

## INFSCI 2710 Database Management

### Transaction Management and Concurrency Control

*Week 10*

### Today's Plan

- Assignments
- Final Project
- Normalization
- Functions
- Transactions
- Concurrency in DBMS

## Online Class

- Q1:

```
select business.name, business.stars,
case
when business.stars=3 then "medium evaluation business"
when business.stars=3.5 then "superior medium evaluation business"
when business.stars=4 then "good evaluation business"
when business.stars=4.5 then "very good evaluation business"
when business.stars=5 then "excellent evaluation business"
else "below average level evaluation business"
End as evaluation
from business,category
```

## Online Class

- Q2:

- Sample table of  $r = 5,000,000$  records of a fixed size
- Record length of  $R = 304$  bytes ( $4 + 100 + 100 + 100$ )
- Stored in a table using the MyISAM engine which is using the default block size  $B = 1,024$  bytes.
- The blocking factor of the table would be  $bfr = (B/R) = 1024/304 = 3$  records per disk block.
- The total number of blocks required to hold the table is  $N = (r/bfr) = 5000000/3 = 1666666$  blocks.
- $\log_2(1666666) = 20.66 \sim 21$  (for PK), or  $N/2$  for name fields

- Q3:  
CREATE view Practice as  
SELECT \*  
FROM business,category,review

## Homework #3

- Task 3: due to 'Sending Data'
- Task 4:
  - Put everything in one table (De-normalization, prevent join)
  - Index is not sufficient as well.
- Task 5:
  - Cache everything in memory;
  - Cache in file systems + B-Tree search

## Schedule

- July 17 – Transactions and Concurrency
- July 24 – Crash Recovery
  - Final Project Due (11:59pm)
- July 25-30 Project Demo
- July 31 – Final Exam

## Final Project: Data

- Customers: customer ID, name, address (street, city, state, zip code), kind (home/business). If business, then business category, company gross annual income, etc. If home, then marriage status, gender, age, income.
- Products: product ID, name, inventory amount, price, product kind w.r.t. some classification.
- Transactions: record of product purchased, including order number, date, salesperson name, product information (price, quantity, etc.), customer information.
- Salespersons: name, address, e-mail, job title, store assigned, salary.
- Store: store ID, address, manager, number of salespersons, region.
- Region: region ID, region name, region manager.
- This is example, you are free to create your database/data.

## Final Project: Operations

- Customer (Item) Browsing
  - The users must be able to search the operational database for particular items based on various attributes and must also be able to do browsing (i.e., less focused searching, itemize the search results and check the details).
- Update Transactions
  - The system must be able to handle payments and sales, new inventory, new users, etc. and other changes to the operational database that are necessary for the day-to-day running of the business (e.g., check out function for purchasing).
- Error Checking
  - The system must be robust and support various application-dependent integrity constraints. For example, items should not be sold if they are not currently in stock, etc.

## Final Project: Operations

- Data Aggregation The system must provide data aggregation queries (at least 3)
- For example...
  - What are the aggregate sales and profit of the products?
  - What are the top product categories?
  - How do the various regions compare by sales volume?
  - Which businesses are buying given products the most?
  - Other interesting aggregate queries that you will come up with.

## Final Project: Rules

- Implementation Tools
  - All projects are expected to be runnable from a web browser from the instructor's office. SQL is a requirement.
  - A group may choose to use any database systems and front-end implementation tools after discussing it with instructor.

## Final Project: Rules

- Additional Requirements:
  - The project must represent a fairly sophisticated database application.
  - In particular the database must contain multiple (e.g., **at least seven**) relations, and the database design must include **indexes, primary keys**, etc.

## Final Project: Rules

- Assumptions:
  - In cases where the above description of the application is incomplete, it is acceptable to make assumptions about the application providing that:
  - 1) they are explicitly stated in the final report, 2) they don't conflict with any of the requirements specified above, and 3) they are "reasonable".
  - If you have a question about the acceptability of any of your assumptions, check with the instructor. Interesting questions should be raised in class.

## Final Project: Rules

- Report:
  - A final report should be handed in for grading at the end of the term.
  - The report must be formatted in a reasonable manner (i.e., using a text processor).
  - The final report is due during class on the "Project Due" date specified in the class schedule. (July 24 11:59 pm)

## Final Project: Rules

- Implementation:
  - The project requires a working implementation of the system to be built, tested, and demonstrated.
  - A large part of the project grade depends on the quality of this implementation.

## Final Project: Report

- A short overview of the system including identification of the various types of users, administrators, etc. who will be accessing the system in various ways.
- A list of assumptions that you have made about the system.
- A graphical schema of the database using the E-R diagram with a short description of each entity set, relationship set and their corresponding attributes.
- A set of relational schemas resulting from the E-R diagram with identification of primary and foreign keys.
- The DDL statements to create the relational schema in some appropriate Normal Form, with identification and justification of the Normal Form.
- A description of your front-end design as well as the front-end to back-end connection.
- A brief overview of the system implementation with example screen shots.
- A description of your testing efforts and erroneous cases that your system can detect and handle.
- A description of the system's limitations and the possibilities for improvements.



