

# Improve concurrency with DB2 9.5 optimistic locking

New optimistic locking feature to avoid maintaining long-lived locks

Skill Level: Intermediate

[Werner Schuetz \(werner\\_schuetz@de.ibm.com\)](mailto:werner_schuetz@de.ibm.com)

IBM Certified IT Specialist

IBM

17 Oct 2008

IBM® DB2®, Version 9.5 for Linux®, UNIX®, and Windows® provides enhanced optimistic locking support, a technique for SQL database applications that does not hold row locks between selecting and updating, or deleting rows. Gain an understanding of this enhancement, and learn how applications using this programming model benefit from this enhanced optimistic locking feature and gain improved concurrency.

## Introduction

### **Pessimistic and optimistic locking**

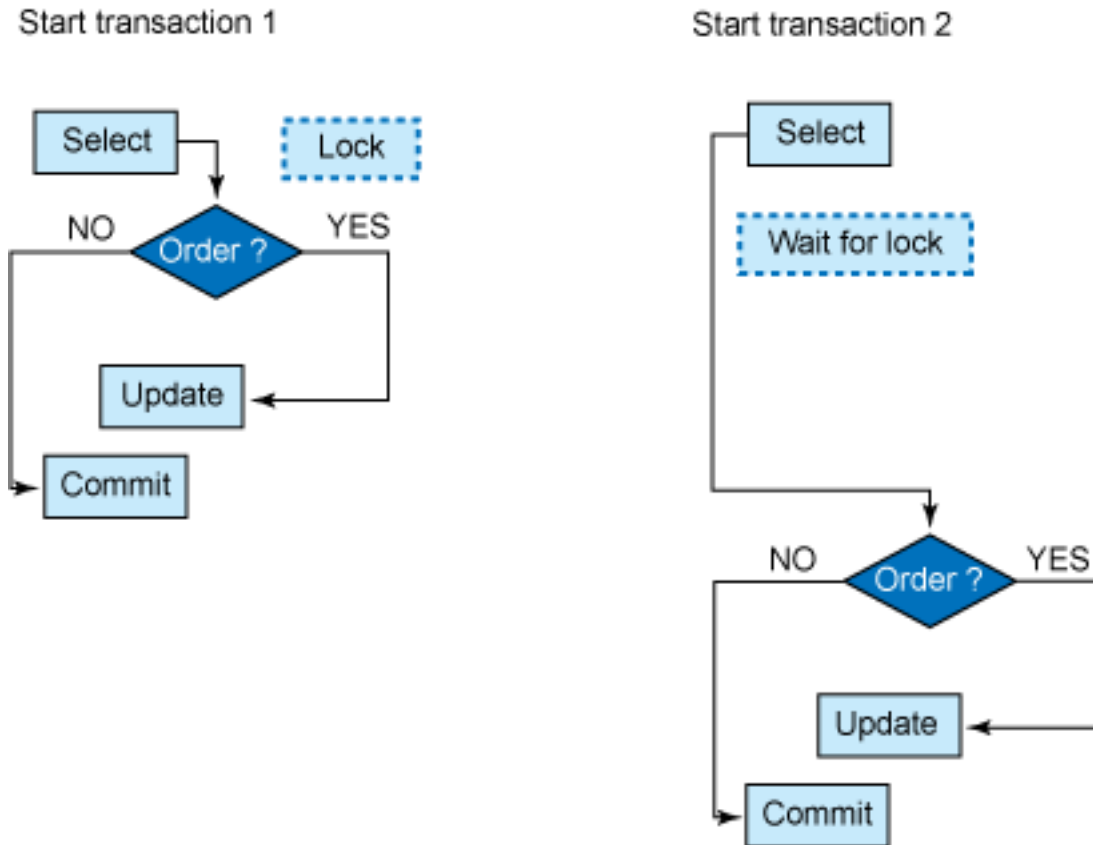
#### **Pessimistic locking**

A pessimistic locking strategy assumes that the probability is high that another user will try to modify the same row in a table that you are changing. A lock is held between the time a row is selected and the time that a searched update or delete operation is attempted on that row (for example, by using the repeatable read isolation level or lock the table in exclusive mode). The advantage of pessimistic locking is that it is guaranteed that changes are made consistently and safely. The major disadvantage is that this locking strategy might not be very scalable. On a system with many users or with long-living transactions, or when transactions involve a greater number of entities, the probability of having to wait for a lock to be

released increases.

Figure 1 illustrates the functioning of pessimistic locking. Transaction 1 reads a specific record and places a lock on that row. It takes some time to decide whether the row will be updated. In the meantime, transaction 2 wants access to that same row, but it has to wait until the lock is released by Transaction 1. Until then, transaction 2 will receive results from its `SELECT` and can continue with its business logic.

