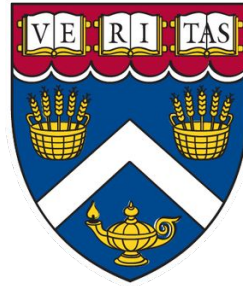


# CSCI E-59 Designing & Developing Relational and Non-Relational Databases

**Harvard University Extension, Spring 2022**

Greg Misicko



Lecture 10 - Transactions, Triggers, Security & Account Management

@GregMisicko

# Agenda

- Stored Routines
- Transactions
- Triggers
- Security
- Account Management

Next section with **Anthony: Monday April 11 at 8pm ET.**

Homework 5 is due April 8, and a new assignment will be released on April 8

# Stored Functions

Remember a few lectures ago we talked about functions. In SQL a built-in function is a piece of programming that takes zero or more inputs and returns a value. We used functions such as AVG, COUNT, MAX, and others.

We have the ability to write and store our own custom functions as well.  
The basic format is:

```
DELIMITER //
```

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

```
RETURN <value of type datatype>;  
DELIMITER ;
```

# Routines vs Procedures vs Functions

That sounds a lot like the Stored Procedures we just covered. Isn't it the same thing?

We have three terms to clarify:

**Routines:** A routine is just a piece of code written specifically for a database. A routine refers to either a procedure or a function, and it will only be usable within the database it was added to. When/if that database is dropped, those routines are dropped with it.

**Procedures:** A block of code that is invoked using a CALL statement. One or more values can only be returned through the use of output variables. Typically used for running business logic.

**Functions:** Functions can be embedded within a statement (just like the other functions we've reviewed) and always returns a single value. Typically used for computations.

<https://dev.mysql.com/doc/refman/8.0/en/stored-routines-syntax.html>

# Stored Functions: Deterministic / Non-Deterministic

<https://dev.mysql.com/doc/refman/8.0/en/create-procedure.html>

A routine is considered “deterministic” if it always produces the same result for the same input parameters, and “not deterministic” otherwise. If neither DETERMINISTIC nor NOT DETERMINISTIC is given in the routine definition, the default is NOT DETERMINISTIC. To declare that a function is deterministic, you must specify DETERMINISTIC explicitly.

Assessment of the nature of a routine is based on the “honesty” of the creator: MySQL does not check that a routine declared DETERMINISTIC is free of statements that produce nondeterministic results. However, misdeclaring a routine might affect results or affect performance. Declaring a nondeterministic routine as DETERMINISTIC might lead to unexpected results by causing the optimizer to make incorrect execution plan choices. Declaring a deterministic routine as NONDETERMINISTIC might diminish performance by causing available optimizations not to be used.

A routine that contains the NOW() function (or its synonyms) or RAND() is nondeterministic, but it might still be replication-safe. For NOW(), the binary log includes the timestamp and replicates correctly. RAND() also replicates correctly as long as it is called only a single time during the execution of a routine. (You can consider the routine execution timestamp and random number seed as implicit inputs that are identical on the primary and secondary.)

# Stored Functions

As an example, let's write a little function to associate a letter grade with the GPA value stored for a student in our good old 'school' database:

```
DELIMITER //
CREATE FUNCTION letterGrade(gpa_value decimal(2,1))

RETURNS VARCHAR(1) DETERMINISTIC

BEGIN
    DECLARE letterGrade VARCHAR(1);

    IF gpa_value = 4.0 THEN SET letterGrade = 'A';
    ELSEIF gpa_value < 4.0 AND gpa_value > 3.0 THEN SET letterGrade = 'B';
    ELSEIF gpa_value <= 3.0 THEN SET letterGrade = 'C';
    END IF;
    RETURN (letterGrade);
END //

DELIMITER ;
```

# Stored Functions

Now that we've defined a function, we can run it just as we would any of the other functions included in MySQL.

For example:

```
select first_name, LetterGrade(gpa) from students;
```

And it can be executed in the same way from within a stored procedure.

