CONCURRENCY

Learning outcomes: the student will be able to describe and explain

1. fundamental problems with multiple users/transaction executing simultaneously in a database system
2. locks and their use in solving concurrency problems
3. Serializability: fundamental criterion of correctness for concurrent transactions
4. Locking and serializabilty : 2Phase Locking protocol
5. Isolation levels: degrees of safety.
6. problems with locks : livelock, deadlock, deadlock resolution.
7. Intent locks :
8. concurrency support in SQL; and use of cursors in programming.

**Introduction.**

Database Systems are normally multi-user systems. In operating systems multiple users complete for system resources such as processor time, printers etc. We noted in the section on Recovery that I/O subsystems may have their own processing power therefore they are able to work independently of the central processing unit. I/O buffers and the CPU can work simultaneously, the former on data transfer to disk and the latter on executing programs/system processes. Programs completing for actual CPU time are managed by the process scheduler to execute in an interleaved manner.

In the database environment, we have multiple transactions (programs/processes) operating apparently simultaneously as before. The shared resource in a database system is the CPU and the data itself. This course is interested in **the concurrent access to data** by multiple transactions.

The main difference between OS and DB concurrency is therefore

1. DB must control a sequence of related operations organized as a logical unit (transaction) for consistency. OS concerned with single operation, no sequence, no consistency
2. The shared resource in Op. Sys. Is a physical unit e.g. a printer or file. In DB it is a value in row in a table. Note, a file can have any structure so it’s harder to control, but a table always is a grid, therefore control is possible.

**Context**: These differences have consequences for the DBMS; Concepts from Op Sys concurrency must be further developed/replaced to handle concurrency in the Database environment. Implementing transactions give rise to unique problems not applicable to standard Op Sys. Also the administration of DBMS concurrency becomes an issue e.g. many more items to administer and check for conflict in a DB.

We deal with the first element: New problems caused by transactions. We will return later with concepts such as Intent Locks that handle the administration issues.

**Three problems with concurrent access to data**

1. **Uncommitted Dependency**
2. **Inconsistent Analysis**
3. **Lost Update**

**Note: OS operations (Read,Write); DBS ops (Select, Insert, Update, Delete)**

**We will use read(cost) for simplicity of display, but Read = DBMS Select**

1. **Uncommitted Dependency (Also called Dirty Read)**

|  |  |  |  |
| --- | --- | --- | --- |
| Time | TA1 | TA2 | cost |
| t1 | begin trans |  | 20 |
| t2 | read(cost) |  | 20 |
| t3 | Update cost=cost-10 |  | 20 |
| t4 | write(cost) | begin trans | 10 |
| t5 |  | read(cost) | 10 |
| t6 | rollback |  | 20 |
| t7 |  | Commit |  |

We will see in the Recovery section that a rollback operation returns the database value to its original value before the failing transaction took place. The problem here, as the name suggests, is that the second transaction has become dependent on the update made by another transaction. If that transaction subsequently fails then the recovery system will automatically work, bring the database once again to a consistent state as far as transaction 1 is concerned i.e. back to the state of the data when it began. However we can see that transaction 2 has been allowed read a value that basically never existed by time t6. The user can make decisions based on that value, therefore this error that must be prevented.

**Inconsistent Analysis** **(Non Repeatable Read)**

No concurrent updates, therefore not a lost update problem.

No rollback, therefore not uncommitted dependency problem.

As the name suggests this problem has to do with reading/analyzing the database rather than updating it. If a recheck (reread, recalculation) is (or was to be!) performed at commit time and found to be different then an inconsistent analysis has taken place.

In its basic form an inconsistent analysis for TA1 is therefore

|  |  |  |  |
| --- | --- | --- | --- |
| Time | TA1 | TA2 |  |
| t1 | begin trans |  |  |
| t2 | read(cost) |  | 20 |
| t3 |  | begin trans |  |
| t4 |  | Update (cost) | 10 |
| t5 |  | write(cost) |  |
| t6 |  | Commit |  |
| t7 | Commit |  | Was 20 now 10!!! |

Note : it is important to realize that no operation is known in advance. On paper you can see the entire execution plan; however the concurrency control must deal with operations as they come i.e. one after the other according to the real time they occur.

**Locks are taken at point of use not in advance at Begin Transaction time.**

Time TA1 TA2

1 Read x

2 Update y ok, TA1 reads latest y value at T4

3 Commit

4 Read y

5 Update x not ok, TA1 x value obsolete

incon analysis

Note if there was no TA2 commit, we would have a 2nd problem: uncommitted dep.

