triton consulting

Db2 for z/OS–locking for Application Developers

By Gareth Copplestone-Jones

triton.co.uk

Contents

Preface 3

Introduction 3

Data Integrity, Performance and Locking 4

The Characteristics of Database Transactions 5

Db2 Transaction Locking Semantics 7
- Lock Size 7
- Lock Modes 10
- Lock Duration 14

Isolation Levels 16

Data Anomalies 18
- Phantom row or phantom read anomaly 18
- Non-repeatable read anomaly 20
- Dirty read anomaly 21
- Lost update anomaly 22
- Isolation Levels and Data Anomalies 22

Cursors and Data Currency 23
- For-update, Read-only and Ambiguous Cursors 23
- Acquiring and releasing locks for read-only cursors 24
- Lock Avoidance and the CURRENTDATA Bind Option 24
- Combining Read-only Cursors with Searched Updates 25
- Access-Path Dependent Cursors 27
- Locking With Searched Update and Delete 27

Update Applications and Data Integrity 28
- A Sidenote on Optimistic Concurrency Control 32

Additional Considerations 32
- Row level locking 32
- Materialised result sets 33
- RR/RS and non-materialised result sets 33
- Lock avoidance 34

Conclusion 35

Preface

This technical paper is based on the Triton 10-part blog, also titled 'Db2 for z/OS-locking for Application Developers', but includes some additional material and addresses comments on the original blog.

The paper is mostly targeted at Db2 for z/OS application developers, but it's also appropriate for application designers, database administrators, Db2 systems programmers and other Db2 professionals who are concerned with data integrity and aspects of application performance related to data integrity and therefore locking.

All references and hyperlinks to the IBM Db2 for z/OS documentation have been updated to point to the Db2 13 documentation, unless otherwise explicitly stated.

Introduction

While this document is called 'Db2 for z/OS-locking for Application Developers', it's really mostly about data integrity.

In essence, the whole point of locking is just that – enabling the application programmer to ensure data integrity. This document concentrates on the locks owned by *database transactions*, the implications for the application programmer, and the impact those locks have on performance.

While this paper mainly focuses on update transactions in an online transaction processing (OLTP) system, it's also relevant to read-only queries and transactions in OLTP systems, and in some online analytical processing (OLAP) environments. For example, those OLAP systems which are updated in near-real time by data replication processes.

It's important to emphasize that data integrity and locking-related performance are not solely the responsibility of the database management system (DBMS). No DBMS can guarantee data integrity and performance if application programs aren't coded correctly. To complicate matters for application developers, not all DBMSs use the same locking mechanism. If you're developing for Db2 for z/OS, it's therefore important for you to understand Db2 locking, the ways it is affected by the Db2 BIND options and the consequent implications for the way you code your application programs. In reality, data integrity is the responsibility of the application designer, the application developer and the database administrator, by implementing appropriate locking options and coding application programs in accordance with those options.

This paper, therefore, is intended as a practical guide as to when, how and why Db2 locks data. It provides guidance on programming techniques designed to avoid logical data corruption whilst minimizing the locking performance overhead given the data integrity requirements of the application.

This document doesn't cover everything you might want to know about locking – there is a whole series of topics which aren't covered, including:

- Drains and claims
- Restrictive states
- Utility compatibility
- Latches
- LOB locks and XML locks

Data sharing locking is covered, but only in sufficient detail to inform the discussion about data integrity and application performance.

The opening sections on data integrity and on database transactions lay the groundwork for the more practical discussions later in the paper. There's a fair bit of terminology to define and some first principles to outline, particularly the characteristics of database transactions. This discussion will help you understand the "why" of locking in a DBMS. We then move onto describe how these principles are implemented in Db2 itself in the form of Db2 transaction locking semantics, including lock size or scope, lock mode, and lock duration. This includes the central concept of hierarchical locking and the role it plays in enabling the acquisition of locks of different sizes on database objects, and at the same time ensuring that thousands of transactions can run concurrently. This brings us onto another crucial concept, transaction isolation levels, and following on from that, four of the most common data anomalies that database transactions can be exposed to, depending on the isolation level in use.

These lengthy but essential descriptions lead us onto two key topics for application development: result set materialization with cursors and the effect on data currency; and best practice for coding update applications which guarantee data integrity. The paper concludes with some additional considerations for application programming techniques and for lock size selection in special cases.

Data Integrity, Performance and Locking

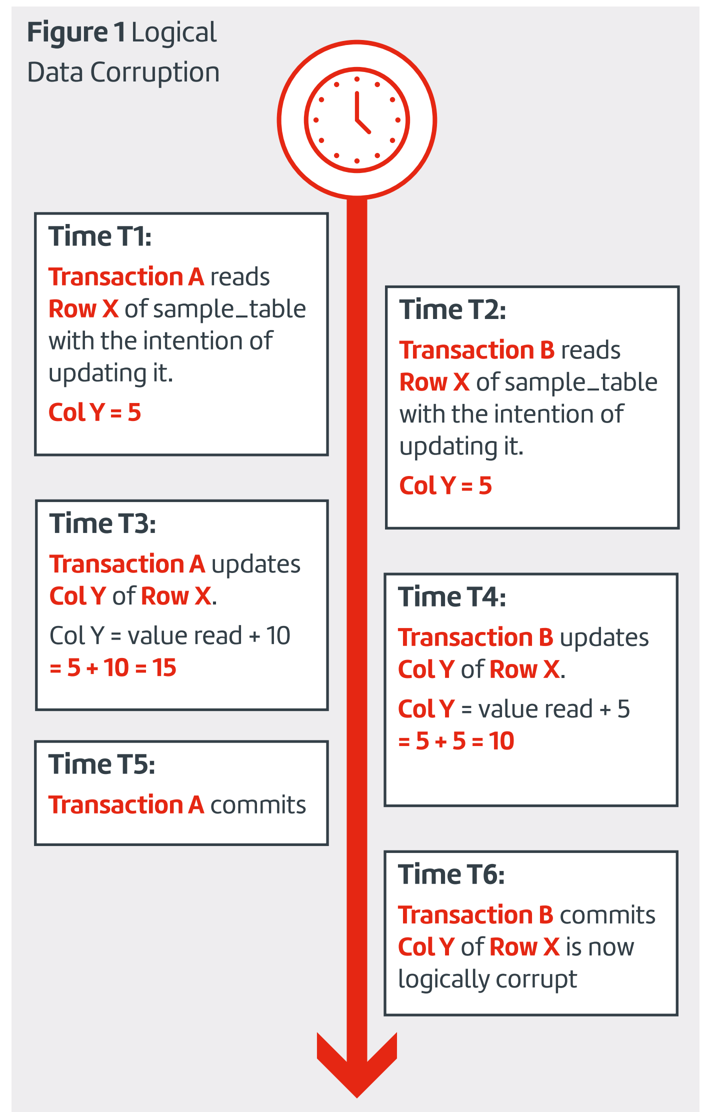
There are two main reasons why you should be concerned with locking: ensuring data integrity; and application performance, which includes CPU consumption, transaction concurrency, and throughput.

Regarding data integrity, if data is not locked, it's impossible to stop two (or more) applications updating the same data records at the same time, inevitably leading to data integrity problems. Without integrity, the value of data is severely downgraded, and could affect the viability of a business. Businesses that are unable to maintain an accurate ledger of financial transactions probably wouldn't stay in business for very long.

What can happen if data is not locked can be illustrated with a simple diagram, where time proceeds in the direction of the arrow:

To summarise, two transactions, Transaction A and Transaction B read table "sample_table" with the intention of updating column Col Y of Row X. When they read Row X, Col Y has a value of '5'. Let's say that Transaction A adds 10 to that value in program working storage and makes a database call to update the value of Col Y of Row X to 15. Transaction B is unaware of the update to Col Y of Row X, and from the point of view of Transaction B, Col Y still has a value of 5. Transaction B adds 5 to that value in program working storage and makes a database call to update the value of Col Y of Row X to 10, overwriting the update made by Transaction A. Col Y of Row X is now logically corrupt.

Figure 1 Logical Data Corruption



This resulting logical data corruption can be avoided by controlling access to the data by using locks. If, taking the diagram above (which we'll see again in a more detailed discussion), Transaction B wants to access data that has been locked and updated by Transaction A, then because Transaction A's lock prevents any other concurrent transactions from reading the data, the database manager makes Transaction B wait until the lock taken by Transaction A is released. This is known as a *lock suspension* or *lock wait*. When Transaction A completes its update and releases the lock, Transaction B can continue processing and sees Transaction A's updates. This process of making transactions wait for locks on database objects and allowing them to continue when those locks can be acquired is known as the *suspend/resume* process. Suspend/resume not only adds to elapsed times, but it also adds to CPU times.

Furthermore, as well as any CPU overhead incurred in the suspend/resume process, taking a single lock requires a very small amount of CPU. Whilst a single lock on its own might not incur significant overhead, taking many locks can, adding to the CPU cost and increasing elapsed times. Therefore, the ideal locking strategy is to lock all the data that needs to be locked in order to guarantee data integrity, but for performance reasons to acquire only the locks absolutely needed, and to hold those locks no longer than necessary, to minimize contention and lock waits.

You've probably noticed that at time T1, that Transaction A reads a row *with the intention of updating it*, as does Transaction B at time T2. The concept of *intent locks* is an important one for facilitating the acquisition of locks of different sizes on a given database object whilst at the same time enabling transaction concurrency. We'll cover intent locks in the section on Db2 Locking Semantics. You've also probably realized that, in this example, for the lock to be effective in protecting data integrity, Transaction A needs to acquire the lock on Row X at time T1 and prevent Transaction B acquiring itS-lock until after time T5, when Transaction A commits. This is another key concept: acquiring and releasing locks at the right time is vital for ensuring data integrity.

Locking in Db2 is more sophisticated than this simple example. But before we get into detail about Db2 locking itself, it's necessary to discuss the characteristics of database transactions.

The Characteristics of Database Transactions

We all instinctively know that locking is required for data integrity, but to understand the implications for application programs, we need to identify what are the characteristics required of database transactions to guarantee data integrity.

Back in 1983, German computer scientists Andreas Reuter and Theo Härder wrote a landmark paper, *"Principles of transaction-oriented database recovery"* in which they set about establishing a framework for "transaction-oriented recovery schemes for database systems". In doing so, they answered the question: what are the characteristics of a database transaction required to guarantee data integrity? Their paper was based on real-world experience of transactional systems in a shared database environment, from which the authors were able to derive a standard and time-tested set of required properties of database transactions or logical units of work (LUW) that guarantee data integrity and consistency even in the event of errors, power failures, and so on. These database transaction properties are known by the acronym **ACID**[1] (you can find a copy of the article in the Association for Computing Machinery Digital Library). Those properties are:

- **Atomicity:** either all of a transaction's updates are made to the database, or none of them are.
- **Consistency:** the database manager must enforce all defined consistency rules; the data must be changed from one consistent state at the start of a transaction to another consistent state at the end.
- **Isolation:** all transactions must be executed, from a data point of view, as though no other transactions are being executed at the same time.
- **Durability:** once a transaction is committed, then its updates are preserved, irrespective of any system failures.

1 Reuter and Härder built on the material in an earlier, pivotal paper by Dr Jim Gray, "The Transaction Concept: Virtues and Limitations" which included the properties of atomicity, consistency and durability, but not isolation.

Reuter and Härder's stated objective was to provide a "clear methodological and terminological framework" for database *recovery*. From their point of view, it's the responsibility of the DBMS to provide logging, recovery, backout and crash restart capabilities for the day-to-day operation of the database, and this is something Db2 does, along with many other database management systems. In terms of database recovery, the database transaction is an important concept as transaction synchronization points provide the basis for consistent database states.

However, as Reuter and Härder make clear, integrity of the data being read and updated in a multiuser environment depends on preventing "uncontrolled and undesired interactions" between concurrent database transactions. The most obvious way of preventing those uncontrolled interactions is by locking the data so that concurrent transactions can't act on the same data. As we'll see, the application programs must be written in such a way as to exploit the features (primarily locking-related) provided by the DBMS to guarantee data integrity. If application programs aren't coded using the correct techniques, then the possible consequences include the *data anomalies* discussed later in this paper.

This paper mostly concentrates on the properties or principles of atomicity and isolation, because the application program and the database server share the responsibility for ensuring compliance with these two principles. Durability can be regarded primarily as the responsibility of the DBMS itself, ensuring that when a transaction commits the changes are made permanent in the database. This is achieved by logging database updates and by writing them to disk. Consistency from an application programming point of view is not discussed in any detail, because this arises from standard good programming practices. For example, when performing a funds transfer between two accounts, the application should ensure that the sum of the two account balances at the start of the transaction is identical to that at the end.

In considering database transactions properties, the next question is, what is a database transaction? A database transaction is a set of interactions between the application and the database, where the database is changed from one consistent state to another. That is, a transaction consists of a set of database reads and updates which form a logically consistent unit of work such as a bank balance transfer from one account to another. A database transaction ends when the application indicates to the DBMS that it wants to make its database changes permanent by issuing a COMMIT. If the transaction terminates before it issues a COMMIT, then the database manager must backout all the updates the transaction made, to ensure the database remains in a consistent state. This complies with the Atomicity principle, where either all the database updates of a transaction are made permanent, or none of them are.

The challenge is to integrate the ACID properties *and* high performance into the design of database transactions. While the isolation property states that all transactions must be executed *from a data point of view* as though no other transactions are being executed at the same time, in a real-world system multiple database transactions must be able to run concurrently against the same database.