## Final Project: Demo

- A demo of the working system will be required.
- All members of the group must attend this demo, and must be prepared to explain and demonstrate those aspects of the project for which they were responsible.
- The source code for the project should be available on-line during the demonstration.
- Demos will be scheduled after the last class and before the final exam (July 25-30)
- 10 min for presentation and 10 min for questions.

## Final Project: Grading

- Final Project: 20% of your final score
  - 2% Project Proposal
  - 10% Report Submission
    - Overview, assumptions, graphical schema, DDL/Normal Form, front&back end design/system screen shots/testing efforts/limitation&improvements
    - Poor/Fair/Good/Very Good
  - 8% Demo and System Implementation
    - Customer (item) browsing, update transaction, error checking, data aggregation -> I will focus on the functionality and implementation
    - Poor/Fair/Good/Very Good
  - (2% Bonus Score): Data Visualization, Analytic and Mining, etc.
    - You must discuss the ideas with the instructor if you plan to earn it.

## **First Normal Form (1NF)**

- Eliminate duplicate columns from the same table.
- Create separate tables for each group of related data and identify each row with a unique column or set of columns (the primary key).
- Atomic: the column only stores one thing.
- No Repeating Group: no duplicate columns

## **Second Normal Form (2NF)**

- Meet all the requirements of the first normal form.
- Remove subsets of data that apply to multiple rows of a table and place them in separate tables.
- Create relationships between these new tables and their predecessors through the use of foreign keys.
- No Partial Functional Dependencies: each non-key attribute is Functionally Dependent on the full Primary Key

## Third Normal Form (3NF)

- Meet all the requirements of the second normal form.
- Remove columns that are not dependent upon the primary key.
- **No Transitive Functional Dependencies:** A Transitive Functional Dependency is when a non-key column is Functionally Dependent on another non-key column, which is Functionally Dependent on the Primary Key.

## Normalization Exercise 2

petID	petName	petType	petAge	ownerName	visitDate	procedure
246	Rover	Dog	12	Sam Cook	JAN 13/2002 MAR 27/2002 APR 02/2002	01 - RABIES VACCINATION 10 - EXAMINE and TREAT WOUND 05 - HEART WORM TEST
298	Spot	Dog	2	Terry Kim	JAN 21/2002 MAR 10/2002	08 - TETANUS VACCINATION 05 - HEART WORM TEST
341	Morris	Cat	4	Sam Cook	JAN 23/2001 JAN 13/2002	01 - RABIES VACCINATION 01 - RABIES VACCINATION
519	Tweedy	Bird	2	Terry Kim	Apr 30, 2002 May 16, 2004	20 - ANNUAL CHECK UP 12 - EYE WASH

Data Redundancy?  
Update Anomaly?  
Insert Anomaly?  
Delete Anomaly?

Adapted from  
[http://cs.senecac.on.ca/~dbs201/pages/Norm\\_Exercise1.doc](http://cs.senecac.on.ca/~dbs201/pages/Norm_Exercise1.doc)

## Normalization Exercise 3

### INVOICE

<b>FROM:</b> HILLTOP ANIMAL HOSPITAL DATE: 01/13/2014 INVOICE # 987	<b>TO:</b> MR. RICHARD COOK  123 Phillips Avenue Pittsburgh, PA 15218
---	--

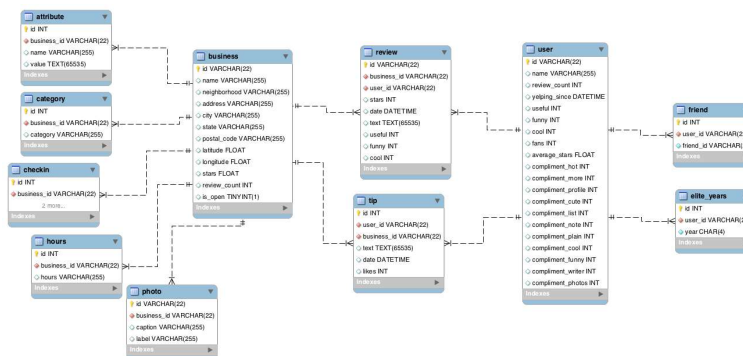
Pet	Procedure	Amount
Rover	Rabies Vaccination	30.00
Morris	Rabies Vaccination	24.00
	Subtotal:	54.00
	PA Tax (7%)	3.78
	Amount Owing	<b>\$57.78</b>

Data Redundancy?  
Update Anomaly?  
Insert Anomaly?  
Delete Anomaly?

