Transactions

Transaction

Definition

A logical unit of work that must either be entirely completed or aborted (indivisible, atomic), a sequence of DML statements

DML statements are already atomic; DBMS also allows for *user-defined* transactions

A successful transaction changes the database from one consistent state to another

Properties (ACID)

1. **Atomicity**
   A transaction is a single, indivisible, logical unit of work. All operations in a transaction must be completed, or the transaction is aborted.
2. **Consistency**
   Constraints that hold before a transaction must also hold after it
   Multiple users accessing the same data see the same value
3. **Isolation**
   Changes made during execution of a transaction cannot be seen by other transactions until this one is completed
4. **Durability**
   When a transaction is complete, the changes made to the database are permanent, even if the system fails

Why do we need a transaction?

Brief: 1. For single users: makes sure that data is consistent (if crashes); 2. For multi-users: prevents concurrency-related conflicts in data reading + writing

Problem 1: Unit of work

Users need the ability to define a unit of work

multiple statements (user-defined transaction)

```
START TRANSACTION;   (or, 'BEGIN')
    SQL statement;
    SQL statement;
    SQL statement;
    ...
COMMIT;     (commits the whole transaction)
    Or ROLLBACK (to undo everything)
```

SQL keywords: `begin/START TRANSACTION` , `commit` , `rollback` .

In the case of an error:

1. Any SQL statements already completed must be reversed
2. Show an error message to the user
3. When ready, the user can try the transaction again This is briefly annoying, but inconsistent data is disastrous.

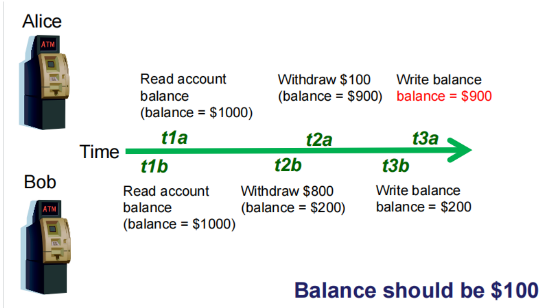
Problem 2: Concurrent access

Concurrent access to data by > 1 user or program

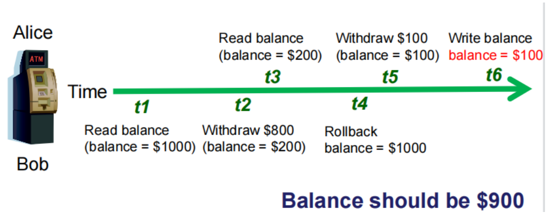
Concurrent execution of DML against a shared database.

Problems:

1. Lost updates (no updates)



2. Uncommitted data
   When two transactions execute concurrently and the first is rolled back after the second has already accessed the uncommitted data.



3. Inconsistent retrievals
   When one transaction calculates some aggregate functions over a set of data, while other transactions are updating the data (some data are read after change and some before, inconsistently)

Time	Trans-action	Action	Value	T1 SUM	Comment
1	T1	Read Salary for EmpID 11	10,000	10,000	
2	T1	Read Salary for EmpID 22	20,000	30,000	
3	T2	Read Salary for EmpID 33	30,000		
4	T2	Salary = Salary * 1.01			
5	T2	Write Salary for EmpID 33	30,300		
6	T1	Read Salary for EmpID 33	30,300	60,300	*after* update
7	T1	Read Salary for EmpID 44	40,000	100,300	*before* update
8	T2	Read Salary for EmpID 44	40,000		
9	T2	Salary = Salary * 1.01			
10	T2	Write Salary for EmpID 44	40,400		
11	T2	COMMIT			
12	T1	Read Salary for EmpID 55	50,000	[illegible]	
13	T1	Read Salary for EmpID 66	60,000	210,300	

we want either *before* $210,000 or *after* $210,700

Serializability

Transactions ideally are serializable.

Multiple, concurrent transactions appear as if they were executed one after another

Ensures that the concurrent execution of several transactions yields consistent results.

But true setial execution (no concurrency) is very expensive.

