

# ISYS2120: Data & Information Management

## Week 10: Transactions

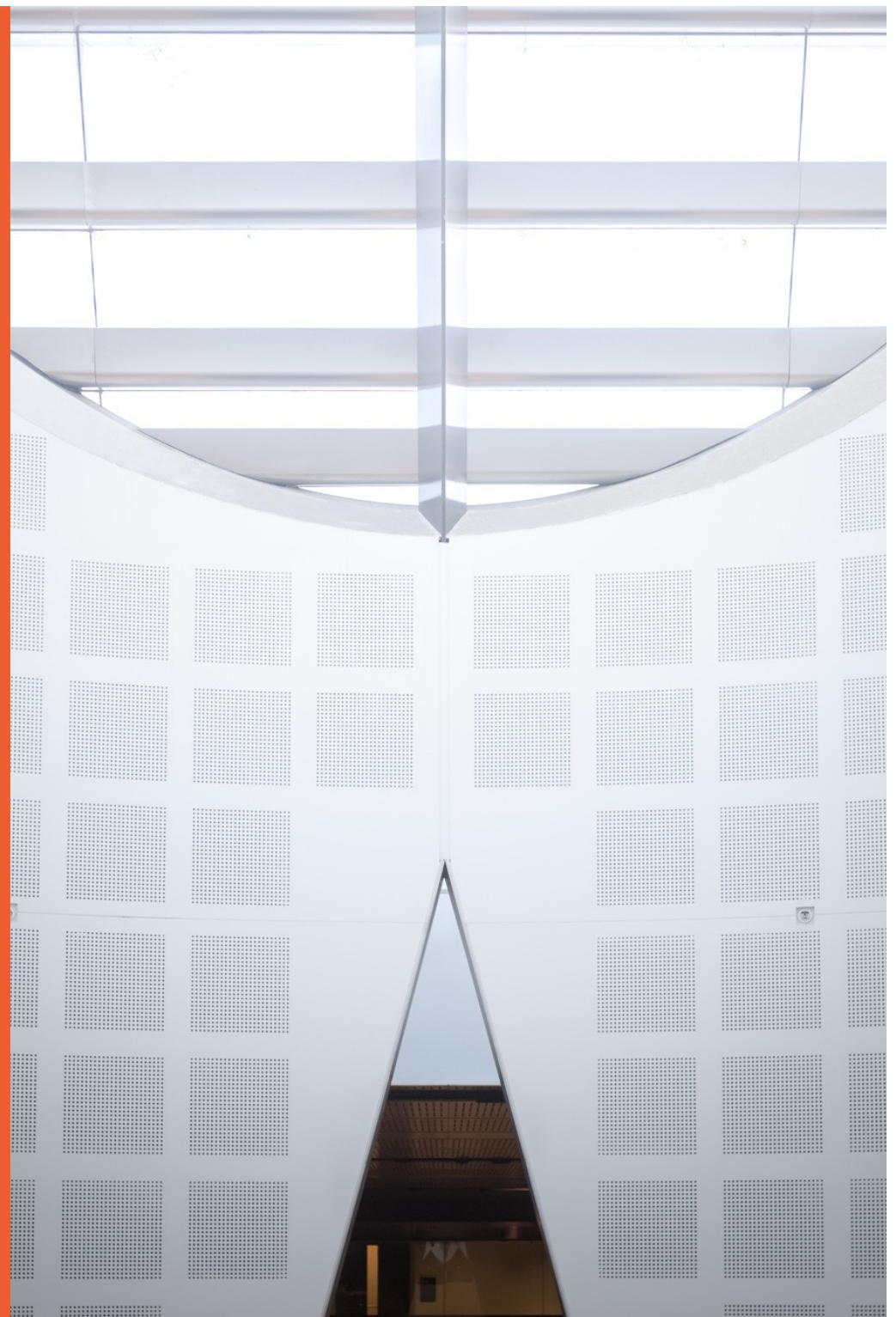
**Alan Fekete**

Cf. Kifer/Bernstein/Lewis – Chapter 18

Ramakrishnan/Gehrke – Chapter 16

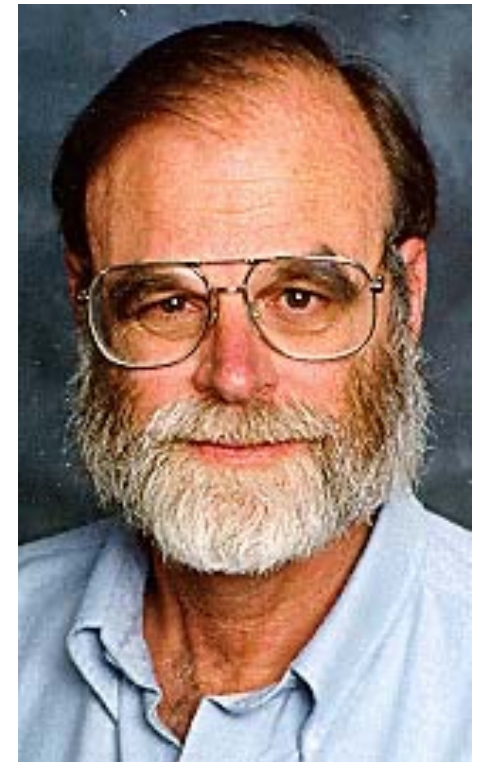
Silberschatz/Korth/Sudarshan – Chapter 17

Ullman/Widom – Chapter 6.6



# Famous DB Researchers: Jim Gray

- One of the most influential database researchers and software designers
  - first Ph.D. from CS at UC Berkeley (1969)
  - worked for  
IBM (at System R)  
Tandem-Computers, DEC, ...,  
Microsoft Research
- Turing Award (1998)
  - "for seminal contributions to database and ***transaction processing research*** and technical leadership in system implementation."



Jim Gray (1944-2007)

# Overview

- Transactions and ACID properties
- Brief overview of Implementation Techniques
- Weak isolation issues

# ACID

- **A**tomic
  - State shows either all the effects of txn, or none of them
- **C**onsistent
  - Txn moves from a state where integrity holds, to another where integrity holds
- **I**solated
  - Effect of txns is the same as txns running one after another (ie looks like batch mode)
- **D**urable
  - Once a txn has committed, its effects remain in the database

# Definition

- A transaction is a collection of one or more operations on one or more databases, which reflects a single real-world transition
  - In the real world, this happened (completely) or it didn't happen at all (**Atomicity**)
  - Once it has happened, it isn't forgotten (**Durability**)
- Commerce examples
  - Transfer money between accounts
  - Purchase a group of products
- Student record system
  - Register for a class (either waitlist or allocated)

# Coding a transaction

- Typically a computer-based system doing OLTP has a collection of *application programs*
- Each program is written in a high-level language, which calls DBMS to perform individual SQL statements
  - Either through embedded SQL converted by preprocessor
