MPCS 53001: Databases

Zach Freeman

Lecture #7

Class Announcements

- You did a midterm last week. Nice job.
  - Tests returned at end of class (& grades posted to Canvas).
- Assignment 7
  - Triggers and Stored Procedures! (programming)
  - Gradiance is back. ⏪

Overview For Today (part 1)

- Review of database constraints
- MySQL triggers
  - Enforcing constraints using triggers
- DB application programming
  - MySQL routines (stored procedures)

Overview For Today (part 2)

- Transaction Management
  - Transactions
  - Properties
  - Deadlocks
  - Isolation Levels
- Authorization
  - Granting and revoking privileges

Constraints

- Restrictions on the data in your database
- E/R diagrams support a limited number of restrictions: keys, one-one, many-one, relationships.
- DBMS allow much more fine-tuning of constraints
- Enforce all constraints at the database level

Enforcing Constraints

- Browsers can enforce constraints via javaScript, jQuery, etc...
  - (password strength, required fields, etc.)
- Applications *should* enforce (some) constraints if done efficiently
- DBMS is the last line of defense

Constraint Types

- Primary key declaration
- Foreign keys (referential integrity)
- Attribute and tuple based checks
  - Within a relation
- SQL assertions (global constraints)

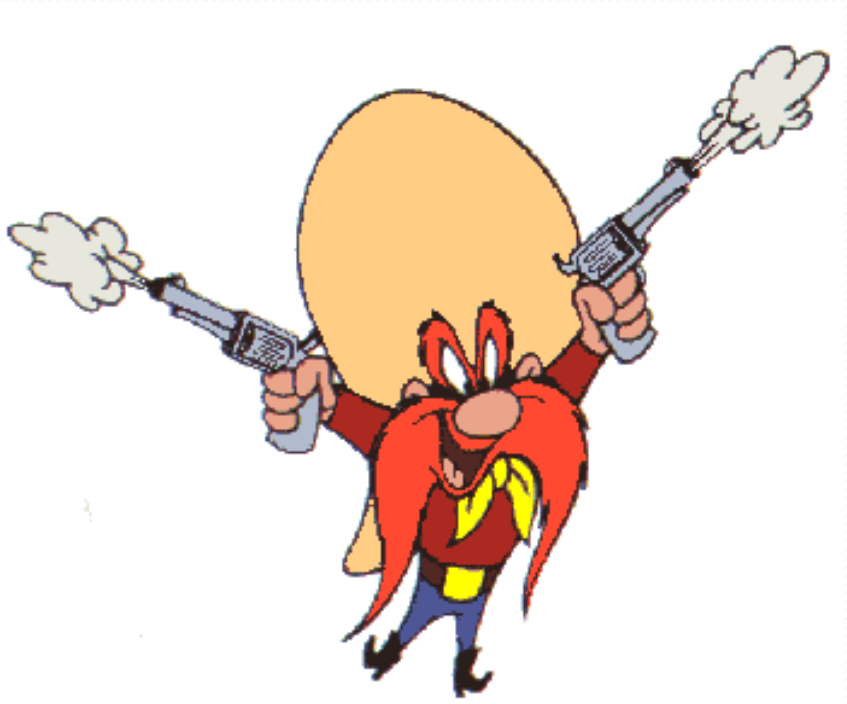
- **MySQL Triggers**

What Are Triggers?

- “Blocks of code executed when a specific database event occurs.” – Beginning Databases
- Event-driven mechanism for enforcing constraints with user-defined actions.
- When certain types of events occur in the DBMS, pre-defined actions are executed (*triggered!*).
  - The actions may counteract the triggering events.

