Craig S. Mullins

Return to Home Page

Vol. 10, No. 3 (November 2003)

From IDUG Solutions Journal...

The Buffer Pool

Know Your Isolation Levels

By Craig S. Mullins

Did you know that DB2 provides a way to change the way that a program or SQL statement acquires locks? That way is known as the isolation level and it can be set to specify the locking behavior for a transaction or statement. Standard SQL defines four isolation levels that can be set using the SET TRANSACTION ISOLATION LEVEL statement:

- Serializable

- Repeatable read
- Read committed
- Read uncommitted

The isolation level determines the mode of page or row locking implemented by the program as it runs.

DB2 supports a variation of the standard isolation levels. DB2 implements page and row locking at the program execution level, which means that all page or row locks are acquired as needed during the program run. Page and row locks are released as the program run depending on the isolation level.

In DB2 you can specify the following four isolation levels: cursor stability (CS), repeatable read (RR), read stability (RS), and uncommitted read (UR).

Using the ISOLATION parameter of the BIND command you can set the isolation level of a package or plan. You also can use the WITH parameter on a SELECT statement to set the isolation level of a single SQL statement.

Cursor stability is the DB2 implementation of the SQL standard read committed isolation level. CS is perhaps the most common DB2 isolation level in use in production applications because it offers a good tradeoff between data integrity and concurrency. When CS is specified the transaction will never read data that is not yet committed; only committed data can be read.

A higher level of integrity is provided with ***repeatable read***. Under an RR isolation level all page locks are held until they are released by a COMMIT (or ROLLBACK), whereas with CS read-only page locks are released as soon as another page is accessed. Repeatable read is the default isolation level if none is specified at BIND time.

An RR page locking strategy is useful when an application program requires consistency in rows that may be accessed twice in one execution of the program, or when an application program requires data integrity that cannot be achieved with CS.

For example of a good reason to use RR page locking, consider a reporting program that scans a table to produce a detail report, and then scans it again to

produce a summarized managerial report. If the program is bound using CS, the results of the first report might not match the results of the second.

Suppose that you are reporting the estimated completion dates for project activities. The first report lists every project and the estimated completion date. The second, managerial report lists only the projects with a completion date greater than one year.

The first report indicates that two activities are scheduled for more than one year. After the first report but before the second, however, an update occurs. A manager realizes that she underestimated the resources required for a project. She invokes a transaction to change the estimated completion date of one of her project's activities from 8 months to 14 months. The second report is produced by the same program, but reports 3 activities.

If the program used an RR isolation level rather than CS, an UPDATE that occurs after the production of the first report but before the second would not have been allowed. The program would have maintained the locks it held from the generation of the first report and the

updater would be locked out until the locks were released.

How about another example? Consider a program that is looking for pertinent information about employees in the information center and software support departments who make more than $30,000 in base salary. In the DB2 sample tables department 'C01' is the information center and department 'E21' is software support.

The program opens a cursor based on the following SELECT statement:

```
SELECT  EMPNO, FIRSTNME, LASTNAME,
        WORKDEPT, SALARY
FROM    DSN8710.EMP
WHERE   WORKDEPT IN ('C01', 'E21')
AND     SALARY > 30000;
```

The program then begins to FETCH employee rows. Assume further, as would probably be the case, that the statement uses the XEMP2 index on the WORKDEPT column. An update program that implements employee modifications is running concurrently. The program handles transfers by moving employees from one

department to another, and implements raises by increasing the salary.

Assume that Sally Kwan, one of your employees, has just been transferred from the information center to software support. Assume further that another information center employee, Heather Nicholls, received a 10 percent raise. The update program running concurrently with the report program implements both of these modifications.

If the report program were bound with an isolation level of CS, the second program could move Sally from 'C01' to 'E21' after she was reported to be in department 'C01' but before the entire report was finished. Thus, she could be reported twice: once as an information center employee and again as a software support employee. Although this circumstance is rare, it can happen with programs that use cursor stability. If the program were bound instead with RR, this problem could not happen. The update program probably would not be allowed to run concurrently with a reporting program, however, because it would experience too many locking problems.

Now consider Heather's dilemma. The raise increases her salary 10 percent, from $28,420 to $31,262. Her salary now fits the parameters specified in the WHERE condition of the SQL statement. Will she be reported? It depends on whether the update occurs before or after the row has been retrieved by the index scan, which is clearly a tenuous situation. Once again, RR avoids this problem.

You might be wondering, "If CS has the potential to cause so many problems, why is it used so ubiquitously? Why not trade the performance and concurrency gain of CS for the integrity of RR?"

The answer is simple: the types of problems outlined are rare. The expense of using RR, however, can be substantial in terms of concurrency. So the tradeoff between the concurrency expense of RR and the efficiency of CS usually is not a sound one.