# Transactions

A transaction is a logical unit of work that must be completed in its entirety, or entirely aborted. For example:

Let's say I have a savings account and a checking account in a bank. I go online to transfer money from my savings to my checking so that I can write out a check to pay for something. This would be considered to be a single bank transaction, but there are multiple database operations involved.

**First**, I need to deduct the amount of money I wish to transfer from my savings account assuming it is available. **Second**, I need to add that money to my checking account. What would happen if one of those steps failed?



# Transactions

When many transactions take place at the same time, they are called **concurrency transactions**. Managing the execution of such transactions is called **concurrency control**.

When would a transaction need to be rolled back?

Typical types of failures:

- **transaction failure:** where the transaction cannot be completed successfully because of bad data, uninitialized variables, bad SQL statements, etc
- **system failure:** the hardware or software fails due to a bug, a power outage, or some other kind of glitch
- **media failure:** the hard drive or other storage (such as flash memory) has an issue

In the case of a system or media failure, we would want to try to perform a recovery of our database.

# ACID

A - Atomicity

All or Nothing Transactions

C - Consistency

Guarantees Committed Transaction State

I - Isolation

Transactions are Independent

D - Durability

Committed Data is Never Lost

(c) <http://blog.sqlauthority.com>

# Transactions: ACID

In computer science, **ACID** (Atomicity, Consistency, Isolation, Durability) is a set of properties of database transactions intended to guarantee validity even in the event of errors, power failures, etc.

(<https://en.wikipedia.org/wiki/ACID>)

**Atomicity:** A database follows the all or nothing rule, i.e., the database considers all transaction operations as one whole unit or atom. Thus, when a database processes a transaction, it is either fully completed or not executed at all. If those operations are partially executed before something fails, a rollback must occur.

# Transactions: ACID

**Consistency:** This SQL ACID property ensures database consistency. It means, whatever happens in the middle of the transaction, this property will never leave your database in a half-completed state.

If the transaction completed successfully, then it will apply all the changes to the database.

If there is an error in a transaction, then all the changes that already made will be rolled back automatically. It means the database will restore to its state that it had before the transaction started.

If there is a system failure in the middle of the transaction, then also, all the changes made already will automatically rollback.

<https://www.tutorialgateway.org/acid-properties-in-sql-server/>

# continued...

**Isolation:** Ensures that transactions are securely and independently processed at the same time without interference, but it does not ensure the order of transactions. For example, user A withdraws \$100 and user B withdraws \$250 from user Z's account, which has a balance of \$1000. Since both A and B draw from Z's account, one of the users is required to wait until the other user transaction is completed, avoiding inconsistent data. If B is required to wait, then B must wait until A's transaction is completed, and Z's account balance changes to \$900. Now, B can withdraw \$250 from this \$900 balance.

**Durability:** Once the transaction completed, then the changes it has made to the database will be permanent. Even if there is a system failure, or any abnormal changes also, this SQL acid property will safeguard the committed data.

<https://www.techopedia.com/definition/23949/atomicity-consistency-isolation-durability-acid>

# Transactions

So far we've always committed our changes immediately to the database. By default MySQL is configured to do this for us; every SQL statement we run is permanent.

When implementing transactions, we don't want to commit our changes until the transaction is finished. To start with, we'll want to turn autocommit to off:

```
SET AUTOCOMMIT = '0';
```

or

```
SET AUTOCOMMIT = OFF;
```

(You can probably guess how to turn it back on again)

# Transactions

Once we've reached a point where we want to make our changes permanent, we use the command `COMMIT;`

Until that point, if for any reason we want to undo our changes we use `ROLLBACK;`

# Transactions

Similarly we can identify when we have begun a transaction by using  
`START TRANSACTION;`

This is effectively the same as setting  
`SET AUTOCOMMIT = OFF;`

*This command will end as soon as we execute either ROLLBACK or COMMIT*



# Transactions

We've just showed the basic operations of transactions, but implementing them through the command line isn't typically how you'd handle them. If we were to implement a transaction through a stored procedure, how do you think we'd likely identify when to use rollback and when to use commit?