Triggers Are…

- Tied to specific tables
- Tied to specific actions on that table
- Set to happen either before OR after the specified ac



MySQL Trigger Syntax

```
CREATE TRIGGER name
    { BEFORE | AFTER }
    { INSERT | UPDATE | DELETE }
    ON relation
    FOR EACH ROW
    [SQL statements]
```

Example Trigger

- Whenever we insert a new tuple into Registrations, check to see if the runner mentioned appears in Runner and insert it (with null info) if not.

```
DELIMITER |
CREATE TRIGGER RunnerExistsTrigger
AFTER INSERT ON Registrations
FOR EACH ROW
BEGIN
INSERT IGNORE INTO Runner(runnerName)
VALUES(NEW.runnerName);
END; |
DELIMITER ;
```

Timing Options

- BEFORE
- AFTER

- BEFORE triggers **can** change the value of the inserted or updated tuple (AFTER triggers **cannot**)

Event Options

- INSERT, DELETE, UPDATE - specify the event on the relation that activates the trigger.

- Triggers can act after *EVERY* tuple change or after *ALL* tuple changes

- FOR EACH ROW means act for every tuple (only available option in MySQL)

Special Variables

- Two special variables available with triggers: old and new
- old is read only, new can be changed.
- INSERT: only new is available.
- DELETE: only old is available.
- UPDATE: **both** available.

Changing the Delimiter

- For parsing of the trigger code the default delimiter (“;”) needs to be redefined temporarily to another character (e.g. “|”)

```
DELIMITER |
<TRIGGER CODE>; |
DELIMITER ;
```

Trigger Actions

- The action can be one or more SQL statements surrounded by BEGIN and END (with some restrictions).

Action Restrictions

- Trigger actions cannot change the **relation** that triggers the action *except for the **tuple** that triggered the action*.
  - Other relations can be modified.

- Why is this restriction in place?

Revisiting Example Trigger

- Whenever we insert a new tuple into Registrations, check to see if the runner mentioned appears in Runner and insert it (with null info) if not.

```
DELIMITER |
CREATE TRIGGER RunnerExistsTrigger
AFTER INSERT ON Registrations
FOR EACH ROW
BEGIN
INSERT IGNORE INTO Runner(runnerName)
VALUES(NEW.runnerName);
END; |
DELIMITER ;
```

Find all triggers

- MySQL: SHOW TRIGGERS
- SQL Server: SELECT * FROM sys.triggers

Sneaky Runners

- Maintain a relation of all runners whose results are lowered by more than 5 places:

FishyResultsChanges(fishyResultsChangesID, oldRunnerName, oldRaceName, oldPlace, newPlace, userMakingChange, dateChanged)

```
DELIMITER |
CREATE Trigger ResultsChangesTrigger
AFTER UPDATE ON Results
FOR EACH ROW
BEGIN
    IF (NEW.place < OLD.place - 5) THEN
        INSERT IGNORE INTO FishyResultsChanges
VALUES(NULL, OLD.runnerName, OLD.raceName, OLD.place,
NEW.place, CURRENT_USER(), NOW());
    END IF;
END; |
```

FLASHBACK: Attribute-Based Checks

- Defined on an attribute by a condition that must hold for attribute values.

  CHECK(condition)

  - Condition may involve the checked attribute
  - Other attributes or relations may be involved but only in a subquery.

- *The CHECK clause is parsed but ignored by all storage engines in MySQL.*

Attribute Checks with Triggers

- Create two triggers: BEFORE INSERT and BEFORE UPDATE

- The triggers check the attribute constraints and if they’re not satisfied they make a modification of the tuple that will be rejected, so the triggering INSERT or UPDATE fails.

