Isolation in Relational Databases

Chapter 21

1

What's Different About Locking in Relational Databases?

- In the simple databases we have been studying, accesses are made to a *named item* (for example *r(x)*).
  - *x* can be locked
- In relational databases, accesses are made to *items that satisfy a predicate* (for example, a SELECT statement)
  - What should we lock?
  - What is a conflict?

2

Conflicts in Relational Databases

Audit:

```
SELECT SUM (balance)
FROM Accounts
WHERE name = 'Mary';

SELECT totbal
FROM Depositors
WHERE name = 'Mary'
```

NewAccount:

```
INSERT INTO Accounts
VALUES ('123','Mary',100);

UPDATE Depositors
SET totbal = totbal + 100
WHERE name = 'Mary'
```

- Operations on Accounts and Depositors conflict
- Interleaved execution is not serializable

3

What to Lock?

- Lock tables:
  - Execution is serializable but ...
  - performance suffers because lock granularity is coarse
- Lock rows:
  - Performance improves because lock granularity is fine but ...
  - execution is not serializable

4

Problem with Row Locking

time ↓

- Audit
  - (1) Locks and reads Mary's rows in Accounts
- NewAccount
  - (2) Inserts and locks new row, *t*, in Accounts
  - (3) Locks and updates Mary's row in Depositors
  - (4) Commits and releases all locks
- Audit
  - (5) Locks and reads Mary's row in Depositors

5

Row Locking

- The two SELECT statements in Audit see inconsistent data
  - The second sees the effect of NewAccount; the first does not
- **Problem**: Audit's SELECT and NewAccount's INSERT do not commute, but the row locks held by Audit did not delay the INSERT
  - The inserted row is referred to as a *phantom*

6

Phantoms

- Phantoms occur when row locking is used and
  - T_1 SELECTs, UPDATEs, or DELETEs using a predicate, *P*
  - T_2 creates a row (using INSERT or UPDATE) satisfying *P*
- Example:

T_1: UPDATE Table
SET Attr = ….
WHERE *P*

T_2: INSERT INTO Table
VALUES (… satisfies *P*…)

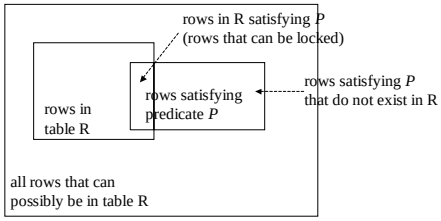
Phantoms

- INSERT and UPDATE cause phantoms with row locking.
- **Question**: Why does DELETE not cause a similar problem with row locking?
  - **Answer**: A row that has been read cannot be deleted because it is locked

Preventing Phantoms

- Table locking prevents phantoms; row locking does not
- *Predicate locking* prevents phantoms
  - A predicate describes a set of rows, some are in a table and some are not; e.g. *name = 'Mary'*
    - A subset of the rows satisfying *name = 'Mary'* are in Accounts
  - Every SQL statement has an associated predicate
  - When executing a statement, acquire a (read or write) lock on the associated predicate
  - Two predicate locks conflict if one is a write and there exists a row (not necessarily in the table) that is contained in both

Phantoms



Preventing Phantoms With Predicate Locks

Audit:
SELECT SUM (balance)
FROM Accounts
WHERE name = 'Mary'

NewAccount:
INSERT INTO Accounts
VALUES ('123','Mary',100)

- Audit gets read lock on predicate *name='Mary'*.
- NewAccount requests a write lock on predicate (*acctnum='123'* $\wedge$ *name='Mary'* $\wedge$ *bal=100*)
  - Request denied since predicates overlap

Conflicts And Predicate Locks

- Example 1

SELECT SUM (balance)
FROM Accounts
WHERE name = 'Mary'

DELETE
FROM Accounts
WHERE bal < 100

  - Statements conflict since predicates overlap and one is a write
    - There might be an account with bal < 100 and name = 'Mary'
    - Locking is conservative: there might be no rows in Accounts satisfying both predicates
    - No phantom involved in this (DELETE) case

Conflicts And Predicate Locks

- Example 2

```
SELECT SUM (balance)          DELETE
 FROM Accounts                FROM Accounts
 WHERE name = 'Mary'          WHERE  name = 'John'
```