To explain the transaction isolation/concurrency challenge simply, let's take a couple of straightforward examples. Firstly, if a database table is locked for update by a transaction, no other transaction can even read the table while the lock is held. Readers must wait until the update (exclusive) lock is released. Secondly, if a table is locked for read, no-one else can update the table while the lock is held. Other transactions can read the table, but update transactions must wait until the read (share) lock is released.

It would be easy to meet the demands of transaction isolation by running database transactions one at a time, serially. Apart from the fact that this is an inefficient use of expensive computing resources, it would prevent enterprises from running the high-volume, high concurrency workloads required to conduct their business. The pragmatic solution to this is for the DBMS to allow the application designer to choose the degree of transaction isolation and push the responsibility for data integrity and performance onto the application programmer. Db2, like other database systems, does this by providing multiple *transaction isolation levels* which the application designer can choose from. These are discussed in detail in the section on Isolation Levels.

The various transaction isolation levels strongly influence the way the DBMS locks data on behalf of the application. There are four lock attributes that together with the transaction isolation level determine the balance between the ideal goal of transaction isolation ("all transactions must be executed, from a data point of view, as though no other transactions are being executed at the same time") and concurrency:

- The size of the lock – whether it is held on a database, table, or row (for example). This is typically known as the *lock size*.
- The state of the lock – whether, for example, it is an exclusive lock or a share lock. This is known as the *lock state* or *lock mode*.
- When the lock is acquired.
- How long the lock is held – whether it is held until the end of the transaction or for a shorter period. This is known as lock *duration*.

We'll look at these in detail in the next section on Db2 transaction locking semantics.

Db2 Transaction Locking Semantics

This brings us onto the Db2 locking mechanism. The locking behaviour of a DBMS is often referred to as locking *semantics*, with each DBMS using its own semantics.

Oracle locking semantics, for example, are different from those used by Db2 for z/OS. In the case of Db2, a solid understanding of how it locks data by acquiring locks at the tablespace, table, page, or row levels is needed for you to be able to guarantee data integrity when striking the balance between transaction isolation and concurrency. The design point of standard[2] Db2 locking semantics is to always present committed data and only committed data to the application (with the *"dirty read"* exception, which is discussed later). That is, the data is always transactionally consistent – if an in-flight transaction has updated some data, other transactions wanting to access that data must wait until the first transaction issues a commit to tell Db2 to make its changes permanent. After the commit, Db2 releases the locks[3].

Given this, an implication is that Db2 decides what to lock, when to lock it, and when to release the lock, subject to the bind options and tablespace attributes in effect. This is mostly true, with a few exceptions such as the SQL DML statement LOCK TABLE IN SHARE MODE, which we'll come across later.

Lock Size

This brings us onto two key parts of Db2 locking semantics – *lock size* and *hierarchical locking*

Db2 allows you to choose the lock size by specifying whether to lock at the tablespace, table, page, or row level via the LOCKSIZE attribute of the CREATE or ALTER TABLESPACE DDL statement. To optimize transaction concurrency and to allow transactions to take locks of differing sizes on a given database object, Db2 uses a mechanism called hierarchical locking (see Figure 2), with tablespace/partition at the top of the hierarchy and page/row at the bottom. A lock is always acquired at the top level of the hierarchy, without exception. If you specify LOCKSIZE TABLESPACE or TABLE,[4] locks are only obtained at the top of the hierarchy. Locks are obtained at the bottom level in the hierarchy only if you specify LOCKSIZE PAGE or LOCKSIZE ROW (or LOCKSIZE ANY, which in most cases results in LOCKSIZE PAGE).

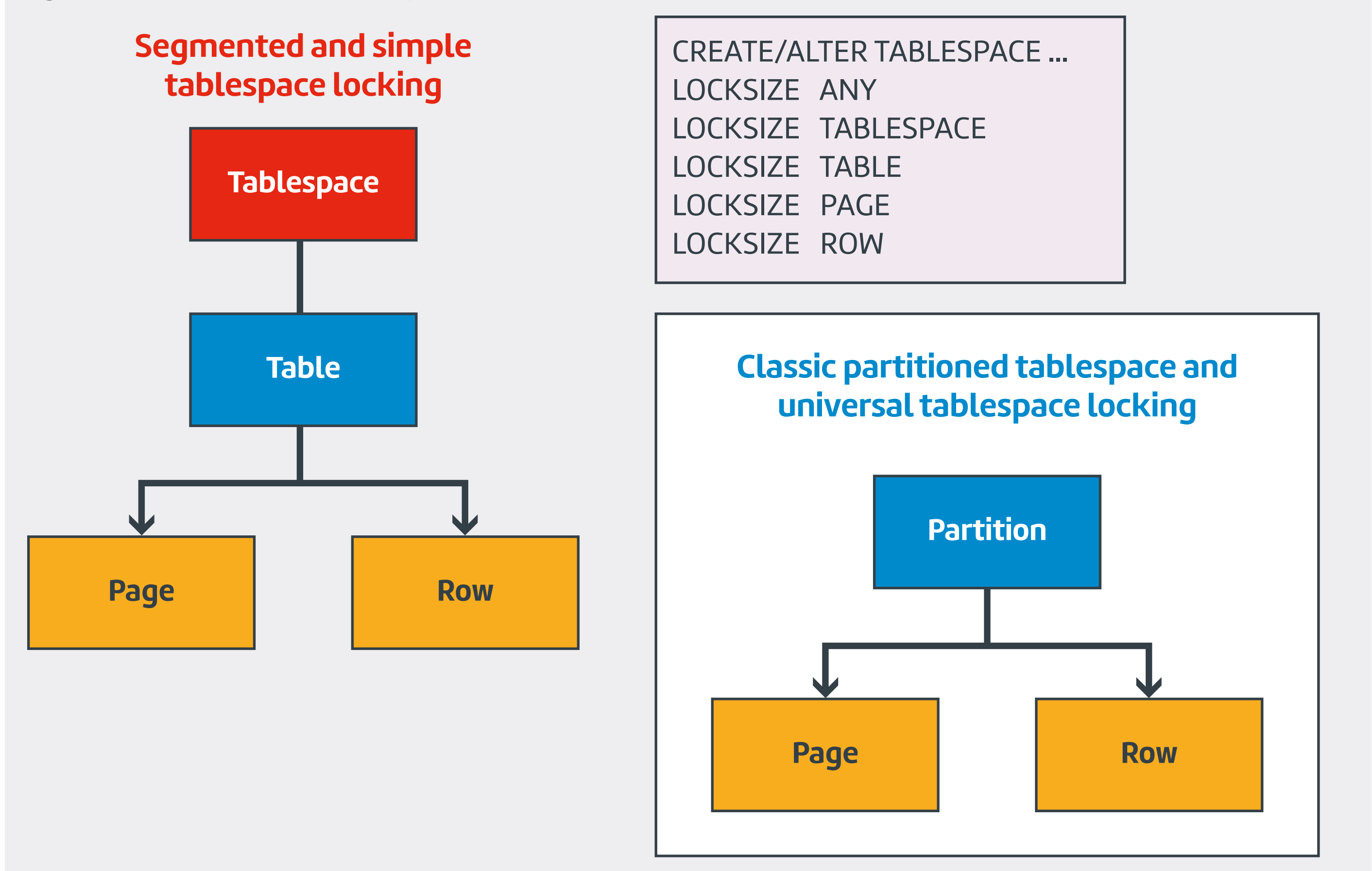
2 I say "standard" because Db2 10 for z/OS introduced limited support for another locking semantic called multi-version concurrency control, referred to in the Db2 documentation as "currently committed data". This was, at least in part, intended to make porting applications from other database managers easier.

3 Alternatively, if the application encounters an error it can't recover from, Db2 rolls back the all the changes made by the transaction to a prior point of consistency. A side effect of a roll back is that it extends the period of time locks are held, increasing any lock contention.

4 According to the Db2 SQL Reference, you should specify LOCKSIZE TABLE only for a segmented (non-UTS) tablespace. In practice, it only makes sense to specify this for a multi-table segmented tablespace, bearing in mind that these are deprecated as of Db2 12 function level 504, or for a multi-table simple tablespace – these have been deprecated since Db2 9 for z/OS

If you specify LOCKSIZE TABLESPACE, even if the tablespace is a classic partitioned tablespace, a UTS partition by range (PBR) tablespace, or a UTS partition by growth (PBG) tablespace, Db2 locks all partitions. However, if you specify LOCKSIZE PAGE or ROW for those type of tablespace, top-level locks are obtained at the partition level, not the tablespace level, as illustrated in the green box in **Figure 2**.

Figure 2 Db2 Hierarchical Locking



Tablespace locks for LOCKSIZE TABLESPACE are sometimes known as *gross locks* because the application acquires a share or exclusive lock on the entire object. If you specify LOCKSIZE PAGE or ROW, you only acquire share and exclusive locks on the page or row, but you also acquire locks on the tablespace or partition known as *intent locks*. You can also get gross locks even with LOCKSIZE PAGE/ROW by coding EXEC SQL LOCK TABLE IN SHARE/EXCLUSIVE MODE.

What's the difference between a gross lock and an intent lock? To expand on the definition of a gross lock as a lock on the entire object (tablespace, partition, or table), it's a way of implicitly obtaining that lock on every single page or row in the tablespace without the CPU and elapsed time cost of acquiring all those locks. Gross locks can be useful for performance reasons in a number of circumstances, including:

- If you know that the tablespace is read-only and is never updated when it is being read. For example, if a reference table only needs to be updated once a day, replacing the contents of the tablespace using the LOAD utility is possible if the daily schedule shuts the read-only application down long enough for the table to be updated.
- If you know that an application program is the only one accessing the table, then it's safe to take an exclusive lock preventing any other user from accessing while the program is running while at the same time minimizing the CPU and elapsed time cost of locking.

In all these cases, If the number of pages/rows accessed is small, it might be difficult to measure the CPU and elapsed time savings. On the other hand, if the number is large, the benefits can be very significant.

While applications accessing that tablespace will run more quickly and cheaply than if they acquired page or row locks, there is a significant impact on concurrency. Share-mode gross locks prevent any concurrent transactions from updating the table, and exclusive-mode gross locks prevent any other concurrent transactions from accessing the table at all. Be aware that gross locks are held until the transaction commits or terminates.[5]

An intent lock is very different. It signals a database transaction's intention to read or update a tablespace concurrently with other transactions which are reading or updating the tablespace. This is dependent on them also taking intent locks[6] which indicate that the transactions might also hold locks on other database objects at a lower level in the hierarchy. With Db2 locking semantics, intent locks are essential if there is concurrent access to a tablespace involving updates as well as reads. Given that most Db2 for z/OS databases are used for OLTP, most tablespaces are defined with LOCKSIZE PAGE or ROW. There is an option to specify LOCKSIZE ANY, and although this nearly always resolves to LOCKSIZE PAGE, it is recommended that you explicitly specify the lock size, if only for documentation purposes. Like gross locks, intent locks are held until the transaction terminates or commits (depending on the package bind option – see the later discussion on the RELEASE package attribute in the section on Lock Duration). Note that you cannot explicitly request intent locks – these are always acquired implicitly when an application requests a page or row lock.

The theoretical underpinning of the concept of intent locks as a necessary feature of hierarchical locking is presented in a 1976 IBM paper by J. Gray, R. Lorie, G. Putzolu and I. Traiger, "Granularity of Locks and Degrees of Consistency in a Shared Data Base". In this paper, they demonstrate that the introduction of hierarchical locking allows a transaction requiring locks of a small scope (such as page, row, or record) to implicitly acquire an intent lock (which they call an "intention lock") at a higher level in the hierarchy, indicating to the lock manager (the DBMS) that the transaction can also acquire locks at a lower level in the hierarchy. This solves the problem of how the DBMS can resolve the compatibility of locks of different scopes or sizes such as in the case where one transaction needs a lock at the table level and another at the row level. In the terms of the authors, hierarchical locking provides a "locking protocol which allows simultaneous locking at various granularities by different transactions", where they use the term 'granularity' to refer to lock scope or size. In the case of Db2, this is done by checking, for example, if a request for a share or exclusive lock at the tablespace or partition level is compatible with an existing intent lock on the same tablespace or partition.

Locks on pages and rows are not *categorized* as either gross or intent locks, because gross locks and intent locks only apply at the object level (table space, table, or partition). With LOCKSIZE PAGE or ROW, concurrent transactions can take share, exclusive or update locks (see the next section on lock modes for information about update locks) on pages or rows in the same table. Assuming LOCKSIZE PAGE, if Transaction A acquires a share lock on page 1500, other concurrent transactions can acquire share or update locks on the same page, but not exclusive locks.

This brings us onto the question of why you should ever use LOCKSIZE PAGE instead LOCKSIZE ROW.
The answer to that depends on several factors:

- In most cases, LOCKSIZE ROW incurs more overhead in a data sharing environment, so LOCKSIZE PAGE is preferred in data sharing where possible.
- If your application is likely to access several rows in the same page, page level locking requires fewer locks and less CPU time.
- If your application accesses a single row per page, the CPU overhead is identical for LOCKSIZE PAGE and LOCKSIZE ROW.
- LOCKSIZE ROW only has an advantage where concurrent transactions require locks on different rows in a given page, where the locks would be incompatible with LOCKSIZE PAGE.

Essentially, page level locking is a bet that that level of lock granularity is fine enough to avoid locking conflicts. In most applications, this works well, but if you experience excessive contention then LOCKSIZE ROW may be justified.

This brings us onto the discussion on lock modes.

5 The BIND option that affects this, RELEASE(COMMIT|DEALLOCATE) is discussed later in this paper.
6 This is with the exception of the rare SIX lock, a sort of hybrid gross/intent lock that is described later.

