

Transactions

With slides by Ph.Bonnet, J.Ulmann, M.Kifer

Outline

- Database Modifications
 - Insert, Update, Delete
- Transactions
 - ACID
 - COMMIT, ROLLBACK
 - Isolation Levels
- Chaining Transactions
- Triggers

Database Modifications

- ◆ A *modification* command does not return a result (as a query does), but changes the database in some way.
- ◆ Three kinds of modifications:
 1. *Insert* a tuple or tuples.
 2. *Delete* a tuple or tuples.
 3. *Update* the value(s) of an existing tuple or tuples.

Insertion

- To insert a single tuple:

```
INSERT INTO <relation>  
VALUES ( <list of values> );
```

- **Example:** add to **Likes(drinker, beer)** the fact that Sally likes Bud.

```
INSERT INTO Likes  
VALUES ( 'Sally' , 'Bud' );
```

Specifying Attributes in INSERT

- ◆ We may add to the relation name a list of attributes.
- ◆ Two reasons to do so:
 1. We forget the standard order of attributes for the relation.
 2. We don't have values for all attributes, and we want the system to fill in missing components with NULL or a default value.

Example: Specifying Attributes

- Another way to add the fact that Sally likes Bud to `Likes(drinker, beer)`:

```
INSERT INTO Likes (beer, drinker)
VALUES ('Bud', 'Sally');
```

Adding Default Values

- In a CREATE TABLE statement, we can follow an attribute by DEFAULT and a value.
- When an inserted tuple has no value for that attribute, the default will be used.

Example: Default Values

```
CREATE TABLE Drinkers (  
    name CHAR(30) PRIMARY KEY,  
    addr CHAR(50)  
        DEFAULT '123 Sesame St.',  
    phone CHAR(16)  
);
```

Example: Default Values

```
INSERT INTO Drinkers (name)
VALUES ('Sally');
```

Resulting tuple:

name	address	phone
Sally	123 Sesame St	NULL

Inserting Many Tuples

- We may insert the entire result of a query into a relation, using the form:

```
INSERT INTO <relation>  
( <subquery> );
```

Example: Insert a Subquery

- Using `Frequents(drinker, bar)`, enter into the new relation `PotBuddies(name)` all of Sally's "potential buddies," i.e., those drinkers who frequent at least one bar that Sally also frequents.

The other
drinker

Solution

Pairs of Drinker
tuples where the
first is for Sally,
the second is for
someone else,
and the bars are
the same.

INSERT INTO PotBuddies

```
(SELECT d2.drinker  
FROM Frequents d1, Frequents d2  
WHERE d1.drinker = 'Sally' AND  
d2.drinker <> 'Sally' AND  
d1.bar = d2.bar  
);
```

Deletion

- To delete tuples satisfying a condition from some relation:

```
DELETE FROM <relation>  
WHERE <condition>;
```

Example: Deletion

- Delete from **Likes(drinker, beer)** the fact that Sally likes Bud:

```
DELETE FROM Likes
WHERE drinker = 'Sally' AND
      beer = 'Bud';
```

Example: Delete all Tuples

- Make the relation Likes empty:

```
DELETE FROM Likes;
```

- Note no WHERE clause needed.

Example: Delete Some Tuples

- Delete from **Beers(name, manf)** all beers for which there is another beer by the same manufacturer.

```
DELETE FROM Beers b
WHERE EXISTS (
  SELECT name FROM Beers
  WHERE manf = b.manf AND
  name <> b.name);
```

Beers with the same manufacturer and a different name from the name of the beer represented by tuple b.

Semantics of Deletion --- (1)

- Suppose Anheuser-Busch makes only Bud and Bud Lite.
- Suppose we come to the tuple b for Bud first.
- The subquery is nonempty, because of the Bud Lite tuple, so we delete Bud.
- Now, when b is the tuple for Bud Lite, do we delete that tuple too?