  - Or through Call Level Interface where application constructs appropriate string and passes it to DBMS

# Why write programs?

- Why not just write a SQL statement to express “what you want”?
- An individual SQL statement can't do enough
  - It can't update multiple tables
  - It can't perform complicated logic (conditionals, looping, etc)

# COMMIT

- As application program is executing, it is “in a transaction”
- Program can execute COMMIT
  - SQL command to finish the transaction successfully
  - The next SQL statement will automatically start a new transaction



# Atomicity

- Two possible outcomes for a transaction
  - It *commits*: all the changes are made
  - It *aborts*: no changes are made
- That is, transaction's activities are **all** or **nothing**
  - Furthermore, once an outcome has been reached, it doesn't change

# ROLLBACK

- If the application code gets to a place where it can't complete the transaction successfully, it can execute ROLLBACK
- This causes the system to “abort” the transaction
  - The database returns to the state without any of the previous changes made by activity of the transaction

# API for Transactions

- A transaction ends by:
  - **COMMIT** *requests* to commit current transaction.
  - **ROLLBACK** causes current transaction to abort - always satisfied.
- The commit command is a request
  - The system might commit the transaction, or it might abort it for one of several reasons.

# Reasons for Rollback

- User changes their mind (“ctl-C”/cancel)
- Explicit in program, when application code finds a problem
  - e.g. when qty on hand < qty being sold
- System-initiated abort
  - System crash
  - Housekeeping
    - e.g. due to timeouts

# Autocommit

- Application programmer can set the connection to autocommit
  - every single SQL statement is a transaction in itself
  - no need for explicit commit/rollback code in application
  - but lose the capacity to group several table activities into a single real-world transition
- Be careful: some platforms make `autocommit=true` as the default!