  - Statements commute since predicates are disjoint.
    - There can be no rows (in or not in Accounts) that satisfy both predicates
    - No phantom involved in this (DELETE) case

13

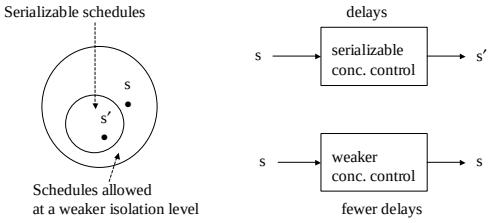
Serializability in Relational Databases

- Predicate locking prevents phantoms and produces serializable schedules, but is too complex to implement
- Table locking prevents phantoms and produces serializable schedules, but negatively impacts performance
- Row locking does not prevent phantoms and can produce nonserializable schedules
- What's an implementor to do?
  - Later we discuss more efficient locking methods (granular locking and index locking) that prevent phantoms and produce serializable schedules

14

Isolation Levels

- SQL defines several isolation levels weaker than SERIALIZABLE that allow non-serializable schedules and hence allow more concurrency



15

Isolation Levels

- Schedules that are produced by concurrency controls operating at isolation levels lower than SERIALIZABLE might be correct for some applications
  - We give examples later.
- SQL standard defines isolation levels in terms of certain anomalies they do or do not allow

16

Anomaly

- We have already talked about some anomalies
  - Dirty Read
  - Dirty Write
  - Lost Update
  - Phantom
- Now we discuss one more
  - Non-Repeatable Read

17

Anomaly - Non-Repeatable Read

T1	T2
SELECT SUM (balance) FROM Accounts WHERE name = 'Mary'	
	UPDATE Accounts SET balance = 1.05 * balance WHERE name = 'Mary'
SELECT SUM (balance) FROM Accounts WHERE name = 'Mary'	*does **not** introduce a phantom into predicate name='Mary'*

18

Non-Repeatable Reads and Phantoms

- With a phantom, execution of same SELECT twice yields *different* sets of rows
  - The second returns at least one row not returned by the first
- With a non-repeatable read, execution of same SELECT twice yields the *same* set of rows, but attribute values might be different

19

SQL Isolation Levels

- READ UNCOMMITTED – dirty reads, non-repeatable reads, and phantoms allowed
- READ COMMITTED - dirty reads not allowed, but non-repeatable reads and phantoms allowed
- REPEATABLE READ – dirty reads, non-repeatable reads not allowed, but phantoms allowed
- SERIALIZABLE – dirty reads, non-repeatable reads, and phantoms not allowed; all schedules must be serializable

20

SQL Isolation Levels

- Defining isolation levels in terms of anomalies leads to an ambiguous specification:
  - At what levels are dirty writes allowed?
  - Are there other anomalies that are not accounted for?

21

Statement Isolation

- In addition – statement execution must be isolated
  - DBMS might be executing several SQL statements (from different transactions) concurrently
  - The execution of statement involves the execution of a program implementing that statement's query plan
    - This might be a complex program
  - While the execution of transactions T_1 and T_2 might *not* be isolated, the execution of each statement within T_1 must be isolated with respect to the execution of each statement within T_2.

22

Locking Implementation of SQL Isolation Levels

- SQL standard does not say *how* to implement levels
- Locking implementation is based on:
  - **Entities locked**: *rows, predicates, …*
  - **Lock modes**: *read* and *write*
  - **Lock duration**:
    - *Short* - locks acquired in order to execute a statement are released when statement completes
    - *Long* - locks acquired in order to execute a statement are held until transaction completes
    - *Medium* – something in between (we give example later)

23

Locking Implementation of SQL Isolation Levels

- Write locks are handled identically at all isolation levels:
  - Long-duration predicate write locks are associated with UPDATE, DELETE, and INSERT statements
    - This rules out dirty writes
    - In practice, predicate locks are implemented with table locks or by acquiring locks on an index as well as the data
      - We discuss index locking later

24

Locking Implementation of SQL Isolation Levels

- Read locks handled differently at each level:
  - READ UNCOMMITTED: no read locks
    - Hence a transaction can read a write-locked item!
    - Allows dirty reads, non-repeatable reads, and phantoms
  - READ COMMITTED: short-duration read locks on rows returned by SELECT
    - Prevents dirty reads, but non-repeatable reads and phantoms are possible