Semantics of Deletion --- (2)

- ◆ **Answer:** we *do* delete Bud Lite as well.
- ◆ The reason is that deletion proceeds in two stages:
 1. Mark all tuples for which the WHERE condition is satisfied.
 2. Delete the marked tuples.

Updates

- To change certain attributes in certain tuples of a relation:

UPDATE <relation>

SET <list of attribute assignments>

WHERE <condition on tuples>;

Example: Update

- Change drinker Fred's phone number to 555-1212:

```
UPDATE Drinkers  
SET phone = '555-1212'  
WHERE name = 'Fred';
```

Example: Update Several Tuples

- Make \$4 the maximum price for beer:

```
UPDATE Sells  
SET price = 4.00  
WHERE price > 4.00;
```

Outline

- Database Modifications
 - Insert, Update, Delete
- Transactions
 - ACID
 - COMMIT, ROLLBACK
 - Isolation Levels
- Chaining Transactions
- Triggers

Why Transactions?

- Database systems are normally being accessed by many users or processes at the same time.
 - Both queries and modifications.
- Unlike operating systems, which *support* interaction of processes, a DMBS needs to keep processes from troublesome interactions.

Example: Bad Interaction

- You and your domestic partner each take \$100 from different ATM's at about the same time.
 - The DBMS better make sure one account deduction doesn't get lost.
- **Compare:** An OS allows two people to edit a document at the same time. If both write, one's changes get lost.

Transactions

- *Transaction* = group of operations for which the database management system guarantees some properties.
- Normally with some strong properties regarding concurrency.
- Formed in SQL from single statements or explicit programmer control.

ACID Transactions

- *ACID transactions* are:
 - *Atomic* : Whole transaction or none is done.
 - *Consistent* : Database constraints preserved.
 - *Isolated* : It appears to the user as if only one process executes at a time.
 - *Durable* : Effects of a process survive a crash.
- **Optional**: weaker forms of transactions are often supported as well.

Consistency

- **Enterprise (Business) Rules** limit the occurrence of certain real-world events
 - Student cannot register for a course if the current number of registrants equals the maximum allowed
- Correspondingly, allowable database states are restricted

cur_reg ≤ *max_reg*
- These limitations are specified through (static) **integrity constraints**: assertions that must be satisfied by the database state

Consistency

- Other static consistency requirements are related to the fact that the database might store the same information in different ways
 - $cur_reg = |list_of_registered_students|$
 - Such limitations are also expressed as integrity constraints
- **Database is consistent** if all static integrity constraints are satisfied

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:** A letter grade in a course (A, B, C, D, F) cannot be changed to an incomplete (I)
- Dynamic constraints cannot be checked by examining the database state

Transaction Consistency

- A **transaction is consistent** if, assuming the database is in a consistent state initially, then it is also in a consistent state when the transaction completes:
 - All static integrity constraints are satisfied (but constraints might be violated in intermediate states)
 - No dynamic constraints have been violated

Checking Integrity Constraints

- Automatic: Embed constraint in schema.
 - CHECK, ASSERTION for static constraints
 - TRIGGER for dynamic constraints
 - 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

Assertions

- These are database-schema elements, like relations or views.

- Defined by:

```
CREATE ASSERTION <name>  
CHECK (<condition>);
```

- Condition may refer to any relation or attribute in the database schema.

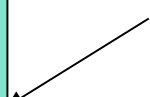
Example: Assertion

- In `Sells(bar, beer, price)`, no bar may charge an average of more than \$5.

```
CREATE ASSERTION NoRipoffBars CHECK (  
  NOT EXISTS (  
    SELECT bar FROM Sells  
    GROUP BY bar  
    HAVING 5.00 < AVG(price)  
  ));
```

```
SELECT bar FROM Sells  
GROUP BY bar  
HAVING 5.00 < AVG(price)
```

Bars with an
average price
above \$5



```
));
```

Example: Assertion

- In `Drinkers(name, addr, phone)` and `Bars(name, addr, license)`, there cannot be more bars than drinkers.

```
CREATE ASSERTION FewBar CHECK (  
    (SELECT COUNT(*) FROM Bars) <=  
    (SELECT COUNT(*) FROM Drinkers)  
);
```

Timing of Assertion Checks

- In principle, we must check every assertion after every modification to any relation of the database.
- A clever system can observe that only certain changes could cause a given assertion to be violated.
 - **Example:** No change to Beers can affect FewBar. Neither can an insertion to Drinkers.