**Figure 1. Pessimistic locking concept**



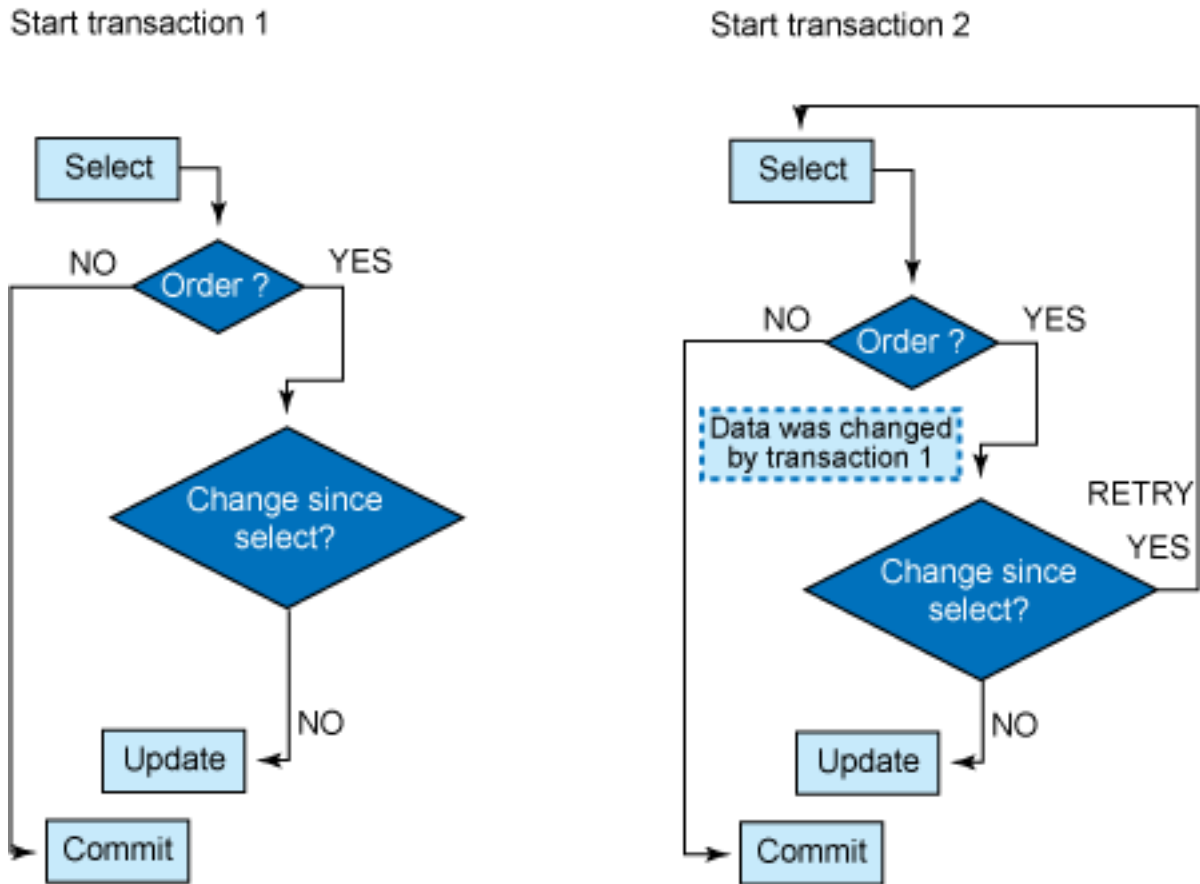
**Optimistic locking**

The main problem with a pessimistic locking approach is that transactions have to wait for each other. A way to avoid this is to follow an optimistic locking strategy and assume that it is very unlikely that another user will try to change the same row that you are changing. If the row does change, the update or delete will fail, and the application logic handles such failures by, for example, retrying the select. With this approach, no locks are held between selecting and updating, or deleting a row. But, consequently, this method requires a way to ensure that the data has not changed between the time of being read and being altered. Although more retry logic in the application is needed, the primary advantage of an optimistic locking strategy is that it minimizes the time for which a given resource is unavailable for use by other transactions and thus will be a more scalable locking alternative than pessimistic

locking.

Figure 2 illustrates the idea behind optimistic locking. Similar to Figure 1, transaction 1 reads a specific record but then releases its lock. Transaction 2 is now not prevented from retrieving that same row. Before committing the transaction, both transaction 1 and transaction 2 must check whether the row has changed after the previous SELECT. If a change has occurred, the transaction must restart with a new SELECT in order to retrieve the current data. However, if that row has not been changed after the previous SELECT, the data can be successfully updated.

Figure 2. Optimistic locking concept



Enhanced optimistic locking with DB2 9.5

Optimistic locking in DB2 9.5 improves scalability by minimizing the time for which a given resource is unavailable for use by other transactions. Because the database manager can determine when a row is changed, it can ensure data integrity while limiting the time that locks are held. With optimistic concurrency control, the database manager releases the row or page locks immediately after a read operation.

DB2 9.5 for Linux, Unix, and Windows adds support for easier and faster optimistic

locking with no false positives. This support is added using the following new SQL functions, expressions, and features:

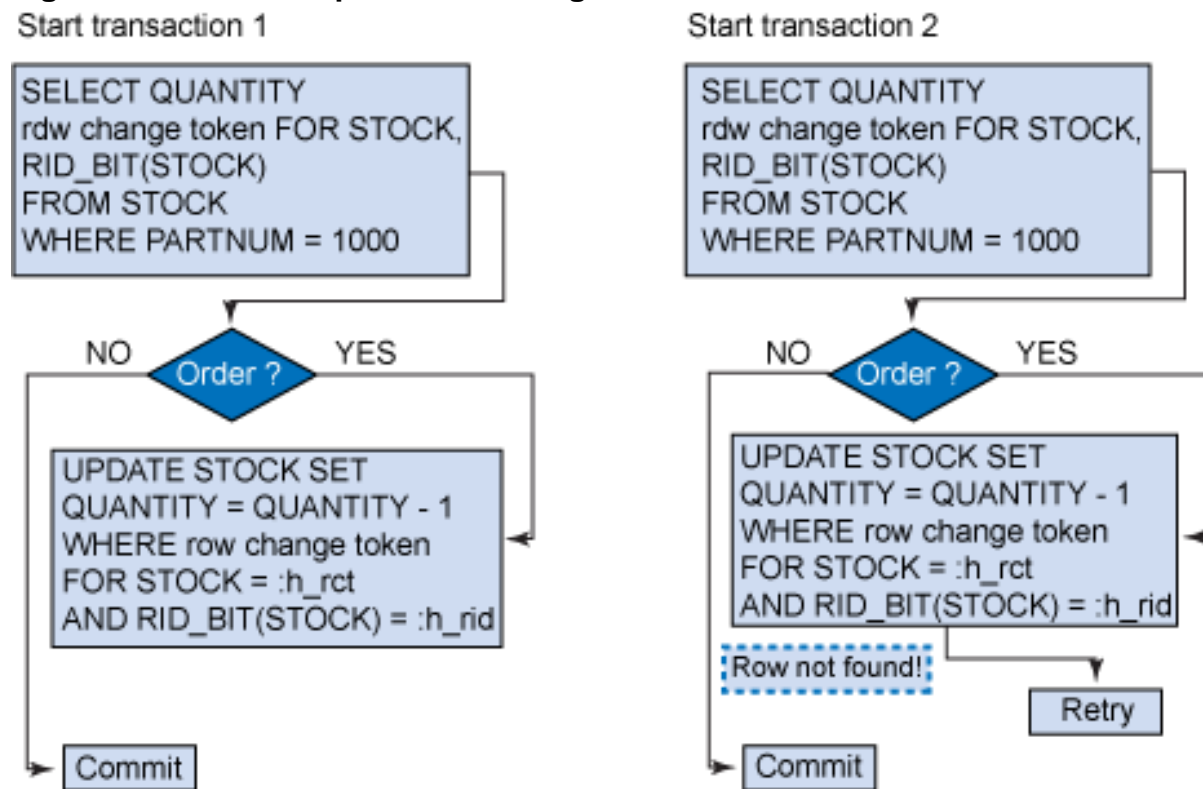
- **Row identifier (RID\_BIT or RID) built-in function:** This built-in function can be used in the `SELECT` list or predicates statement. In a predicate, for example, `WHERE RID_BIT(tab)=?`, the `RID_BIT` equals predicate is implemented as a new direct access method in order to efficiently locate the row. Previously, thus called values optimistic locking with values was done by adding all the selected column values to the predicates and relying on some unique column combinations to qualify only a single row, with a less efficient access method.
- **ROW CHANGE TOKEN expression:** This new expression returns a token as `BIGINT`. The token represents a relative point in the modification sequence of a row. An application can compare the current row change token value of a row with the row change token value that was stored when the row was last fetched to determine whether the row has changed.
- **Time-based update detection:** This feature is added to SQL using the `ROW CHANGE TIMESTAMP` expression. To support this feature, the table needs to have a new generated row change timestamp column defined to store the timestamp values. This can be added to existing tables using the `ALTER TABLE` statement, or the row change timestamp column can be defined when creating a new table. The row change timestamp column's existence also affects the behavior of optimistic locking in that the column is used to improve the granularity of the row change token from page level to row level, which could greatly benefit optimistic locking applications.
- **Implicitly hidden columns:** For compatibility, this feature eases the adoption of the row change timestamp columns to existing tables and applications. Implicitly hidden columns are not externalized when implicit column lists are used. For example a `SELECT *` against the table does not return a implicitly hidden columns in the result table and an `INSERT` statement without a column list does not expect a value for implicitly hidden columns, but the column should be defined to allow nulls or have another default value.

Applications using this programming model will benefit from the enhanced optimistic locking feature. Note that applications that do not use this programming model are not considered optimistic locking applications, and they will continue to work as before.

[Figure 3](#) illustrates the functioning of DB2 9.5 optimistic locking. Both transaction 1 and transaction 2 read the same row, including the `RID_BIT` and the `ROW CHANGE TOKEN` value. Then transaction 1 updates the row after ensuring that the row has

not changed after the previous SELECT by adding a RID\_BIT and ROW CHANGE TOKEN predicate to the UPDATE statement. When transaction 2 now tries to update that same row using the same predicate as transaction 1, the row will not be found because the value of the ROW CHANGE TOKEN has changed regarding to the UPDATE of transaction 1. Transaction 2 has to start a retry in order to retrieve the current data.

**Figure 3. Enhanced optimistic locking with DB2 9.5**



### Enabling optimistic locking

Since the new SQL expressions and attributes for optimistic locking can be used with no DDL changes to the tables involved, you can easily try optimistic locking in your test applications.

Note that without DDL changes, optimistic locking applications may get more false negatives than with DDL changes. An application that does get false negatives may not scale well in a production environment because the false negatives may cause too many retries. Therefore, to avoid false negatives, optimistic locking target tables should be either:

- Created with a ROW CHANGE TIMESTAMP column
- Altered to contain the ROW CHANGE TIMESTAMP column

These are a basic steps to be performed in order to enable optimistic locking support in your applications:

- In the initial query, `SELECT` the row identifier (using the `RID_BIT( )` and `RID( )` built-in function) and row change token for each row that you need to process.
- Release the row locks so that other applications can `SELECT`, `INSERT`, `UPDATE`, and `DELETE` from the table (for example, use isolation level cursor stability or uncommitted read).
- Perform a searched `UPDATE` or `DELETE` on the target rows, using the row identifier and row change token in the search condition, optimistically assuming that the unlocked row has not changed since the original `SELECT` statement.
- If the row has changed, the `UPDATE` operation will fail and the application logic must handle the failure. For instance, the application retries the `SELECT` and `UPDATE` operations.

After running the above steps:

- If the number of retries performed by your application seems higher than expected or is desired, then adding a row change timestamp column to your table to ensure that only changes to the row identified by the `RID_BIT` function will invalidate only the row change token, and not other activity on the same data page.
- To see rows that have been inserted or updated in a given time range, create or alter the table to contain a row change timestamp column. This column will be maintained by the database manager automatically and can be queried using either the column name or the `ROW CHANGE TIMESTAMP` expression.
- For row change timestamp columns only, if the column is defined with the `IMPLICITLY HIDDEN` attribute, then it is not externalized when there is an implicit reference to the columns of the table. However, an implicitly hidden column can always be referenced explicitly in SQL statements. This can be useful when adding a column to a table can cause existing applications using implicit column lists to fail.

