Problems Caused by Failures

- Update all account balances at a bank branch.

  `Accounts(Anum, CId, BranchId, Balance)`

```
update Accounts
set Balance = Balance * 1.05
where BranchId = 12345
```

If the system crashes while processing this update, some, but not all, tuples with `BranchId = 12345` may have been updated.

Another Failure-Related Problem

- transfer money between accounts:

```
update Accounts
set Balance = Balance - 100
where Anum = 8888

update Accounts
set Balance = Balance + 100
where Anum = 9999
```

If the system fails between these updates, money may be withdrawn but not redeposited

Problems Caused by Concurrency

- Application 1:

```
update Accounts
set Balance = Balance - 100
where Anum = 8888

update Accounts
set Balance = Balance + 100
where Anum = 9999
```

- Application 2:

```
select Sum(Balance)
from Accounts
```

> If the applications run concurrently, the total balance returned to application 2 may be inaccurate.

Another Concurrency Problem

- Application 1:

```
select balance into :balance
from Accounts
where Anum = 8888
compute :newbalance using :balance
update Accounts
set Balance = :newbalance
where Anum = 8888
```

- Application 2: same as Application 1

> If the applications run concurrently, one of the updates may be "lost".

Transaction Properties

- Transactions are *durable*, *atomic* application-specified units of work.

  Atomic: indivisible, all-or-nothing.

  Durable: effects survive failures.

A tomic: a transaction occurs entirely, or not at all

C onsistent

I solated: a transaction's unfinished changes are not visible to others

D urable: once it is complete, a transaction's changes are permanent

---

Serializability (informal)

- Concurrent transactions must appear to have been executed sequentially, i.e., one at a time, in some order. If T_i and T_j are concurrent transactions, then either:
  - T_i will appear to precede T_j , meaning that T_j will "see" any updates made by T_i , and T_i will not see any updates made by T_j , or
  - T_i will appear to follow T_j , meaning that T_i will see T_j's updates and T_j will not see T_i's.

Serializability: An Example

- An interleaved execution of two transactions, T_1 and T_2:

$$H_a = w_1[x]\ r_2[x]\ w_1[y]\ r_2[y]$$

- An equivalent serial execution of T_1 and T_2:

$$H_b = w_1[x]\ w_1[y]\ r_2[x]\ r_2[y]$$

- An interleaved execution of T_1 and T_2 with no equivalent serial execution:

$$H_c = w_1[x]\ r_2[x]\ r_2[y]\ w_1[y]$$

H_a is serializable because it is equivalent to H_b , a serial schedule. H_c is not serializable.

Transactions and Histories

- Two operations conflict if:
  - they belong to different transactions
  - they operate on the same object
  - at least one of the operations is a write
- A transaction is a sequence of read and write operations.
- An *execution history* over a set of transactions $T_1 \ldots T_n$ is an interleaving of the the operations of $T_1 \ldots T_n$ in which the operation ordering imposed by each transaction is preserved.
- Two important assumptions:
  - transactions interact with each other only via database reads and writes
  - a database is a *fixed* set of *independent* objects

Serializability

- Two histories are *(conflict) equivalent* if
  - they are over the same set of transactions, and
  - the ordering of each pair of conflicting operations is the same in each history
- A history H is said to be *(conflict) serializable* if there exists some *serial* history H' that is (conflict) equivalent to H

Testing for Serializability

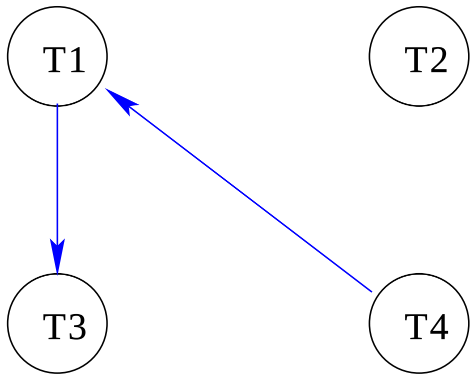
$$r_1[x]\ r_3[x]\ w_4[y]\ r_2[u]\ w_4[z]\ r_1[y]\ r_3[u]\ r_2[z]\ w_2[z]\ r_3[z]\ r_1[z]\ w_3[y]$$

Is this history serializable?