Atomicity

- A real-world event either happens or does not happen
 - Student either registers or does not register
- Similarly, the system must ensure that either the corresponding transaction runs to completion or, if not, it has no effect at all
 - Not true of ordinary programs. A crash could leave files partially updated on recovery

Commit and Abort

- If the transaction successfully completes it is said to **commit**
 - The system is responsible for ensuring that all changes to the database have been saved
- If the transaction does not successfully complete, it is said to **abort**
 - The system is responsible for undoing, or **rolling back**, all changes the transaction has made

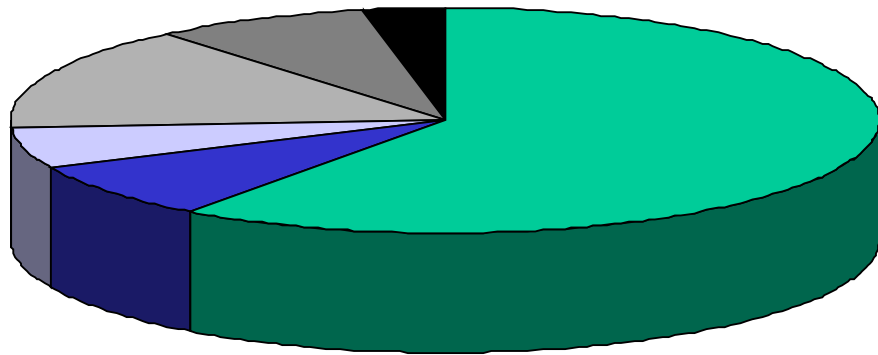
Reasons for Abort

- System crash
- Transaction aborted by system
 - Execution cannot be made atomic (a site is down)
 - Execution did not maintain database consistency (integrity constraint is violated)
 - Execution was not isolated
 - Resources not available (deadlock)
- Transaction requests to roll back

Atomicity Requests

- DBMS and TP monitor provide commands for setting transaction boundaries. For example:
 - begin transaction
 - commit
 - rollback
- The commit command is a request
 - The system might commit the transaction, or it might abort it for one of the reasons on the previous slide
- The rollback command is always satisfied

Outages



From J.Gray and A.Reuters
Transaction Processing: Concepts
and Techniques

- A fault tolerant system must provision for all causes of outages (see case studies)
- Software is the problem
 - Hardware failures cause under 10% of outages
 - Heisenbugs stop the system without damaging the data.

Database systems protect integrity against single hardware failure and some software failures.

Isolation

- Each transaction is executed as if it was the only one in the system.
- Concurrent executions
 - A computer system has multiple resources capable of executing independently (e.g., cpu's, I/O devices), *but*
 - A transaction typically uses only one resource at a time
 - Hence, only concurrently executing transactions can make effective use of the system
 - Concurrently executing transactions yield interleaved schedules

Correctness Criteria

- An interleaved schedule of transactions is **isolated** if its effect is the same as if the transactions had executed serially in some order (**serializable**)

$T1: r(x) \quad w(x)$

$T2: \quad r(y) \quad w(y)$

- It follows that serializable schedules are always correct (for any application)
- Serializable is better than serial from a performance point of view
- DBMS uses **locking** to ensure that concurrent schedules are serializable (more next lecture)

Example: Interacting Processes

- Assume the usual **Sells(bar,beer,price)** relation, and suppose that Joe's Bar sells only Bud for \$2.50 and Miller for \$3.00.
- Sally is querying **Sells** for the highest and lowest price Joe charges.
- Joe decides to stop selling Bud and Miller, but to sell only Heineken at \$3.50.