## Granularity of row change tokens and false negatives

The `RID_BIT( )` built-in function and the row change token are the only requirements for optimistic locking. However, the schema of the table also affects the behavior of optimistic locking.

For example, a row change timestamp column causes the DB2 server to store the time when a row is last changed (or initially inserted). This provides a way to capture the timestamp of the most recent change to a row. A row change timestamp column is defined using either of the following statement clauses:

- **GENERATED ALWAYS FOR EACH ROW ON UPDATE AS ROW CHANGE TIMESTAMP**  
This timestamp column is maintained always by the database manager
- **GENERATED BY DEFAULT FOR EACH ROW ON UPDATE AS ROW CHANGE TIMESTAMP**  
This timestamp column is maintained by default by the database manager, but a user-provided input value is also accepted.

When an application uses the new ROW CHANGE TOKEN expression on a table, there are two possibilities to consider:

- The table does not have a row change timestamp column: A ROW CHANGE TOKEN expression returns a derived BIGINT value that is shared by all rows located on the same page. If one row on a page is updated, the row change token is changed for all the rows on the same page. This means an update can fail when changes are made to other rows, a property referred to as a false negative.  
**Note:** Use this mode only if the application can tolerate false negatives and does not want to add additional storage to each row for a ROW CHANGE TIMESTAMP column.
- The table has a row change timestamp column: A ROW CHANGE TOKEN expression returns a BIGINT value derived from the timestamp value in the column. In this case, false negatives may occur but are more infrequent: If the table is reorganized or redistributed, false negatives can occur if the row is moved and an application uses the prior RID\_BIT( ) value.

The existence of a row change timestamp column can be queried by the following SELECT:

### Listing 1. SELECT query

```
SELECT COLNAME, ROWCHANGETIMESTAMP, GENERATED FROM SYSCAT.COLUMNS
       WHERE TABNAME='tablename' AND ROWCHANGETIMESTAMP='Y'
```

COLNAME	ROWCHANGETIMESTAMP	GENERATED
-----	-----	-----
ROWCHGTS	Y	A

In this sample a row change timestamp column ROWCHGTS exists and is defined with the GENERATED ALWAYS clause (a value of "A" means GENERATED ALWAYS, where a value of "D" stands for GENERATED BY DEFAULT)

## Time-based update detection

Some applications need to know database updates for certain time ranges, which may be used for replication of data, auditing scenarios, and so forth. This is implemented by a table containing a row change timestamp column defined to store the timestamp values generated by a `ROW CHANGE TIMESTAMP` expression. This new `ROW CHANGE TIMESTAMP` expression returns a timestamp representing the time when a row was last changed, expressed in local time similar to `CURRENT TIMESTAMP`. For a row that has been updated, this reflects the most recent update to the row. Otherwise, the value corresponds to the original insert of the row.

Rows that have not been updated since the `ALTER TABLE` statement will return the type default value for the column, which is midnight Jan 01, year 1. Only rows that have been updated will have a unique timestamp. Rows that have the timestamp materialized using an offline table reorganization will return a unique timestamp generated during the reorganization of the table. `REORG` using the `INPLACE` option is not sufficient as it does not materialize schema changes.

### Listing 2. Table created with a row change timestamp column

```
CREATE TABLE EMPLOYEE (EMPNO CHAR(6) NOT NULL,
    .....
    ROWCHGTS TIMESTAMP NOT NULL
    GENERATED ALWAYS
    FOR EACH ROW ON UPDATE AS
    ROW CHANGE TIMESTAMP)
```

### Listing 3. Table not created with a row change timestamp column, but one was later added through an ALTER TABLE statement

```
ALTER TABLE EMPLOYEE ADD COLUMN
    ROWCHGTS TIMESTAMP NOT NULL
    GENERATED ALWAYS
    FOR EACH ROW ON UPDATE AS
    ROW CHANGE TIMESTAMP
```

### Listing 4. Select all rows that have changed in the last 30 days

```
SELECT * FROM EMPLOYEE WHERE
    ROW CHANGE TIMESTAMP FOR EMPLOYEE <= CURRENT TIMESTAMP AND
    ROW CHANGE TIMESTAMP FOR EMPLOYEE >= CURRENT TIMESTAMP - 30 days
```

**Table 1. The content of the ROW CHANGE TIMESTAMP column after populating with INSERT, IMPORT, or LOAD when the table was created with a row change timestamp column**

EMPNO	FIRSTNME	LASTNAME	PHONENO	ROW CHANGE
-------	----------	----------	---------	------------

				<b>TIMESTAMP</b>
<b>CHRISTINE</b>	<b>2007-12-20 13:53:01.296000</b>	000010	HAAS	3978
<b>MICHAEL</b>	<b>2007-12-20 13:53:01.312000</b>	000020	THOMPSON	3476
<b>SALLY</b>	<b>2007-12-20 13:53:01.312001</b>	000030	KWAN	4738

**Table 2. The content of the ROW CHANGE TIMESTAMP column when a row change timestamp column was added to an existing table**

<b>EMPNO</b>	<b>FIRSTNME</b>	<b>LASTNAME</b>	<b>PHONENO</b>	<b>ROW CHANGE TIMESTAMP</b>
<b>CHRISTINE</b>	<b>0001-01-01 00:00:00.000000</b>	000010	HAAS	3978
<b>MICHAEL</b>	<b>0001-01-01 00:00:00.000000</b>	000020	THOMPSON	3476
<b>SALLY</b>	<b>0001-01-01 00:00:00.000000</b>	000030	KWAN	4738

### Implicitly hidden columns

This feature eases the adoption of the row change timestamp columns to existing tables and applications. The `IMPLICITLY HIDDEN` attribute on a `CREATE` or `ALTER TABLE` statement specifies that the column is not visible in SQL statements unless the column is explicitly referenced by name. For example, assuming that a table includes a column defined with the `IMPLICITLY HIDDEN` clause, the result of a `SELECT *` does not include the implicitly hidden column. However, the result of a `SELECT` that explicitly refers to the name of an implicitly hidden column will include that column in the result table. `IMPLICITLY HIDDEN` must only be specified for a `ROW CHANGE TIMESTAMP` column.