Cap Check – INSERT Trigger

- Check that the registration cap of a race is less than 50,000.

```
CREATE TRIGGER RegistrationCapCheckInsertTrigger
    BEFORE INSERT ON Race
    FOR EACH ROW
    BEGIN
    IF (NEW.registrationCap > 50000) THEN
    SET NEW.registrationCap = NULL;
    END IF;
    END;
```

Cap Check – UPDATE Trigger

- Check that the registration cap of a race is less than 50,000.

```
CREATE TRIGGER RegistrationCapCheckUpdateTrigger
    BEFORE UPDATE ON Race
    FOR EACH ROW
    BEGIN
    IF (NEW.registrationCap > 50000) THEN   SET
NEW.registrationCap = OLD.registrationCap;
    END IF;
    END;
```

DB Application Programming

- Applications are written in general purpose languages: C#, Java, PHP (**not SQL**).
- DB queries are application-driven:
  - User registers, buys product, submits form, etc.
- Need to connect application actions to DB

Interface Solutions

- Execute queries within application code: embedded SQL
- Can extend SQL with general-purpose programming:
  - Persistent Stored Modules (PSM), called routines in MySQL (stored procedures in MS SQL)

MySQL Routines

- MySQL's version of PSM
  - Stored procedures
  - Functions

Procedure Syntax

```
CREATE PROCEDURE [name](argument list)
BEGIN
    declarations
    statements
END;
```

Function Syntax

```
CREATE FUNCTION [name](argument list)
RETURNS type
BEGIN
    declarations
    statements
END;
```

Parameters

- Can be IN, OUT or INOUT
- Types: standard SQL data types

Procedure Example

- A procedure to add a runner to the results table:

```
CREATE PROCEDURE addRunnerToResults(
    IN addRace    VARCHAR(50),
    IN addRunner     VARCHAR(50),
    IN addPlace    INT)
BEGIN
    INSERT INTO Results (raceName, runnerName,
place)
    VALUES(addRace,addRunner,addPlace);
END |
```

Procedure Calls

- Invoking procedures is similar to other programming languages but prefaced with CALL

```
CALL addRunnerToResults('Shamrock Shuffle','Ryan Hall',10);

CALL addRunnerToResults('Frozen Gnome','Tera Moody',5);
```

Declarations

- Variables
- Conditions
- Cursors
- Handlers

- Must be declared in this particular order.

Variables

Syntax:

DECLARE variableName dataType(size) DEFAULT defaultValue

Example:

DECLARE raceCountry CHAR(3) DEFAULT 'USA';

- Can set with SET or SELECT INTO

Procedure Example (part 1)

- A procedure to add a runner to the results table:

```
CREATE PROCEDURE addRunnerToResults(
    IN addRace    VARCHAR(50),
    IN addRunner     VARCHAR(50),
    IN addPlace    INT)
BEGIN
    DECLARE runnerAgeGroupCode CHAR(4);
    DECLARE runnerAge INT;
```

Procedure Example (part 2)

```
    SET runnerAge = (SELECT age
    FROM Runner
    WHERE runnerName = addRunner);
    SET runnerAgeGroupCode = (SELECT ageGroupCode
    FROM AgeGroup
    WHERE runnerAge BETWEEN ageGroupMin  AND
ageGroupMax);
    INSERT INTO Results

VALUES(addRace,addRunner,addPlace,runnerAgeGroupCode);
END |
```

Conditions

```
DECLARE conditionName
    CONDITION FOR SQLSTATE errorStr

DECLARE conditionName
    CONDITION FOR errorNumber
```

- The following conditions are predefined:
  - NOT FOUND (no more rows)
  - SQLEXCEPTION (run-time error)
  - SQLWARNING(warning)

Handlers

- Declare what to do in case of errors (or conditions)

DECLARE {EXIT | CONTINUE}
    HANDLER FOR
{errorNum | SQLSTATE errorStr | conditionName}
SQL statement

- Common practice: set a flag for CONTINUE handlers and check inside stored procedure

Body Constructs

- Assignments

SET var = expr;

  - Variables must be declared

- Branches

IF <condition> THEN
<statements>
ELSE
<statements>
END IF;

Queries in Routines

- Single-row selects allow retrieval into a variable of the result of a query that is guaranteed to produce one tuple.

- Cursors allow the retrieval of many tuples with the cursor and a loop used to process each in turn.