Lock Modes

With standard Db2 locking semantics, the *lock mode* or *lock state* determines the degree of concurrent access allowed to a row, page, table, or tablespace; for example, whether multiple transactions can concurrently read the same data row. Different lock modes are acquired depending on whether the database object (tablespace, page, or row) is being updated or read, and on the lock size.

Now, some lock modes are only available at the tablespace level, some are available at all levels in the hierarchy, but the effect on concurrency varies considerably depending on whether gross locks or intent locks are taken at the tablespace level. The design default for all tablespaces should be LOCKSIZE PAGE (or ROW where justified, of which more later), as this causes Db2 to acquire intent locks rather than gross locks at the tablespace level, allowing much greater concurrency.

With the exception of the SQL LOCK TABLE statement and some other rarely used SQL clauses (for example, USE AND KEEP UPDATE LOCKS), Db2 selects the lock mode for the object based on the LOCKSIZE attribute and the SQL DML statements that reference the object. We'll go through the lock modes in some detail before explaining how the SQL statement determines the lock mode chosen.

The lock modes available at the partition, tablespace, and table levels only are:

- **IS** (Intent Share)
- **IX** (Intent Exclusive)
- **SIX** (Share with Intent Exclusive)

The lock modes available at the partition, tablespace, table, page, and row levels (that is, all levels of the locking hierarchy) are:

- **S** (Share)
- **X** (Exclusive)
- **U** (Update)

There are some subtle differences between the way the lock modes operate at the top of the hierarchy and the bottom, so we'll deal with lock modes at the tablespace level first and lock modes at the page and row levels afterwards.

Tablespace lock modes

The two main intent lock modes, **IS** and **IX**, provide the best transaction concurrency, as they are the least restrictive and rely on page/row level locks to ensure data integrity. Intent locks are only acquired when the tablespace is defined via the CREATE or ALTER SQL DDL statements with the LOCKSIZE PAGE or LOCKSIZE ROW attribute.

The **IS** (intent share) lock indicates that the application is accessing the tablespace with the intention of reading one or more rows. An application that acquires an **IS-lock** can read but not change data in the tablespace. It might also acquire page or row locks. Concurrent processes with **IS** or IX-locks can read and change the data using page or row locks – this is a key advantage of the **IS-lock**. IS-locks are chosen with LOCKSIZE PAGE/ROW if Db2 can detect at BIND time (or PREPARE time for dynamic SQL) that the table is only being accessed for read.

The **IX** (intent exclusive) lock indicates that the application is accessing the tablespace with the intention of reading and possibly updating one or more rows. An application that acquires an **IX-lock** can both read and change data in the table and might also acquire page or row locks – a page or row lock is always required on any data changed. Concurrent processes with **IS** or IX-locks can read and change the data, using page or row locks – as with IS-locks, this is a key point of the **IX-lock**. IX-locks are chosen with LOCKSIZE PAGE/ROW if Db2 detects at BIND time (or PREPARE time for dynamic SQL) that there is an actual or possible intent to update the table; for example, if the application program includes a cursor defined with a FOR UPDATE OF clause or an INSERT statement.

The two main gross lock modes, **S** and **X**, provide very limited concurrency, and are acquired with LOCKSIZE TABLE/TABLESPACE or the SQL DML statement LOCK TABLE IN SHARE/EXCLUSIVE MODE. Because page/row locks are not needed, applications acquiring **S** and X-locks use less CPU resource.

The **S-lock** (share) allows the application to read, but not change, data in the tablespace. Concurrent processes acquiring **S**, **IS** or **U** tablespace locks can read but not change the data.

The **X-lock** (exclusive) allows the application to read and change data in the tablespace. No other concurrent processes can change or read the data.

There are two other locks which can be acquired at the tablespace level, but they are rare cases. These are the **U** and SIX-locks.

The **U-lock** (update) is rare at the tablespace level, as it requires LOCKSIZE TABLE or TABLESPACE and a cursor-based select with a FOR UPDATE OF clause. There is no such SQL statement as LOCK TABLE IN UPDATE MODE for acquiring a tablespace **U-lock**. An application with a **U** tablespace lock can read but not change locked data: when the application tries to change the data, Db2 attempts to promote the **U** to an **X** tablespace lock. Like the **S** and **X** tablespace locks, the **U** tablespace lock does not need page or row locks. Concurrent processes can acquire **S** or IS-locks and read the data, but they cannot acquire any kind of update lock.

The **SIX-lock** (share with intent exclusive) is possibly even rarer and is acquired when the application already holds an **IX lock**, then issues the SQL statement LOCK TABLE IN SHARE MODE. The holder of a **SIX lock** can read and change data in the table, but only when data is changed are page or row locks acquired. Concurrent processes can read data in the tablespace but not change it.

The following table, taken from Db2 13 for z/OS online documentation, illustrates the compatibility of any two lock modes for partition, tablespace or table locks :

Lock mode	IS	IX	S	U	SIX	X
IS	Yes	Yes	Yes	Yes	Yes	No
IX	Yes	Yes	No	No	No	No
S	Yes	No	Yes	Yes	No	No
U	Yes	No	Yes	No	No	No
SIX	Yes	No	No	No	No	No
X	No	No	No	No	No	No

If two or more different database transactions hold compatible intent locks, it indicates to Db2 that it will need to check the compatibility of the page or row locks held by those transactions at a lower level in the locking hierarchy

Page and row lock modes

Page and row-level locks are much simpler but bear in mind that the holder of a page or row lock must also hold an **IS**, **IX**, or **SIX-lock**.

An application holding a page/row **S-lock** can read but not change the locked page or row. Concurrent processes can also read but not change the locked page or row, and they can acquire S or U-locks on the page/row or might read data without acquiring a page or row lock (see the later discussion on lock avoidance).

Similarly, an application holding a page/row **U-lock** can read, but not change, the locked page or row; however, when the application attempts to change the page/row, Db2 attempts to *promote*[7] the U-lock to an **X-lock**. Concurrent processes can acquire S-locks or read data without acquiring a page or row lock but cannot acquire a **U** or **X-**lock. U-locks reduce the chance of deadlocks when reading a page or row to determine whether to change it.

This table shows the compatibility matrix for page/row locks:

Lock mode	S	U	X
S	Yes	Yes	No
U	Yes	No	No
X	No	No	No

An application holding a page/row X-lock can read or change the locked page or row, and concurrent processes cannot acquire S, U, or X-locks on the page or row. Conversely, an application cannot acquire an X-lock on a page/row if a concurrent process already holds an S or U-lock on that page or row.

Bear in mind that there is no compatibility matrix for page/row and tablespace locks. One of the principles of hierarchical locking is that lock requests are only ever checked for compatibility with other locks at the same level in the hierarchy.

Incompatible locks

Let's briefly consider incompatible locks. What happens if two transactions want to access the same data with incompatible lock modes? For example, transactions A and B both want to update row Z, and transaction A is the first to issue an UPDATE statement and acquire an **X-lock** on row Z. Now, transaction B must wait until transaction A has committed its update before even reading row Z. Db2 detects that transaction B needs a lock on the row, and suspends transaction B, forcing it to wait for transaction A to commit at which point the lock can be acquired and transaction B can continue processing. Depending on how long a transaction continues processing between acquiring a lock on a row and releasing it, other transactions can simply be suspended waiting for a lock, elongating elapsed times, or can be the victim of a lock timeout or a deadlock.

Deadlocks

If you don't already understand the concept of a deadlock, consider the example where transaction A has acquired an exclusive lock on row Z and wants a share lock on row Y. However, transaction B has acquired an exclusive lock on row Y and wants a share lock on row Z. We now have the situation where transaction A is waiting for a lock on row Y but transaction B holds an incompatible lock on that row, and transaction B is waiting for a lock on row Z but transaction A holds an incompatible lock on that row. Without intervention, the two transactions will wait forever. Db2's deadlock detection mechanism identifies deadlocks very quickly and chooses one of the transactions as a "victim". Db2 abnormally terminates that transaction and rolls back any updates to the start of the transaction (or previous commit), meaning the other transaction can continue processing.

To express this more formally, a deadlock occurs when:

- Transaction A holds a lock on a database object X (tablespace, page, or row) which is incompatible with a lock requested, also on object X, by Transaction B.
- At the same time, Transaction B holds a lock on a database object Y which is incompatible with a lock requested, also on object Y, by Transaction A.

7 When Db2 tries to change the mode of lock to a more restrictive one, this is known as lock promotion. There are some cases where Db2 can scale back a lock mode to a less restrictive one. This is known as lock demotion.

Deadlock situations can be more complex and therefore more difficult to diagnose, involving more than two transactions and more than two database objects, but the principles are the same.

Timeouts

The concept of lock wait timeout is much simpler. Db2 system parameter IRLMRWT determines how long a transaction should wait for a lock before Db2 abnormally terminates it and rolls it back. The longer a transaction holds a lock, the likelier it is that other transactions will timeout while waiting for a lock on that page or row.

Limiting the number of locks held by a transaction

If there are many page or row locks on a given tablespace, that can stress the system for reasons including the memory and CPU resource required to manage those locks. To limit those effects, Db2 provides the system administrator with two controls. The first control, the system parameter (ZPARM) *NUMLKTS* specifies a system-wide limit on the maximum number of locks that can be acquired on a tablespace. This is a default attribute which can be over-ridden at the tablespace level by the second control, the attribute *LOCKMAX*. If the number of page or row locks exceeds this value, then *lock escalation* occurs. With lock escalation, Db2 attempts to release the page or row locks, and then acquire a tablespace gross lock to replace the intent lock. That is, Db2 attempts to replace an IX-lock with an X-lock, and an IS-lock with an S-lock.

In one way this is good, because it can improve performance by reducing the CPU cost of locking and in some specific cases can work well. On the other hand, lock escalation usually impacts the production service adversely, because it inhibits concurrency and increases the chance of lock contention, timeouts, and deadlocks. A transaction holding an IS-lock on a tablespace is compatible with other transactions holding IX-locks, whereas a transaction holding an S-lock is not.

Because lock escalation can be disruptive, Db2 records every occurrence in the Db2 System Services message log with message DSNI031I in the as illustrated in **Figure 3**. Many Db2 sites use automation to capture these messages and report on them so that action can be taken to avoid lock escalation.

Figure 3 Lock Escalation message

```
01.28.07 STC46358  DSNI031I  +DBA1 DSNILKES - LOCK ESCALATION HAS  274
   274             OCCURRED FOR
   274               RESOURCE NAME = XYZDM0C.SGET_G2LTORAN
   274               LOCK STATE =  X
   274               PLAN NAME : PACKAGE NAME = ABCDPB01 : A2YZAG10
   274               COLLECTION-ID = ABCDKB01
   274               STATEMENT NUMBER = 00000212
   274               CORRELATION-ID = EFGHIJ15
   274               CONNECTION-ID = BATCH
   274               LUW-ID = XXXPT.DB09DBA0.D3CEF94F0DE0
   274               THREAD-INFO = GGG9OPC:BATCH            :GGG9OPC          :EFGHIJ15
   274                                   :STATIC :656195:*
```

Lock size recommendations

This brings us on to some lock size recommendations:

- When coding for concurrent access to Db2 data, you should avoid LOCKSIZE TABLESPACE, as this is very likely to serialize access to the tablespace. Exclusive (X) locks definitely will, while share (S) locks allow concurrent read access but no updates.
- Use LOCKSIZE PAGE as a design default. In most cases, this will provide the best balance between locking the data for integrity reasons, transaction concurrency, CPU resource consumption and transaction performance.

- Use LOCKSIZE ROW where justified – some applications are dependent on LOCKSIZE ROW for concurrency reasons. The disadvantage with LOCKSIZE ROW is that it will increase the number of locks taken and therefore CPU overhead where many rows per page are accessed and is more likely to lead to lock escalation than LOCKSIZE PAGE.[8] However, if the application only accesses one row per page, then the number of locks taken is the same with LOCKSIZE PAGE and LOCKSIZE ROW. Nevertheless, bear in mind that LOCKSIZE ROW increases the data sharing overhead.
- Avoid lock escalation:
  - For long-running applications which update many rows, typically batch applications, commit frequently.
  - Set NUMLKTS and LOCKMAX realistically to prevent lock escalation without causing unnecessary application failures.
  - In some rare cases, allowing lock escalation can be an effective strategy. For example, where there is a clear distinction between OLTP and overnight batch processing.
- Avoid RR and RS isolation levels (of which more later).

There are no real lock mode recommendations: for LOCKSIZE PAGE or ROW, Db2 uses the least restrictive lock mode required to guarantee data consistency/integrity in line with the transaction isolation level in effect (discussed later) while maintaining transaction concurrency.

Lock Duration

This brings us onto the next characteristic of Db2 for z/OS-locking: *lock duration*. Lock duration is important not only for concurrency but also for atomicity.

There are two lock durations to consider: those for tablespace or partition locks; and those for page or row locks.

Tablespace locks, be they gross locks or intent locks are always acquired on first use, regardless of the setting for the ACQUIRE bind option. That is, when the first SQL request that references the associated table is issued. Lock duration for tablespaces/partitions is determined by the RELEASE option of the BIND/REBIND command, which has two options:

- RELEASE(COMMIT): with this options, tablespace and partition locks are held for the duration of the commit scope.
- RELEASE(DEALLOCATE): with this option, tablespace and partition locks can potentially be held across multiple commit scopes. The objective is to avoid the repeated CPU cost of frequently acquiring and releasing intent locks. There are two potential uses for RELEASE(DEALLOCATE):
  - Batch programs with intermediate commits, as discussed earlier.
  - High-volume transactions running under CICS protected entry threads, pseudo-WFI IMS transactions or high performance DBATs can hold tablespace locks (typically intent locks) across many transactions.