Sally's Program

- Sally executes the following two SQL statements called **(min)** and **(max)** to help us remember what they do.

(max) SELECT MAX(price) FROM Sells
WHERE bar = 'Joe''s Bar';

(min) SELECT MIN(price) FROM Sells
WHERE bar = 'Joe''s Bar';

Joe's Program

- At about the same time, Joe executes the following steps: **(del)** and **(ins)**.

(del) DELETE FROM Sells

 WHERE bar = 'Joe's Bar';

(ins) INSERT INTO Sells

 VALUES('Joe's Bar', 'Heineken', 3.50);

Interleaving of Statements

- Although **(max)** must come before **(min)**, and **(del)** must come before **(ins)**, there are no other constraints on the order of these statements, unless we group Sally's and/or Joe's statements into transactions.

Example: Strange Interleaving

- Suppose the steps execute in the order **(max)(del)(ins)(min)**.

Joe's Prices:	{2.50,3.00}	{2.50,3.00}		{3.50}
Statement:				
Result:	(max)	(del)	(ins)	(min)
	3.00			3.50

- Sally sees MAX < MIN!

Using Transactions

- If we group Sally's statements (max)(min) into one transaction, then she cannot see this inconsistency.
- She sees Joe's prices at some fixed time.
 - Either before or after he changes prices, or in the middle, but the MAX and MIN are computed from the same prices.

Another Problem: Rollback

- Suppose Joe executes **(del)(ins)**, not as a transaction, but after executing these statements, thinks better of it and issues a ROLLBACK statement.
- If Sally executes her statements after **(ins)** but before the rollback, she sees a value, 3.50, that never existed in the database.

Solution

- If Joe executes **(del)(ins)** as a transaction, its effect cannot be seen by others until the transaction executes COMMIT.
 - If the transaction executes ROLLBACK instead, then its effects can *never* be seen.

Isolation Levels

- SQL defines four *isolation levels* = choices about what interactions are allowed by transactions that execute at about the same time.
- Only one level (“serializable”) = ACID transactions.
- Each DBMS implements transactions in its own way.

Choosing the Isolation Level

◆ Within a transaction, we can say:

SET TRANSACTION = X

where X =

1. SERIALIZABLE
2. REPEATABLE READ
3. READ COMMITTED
4. READ UNCOMMITTED

Serializable Transactions

- If Sally = (max)(min) and Joe = (del)(ins) are each transactions, and Sally runs with isolation level SERIALIZABLE, then she will see the database either before or after Joe runs, but not in the middle.

Isolation Level Is Personal Choice

- Your choice, e.g., run serializable, affects only how *you* see the database, not how others see it.
- **Example:** If Joe Runs serializable, but Sally doesn't, then Sally might see no prices for Joe's Bar.
 - i.e., it looks to Sally as if she ran in the middle of Joe's transaction.

Read-Committed Transactions

- If Sally runs with isolation level READ COMMITTED, then she can see only committed data, but not necessarily the same data each time.
- **Example:** Under READ COMMITTED, the interleaving **(max)(del)(ins)(min)** is allowed, as long as Joe commits.
 - Sally sees $MAX < MIN$.

Repeatable-Read Transactions

- Requirement is like read-committed, plus: if data is read again, then everything seen the first time will be seen the second time.
 - But the second and subsequent reads may see *more* tuples as well.

Example: Repeatable Read

- Suppose Sally runs under REPEATABLE READ, and the order of execution is (max)(del)(ins)(min).
 - (max) sees prices 2.50 and 3.00.
 - (min) can see 3.50, but must also see 2.50 and 3.00, because they were seen on the earlier read by (max).

Read Uncommitted

- A transaction running under READ UNCOMMITTED can see data in the database, even if it was written by a transaction that has not committed (and may never).
- **Example:** If Sally runs under READ UNCOMMITTED, she could see a price 3.50 even if Joe later aborts.