Cursors in MySQL

- The cursor declaration is:

```
DECLARE cursorName
    CURSOR FOR query;
```

- Fetching is done with:

```
FETCH cursorName INTO variables;
```

Procedure Example 1/4

- If a runner is found to have cheated – every position after his/her place should be moved up.

```
DELIMITER |
CREATE PROCEDURE RecalibrateResults(
IN RaceToRecalibrate  VARCHAR(50),
IN DroppedRunnerPlace  INT)
BEGIN
    DECLARE aRunner  VARCHAR(50);
    DECLARE aRace     VARCHAR(50);
    DECLARE aPlaceINT;
    DECLARE flagINT DEFAULT 0;
```

Procedure Example 2/4

```
DECLARE ResultsAfterDroppedRunner CURSOR
FOR
    SELECT runnerName, racename, place
    FROM Results
    WHERE raceName = RaceToRecalibrate
    AND place > DroppedRunnerPlace;
DECLARE CONTINUE HANDLER
    FOR NOT FOUND
    SET flag = 1;
OPEN ResultsAfterDroppedRunner;
```

Procedure Example 3/4

```
REPEAT
    FETCH ResultsAfterDroppedRunner
    INTO aRunner, aRace, aPlace;
    IF aPlace > DroppedRunnerPlace THEN    UPDATE Results
    SET place = aPlace - 1        WHERE raceName = aRace
    AND runnerName = aRunner;
    END IF;
    UNTIL flag = 1
END REPEAT;
CLOSE ResultsAfterDroppedRunner;
END |
```

Procedure Example 4/4

- What happens now?
- The procedure has only been defined, not invoked.

```
CALL RecalibrateResults([race],
[droppedRunnerPlace]);

CALL RecalibrateResults('NYC Marathon',4);
```

Questions?

What is a Transaction?

- A transaction is simply **a block of SQL statements** treated as a unit.
  - “A unit of work that may include multiple activities that query and modify data and that possibly change data definition.” – Microsoft T-SQL Fundamentals, 2009
  - These can include SELECT, INSERT, UPDATE and DELETE statements, among others.

About Transactions

- Transaction: A block of SQL statements
- ACID
- COMMIT
- ROLLBACK
- SAVEPOINT
- LOCK (and DEADLOCK)
- ISOLATION LEVELS

Why Transactions?

- Databases are usually being accessed by multiple users at once, processing both queries and modifications (INSERT, UPDATE, DELETE, etc.)

- Transactions help prevent troublesome interactions between these various processes.

Why Transactions?

- “Troublesome interactions”
- Example: Two people withdraw $100 from the same account at different ATMs at almost the same time. Database better not lose either deduction.
- Multiple SQL statements are tied together and must all be executed for a valid result (e.g. new customer created, new order created, new items added to order...)

Example of a Transaction

```
START TRANSACTION

UPDATE Results
SET place = place - 1
WHERE runnerName != 'Rita Jeptoo'
AND raceName = 'Chicago Marathon'

DELETE FROM Results
WHERE name = 'Rita Jeptoo'

COMMIT
```