Logging

Allow us to restore the database to a previous consistent state (e.g. not completed, aborted => roll back; restore a corrupted database)

By tracking all updates to data. Contains:

1. A record for the beginning of the transaction

2. For each SQL statement: operation being performed (e.g. update, delete, insert); objects affected by the transaction; before and after values for updated fields; pointers to previous and next transaction log entries
3. COMMIT (ending of the transaction)

When failure occurs, DBMS will examine the log for all uncommitted or incomplete transactions, and restore the database to a precious state.

Concurrency

TO achieve efficient execution of transactions, the DBMS creates a schedule of read and write operations for concurrent transactions. Interleaves the execution of operations, based on concurrency control algorithms:

1. Locking (Main method)
2. Time stamping
3. Optimistic Concurrency Control

Control: Locking

Basics

Lock: Guarantees exclusive use of a data item to a current transaction Required to prevent another transaction from reading inconsistent data

Lock manager: Responsible for assigning and policing the locks used by the transactions

Lock Granularity Options

1. Database-level lock
   Entire database is locked; T1 and T2 can not access the same data base concurrently even if they use different tables
   Good for batch processing but unsuitable for multi-user DBMSs
2. Table-level lock
   Entire table is locked; T1 and T2 can access the same database concurrently as long as they use different tables
   Can cause bottlenecks: transactions want to access different parts of the table; Not suitable for highly multi-user DBMSs
3. Page-level lock
   An entire disk page is locked;
   Not commonly used now
4. Row-level lock
   Allows concurrent transactions to access different rows of the same table, even if the rows are located on the same page
   Improves data availability but with high overhead (each row has a lock that must be read and written to)
   Currently the most popular approach (MySQL, Oracle)
5. Field-level lock
   Allows concurrent transactions to access the same row, as long as they access different attributes within that row
   Most flexible lock but requires an extremely high level of overhead
   Not commonly used

Types of Locks: Binary Lock & Shared and Exclusive (/Read and Write) Locks

Type: Binary Locks

1(locked)/0(unlocked); the lock is not released until the statement is completed

Eliminates "Lost Update" problem

But too restrictive to yield optimal concurrency, as even two READS are locked

Type: Shared and Exclusive Locks

- Exclusive Lock: (when transaction intends to WRITE)
  Access is reserved for the transaction that locked the object; Granted iff no other locks are held on the data item (both exclusive & shared locks)
  e.g. MySQL: `SELECT ... FOR update`
- Shared Lock: (when a transaction wants to READ)
  Other transactions are also granted Read access; Issued when no Exclusive lock is held on that data item (can have multiple shared locks)
  e.g. MySQL: `SELECT ... FOR share`

Deadlock

When two transactions wait for each other to unlock data with exclusive locks (could wait forever)

Solutions

1. Prevention
2. Detection (then kill one of them; e.g. MySQL-kill the 2nd one)

Control: Alternative concurrency control methods

1. Timestamp

- Assigns a global unique timestamp to each transaction
- Each data item accessed by the transaction gets the timestamp
- When a transaction wants to read or write, the DBMS compares its timestamp with the timestamps already attached to the item, decides whether to allow access

1. Optimistic

- Based on the **assumption** that the majority of database operations do not conflict
- Transaction is executed without restrictinos or checking
- When commit, the DBMS checks whether any of the data it read has been altered, if so then rollback

Locking and COMMIT

What if we try once more, this time committing our changes in Window 1?

◆ **Task 3.22** In **Window 1**, change the worldwide gross of 'Shrek 2' again:

```
UPDATE movie
SET worldwide_gross = 10000000
WHERE name = 'Shrek 2';
```

◆ **Task 3.23** In **Window 2**, try to change the same row:

```
UPDATE movie
SET worldwide_gross = 5000
WHERE name = 'Shrek 2';
```

◆ **Task 3.24** While the UPDATE query in Window 2 is pending, commit **Window 1**'s transaction:

```
COMMIT;
```

User 1's update takes place. Then User 1's lock is released, meaning that User 2's update is performed immediately afterwards.