25

Locking Implementation

- REPEATABLE READ: long-duration read locks on *rows* returned by SELECT
  - Prevents dirty and non-repeatable reads, but phantoms are possible
- SERIALIZABLE: long-duration read lock on *predicate* specified in WHERE clause
  - Prevents dirty reads, non-repeatable reads, and phantoms and …
  - guarantees serializable schedules

26

Bad Things Can Happen

- At every isolation level lower than SERIALIZABLE, bad things can happen
- Schedules can be
  - Non-serializable
  - Specifications of transactions might not be met

27

Some Problems at READ UNCOMMITTED

- Since no read locks are obtained, T_2 can read a row, *t*, write locked by T_1

T_1: *w(t)* *abort*
T_2: *r(t)* *w(t′) commit*
T_2 uses an aborted value to update db

T_1: *w(t)* *w(t) commit*
T_2: *r(t) w(t′) commit*
T_2 uses an intermediate value to update db

T_1: *w(t)* *w(t′) commit*
T_2: *r(t) r(t′) commit*
T_2 does not see a committed snapshot

  - Some DBMSs allow only read-only transactions to be executed at this level

28

Some Problems at READ COMMITTED

- Non-repeatable reads:

T_1: *r(t)* *r(t) commit*
T_2: *w(t) commit*

- Lost updates:

T_1: *r(t)* *w(t) commit*
T_2: *r(t) w(t) commit*

29

Problems at REPEATABLE READ

- Phantoms:

Audit: *r(pred)* *r(t′) commit*
NewAccount: *insert(t) update(t′) commit*

  - *t* satisfies *pred*
  - A constraint relates rows satisfying *pred* and *t′*

30

Implications of Locking Implementation

- Transactions can be assigned to different isolation levels and can run concurrently.
  - Since all write locks are long-duration predicate locks and SERIALIZABLE transactions have long-duration predicate read locks, SERIALIZABLE transactions are serialized with respect to all writes.
    - A SERIALIZABLE transaction either sees the entire effect of another transaction or no effect.
  - A transaction at a lower level does not see the anomalies prohibited at that level.

31

Implications of Locking Implementation

- Even though all transactions are designed to be consistent,
  - Transactions executed at lower isolation levels can see anomalies that can cause them to write inconsistent data to the database
  - Transactions executed at *any* isolation levels can see that inconsistent data and as a result return inconsistent data to user or store additional inconsistent data in database

32

CURSOR STABILITY

- A commonly implemented isolation level (not in the SQL standard) deals with cursor access
- An extension of READ COMMITTED:
  - Long-duration write locks on predicates
  - Short-duration read locks on rows
  - Additional locks for handling cursors

33

Cursors at READ COMMITTED

- Access by T_1 through a cursor, C, generally involves OPEN followed by a sequence of FETCHs
  - Each statement is atomic and isolated
  - C is INSENSITIVE: rows FETCHed cannot be affected by concurrent updates (since OPEN is isolated)
  - C is not INSENSITIVE: some rows FETCHed might have been updated by a concurrent transaction, T_2, and others might not
    - Furthermore, T_1 might fetch a row, T_2 might update the row and commit, and then T_1 might overwrite the update

34

CURSOR STABILITY

- Read lock on row accessed through cursor is *medium-duration*; held until cursor is moved
- Example

T_1: *fetch(t)* *update(t)*
T_2: *update(t)* *commit*

  - *Allowed* at READ COMMITTED, hence lost update possible
  - *Not allowed* at CURSOR STABILITY (since T_1 accesses *t* through a cursor)

35

CURSOR STABILITY

- Beware - CURSOR STABILITY does not solve all problems
  - Does not eliminate all lost updates: T_1 accesses *t* through cursor, T_2 (also at CURSOR STABILITY) accesses *t* directly (*e.g.*, through an index)

T_1: *fetch(t)* *update(t)* *commit*
T_2: *r(t)* *update(t)* *commit*

  - Can be prone to deadlock: Both T_1 and T_2 accesses *t* through cursor,

T_1: *fetch(t)* *request_update(t)*
T_2: *fetch(t)* *request_update(t)*

36

Update Locks

- Some DBMS provide update locks to alleviate deadlock problem
- A transaction that wants to read an item now and possibly update it later requests an update lock on the item (manual locking)
  - An update lock is a read lock that can be upgraded to a write lock
- Often used with updatable cursors

37

Update Locks

- An update lock conflicts with other update locks and with write locks, but not with read locks.