**Lost Update (Also called Non Repeatable Read)**

TA1 = Transaction 1, TA2 = Transaction 2. (read = select)

|  |  |  |  |
| --- | --- | --- | --- |
| Time | TA1 | TA2 | cost  SQL code issues by both transactions  Select cost  From Product  Where Prod# = ‘P3’ |
| t1 | begin trans |  | 20 |
| t2 | read(cost) | begin trans | 20 |
| t3 |  | read(cost) | 20 |
| t4 | Update cost=cost-10 |  | 10 |
| t5 | write(cost) | Update cost=cost\*10 | 200 |
| t6 | commit | write(cost) | 200 |
| t7 |  | Commit |  |

Transaction TA1 update at time t3 is lost i.e. it is overwritten by a later update from a concurrent transaction TA2 (time t4). The data in the database is now inconsistent with what it would have been if the transactions were run serially i.e. one after the other with no interleaved access to data.

Note: it looks like TA2 operates on an obsolete value at t5, however it is not the issue (assume both read to local variable/cache); the last write by TA2 causes TA1 to be incorrect.

There are two possible correct values for cost that we could expect given the two transactions. They are ? and ? .

Note, in terms of non-repeatable read, effectively after time t4, Transaction TA2 is suffering a non repeatable read due to the Update by TA1. Therefore, a lost update can never occur in a system that controls non repeatable reads.

A second version of this problem occurs if the second transaction updates the database based on the inconsistent value it had just read.

|  |  |  |  |
| --- | --- | --- | --- |
| Time | TA1 | TA2 | cost |
| t1 | begin trans |  | 20 |
| t2 | read(cost) |  | 20 |
| t3 | Update cost=cost-10 |  | 20 |
| t4 | write(cost) | begin trans | 10 |
| t5 |  | read(cost) | 10 |
| t6 |  | Update cost=cost\*10 | 100 |
| t7 |  | write(cost) | 100 |
| t8 |  | commit | 100 |
| t9 | rollback |  | 20 |

This is effectively a lost update at t9 and uncommitted dependency at t5.

**Locking : a solution for concurrency problems**

**Context:**

Locking is standard Op Sys concurrency control technique. It is usually introduced as part of the concurrent programming ‘readers and writers’ section of college courses. Note in Op.Sys concurrency the unit to be locked is the full file. We’ll take the database object to be locked as a tuple/row of a table.

A lock is a restriction of some kind placed on a database item. The two basic locks are Read, known as Shared (S), and Write (X), known as an eXclusive locks.

Locking is a concurrency control technique which requires a transaction to acquire a suitable lock on an object before it manipulates it. Concurrent activity is then handled using the following rules.

**Properties of Locks**

1. If a transaction 1 has an X lock on a tuple p, a request of any type (X or S) from any transaction 2 will be denied.
2. If T1 has an S lock on p then requests from T2 for an
3. S lock will be granted Request
4. X lock will be denied S X

S Y N

HAS X N N

Note that X subsumes S; i.e. if you have an X lock you can both Read and Write.

Context:

The properties of locks on their own do not control concurrency. It is the rules by which transactions are forced to use these locks that are responsible for controlling the interactions of concurrent transactions.

Remember, each transaction is a sequence of operations submitted by the user for execution. Each operation is part of a logical unit but submitted by the user at different times spanning the lifetime of the transaction.

To define the use of locks we define a

**Data Access Protocol ( Version 1 Weak)**

1. before reading you must acquire an S lock on the object. For update you must acquire an X lock. **Lock Promotion** : to cover the common case of ‘look first then update’, you can acquire the S lock and then **promote** the S to an X lock. (never hold S and X for the same object as X subsumes S)
2. if a request of any type cannot be granted for incompatibility reasons then the requesting transaction will go into a WAIT state until the request can be granted
3. commit or rollback releases all locks held by a transaction.

**Why promote locks?** Ans: do not administrate 2 locks when 1 would do!

We will revisit DQP later, to secure one aspect of the protocol which allows for the potential problem of releasing locks early.

We can now revisit the concurrency problems described earlier to see what effect locks and the above procedure(s) have on control of the concurrency.