◆ **Task 3.25** In **Window 2**, confirm that the gross of 'Shrek 2' is now set to 5000:

```
SELECT *
FROM movie
WHERE name = 'Shrek 2';
```

◆ **Task 3.26** In **Window 2**, roll back the transaction:

```
ROLLBACK;
```

◆ **Task 3.27** In **Window 2**, confirm that the gross of 'Shrek 2' has returned to the value from Window 1:

```
SELECT *
FROM movie
WHERE name = 'Shrek 2';
```

Database Administration

The role of the DBA: Capacity planning and Backup and recovery

Capacity planning

"is the process of predicting when future load levels will saturate the system and determining the most cost-effective way of delaying system saturation as much as possible."

Consider: disk space requirements; transaction throughput; go-live and throughout the life of the system storage (7~20 years)

Consider: Data volumes; Access Frequencies

Estimating disk space requirement

Core Idea: Treat database size as the sum of all table sizes

$$Table\ size = number\ of\ rows \times average\ row\ width$$

These sizes are for MySQL and are slightly different for other vendors: https://dev.mysql.com/doc/refman/8.0/en/storage-requirements.html

Storage Requirements for Date and Time Types

Data Type	Storage Required
DATE	3 bytes
TIME	3 bytes
DATETIME	8 bytes
TIMESTAMP	4 bytes
YEAR	1 byte

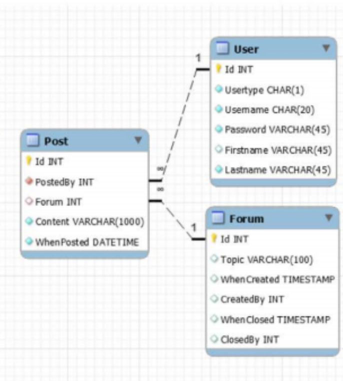
1. Calculating row widths (VARCHAR/BLOB use the average size from catalog)
2. Estimate growth of tables: Gather estimates during system analysis;
   Event Tables-most frequent tables; dominant the storage later

– "The company sells 1000 products. There are 2,000,000 customers who place, on average, 5 orders each per month. An average order is for 8 different products."

therefore:
the Product table has 1000 rows.
the Customer table has 2,000,000 rows.
the Orders table grows by 10,000,000 rows per month.
the OrderItems table grows by 80,000,000 rows per month.

event tables: dominant later

column	type	width	rows	1 month	1 year
USER					
Id	int	4			
UserType	char(1)	1			
UserName	char(10)	10			
Password	char(10)	10			
FirstName	varchar(45)	12			
LastName	varchar(45)	15	go-live		
ROW WIDTH		52	1,000,000	1,100,000	2,000,000
DISK SPACE			52,000,000	57,200,000	104,000,000
FORUM					
Id	int	4			
Topic	varchar(100)	50		per month	
WhenCreated	timestamp	4		1	
CreatedBy	int	4			
ClosedBy	int	4			
ROW WIDTH		66	100	101	113
DISK SPACE			6,600	6,666	7,458
POST					
Id	bigint	8			
PostedBy	int	4		per user per month	
Forum	int	4		30	
Content	varchar(1000)	500			
WhenPosted	datetime	8			
ROW WIDTH		524	0	30,000,000	390,000,000
DISK SPACE			0	15,720,000,000	204,360,000,000



Estimate Transaction Load

1. how often will transaction each be run?
2. for each transaction, what SQL statements are being run?

For example, consider this fictitious banking application:

3x20x1000000

Transaction	Selects	Inserts	Updates	Delete	SQL/tr	Tr/cust/month	SQL/month	SQL/second
Withdraw	1	1	1		3	20	60,000,000	23
Deposit		1	1		2	5	10,000,000	4
Transfer	1	1	2		4	8	32,000,000	12
								39
no. customers		1,000,000						

Summary

In fact, also need to store index; so much more!!

remember the size at go-live; how fast it grows; Frequencies of accessing

NOTE: capacity planning is a rough estimation. (but good enough)