	Granted mode		
Requested mode	read	write	update
read		x	
write	x	x	x
update		x	x

38

Update Locks

- Schedule that causes a deadlock at CURSOR STABILITY:

  T_1: *fetch(t)* *request_update(t)*
  T_2: *fetch(t)* *request_update(t)*

- If both fetches had requested update locks, T_2's fetch would be made to wait until T_1 had completed, avoiding the deadlock

39

OPTIMISTIC READ COMMITTED

- Some systems provide a version of READ COMMITTED called OPTIMISTIC READ COMMITTED
  - Transactions get the same short-term read locks on tuples as at READ COMMITTED
  - If such a transaction, T, later tries to write a tuple it has previously read, if some other transaction has written that tuple and then committed, T is aborted

40

OPTIMISTIC READ COMMITTED

- Called optimistic because the transaction "optimistically" assumes that no transaction will write what it has read and hence it gives up its read lock
  - If that assumption is not true, it is aborted.
- Prevents lost updates, but can still lead to nonserializable schedules

41

Sometimes Good Things Happen

- For *some* applications, schedules are serializable and/or correct even though transactions are executing at isolation levels lower than SERIALIZABLE
- Designers must analyze applications to determine correctness

42

Correct Execution at READ UNCOMMITTED

- **Example** - Print_Alumni_Transcript(*s*)
  - Reads Transcript table and prints a transcript for a student, *s*, that has graduated. Since no concurrently executing transaction will be updating *s*'s record, the transaction executes correctly at READ UNCOMMITTED

43

Correct Execution READ COMMITTED

- **Example** - Register(*s*,*c*)
  - **Step 1**: Read table Requires to determine *c*'s prerequisites
  - **Step 2**: Read table Transcript to check that *s* has completed all of *c*'s prerequisites
  - **Step 3**: Read table Transcript to check that *s* does not enroll for more than 20 credits
  - **Step 4**: If there is room in *c*, update Class:

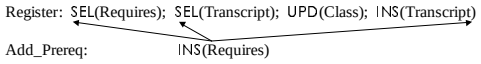
```
UPDATE Class C
SET C.Enrollment = C.Enrollment +1
WHERE C.CrsCode = :c AND
        C.Enrollment < C.MaxEnrollment
```

  - **Step 5**: Insert row for *s* in Transcript

44

Possible Interactions

- Register(*s*,*c*) executed at READ COMMITTED concurrently with a transaction that:
  - adds/deletes prerequisite for *c*
    - either Register sees new prerequisite or does not



  - However, application specification states that prerequisites added this semester do not apply to the registration this semester, but the following semester
    - Hence it does not matter if Register sees the new prerequisite

45

Possible Interactions

- Register(*s*,*c*) executed at READ COMMITTED concurrently with a transaction that
  - registers another student in *c*
    - Can a lost update occur and the Enrollment exceed MaxEnrollment?
    - No, since check and increment are done in a single (isolated) UPDATE over enrollment and lost update not possible
  - registers the same student in a different class
    - Each can execute step 3 and determine that the 20 credit maximum is not exceeded
    - Each can then complete and the maximum can be exceeded
    - Each does not see the phantom inserted in Transcript by the other
    - But this interaction might be ignored since it is highly unlikely

46

Possible Interactions

- These checks are necessary, but not sufficient to guarantee correct execution
  - Must look at interactions with other transactions
  - Schedules involving multiple transactions that might be non-serializable

47

Serializable, SERIALIZABLE, and Correct

- Serializable - Equivalent to a serial schedule
- SERIALIZABLE - An SQL isolation level defined in the standard
- Correct - Leaves the database consistent and a correct model of the real world

48

Serializable, SERIALIZABLE, and Correct

- If a schedule is serializable, it is correct
- If a schedule is produced at the SERIALIZABLE isolation level, it is serializable, and hence correct
- But as we have seen ...