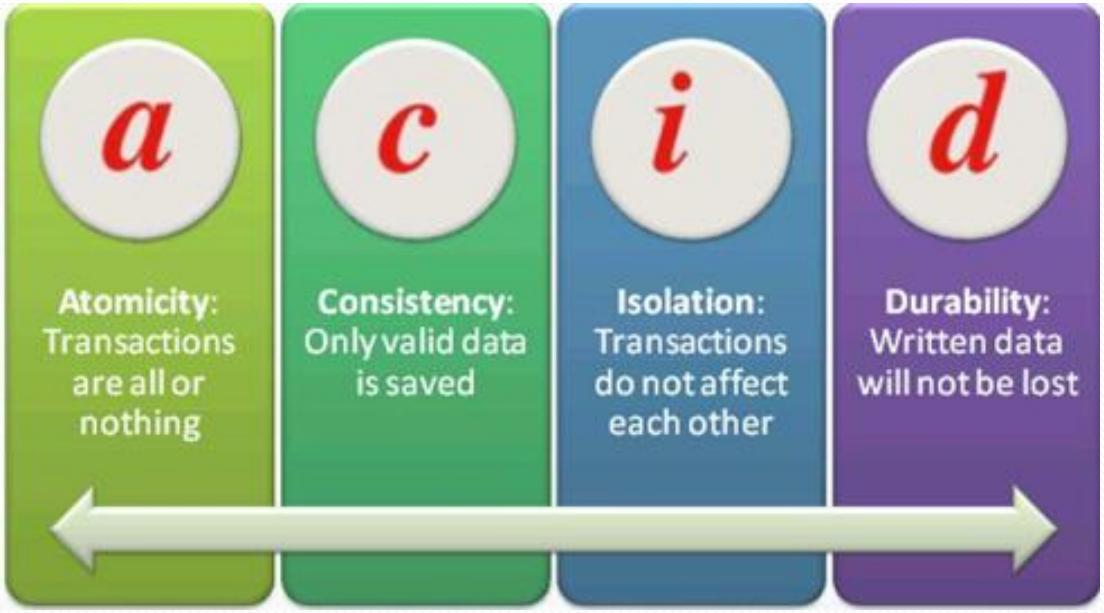
Transaction starts

1st SQL statement

2nd SQL statement

Transaction ends

ACID



*Andreas Reuter and Theo Harder created this acronym in 1983 to describe properties of a reliable transaction as described by Jim Gray in the late 1970s.

A is for Atomicity

- Saying that a transaction is Atomic means that **either all the changes in the transaction take place or none do**.
- A transaction is an atomic unit of work.
- Example: in a transaction with three statements, two of them execute and then an error occurs. What happens?

C is for Consistency

- Saying that a transaction is Consistent means that **database constraints are preserved and only valid data (according to all defined rules) is saved**.
- Example: Results has a Foreign Key constraint on the “ageGroup” field (it must be a value in the AgeGroup table)

INSERT INTO Results

VALUES ('Athens Marathon', 'Kara Goucher',

I is for Isolation

- Saying that a transaction is Isolated means that **access to data is controlled so that transactions do not affect each other**.

- Example: UPDATE happening simultaneously with a SELECT will not corrupt the result set.

D is for Durability

- Saying that a transaction is durable means that **all updates to the database will be stored permanently**.
- If the database loses power immediately after a data change is executed, the changes will be there after a restart.

COMMIT

- The SQL COMMIT statement causes a transaction to complete, **making the database modifications permanent in the database**.

- Two types of COMMIT:
  - Implicit
  - Explicit

COMMIT - Implicit

- Implicit Commit: by default, individual statements are treated as unique transactions.
- Essentially, in AUTO COMMIT mode (default), a transaction automatically begins when a SQL command is issued and commits when the statement ends.

- Example:

```
DELETE FROM Race
WHERE raceName = 'Athens Marathon';
```

COMMIT - Explicit

- SQL Server explicit commit:

```
START TRANSACTION;
UPDATE Race
SET distance = 31
WHERE raceName = 'Frozen Gnome';
    DELETE FROM Runner
    WHERE runnerName = 'Zach Freeman';
COMMIT;
```

ROLLBACK

- A ROLLBACK is **the process of undoing specified SQL statements**.
- The ROLLBACK statement also causes a transaction to complete, but by aborting the operation(s).

```
START TRANSACTION;
    UPDATE Race
    SET raceName = 'Chicago Marathron'
    WHERE raceName = 'Chicago Marathon';

    DELETE FROM Likes
    WHERE raceName = 'Chicago Marathron'
    AND runnerName = 'Meb Keflezighi';
ROLLBACK;
```

ROLLBACK - Implicit

- Implicit Rollback: if an error (such as a constraint violation) occurs while executing a transaction the transaction will rollback automatically

```
START TRANSACTION;
    UPDATE Likes
    SET runnerName = 'Meb Keflezaggah'
    WHERE runnerName = 'Meb Keflezighi';
COMMIT;
```

SAVEPOINT

- A SAVEPOINT is a temporary placeholder in a transaction that allows you to issue a rollback to a set point (rather than rolling back the entire transaction)
- You can have multiple savepoints in one transaction.