A history is serializable iff its serialization graph is acyclic.

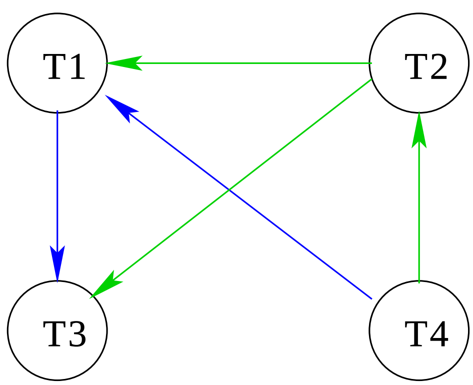
Serialization Graphs

$r_1[x]\ r_3[x]\ w_4[y]\ r_2[u]\ w_4[z]\ r_1[y]\ r_3[u]\ r_2[z]\ w_2[z]\ r_3[z]\ r_1[z]\ w_3[y]$



Serialization Graphs (cont'd)

$r_1[x]\ r_3[x]\ w_4[y]\ r_2[u]\ w_4[z]\ r_1[y]\ r_3[u]\ r_2[z]\ w_2[z]\ r_3[z]\ r_1[z]\ w_3[y]$



The history above is equivalent to

$w_4[y]\ w_4[z]\ r_2[u]\ r_2[z]\ w_2[z]\ r_1[x]\ r_1[y]\ r_1[z] r_3[x]\ r_3[u]\ r_3[z]\ w_3[y]$

That is, it is equivalent to executing T_4 followed by T_2 followed by T_1 followed by T_3.

Abort and Commit

- A transaction may terminate in one of two ways:
  - When a transaction *commits*, any updates it made become durable, and they become visible to other transactions. A commit is the “all” in “all-or-nothing” execution.
  - When a transaction *aborts*, any updates it may have made are undone (erased), as if the transaction never ran at all. An abort is the “nothing” in “all-or-nothing” execution.
- A transaction that has started but has not yet aborted or committed is said to be *active*.

Transactions in SQL

- A new transaction is begun when an application first executes an SQL command.
- Two SQL commands are available to terminate a transaction:
  - **commit work**: commits the transaction
  - **rollback work**: abort the transaction
- A new transaction begins with the application’s next SQL command after **commit work**or **rollback work**.

SQL Isolation Levels

- SQL allows the serializability guarantee to be relaxed, if necessary.
- For each transaction, it is possible to specify an *isolation level*.
- Four isolation levels are supported, with the highest being serializability:

  Level 0 (Read Uncommitted): transaction may see uncommitted updates

  Level 1 (Read Committed): transaction sees only committed changes, but non-repeatable reads are possible

  Level 2 (Repeatable Read): reads are repeatable, but "phantoms" are possible

  Level 3 (Serializability)

Non-Repeatable Reads

- Application 1:

```
update Employee
set Salary = Salary + 1000
where WorkDept = 'D11'
```

- Application 2:

```
select * from Employee
where WorkDept = 'D11'

select * from Employee
where Lastname like 'A%'
```

If there are employees in D11 with surnames that begin with "A", Application 2's queries may see them with different salaries.

Phantoms

- Application 1:

```
insert into Employee
values ('000123','Shel','Q','Jetstream','D11',
        '05/01/00',52000.00)
```

- Application 2:

```
select *
from Employee
where WorkDept = 'D11'

select *
from Employee
where Salary > 50000
```

Application 2's second query may see Sheldon Jetstream, even though its first query does not.