49

Serializable, SERIALIZABLE, and Correct

- All schedules of an application run at an isolation level lower than SERIALIZABLE might be serializable
- A schedule can be correct, but not serializable
- One challenge of the application designer is to design applications that execute correctly at the lowest isolation level possible

50

Granular Locks

- Transactions access data at different levels of granularity
- Many DBMSs provide both fine and coarse granularity locks
  - DBMS attempts to automatically choose appropriate granularity
  - A particular application might be able to force a particular granularity

51

Granular Locks

- **Problem**: T_1 holds a (fine grained) lock on field F1 in record R1. T_2 requests a conflicting (coarse grained) lock on R1. How does the concurrency control detect the conflict since it sees F1 and R1 as different items?
- **Solution**: Organize locks hierarchically by containment and require that in order for a transaction to get a fine grained lock it must first get a coarse grained lock on the containing item
  - T_1 must first get a lock on R1 before getting a lock on F1. The conflict with T_2 is detected at R1

52

Intention Locking

- Performance improvement results if lock on parent is weak
- *Intention shared* (*IS*) lock: in order to get an *S* lock on an item, T must first get *IS* locks on *all* containing items (to root of hierarchy)
- *Intention exclusive* (*IX*) lock: in order to get an *X* lock on an item, T must first get *IX* locks on *all* containing items (to root of hierarchy)
- *Shared Intention Exclusive (SIX)*: Equivalent to an S lock and an IX lock on an item
- Intention lock indicates transaction's intention to acquire conventional lock on a contained item

53

Conflict Table

Requested mode \ Granted mode	*IS*	*IX*	*S*	*X*	*SIX*
IS				x	
IX			x	x	x
S		x		x	x
X	x	x	x	x	x
SIX		x	x	x	x

- **Example 1**: T_2 denied an *IX* lock (intends to update *some* contained items) since T_1 is reading *all* contained items
- **Example 2**: T_2 granted *IS* lock even though T_1 holds *IX* lock (since they may be accessing different subsets of contained items)

54

Preventing Phantoms With Granular Locks

- Preventing phantoms (SERIALIZABLE):
  - Lock entire table - this works
    - T_1 executes SEL(*P*) (where *P* is a predicate); obtains long-duration *S* lock on table
    - T_2 executes INS(*t*); requires long-duration X lock on table
    - Phantom prevented
  - Lock the predicate *P* - this works but entails too much overhead
  - Can granular locking be used?

55

Granular Locking and Phantoms

- Assume containment hierarchy is table/pages
- **Case 1**: no appropriate index for predicate *P*
  - T_1 does SEL(*P*) - obtains long-duration *S* lock on table
    - Since it must read every page to find rows satisfying *P*
  - T_2 requests INS(*t*) – obtains long-duration *IX* lock on table (lock conflict detected) and *X* lock on page into which *t* is to be inserted.
    - Hence (a potential) phantom is prevented
    - However other transaction can read parts of the table that are stored on pages other than the one on which *t* is stored

56

Granular Locking and Phantoms

- **Case 2**: index, I, exists on an attribute in *P*
  - T_1 obtains long-duration *IS* lock on table, uses I to locate pages containing rows satisfying *P*, and acquires long-duration *S* locks on them.
  - T_2 obtains long-duration *IX* lock on table (no conflict) and *X* lock on page, *p*, into which *t* is to be inserted.
    - **Problem**: Since *p* might not be locked by T_1, a phantom can result.

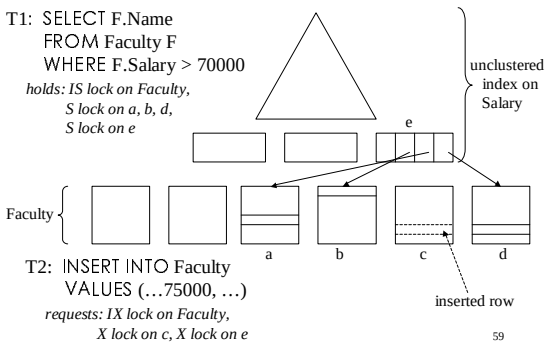
57

Index Locking

- **Solution**: lock pages of the index in addition
- Example: I is an unclustered B^+ tree.
  - T_1 obtains long-duration *IS* lock on table, long-duration *S* locks on the pages containing rows satisfying *P*, and *long-duration S locks on the leaf index pages containing entries satisfying P*
  - T_2 requests long-duration *IX* lock on table (granted), long-duration *X* locks on the page into which *t* is to be inserted (might be granted), and *long-duration X lock on the leaf index page into which the index entry for t will be stored* (lock conflict if *t* satisfies *P*)
    - The phantom is prevented.