Like tablespace and partition locks, page and row locks are always acquired when the page or row is first accessed. That can be at cursor open time if the result set is materialised in a work-file (for example, if a sort is required) or at fetch time. In other words, when the locks are acquired is dependent on the access path. Be aware that with SELECT, UPDATE and DELETE, locks can be acquired on rows when they are evaluated, and not just on the qualifying rows; with UPDATE and DELETE they always are, whereas with SELECT this is dependent on the CURRENTDATA bind option. Depending on a variety of factors, if the rows qualify, the locks may be released, retained, or promoted. Which rows are evaluated is dependent on the access path.

8 Some applications require row-level locking to avoid contention, and some tablespace design options such as MEMBER CLUSTER with APPEND also require row-level locking (in the MEMBER CLUSTER case, to avoid exponential growth in the tablespace size resulting from massively parallel insert processing). LOCKSIZE ROW can impact performance as it increases the data sharing overhead.

How long are page and row locks held? Lock duration is determined by several factors, excluding factors such as lock escalation:

- The *isolation level* – see Isolation Levels, which details how long page and row locks are held for each isolation level.
- The lock mode (S, U, X).
- The CURRENDATA option of the BIND/REBIND command – see the later topic on cursors and data currency.
- Whether or not a cursor is declared using the WITH HOLD attribute.
  - When a cursor specification includes WITH HOLD, the cursor is not closed when the application issues a commit. In addition, any locks that are necessary to maintain the cursor position, including intent locks, and page locks or row locks, are held past the commit point, into the next commit scope.

With short lock durations, there is less potential for locking conflicts between concurrent applications, and a smaller impact in terms of lock suspension times and elapsed times. Exclusive locks, taken when data is changed, must be held until the commit point. This applies to all isolation levels and enforces compliance with the atomicity principle of ACID – that is, that either all of a transaction's updates are made to the database, or none of them are made. Before clarifying the role COMMIT plays for a transaction, let's consider what would happen if all the changes to data made by an application were made permanent in the database for every single SQL call.[9]

Consider a business process which makes several SQL calls when processing a funds transfer between two bank accounts, with autocommit in effect. That is, the transaction issues a commit after every database call. The business transaction requires at least two SQL calls to change the data, but probably more if it: reads the data before updating it; writes audit data to database tables; records historical data such as a transaction log; and so on. Imagine the worst case, a system failure part way through the business process (for example, a subsystem abend or a power failure). When the system comes back up the data no longer has integrity. The logical relationship between the database calls cannot be detected by the DBMS itself, which sees each call as a separate database transaction and logical unit of work. This situation is also extremely difficult for the application or the DBA to detect and correct. The same applies to transaction abnormal terminations.[10]

It's clear, therefore, that to comply with the atomicity principle, a database transaction must group its SQL calls into a logical unit of work demarcated by a commit scope. This brings us onto two clarifications about commits. Firstly, a *commit point* marks the end of a logical unit of work and indicates that all updates are to be made permanent in the database – from the DBMS point of view, the database transaction has complied with the atomicity, consistency, and durability properties. Well-behaved, long-running processes with restart capability, typically batch jobs, make intermediate commits to limit the number of locks held, to improve concurrency, and to reduce the length of the backout process in the event of a failure. Secondly, a *commit* scope is the period between two commit points (the start of the transaction can be regarded as an implicit commit point), or to put it another way, the period between two consistent database states.

To round out the topic on lock duration, we need to move onto the isolation levels supported by Db2 many other databases, and many transaction servers.

9 This is known as "autocommit", which is supported as a connection property for distributed clients using the IBM Data Server Driver. Autocommit is rarely used for update applications, but often used for read-only applications.

10 If autocommit is used for a business process that includes multiple update statements (INSERT, UPDATE, DELETE), then the business process needs to record every update SQL call it makes in an application log. This log will be needed for backout, restart and recovery to ensure the data is consistent from the business process point of view. All that's needed for guaranteed data corruption after a subsystem or transaction failure is for the failure to occur between an SQL call and the update to the transaction log that records the update SQL event.

Isolation Levels

The four isolation levels supported by Db2 (RR, RS, CS and UR) comprise the next important component of the Db2 locking mechanism.

They balance transaction isolation (“all transactions must be executed, from a data point of view, as though no other transactions are being executed at the same time”) and concurrency – multiple transactions potentially accessing the same data running alongside each other – with some isolation levels favouring strong transaction isolation and others favouring concurrency. Given the different degrees of transaction isolation, it becomes the responsibility of application programmers to comply with the principles embodied in the ACID properties by adjusting their programming techniques accordingly.

The four Db2 isolation levels are determined at the package level at BIND time for both static and dynamic SQL and control the trade-off between isolation and concurrency. The isolation level specified for a package can be overridden for individual SELECT statements by adding a ‘WITH UR/CS/RS/RR’ clause. Moreover, if you specify WITH RS or RR, you can specify which lock mode Db2 should use by additionally specifying USE AND KEEP EXCLUSIVE/UPDATE/SHARE LOCKS. Remember, however, that executing any UPDATE or DELETE statement will result in an exclusive lock being acquired.

The isolation level determines how long locks are held – lock duration – and to some extent which locks are held. This is also affected by the CURRENDATA BIND option, which will be covered later.

The four isolation levels are:

- Repeatable Read (**RR**)
- Read Stability (**RS**)
- Cursor Stability (**CS**)
- Uncommitted Read (**UR**)

These are the Db2 names, which is important because these are the terms used when binding a package or specifying the isolation level for a SELECT statement. Confusingly, ANSI[11] isolation level names are different from Db2’s. As we go through the isolation levels in order of least to most concurrency, I’ll also give you the equivalent ANSI isolation level names which may well be familiar to Java developers, together with some explanatory history.[12] These isolation levels became the industry standard in the ANSI/ISO SQL-92 specification.

The following discussion assumes the use of LOCKSIZE PAGE or ROW, with Db2 using intent locks rather than gross locks at the tablespace or partition level.

We’ll start off with **RR – Repeatable Read**, which is one of the two original Db2 isolation levels. If a single RR transaction runs the same query twice, it is certain to see the same set of values (no more and no less). An INSERT, UPDATE or DELETE by another concurrent transaction which would change the result set is not allowed (the current transaction is able to do so, however). In Db2, all accessed rows or pages are locked until the next commit point, even if they do not satisfy the predicate. That is, all evaluated rows are locked until the next commit. For page level locking, if a row on the locked page is not even evaluated, that row remains effectively locked until the next commit. Not surprisingly, the RR isolation level is more likely to lead to lock escalation, as this isolation level is likely to acquire the most locks.

11 Strictly speaking, the current standard is not an ANSI standard, but a standard written by an ANSI-approved committee, and the current standard is “The SQL Standard – ISO/IEC 9075:2016”. You can read more about it in this ANSI blog entry. Sadly, the content of the SQL Standard is only available to subscribers.

12 This topic is more complicated and controversial than presented here; the topic of database transaction isolation levels and potential data anomalies is widely debated in industry and academic circles. This paper is intended as a practical guide for Db2 for z/OS development, but for a broader discussion the 1995 ACM article “*A Critique of ANSI SQL Isolation Levels*”

The standard name for this isolation level is ***Serializable***, which makes sense if you remember that it appears to a repeatable read transaction that it's being executed, from a data point of view, as if no other transactions are being executed at the same time – that is, it appears as if the transactions are being executed serially.

The next isolation level is **RS – Read Stability**. This makes sure that, if a transaction reads the same row twice, it will have the same value, but does not prevent new rows from appearing during the execution of the transaction. That is, DELETE by another transaction is prohibited, but INSERT is allowed, as is an UPDATE which changes an existing, non-qualifying row such that it now qualifies for and appears as a new row in the result set.[13] All rows or pages satisfying any stage 1 predicates are locked until the application commits, plus all rows or pages evaluated during stage 2 processing, whether or not they qualify. Like RR, the RS isolation level can cause many locks to be held and therefore is more likely to lead to lock escalation than other isolation levels (apart from RR). Read Stability was introduced into Db2 after the SQL-92 standard was published. However, the SQL-92 name for this isolation level is ***Repeatable Read***.

The factor that most inhibits concurrency with both **RR** and **RS**, therefore, is the fact that both of them hold all page and row locks – S, U and X – from the time they are acquired to the next commit point, with U-locks being promoted to X-locks if a row or page is actually updated.

The other original Db2 isolation level, **CS** or **Cursor Stability** also differs from the SQL-92 name, ***Read Committed***. With CS, Db2 locks the row on which the cursor is positioned but keeps the lock for the minimum time necessary, that is, until the next row is fetched, or a commit point is reached. This applies to both U-locks and S-locks; U-locks are released when the application fetches the next row but are promoted to an X-lock if the row/page is updated, and the X-lock is retained until the application reaches a commit point.

Now, there are some implications with **Cursor Stability**. It ensures your transaction doesn't read a row that's been changed by another uncommitted unit of work, but it does allow other concurrent transactions to change a row that's already been read by your application before you reach a commit point. The consequence is that, if the same query is executed more than once in a CS transaction, it might get a different result set. However, provided you update rows read via a cursor by using a positioned update (WHERE CURRENT OF *<cursor_name>*) combined with a cursor defined with FOR UPDATE OF, then you can update the row with confidence because Db2 uses a U-lock to protect the row or page.

Another consequence of **Cursor Stability** is that it is less likely to lead to lock escalation, certainly in the case of read-intensive applications, which is better for concurrency. And as S and U page or row locks are typically held for a much shorter period, there are fewer and shorter lock waits, and reduced application contention with **CS** than **RR** or **RS**. This should result in improved performance – faster transactions and higher throughput.

Before we finish the discussion on **Cursor Stability**, there is one more topic to introduce. I previously said that, with **CS**, "Db2 locks the row on which the cursor is positioned but keeps the lock for the minimum time necessary". That is not completely true. Depending on the setting of the CURRENTDATA bind option, Db2 *might* lock the row on which the cursor is positioned or *might avoid taking a lock altogether*. We'll discuss this in detail later, in Lock Avoidance and the CURRENTDATA Bind Option.

The final Db2 isolation level is **Uncommitted Read** (**UR**), whose name is almost the same as that of the corresponding SQL-92 isolation level, ***Read Uncommitted***. Known also by the unflattering name Dirty Read, it allows a transaction to see rows that have been updated by another concurrent transaction which has not yet committed its updates. This can lead to inconsistent data anomalies, as it is possible that the other transaction terminates abnormally, and its updates are rolled back by Db2.

13 This is only true if the row failed stage 1 processing – if it was eliminated from the result set by stage 2 processing, then it is locked until commit and therefore cannot be updated by a concurrent transaction.

However, **Uncommitted Read** avoids the CPU and elapsed time cost of locking as it takes no row or page locks. However, it does acquire a special lock called a *mass-delete lock*. This prevents any other transaction executing an SQL DELETE with no WHERE clause, which of course empties the table out. An **Uncommitted Read** transaction will not result in lock escalation.

Uncommitted Read is only recommended where the application can tolerate inconsistent data. I mentioned above the case where one transaction reads another transaction's uncommitted updates are subsequently rolled back because the second transaction terminates abnormally. Consider another case where the second (concurrent transaction) is partway through a number of updates which form a logically consistent unit of work. The UR transaction risks seeing some rows which have been updated, and some which haven't, meaning it's view of the data is logically inconsistent. This is often referred to as *transactionally inconsistent data* – all the other isolation levels provided *transactionally consistent data*. Having said that, some applications can tolerate inconsistent data and for these UR is highly recommended because of the reduced CPU cost, better performance benefits and improved concurrency.

In concluding this discussion on transaction isolation levels, it's hopefully clear by now that for OLTP (online transaction processing) or for high concurrency read-write applications, the default design decision should be to use Cursor Stability, including any batch programs. Use the other isolation levels where necessary but be very wary of the severe impact on concurrency with isolation **RR** or **RS**, and the potential to see uncommitted updates with isolation **UR**.

Data Anomalies

Behind the discussion on locking and isolation levels are several data anomalies that applications are potentially exposed to, depending on the level of transaction isolation in effect.