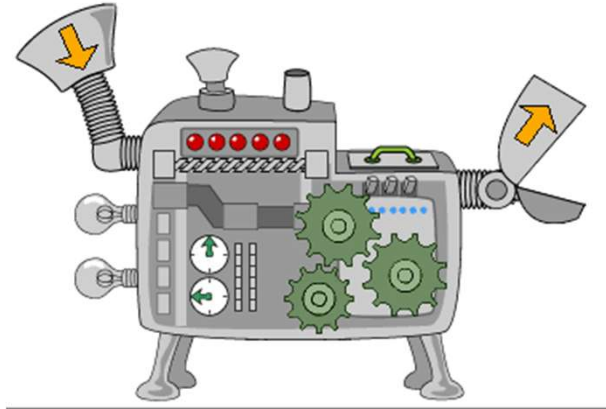
Adapted from [http://cs.senecac.on.ca/~dbs201/pages/Norm\\_Exercise2.doc](http://cs.senecac.on.ca/~dbs201/pages/Norm_Exercise2.doc)

## Discussion

- Why do we learn from the Yelp dataset?



## Functions



## Stored Functions

- **Stored Function** is a stored program that returns a **single value**.
- You can use stored functions to **encapsulate** common formulas or business rules that may be **reusable** among SQL statements or stored programs.
- You can use a stored function in SQL statements wherever an expression is used. This helps improve the readability and maintainability of the procedural code.

## Stored Functions

- Functions cannot perform permanent environmental changes to SQL Server (i.e. no INSERT or UPDATE statements allowed).
- A Function can be used inline in SQL Statements if it returns a scalar value or can be joined upon if it returns a result set

## Stored Functions



Function

```
SELECT * FROM orders WHERE YEAR(shippedDate) = 2003;
```

## Create Function Syntax

```
CREATE FUNCTION function_name(param1,param2,...)
  RETURNS datatype
  [NOT] DETERMINISTIC
  statements
```

## Create Function Syntax

- Specify the name of the stored function after CREATE FUNCTION keywords.

```
CREATE FUNCTION addTwoNumbers
```

- List all parameters of the function.

```
CREATE FUNCTION addTwoNumbers(num1 double, num2 double)
```

## Create Function Syntax

- Specify the data type of the return value in the RETURNS statement. It can be any valid MySQL data type.

```
CREATE FUNCTION addTwoNumbers(num1 double, num2 double)  
    RETURNS double
```

## Create Function Syntax

- For the same input parameters, if the function returns the same result, it is considered deterministic and not deterministic otherwise.

```
CREATE FUNCTION addTwoNumbers(num1 double, num2 double)  
    RETURNS double  
    DETERMINISTIC
```



## Create Function Syntax

- For the same input parameters, if the function returns the same result, it is considered deterministic and not deterministic otherwise.

```
CREATE FUNCTION addTwoNumbers(num1 double, num2 double)  
    RETURNS double  
    DETERMINISTIC
```

- You have to decide whether a stored function is deterministic or not. If you declare it incorrectly, the stored function may produced an unexpected result, or the available optimization is not used which degrade the performance.

## Create Function Syntax

- Fifth, you write the code in the statements section. It can be a single statement or a compound statement. Inside the statements section, you have to specify at least one **RETURN** statement.
- The RETURN statement returns a value to the caller. Whenever the RETURN statement is reached, the stored function's execution is terminated immediately.

## Create Function Syntax

```

DELIMITER $$

CREATE FUNCTION addTwoNumbers(num1 double, num2 double)
    RETURNS double
    DETERMINISTIC
BEGIN
    DECLARE sumOfTwoNumbers double;

    sumOfTwoNumbers = num1 * num2;

    RETURN (sumOfTwoNumbers);
END

```

## Create Function Syntax

```

DELIMITER $$

CREATE FUNCTION customerLevel(p_creditLimit double) RETURNS VARCHAR(10)
    DETERMINISTIC
BEGIN
    DECLARE lvl varchar(10);

    IF p_creditLimit > 50000 THEN
        SET lvl = 'PLATINUM';
    ELSEIF (p_creditLimit <= 50000 AND p_creditLimit >= 10000) THEN
        SET lvl = 'GOLD';
    ELSEIF p_creditLimit < 10000 THEN
        SET lvl = 'SILVER';
    END IF;

    RETURN (lvl);
END

```

## Definition of Stored Procedures

- Stored Procedure (SP) is a segment of declarative SQL statements stored inside the database catalog.
- SP can be invoked by triggers, other stored procedures or applications such as Java, C#, PHP, etc.
- SP that calls itself is known as a **recursive stored procedure**. MySQL does not support recursion very well.

## SP Advantages

- SPs help increase the performance of the applications. Once created, stored procedures are compiled and stored in the database.
- MySQL SPs are compiled on demand.
  - After compiling a stored procedure, MySQL puts it to a cache.
  - MySQL maintains its own SP cache for every single connection.
  - If an application uses a stored procedure multiple times in a single connection, the compiled version is used, otherwise the stored procedure works like a query.

