Transactions and Concurrency

Transactions, Concurrency, Recovery

DBMSs provide access to valuable information resources in an environment that is:

- shared – concurrent access by multiple users
- unstable – potential for hardware/software failure

Each user should see the system as:

- unshared – their work is not inadvertantly affected by others
- stable – the data survives in the face of system failures

Goal: data integrity is maintained at all times.

Transactions, Concurrency, Recovery (cont)

Transaction processing

- techniques for describing "logical units of work" in applications in terms of underlying DBMS operations

Concurrency control

- techniques for ensuring that multiple concurrent transactions do not interfere with each other

Recovery mechanisms

- techniques to restore information to a consistent state, even after major hardware shutdowns/failures

Transaction Processing

A transaction is a "logical unit of work" in a DB application.

Examples:

- booking an airline or concert ticket
- transferring funds between bank accounts
- updating stock levels via point-of-sale terminal
- enrolling in a course or class

A transaction typically comprises multiple DBMS operations.

E.g. select ... update ... insert ... select ... insert ...

Transaction Processing (cont)

Transaction processing (TP) systems can be viewed as highly dynamic database applications.

Common characteristics of transaction-processing systems:

- multiple concurrent updates (10^2 .. 10^4 operations per second)
- real-time response requirement (preferably < 1 sec; max 5 secs)
- high availability (24×7) (especially for e.g. ecommerce systems)

TP benchmarks: important measure of DBMS performance.

Example Transaction

Problem: transfer funds between two accounts in same bank.

Possible implementation in PLpgSQL:

```
create or replace function
    transfer(src int, dest int, amount float) returns void
```

```
as $$
declare
    oldBalance float;
    newBalance float;
begin
    -- error checking
    select * from Accounts where id=src;
    if (not found) then
        raise exception 'Invalid Withdrawal Account';
    end if;
    select * from Accounts where id=dest;
    if (not found) then
        raise exception 'Invalid Deposit Account';
    end if;
...
```

Example Transaction (cont)

```
...
    -- action
(A) select balance into oldBalance
    from Accounts where id=src;
    if (oldBalance < amount) then
        raise exception 'Insufficient funds';
    end if;
    newBalance := oldBalance - amount;
```

```
(B) update Accounts
    set    balance := newBalance
    where  id = src;
    -- partial completion of transaction
(C) update Accounts
    set    balance := balance + amount
    where  id = dest;
    commit; -- redundant; function = transaction
end;
$$ language plpgsql;
```

Example Transaction (cont)

Consider two simultaneous transfers between accounts, e.g.

- T1 transfers $200 from account X to account Y
- T2 transfers $300 from account X to account Y

If the sequence of events is like:

```
T1:  ...  A  B  C  ...
T2:                ...  A  B  C  ...
```

everything works correctly, i.e.

- overall, account X is reduced by $500
- overall, account Y is increased by $500

Example Transaction (cont)

What if the sequence of events is like?

```
T1:  ...  A   B          C   ...
T2:     ...  A   B   C  ...
```

In terms of database operations, this is what happens:

- T1 gets balance from X ($A)
- T2 gets same balance from X ($A)
- T1 decrements balance in X ($A – 200)
- T2 decrements balance in X ($A – 300)