**The Uncommitted Dependency Problem & locking**

The original problem was that TA2 was allowed see the value of 10 which was a partial updated value from TA1 i.e. the value is unstable as it is subject to possible failure. For locking to work we must prevent the read by TA2 until the value of cost is stable.

|  |  |  |  |
| --- | --- | --- | --- |
| Time | TA1 | TA2 | cost |
| t1 | begin trans |  | 20 |
| t2 | read(cost) S lock OK |  | 20 |
| t3 | Update cost=cost-10 X lock OK |  | 20 |
| t4 | write(cost) | begin trans | 10 |
| t5 |  | read(cost) S lock No | 10 |
| t6 | rollback | wait | 20 |
| t7 |  | S lock granted Read(cost) | 20 |
| t8 |  | commit | 20 |

**The Inconsistent Analysis problem & locking**

Exercise: explain, using an example, how locking would prevent an Inconsistent analysis.

**The Lost Update problem & locking**

The problem here is that Transaction TA1 update is lost i.e. it is overwritten by a later update from a concurrent transaction TA2. We must therefore prevent multiple, concurrent updates on a database object.

|  |  |  |  |
| --- | --- | --- | --- |
| Time | TA1 | TA2 | cost |
| t1 | begin trans |  | 20 |
| t2 | read(cost) S lock OK | begin trans | 20 |
| t3 | get X lock OK |  | 20 |
| t4 | Update cost=cost-10 | read(cost) S lock NO | 10 |
| t5 | write(cost) | Wait | 10 |
| t6 | commit | Wait | 10 |
| t7 |  | S lock granted Read(cost) | 10 |
| t8 |  | get X lock OK | 10 |
| t9 |  | Update cost=cost\*10 | 100 |
| t10 |  | write(cost) | 100 |
| t11 |  | commit | 100 |

**SQL : Phantom** : problem not covered by locking rules we have to date but mentioned in the SQL standard. **Similar to Non repeatable read for a set of rows**.

Transaction TA1 = Select Count(\*) from Supplier Where City = ‘Paris’

Transaction TA2 = Insert into Supplier Values(‘S6, ‘Paris’)

TA1 TA2

Select

- Read & lock first tuple -

- Read & lock next tuple -

- Read & lock next tuple -

- - Insert new tuple, X lock

- Commit, release X lock

- Reread & relock all Paris tuples

- New result

|  |  |  |  |
| --- | --- | --- | --- |
| Sno | Sname |  | City |
| S1 |  |  | Paris |
| S2 |  |  | London |
| S3 |  |  | Paris |
|  |  |  |  |
|  |  |  |  |

S6 Paris Ta2 Xlock & insert ok, no problem

according to our locking rules to date, but phantom occurs so need solution

Solution: Lock the table? Why/Why not?

Alternative Solution: if there is a secondary key with an index defined on it then an insert must update that index e.g. city index. If TA1 has a read lock on the index, then any concurrent update activity can be handled i.e. deny request & place on wait. (TA2 cannot (get X lock) on the S locked index)

Another example for phantom:

When you do a select with a where condition e.g. select \* from student where course\_code = 'ITS3', the system might (depending on the implementation) S lock all the rows for the students in that ITS class; let’s say 30 rows for the class of 30 students. But there is nothing to stop a concurrent transaction inserting a new student into the table using the ITS3 course\_code; All of a sudden there are 31 students in ITS3 but the reading(selecting) Transaction originally saw only 30. This is a phantom.

The question is can the reading(selecting) transaction prevent something that didn’t exist when they began their transaction. This new row did not exist when the reading(select) transactions got S locks on the 30 rows; Instead of tuple locks, you could lock the entire data table to stop any inserts/updates. Alternatively, if there is an access path such as an index on the course\_code column, simply lock that index for value ITS3; concurrent transactions that need to modify the table must also update the index. These can not proceed as they are locked out of updating the index.

**Context:**

We have our basis elements and the rules governing how they are used. But how do we gauge if these rules work correctly? We need to define some notion of correctness.

**Serializability**

Serializability is the **criterion for correctness for concurrency control**. We know that running transactions serially i.e. one after the other is a guaranteed way of insuring correctness, however in an efficient multi-user system, concurrency is a must. We also know that transactions operating on different/disjoint sections of the database or read only, must also be correct as no interference is possible and therefore no inconsistencies can occur. We need to define what general concurrent activity is correct. Note: assume interactive/on-line systems therefore we do not know in advance (except for embedded/precompiled) what to expect i.e. cannot analyse and determine correctness.

Time

Serial T1—T2—T3……

zero concurrency but 100% correct i.e. maximum serializability

Time

T1 100% concurrency: all transactions executed in parallel.