58

Index Locking - Example



59

Index Locks, Predicate Locks, and Key-Range Locking

- If a **WHERE** clause refers to a predicate **name = mary** and if there is an index on **name**, then an index lock on the index entries for **name = mary** is like a predicate lock on that predicate
- If a **WHERE** clause refers to a predicate such as **50000< salary < 70000** and if there is an index on **salary**, then a **key-range** index lock can be used to get the equivalent of a predicate lock on the predicate **50000<salary<70000**

60

Key-Range Locking

- Instead of locking index pages, index entries at the leaf level are locked
  - Each such lock is interpreted as a lock on a range
- Suppose the domain of an attribute is **A…Z** and suppose at some time the entries in the index are
  C G P R X
- A lock on G is interpreted as a lock on the half-open interval
  [G P)
  - Which includes **G** but not **P**

61

Key-Range Locking (cont)

- Recall the index entries are: **C G P R X**
- Two special cases
  - A lock on **X** locks everything greater than **X**
  - A new lock must be provided for **[A C)**
- Then for example to lock the interval **H < K < Q**, we would lock **G** and **P**

62

Key-Range Locking (cont)

- Recall the index entries are: **C G P R X**
- To insert a new key, **J**, in the index
  - Lock **G** thus locking the interval **[G P)**
  - Insert **J** thus splitting the interval into **[G J)** **[J P)**
  - Lock **J** thus locking **[J P)**
  - Release the lock on **G**
- If a **SELECT** statement had a lock on **G** as part of a key-range, then the first step of the insert protocol could not be done
  - Thus phantoms are prevented and the key-range lock is equivalent to a predicate lock

63

Locking a B-Tree Index

- Many operations need to access an index structure concurrently
  - This would be a bottleneck if conventional two-phase locking mechanisms were used
- Because we understand the semantics of the index, we can develop a more efficient locking algorithm
  - The goal is to maintain isolation amount different operations that are concurrently accessing the index
  - The short term locks on the index structure are called **latches**
    - The long term locks on leaf entries we have been discussing are still obtained

64

Locking a B-Tree Index (cont)

- Read Locks
  - Obtain a read lock on the root, and work your way down the tree locking each entry as it is reached
  - When a new entry is locked, the lock on the previous entry (its parent) can be released
    - This operation will never revisit the parent
    - No write operation of a concurrent transaction can pass this operation as it goes down the tree
    - Called **lock coupling** or **crabbing**

65

Locking a B-Tree Index (cont)

- Write Locks
  - Obtain a write lock on the root, and work your way down the tree locking each entry as it is reached
  - When a new entry, n, is locked, if that entry is not full, the locks on all its parents can be released
    - An insert operation might have to go back up the tree, revisiting and perhaps splitting some nodes
    - Even if that occurs, because n is not full, it will not have to split n and therefore will not have to go further up the tree
    - Thus it can release locks further up in the tree.

66

Granular and Index Locking Summary

- Algorithm has property that a lock conflict that prevents phantoms will occur:
  - In the index, when an index is used
  - At the table level, when no index is used
- Even if there is no index, write operations need not get an *X* lock on whole table, only an *IX* lock, which allows more concurrency

67

UPDATE Statement

- An UPDATE can be treated as if it were a DELETE followed by an INSERT
  - If an index attribute is changed, the index entry for the tuple must be moved to a new position
  - The transaction must obtain write locks on both the old and new index pages

68

Lock Escalation

- To avoid acquiring many fine grain locks on a table, a DBMS can set a *lock escalation threshold*. If more than the threshold number of tuple (or page) locks are acquired, the DBMS automatically trades them in for a table lock but …
- Beware of deadlock

69

Granular Locking in an Object Database

- Containment hierarchy exists in two ways in an object database
  - Class contains object instances
  - Class contains subclasses (and hence object instances of subclasses)
- Intentions locking can be used over this hierarchy in the same way as in table/page/row hierarchy