Why Transactions?

- Bill wants to find out which runner in the database has been running the longest as well as which runner has been running the least amount of years.
- Bill executes two queries to get this information (MAX and MIN).
- At the same time Zach decides to update his very outdated database to reflect new runner years running values.

Why Transactions?

Bill is executing these two queries:

MAX: SELECT MAX(yearsRunning) AS 'MostYears'
FROM Runner;

MIN: SELECT MIN(yearsRunning) AS 'LeastYears'
FROM Runner;

Why Transactions?

- At the same time Zach is updating the database.

UPDATE:

```
UPDATE Runner
SET yearsRunning = yearsRunning + 10;
```

Why Transactions?

- What if the steps execute in this order:

  MAX

  UPDATE

  MIN

Bill has a MAX of 25

And a MIN of 26

MIN > MAX!

Transactions fix this problem

- If Bill runs his query as a transaction he won’t see this inconsistency. He’ll either see the data BEFORE Zach makes the update or AFTER. Either way the MAX and MIN he gets will be from the same dataset.

[MAX, MIN], UPDATE

UPDATE, [MAX, MIN]

But wait there's more...

- Interleaving Example in transactions...

T1	T2	A (disk)	A (T1)	A (T2)
READ A		5	5	-
	READ A	5	5	5
A = A + 10		5	15	5
	A = 2 * A	5	15	10
	WRITE A	10	15	10
WRITE A		15	15	-

LOCKS

- Locks guard data resources and prevent conflicting or incompatible access by other transactions.
- There are two main lock modes:
  - Exclusive and Shared.
- Modifying data requires an Exclusive Lock, whereas querying requires a Shared Lock.

LOCKS

- Multiple users can query at the same time and all have SHARED locks, but only one user at a time can have an EXCLUSIVE lock.

LOCKS

Lock type	Statement type
SHARED	SELECT
EXCLUSIVE	INSERT, UPDATE, DELETE

	READ	WRITE
READ	⏪	X
WRITE	X	X

Interleaving with Locks

T1	T2	A (disk)	A (T1)	A (T2)
WLOCK A		5	-	-
READ A		5	5	-
	WLOCK A	5	5	-
A = A + 10	Waits…	5	15	-
WRITE A		15	15	-
UNLOCK A		15	-	-
	READ A	15	-	15
	A = 2 * A	15	-	30
	WRITE A	30	-	30
	UNLOCK A	30	-	-

DEADLOCK

- “A deadlock occurs when two or more tasks permanently block each other by each task having a lock on a resource which the other tasks are trying to lock.”

Deadlock with Locks

T1	T2	A (disk)	A (T1)	A (T2)
RLOCK A		5	-	-
READ A		5	5	-
	RLOCK A	5	5	-
	READ A	5	5	5
A = A + 10		5	15	5
	A = 2 * A	5	15	10
	WLOCK A	5	15	10
WLOCK A	Wait…	5	15	10
Wait…		5	15	10
DEADLOCK		THE	WORLD	ENDS!

Deadlock Conditions

1. Hold some locks while waiting for other locks.
2. Circular chain of waiters in the wait-for-graph.
3. No pre-emption.

- Systems avoid deadlock by doing at least one of these:
  - Get all of your locks at once.
  - Apply an ordering to acquiring locks.
  - Allow pre-emption (timeout on waits).