Backup and Recovery

Why? Protect data from

1. **Human error**: e.g. accidental drop or delete
2. **Hardware or software malfunction**: e.g. bug in application, hard drive, CPU, memory
3. **Malicious activity**: e.g. security compromise (server, database, application)
4. **Natural or man made disasters**: need consider the scale of the damage
5. **Government Regulation**: e.g. historical archiving rules, Metadata collection, HIPPA,EU data retention regulations, Privacy Rules

Categories of Failure

1. Statement failure: Syntactically incorrect
2. User Process failure: The process doing the work fails (errors, dies)
3. Network failure: between the user and the database
4. User error: User accidentally drop the row, table, database
5. Memory failure: Memory fails, becomes corrupt
6. Media failure: Disk failure, corruption, deletion

Backup Types

Physical vs Logical

- **Physical** (Binary data being stored, clone the HDD)
  backup = exact copies of the database directories and files
  Database is preferably offline (cold backup) when backup occurs (MySQL is not wholly offline)
  Backup is only portable to machines with a similar configuration
  Suiltable for large databases that need fast recovery
  Restore:
  1.shut down DBMS
  2.copy backup over current structure on disk
  3.restart DBMS

- **Logical** (export of data as SQL statements)
  backup completed through SQL queries; doesn't include log/config files
  Server is available during the backup; machine independent
  Slower than physical; output is larger than physical
  in MySQL: `Mysqldump` / `SELECT...INTO OUTFILE`
  Restore: `mysqlimport` / `LOAD DATA INFILE`

Online vs Offline

- **Online (or Hot)**
  backups occur when the database is live
  Need to have appropriate locking to ensure integrity of data
  clients don't realise a backup is in progress

- **Offline (or COLD)**
  backups occur when the database is stopped
  Take backup from replication server not live server (To maximize availability to users)
  Simpler to perform; preferable, but not available in all situations (e.g. applications without downtime)

Full vs Incremental

- **FUll**
  The complete database is backed up (physical/logical/online/offline)
  Includes everything you need to get the database operational in the event of a failure

- **Incremental**
  Only the changes since the last backup are backuped; most databases: only backup log files
  Restore:
  1.Stop the database, copy backed up log files to disk
  2.start the database and tell it to redo the log files

b) Media failure at Friday 9:23am, how do we restore?

So here, need to restore: Sunday + Tuesday + Thursday + Crashlog

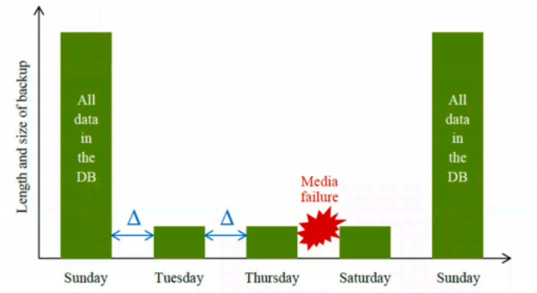


Figure 1. A timeline of full and incremental backups showing the media failure on Friday morning.

Onsite vs Offsite

- **Onsite**: the same equipment you are using now / Keep at same site as server (but different computer)
- **offsite**: store it somewhere else / Keep everything at a physically removed site (>160km = 100Mi)
  Enables disaster recovery
  Examples: backup tapes transported to underground vault; remote mirror database maintained via replication; backup to Cloud

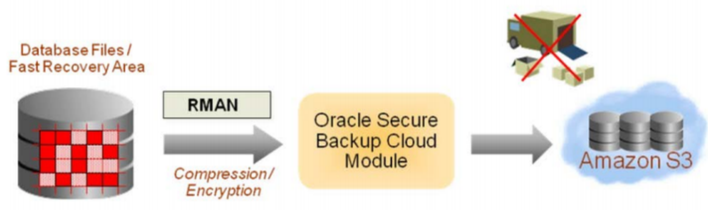


Figure 1. Oracle Database backup in the Cloud

Backup Policy

Backup strategy is usually a combination of full and incremental backups. (e.g. weekly full backup, weekday incremental backup)

Conduct backups when database load is low

If using replication, use the mirror database for backups to negate any performance concerns with the primary database

CRUTIAL: TEST your backup before you NEED your backup