## SP Advantages

- SPs helps reduce the traffic between application and database server
  - Instead of sending multiple lengthy SQL statements, the application has to send only name and parameters of the stored procedure.
- SPs are reusable and transparent to any applications.
  - Stored procedures expose the database interface to all applications so that developers don't have to develop functions that are already supported in stored procedures.

## SP Advantages

- SPs are more secure than SQL statements or scripts.
  - Database administrator can grant appropriate permissions to applications that access stored procedures in the database without giving any permission on the underlying database tables.

## **SP Disadvantages**

- If you use a lot of SPs, the memory usage of every connection that is using those stored procedures will increase substantially.
- If you overuse a large number of logical operations inside SPs, the CPU usage will also increase because database server is not well-designed for logical operations.

## **SP Disadvantages**

- It is difficult to debug stored procedures. Only few database management systems allow you to debug stored procedures. Unfortunately, MySQL does not provide facilities for debugging stored procedures.

## CREATE STORED PROCEDURE

```
CREATE PROCEDURE GetAllProducts()  
  BEGIN  
    SELECT * FROM products;  
  END;
```

## CREATE STORED PROCEDURE

- **CREATE PROCEDURE** statement creates a new stored procedure.
- Specify the name of SP after the CREATE PROCEDURE statement. In this case, the name of the stored procedure is GetAllProducts.
- We put the parentheses after the name of the stored procedure.

## CREATE STORED PROCEDURE

- The section between BEGIN and END is called the body of the stored procedure.
- You put the declarative SQL statements in the body to handle business logic.
- In this stored procedure, we use a simple SELECT statement to query data from the products table.

## CALL STORED PROCEDURE

CALL STORED\_PROCEDURE\_NAME()

- You use the CALL statement to call a stored procedure e.g., to call the *GetAllProducts* stored procedure, you use the following statement:

CALL GetAllProducts();

## DECLARE SP VARIABLES

To declare a variable inside a stored procedure, you use the **DECLARE** statement as follows:

```
DECLARE variable_name datatype(size) DEFAULT default_value;
```

## DECLARE SP VARIABLES

```
DECLARE variable_name datatype(size) DEFAULT default_value;
```

- Specify the variable name after the **DECLARE** keyword.
  - The variable name must follow the naming rules of MySQL table column names.
- Specify the data type of the variable and its size.
  - A variable can have any MySQL data types such as INT, VARCHAR, DATETIME, etc.



## DECLARE SP VARIABLES

- When you declare a variable, its initial value is NULL.
  - You can assign the variable a default value by using DEFAULT keyword.

```
DECLARE total_sale INT DEFAULT 0
```

## DECLARE SP VARIABLES

- Once you declared a variable, you can start using it. To assign a variable another value, you use the SET statement, for example:

```
DECLARE total_count INT DEFAULT 0  
SET total_count = 10;
```

## SP PARAMETERS

- Almost stored procedures that you develop require parameters.
- The parameters make the stored procedure more flexible and useful.
- In MySQL, a parameter has one of three modes
  - IN
  - OUT
  - INOUT

## SP PARAMETERS - IN

- IN is the default mode.
- When you define an IN parameter in a SP, the calling program has to pass an argument to the SP.
- The value of an IN parameter is protected.
  - It means that even the value of the IN parameter is changed inside the SP, its original value is retained after the SP ends.
  - The SP only works on the copy of the IN parameter.

## **SP PARAMETERS - OUT**

- OUT – the value of an OUT parameter can be changed inside the SP and its new value is passed back to the calling program.
- It is the equivalent of the RETURN statement in a function

## **SP PARAMETERS - INOUT**

- INOUT – an INOUT parameter is the combination of IN parameter and OUT parameter.
- The calling program may pass the argument, and the stored procedure can modify the INOUT parameter and pass the new value back to the calling program.

## IN PARAMETER EXAMPLE

```
CREATE PROCEDURE GetOfficeByCountry
    (IN countryName VARCHAR(255))
    BEGIN
        SELECT * FROM offices
        WHERE country = countryName;
    END;
```

## OUT PARAMETER EXAMPLE

```
CREATE PROCEDURE CountOrderByStatus(
    IN orderStatus VARCHAR(25),
    OUT total INT)
    BEGIN
        SELECT count(orderNumber)
        INTO total
        FROM orders
        WHERE status = orderStatus;
    END;
```

## OUT PARAMETER EXAMPLE

To get the number of shipped orders, we call the CountOrderByStatus stored procedure and pass the order status as Shipped, and also pass an argument ( @total) to get the return value.

```
CALL CountOrderByStatus('Shipped',@total);  
SELECT @total;
```

## INOUT PARAMETER EXAMPLE

```
CREATE PROCEDURE set_counter  
    (INOUT count INT(4),  
    IN inc INT(4))  
BEGIN  
    SET count = count + inc;  
END;
```

## INOUT PARAMETER EXAMPLE

- The *set\_counter* SP accepts:
  - one INOUT parameter ( count)
  - one IN parameter ( inc).
- Inside the stored procedure, we increase the counter ( count) by the value of the inc parameter.

```
SET @counter = 1;  
CALL set_counter(@counter,1); -- 2  
CALL set_counter(@counter,1); -- 3  
CALL set_counter(@counter,5); -- 8  
SELECT @counter; -- 8
```

## Triggers

- Trigger is a set of SQL statements stored in the database catalog.
- Trigger is executed or fired whenever an event associated with a table occurs e.g., insert, update or delete.