70

Granular Locking Protocol for Object Databases

- Before obtaining a lock on an object instance, the system must obtain the appropriate intention locks on the object's class and all the ancestor classes
- Before obtaining a lock on a class, the system must get the appropriate intention locks on all ancestors of that class

71

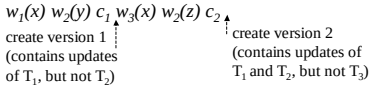
Performance Hints

- Use lowest correct isolation level
- Embedding constraints in schema *might* permit the use of an even lower level
  - Constraint violation due to interleaving detected at commit time (an optimistic approach)
- No user interaction after a lock has been acquired
- Use indexes and denormalization to support frequently executed transactions
- Avoid deadlocks by controlling the order in which locks are acquired

72

Multi-Version Controls

- *Version*: a snapshot of the database containing the updates of all and only committed transactions



- A multi-version DBMS maintains all versions created in the (recent) past
- Major goal of a multi-version DBMS: avoid the need for read locks

Read Consistency

- All DBMSs guarantee that statements are isolated:
  - Each statement sees state produced by the complete execution of other statements, but state might not be committed
- A multiversion control guarantees that *each* statement sees a committed state:
  - A statement is executed in a state whose value is a version
  - Referred to as *statement-level read consistency*
- A multiversion control *can* also guarantee that *all* statements of a transaction see the *same* committed state:
  - All statements of a transaction access the same version
  - Referred to as *transaction-level read consistency*

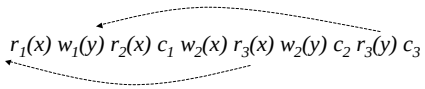
Read-Only Multi-Version Control

- Distinguishes in advance *read-only* (R/O) transactions from *read/write* (R/W) transactions.
  - R/W transactions use a (conventional) immediate-update, pessimistic control. Hence, transactions access the most current version of the database.
  - All the reads of a particular R/O transaction, T_{RO}, are satisfied using the most recent version that existed when T_{RO} requested its first read.

Read-Only Multi-Version Control

- Assuming R/W transactions are executed at SERIALIZABLE, all schedules are serializable
  - R/W transactions are serialized in commit order
  - Each R/O transaction is serialized after the transaction that created the version it read.
  - Equivalent serial order is not commit order
- All transactions see transaction-level read consistency

Example

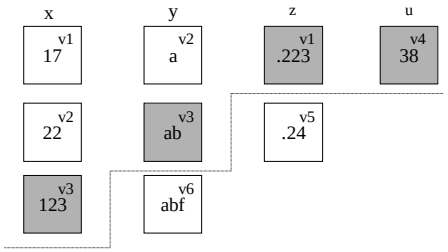


- T_1 and T_2 are read/write transactions, T_3 is read/only
- T_3 sees the version produced by T_1
- The equivalent serial order is T_1, T_3, T_2

Implementation

- DBMS maintains a *version counter* (VC)
  - Incremented each time a R/W transaction commits
- The new version of a data item created by a R/W transaction is tagged with the value of VC at the time the transaction commits
- When a R/O transaction makes its first read request, the value of VC becomes its counter value. Each request to read an item is satisfied by the version of the item having the largest version number less than or equal to the transaction's counter value

Multiversion Database



- Values read by a R/O transaction with counter value 4

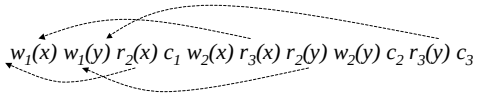
Read-Only Multi-Version Control

- R/O transactions do not use read locks.
  - They never wait
  - They never cause R/W transactions to wait

Read Consistency Multi-Version Control

- **R/O transactions**
  - Treated as before: get transaction-level read consistency
- **R/W transactions**
  - Write statements acquire long-duration write locks (delay other write statements)
  - Read statements use most recent (committed) version at time of read
    - Not delayed by write locks (since read locks are not requested).