T2

T3

--

All running together, no restrictions, correctness almost impossible to maintain in a shared data environment.

A **schedule** is any execution of a set of transactions i.e. the scheduling of the reads and writes of the transactions. A **serial schedule** is transactions run one after another. An **interleaved schedule** is concurrent execution of the reads/writes.

**An interleaved schedule is termed serializable if it produces the same results as some serial execution of the same transactions.**

Note that two different schedules of the same set of transactions can produce different results; i.e. TransactionA followed by TransactionB m ay produce different results than B then A yet both are correct : see lost update problem. The same therefore applies to interleaved schedules. A given interleaved schedule can be serialised in a number of ways, each of which can be correct.

Obviously if transactions read only or read/update separate items no problems can arise and any concurrency (schedule) is guaranteed correct. From our earlier examples we can see that it is ordering of updates and reads that can cause problems/conflicts and so we must examine in more detail. The order of conflicting activity must be the same as some serial ordering of the operations.

**Context:** We are now in a position to gauge if our rules of lock operation adhere to our rules of correctness for concurrent transactions working in a DB environment.

**Locking and Serializability**

The Data Access Protocol outlined earlier **does not guarantee serializability**. The problem with the protocol is that it **does NOT explicitly prohibit the release of a lock(s) early** i.e. before the transaction terminates.

This can lead to the following 2 possibilities:

If a transaction, say TA1, is permitted to

1. release or
2. downgrade a lock

before a COMMIT/ROLLBACK and continues processing other data items, then it is always possible to devise a set of concurrent transactions to render the database inconsistent from TA1s perspective i.e. update the values of the data items released; these have now changed during the lifetime of TA1 and are therefore unreliable.

We need to introduce a protocol that guarantees serializability.

**Two-Phase Locking Protocol**.

1. Before operating on any object a transaction must acquire a suitable lock on that object.
2. After releasing a lock, a transaction must never go on to acquire any more locks.

We can state that if all transactions follow the 2PL protocol then all interleaved schedules are serializable.

**DAP Version 2 Strong (preventing early release of locks)**

**Modify the DAP point 3 to state that**

* **all locks must be held until COMMIT or ROLLBACK**

**Why might a lock be released? See note on U locks for Index Root nodes**

**Context:** Do we always require absolute correctness? This is obviously an ideal, but enforcing 100% correctness comes at a cost and may have undesirable consequences in certain types of situations (i.e. types of applications). We therefore introduce Isolation Levels.

**Isolation Levels**:

What are they?

These are levels of correctness for concurrent transactions. Different levels can be defined which describe the amount of interference a transaction is willing to accept from other concurrent transactions. The highest level permits serializable schedules only (100% correct). Lower levels allow more interference and therefore tolerate consistency errors occurring. Note they may not actually occur but the levels carry more **risk** of inconsistency (note this is optimistic approach hoping no error occurs).

Why are Isolation levels useful?

Applying 2PL to guarantee serializability/ correctness has implications for the amount of concurrency on the system e.g. if we can release locks early then other transactions can gain access to the database objects without waiting and therefore throughput on the system is increased i.e. concurrency is high. On the other hand the stricter we are in implementing serializability means that we have less concurrency but we have greater guarantee of correctness. To reduce resource contention thereby increasing concurrency some systems allow less than 100% serializability.

Further detail:

The DBMS provides facilities for the user (application programmer) to set the level for the transaction. Note that systems vary in their implementation of isolation levels, some have limited levels, and others only have one (serializable).

We will see in the concurrency practical that Ingres allows user specification of how locks are to be treated by the system thereby effectively implementing a level even though it does call it that i.e. In Ingres we can specify that Select is to take no Slock; this is not a level, but allows less than 100% serializability as it basically turns off DAP.

SQL recommends a SET ISOLATION LEVEL command for programmers to use and it defines 4 levels. The figure below defines what problems each level handles.

MySQL transactions ref: <http://zetcode.com/databases/mysqltutorial/transactions/>

|  |  |  |  |
| --- | --- | --- | --- |
| Isolation Level | Dirty Read | Non-repeatable Read | Phantom |
| READ UNCOMMITTED | Y | Y | Y |
| READ COMMITTED | N | Y | Y |
| REPEATABLE | N | N | Y |
| SERIALIZABLE | N | N | N |

**Problems with Locking.**

We have seen how locking can be used as a method to control concurrency in a safe way. However locking can introduce problems of its own.