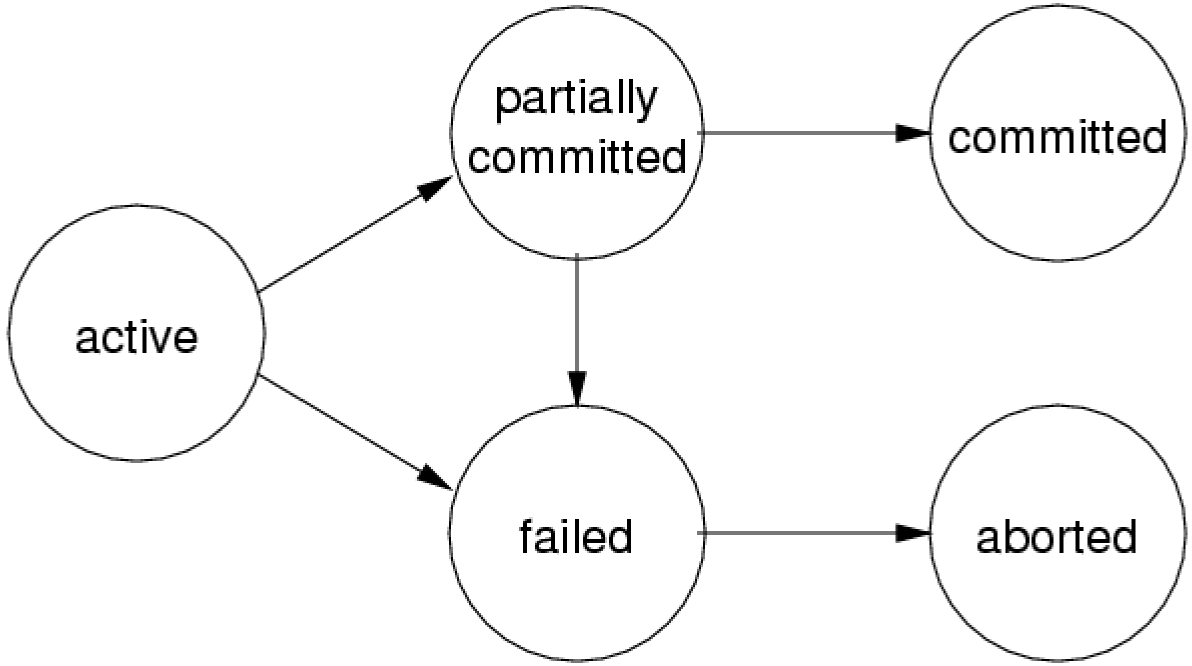
- T2 increments balance in Y ($B + 300)
- T1 increments balance in Y ($B + 300 + 200)

Final balance of Y is ok; final balance of X is wrong.

Transaction Concepts

A transaction must always terminate, either:

- successfully (**`COMMIT`**), with all changes preserved
- unsuccessfully (**`ABORT`**), with database unchanged



Transaction Concepts (cont)

To describe transaction effects, we consider:

- **`READ`** – transfer data from disk to memory
- **`WRITE`** – transfer data from memory to disk
- **`ABORT`** – terminate transaction, unsuccessfully
- **`COMMIT`** – terminate transaction, successfully

`SELECT` produces **`READ`** operations on the database.

`INSERT`, **`UPDATE`**, **`DELETE`** produce **`WRITE`**/**`READ`** operations.

Transaction Concepts (cont)

The **`READ`**, **`WRITE`**, **`ABORT`**, **`COMMIT`** operations:

- occur in the context of some transaction *T*
- involve manipulation of data items *X, Y, ...* (`READ` and `WRITE`)

The operations are typically denoted as:

$R_T(X)$	read item *X* in transaction *T*
$W_T(X)$	write item *X* in transaction *T*
A_T	abort transaction *T*
C_T	commit transaction *T*

Transaction Concepts (cont)

Execution of the above funds transfer example can be described as

```
T: READ(S);   READ(D);   -- S = source tuple, D = dest tuple
   READ(S);   S.bal := S.bal-amount;  WRITE(S)
   READ(D);   D.bal := D.bal+amount;  WRITE(D)
   COMMIT;
```

