	no	no	no
NW	yes	no	no	no	no	no	yes
W	no	no	no	no	no	yes	no

Table Locks

- Row locks can be escalated to table locks:
  - S (Share), U (Update), X (Exclusive)
- In addition there are special table locks:
  - IS (Intent Share):
  - Indicates that an application holds S locks on rows in the table
  - IX (Intent Exclusive):
  - Indicates that an application holds X locks on rows in the table
  - SIX (Share with Intent Exclusive):
  - Appl. needs S on table but X on rows
- Plus
  - Z (Superexclusive):
  - For table administration (alter,drop, reorg...)

Table locks can be obtained ...

- Explicitly
  - LOCK TABLE IN SHARE MODE
  - LOCK TABLE IN EXCLUSIVE MODE
- Implicitly through the locking process
  - Intent locks on tables when rows in the table are locked
- Full locks (S, U, X) instead of row locks, when the whole table is accessed, for example in a table scan
  - Through escalation (see below)

Intent Locks: What for?

- Assume application A runs with isolation RS and reads many rows in a table
  - It holds
    - Many shared locks on rows
    - One IS lock on the table.
- Assume application B wants to alter the table
  - It requests a Z lock on the table.
- Check one lock only:
  - Z is incompatible with IS.
  - Application B waits until IS on table is released.
- No need to check all the row locks!

Lock Duration

- Exclusive locks are (normally) held until end of LUW
- Duration of read locks depends on
  - Isolation level
    - RR and RS: until end of LUW
    - CS: until cursor is moved off the row
  - Coding
    - CLOSE CURSOR WITH RELEASE

Normally no locks are held across LUW boundary.
Exception: Cursor WITH HOLD
➡ *IS or IX is held until cursor is closed*

Lock Conversion

- Within any LUW a process can only have one lock on a data object.
  - Locks are hierarchical!
  - Conversion occurs when lower lock is held and a higher lock is needed.
    - If I hold an S lock on a row and I request a U lock
    - The S lock is converted to U
    - If I hold an U lock on a row and I request a X lock
    - The U lock is converted to X

Lock Escalation

- Locks are kept in lock lists
- Lock list entries occupy storage (32 or 64 bytes)
  - Storage for lock list is a DB configuration parameter (locklist - number of 4K pages)
  - Storage used by individual application can be limited by DB configuration parameter (maxlocks - % of locklist per application)
- If maxlocks is exceeded
  - Lock escalation
  - **Lots of row locks are replaced by one table lock!**

Deadlock Detection

- If two applications run into a deadlock they would wait until doomsday.
- Asynchronous background process to detect deadlocks
  - The time between activation is a DB configuration parameter (dlchktime - millisecs.)
- If deadlock is detected:
  - One of the processes is rolled back
  - Its locks are freed
  - The other process can get the locks requested.

Lock Time-out

- Even without a deadlock an applications can make other wait for a long time.
  - Typically: coffee breaks during LUWs spanning terminal I/O!
- Locks tend to accumulate:
- Appl. 1 waits on appl. 2 which waits on appl.3 which waits on appl 4...
- To prevent long waits
  - Maximum time an application waits for a lock is a DB configuration parameter (locktimeout - secs.)
- Unfortunately the innocent are punished!

Summary

Locks protect your data.

Locks decrease concurrency.

You can maximise concurrency
without compromising protection:

- **Design your application with locking in mind**
- **Use short LUWs**
- **Use the right isolation level**
- **Use RELEASE when you close the cursor**
- **Setup DB configuration parameters.**