The main problem with Locking is called **Deadlock**. This is where two concurrent transactions have locks on some database items but cannot proceed further because each one is waiting (indefinitely ) for the database item that is locked by the other to be released. This is similar to deadlock found in Operating Systems where the lockable object is usually a system resource e.g. a printer. Let’s take the lost update problem introduced earlier.

|  |  |  |
| --- | --- | --- |
| Time | TA1 | TA2 |
| t1 | begin trans |  |
| t2 | read(cost) S lock OK | begin trans |
| t3 |  | read(cost) S lock OK |
| t4 | Update get X lock NO |  |
| t5 | Wait | Update get X lock NO |
| t6 | Wait | Wait |
| t7 | Wait | Wait |
| t8 | Wait | Wait |

**Deadlock** can be handled

(note: must be handled independent of concurrency correctness control i.e. removing DAP not a solution/prevention)

**Context:** although in medical terms, ‘prevention is better than cure’, this doesn’t translate to deadlock handling. We can assume deadlock will **not occur in most cases**, and so we simply track transaction activity and ‘cure’ deadlock after it occurs.

**Deadlock detection and resolution** Most systems opt for this strategy. To detect deadlock we must firstly monitor and track locks on database objects and secondly have some test for deadlock occurring between different transactions. The tracking is performed in a Wait For Graph (WFG) see below.

TAj TAi

TAx

T1 T2

Nodes are transactions and directed edges between node represent a wait by one transaction for the item locked by the other. A cycle then represents deadlock, therefore deadlock detection programs test the WFG for cycles.

One of the transactions involved in the deadlock is chosen as victim and rollback.

**How to choose the victim**? We should aim to minimize the impact/workload on the system due to deadlock resolution? Note that choosing the oldest may result in large undo, but that old transaction may be a problem transaction blocking up the system for some time (Recall long trans). Possible use activity i.e. old and very active may be important.

Note, the DBMS can automatically restart the victim (all operations logged) without user interaction. By the end of the Rollback and rerun we can expect the problem causing data item will be now available.

2nd problem of locking. (Livelock / Lock out)

This problem is a by-product of using locks. The situation **is essentially a problem of an idle or long transaction (as in long lasting in time). Note, we will also see this issue of long transactions in the Recovery section.**

|  |  |  |
| --- | --- | --- |
| Time | TA1 | TA2 |
| t1 | begin trans |  |
| t2 | read(cost) S lock OK | begin trans |
| t3 |  | read(cost) S lock OK |
| t4 | Update get X lock NO |  |
| t5 | Wait | idle |
| t6 | Wait | Idle |
| t7 | Wait | Idle |
| t8 | Wait | idle |

Transaction TA1 is prevented from continuing due to a lock request that has been denied. TA2 is causing the ‘blockage’ in the system.

Solution? Programmers could Set Timeouts for their transaction(s). If a request is denied, and put on a wait, when the timeout limit is reached, the request is terminated (they come off the wait queue) and control is returned to the user. This is useful as now the application program can at least continue e.g. by giving the end user a message ‘Sorry that Update could not be completed at this time’ or ‘Sorry, operation failed, would you like me to retry?’

**Intent Locking: is a Lock management issue, it is not for concurrency correctness.**

Although it should be noted that intent locking does restrict some database activity in certain situations, the notion of correctness in the sense **of serializability is not an intent locking concern**. Concurrency is controlled by S,X at lower level as normal using a protocol such as the DAP or variant(2PL).

Intent locks describe transaction activity on the child object. This enables faster processing of lock requests and conflict checks by the DBMS lock manager.

**Granularity hierarchy (parent-child lock hierarchy)**

DB -----high level lock; simple admin and conflict check but low concurrency

Table

Table/tuple in notes but open to any 2 levels of hierarchy.

Tuple

Field------ expensive admin and compatibility check but high concurrency

**Trade-off:** the finer the granularity of the locks, the more concurrency; but the coarser (higher level) the fewer locks and therefore less admin/overhead but less concurrency.

The admin cost relates to storing and processing the current set of locks on the database. In a database with tables of millions of rows this is considerable work. In addition the issue is further complicated each time new lock requests are made. The DBMS must have some way to check if the new lock request conflicts with any existing locks. For example, if a transaction wants to X lock an entire table; **does the system need to check every row in that table for locks? Examining each lock for conflict is not acceptable. Intent locks are introduced to solve this problem.**

We introduce two intent lock, IS (intent to Share) and IX (Intent to Write).