or simply as

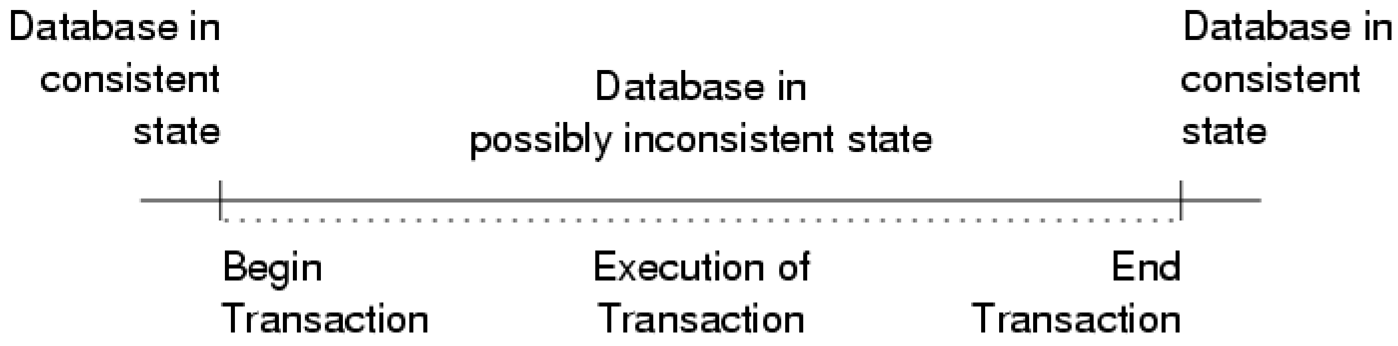
$R_T(S)$ $R_T(D)$ $R_T(S)$ $W_T(S)$ $R_T(D)$ $W_T(D)$ C_T

This is known as a schedule for the transaction.

Transaction Consistency

Transactions typically have intermediate states that are inconsistent.

However, states before and after transaction must be consistent.



ACID Properties

Data integrity is assured if transactions satisfy the following:

Atomicity

- Either all operations of transaction are reflected in database or none are.

Consistency

- Execution of a transaction in isolation preserves data consistency.

Isolation

- Each transaction is "unaware" of other transactions executing concurrently in the system.

Durability

- After a transaction completes successfully, its changes persist even after subsequent system failure.

ACID Properties (cont)

Atomicity is handled by the commit and rollback mechanisms.

- **commit** saves all changes and ends the transaction
- **rollback** *undoes* changes already made by the transaction

Durability is handled by implementing stable storage, via

- redundancy, to deal with hardware failures
- logging/checkpoint mechanisms, to recover state

Here, we consider primarily consistency and isolation.

Transaction Anomalies

If concurrent transactions access shared data objects, various anomalies can arise.

We give examples using the following two transactions:

```
T1: read(X)                T2: read(X)
    X := X + N                 X := X + M
    write(X)                   write(X)
    read(Y)
    Y := Y - N
    write(Y)
```

and initial DB state **X=100**, **Y=50**, **N=5**, **M=8**.

Serial Schedules

Serial execution means no overlap of transaction operations.

If **T1** and **T2** transactions are executed serially:

```
T1: R(X) W(X) R(Y) W(Y)
T2:                     R(X) W(X)
```

or

```
T1:           R(X) W(X) R(Y) W(Y)
T2: R(X) W(X)
```

the database is left in a consistent state.

Serial Schedules (cont)

The basic idea behind serial schedules:

- each transaction is correct
  (leaves the database in a consistent state if run to completion individually)
- the database starts in a consistent state
- the first transaction completes, leaving the DB consistent
- the next transaction completes, leaving the DB consistent

As would occur e.g. in a single-user database system.

Serial Schedules (cont)

For the first schedule in our example:

```
Database    T1                   T2
---------   -------------------  --------------
X    Y                X    Y               X
100  50               ?    ?               ?
            read(X)   100
            X:=X+N    105
105         write(X)
            read(Y)        50
            Y:=Y-N         45
     45     write(Y)
                                 read(X)   105
                                 X:=X+M    113
113                              write(X)
---------
113  45
```

Serial Schedules (cont)

For the second schedule in our example:

```
Database    T1                   T2
---------   -------------------  --------------
X    Y                X    Y               X
100  50               ?    ?               ?
                                 read(X)   100
                                 X:=X+M    108
108                              write(X)
            read(X)   108
            X:=X+N    113
113         write(X)
            read(Y)        50
            Y:=Y-N         45
     45     write(Y)
---------
113  45
```

Serial Schedules (cont)

Note that serial execution doesn't mean that each transaction will get the same results, regardless of the order.