The third isolation level provided by DB2 is ***read stability*** (RS). Read stability is similar in functionality to the RR isolation level, but a little less. A retrieved row or page is locked until the end of the unit of work; no other program can modify the data until the unit of work is

complete, but other processes can insert values that might be read by your application if it accesses the row a second time.

Consider using read stability over repeatable read only when your program can handle retrieving a different set of rows each time a cursor or singleton SELECT is issued. If using read stability, be sure your application is not dependent on having the same number of rows returned each time.

Finally, we come to the last, and most maligned isolation level, ***uncommitted read*** (UR). The UR isolation level provides read-through locks, also know as *dirty read* or *read uncommitted.* Using UR can help to overcome concurrency problems. When you're using an uncommitted read, an application program can read data that has been changed but is not yet committed.

UR can be a performance booster, too, because application programs bound using the UR isolation level will read data without taking locks. This way, the application program can read data contained in the table as it is being manipulated. Consider the following sequence of events:

1. To change a specific value, at 9:00 a.m. a transaction containing the following SQL is executed:

```
UPDATE EMP
   SET FIRSTNME = "MICHELLE"
WHERE  EMPNO = 10020;
```

The transaction is a long-running one and continues to execute without issuing a COMMIT.

2. At 9:01 a.m., a second transaction attempts to SELECT the data that was changed, but not committed.

If the UR isolation level were used for the second transaction, it would read the changed data even though it had yet to be committed. Obviously, if the program doesn't need to wait to take a lock and merely reads the data in whatever state it happens to be at that moment, the program will execute faster than if it had to wait for locks to be taken and resources to be freed before processing.

The implications of reading uncommitted data, however, must be carefully examined before being implemented. Several types of problems can occur. Using the previous example, if the long-running

transaction rolled back the UPDATE to EMPNO 10020, the program using dirty reads may have picked up the wrong name ("MICHELLE") because it was never committed to the database.

Inaccurate data values are not the only problems that can be caused by using UR. A dirty read can cause duplicate rows to be returned where none exist. Alternatively, a dirty read can cause no rows to be returned when one (or more) actually exists. Additionally, an ORDER BY clause does not guarantee that rows will be returned in order if the UR isolation level is used. Obviously, these problems must be taken into consideration before using the UR isolation level.

Keep in mind, too, that the UR isolation level applies to read-only operations: SELECT, SELECT INTO, and FETCH from a read-only result table. Any application plan or package bound with an isolation level of UR will use uncommitted read functionality for any read-only SQL. Operations contained in the same plan or package and are not read-only will use an isolation level of CS.

When is it appropriate to use UR isolation? The general rule of thumb is to avoid UR whenever the results must

be 100 percent accurate. Following are examples of when this would be true:

- Calculations that must balance are being performed on the selected data
- Data is being retrieved from one source to insert to or update another
- Production, mission-critical work is being performed that cannot contain
  or cause data integrity problems

In general, most DB2 applications are not serious candidates for dirty reads. In a few specific situations, however, the dirty read capability will be of major benefit. Consider the following cases in which the UR isolation level could prove to be useful:

- Access is required to a reference, code, or look-up table that basically is static in nature. Due to the non-volatile nature of the data, a dirty read would be no different than a normal read the majority of the time. In those cases when the code data is being modified, any application reading the data would incur minimum, if any, problems.

- Statistical processing must be performed on a large amount of data. Your company, for example, might want to determine the average age of female employees within a certain pay range. The impact of an uncommitted read on an average of multiple rows will be minimal because a single value changed will not greatly impact the result.

- Dirty reads can prove invaluable in a data warehousing environment that uses DB2 as the DBMS. A data warehouse is a time-sensitive, subject-oriented, store of business data that is used for online analytical processing. Other than periodic data propagation and/or replication, access to the data warehouse is read-only. Because the data is generally not changing, an uncommitted read is perfect in a read-only environment due to the fact that it can cause little damage. More data warehouse projects are being implemented in corporations worldwide and DB2 with dirty read capability is a very wise choice for data warehouse implementation.

- In those rare cases when a table, or set of tables, is used by a single user only, UR can make a lot of

sense. If only one individual can be modifying the data, the application programs can be coded such that all (or most) reads are done using UR isolation level, and the data will still be accurate.

- Finally, if the data being accessed already is inconsistent, little harm can be done using a dirty read to access the information.

Although the dirty read capability can provide relief to concurrency problems and deliver faster performance in specific situations, it also can cause data integrity problems and inaccurate results. Be sure to understand the implications of the UR isolation level and the problems it can cause before diving headlong into implementing it in your production applications.

Summary

It is important for DB2 DBAs and application programmers to know the four isolation levels and their impact on SQL. Using the isolation levels is an effective way to control concurrency and locking for your DB2 applications.

From *IDUG Solutions Journal*, November 2003.

Home.