**Intent Lock Protocol:**

**Before a transaction can obtain a protecting lock on an object (child), it must first acquire a suitable *intent* lock on the parent object above it on the lock hierarchy. i.e. S row lock requires IS table lock, X lock requires IX.**

The DBMS can now examine quickly the table, to get an indication of the type of activity currently occurring on the rows of that table.

* An IX table lock will allow concurrent readers and writers on child objects but disallow any readers or writers (at parent level).
* An IS table lock allows concurrent readers and writers below it but will prevent any attempt to X lock the parent level.

So, in general Intent Locks describe transaction activity on the child objects and enable fast decision making by the DBMS lock manager for lock requests & conflict resolution.

Transaction Requests Lock

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | X | IX | S | IS |
| X  Transaction  Has lock | N | N | N | N |
| IX | N | Y | N | Y |
| S | N | N | Y | Y |
| IS | N | Y | Y | Y |

These are levels of correctness for concurrent transactions. Different levels can be defined which describe the amount of interference a transaction is willing to accept from other concurrent transactions. The highest level permits serializable schedules only (100% correct). Lower levels allow more interference and therefore tolerate consistency errors occurring. Note they may not actually occur but the levels carry more **risk** of inconsistency (note this is optimistic approach hoping no error occurs).

Why are Isolation levels useful?

Applying 2PL to guarantee serializability/ correctness has implications for the amount of concurrency on the system e.g. if we can release locks early then other transactions can gain access to the database objects without waiting and therefore throughput on the system is increased i.e. concurrency is high. On the other hand the stricter we are in implementing serializability means that we have less concurrency but we have greater guarantee of correctness. To reduce resource contention thereby increasing concurrency some systems allow less than 100% serializability.

Discuss in class the following SQL for a student table in a database

Select \*

From Student

Where Faculty = ‘Science’ OR

Faculty = ‘Business’

Select \*

From Student

Where Dept\_code = ‘Computing’

Select \*

From Students

Where Cit\_no = ‘R0124’

Note: TA1 = one row lock; TA2 = potentially small set of locks; TA3 = potentially large set of locks (i.e. many student in Business or computing).

Another example: Bank application; monthly statement report

Select \*

From CustomerAccounts;

The system doesn’t know in advance how many records will be locked; it just takes locks as each row is read.

This is very inefficient and costly. If you knew you were going to ‘sooner or later’ going to lock most rows, you could have just locked the entire table.

Is there any Solution to this problem? **Lock Escalation**

Accumulate locks as you use them **but trade in when you hit the preset limit**

|  |  |  |  |
| --- | --- | --- | --- |
| C# |  | Bal | Lock  -Accumulate many tuple S locks  -Reach limit  -Trade for one table S lock |
|  |  |  | S |
|  |  |  | S |
|  |  |  | S |
|  |  |  |  |
|  |  |  |  |

**To Escalate means to trade in a set of lower level child locks for 1 parent lock.**

Note, both lock escalation and promotion are not guaranteed. Why?

**Multi-Version Concurrency Control (MVCC)**

In certain application such as Web browsing, where there are many transactions(sessions) reading, but few that update, and even less few that commit an update. A problem with ‘locks’ is that the writers effectively block readers. If this delay cost is unacceptable, an alternative concurrency control mechanism is to use ‘versioning’. Essentially, each transaction operates on its own private copy (version) of the shared data resource (be it a row, table or entire document).

The DBMS longer needs to manage slow locks and queues for access; it now needs to manage the versions of the data, particularly when any modification commits.

Note, any updates that are abandoned (i.e. not committed) can be just deleted.

A DBMS can control what version is correct in time, by using timestamps & transaction identifiers.

<http://en.wikipedia.org/wiki/Multiversion_concurrency_control>

**Cursors. Concurrency and Programming**

Cursors are a procedural programming construct used by application programmers to manage a set of records retrieved from a database server for use in their client application program e.g. a web application. Alternatively they are used in stored procedures (e.g. Triggers ,Events) to process individual records in a set. Recall, SQL is non procedural and set based; Java/C programs must use loops and cursors to process a set or records as input. The Java ResultSet maintains a (hidden) cursor into the database: <http://docs.oracle.com/javase/1.4.2/docs/api/java/sql/ResultSet.html>

DECLARE CURSOR Computing\_Students As

Select \*

From College

Where Dept\_code = ‘Computing’

The application program can execute this to retrieve a set of records from the database server. **The problem for the DBA is that there are different types of Cursor, and each one have different impacts of database management and performance.**

Cursors differ in that they

* Where the record data set is stored (on the server or on the client)
* Static or Dynamic? How the record set is maintained as concurrent transactions execute on the database. That is, is the record set kept up to date as updates, inserts and deletes occur? Some cursors are ‘Static’ in that they are effectively a snapshot of the database at the time (i.e. not kept up to date). Other are ‘Dynamic’ in that they must be kept up to date.
* Are they Scrollable or not. A simple cursor is read forward only, no re-read.