## Triggers

- Trigger is a special type of stored procedure.
- It is not called directly like a stored procedure.
  - Trigger is called automatically when a data modification event is made against a table
  - SP must be called explicitly.

## Advantages of Triggers

- Triggers provide an alternative way to check the integrity of data.
- Can catch errors in business logic in the database layer.
- Provide an alternative way to run scheduled tasks.
- Very useful to audit the changes of data in tables.

## Disadvantages of Triggers

- Triggers only can provide an extended validation and they cannot replace all the validations.
  - Some simple validations have to be done in the application layer. For example, you can validate user's inputs in the client side by using JavaScript or in the server side using server side scripting languages such as JSP, PHP, ASP.NET, Perl, etc.
- SQL triggers are invoked and executed invisibly from client-applications therefore it is difficult to figure out what happen in the database layer.
- SQL triggers may increase the overhead of the database server.

## Trigger Activation

- BEFORE INSERT – activated before data is inserted into the table.
- AFTER INSERT – activated after data is inserted into the table.
- BEFORE UPDATE – activated before data in the table is updated.
- AFTER UPDATE – activated after data in the table is updated.
- BEFORE DELETE – activated before data is removed from the table.
- AFTER DELETE – activated after data is removed from the table.



## Create Trigger

```
CREATE TRIGGER trigger_name trigger_time trigger_event  
    ON table_name FOR EACH ROW  
BEGIN  
  
END
```

## Create Trigger

- You put the trigger name after the CREATE TRIGGER statement.
  - The trigger name should follow the naming convention [trigger time]\_[table name]\_[trigger event], for example before\_employees\_update.
- Trigger activation time can be BEFORE or AFTER.
  - Must specify the activation time when you define a trigger.
  - Use BEFORE keyword if you want to process action prior to the change is made on the table
  - Use AFTER if you need to process action after the change is made

## Create Trigger

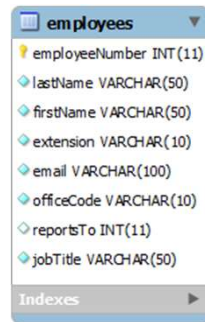
- Trigger event can be INSERT, UPDATE or DELETE.
- This event causes trigger to be invoked. A trigger only can be invoked by one event.
- To define a trigger that is invoked by multiple events, you have to define multiple triggers, one for each event.

## Create Trigger

- A trigger must be associated with a specific table.
- The SQL statements are placed between BEGIN and END block.

## Trigger Example

- We have employees table in our *classicmodels* database as follows:



The screenshot shows the 'employees' table structure in MySQL Workbench. The table has the following columns:

Column Name	Data Type
employeeNumber	INT(11)
lastName	VARCHAR(50)
firstName	VARCHAR(50)
extension	VARCHAR(10)
email	VARCHAR(100)
officeCode	VARCHAR(10)
reportsTo	INT(11)
jobTitle	VARCHAR(50)

Below the columns, there is an 'Indexes' section with a right-pointing arrow.

## Trigger Example

- Create a new table named *employees\_audit* to keep the changes of the employee records. The following script creates the *employee\_audit* table.

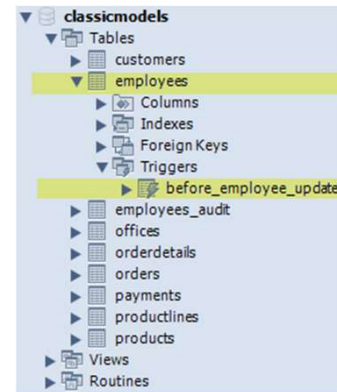
```
CREATE TABLE employees_audit (
  id int(11) NOT NULL AUTO_INCREMENT,
  employeeNumber int(11) NOT NULL,
  lastname varchar(50) NOT NULL,
  changedon datetime DEFAULT NULL,
  action varchar(50) DEFAULT NULL,
  PRIMARY KEY (id)
)
```

## Trigger Example

Create a BEFORE UPDATE trigger to be invoked before a change is made to the employees table.

```
DELIMITER $$
CREATE TRIGGER before_employee_update
BEFORE UPDATE ON employees
FOR EACH ROW BEGIN

INSERT INTO employees_audit
SET action = 'update',
employeeNumber = OLD.employeeNumber,
lastname = OLD.lastname,
changedon = NOW();
END$$
DELIMITER ;
```



## Trigger Example

*UPDATE employees SET lastName = 'Phan' WHERE employeeNumber = 1056*

To check if the trigger was invoked by the UPDATE statement, we can query the employees\_audit table by using the following query:

```
SELECT * FROM employees_audit;
```

	id	employeeNumber	lastname	changedon	action
▶	1	1056	Phan	2013-01-16 15:59:36	update

# Transactions

## Transactions

- Concurrent execution of user programs is essential for good DBMS performance.
  - Because disk accesses are frequent, and relatively slow, it is important to keep the cpu humming by working on several user programs concurrently.
- A user's program may carry out many operations on the data retrieved from the database, but the DBMS is only concerned about what data is read/written from/to the database.
- A transaction is the DBMS's abstract view of a user program: a sequence of reads and writes.