Consider the following two transactions:

```
T1: select sum(salary)
    from Employee where dept='Sales'

T2: insert into Employee
    values (....,'Sales',...)
```

If we execute **T1** then **T2**, we get a smaller salary total than if we execute **T2** then **T1**.

In both cases, however, the salary total is consistent with the state of the database at the time the query is executed.

Concurrent Schedules

A serial execution of consistent transactions is always consistent.

If transactions execute under a concurrent (nonserial) schedule, the potential exists for conflict among their effects.

In the worst case, the effect of executing the transactions ...

- is to leave the database in an inconsistent state
- even though each transaction, by itself, *is* consistent

So why don't we observe such problems in real DBMSs? ...

- concurrency control mechanisms handle them (see later).

Valid Concurrent Transaction

Not all concurrent executions cause problems.

For example, the schedules

```
T1: R(X) W(X)           R(Y) W(Y)
T2:           R(X) W(X)
```

or

```
T1: R(X) W(X)      R(Y)      W(Y)
T2:           R(X)      W(X)
```

or ...

leave the database in a consistent state.

Lost Update Problem

Consider the following schedule where the transactions execute in parallel:

```
T1: R(X)      W(X) R(Y)      W(Y)
T2:      R(X)           W(X)
```

In this scenario:

- **T2** reads data (**X**) that **T1** is currently operating on

- then makes changes to **X** and overwrites **T1**'s result

This is called a Write-Read (WR) Conflict or dirty read.

The result: **T1**'s update to **X** is lost.

Lost Update Problem (cont)

Consider the states in the WR Conflict schedule:

```
Database    T1                     T2
---------   -------------------    --------------
X     Y                X     Y                X
100   50               ?     ?                ?
            read(X)    100
            X:=X+N     105
                                   read(X)    100
                                   X:=X+M     108
105         write(X)
            read(Y)          50
108                                write(X)
```

```
          Y:=Y-N         45
     45   write(Y)
---------
108  45
```

Temporary Update Problem

Consider the following schedule where one transaction fails:

```
T1: R(X) W(X) A
T2:             R(X) W(X)
```

Transaction **T1** aborts after writing **X**.

The abort *will* undo the changes to **X**, but where the undo occurs can affect the results.

Consider three places where undo might occur:

```
T1: R(X) W(X) A [1]      [2]      [3]
T2:                R(X)     W(X)
```

Temporary Update – Case 1

This scenario is ok. **`T1`**'s effects have been eliminated.

```
Database     T1                    T2
---------   -------------------   --------------
X    Y                X    Y                X
100  50               ?    ?                ?
            read(X)   100
            X:=X+N    105
105         write(X)
            abort
100         undo
                                  read(X)   100
                                  X:=X+M    108
108                               write(X)
---------
108  50
```

Temporary Update – Case 2

In this scenario, some of **`T1`**'s effects have been retained.

```
Database     T1                     T2
---------    -------------------    --------------
X    Y                X    Y                  X
100  50               ?    ?                  ?
             read(X)  100
             X:=X+N   105
105          write(X)
             abort
                                    read(X)   105
                                    X:=X+M    113
100          undo
113                                 write(X)
---------
113  50
```

Temporary Update – Case 3

In this scenario, **`T2`**'s effects have been lost, even after commit.

```
Database    T1                    T2
---------   -------------------   --------------
X    Y                X     Y               X
100  50               ?     ?               ?
            read(X)   100
            X:=X+N    105
105         write(X)
            abort
                                  read(X)   105
                                  X:=X+M    113
113                               write(X)
100         undo
---------
100  50
```

Valid Schedules

For ACID, we must ensure that schedules are:

- serializable

The effect of executing *n* concurrent transactions is the same as the effect of executing them serially in some order.