# Transactions in Python DB-API

- Python DB-APIs Connection Class:
  - Provides **commit()** and **rollback()** methods
  - allows to set SQL Isolation levels (see later)
  - allows to set **autocommit** mode on/off

# Python DB-API

- By default, DB-API connections are in explicit mode
  - a new transaction starts implicitly with first SQL statement
  - must be explicitly ended by call to **commit()** or **rollback()** !
  - if no call to **commit()**, all changes get lost once connection closes !
  - no auto-commit, no beginTransaction statement
- Hence: need to call commit() after data manipulations

```
import pg8000
```

```
def bookFlight ( flight_num, flight_date, seat_no ) :
```

```
    try: # connect to the database
```

```
        conn = pg8000.connect(database="postgres", user="X", password="secret")
```

```
        # execute the SQL statements within the transaction
```

```
        curs = conn.cursor()
```

```
        stmt = """SELECT occupied
```

```
                FROM Flight
```

```
                WHERE flightNum=%s AND flightDate=%s AND seat=%s"""
```

```
        curs.execute(stmt, (flight_num, flight_date, seat_no))
```

```
        result = curs.fetchone()
```

```
        if ( result is None ) or ( result[0] = False ) :
```

```
            update_stmt = """UPDATE Flight SET occupied=TRUE
```

```
                            WHERE flightNum=%s AND flightDate=%s AND seat=%s"""
```

```
            curs.execute(update_stmt, (flight_num, flight_date, seat_no))
```

```
        # COMMIT the transaction
```

```
        conn.commit()
```

```
    except:
```

```
        # error handling
```

```
        ...
```

```
        conn.rollback()
```

```
    finally:
```

```
        curs.close()
```

```
        conn.close()
```



# Integrity

- A real world state is reflected by collections of values in the tables of the DBMS
- But not every collection of values in a table makes sense in the real world
- The state of the tables is restricted by **integrity constraints**
- e.g. account number is unique
- e.g. stock amount can't be negative

# Integrity (ctd)

- Many constraints are explicitly declared in the schema
  - So the DBMS will enforce them
  - Especially: primary key (some column's values are non null, and different in every row)
  - And referential integrity: value of foreign key column is actually found in another “referenced” table
- Some constraints are not declared
  - They are business rules that are supposed to hold

# Dynamic Integrity Constraints

- Some constraints restrict allowable state transitions
  - A transaction might transform the database from one consistent state to another, but the transition might not be permissible
  - **Example:** Students can only progress from Junior via Intermediate to the Senior year, but can never be degraded.
- Dynamic constraints cannot be checked by examining the database state

# Transaction **C**onsistency

- A **transaction** is **consistent** if, assuming the database is in a consistent state initially, when the transaction completes:
  - All static integrity constraints are satisfied (but constraints might be violated in intermediate states)
    - Can be checked by examining snapshot of database
  - New state satisfies specifications of transaction
    - Cannot be checked from database snapshot
  - No dynamic constraints have been violated
    - Cannot be checked from database snapshot
- This is an obligation on the programmer
  - Usually the organization has a testing/checking and sign-off mechanism before an application program is allowed to get installed in the production system

# Checking Integrity Constraints

- Automatic: Embed constraint in schema.
  - With CHECK, ASSERTION, TRIGGER
  - Increases confidence in correctness and decreases maintenance costs
  - Not always desirable since unnecessary checking (overhead) might result
    - Deposit transaction modifies *balance* but cannot violate constraint  $balance \geq 0$
- Manual: Perform check in application code.
  - Only necessary checks are performed
  - Scatters references to constraint throughout application
  - Difficult to maintain as transactions are modified/added

# Integrity Constraints in Transactions

- When do we check integrity constraints?
- Integrity constraints may be declared:

- **NOT DEFERRABLE**

The default. It means that every time a database modification occurs, the constraint is checked immediately afterwards.