Example



- T_1 and T_2 are R/W, T_3 is R/O
- T_3 uses v1
- T_2 takes the value of *x* from v0, *y* from v1
- There is *no* equivalent serial order

Read Consistency Multi-Version Control

- Satisfies the ANSI definition of the READ COMMITTED isolation level, but in addition ...
  - Provides transaction-level read consistency for R/O transactions
  - No read locks: reads do not wait for writes and writes do not wait for reads
- Version of READ COMMITTED supported by Oracle

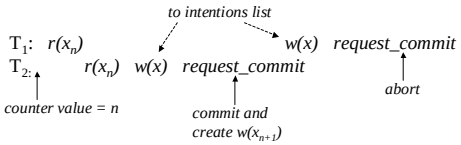
SNAPSHOT Isolation

- Does not distinguish between R/W and R/O transactions
- A transaction reads the most recent version that existed at the time of its first read request
  - Guarantees transaction-level read consistency
- The write sets of any two *concurrently executing* transactions must be disjoint
  - Two implementations of this specification
    - First Committer Wins
    - Locking implementation

First Committer Wins Implementation

- Writes use deferred-update (intentions list)
- *T* is allowed to commit only if no concurrent transaction
  - committed before *T* and
  - updated a data item that *T* also updated

First Committer Wins



- Control is optimistic:
  - It can be implemented without *any* locks
  - Deadlock not possible
  - Validation (write set intersection) is required for R/W transactions and abort is possible
  - Schedules might not be serializable

Locking Implementation of SNAPSHOT Isolation

- Immediate update pessimistic control
- Reads do not get any locks and execute as in the previous implementation
- A transaction T that wants to perform a write on some item must request a write lock
  - If the version number of that item is greater than that of T, T is aborted (first committer wins)
  - Otherwise, if another transaction has a write lock on that item, T waits until that transaction completes
    - If that transaction commits, T is aborted (first committer wins)
    - If that transaction aborts, T is given the write lock and allowed to write

Anomalies at SNAPSHOT Isolation

- Many anomalies are impossible:
  - Dirty read, dirty write, non-repeatable read, lost update
- However, schedules might not be serializable.
- Example:

T_1: *r(a:10) r(b:10) w(a:-5) commit*
T_2: *r(a:10) r(b:10) w(b:-5) commit*

  - Constraint $a+b \geq 0$ violated
  - Referred to as *write skew*

Phantoms at SNAPSHOT Isolation

Audit:
```
SELECT SUM (balance)
FROM Accounts
WHERE name = 'Mary'

SELECT totbal
FROM Depositors
WHERE name = 'Mary'
```

NewAccnt:
```
INSERT INTO Accounts
VALUES ('123','Mary',100)

UPDATE Depositors
  SET totbal = totbal + 100
  WHERE name = 'Mary'
```

- Both transactions commit.
- All reads of a transaction are satisfied from the same version.
- Hence *Audit* works correctly.

Phantoms at SNAPSHOT Isolation

- After a transaction executes SELECT, a concurrent transaction might insert a phantom
  - If the SELECT is repeated, the phantom will not be in the result set
  - Therefore, *apparently*, phantoms cannot occur at SNAPSHOT isolation
    - But …

Phantoms at SNAPSHOT Isolation

- Non-serializable schedules due to phantoms are possible
- **Example**: concurrent transactions each execute SEL(*P*) and then insert a row satisfying *P*
  - Neither sees the row inserted by the other.
  - The schedule is not serializable.
  - This would be considered a phantom if it occurred at REPEATABLE READ.
  - Can be considered write skew

91

Correct Execution at SNAPSHOT Isolation

- Many applications execute correctly at SNAPSHOT isolation, even though schedules are not serializable
- **Example**: reserving seats for a concert
  - Integrity constraint: a seat cannot be reserved by more than one person

92

Reserving Seats for a Concert

- A reservation transaction checks the status of two seats and then reserves one that is free
  - Schedule below is non-serializable, but is correct and preserves the constraint

T_1: *r(s1:Free) r(s2:Free) w(s1:Res) commit*
T_2: *r(s1:Free) r(s2:Free) w(s2:Res) commit*

  - Alternatively, if both transactions had tried to reserve the same seat

T_1: *r(s1:Free) r(s2:Free) w(s1:Res) abort*
T_2: *r(s1:Free) r(s2:Free) w(s1:Res) commit*

93

Not Serializable, but Correct

- Note that the first schedule on the previous slide has a write skew and is not serializable
  - Neverthless it is correct for this application!

94