### Listing 5. Implicitly hiding columns for a row change timestamp

```
CREATE TABLE SALARY_INFO (
    LEVEL INT NOT NULL,
    SALARY INT NOT NULL,
    UPDATE_TIME TIMESTAMP NOT NULL
    IMPLICITLY HIDDEN
    GENERATED ALWAYS FOR EACH ROW ON UPDATE AS ROW CHANGE TIMESTAMP)

OR

ALTER TABLE SALARY_INFO
ADD COLUMN UPDATE_TIME TIMESTAMP NOT NULL
IMPLICITLY HIDDEN
GENERATED ALWAYS FOR EACH ROW ON UPDATE AS ROW CHANGE TIMESTAMP
```

### Listing 6. Displays columns of the table using DESCRIBE command

```
DESCRIBE TABLE SALARY_INFO
```

Column name	Data type schema	Data type name	Column Length	Scale	Nulls
LEVEL	SYSIBM	INTEGER		4	0 No
SALARY	SYSIBM	INTEGER		4	0 No
UPDATE_TIME	SYSIBM	TIMESTAMP		10	0 No

**Listing 7. INSERT and SELECT for an implicitly hidden column**

```
INSERT INTO SALARY_INFO VALUES (1, 50000)

SELECT * FROM SALARY_INFO
```

LEVEL	SALARY
1	50000

**Listing 8. INSERT and SELECT where implicitly hidden column is explicitly referenced**

```
INSERT INTO SALARY_INFO (LEVEL, SALARY, UPDATE_TIME)
VALUES (2, 30000, DEFAULT)

SELECT LEVEL, SALARY, UPDATE_TIME
FROM SALARY_INFO
WHERE LEVEL = 2
```

LEVEL	SALARY	UPDATE_TIME
2	30000	2007-12-18-15.34.24.437000

**Optimistic locking restrictions and considerations**

- ROW CHANGE TIMESTAMP columns are not supported in primary keys, foreign keys, multidimensional clustered (MDC) columns, range partition columns, database hashed partitioning keys, DETERMINED BY constraint columns and nicknames.
- The RID( ) function is not supported in Database Partitioning Feature (DPF) configurations.
- Online or offline table REORG performed between the fetch and update operations in an optimistic locking scenario may cause the update to fail, but this should be handled by normal application retry logic.
- In Version 9.5, the IMPLICITLY HIDDEN attribute is restricted to only ROW CHANGE TIMESTAMP columns for optimistic locking.
- INPLACE REORG is restricted for tables where a ROW CHANGE

TIMESTAMP column was added to an existing table until all rows are guaranteed to have been materialized (SQL2219, reason code 13, is returned for this error). This can be accomplished with a `LOAD REPLACE` command or with a classic table `REORG`. This will prevent false positives. Tables created with the `ROW CHANGE TIMESTAMP` column have no restrictions.

## Usage scenarios

An employee got a new job responsibility and is now working for another department. Two managers (the manager of the old department called Manager1 and Manager2 from the new department) are using a personnel administration application to update employee records in the `EMPLOYEE` table in the `SAMPLE` database. There is a possibility that both managers try to update the same employee record at the same time. While Manager1 selects and updates the employee record, Manager2 updates the same record. The Optimistic Locking feature lets, for example, Manager2 know that a particular record has been updated before the current application update. This makes this application easier to program because it does not have to implement its own update detection logic.

### Scenario 1

The `EMPLOYEE` table contains a implicitly hidden `ROW CHANGE TIMESTAMP` column (which was added) and is accessed only by Manager1. Manager1 selects the data from the `EMPLOYEE` table and later tries to update the phone number of Christine Haas from 3978 to 1092. The update is successful.

#### Listing 9. SELECT statement(Manager1)

```
SELECT RID_BIT(EMPLOYEE),
       ROW CHANGE TOKEN FOR EMPLOYEE,
       EMPNO, FIRSTNME, LASTNAME, PHONENO, ROWCHGTS
FROM EMPLOYEE FETCH FIRST 3 ROWS ONLY
```

**Table 3. Result from the SELECT**

Optimistic locking expression		EMPLOYEE table				
RID_BIT	ROW CHANGE TOKEN	EMPNO	FIRSTNME	LASTNAME	PHONENO	ROW CHANGE TIMESTAMP
x'04004001000000000000F	CHRISTINE	0001-01-01 00:00:00.000000	749042296422900	0010	HAAS	3978
x'05004001000000000000F	MICHAEL	0001-01-01 00:00:00.000000	749042296422900	0020	THOMPSON	3476
x'06004001000000000000F	SALLY	0001-01-01 00:00:00.000000	749042296422900	0030	KWAN	4738

### Listing 10. UPDATE statement

```
UPDATE EMPLOYEE SET
  (FIRSTNME, LASTNAME, PHONENO) = ('CHRISTINE', 'HAAS', '1092')
WHERE RID_BIT(EMPLOYEE)=x'0400400100000000000000000000FA9023' AND
ROW CHANGE TOKEN FOR EMPLOYEE=74904229642240
```

**Table 4. Result from the UPDATE**

RID_BIT	ROW CHANGE TOKEN	EMPNO	FIRSTNME	LASTNAME	PHONENO	ROW CHANGE TIMESTAMP
x'0400400100000000000000000000FA9023	CHRISTINE	141285645885081032	CHRISTINE	HAAS	1092	2007-12-20 11:55:45.593000
x'0500400100000000000000000000FA9023	MICHAEL	0001-01-01 00:00:00.000000	749042296422400020	THOMPSON	3476	
x'0600400100000000000000000000FA9023	SALLY	0001-01-01 00:00:00.000000	749042296422400030	KWAN	4738	

### Scenario 2

The EMPLOYEE table contains a implicitly hidden ROW CHANGE TIMESTAMP column and is accessed by Manager1 and Manager2 simultaneously. Manager1 selects the data from the EMPLOYEE table and later tries to update the same data. However, between his select and his update, Manager2 updates the same data. Manager2's update is successful, but Manager1's update fails.