- **DEFERRABLE**

Gives the option to wait until a transaction is complete before checking the constraint.

```

CREATE TABLE UnitOfStudy (
    uos_code          VARCHAR(8) ,
    title             VARCHAR(220) ,
    lecturer          INTEGER,
    credit_points     INTEGER,
    CONSTRAINT UnitOfStudy_PK PRIMARY KEY
(uos_code) ,
    CONSTRAINT UnitOfStudy_FK FOREIGN KEY
(lecturer)
    REFERENCES Lecturer DEFERRABLE INITIALLY
DEFERRED
);

BEGIN TRANSACTION;
    SET CONSTRAINTS UnitOfStudy_FK DEFERRED;
    INSERT INTO UnitOfStudy
VALUES ('info1000' , 'Intro to...' , 42 , 6) ;
    INSERT INTO Lecturer VALUES (42 , 'Steve McQueen' ,
...);

COMMIT;

```

# System obligations

- Provided the application programs have been written properly,
- Then the DBMS is supposed to make sure that the state of the data in the DBMS reflects the real world accurately, as affected by all the committed transactions



# Local to global reasoning

- Organization checks each application program as a separate task
  - Each application program running on its own moves from state where integrity constraints are valid to another state where they are valid
- System makes sure there are no nasty interactions
- So the final state of the data will satisfy all the integrity constraints

# Example - Tables

- System for managing inventory
- InStore(prodID, storeID, qty)
- Product(prodID, desc, mnfr, ..., warehouseQty)
- Order(orderNo, prodID, qty, rcvd, ....)
  - Rows never deleted!
  - Until goods received, rcvd is null
- Also Store, Staff, etc etc

# Example - Constraints

- Primary keys
  - InStore: (prodID, storeID)
  - Product: prodID
  - Order: orderId
  - etc
- Foreign keys
  - Instore.prodID references Product.prodID
  - etc

# Example - Constraints

- Data values
  - `Instore.qty`  $\geq 0$
  - `Order.rcvd`  $\leq$  `current_date` or `Order.rcvd` is null
- Business rules
  - for each `p`, (Sum of `qty` for product `p` among all stores and warehouse)  $\geq 50$
  - for each `p`, (Sum of `qty` for product `p` among all stores and warehouse)  $\geq 70$  or there is an outstanding order of product `p`

# Example - transactions

- MakeSale(store, product, qty)
- AcceptReturn(store, product, qty)
- RcvOrder(order)
- Restock(store, product, qty)
  - // move from warehouse to store
- ClearOut(store, product)
  - // move all held from store to warehouse
- Transfer(from, to, product, qty)
  - // move goods between stores

# Example - ClearOut

- Validate Input (appropriate product, store)
- ```
SELECT qty INTO :tmp
FROM InStore
WHERE storeID = :store AND prodID = :product
```
- ```
UPDATE Product
SET warehouseQty = warehouseQty + :tmp
WHERE prodID = :product
```
- ```
UPDATE InStore
SET qty = 0
WHERE storeID = :store AND prodID = :product
```
- COMMIT

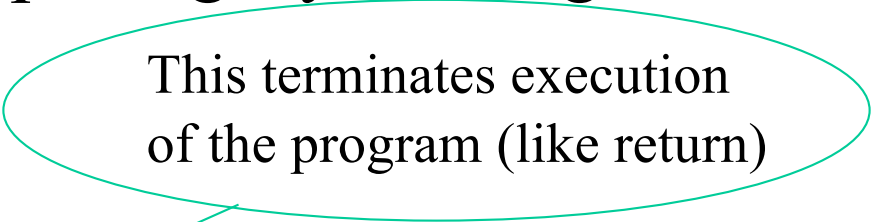
This is one way to write the application; other algorithms are also possible

# Example - Restock

- Input validation
  - Valid product, store, qty
  - Amount of product in warehouse  $\geq$  qty
- UPDATE Product
  - SET warehouseQty = warehouseQty - :qty
  - WHERE prodID = :product
- **If** no record yet for product in store
  - INSERT INTO InStore (:product, :store, :qty)
- **Else**, UPDATE InStore
  - SET qty = qty + :qty
  - WHERE prodID = :product and storeID = :store
- COMMIT