Implementing Transactions

- The implementation of transactions in a DBMS has two parts:

Concurrency Control: guarantees that the execution history has the desired properties (such as serializability)

Recovery Management: guarantees that committed transactions are durable (despite failures), and that aborted transactions have no effect on the database

Concurrency Control

- Serializability can be guaranteed by executing transactions serially, but it many environments this leads to poor performance.
- Typically, many transactions are in progress concurrently, and a concurrency control protocol is used to ensure that the resulting history is serializable.
- Many concurrency control protocols have been proposed, based on:
  - locking, or
  - timestamps, or
  - serialization graph analysis
- By far the most commonly implemented protocol is *strict two-phase locking*.
- The strict two-phase locking protocol can be relaxed, as necessary, to accommodate isolation levels below serializability.

Strict Two-Phase Locking

- The rules
  1. Before a transaction may read or write an object, it must have a lock on that object.
     - a *shared lock* is required to read an object
     - an *exclusive lock* is required to write an object
  2. Two or more transactions may not hold locks on the same object unless all hold shared locks.
  3. A transaction may not release any locks until it commits (or aborts).

If all transactions use strict two-phase locking, the execution history is guaranteed to be serializable.

Transaction Blocking

- Consider the following sequence of events:
  - T_1 acquires a shared lock on x and reads x
  - T_2 attempts to acquire an exclusive lock on x (so that it can write x)
- The two-phase locking rules prevent T_2 from acquiring its exlusive lock - this is called a *lock conflict*.
- Lock conflicts can be resolved in one of two ways:
  1. T_2 can be *blocked* - forced to wait until T_1 releases its lock
  2. T_1 can be *pre-empted* - forced to abort and give up its locks

Deadlocks

- transaction blocking can result in *deadlocks* For example:
  - T_1 reads object x
  - T_2 reads object y
  - T_2 attempts to write object x (it is blocked)
  - T_1 attempts to write object y (it is blocked)

A deadlock can be resolved only by forcing one of the transactions involved in the deadlock to abort.

Recovery Management

- recovery management means:
  - implementing voluntary or involuntary rollback of individual transactions
  - implementing recovery from system failures
  - *system failure* means:
    * the database server is halted abruptly
    * processing of in-progress SQL command(s) is halted abruptly
    * connections to application programs (clients) are broken.
    * contents of memory buffers are lost
    * database files are not damaged.

Failures and Transactions

- To ensure that transactions are atomic, every transaction that is active when a system failure occurs must either be
  - restarted after the failure from the point it which it left off, or
  - rolled back after the failure
- It is difficult to restart applications after a system failure, so the recovery manager does the following:
  - abort transactions that were active at the time of the failure
  - ensure that changes made by transactions that committed before the failure are not lost

Recovery Management

- Recovery management is usually accomplished using a *log*.
- A log is a read/append data structure located in persistent storage (it must survive the failure)
- When transactions are running, *log records* are appended to the log. Log records contain:

  UNDO information: old versions of objects that have been modified by a transaction. UNDO information can be used to undo database changes made by a transaction that aborts.

  REDO information: new versions of objects that have been modified by a transaction. REDO records can be used to redo the work done by a transaction that commits.

  BEGIN/COMMIT/ABORT: records are recorded whenever a transaction begins, commits, or aborts.

Write-Ahead Log Protocol

- A log record must always be written *before* the corresponding update is applied to the database

log head	$\rightarrow$	T_0,begin
(oldest part of the log)		T_0,X,99,100
		T_1,begin
		T_1,Y,199,200
		T_2,begin
		T_2,Z,51,50
		T_1,M,1000,10
		T_1,commit
		T_3,begin
		T_2,abort
		T_3,Y,200,50
		T_4,begin
(newest part of the log)		T_4,M,10,100
log tail	$\rightarrow$	T_3,commit

Recovery

- recovering from a system failure
  1. Scan the log from tail to head:
     - Create a list of committed transactions
     - Undo updates of active and aborted transactions
  2. Scan the log from head to tail:
     - Redo updates of committed transactions.
- rolling back a single transaction
  1. Scan the log from the tail to the transaction's BEGIN record.
     - Undo the transaction's updates.