Isolation Levels

- Isolation Levels determine the behavior of concurrent users reading and writing data.
- Reader: any statement selecting data (shared lock)
- Writer: any statement modifying data (exclusive lock)

- You can only manage the Isolation Level for readers.

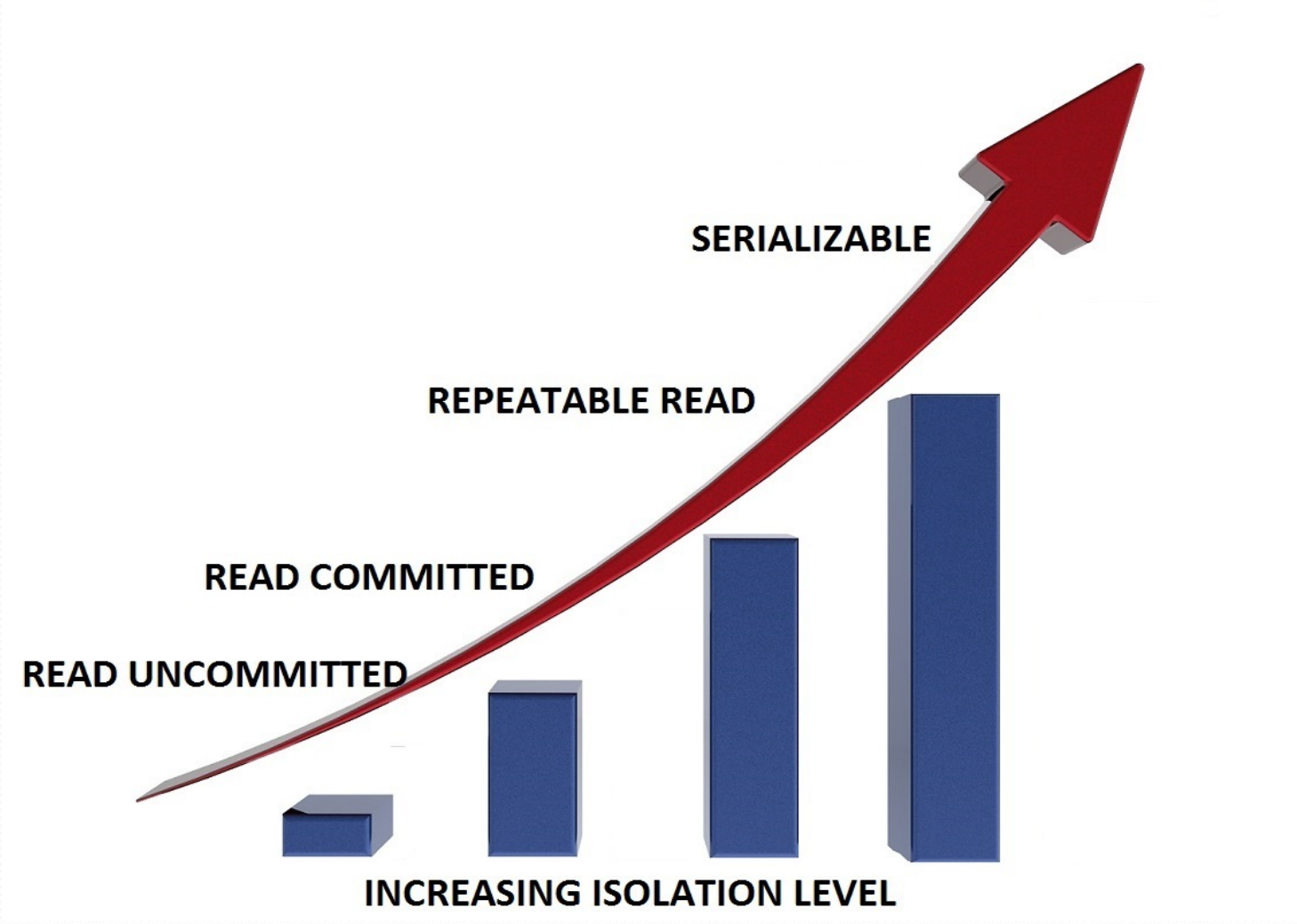
Isolation Levels

- Six Isolation Levels in SQL Server (essentially):