# Example - Consistency

- How to write the app to keep integrity holding?
- MakeSale logic:
  - Reduce Instore.qty
  - Calculate sum over all stores and warehouse
  - If  $\text{sum} < 50$ , then ROLLBACK // Sale fails
  - If  $\text{sum} < 70$ , check for order of this product where date is null
    - If none found, insert new order for say 25
  - COMMIT



This terminates execution of the program (like return)



# Example - Consistency

- We don't need any fancy logic for checking the business rules in Restock, ClearOut, Transfer
  - Because sum of qty not changed; presence of order not changed
    - *provided integrity holds before txn, it will still hold afterwards*
- We don't need fancy logic to check business rules in AcceptReturn
  - *why?*
- Is checking logic needed for RcvOrder?

# Threats to data integrity

- Need for application rollback
  - System crash
  - Concurrent activity
- 
- The system has mechanisms to handle these

# Application rollback

- A transaction may have made changes to the data before discovering that these aren't appropriate
  - the data is in state where integrity constraints are false
  - Application executes ROLLBACK
- System must somehow return to earlier state
  - Where integrity constraints hold
- So the aborted transaction *has no effect* at all

# Example

- While running MakeSale, app changes InStore to reduce qty, then checks new sum
- If the new sum is below 50, txn aborts
- System must change InStore to restore previous value of qty
  - Somewhere, system must remember what the previous value was!

# System crash

- At time of crash, an application program may be part-way through (and the data may not meet integrity constraints)
- Also, buffering can cause problems
  - Note that system crash loses all buffered data, restart has only disk state
  - Effects of a committed txn may be only in buffer, not yet recorded in disk state
  - Lack of coordination between flushes of different buffered pages, so even if current state satisfies constraints, the disk state may not

# Example

- Suppose crash occurs after
  - MakeSale has reduced InStore.qty
  - found that new sum is 65
  - found there is no unfilled order
  - // but before it has inserted new order
- At time of crash, integrity constraint did not hold
- Restart process must clean this up (effectively aborting the txn that was in progress when the crash happened)

# Restart and Durability

- After a crash, system must be restored to a state which includes all the effects of committed transactions
  - And none of the effects of aborted transactions, or those which were active at the time of the crash
- Each item should have the value placed there by the last committed transaction that modified the item

# Concurrency

- When operations of concurrent **threads are interleaved**, the effect on shared state can be unexpected
- Well known issue in operating systems, thread programming
  - see OS textbooks on critical section
  - Java use of synchronized keyword



# Famous anomalies

- Dirty data
  - One task T reads data written by T' while T' is running, then T' aborts (so its data was not appropriate)
- Lost update
  - Two tasks T and T' both modify the same data
  - T and T' both commit
  - Final state shows effects of only T, but not of T'
- Inconsistent read
  - One task T sees some but not all changes made by T'
  - The values observed may not satisfy integrity constraints
  - This was not considered by the programmer, so code moves into absurd path

# Example – Dirty data

|     |     |     |
|-----|-----|-----|
| p1  | s1  | 25  |
| p1  | s2  | 70  |
| p2  | s1  | 60  |
| etc | etc | etc |

|     |     |     |
|-----|-----|-----|
| p1  | etc | 10  |
| p2  | etc | 44  |
| etc | etc | etc |

Initial state of InStore, Product

|     |     |     |
|-----|-----|-----|
| p1  | s1  | 25  |
| p1  | s2  | 5   |
| p2  | s1  | 60  |
| etc | etc | etc |

|     |     |     |
|-----|-----|-----|
| p1  | etc | 10  |
| p2  | etc | 44  |
| etc | etc | etc |

Final state of InStore, Product

- (Thread 1)
  - AcceptReturn(p1,s1,50)
  - Update row 1: 25 -> 75
  - 
  - 
  - 
  - 
  - 
  - 
  - Abort
  - // rollback row 1 to 25
  -
- (Thread 2)
- MakeSale(p1,s2,65)
  - update row 2: 70->5
  - find sum: 90
  - // no need to insert
  - // row in Order
  - COMMIT

Integrity constraint is false:  
Sum for p1 is only 40!