For assessing the correctness of concurrency control methods, need a test to check whether it produces serializable schedules.

- recoverable

  A failed transaction should not affect the ability to recover the system to a consistent state.

  This can be ensured if transactions commit only after all transactions whose changes they read commit.

Serializability

If a concurrent schedule for transactions T_1 ..T_n acts like a serial schedule for T_1 ..T_n, then consistency is guaranteed.

To determine this requires a notion of schedule equivalence.

Note: we are not attempting to determine equivalence of entire computations, simply of the interleaved sequences of read/write operations.

A serializable schedule is a concurrent schedule that produces a final state that is the same as that produced by some serial schedule.

There are two primary formulations of serializability:

- conflict serializibility (read/write operations occur in the "right" order)
- view serializibility (read operations *see* the correct version of data)

Conflict Serializability

Consider two transactions T_1 and T_2 acting on data item X.

Considering only read/write operations, the possibilities are:

T_1 first	**T_2 first**	Equiv?
$R_1(X)\ R_2(X)$	$R_2(X)\ R_1(X)$	yes

$R_1(X)\ W_2(X)$	$W_2(X)\ R_1(X)$	no
$W_1(X)\ R_2(X)$	$R_2(X)\ W_1(X)$	no
$W_1(X)\ W_2(X)$	$W_2(X)\ W_1(X)$	no

If T_1 and T_2 act on different data items, result is equivalent regardless of order.

Conflict Serializability (cont)

Two transactions have a potential conflict if

- they perform operations on the same data item
- at least one of the operations is a write operation

In such cases, the order of operations affects the result.

Conversely, if two operations in a schedule don't conflict,
we can swap their order without affecting the overall result.

This gives a basis for determining equivalence of schedules.

Conflict Serializability (cont)

If we can transform a schedule

- by swapping the orders of non-conflicting operations
- such that the result is a serial schedule

then we say that the schedule is conflict serializible.

If a concurrent schedule is equivalent to some (any) serial schedule, then we have a consistency guarantee.

Conflict Serializability (cont)

Example: transform a concurrent schedule to serial schedule

```
T1: R(A) W(A)      R(B)      W(B)
T2:           R(A)      W(A)      R(B) W(B)
swap
```

```
T1: R(A) W(A) R(B)           W(B)
T2:                R(A) W(A)      R(B) W(B)
swap
T1: R(A) W(A) R(B)      W(B)
T2:                R(A)      W(A) R(B) W(B)
swap
T1: R(A) W(A) R(B) W(B)
T2:                     R(A) W(A) R(B) W(B)
```

View Serializability

View Serializability is

- an alternative formulation of serializability
- that is less conservative than conflict serializability (CS)

  (some safe schedules that are view serializable are not conflict serializable)

As with CS, it is based on a notion of schedule equivalence

- a schedule is "safe" if *view equivalent* to a serial schedule

The idea: if all the read operations in two schedules ...

- always read the result of the same write operations
- then the schedules must produce the same result

View Serializability (cont)

Two schedules S and S' on T_1 .. T_n are view equivalent iff

- for each shared data item X
    - if T_j reads the initial value of X in S, then it also reads the initial value of X in S'
    - if T_j reads X in S and X was produced by T_k, then T_j must also read the value of X produced by T_k in S'
    - if T_j performs the final write of X in S, then it must also perform the final write of X in S'

To check serializibilty of S, find a serial schedule that is view equivalent to S

Testing Serializability

In practice, we don't test specific schedules for serializability.

However, in designing concurrency control schemes, we need a way of checking whether they produce "safe" schedules.

This is typically achieved by a demonstration that the scheme generates only serializable schedules, and we need a serializability test for this.

There is a simple and efficient test for conflict serializability;
there is a more complex test for view serializablity.