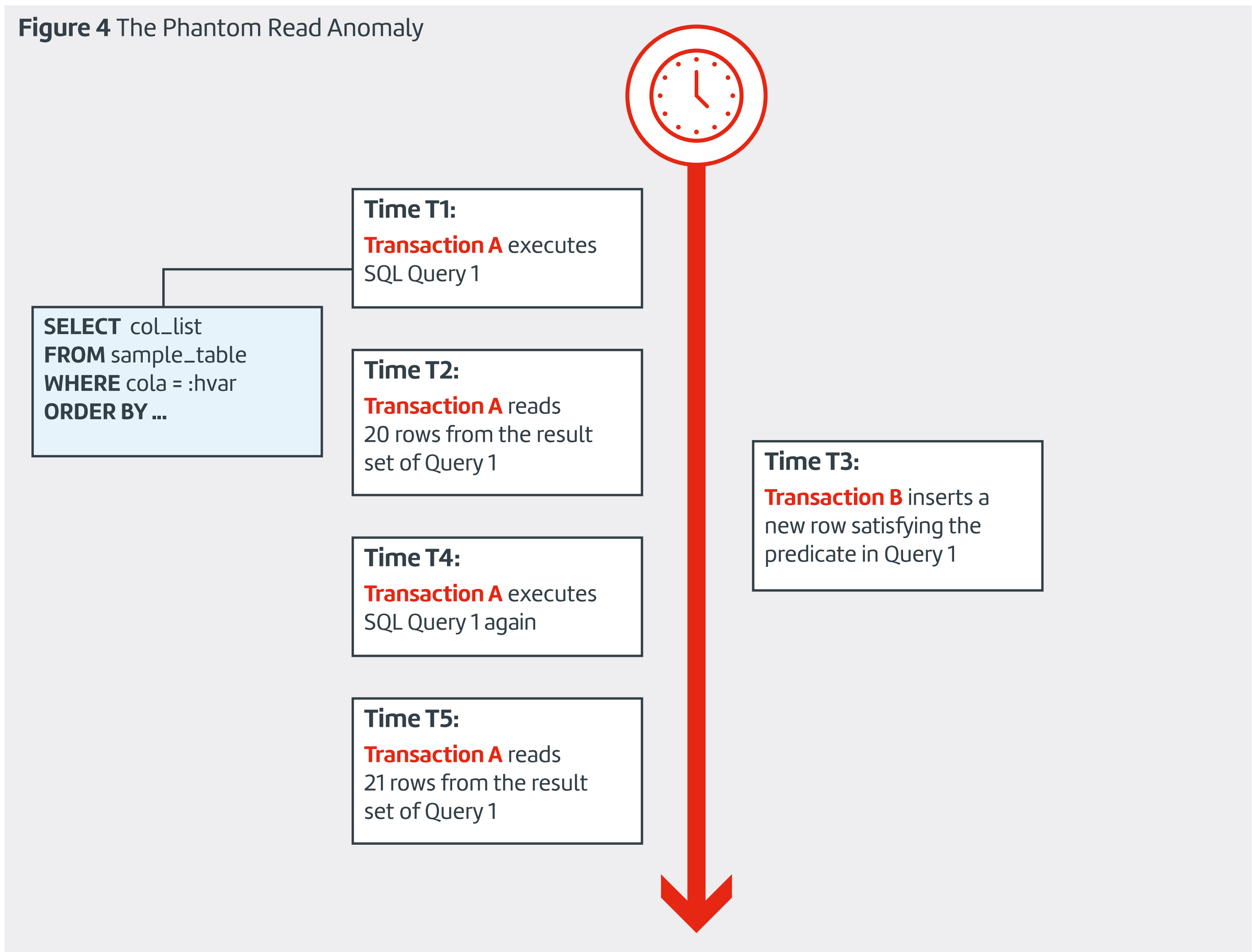
The four anomalies discussed here are not the only ones you'll find in the technical literature, but they are the most relevant for this discussion:

- The phantom row or phantom read anomaly
- The non-repeatable read anomaly
- The dirty read anomaly
- The lost update anomaly

Phantom row or phantom read anomaly

The standard definition of this is that, during the course of a transaction which accesses the same set of rows twice, new rows are added to the result set – the result set changes. This can be illustrated in **Figure 4** on page 19:

Figure 4 The Phantom Read Anomaly



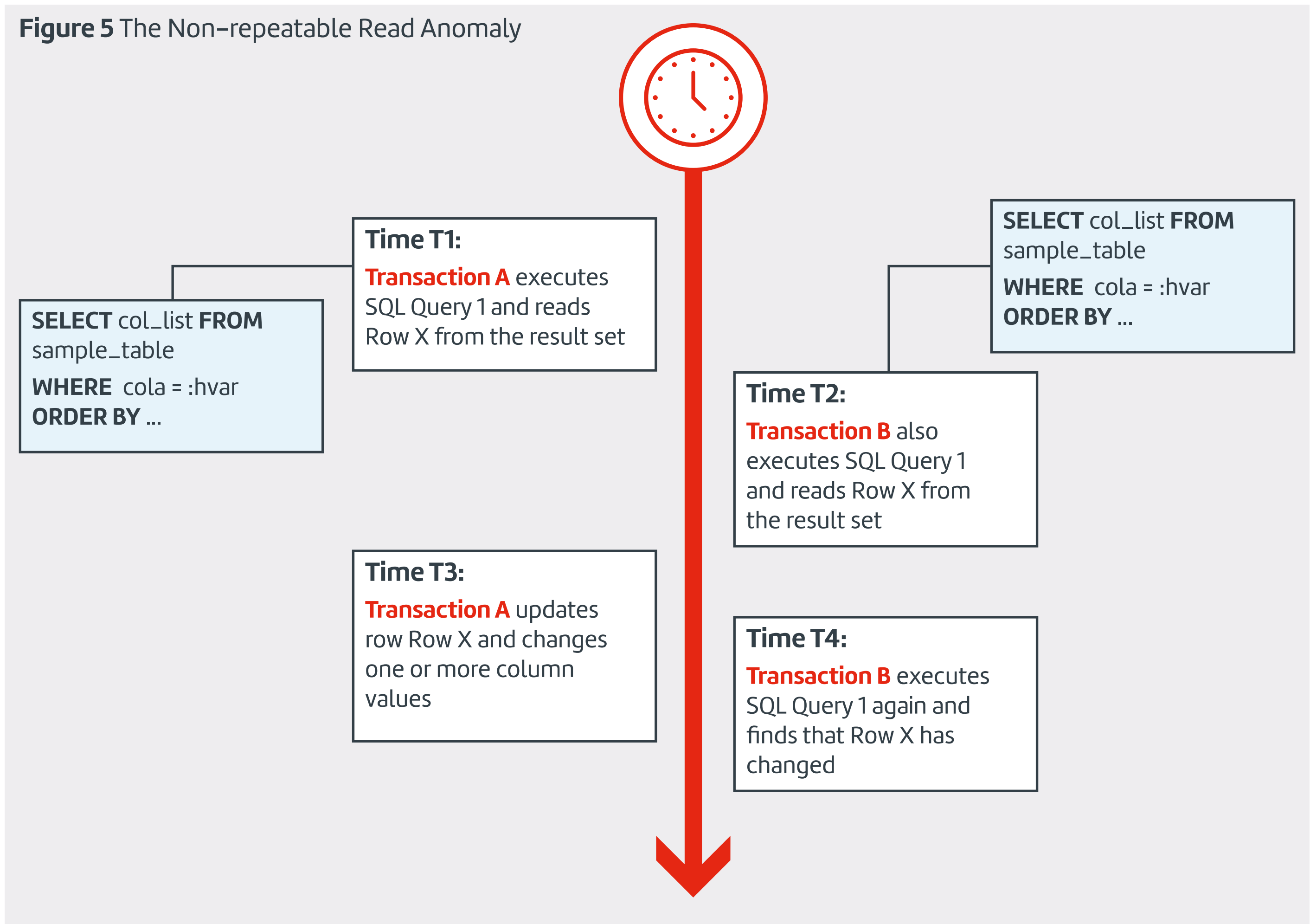
Phantom Read Example 1: A complex analytical transaction reads a large number of rows from a table several times for joining with multiple other data sets, including other tables and data sources external to the database manager. Some joins are performed programmatically by the application, others are performed by Db2 itself. Because of the large number of rows involved, the transaction cannot store them all in its working storage area. However, if additional rows appear in the result set of two or more SQL statement executions, the analysis is invalidated.

Phantom Read Example 2: A transaction T1 reads rows from a table using one or more predicates, the predicate set P1. It programmatically evaluates the result set and makes a decision whether or not to proceed with a searched update of the entire result set, again using the predicate set P1. If the result set changes with new rows appearing in the result set because of INSERT or UPDATE operations performed by other transactions between the SELECT and UPDATE operations performed by transaction T1, then although the 'phantom' row is not seen directly by transaction T1, the premise on which the decision to update the result set is invalidated, and the phantom rows are updated as well as the rows previously read.

Non-repeatable read anomaly

With the non-repeatable read anomaly, a row is accessed more than once during the course of a transaction and the row values change between accesses:

Figure 5 The Non-repeatable Read Anomaly



Again, there is a few use cases where the non-repeatable read anomaly could be encountered, but only one example is presented:

Non-repeatable Read Example: This example is very like phantom read example 1, but a small difference distinguishes these two anomalies. A complex analytical transaction reads a large number of rows from a table multiple times for joining with multiple other data sets, which could be other tables or other data sources external to the database manager. The join could be performed programmatically by the application, or it could be performed by the DBMS itself. Because of the large number of rows involved, the transaction cannot store them all in its working storage area. However, as well as the problem of additional rows appearing in the result set, if any changed rows appear in the result set or rows from the result set are deleted between executions, the analysis is potentially invalidated.

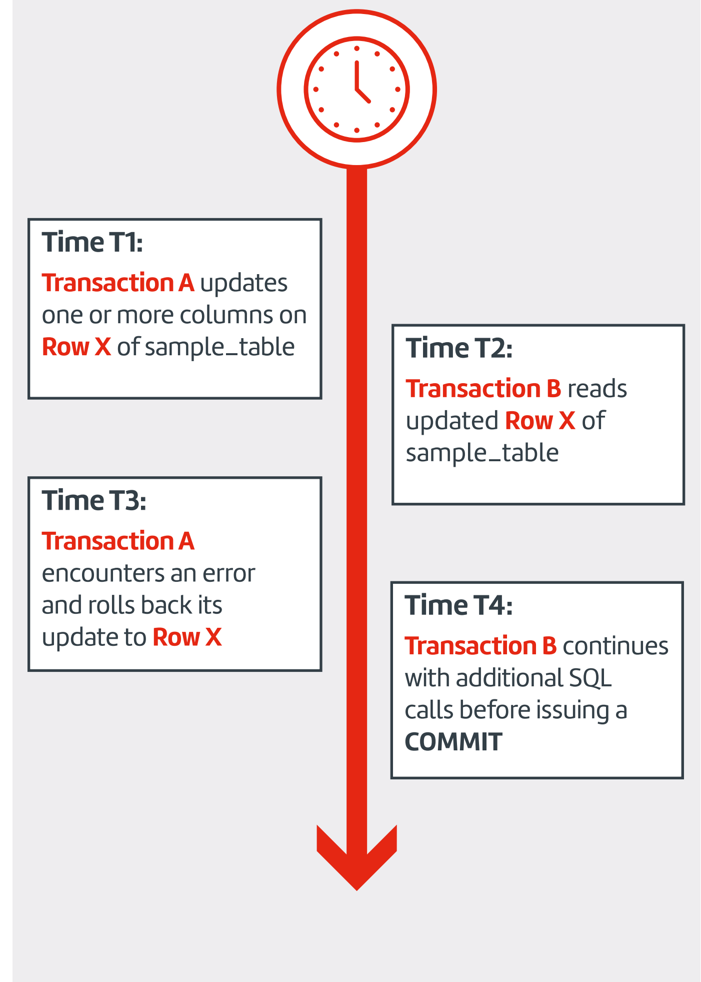
Dirty read anomaly

The dirty read anomaly is where a transaction reads data changed by another transaction, before that update transaction issued a commit. For a data integrity exposure to occur, the update transaction must be rolled back, and the transaction that read the uncommitted update must commit:

Dirty Read Anomaly Example 1: A bank account transaction reads rows without being aware if the rows it sees contain committed data or have been updated by another transaction which has not yet committed. It returns the information retrieved back to the end user. If one or more of the rows read contained uncommitted updates from another transaction which is subsequently rolled back, then incorrect information is reported back to the end user.

Dirty Read Anomaly Example 2: A bank account update transaction reads rows without being aware whether the rows it sees contain committed data or have been updated by another transaction which has not yet committed. It uses the retrieved data to update other banking information, either in the same table or one or more other tables. If any of the data used to drive updates was read from a row containing uncommitted updates from another transaction which is subsequently rolled back, undoing the updates, then data corruption is possible if not probable.

Figure 6 The Dirty Read Anomaly



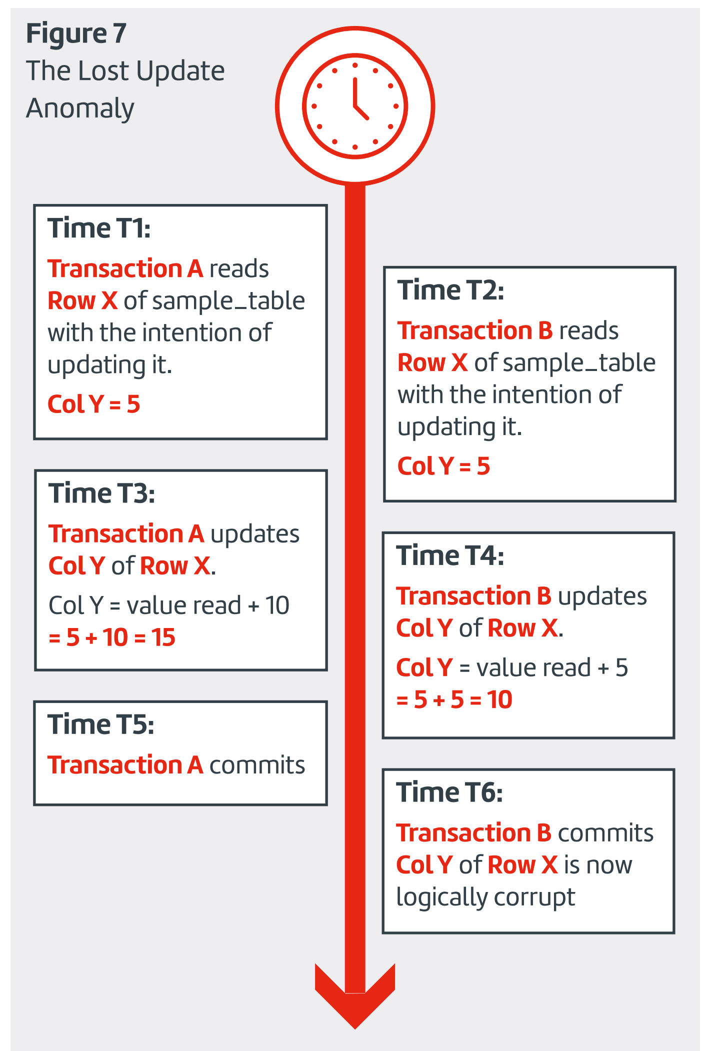
Lost update anomaly

The lost update anomaly is perhaps the most obvious anomaly leading to data corruption and occurs when two transactions both read and update the same row, but the second transaction overwrites the first transaction's update:

You'll recognise **Figure 7** as the **Figure 1** Logical Data Corruption diagram from the introductory section on data integrity, performance, and locking. It illustrates that the lost update anomaly occurs when two concurrent transactions access the same data without the accessed data being protected by locks. In this case, when transaction B commits, Col Y of Row X should have a value of 20, not 10 (the original value of 5 incremented by 10 and then incremented by a further 5, that is, 5 + 10 = 15, 15 + 5 = 20). Unlike the other data anomalies, where potentially inconsistent data is returned to the application, the lost update causes logical data corruption in the database.[14]

Later in this paper we'll see how the lost update anomaly can occur even when the updated data is protected by locks if the wrong programming techniques are used.