# Example – Lost update

- ClearOut(p1,s1) **AcceptReturn(p1,s1,60)**
- Query InStore; qty is 25
- Add 25 to warehouseQty: 40->65
- **Update row 1: 25->85**
- Update row 1, setting it to 0
- COMMIT
- **COMMIT**

60 returned p1's have vanished from system; total is still 115

|     |     |     |
|-----|-----|-----|
| p1  | s1  | 25  |
| p1  | s2  | 50  |
| p2  | s1  | 45  |
| etc | etc | etc |

|     |     |     |
|-----|-----|-----|
| p1  | etc | 40  |
| p2  | etc | 55  |
| etc | etc | etc |

Initial state of InStore, Product

|     |     |     |
|-----|-----|-----|
| p1  | s1  | 0   |
| p1  | s2  | 50  |
| p2  | s1  | 45  |
| etc | etc | etc |

|     |     |     |
|-----|-----|-----|
| p1  | etc | 65  |
| p2  | etc | 55  |
| etc | etc | etc |

Final state of InStore, Product

# Example – Inconsistent read

- ClearOut(p1,s1)      **MakeSale(p1,s2,60)**
- Query InStore: qty is 30
- Add 30 to warehouseQty: 10->40
- **update row 2: 65->5**
- **find sum: 75**
- **// no need to insert**
- **// row in Order**
- Update row 1, setting it to 0
- COMMIT
- **COMMIT**

Integrity constraint is false:  
Sum for p1 is only 45!

|     |     |     |
|-----|-----|-----|
| p1  | s1  | 30  |
| p1  | s2  | 65  |
| p2  | s1  | 60  |
| etc | etc | etc |

|     |     |     |
|-----|-----|-----|
| p1  | etc | 10  |
| p2  | etc | 44  |
| etc | etc | etc |

Initial state of InStore, Product

|     |     |     |
|-----|-----|-----|
| p1  | s1  | 0   |
| p1  | s2  | 5   |
| p2  | s1  | 60  |
| etc | etc | etc |

|     |     |     |
|-----|-----|-----|
| p1  | etc | 40  |
| p2  | etc | 44  |
| etc | etc | etc |

Final state of InStore, Product

# Serializability

- To make isolation precise, we say that an execution is serializable when
- There exists some serial (ie batch, no overlap at all) execution of the same transactions which has the same final state
  - Hopefully, the real execution runs faster than the serial one!
- NB: different serial txn orders may behave differently; we ask that *some* serial order produces the given state
  - Other serial orders may give different final states

# Example – Serializable execution

- ClearOut(p1,s1)      MakeSale(p1,s2,20)
- Query InStore: qty is 30
- update row 2: 45->25
- find sum: 65
- no order for p1 yet
- Add 30 to WarehouseQty: 10->40
- Update row 1, setting it to 0
- COMMIT
- Insert order for p1
- COMMIT

Execution is like serial  
MakeSale; ClearOut

|     |     |     |
|-----|-----|-----|
| p1  | s1  | 30  |
| p1  | s2  | 45  |
| p2  | s1  | 60  |
| etc | etc | etc |

|     |     |     |
|-----|-----|-----|
| p1  | etc | 10  |
| p2  | etc | 44  |
| etc | etc | etc |

Order: empty

Initial state of InStore, Product, Order

|     |     |     |
|-----|-----|-----|
| p1  | s1  | 0   |
| p1  | s2  | 25  |
| p2  | s1  | 60  |
| etc | etc | etc |

|     |     |     |
|-----|-----|-----|
| p1  | etc | 40  |
| p2  | etc | 44  |
| etc | etc | etc |

|    |    |      |     |
|----|----|------|-----|
| p1 | 25 | Null | etc |
|----|----|------|-----|