Both tests are based on notions of

- building a graph to represent transaction interactions
- testing properties of this graph (checking for cycles)

Testing Serializability (cont)

A precedence graph $G = (V,E)$ for a schedule S consists of

- a vertex in V for each transaction from T_1 .. T_n
- an edge in E for each pair T_j and T_k, such that
  - there is a pair of conflicting operations between T_j & T_k
  - the T_j operation occurs before the T_k operation

Note: the edge is directed from $T_j \rightarrow T_k$

Testing Serializability (cont)

If an edge $T_j \rightarrow T_k$ exists in the precedence graph

- then T_j must appear before T_k in any serial schedule

Implication: if the precedence graph has cycles, then S can't be serialized.

Thus, the serializability test is reduced to cycle-detection

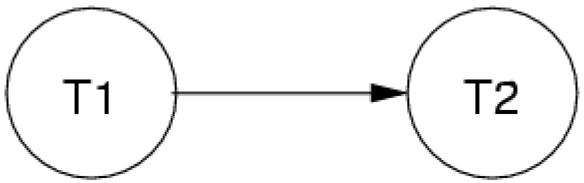
(and there are cycle-detection algorithms available in many algorithms textbooks)

Serializability Test Examples

Serializable schedule (with conflicting operations shown in red):

```
T1: R(A) W(A)      R(B)      W(B)
T2:           R(A)      W(A)      R(B) W(B)
```

Precedence graph for this schedule:



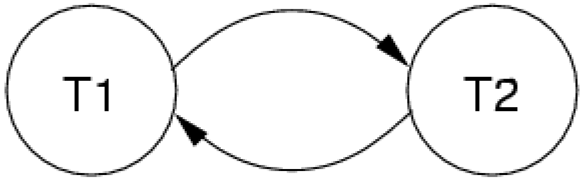
No cycles ⇒ serializable (as we already knew)

Serializability Test Examples (cont)

Consider this schedule:

```
T1: R(A)                W(A) R(B) W(B)
T2:      R(A) W(A) R(B)                W(B)
```

Precedence graph for this schedule:



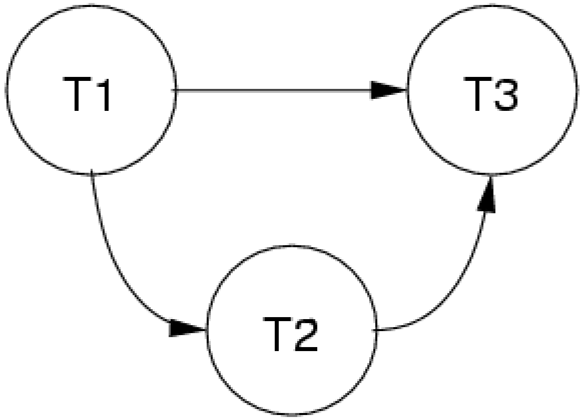
Has a cycle ⇒ not serializable

Serializability Test Examples (cont)

Consider this 3-transaction schedule:

```
T1: R(A)R(C)W(A)    W(C)
T2:             R(B)    R(A)    W(B)            W(A)
T3:                         R(C)    R(B)W(C)        W(B)
```

Precedence graph for this schedule:



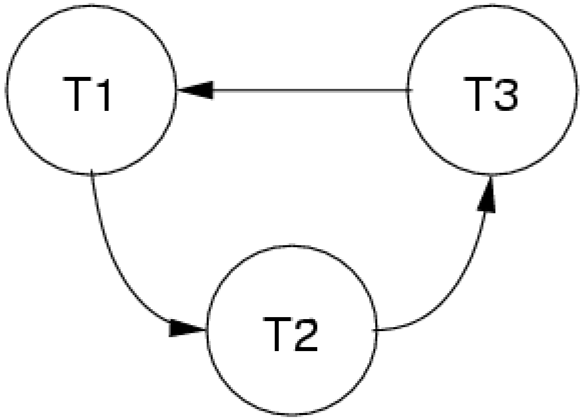
No cycles ⇒ serializable

Serializability Test Examples (cont)

Consider this 3-transaction schedule:

```
T1: R(A)                W(A)          R(C)    W(C)
T2:     R(B)    W(B)              R(A)    W(A)
T3:         R(C)    W(C)    R(B)                  W(B)
```

Precedence graph for this schedule:



Has a cycle ⇒ not serializable

Concurrency Control

Having serializability tests is useful theoretically, but they do not provide a practical tool for organising schedules.