Outline

- Database Modifications
 - Insert, Update, Delete
- Transactions
 - ACID
 - COMMIT, ROLLBACK
 - Isolation Levels
- Chaining Transactions
- Triggers

Flat Transaction

- Abort causes the execution of a program that restores the variables updated by the transaction to the state they had when the transaction first accessed them.

```
begin transaction
  .....
EXEC SQL .....
  .....
EXEC SQL .....
  .....
if condition then abort
  .....
commit
```

Limitations of Flat Transactions

- Only total rollback (abort) is possible
 - Partial rollback not possible
- All work lost in case of crash
- Limited to accessing a single DBMS
- Entire transaction takes place at a single point in time

Chained Transactions

- **Problem 1** (trivial): Invoking `begin_transaction` at the start of each transaction involves communication overhead
- With chaining, a new transaction is started automatically for an application program when the program commits or aborts the previous one
 - This is the approach taken in SQL

Chained Transactions

begin transaction
S1
commit
S2
begin transaction
S3
commit



S2 not included in a transaction since it has no db operations

begin transaction
S1
commit
begin transaction
S2
S3
commit

Equivalent since S2 does not access the database

transaction starts implicitly

S1
commit
S2
S3
commit

Chaining equivalent

Chained Transactions

- **Problem 2:** If the system crashes during the execution of a long-running transaction, considerable work can be lost
- Chaining allows a transaction to be decomposed into sub-transactions with intermediate commit points
- Database updates are made durable at intermediate points => less work is lost in a crash

S1		S1;
S2	=>	commit;
S3		S2;
commit		commit;
		S3;
		commit;

Chaining Considerations

- However, the transaction as a whole is not atomic. If crash occurs:
 - *DBMS* cannot roll the entire transaction back
 - Initial subtransactions have committed,
 - Their updates are durable
 - The updates might have been accessed by other transactions (locks have been released)
 - Hence, the *application* must roll itself forward

Chaining Considerations

- Roll forward requires that on recovery the application can determine how much work has been committed
 - Each subtransaction must tell successor where it left off
- Communication between successive subtransactions cannot use local variables (they are lost in a crash)
 - Use the database to communicate between subtransactions

```
r(rec_index:0);
S1;                -- update records 1 - 1000
w(rec_index:1000); -- save position of last record updated
commit;
r(rec_index:1000); -- get position of last record updated
S2;                -- update records 1001 – 2000
w(rec_index:2000);
commit;
```

Outline

- Database Modifications
 - Insert, Update, Delete
- Transactions
 - ACID
 - COMMIT, ROLLBACK
 - Isolation Levels
- Chaining Transactions
- Triggers

Trigger Details

- **Activation** - Occurrence of the *event*
- **Consideration** - The point, after activation, when *condition* is evaluated
 - Immediate or deferred (when the transaction requests to commit)
 - *Condition* might refer to both the state before and the state after *event* occurs

Trigger Details

- **Execution** – point at which *action* occurs
 - With deferred consideration, execution is also deferred
 - With immediate consideration, execution can occur immediately after consideration or it can be deferred
 - If execution is immediate, execution can occur before, after, or instead of triggering event.
 - Before triggers adapt naturally to maintaining integrity constraints: violation results in rejection of event.

Trigger Details

- **Granularity**

- *Row-level granularity*: change of a single row is an event (a single UPDATE statement might result in multiple events)
- *Statement-level granularity*: events are statements (a single UPDATE statement that changes multiple rows is a single event).

Trigger Details

- **Multiple Triggers**

- How should multiple triggers activated by a single event be handled?
 - Evaluate one condition at a time and if true immediately execute action or
 - Evaluate all conditions, then execute actions
- The execution of an action can affect the truth of a subsequently evaluated condition so the choice is significant.

Take Away Points

- A transaction is a group of operations for which the database provide some garantees
 - Atomicity
 - Consistency
 - Isolation
 - Durability
- A programmer groups read (select), and write (insert, update, delete) operations into transactions and either commit or rollback.