Depending on the data set size and activity, it might be better to have a Dynamic cursor on the client than a Static cursor on the server. For example, all users storing the record sets on the server may result in significant workload on the server.

So ‘record sets’ have implications for data and memory management (reference the Configuration/Architecture section).  We should be aware that we have caches for data; global data allocation (shared) and per connection.

As a DBA, you should understand that there is no general rule for this, but the effect on performance can be considerable, so some analysis work will have to be done in consultation with programmers. In addition, **programmers must be made aware that if they do not specify the cursor type, defaults might be used which are totally inappropriate.**

**Cursors** may be called different names in different implementations e.g. record set or result set. <http://docs.oracle.com/javase/1.4.2/docs/api/java/sql/ResultSet.html>

Do Cursors take locks?

Cursors manage sets of records and may take and release locks as required to do its work. Transactions are a different database construct. Transaction locks are defined by its isolation level and implemented by the DAP (or 2PL). Some DBMS manage Cursors within the scope of isolation levels, others may just have Cursor locks.

References

<http://dev.mysql.com/doc/refman/5.0/en/sql-syntax-compound-statements.html>

<http://msdn2.microsoft.com/en-us/library/ms180169.aspx>

<http://www.databasejournal.com/features/mssql/article.php/1439731>

<http://docs.oracle.com/javase/6/docs/technotes/guides/jdbc/getstart/resultset.html>

<http://java.dzone.com/articles/jdbc-faq-resultsets>

**Update Locks ( U locks)**

These are similar to X locks, in that they are taken for update statements. However they are normally used to lock access paths e.g. index or cursor. If these items are not actually updated the lock can be downgraded or reduced in class (U to lower S lock) e.g. a cursor may position itself on a tuple (to read and check if it’s to be updated) however it may not qualify for the update so the cursor moves on.

Serializability implemented by 2 phase and S&X locks; nothing to do with access mechanism.

Release or downgrade for performance reasons; in B.Tree indexes all transactions must access via Root node; i.e. contention point(bottle neck); the transaction may want to lock the access mechanism(index) to prevent phantoms so only lock further down the tree index; do not lock the entire set of index nodes in the path from the root. Therefore once you pass through (i.e. finish reading) the root node, downgrade or release the lock to allow concurrent activity. When the system finds the section of the index of interest to the query; you can then lock that node if a modification operation is executed.

We also know that DBMS use access mechanisms to improve performance e.g. Indexes. However, we must now consider that these access mechanisms are shared resources, subject to concurrency. These mechanisms are used ‘in the background’, e.g. by optimisers, but users may be unaware of their existence/use. Accessing a shared resource for reading is benign as it is an operation that accepts sharing, but updating is an eXclusive operation. Mixed activity (reads/writes) incur waits if consistency is threatened (if it is against the protocol in use) so the DBA should be aware of not just the quantity of transactions, but the operations performed, and their frequency of occurrence (how often they occur, do they incur waits etc). Database objects like these are potential bottlenecks for performance.