Figure 7
The Lost Update Anomaly



Isolation Levels and Data Anomalies

The following **table** shows which transaction isolation levels are susceptible, in theory, to which data anomalies.

Note that the table specifies 'might occur' and not 'does occur' – data anomalies are not guaranteed to occur for these isolation levels but can occur in the right (or wrong) combination of circumstances.

ANSI/ISO Isolation	Db2 Isolation	Phantom Row	Non-repeatable Read	Dirty Read	Lost Update
Serializable	RR	Does not occur	Does not occur	Does not occur	Does not occur
Repeatable Read	RS	Might occur	Does not occur	Does not occur	Does not occur
Read Committed	CS	Might occur	Might occur	Does not occur	Does not occur
Read Uncommitted	UR	Might occur	Might occur	Might occur	Does not occur

14 Bear in mind that returning inconsistent data to an application could result in logical data corruption if the application makes subsequent updates to the database based on that inconsistent data.

Note that by *design*, the RS, CS and UR isolation levels accommodate the possibility of encountering one or more of the anomalies. This means that when you code your application, you must be aware of the potential exposure to the associated data anomalies. If you never need to access a row more than once, then you won't need to use isolation RR or RS. Be wary of that word 'access': it includes update access as well as read access. More of that later on.

Also note that for CS and UR, the table indicates that the lost update does not occur. This is not strictly true: with Db2 for z/OS, if you use the wrong coding techniques, then the lost update anomaly *could occur*. For that reason, let's look at data currency options and, first, some data currency issues associated with cursors.

Cursors and Data Currency

As we've seen, Db2 manages the various levels of transaction isolation, principally by using different lock durations for RR and RS versus CS.

However, CS introduces a series of data currency considerations around *when* a lock is taken and if a lock is taken, related to the type of cursor and the CURRENTDATA BIND option. It's vital to emphasise that the following discussion concerns isolation CS only – none of the issues can be encountered with isolation RS or RR.[15]

In this section we'll discuss various cursor types, when locks are acquired for the read-only cursor, and the implications for data currency of the mechanism know as *lock avoidance*. We'll then go into some detail about the data integrity implications of combining read-only cursors with searched updates, a common practice in some environments and with some programming languages such as Java. Then, we move onto the perils of access-dependent cursors and why you should avoid relying on them.

For-update, Read-only and Ambiguous Cursors

The cursor types relevant to this discussion do not include the set of cursors known as *scrollable cursors*, but rather non-scrollable read-only cursors, ambiguous cursors, and for-update cursors – the types of cursor most commonly used. We'll deal with for-update cursors first.

A for-update cursor is essentially a cursor that allows data to be updated via a positioned update statement. A positioned update statement is of the type UPDATE … WHERE CURRENT OF *<cursor_name>* (or DELETE … WHERE CURRENT OF). Typically, a for-update cursor declared using static SQL requires a FOR UPDATE clause when it is declared.[16] The declaration of a for-update cursor must not include an ORDER BY clause nor any function or clause that implies sorting or aggregation such as MAX or GROUP BY. With dynamic SQL, to use a positioned update, the cursor declaration must always include a FOR UPDATE clause. Locking with update cursors running with CS isolation is always the same: when a row is read, a U-lock is taken; if the row is actually updated, an X-lock is taken and is held until commit time; if the row is not updated then the U-lock is released when the next row is read or the cursor is closed, whichever happens first.

Next, we'll deal with read-only cursors, but as they are the most important cursors in terms of the data currency and integrity discussion, we'll return to them again later. If the result set is read-only – that is, it's not eligible for update – then the cursor is read-only. For example: if the query contains any of ORDER BY, GROUP BY, DISTINCT, UNION, INTERSECT or EXCEPT; if the FROM clause represents more than one table or view, or a nested table expression; if the query uses FOR FETCH ONLY; or if the result set is materialized in a work file.

15 See, however, the discussion on RR/RS and non-materialised result sets later in this paper.

16 For a cursor declared in a static SQL statement, the "FOR UPDATE" clause is not required if the STDSQL(YES) or NOFOR options are in effect at the program preparation stage. The default settings for the STQSQL system parameter is NO, but this can be overridden at program preparation time.

We'll come back to read-only cursors once we've discussed the thorny issue of *ambiguous cursors*. Simply put, a cursor is ambiguous if Db2 can't tell whether it's used for update or for read-only purposes. The topic of ambiguous cursors is more complicated than defining cursors without the FOR UPDATE or FOR FETCH ONLY attribute, or implicit read-only cursors. We won't go into all the details of ambiguous cursors here, other than to highlight why you should avoid their use wherever possible. Firstly, in some circumstances, Db2 stops you updating rows via an ambiguous cursor – see SQL Code -510 for more details. Secondly, whereas in some circumstances use of an ambiguous cursor can result in Db2 lock avoidance techniques being used (discussed later), in other circumstances it can inhibit the use of those lock avoidance techniques. Because of the difficult nature of ambiguous cursors, you should always code FOR FETCH ONLY or FOR UPDATE on a cursor, to avoid ambiguous cursors and to document the use of the cursor in the program code. Theoretically, you could rely on ORDER BY to implicitly define a cursor as read only, but if you were to remove the ORDER BY when maintaining the application program, then the cursor might move from read-only to ambiguous.

Acquiring and releasing locks for read-only cursors

Having outlined the difference between for-update, ambiguous and read-only cursors, it's time to return to the topic of *when* locks are acquired and released with read-only cursors. If the result set is not materialised in a work file or there is no in-memory sort, then the S-locks on the pages or rows are acquired at fetch time. They are released when the application fetches a row on another page for LOCKSIZE PAGE or fetches the next row for LOCKSIZE ROW. So far, so good. On the other hand, if the rows are sorted or the result set is materialised in a work file, then any necessary locks are acquired at open cursor time and will have all been released before the first fetch. This means that when a transaction with this kind of access path gets to read each row from the materialised result set, it holds no lock on that row which could already have been updated by another transaction.

In the case of a read only isolation CS cursor, once a row has been read and the cursor is positioned on another page or row (depending on the lock size), there is no lock protecting that page or row and preventing other concurrent transactions from updating that page or row, regardless of when a share lock on the page or row was acquired.

The time at which locks are taken also affects, of course, when any lock waits are incurred, either at cursor open time or row fetch time. And following from that, if there is lock contention within the application, the time when lock waits are incurred determines when any timeouts are experienced and, depending on the mix of SQL statements used, can affect the probability of deadlocks.

To summarise, the exact currency of each row accessed via a read-only cursor is dependent on the access path chosen by Db2. The row seen by the application could be the state of the row as of open cursor time or the state of the row as of fetch time. The application cannot tell if it is seeing the latest version of the row, unless it is access-path dependent and knows with certainty that an S-lock was acquired at fetch time.

Lock Avoidance and the CURRENTDATA Bind Option

For now, we'll move on to discuss the 'if' of locking, in conjunction with some more data currency issues. As well as the options we've already discussed, there is another BIND option, CURRENTDATA. The default setting for this option has changed over Db2 releases more than once, but since at least Db2 Version 8 it has defaulted to NO.

Let's start off with CURRENDATA(YES), which means that the data cannot change while the cursor is positioned on it. If a cursor declared by Transaction A is positioned on a base table row or index (for index-only access), then the data returned by the cursor is guaranteed to be current. Be aware that if the cursor moves onto the next row, then the lock on the previous row is released and a lock on the next row is obtained. This means that another transaction could acquire an exclusive lock on the row and update it, even while Transaction A continues to read other rows via the cursor.

However, if the cursor is positioned on data in a work file (including rows sorted in memory), the data returned by the cursor only matches the contents of the work file, not the row in the base table. Locks on the base table rows are acquired at cursor open time and will have been released by the time the transaction issues the first fetch. Therefore, by the time the transaction reads a row in the result set, the row in the base table might be unchanged or could have been changed by another transaction, because the transaction won't hold any locks on the base table rows – guaranteed. A side of effect of CURRENDATA(YES) is that it disables lock avoidance, described below.

The default BIND option of CURRENTDATA(NO) means that the data can change while the cursor is positioned on it if lock avoidance is in effect for the reader, and the update transaction can then acquire a U-lock or an X-lock on the data. The benefit of lack avoidance is that it improves concurrency and reduces CPU consumption. CURRENDATA(NO) enables but does not guarantee lock avoidance. That is, Db2 can avoid locking the data if it can determine that the row or page contains committed data.[17] If taking a lock can be avoided, then the data is returned to the application without a lock being taken, and the data can be changed even while the cursor is positioned on the row. Even if you're not planning to update the row, this might be an issue, but you can use the techniques outlined later in this paper to allow you to use CURRENTDATA(NO) combined with searched updates safely. If Db2 is unable to determine that the row or page contains committed data, it acquires an S-lock to make sure that no other transaction has updated the row without committing the update. If there is an uncommitted update, then the transaction requesting the S-lock is suspended until the other transaction's X-lock is released at COMMIT time.

Experience shows that effective lock avoidance brings great concurrency benefits, especially in a data sharing environment. It also plays a role in reducing CPU consumption and for update-intensive batch can help control MLC costs. It's important to emphasise that for performance reasons as well as concurrency, CURRENTDATA(NO) is strongly recommended in a data sharing environment, provided the application is coded correctly and does not depend on share locks always being obtained.

There is an important point to make here about singleton select statements – non-cursor SELECT statements that return a single row. There is a clue as to the locking implications for singleton selects in the name of the isolation level: *cursor* stability. With a singleton select, regardless of the CURRENTDATA setting, by the time the row is returned to the application, there will not be a lock on the page or row. If a lock is required, it is obtained when Db2 determines the row qualifies, and then released almost immediately, and before the row is returned to the application. This is because the transaction is not positioned on the row, as position is only possible when the row is accessed via a cursor.

Combining Read-only Cursors with Searched Updates

Before expanding the discussion about read-only cursors running with a transaction isolation level of cursor stability (CS) to include the fairly common practice of combining read-only cursors with searched updates, it's important to point out that if you intend to read rows via a cursor with the intention of updating them (or deciding whether or not to update them), then where possible you should declare the cursor with the FOR UPDATE clause and use positioned updates: UPDATE … WHERE CURRENT OF … . This is because when you read a row via a for-update cursor, Db2 acquires a U-lock[18] which is held for as long as the cursor is positioned on the row, and only released when the cursor is moved off the row without updating it. The U-lock allows simultaneous concurrent transactions to acquire S-locks but not U or X-locks. If the row is updated, the U-lock is promoted to an X-lock which is retained until the application reaches a commit point. This mechanism prevents any concurrent transactions from updating the row (or even evaluating the row for update) when the cursor is positioned on it.

17 Lock avoidance is covered in some detail in the Redbook "Db2 9 for z/OS: Resource Serialization and Concurrency Control", SG24-4725. Although this book is over 10 years old, it's still a useful source.

18 With the exception of declaring the cursor using the WITH RS/RR and USE AND KEEP EXCLUSIVE LOCKS clauses, in which case an X-lock is acquired at fetch time.

However, using an update cursor is not always possible, so some applications have to combine a read-only cursor with a searched update (UPDATE ... WHERE cola = :hostvar ...). For example:

- ORDER BY is used to ensure that the rows are read by the application in a required order.
- The application coding framework's default behaviour is to generate read-only cursor declarations combined with searched update statements.
- The application requires maximum concurrency and minimum locking overhead.

Let's deal with the more obvious integrity problem of using this technique with isolation CS and CURRENTDATA(NO). Because the application cannot know whether or not there is an S-lock, it cannot know whether or not another application is in the process of updating the data it is reading. The data integrity exposure is a sequence something like the following, where two transactions are trying to update the same row:

Time T0 : Transaction A reads Row Z from Table T via a read-only cursor without acquiring a lock

Time T1 : Transaction B reads Row Z from Table T via a read-only cursor without acquiring a lock

Time T2 : Transaction A updates Row Z (Z0 > Z1A) and acquires an X-lock

Time T3 : Transaction B tries to update Row Z and is suspended because of an incompatible lock held by Transaction A

Time T4 : Transaction A terminates, issuing a commit, and the X-lock is released

Time T5 : Transaction B is resumed, acquires the X-lock on Row Z and updates Row Z (Z0 > Z1B) and overwrites Transaction A's update.

Time T6 : Transaction B terminates, issuing a commit, and the X-lock is released

The end result is an instance of the lost update anomaly, and logical data corruption.

Switching to CURRENTDATA(YES) is not guaranteed to solve the problem of the exposure to the lost update anomaly. If the result set is sorted or materialised in the work-file database, then by the time the application reads the row, there will be no lock on the row, and the application is still exposed to the lost update anomaly. This applies regardless of the CURRENTDATA setting. Unless you use the correct coding techniques, then the only alternative is the unattractive one (from an OLTP performance/concurrency point of view) of using isolation RR or RS.