**Table 5. Result from the SELECT (Manager1 and Manager2)**

RID_BIT	ROW CHANGE TOKEN	EMPNO	FIRSTNME	LASTNAME	PHONENO	ROW CHANGE TIMESTAMP
x'0400400100000000000000000000FA9023	CHRISTINE	0001-01-01 00:00:00.000000	749042296422400010	HAAS	3978	
x'0500400100000000000000000000FA9023	MICHAEL	0001-01-01 00:00:00.000000	749042296422400020	THOMPSON	3476	
x'0600400100000000000000000000FA9023	SALLY	0001-01-01 00:00:00.000000	749042296422400030	KWAN	4738	

### Listing 11. UPDATE statement (Manager2)

```
UPDATE EMPLOYEE SET
  (FIRSTNME, LASTNAME, PHONENO) = ('CHRISTINE', 'HAAS', '1092')
WHERE RID_BIT(EMPLOYEE)=x'0400400100000000000000000000FA9023' AND
ROW CHANGE TOKEN FOR EMPLOYEE=74904229642240
```

**Table 6. Result from the UPDATE (Manager2)**

RID_BIT	ROW CHANGE TOKEN	EMPNO	FIRSTNME	LASTNAME	PHONENO	ROW CHANGE TIMESTAMP
x'04004001000000000000F	CHRISTINE	141285645885181032	749042296422400010	HAAS	1092	2007-12-20 11:55:45.593000
x'05004001000000000000F	MICHAEL	0001-01-01 00:00:00.000000	749042296422400020	THOMPSON	3476	
x'06004001000000000000F	SALLY	0001-01-01 00:00:00.000000	749042296422400030	KWAN	4738	

**Listing 12. UPDATE statement (Manager1)**

```
UPDATE EMPLOYEE SET
(FIRSTNME, LASTNAME, PHONENO) = ('CHRISTINE', 'HAAS', '1092')
WHERE RID_BIT(EMPLOYEE)=x'0400400100000000000000000000FA9023' AND
ROW CHANGE TOKEN FOR EMPLOYEE=74904229642240
```

**Result from the UPDATE (Manager1)**

The update from Manager1 is unsuccessful. Since the ROW CHANGE TOKEN has changed by the UPDATE of Manager2, the ROW CHANGE TOKEN predicate of Manager1's UPDATE statement fails while comparing the token retrieved at the point of the SELECT and the current value after having been updated by Manager2's application. Thus the UPDATE fails to find the specified row. A message "SQL0100W No row was found for FETCH, UPDATE or DELETE; or the result of a query is an empty table. SQLSTATE=02000" is returned.

**Scenario 3**

The EMPLOYEE table contains a implicitly hidden ROW CHANGE TIMESTAMP column and is accessed by Manager1 and Manager2 simultaneously. Manager1 updates rows but has not yet committed his changes. Manager2 selects the data from the Employee table with an isolation level of Uncommitted Read. Manager1 commits his changes. Manager2 tries to update the same data. This final update from Manager2 succeeds because it reads the uncommitted updates from Manager1. However, if Manager1 rolls back the update, instead of committing it, Manager2's update fails.

**Table 7. Result from the SELECT (Manager1)**

RID_BIT	ROW CHANGE TOKEN	EMPNO	FIRSTNME	LASTNAME	PHONENO	ROW CHANGE TIMESTAMP
x'04004001000000000000F	CHRISTINE	0001-01-01 00:00:00.000000	749042296422400010	HAAS	1092	3978
x'05004001000000000000F	MICHAEL	0001-01-01	749042296422400020	THOMPSON	3476	

0000000000F	00:00:00.000000					
x'060040010( SALLY	0001-01-01	749042296422400030		KWAN		4738
0000000000F	00:00:00.000000					

**Listing 13. UPDATE statement without commit (Manager1)**

```
UPDATE EMPLOYEE SET
  (FIRSTNME, LASTNAME, PHONENO) = ('CHRISTINE', 'HAAS', '1092')
WHERE RID_BIT(EMPLOYEE)=x'0400400100000000000000000000FA9023' AND
ROW CHANGE TOKEN FOR EMPLOYEE=74904229642240
```

**Table 8. Result from the SELECT (Manager2)**

RID_BIT	ROW CHANGE TOKEN	EMPNO	FIRSTNME	LASTNAME	PHONENO	ROW CHANGE TIMESTAMP
x'040040010( CHRISTINE	141285665533242120		HAAS		1092	2007-12-20 16:47:03.125000
x'050040010( MICHAEL	0001-01-01	749042296422400020		THOMPSON		3476
0000000000F	00:00:00.000000					
x'060040010( SALLY	0001-01-01	749042296422400030		KWAN		4738
0000000000F	00:00:00.000000					