46

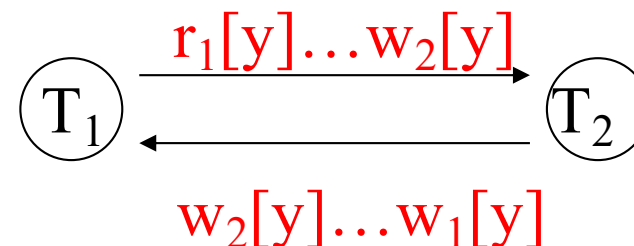
Final state of InStore, Product, Order

# Serializability Theory

- There is a beautiful mathematical theory, based on formal languages
  - Model an execution as a sequence of operations on data items
    - eg  $r_1[x] \ w_1[x] \ r_2[y] \ r_2[x] \ c_1 \ c_2$
  - Serializability of an execution can be defined by equivalence to a rearranged sequence (“view serializability”)
- There is a nice sufficient condition (ie a conservative approximation) called **conflict serializable**, which can be efficiently tested
  - Draw a **precedes graph** whose nodes are the transactions
  - Edge from  $T_i$  to  $T_j$  when  $T_i$  accesses  $x$ , then later  $T_j$  accesses  $x$ , and the accesses conflict (not both reads)
  - The execution is conflict serializable iff the graph is acyclic

# Example – Lost update

- ClearOut(p1,s1)
- AcceptReturn(p1,s1,60)
- Query InStore; qty is 25
- Add 25 to warehouseQty: 40->65
- Update row 1: 25->85
- Update row 1, setting it to 0
- COMMIT
- COMMIT
- Items: Product(p1) as x, Instore(p1,s1) as y
- Execution is
  - $r_1[y] \ r_1[x] \ w_1[x] \ r_2[y]$   
 $w_2[y] \ w_1[y] \ c_1 \ c_2$
- Precedes Graph





# ACID

- **A**tomic
  - State shows either all the effects of txn, or none of them
- **C**onsistent
  - Txn moves from a state where integrity holds, to another where integrity holds
- **I**solated
  - Effect of txns is the same as txns running one after another (ie looks like batch mode)
- **D**urable
  - Once a txn has committed, its effects remain in the database

# Big Picture

- If programmer writes applications so each txn is consistent
- And DBMS provides atomic, isolated, durable execution
  - i.e. actual execution has same effect as some serial execution of those txns that committed (but not those that aborted)
- Then the final state will satisfy all the integrity constraints

NB true even though system does not know all integrity constraints!

# Vital skills

- In coding an application, recognize which SQL commands need to be placed in the same transaction
  - Because problems would occur if some but not all are performed, or if other code intervenes between their execution
- Make sure there are appropriate settings (SERIALIZABLE, autocommit off)

# Overview

- Transactions and ACID properties
- Brief overview of Implementation Techniques
- Weak isolation issues

# Logging

- The log is an append-only collection of entries, showing all the changes to data that happened, in order as they happened
- e.g. when T1 changes qty in row 3 from 15 to 75, this fact is recorded as a log entry
- Log also shows when txns start/commit/abort
- Log must be frequently flushed to persistent store (eg disk) in particular, before commit <sup>53</sup>

# Using the log

- To abort/rollback txn T
  - Follow chain of T' s log entries, *backwards*
  - For each entry, restore data to old value, and produce new log record showing the restoration
  - Produce log record for “abort T”
- After a crash, the goal is to get to a state of the database which has the effects of those transactions that committed before the crash
  - it does not show effects of transactions that aborted or were active at the time of the crash
  - To reach this state, follow the log *forward*, replaying the changes
    - i.e. re-install new value recorded in log
  - Then rollback all txns that were active at the end of the log
  - Now normal processing can resume

# Locks

- DBMS remembers which txns have set locks on which data, in which modes
  - locks requested by system itself when needed
    - or requested explicitly by application code (SQL “LOCK TABLE” statement)
- Lock requests that conflict with already held locks are delayed until the lock is available
  - This blocks any progress by the transaction that needs the lock

# Distributed commit

- For a transaction which involves activity at multiple databases
- When txn requests two commit, use two round-trips of messages
  - First check that all databases have log entries safe on disk (“prepare”)
  - Second round to inform each database of outcome, and they do what is necessary (eg release locks; also rollback if txn aborts)



# Implications

- Logging:
  - Extra storage needed
  - DBA must be very careful in configuring log
  - Possible delay in transaction commit
- Locks
  - blocking delays transactions
  - risk of deadlock
- Distributed commit
  - distributed transactions have very long delays
  - locks held during these delays may block other txns

# Overview

- Transactions and ACID properties
- Brief overview of Implementation Techniques
- Weak isolation issues