Why not practical?

- the # possible schedules for *n* transactions is *O(n!)*
- the cost of testing for serializability via graphs is $O(n^2)$

What is required are methods

- that can be applied to each transaction individually
- which guarantee that any combination of transactions is serializable

---

Concurrency Control (cont)

Approaches to ensuring ACID transactions:

- lock-based

  Synchronise transaction execution via locks on some portion of the database.

- version-based

  Allow multiple consistent versions of the data to exist, and allow each transaction exclusive access to one version.

- timestamp-based

  Organise transaction execution in advance by assigning timestamps to operations.

- validation-based (optimistic concurrency control)

Exploit typical execution-sequence properties of transactions to determine safety dynamically.

Lock-based Concurrency Control

Synchronise access to shared data items via following rules:

- before reading *X*, get shared (read) lock on *X*
- before writing *X*, get exclusive (write) lock on *X*
- an attempt to get a shared lock on *X* is blocked if another transaction already has exclusive lock on *X*
- an attempt to get an exclusive lock on *X* is blocked if another transaction has any kind of lock on *X*

These rules alone do not guarantee serializability.

Two-Phase Locking

To guarantee serializability, we require an additional constraint on how locks are applied:

- no transaction can request a lock after it has released one of its locks

Each transaction is then structured as:

- growing phase where locks are acquired
- action phase where "real work" is done
- shrinking phase where locks are released

Problems with Locking

Appropriate locking can guarantee correctness.

However, it also introduces potential undesirable effects:

- deadlock

No transactions can proceed; each waiting on lock held by another.

- starvation

  One transaction is permanently "frozen out" of access to data.

- reduced performance

  Locking introduces delays while waiting for locks to be released.

Deadlock

Deadlock occurs when two transactions are waiting for a lock on an item held by the other.

Example:

```
T1                T2
--------------    ---------------
write_lock(X)
read(X)
                  write_lock(Y)
```

```
                    read(Y)
write_lock(Y)
waiting for Y       write_lock(X)
waiting for Y       waiting for X
```

Deadlock (cont)

Handling deadlock involves forcing a transaction to "back off".

- select a process to "back off"
    - choose on basis of how far transaction has progressed, # locks held, ...
- roll back the selected process
    - how far does this it need to be rolled back? (less roll-back is better)
- prevent starvation
    - need methods to ensure that same transaction isn't always chosen

---

Locking and Starvation

Starvation occurs when one transaction

- waits on a lock indefinitely
- while other transactions continue normally

Whether it occurs depends on the lock wait/release strategy.

Multiple locks ⇒ need to decide which to release first.

Solutions:

- implement a fair wait/release strategy (e.g. first-come-first-served)
- use deadlock prevention schemes, such as "wait-die"

Locking and Performance

Locking typically reduces concurrency ⇒ reduces throughput.

Granularity of locking can impact performance:

+ lock a small item ⇒ more of database accessible

+ lock a small item ⇒ quick update ⇒ quick lock release

– lock small items ⇒ more locks ⇒ more lock management

Granularity levels: field, row (tuple), table, whole database

Multiple lock-granularities give best scope for optimising performance.

Multi-version Concurrency Control

One approach to reducing the requirement for locks is to

- provide multiple (consistent) versions of the database
- give each transaction access to an "appropriate" version
  (i.e. a version that will mantain the serializability of the transaction)

This approach is called multi-version concurrency control (MVCC).