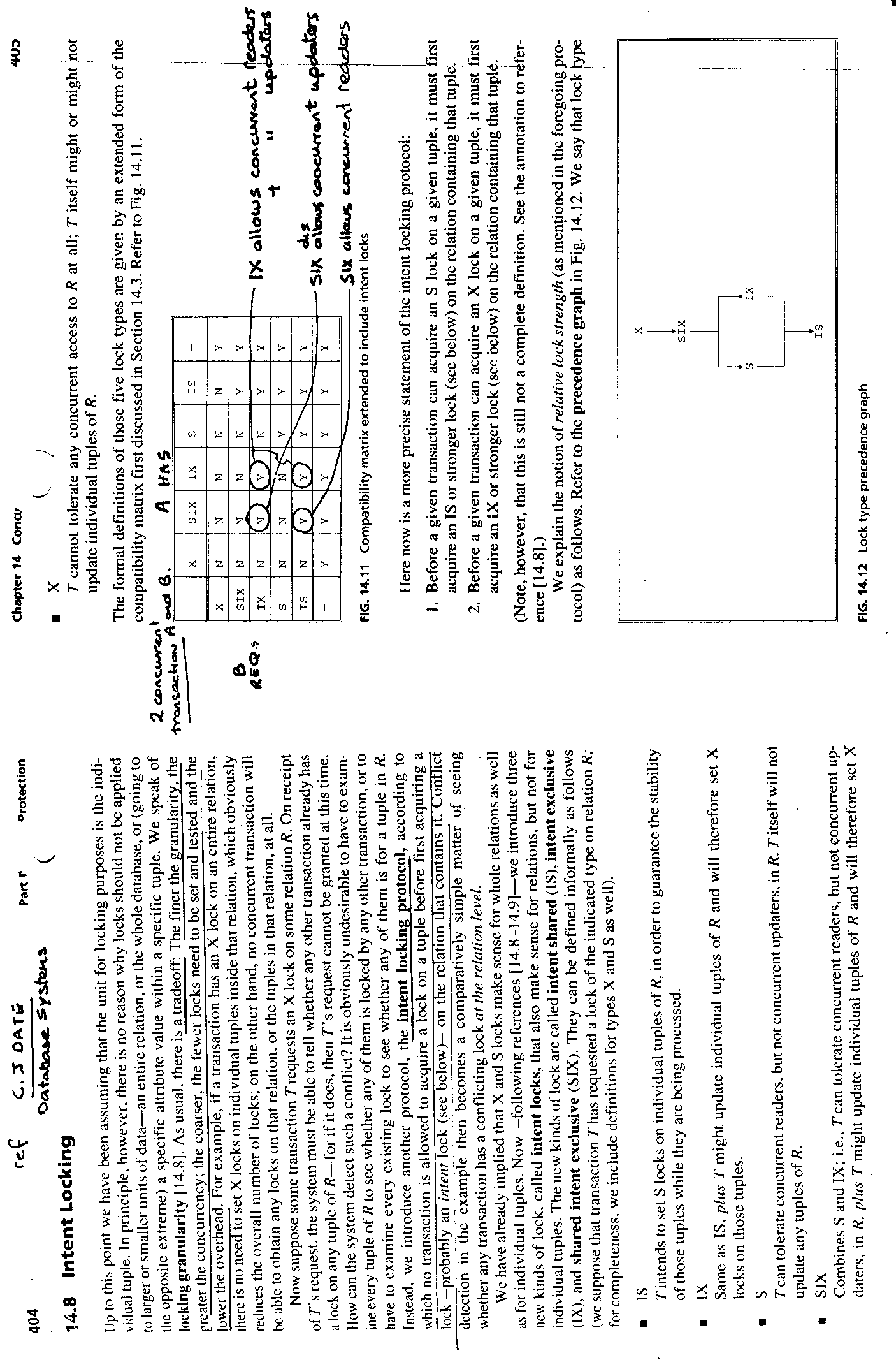
Concurrent users consume / complete for resources (CPU and Memory). Therefore we must balance the demands for more concurrency with the limitations of the resources.

Reference for MySQL implementation of concurrency

<http://dev.mysql.com/doc/refman/5.0/en/innodb-transaction-model.html>

Typical exam questions seek to test whether the student has attained the learning outcomes on page 1.

1. fundamental problems with multiple users/transaction executing simultaneously in a database system
2. What are the major differences between concurrency control in an operating system and a database system?
3. locks and their use in solving concurrency problems; give examples
4. Serializability: fundamental criterion of correctness for concurrent transactions
5. Locking and serializabilty : 2Phase Locking protocol
6. Isolation levels: degrees of safety. What are they, why are they used
7. How to implement an isolation level?
8. problems with locks : livelock, deadlock, deadlock resolution.
9. Intent locks: What are they, why are they used.
10. Does the use of Intent locks guarantee serializability? Explain.
11. Differentiate between local promotion and lock escalation. Why are they both used; explain using an example.
12. concurrency support in SQL;
13. General theory uses the terms Lost Update etc to describe fundmental problems in concurrency; What are the equivalent terms in SQL?
14. What are phantoms? Explain using an example with tuple(row) level locks and the 2PL protocol. Can an SQL Delete create a Phantom?
15. Cursors in programming; explain the different types from a performance perspective.
16. Indexing and Concurrency

APPENDIX: EXTRA REFERENCE NOTES ON INTENT LOCKS AND CURSORS. ****