## Concurrency in a DBMS

- Users submit transactions, and can think of each transaction as executing by itself.
  - Concurrency is achieved by the DBMS, which interleaves actions (reads/writes of DB objects) of various transactions.
  - Each transaction must leave the database in a consistent state if the DB is consistent when the transaction begins.
    - DBMS will enforce some Integrity Constraints (ICs), depending on the ICs declared in CREATE TABLE statements.
    - Beyond this, the DBMS does not really understand the semantics of the data. (e.g., it does not understand how the interest on a bank account is computed).
- Issues: Effect of *interleaving* transactions, and *crashes*.

## Atomicity of Transactions

- A transaction might *commit* after completing all its actions, or it could *abort* (or be aborted by the DBMS) after executing some actions.
- A very important property guaranteed by the DBMS for all transactions is that they are atomic. That is, a user can think of a Xact as always executing all its actions in one step, or not executing any actions at all.
  - DBMS *logs* all actions so that it can *undo* the actions of aborted transactions.

## Example

- Consider two transactions (*Xacts*):

T1:	BEGIN	A=A+100,	B=B-100	END
T2:	BEGIN	A=1.06*A,	B=1.06*B	END

- ❖ Intuitively, the first transaction is transferring \$100 from B's account to A's account. The second is crediting both accounts with a 6% interest payment.
- ❖ There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together. However, the net effect *must* be equivalent to these two transactions running serially in some order.

## Example (Contd.)

- Consider a possible interleaving (*schedule*):

T1:	A=A+100,	B=B-100
T2:	A=1.06*A,	B=1.06*B

- ❖ This is OK. But what about:

T1:	A=A+100,	B=B-100
T2:	A=1.06*A, B=1.06*B	

- ❖ The DBMS's view of the second schedule:

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

## ACID Properties

- A transaction is a very small unit of a program and it may contain several low-level tasks.
- A transaction in a database system must maintain Atomicity, Consistency, Isolation, and Durability – commonly known as ACID properties – in order to ensure accuracy, completeness, and data integrity.

## Atomicity

- This property states that a transaction must be treated as an atomic unit, that is, either all of its operations are executed or none.
- There must be no state in a database where a transaction is left *partially completed*.
- States should be defined either before the execution of the transaction or after the execution/abortion/failure of the transaction.



## Consistency

- The database must remain in a consistent state after any transaction.
- No transaction should have any adverse effect on the data residing in the database.
- If the database was in a consistent state before the execution of a transaction, it must remain consistent after the execution of the transaction as well.

## Durability

- The database should be durable enough to hold all its latest updates even if the system fails or restarts.
- If a transaction updates a chunk of data in a database and commits, then the database will hold the modified data.
- If a transaction commits but the system fails before the data could be written on to the disk, then that data will be updated once the system springs back into action.

## Isolation

- In a database system where more than one transaction are being executed simultaneously and in parallel, the property of isolation states that all the transactions will be carried out and executed as if it is the only transaction in the system.
- No transaction will affect the existence of any other transaction.

## Scheduling Transactions

- When multiple transactions are being executed by the operating system in a multiprogramming environment, there are possibilities that instructions of one transactions are interleaved with some other transaction.

## Serial schedule

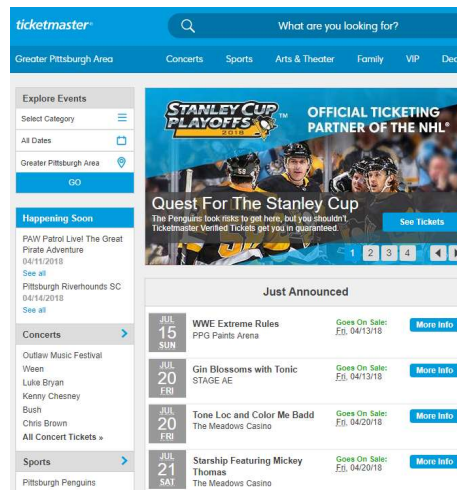
- Serial schedule: Schedule that does not interleave the actions of different transactions.
  - It is a schedule in which transactions are aligned in such a way that one transaction is executed first. When the first transaction completes its cycle, then the next transaction is executed. Transactions are ordered one after the other. This type of schedule is called a serial schedule, as transactions are executed in a serial manner