The primary difference between MVCC and standard locking models:

- read locks do not conflict with write locks, so that
- reading never blocks writing, and writing never blocks reading

---

MVCC and Transactions

Database systems using MVCC ensure

- statement-level read consistency
  (i.e. once an SQL SELECT statement starts, its view of the data is "frozen")
- readers do not wait for writers or other readers of the same data
- writers do not wait for readers of the same data
- writers only wait for other writers if they attempt to update identical rows in concurrent transactions

With this behaviour:

- a SELECT statement sees a consistent view of the database

- but it may not see the "current" view of the database

  E.g. *T1* does a select and then concurrent *T2* deletes some of *T1*'s selected tuples

---

PostgreSQL and MVCC

PostgreSQL uses MVCC to reduce locking requirements.

Consequences:

- several versions of each tuple may exist ⇒ uses more storage
- each transaction needs to check each tuple's visibility
- periodically, clean up "old" tuples (**`vacuum`**)

An "old" tuple is one that is no longer visible to any transaction.

Concurrency control is still needed (via implicit locking):

- amount of locking is determined by user-chosen isolation level
- the system then applies appropriate locking automatically

PostgreSQL and MVCC (cont)

A transaction sees a consistent view of the database, but it may not see the "current" view of the database.

E.g. *T1* does a select and then concurrent *T2* deletes some of *T1*'s selected tuples

This is not a problem unless the transactions communicate outside the database system.

For applications that require that every transaction accesses the current consistent version of the data, explicit locking must be used.

Concurrency Control in SQL

Transactions in SQL are specified by

- **`BEGIN`** ... start a transaction
- **`COMMIT`** ... successfully complete a transaction

- **ROLLBACK** ... undo changes made by transaction + abort

In PostgreSQL, other actions that cause rollback:

- **raise exception** during execution of a function
- returning null from a **before** trigger

Concurrency Control in SQL (cont)

More fine-grained control of "undo" via savepoints:

- **SAVEPOINT** ... marks point in transaction
- **ROLLBACK TO SAVEPOINT** ... undo changes, continue transaction

Example:

```
begin;
  insert into numbersTable values (1);
  savepoint my_savepoint;
  insert into numbersTable values (2);
  rollback to savepoint my_savepoint;
```

```
   insert into numbersTable values (3);
commit;
```

will insert 1 and 3 into the table, but not 2.

Concurrency Control in SQL (cont)

SQL standard defines four levels of transaction isolation.

- serializable – strongest isolation, most locking
- repeatable read
- read committed
- read uncommitted – weakest isolation, less locking

The weakest level allows dirty reads, phantom reads, etc.

PostgreSQL implements: repeatable–read = serializable, read–uncommitted = read–committed

---

Concurrency Control in SQL (cont)

Using the serializable isolation level, a **`select`**:

- sees only data committed before the transaction began
- never sees changes made by concurrent transactions

Using the serializable isolation level, an update fails:

- if it tries to modify an "active" data item

  (active = affected by some other transaction, either committed or uncommitted)

The transaction containing the failed update will rollback and re-start.

Concurrency Control in SQL (cont)

Explicit control of concurrent access is available, e.g.

Table-level locking: **`LOCK TABLE`**

- various kinds of shared/exclusive locks are available

  - **access share** allows others to read, and some writes
  - **exclusive** allows others to read, but not to write
  - **access exclusive** blocks all other access to table
- SQL commands automatically acquire appropriate locks
  - e.g. **`ALTER TABLE`** acquires an **access exclusive** lock

Row-level locking: **`SELECT FOR UPDATE`**, **`UPDATE`**, **`DELETE`**

- allows others to read, but blocks write on selected rows

All locks are released at end of transaction (no explicit unlock)

PostgreSQL, Transactions, Concurrency

For more details on PostgreSQL's handling of these:

- Chapter 12: Concurrency Control
- SQL commands: BEGIN, COMMIT, ROLLBACK, LOCK, etc.

Produced: 13 Sep 2020