But let's assume that the result set is not sorted or materialised in a work-file or in memory, and that S-locks are obtained when the row is read. Taking the example above, where two transactions are trying to update the same row, having read it via a read-only cursor. When Transaction A, which already has an S-lock on Row Z, now attempts to promote the lock to an X-lock on that row, it is suspended by Db2 because Transaction B holds an incompatible lock on Row Z – an S-lock. Nevertheless, Transaction A retains its S-lock on Row Z, because the cursor is still positioned on that row. When Transaction B also attempts to promote its S-lock to an X-lock on Row Z, then it too is suspended, because transaction A holds an incompatible S-lock on it. Both transactions are now suspended and without Db2 deadlock detection would wait for an indefinite time for the lock promotion request to succeed. This is an example of how you can get a deadlock on a single row. On the other hand, although the application is vulnerable to deadlocks, data integrity is guaranteed provided that the access path doesn't change so that the result set is materialised in a work-file.

Similar considerations apply to the singleton SELECT. As discussed in the previous article, because there is no S-lock on the row by the time it is returned to the application, using a singleton SELECT followed by a searched update with cursor stability isolation is vulnerable to the lost update anomaly, regardless of the CURRENTDATA setting.

Access-Path Dependent Cursors

To continue the discussion about cursors, it might be tempting to rely on combining ISOLATION(CS) with CURRENTDATA(YES) and an access path that avoids result set materialisation for a read-only cursor (typically with an ORDER BY) to ensure that there is an S-lock on the row when the transaction read it, so although the transaction might be vulnerable to lock waits, timeouts or deadlocks, data integrity is nevertheless guaranteed.

For example, consider a cursor-based SELECT in an ISOLATION(CS), CURRENTDATA(YES) package, where the access path uses index which satisfies the predicates, provides implicit ordering for an ORDER BY clause, and ensures that there is an S-lock on the page or row. The searched UPDATE is now safe because the S-lock prevents any other transaction from taking an X-lock on the page or row.

However, if the access path changes for any of many different possible reasons,[19] then the assumption that the searched update is safe is now flawed. There is no guarantee that an ORDER BY will be satisfied by an index, but there is a risk that the ORDER BY causes a sort and result set materialisation either in memory or in a work-file. If this happens, then once again, the application is exposed to the lost update anomaly. For this reason, applications which combine read-only cursors with searched updates of rows in the result set should not rely on access path dependent cursors to guarantee data integrity.

This brings us back to the discussion about FOR UPDATE cursors and ordering of the result set. If the application relies on an index to impose ordering on a FOR UPDATE cursor because the columns in the index match the columns in the list of predicates, then that ordering is vulnerable to unwanted or unexpected access changes. Although data integrity is guaranteed because a FOR UPDATE cursor is used to exploit Db2'S-locking semantics, the application logic can break down if it is dependent on the order in which the rows are returned. This can happen, for example, if an index with different ordering is selected by the Optimizer at BIND or PREPARE time, or access to the rows reverts to a tablespace scan.

Examples of this include batch programs with intermediate commits and restart logic to resume processing after a failure, or batch programs which reopen their cursors after a commit and reposition on the data. If the application relies on *implied ordering*, then data corruption can occur if the *implied ordering* is not honoured because of an access path change. The form of corruption in this example is where rows are updated multiple times, or not updated at all.

Locking With Searched Update and Delete

We've discussed the locking mechanism in effect for cursors, but it's also important to understand what locks Db2 acquires for searched updates and deletes. The exact mechanism depends on a system parameter (ZPARM), XLKUPDLT (X-lock for searched update and delete), which has three possible settings: NO, YES and TARGET. Starting with the default setting, the effects are as follows:

XLKUPDLT=NO: Db2 uses an S-lock or U-lock when searching for rows that qualify. For rows or pages that qualify, the lock is promoted to an X-lock before performing the update or delete. For non-qualifying ISOLATION(CS) rows or pages, the lock is released. An S-lock is used for ISOLATION(RR|RS) transactions, when ZPARM RRULOCK is set to NO. See below for more information on this ZPARM.

XLKUPDLT=YES: Db2 immediately acquires an X-lock when searching for rows that qualify. For ISOLATION(CS), the lock is released if the rows or pages do not qualify, but for ISOLATION(RR|RS) the X-lock is retained until the next commit. The X-lock is also used for rows or pages for additional tables that are referenced in the query but not updated, for example tables referenced in the WHERE clause. This setting typically increases the potential for contention, but in some cases can reduce the CPU overhead without impacting concurrency. For example, this might be useful in a data sharing environment where a unique index is used to identify qualifying rows.

19 For example, a REBIND following a RUNSTATS, Db2 maintenance, database schema changes, the DBA dropping the index, the index being in rebuild pending etc.

XLKUPDLT=TARGET: This is similar to XLKUPDLT=YES, except that an S-lock or U-lock is acquired on rows or pages for any additional tables referenced in the query, for example, tables referenced in the WHERE clause.

RRULOCK: as indicated above, the acquisition of S-locks instead of U-locks for a searched update or delete is controlled by another ZPARM, RRULOCK. This specifies what lock is taken for transactions running with ISOLATION(RR|RS) and for cursor-driven positioned updates (SELECT with FOR UPDATE OF) as well as for searched UPDATE and DELETE. For RRULOCK=YES, Db2 acquires a U-lock. If the row is not updated, the U-lock is demoted to an S-lock on the next fetch. Otherwise, the U-lock is promoted to an X-lock. For RRULOCK=NO, Db2 acquires an S-lock, which will be promoted to an X-lock if the row is updated. If your RR or RS applications make frequent updates, a U-lock reduces the potential for deadlocks. For read-only applications or a mix of read and write applications, S-locks generally provide more concurrency and better performance (reduced CPU time). Whatever you code for RRULOCK is over-ruled by XLKUPDLT=YES or TARGET.

Bear in mind that these ZPARMs are system wide and in general don't affect the way you code your applications except in the case where programming standards mandate how searched updates and deletes are coded.

Update Applications and Data Integrity

It's time, at last, to move on to recommendations for coding update application for data integrity – even with read-only cursors – to deliver high performance combined with guaranteed data integrity.

This section outlines the coding techniques required to protect against data anomalies, most specifically the lost update anomaly. One of the most important factors determining the available safe techniques is the transaction isolation level in effect at run time.

Let's start with a restatement of why this is important. In Db2 for z/OS, the recommended programming technique for reading rows via a cursor and then updating some or all of those rows is to specify the FOR UPDATE clause on the cursor declaration and use positioned updates – UPDATE WHERE CURRENT OF. This has the advantage that, when you read a row, Db2 takes a U-lock on the row or page. This allows concurrent readers with an S-lock, but any concurrent transactions requesting U or X-locks will have to wait until the U-lock is released. When the transaction issues UPDATE WHERE CURRENT OF <cursor-name>, Db2 attempts to promote the U-lock to an X-lock. This ensures that no other transaction can have updated the row between the SELECT, which protects the row with a U-lock, and the UPDATE.

However, it's not always possible to use FOR UPDATE cursors. A lot of Java tooling, for example, generates read-only cursors with searched updates (UPDATE ... WHERE cola = :hostvar). For all applications, regardless of the tooling, using FOR UPDATE cursors is either not possible or safe if ordering of the result set is required. Why not safe? In cases like this, even if an implicitly-ordering index is available, because it's unsafe for the application to rely on that index in case the access path changes, it's necessary for ORDER BY to be coded on the SELECT statement referenced by the cursor.

This introduces a topic that so far has only been hinted at, *optimistic locking*. Optimistic locking is based on the assumption that concurrent transactions can run successfully without affecting each other, and that each individual transaction doesn't need to lock the rows being accessed. Db2 transactions running with page/row level locking, cursor stability isolation and CURRENTDATA(NO) can exploit one of the variants of optimistic locking.[20] For read-only transactions, the question is, can they tolerate accessing data without locking it. For update transactions, an additional consideration is that the row they are updating might have been changed since they read it.

20 Some might argue that this is not pure optimistic locking as Db2 might take a lock on a page or row. Whether or not Db2 obtains a lock depends on the effectiveness of the lock avoidance mechanism, that is, whether or not Db2 can determine that the row/page contains committed data. And bear in mind that Db2 will in any event take an intent lock on the tablespace/partition.

It is also the case that any transaction using a read-only cursor combined with searched updates, even with CURRENTDATA(YES), must be coded as if it were using optimistic locking as it can't know with certainty when a lock was taken on the row they are reading: at fetch time or at open cursor time, without using an access path-dependent cursor, which is (as already described) a high-risk technique.

For this kind of application – combining read-only cursors with searched updates – optimistic locking should be used. This technique typically involves the use of ISOLATION(CS) and CURRENTDATA(NO), although as we've seen it's also preferable for CURRENTDATA(YES). It combines read-only cursors with searched updates, even where a FOR UPDATE cursor is possible. It's based on the optimistic premise that a lock is not required for the fetch but is only needed if the transaction tries to update the row, saving the CPU cost of acquiring a lock at fetch time. For some applications, the driving requirement behind optimistic locking is to minimise the CPU cost of locking. The smaller the ratio of updated rows, the more favourable this technique is, as each update means the row must be accessed twice: once for FETCH and once for UPDATE; the latter requires its own access path.[21]

Bear in mind that even for a transaction using optimistic locking – that is, running under an ISOLATION(CS) CURRENTDATA(NO) package – read locks will still be required if another concurrent uncommitted transaction has updated but not committed the row being read. Provided that the result set is not materialised at open cursor time, then an optimistic locking transaction trying to read the row via a read-only cursor will enter a lock wait at fetch time, when Db2 will delay obtaining an S-lock on its behalf until the concurrent update transaction has committed. For a materialised result set, any lock waits occur at open cursor time.

Some applications adopt a different strategy by declaring the cursor using the WITH UR clause. The technique for UPDATE (discussed below) remains the same, but the difference between the two approaches is that the UR reader might see uncommitted updates to the row after reading the uncommitted update without acquiring a lock, and only trying to acquire an X-lock if and when it issues an UPDATE.

Which of these two methods you choose for reading the data will depend on a number of factors, including: the level of contention within the application as a whole (the intensity with which the transactions contend for the same data); the percentage of rows read that are updated; the toleration for reading transactionally inconsistent data; the importance of minimising CPU consumption; and other considerations.

To return to isolation CS transactions with CURRENTDATA NO, if Db2 can avoid taking a lock for Transaction A's read-only cursor, it will – this allows other concurrent transactions to subsequently take U or X-locks, potentially even while Transaction A's cursor is still positioned on the row. There will definitely be no lock on the row at fetch time if the result set was materialised e.g., in a work-file. This opens up the possibility of the lost update anomaly, which can occur if Transaction A updates the row without verifying that the row hasn't changed since it read it. Very similar considerations apply not only with isolation UR but also with singleton SELECTs (SELECT without a cursor). In both cases, the transaction reading the row will not have a lock on the row when the application reads it.

The techniques for guaranteeing data integrity when using read-only cursors with searched updates are quite simple in principle but the performance profile can vary considerably. Therefore, you should factor your performance requirements into the criteria for selecting one of the techniques. Any application using these techniques must handle the "row not found" condition for updates (SQL Code 100, SQLSTATE 01568), as we'll see.

21 Why does UPDATE require its own access path? Because at BIND time (for static SQL statements) or PREPARE time (for dynamic SQL), Db2 can't know that the row being updated is the same one as the row on which the cursor is positioned. Indeed, it can't even know that there is a cursor, because BIND/PREPARE does not take into account any other statements which the program may (or may not) have executed. Just to reinforce the point, BIND does not know the order in which SQL statements will be executed; and PREPARE can take place at any time prior to the statement being executed.

The first technique is to select all columns from the table, and when updating a row specify all retrieved column values as predicates on the searched UPDATE statement. If any value has changed since the row was read, the "row not found" condition will be returned when the transaction attempts to perform the update. At least one software vendor application uses this method. The technique is effective but could have a significant CPU cost, dependent on the table design. There are several factors to take into account when considering whether to use this technique, including:

- Each column returned to the application and each predicate evaluated on the UPDATE increases the CPU cost. The more columns in the table, the less attractive this solution from a performance/CPU cost perspective. This technique suits tables with a smaller number of columns better than tables with a larger number.
- A high ratio of rows updated / rows read can make this technique more costly in CPU terms if the number of searched UPDATE predicates is large.
- All programs using this technique must be coded accordingly. If any updateable columns are added to the table, the programs must be modified.

One temptation might be to select all columns that are updated and to include only those columns in the predicate list for the update along with the predicates needed to uniquely identify the row. However, that is potentially dangerous – if the list of updated columns changes, then existing update applications are potentially susceptible to the lost update anomaly because the list of predicates excludes rows that might have been updated. As well as applications that issue update statements, applications that use read-only cursors and searched deletes must be coded to deal with the 'not found' condition. If you're planning to use this technique, it's better to err on the side of caution than risk compromising data integrity by including all columns in the list of predicates.

There is one further drawback to this approach. If new columns are added to the table, then all application programs that update the table must be updated to include the extra columns in the select and in the list of predicates in the WHERE clause of the UPDATE statement.

The second technique is to add one or more columns that indicate whether the row has been updated. A typical technique is to add a timestamp column which is updated with the current timestamp by every UPDATE or INSERT statement. Other column types are possible, but in this discussion let's assume that the added column is a timestamp column. All searched UPDATE statements must include the timestamp column in the list of predicates, which means that if any value has changed since the row was read, the "row not found" condition will be returned when the transaction attempts to perform the update. Factors to consider for this technique include:

- All programs updating rows in the affected tables must update the timestamp column regardless of whether they use searched updates or positioned updates.
- All programs reading rows via a read-only cursor with the possible intent of updating them must include the timestamp column in the list of columns returned and must include it in the predicate list.