## Serializable schedule

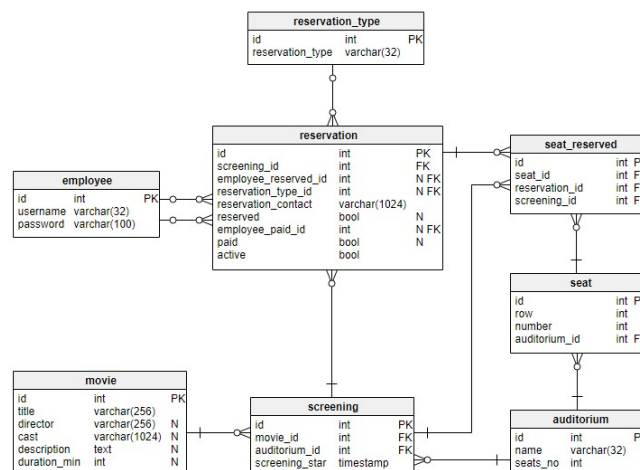
- Serializable schedule: A schedule that is equivalent to some serial execution of the transactions.
    - A schedule is called **serializable** whenever executing the transactions sequentially, in some order, could have left the database in the same state as the actual schedule.
- (Note: If each transaction preserves consistency, every serializable schedule preserves consistency. )

## Issue of Serial schedule

- Performance
- Cost



## Event-Booking System



## Event-Booking System

- Action Items
  - Login
  - Select Event
  - Browse Seats
  - Select Seats
  - Confirm Order
  - Payment
  - Send out Ticket by email
- What's the performance bottleneck?

## Scheduling Transactions: issue

- In a multi-transaction environment, serial schedules are considered as a benchmark. The execution sequence of an instruction in a transaction cannot be changed, but two transactions can have their instructions executed in a random fashion.
- This execution does no harm if two transactions are mutually independent and working on different segments of data; but in case these two transactions are working on the same data, then the results may vary.
- **This ever-varying result may bring the database to an inconsistent state.**
- To resolve this problem, we allow parallel execution of a transaction schedule, if its transactions are either serializable or have some equivalence relation among them.

## Equivalent schedules

- For any database state, the effect (on the set of objects in the database) of executing the first schedule is identical to the effect of executing the second schedule.
  - Result Equivalence
  - View Equivalence
  - Conflict Equivalence

## Result Equivalent

- If two schedules produce the same result after execution, they are said to be result equivalent.
- They may yield the same result for some value and different results for another set of values. That's why this equivalence is not generally considered significant.

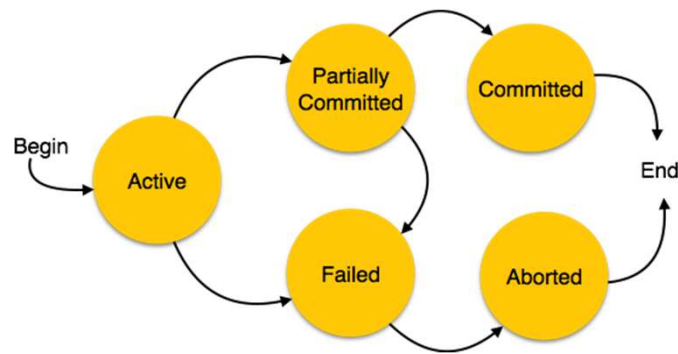
## View Equivalent

- if the transactions in both the schedules perform similar actions in a similar manner.
  - If T reads the initial data in S1, then it also reads the initial data in S2.
  - If T reads the value written by J in S1, then it also reads the value written by J in S2.
  - If T performs the final write on the data value in S1, then it also performs the final write on the data value in S2.

## Conflict Equivalent

- Two schedules would be conflicting if they have the following properties
  - Both belong to separate transactions.
  - Both accesses the same data item.
  - At least one of them is "write" operation.
- Two schedules having multiple transactions with conflicting operations are said to be conflict equivalent if and only if –
  - Both the schedules contain the same set of Transactions.
  - The order of conflicting pairs of operation is maintained in both the schedules.

## States of Transactions



## Aborting a Transaction

- If a transaction  $T_i$  is aborted, all its actions have to be undone. Not only that, if  $T_j$  reads an object last written by  $T_i$ ,  $T_j$  must be aborted as well!
- Most systems avoid such *cascading aborts* by releasing a transaction's locks only at commit time.
  - If  $T_i$  writes an object,  $T_j$  can read this only after  $T_i$  commits.
- In order to *undo* the actions of an aborted transaction, the DBMS maintains a *log* in which every write is recorded. This mechanism is also used to recover from system crashes: all active Xacts at the time of the crash are aborted when the system comes back up.