# Problems with serializability

- The performance reduction from serializable isolation is high
  - Transactions are often blocked because they want to read data that another txn has changed
- For many applications, the accuracy of the data they read is not crucial
  - e.g. overbooking a plane is ok in practice
  - e.g. your banking decisions would not be very different if you saw yesterday's balance instead of the most up-to-date

# A and D matter!

- Even when isolation isn't needed, no one is willing to give up atomicity and durability
  - These deal with modifications a txn makes
  - Writing is less frequent than reading, so log entries, and locks for writes (eg insert, update), are considered worth the effort

# Explicit isolation levels

- A transaction can be declared to have isolation properties that are less stringent than serializability
  - However SQL standard says that **default should be serializable**
  - In practice, many systems have weaker default level, and most txns run at weaker levels!

# Set Transaction for Browsing

- SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
  - Many platforms force the txn to be read-only
  - Allows txn to read dirty data (from a txn that will later abort)

# Read Committed

- SET TRANSACTION ISOLATION LEVEL READ COMMITTED
  - Data is never dirty (any value that gets read, was produced by a committed transaction)
  - but it can be inconsistent (between reads of different items, or even between one read and a later one of the same item)
    - Especially, weird things happen between different rows returned by a cursor

Most common in practice!

Also, default level in PostgreSQL

# Repeatable read

- SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
  - Whenever txn T examines a record several times in a transaction, always see the same value (unless T itself modified the record in between)
- In some systems, including PostgreSQL, you get a stronger property (sometimes called “Snapshot” or “MVCC”): all data read by transaction T comes from a common state, reflecting the transactions that committed before T started (and no concurrent ones)
  - read-only transaction never blocks
  - a transaction may be aborted by the system if a concurrent one is modifying same data



# Write Skew

- Snapshot breaks serializability when txns modify different items in each other's read sets
  - Neither txn sees the other, but in a serial execution one would come later and so see the other's impact
  - This is fairly rare in practice, but possible in some applications
    - if it occurs, it can lead to violation of undeclared integrity constraints

NB: sum uses old value of row1 and Product,  
and self-changed value of row2

## Example – Write Skew

|     |     |     |
|-----|-----|-----|
| p1  | s1  | 30  |
| p1  | s2  | 35  |
| p2  | s1  | 60  |
| etc | etc | etc |

|     |     |     |
|-----|-----|-----|
| p1  | etc | 32  |
| p2  | etc | 44  |
| etc | etc | etc |

Order: empty

Initial state of InStore, Product, Order

|     |     |     |
|-----|-----|-----|
| p1  | s1  | 4   |
| p1  | s2  | 10  |
| p2  | s1  | 60  |
| etc | etc | etc |

|     |     |     |
|-----|-----|-----|
| p1  | etc | 32  |
| p2  | etc | 44  |
| etc | etc | etc |

Order: empty<sub>66</sub>

Final state of InStore, Product, Order

- MakeSale(p1,s1,26) MakeSale(p1,s2,25)
- Update row 1: 30->4
- update row 2: 35->10
- find sum: 72
- // No need to Insert row in Order
- Find sum: 71
- // No need to insert row in Order
- COMMIT
- COMMIT

Integrity constraint is false:  
Sum is 46

# Platform variations

- PostgreSQL
  - default isolation is “read committed”
  - can get true serializability (but performance may suffer) by “SET TRANSACTION ISOLATION LEVEL SERIALIZABLE”
- See <https://www.postgresql.org/docs/9.1/static/transaction-iso.html>  
**PostgreSQL serializable mechanism based on design by Michael Cahill in his USyd PhD**
- Oracle
  - default isolation is “read committed”
  - Using “SET TRANSACTION ISOLATION LEVEL SERIALIZABLE” gives snapshot, not serializability
- See [https://docs.oracle.com/cd/E25054\\_01/server.1111/e25789/consist.htm](https://docs.oracle.com/cd/E25054_01/server.1111/e25789/consist.htm)  
**Theory by Fekete et al: how to determine if this is a problem, how to set explicit locks if <sup>67</sup> necessary to cope**

# Implications

- Know what isolation level your code is using
- Think carefully whether there are undeclared integrity constraints that matter to you
  - if so, you may need to run with serializable isolation
- Wise approach: start with serializable, then reduce isolation if necessary to meet user performance requirements