  1. READ UNCOMMITTED
  2. READ COMMITTED (default)
  3. REPEATABLE READ
  4. SERIALIZABLE (only one that is ACID)

- Two others:
  - (SNAPSHOT, READ COMMITTED SNAPSHOT)

Isolation Levels



Isolation Level: READ UNCOMMITTED

- Setting your isolation level to READ UNCOMMITTED means that when you execute a query you can read data that has not yet been committed (and may never be committed). This can result in what is known as a “dirty read.”

READ UNCOMMITTED example

- If Bill was running his MIN/MAX transaction under READ UNCOMMITTED he might see new yearsRunning even if Zach never committed them (and later rolled them back).

Isolation Level: READ COMMITTED

- Setting your isolation level to READ COMMITTED (which is the default in SQL Server) means that when you execute a query you only read data that has been committed.

- HOWEVER – based on our previous example with Zach and Bill, Bill might still end up with MIN > MAX.
- Zach's transaction could COMMIT between Bill's two queries because the shared lock that Bill's query requests is relinquished between the two statements – allowing Zach to make an update.

Isolation Level: REPEATABLE READ

- Similar to READ COMMITTED, but adds the constraint that data that has been read once will not be altered and will be read again in its same form (even if the data has subsequently been deleted).
- Any new rows will also be read.

- In previous example, if Bill runs MAX and then the update happens, he will see same data for MIN (but could also see new rows)

Isolation Level: SERIALIZABLE

- Only ACID isolation level
- Similar to REPEATABLE READ, but also locks all current *and future* rows in the query.
- If new data is added during transaction, it will not be included in the result set. So your entire transaction is processed with the same data set.

- Example: Bill will use the same data set for both his MAX and MIN queries.

About Transactions

- Transaction: A block of SQL statements
- ACID
- COMMIT
- ROLLBACK
- SAVEPOINT
- LOCK
- ISOLATION LEVELS

Authorization in SQL

- In SQL, there are several access privileges on relations. The most important ones are:
  1. SELECT: query the relation
  2. INSERT: insert tuples into the relation
  3. UPDATE: update tuples in the relation
  4. DELETE: delete tuples from the relation
  5. ALTER: altering the structure of a relation
- INSERT and UPDATE may refer to a specific attribute.

Granting Privileges

- Creator has all privileges on the relations they create.
- Users may grant privileges to any user if they have those privileges "with grant option."

Grant Example

```
GRANT SELECT ON Results,
UPDATE ON Results
TO Kathryn;

GRANT SELECT ON Results,
DELETE ON Results
TO Nirajan
WITH GRANT OPTION;
```

Revoking Privileges

- Syntax is similar to granting REVOKE… FROM instead of GRANT… TO
- Determining privileges involves “grant diagrams”
- Basic principles:
  - If you have been granted a privilege by several different users, then all of them have to revoke it for you to lose it.
  - Revocation is transitive.

Role-based Access

- Users assigned to roles with predefined privileges.
- GRANT/REVOKE used in the same way but with a role rather than an individual user.

```
GRANT developer
TO jk;
```

Summary

- Transaction Management
  - Transactions
  - Properties
  - Deadlocks
  - Isolation Levels
- Authorization
  - Granting and revoking privileges

Trigger/Procedure Recap

- Review of Database constraints
- MySQL triggers
  - Enforcing constraints using triggers
- DB application programming
- MySQL routines (stored procedures)
  - Cursors
  - Handlers

- Next week: PHP

Homework 7

- Triggers
- Stored Procedures
- Queries showing these work

Be sure to include CREATE/POPULATE scripts with your homework 7 submission!

Questions?