## Anomalies with Interleaved Execution

- Reading Uncommitted Data (WR Conflicts, “dirty reads”):

T1:	R(A), <b>W(A)</b> ,	R(B), W(B), Abort
T2:	<b>R(A)</b> , W(A), C	

- Unrepeatable Reads (RW Conflicts):

T1:	<b>R(A)</b> ,	<b>R(A)</b> , W(A), C
T2:	R(A), W(A), C	

## Anomalies (Continued)

- Overwriting Uncommitted Data (WW Conflicts):

T1:	<b>W(A)</b> ,	W(B), C
T2:	<b>W(A)</b> , W(B), C	

## Conflict Serializable Schedules

- Two schedules are **conflict equivalent** if:
  - Involve the same actions of the same transactions
  - Every pair of conflicting actions is ordered the same way
- Schedule S is **conflict serializable** if S is conflict equivalent to some serial schedule

## Example

- Is the schedule conflict serializable?

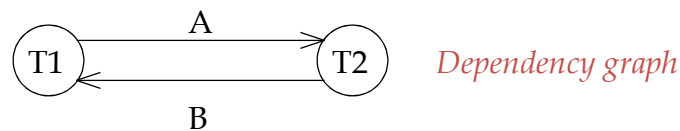
S1:  $R_1(X) R_1(Y) R_2(X) R_2(Y) W_2(Y) W_1(X)$

S2:  $R_1(X) R_2(X) R_2(Y) W_2(Y) R_1(Y) W_1(X)$

## Example

- A schedule that is not conflict serializable:

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



- The cycle in the graph reveals the problem. The output of T1 depends on T2, and vice-versa.

## Dependency Graph

- Dependency graph: One node per Xact; edge from  $T_i$  to  $T_j$  is created if one of the operations in  $T_i$  appears in the schedule before some conflicting operation in  $T_j$ .
- Theorem: Schedule is conflict serializable if and only if its dependency graph is acyclic

## Two-Phase Locking (2PL)

- Each Xact must obtain a S (*shared*) lock on object before reading, and an X (*exclusive*) lock on object before writing.
- A transaction can not request additional locks once it releases any locks.
- If an Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object.

## Strict 2PL

- Each Xact must obtain a S (*shared*) lock on object before reading, and an X (*exclusive*) lock on object before writing.
- All locks held by a transaction are released when the transaction completes
- If an Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object.

Strict 2PL allows only serializable schedules

## Lock Management

- Lock and unlock requests are handled by the lock manager
- Lock table entry:
  - Number of transactions currently holding a lock
  - Type of lock held (shared or exclusive)
  - Pointer to queue of lock requests
- Locking and unlocking have to be atomic operations
- Lock upgrade: transaction that holds a shared lock can be upgraded to hold an exclusive lock

## Deadlocks

- Deadlock: Cycle of transactions waiting for locks to be released by each other.
- Two ways of dealing with deadlocks:
  - Deadlock prevention
  - Deadlock detection

## Deadlock Prevention

- Assign priorities based on timestamps. Assume  $T_i$  wants a lock that  $T_j$  holds. Two policies are possible:
  - Wait-Die: If  $T_i$  has higher priority,  $T_i$  waits for  $T_j$ ; otherwise  $T_i$  aborts
  - Wound-wait: If  $T_i$  has higher priority,  $T_j$  aborts; otherwise  $T_i$  waits
- If a transaction re-starts, make sure it has its original timestamp

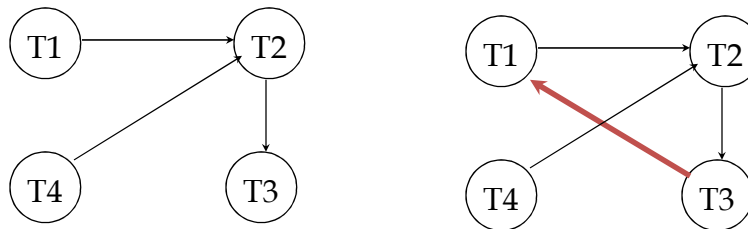
## Deadlock Detection

- Create a **waits-for graph**:
  - Nodes are transactions
  - There is an edge from  $T_i$  to  $T_j$  if  $T_i$  is waiting for  $T_j$  to release a lock
- Periodically check for cycles in the waits-for graph

## Deadlock Detection (Continued)

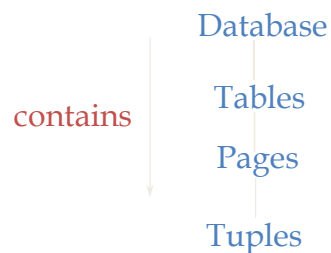
Example:

T1: S(A), R(A), S(B)  
 T2: X(B), W(B) X(C)  
 T3: S(C), R(C) X(A)  
 T4: X(B)



## Multiple-Granularity Locks

- Hard to decide what granularity to lock (tuples vs. pages vs. tables).
- Shouldn't have to decide!
- Data "containers" are nested:



## Summary

- There are several lock-based concurrency control schemes (Strict 2PL, 2PL). Conflicts between transactions can be detected in the dependency graph
- The lock manager keeps track of the locks issued. Deadlocks can either be prevented or detected.

- Questions?