**Table 9. Result from the committing the UPDATE (Manager1)**

RID_BIT	ROW CHANGE TOKEN	EMPNO	FIRSTNME	LASTNAME	PHONENO	ROW CHANGE TIMESTAMP
x'040040010( CHRISTINE	141285665533242120		HAAS		1092	2007-12-20 16:47:03.125000
x'050040010( MICHAEL	0001-01-01	749042296422400020		THOMPSON		3476
0000000000F	00:00:00.000000					
x'060040010( SALLY	0001-01-01	749042296422400030		KWAN		4738
0000000000F	00:00:00.000000					

**Listing 14. UPDATE statement (Manager2)**

```
UPDATE EMPLOYEE SET
  (FIRSTNME, LASTNAME, PHONENO) = ('CHRISTINE', 'HAAS', '1090')
WHERE RID_BIT(EMPLOYEE)=x'0400400100000000000000000000FA9023' AND
ROW CHANGE TOKEN FOR EMPLOYEE=141285665533242120
```

**Table 10. Result from the UPDATE (Manager2)**

RID_BIT	ROW CHANGE TOKEN	EMPNO	FIRSTNME	LASTNAME	PHONENO	ROW CHANGE TIMESTAMP
x'040040010( CHRISTINE	14128566709502664		HAAS		1090	2007-12-20

0000000000F						16:51:53.125000
x'0500400100 0000000000F	MICHAEL	0001-01-01 00:00:00.000000	749042296422400020	THOMPSON	3476	
x'0600400100 0000000000F	SALLY	0001-01-01 00:00:00.000000	749042296422400030	KWAN	4738	

**Result from Manager1 committing his changes and Manager2 trying to update the same data:**

This final update from Manager2 succeeds because it reads the uncommitted updates from Manager1, and the ROW CHANGE TOKEN predicate in the UPDATE statement of Manager2 succeeds, as Manager1 committed the changes with the new token.

**Table 11. Result if Manager1 issues a ROLLBACK instead of the COMMIT**

RID_BIT	ROW CHANGE TOKEN	EMPNO	FIRSTNME	LASTNAME	PHONENO	ROW CHANGE TIMESTAMP
x'0400400100 0000000000F	CHRISTINE	749042296422400010	HAAS		3978	0001-01-01 00:00:00.000000
x'0500400100 0000000000F	MICHAEL	0001-01-01 00:00:00.000000	749042296422400020	THOMPSON	3476	
x'0600400100 0000000000F	SALLY	0001-01-01 00:00:00.000000	749042296422400030	KWAN	4738	

**Result from Manager2 trying to update the data as of Manager1's uncommitted UPDATE after Manager1 has rolled back his changes:**

This final update from Manager2 is unsuccessful, because the ROW CHANGE TOKEN predicate fails, as Manager1 has rolled back to the original token, thus the UPDATE fails to find the row.

**Scenario 4**

The EMPLOYEE table does **not** have a ROW CHANGE TIMESTAMP column and is accessed by Manager1 and Manager2 simultaneously. Manager1 selects a row and tries to update it. However, between his select and the updates, Manager2 updates other data (not necessarily the same data as Manager1, but in another row) on the same data page. Therefore, when Manager1 tries to update the data, the update fails.

**Listing 15. UPDATE statement (Manager2)**

```
UPDATE EMPLOYEE SET
    (FIRSTNME, LASTNAME, PHONENO) = ('CHRISTINE', 'HAAS', '1092')
```

```
WHERE RID_BIT(EMPLOYEE)=x'0400400100000000000000000000FA9023' AND
ROW CHANGE TOKEN FOR EMPLOYEE=74904229642240
```

**Table 12. Result from the UPDATE (Manager2)**

RID_BIT	ROW CHANGE TOKEN	EMPNO	FIRSTNME	LASTNAME	PHONENO
x'0400400100000000000000000000FA9023'	( CHRISTINE	14128564588518003210	CHRISTINE	HAAS	1092
x'0500400100000000000000000000FA9023'	( MICHAEL	14128564588518003220	MICHAEL	THOMPSON	3476
x'0600400100000000000000000000FA9023'	( SALLY	14128564588518003230	SALLY	KWAN	4738

**Listing 16. UPDATE statement for another row (Manager1)**

```
UPDATE EMPLOYEE SET
(FIRSTNME, LASTNAME, PHONENO) = ('MICHAEL', 'THOMPSON', '9012')
WHERE RID_BIT(EMPLOYEE)=x'0400400100000000000000000000FA9023' AND
ROW CHANGE TOKEN FOR EMPLOYEE=74904229642240
```

**Result from Manager1 trying to update another row on the same data page:**

The update from Manager1 is unsuccessful since the ROW CHANGE TOKEN predicate fails while comparing the token, as ROW CHANGE TOKEN values for all the rows have changed even though the row that Manager1 tries to update has actually not changed. This false negative scenario would not have the UPDATE cause a fail if a row change timestamp column was added to the EMPLOYEE table.

**Scenario 5**

The EMPLOYEE table does **not** have a ROW CHANGE TIMESTAMP column. It is altered and a ROW CHANGE TIMESTAMP column **is added**. Manager1 and Manager2 access the same table. Manager1 selects a row and tries to update it. However, between his select and the updates, Manager2 updates other data (not necessarily the same data as Manager1, but in another row) on the same data page. Given that the ROW CHANGE TIMESTAMP column has been added, updates to different rows, even if they are on the same page, will succeed.