The third technique is to use a feature introduced in Db2 9 for z/OS, the ROW CHANGE TIMESTAMP column. Db2 automatically maintains the contents of ROW CHANGE TIMESTAMP columns so there is no need for the application to update them. To be more precise, Db2 generates a value for the column for each row as it is inserted, and whenever it is updated. Factors to consider for this technique include:

- Programs don't need to maintain the ROW CHANGE TIMESTAMP column.
- However, all programs reading rows via a read-only cursor with the possible intent of updating them must retrieve the ROWCHANGE TIMESTAMP COLUMN value as well as the list of columns they are interested in and must include it in the predicate list when performing the UPDATE.

For example, if a column is added to a table defined with the column RCT_COL NOT NULL GENERATED ALWAYS FOR EACH ROW ON UPDATE AS ROW CHANGE TIMESTAMP, the list of columns selected should include RCT_COL:

```
SELECT Col1, Col2, ... , RCT_COL FROM Table1 WHERE Col1 = :hvar1 ...
AND ... ORDER BY ...
```

The UPDATE statement must then include RCT_COL in the list of predicates:

```
UPDATE Table1 SET Col2 = :hvar2, ... , WHERE Col1 = :hvar1 AND ... AND
RCT_COL = :rcthvar
```

Remember, you cannot update the ROW CHANGE TIMESTAMP column yourself: that column is maintained by Db2.

An alternative to retrieving the ROW CHANGE TIMESTAMP column itself is to use the ROW CHANGE TIMESTAMP or ROW CHANGE TOKEN function instead.

The syntax for dealing with row change timestamp columns looks a little strange at first, but all you need to remember that it is, in fact, just an expression in the column list. I've simplified the statements below, starting off with the SELECT, where the row change timestamp value is selected into host variable :rcthvar . In this example, only one column, Col2, is shown being updated

```
SELECT Col1, Col2, ... , ROW CHANGE TIMESTAMP FOR Table1 FROM Table1
WHERE Col1 = :hvar1 ... AND ... ORDER BY ...
```

And the UPDATE statement is very similar:

```
UPDATE Table1 SET Col2 = :hvar2, ... , WHERE Col1 = :hvar1 AND ... AND
ROW CHANGE TIMESTAMP FOR Table1 = :rcthvar
```

You might be tempted to improve the performance of the UPDATE by selecting the value of a ROWID column or using the ROWID or RID functions. For example, having retrieved the ROWID for a row into a host variable in the SELECT, you can use direct row access for the UPDATE:

```
SELECT Col1, Col2, ... , ROWID_Col, ROW CHANGE TIMESTAMP FOR Table1
FROM Table1 WHERE Col1 = :hvar1 ... AND ... ORDER BY ...

UPDATE Table1 SET Col2 = :hvar2, ... , WHERE ROWID_Col =
ROWID(:rowidhvar1) AND ROW CHANGE TIMESTAMP FOR Table1 = :rcthvar
```

However, this can be a risky strategy as both the RID and ROWID for a given row can change if a REORG SHRLEVEL CHANGE utility runs concurrently with the application, more particularly if the application program uses intermediate commits, and FETCH and UPDATE occur in different commit scopes.

As is usual with Db2, the full story is more complicated than first appears. The reason you need to be careful is that the values for these functions can be changed by a REORG, and you should count on this being the case. However, if you select a ROWID or RID value for a given row and then re-use it for direct row access in the same commit scope, then the use of these functions is safe. This is because the earliest time your application will see the data in its reorganised state is after the commit, that is, after REORG has completed the SWITCH phase.

There is more to row change timestamp columns than can be covered here. There is some good information in the archived IBM Redbook, Db2 9 for z/OS Technical Overview. If you don't already have a copy, it's strongly recommended that you download one. It's also strongly recommended that when you use row change timestamp columns, you specify them as NOT NULL GENERATED ALWAYS FOR EACH ROW ON UPDATE AS ROW CHANGE TIMESTAMP, as NOT NULL GENERATED BY DEFAULT FOR EACH ROW ON UPDATE AS ROW CHANGE TIMESTAMP allows the application to provide a value for the row change timestamp column.

The most important point with optimistic locking strategies is that they all require a programming standard to be defined, documented, and enforced. That standard must include one – and only one – of the techniques for combining updates with optimistic locking and a strategy for handling 'row not found' on the UPDATE. One badly written application program that doesn't follow the standard can undo all the good work of the applications that do comply to the standard, and to make matters worse these kinds of logical data inconsistencies can be very difficult to detect and correct. In some cases, it can take weeks or months for the data inconsistencies to become apparent. Recovering from logical data corruption is too big a topic to be covered in this paper.

A Sidenote on Optimistic Concurrency Control

You might see the terms "optimistic locking" and "optimistic concurrency control" used interchangeably. This is inaccurate, as optimistic concurrency control is a feature introduced in Db2 9 for z/OS that depends on two things: firstly, the existence of a row change timestamp on the table being queried; secondly, the use of an updateable static scrollable cursor. If both conditions are satisfied, Db2'S-locking behaviour with ISOLATION(CS) changes. If a lock is obtained on a row at fetch time, it is obtained immediately before the fetch and then released immediately. A new lock is then taken on a row only for a positioned update or delete, and Db2 re-evaluates the predicate to ensure that the row has not changed since the fetch operation. The use of scrollable cursors, static or dynamic, is not considered in this paper.

If you need more information, you can read about optimistic concurrency control in the archived IBM Redbook DB2 9 for z/OS Technical Overview and in the Db2 documentation.

Additional Considerations

There are several additional topics that arise out of the previous discussions, and while not central to the key messages of this paper, they still address some useful issues.

Row level locking

The first item under discussion is row-level locking. As mentioned previously, the default lock size should be page-level, with row-level locking only being used where justified. This is to reduce the CPU and elapsed time overhead typically required with row-level locking, but more importantly to avoid the data sharing overhead involved. The Db2 for z/OS documentation has further information about locking in a data sharing environment, but for the purposes of this paper it's important to stress that row level locking in a data sharing environment can and typically does introduce significant overheads except in some particular cases, such as high-volume concurrent insert processing.

Row-level locking is only required for update applications, but even so many applications don't obtain much benefit if any from using row-level locking instead of page-level. That's for several reasons. For example, in OLTP environment characterised by random access, the rows being retrieved or updated by concurrent transactions might be rarely found on the same data page as each other. Or consider a batch program reading and updating data sequentially: if there is little or no contention with other batch jobs or online transactions, then row-level locking performance will be worse because more locks will be acquired, with a higher CPU cost and longer elapsed times.

However, there are some cases which mandate row-level locking. Amongst the sort of applications in this category are:

- Where rows being updated by concurrent transactions are likely to be located on the same page, resulting in unacceptable locking contention. This applies where the rows themselves are not subject to contention.
- For tables which are subject to high-volume concurrent inserts and which are defined with APPEND and MEMBER CLUSTER, to prevent exponential growth in the size of the table combined with inefficient space use.

Materialised result sets

To return to materialised result sets, another consideration for applications reading but not updating rows via a read-only cursor on a materialised result set is the fact that the application doesn't know which version of the row it's reading: for example, a 'hot' row that is frequently updated by other transactions could be updated one or more times since the result set was materialised, depending on the latency between result set materialisation and the row fetch operation. In effect, the application doesn't know if the row data values are current or have been updated since the result set was materialised. An application reading and then updating the row using safe optimistic locking techniques is protected, but this is a more difficult problem for the read-only application where data currency is important. In an OLTP environment, the number of affected applications is probably quite small, but in some cases, this could be an important consideration. Having cautioned against access-path dependent cursors, one way to protect against this is to make sure there is at least one index that satisfies the predicates used. As always, the need for the index must be balanced against the cost of index maintenance.

One other consideration is that if a materialised result set is very large, and therefore takes a long time to materialise, some of the data rows can be changed by other concurrent transactions during the materialisation process. The only way to control this is to run with LOCKSIZE TABLE or TABLESPACE, which will cause contention in an OLTP environment.

There is no 'silver bullet' solution to this problem. Resolving this requires understanding the data currency requirements of the application and choosing the best mitigation against unwanted side effects.

RR/RS and non-materialised result sets

There are some considerations for RR/RS applications and non-materialised result sets. If the result set is materialised, then the application knows all the data row values were current as of cursor open time; the data row values will therefore not change while the application is running because the share locks acquired at cursor open time will be retained until the next commit. However, for a non-materialised result set which takes a long time to consume, it is quite possible that rows are updated between cursor open time and row fetch time, when the rows are evaluated, and the locks are acquired. Again, this probably affects a small number of applications.

It's quite possible that there is no effective solution to a problem like this, and the best that can be done is to understand the risks and decide whether those risks are acceptable or not. However, it's unlikely that RR/RS would be used in an OLTP environment because they inhibit concurrency.

Lock avoidance

We've discussed lock avoidance in several previous sections, without going into any detail about the lock avoidance mechanism or the impact it can have on performance, especially in data sharing. Here I want to provide just enough information to help application developers understand the role they play in ensuring effective lock avoidance. Lock avoidance is a technique used by Db2 to avoid taking locks for read-only requests where possible. This does not imply an uncommitted or dirty read, but it does mean that Db2 will avoid taking a lock for a read-only request if it can guarantee that the pages where the target rows are located are logically and physically consistent. This is important for two reasons: it can improve concurrency and reduce contention by avoiding unnecessary S-locks which can delay updaters; and it avoids the CPU overhead of lock acquisition and release, and more importantly, of contention management, especially in a data sharing environment.

The discussion in this article is limited to just one part of the lock avoidance mechanism, known as the *commit log sequence number* or CLSN. Db2 tracks the time of the latest update to each page in a partition (including all UTS partitions) or tablespace (for segmented and simple tablespaces). In a non-data sharing environment, it also tracks the start time of the oldest uncommitted unit of work that has updated the partition or tablespace. This is the page-set CLSN (where page-set means the tablespace or partition). When these two values are compared, if the time of the update to the data page is earlier than the page-set CLSN, then Db2 knows all the data on the page has been committed and can therefore employ lock avoidance.[22] Otherwise, Db2 is unable to use lock avoidance for rows on this page.

The CLSN checking part of the lock avoidance mechanism operates differently in a data sharing environment, but the behaviour changes somewhat in Db2 12. To start off with the behaviour prior to Db2 12, when a partition or tablespace is group buffer pool-dependent (at least two members accessing the object, with at least one of these updating the object), then Db2 uses a *global commit log sequence number* (global CLSN) value, which is in fact the earliest start time of all uncommitted URs, across all members, across all page sets. This means that a single long-running UR can make lock avoidance less effective for all applications running in a data sharing group: the earlier the global CLSN value, the more likely it is that the most recent update to a page is later than the global CLSN, meaning that lock avoidance is less likely to be successful, even with the additional checking Db2 does.

Db2 12 made a big step forward in reducing the impact of long-running URs, with each member of the data sharing group maintaining a list of approximately the 500 oldest partition/tablespace CLSN values. When Db2 12 performs CLSN lock avoidance checking this list is checked first. If the partition being accessed is in the list, Db2 uses the associated CLSN. Otherwise, the highest CLSN value in the list is used as the global CLSN value for lock avoidance checking.

Nevertheless, the lesson from this, even in Db2 12, is to keep the CLSN values as recent as possible, to maximise the opportunity for lock avoidance and to minimise contention. It's hard to overestimate how important this is, as locking is an expensive business, especially in data sharing. There are two best practices to follow to maximise lock avoidance:

- Commit as frequently as practical, to avoid long-running URs pushing the CLSN value far into the past.
- Defer any updates to as late as possible in the transaction, to keep the CLSN value for each affected tablespace or partition as recent as possible. This is because the CLSN value associated with a UR is determined by the time of the first update statement, not the start of the transaction. In any event, this is a best practice for minimising contention by holding onto X-locks for as short a time as possible.

22 In fact, Db2 doesn't give up on lock avoidance if the latest update on the page is more recent than the start time of the oldest UOR against the partition – it uses additional techniques to check whether or not the row being accessed consists of committed data.

Db2 has one further technique to use when checking to see if can exploit lock avoidance, the 'possibly uncommitted' (PUNC) bit in the prefix of each row as stored in the data page. This bit is inaccessible to the application and is only for Db2 internal use. In summary, if a row is being evaluated and the page fails the page-set CLSN test, then Db2 tests that row's PUNC bit. If it is off, then the row is committed and Db2 can avoid taking the lock. Db2 periodically resets these PUNC bits (if on). This is done asynchronously to any units of recovery, which is why the bit is called the 'possibly uncommitted' bit – it can remain on even after the row has been committed, until Db2 resets it. However, there is nothing the application developer can do to influence PUNC bit setting, testing, or unsetting.

Conclusion

That wraps up this paper on Db2 locking for application developers.

Once the mechanics of Db2 locking have been grasped, together with the issues around data currency, then you should be in a good position to code your applications outlined in the later sections of the paper, thus avoiding loss of data integrity. If you have any questions, please submit them via the document download page at https://www.triton.co.uk/db2-for-z-os-locking-for-application-developers

Talk to our **expert team** about how to combine your application programming techniques with your Db2 for z/OS locking strategy to ensure data integrity.

Gareth Copplestone-Jones
Infrastructure Services Director
+ 44 (0) 7734 325293
gareth.copplestone-jones@
triton.co.uk

Rob Gould
Business Development Lead
+44 (0) 7766 838 904
rob.gould@triton.co.uk

triton consulting

triton.co.uk