**Listing 17. UPDATE statement (Manager2)**

```
UPDATE EMPLOYEE SET
(FIRSTNME, LASTNAME, PHONENO) = ('CHRISTINE', 'HAAS', '1092')
WHERE RID_BIT(EMPLOYEE)=x'0400400100000000000000000000FA9023' AND
ROW CHANGE TOKEN FOR EMPLOYEE=74904229642240
```

**Table 13. Result from the UPDATE (Manager2)**

RID_BIT	ROW CHANGE TOKEN	EMPNO	FIRSTNME	LASTNAME	PHONENO	ROW CHANGE TIMESTAMP
x'04004001000000000000F	CHRISTINE	141285673714388072	<del>CHRISTINE</del>	HAAS	1092	2007-12-20 18:22:25.593000
x'05004001000000000000F	MICHAEL	0001-01-01 00:00:00.000000	74904229642240	THOMPSON	3476	
x'06004001000000000000F	SALLY	0001-01-01 00:00:00.000000	74904229642240	KWAN	4738	

**Listing 18. UPDATE statement for another row (Manager1)**

```
UPDATE EMPLOYEE SET
(FIRSTNME, LASTNAME, PHONENO) = ('MICHAEL', 'THOMPSON', '9012')
WHERE RID_BIT(EMPLOYEE)=x'0400400100000000000000000000FA9023' AND
ROW CHANGE TOKEN FOR EMPLOYEE=74904229642240
```

**Table 14. Result from the UPDATE (Manager1)**

RID_BIT	ROW CHANGE TOKEN	EMPNO	FIRSTNME	LASTNAME	PHONENO	ROW CHANGE TIMESTAMP
x'04004001000000000000F	CHRISTINE	141285673714388072	<del>CHRISTINE</del>	HAAS	1092	2007-12-20 18:22:25.593000
x'05004001000000000000F	MICHAEL	141285673726689984	<del>CHRISTINE</del>	THOMPSON	9012	2007-12-20 18:22:37.312000
x'06004001000000000000F	SALLY	0001-01-01 00:00:00.000000	74904229642240	KWAN	4738	

**Scenario 6**

The EMPLOYEE table has a ROW CHANGE TIMESTAMP column and is accessed only by Manager1. Manager1 selects some rows and tries to update them. However, between his select and the updates, the table has been reorganized offline. Later, when Manager1 tries to update the data, the update fails. The rows are not updated since the ROW CHANGE TIMESTAMP column has changed as the result of the REORG.

**Listing 19. Reorganizing table employee**

```
REORG TABLE EMPLOYEE
```

**Table 15. Changes after reorganizing table EMPLOYEE**

RID_BIT	ROW CHANGE	EMPNO	FIRSTNME	LASTNAME	PHONENO	ROW CHANGE
---------	------------	-------	----------	----------	---------	------------

TOKEN				TIMESTAMP
x'04004001000000000000F	CHRISTINE	141285781563232400	HAAS 3978	2007-12-21 11:29:30.250000
x'05004001000000000000F	MICHAEL	141285781563232401	THOMPSON 3476	2007-12-21 11:29:30.250001
x'06004001000000000000F	SALLY	141285781563232402	KWAN 4738	2007-12-21 11:29:30.250002

**Listing 20. UPDATE statement, which is executed after for the table REORG has run (Manager1)**

```
UPDATE EMPLOYEE SET
  (FIRSTNAME, LASTNAME, PHONENO) = ('CHRISTINE', 'HAAS', '1092')
WHERE RID_BIT(EMPLOYEE)=x'0400400100000000000000000000FA9023' AND
ROW CHANGE TOKEN FOR EMPLOYEE=74904229642240
```

**Result from Manager1 trying to update data after an REORG has been run:**

The update from Manager1 is unsuccessful since the ROW CHANGE TOKEN predicate fails while comparing the token retrieved at the time of the SELECT and currently, as the table has been offline reorganized by another task between the SELECT and the UPDATE of Manager1. Thus the UPDATE statement fails to find the row with the ROW CHANGE TOKEN that was retrieved before the REORG occurred.

**Conclusion**

In order to avoid lock waits that may occur when using a pessimistic locking strategy, DB2 9.5 optimistic locking minimizes the time for which a given resource is unavailable for use by other transactions. Because the database manager can determine when a row is changed, it can ensure data integrity while limiting the time that locks are held. With optimistic concurrency control, the database manager releases the row or page locks immediately after a read operation.

DB2 9.5 adds support for easier and faster optimistic locking with no false positives. This support is added using the row identifier (RID\_BIT or RID) built-in function, the ROW CHANGE TOKEN expression, the time-based update detection, and implicitly hidden columns. Applications using this programming model benefit from the enhanced optimistic locking feature and gain improved concurrency.

# Resources

## Learn

- [WebSphere sample \(OptimisticLocking.war\)](#): Try out a demo WebSphere sample application for DB2 9.5 optimistic locking.
- [IBM DB2 9.5 product page](#): Find out more about DB2 9.5, and its features and benefits.
- [DB2 9.5 Information Center: What's new](#): Get information about new features and functionality available with DB2 9.5, as well as changes in existing functionality from previous versions.
- [developerWorks Information Management zone](#): Learn more about Information Management. Find technical documentation, how-to articles, education, downloads, product information, and more.
- Stay current with [developerWorks technical events and webcasts](#).
- [Technology bookstore](#): Browse for books on these and other technical topics.

## Get products and technologies

- [DB2 9.5](#): Take DB2 9.5 for a test drive.
- Build your next development project with [IBM trial software](#), available for download directly from developerWorks.

## Discuss

- [Participate in the discussion forum for this content](#).
- Participate in [developerWorks blogs](#) and get involved in the developerWorks community.

## About the author

Werner Schuetz

Werner Schuetz is an IBM Certified IT Specialist, IBM Certified Advanced Database Administrator, and Certified Application Developer for DB2 9 for Linux, UNIX, and Windows. He works as a DB2 Technical Consultant in the IBM Innovation Center in Stuttgart, Germany. The [IBM Innovation Center](#) offers Independent Software Vendors (ISVs) cross-platform technical application enablement support for porting, testing, and migration. In this context, Werner Schuetz assists ISVs in testing DB2 solutions, executing performance and tuning sessions, and running competitive database migrations.