# Transactions

We would commonly use an EXIT HANDLER such as this to tell our stored procedure to effectively cancel out everything IF an exception (or some other type of problem) were to occur.

```
DECLARE EXIT HANDLER FOR SQLEXCEPTION
BEGIN
    ROLLBACK;
END;
```

# Triggers

A trigger is an action that is set to automatically activate when a specific event such as an insert, update or delete, occurs.

Triggers can be configured to activate either before, or after the event you are watching for.

Triggers are defined within a database, and applied to one specific table. You cannot associate a trigger with a TEMPORARY table or a VIEW

# Triggers

Why are triggers valuable to us? Let's think of some examples:

- imagine in our school database we stored the final grade of each class a student took. A trigger could be used to update the students overall GPA each time a grade is added
- imagine in our bike\_stores database that an order is placed for a specific product. Our `stocks` table holds an inventory count of products available, a trigger could subtract the number of items purchased automatically

Without triggers, how would we handle these situations?

# Triggers

Basic Trigger operations:

- CREATE TRIGGER
  - BEFORE/AFTER INSERT
  - BEFORE/AFTER UPDATE
  - BEFORE/AFTER DELETE
- DROP TRIGGER
- SHOW TRIGGERS

# Triggers: Syntax

Creating a TRIGGER: <https://dev.mysql.com/doc/refman/5.7/en/create-trigger.html>

CREATE

```
[DEFINER = user]  
TRIGGER trigger_name  
trigger_time trigger_event  
ON tbl_name FOR EACH ROW  
[trigger_order]  
trigger_body
```

trigger\_time: { BEFORE | AFTER }

trigger\_event: { INSERT | UPDATE | DELETE }

trigger\_order: { FOLLOWS | PRECEDES } other\_trigger\_name

# Triggers: Advantages

<https://www.w3resource.com/mysql/mysql-triggers.php>

## **Uses for triggers:**

- Enforce business rules
- Validate input data
- Generate a unique value for a newly-inserted row in a different file.
- Write to other files for audit trail purposes
- Query from other files for cross-referencing purposes
- Access system functions
- Replicate data to different files to achieve data consistency

## **Benefits of using triggers in business:**

- Faster application development. Because the database stores triggers, you do not have to code the trigger actions into each database application.
- Global enforcement of business rules. Define a trigger once and then reuse it for any application that uses the database.
- Easier maintenance. If a business policy changes, you need to change only the corresponding trigger program instead of each application program.
- Improve performance in client/server environment. All rules run on the server before the result returns.

# Triggers: Disadvantages

## Some disadvantages to using triggers:

- debugging can be challenging since triggers do not reside within an application environment
- cascade effects can lead to an infinite loop where a trigger triggers a trigger which triggers a trigger...
- uncertain outcomes when multiple triggers apply to the same database object and event
- managing triggers can be cumbersome, as they are not easily visible



# Triggers: OLD versus NEW

<https://dev.mysql.com/doc/refman/8.0/en/trigger-syntax.html>

Within the trigger body, the OLD and NEW keywords enable you to access columns in the rows affected by a trigger. OLD and NEW are MySQL extensions to triggers; they are not case-sensitive.

In an INSERT trigger, only NEW.col\_name can be used; there is no old row. In a DELETE trigger, only OLD.col\_name can be used; there is no new row. In an UPDATE trigger, you can use OLD.col\_name to refer to the columns of a row before it is updated and NEW.col\_name to refer to the columns of the row after it is updated.

# Triggers

Let's update our school database so that each student GPA is determined by the grades received in each individual class.

We'll need to make the following updates:

- **add a grade column to enrollments:** assume that a record is created at the start of each semester without a grade, and that at the end of the semester a grade will be added
- **reset all student gpa's to 0.0:** if all grades are now based on what exists in enrollments, we should clear everything out of the students table
- **create a trigger** which will lead to the gpa being recalculated each time a grade is updated in the enrollments table

# Triggers

Solution:

```
DELIMITER //
```

```
CREATE TRIGGER gpa_update
AFTER UPDATE
  ON enrollments FOR EACH ROW
BEGIN
  DECLARE number_of_classes INT;
  DECLARE current_gpa decimal (2,1);

  select gpa into current_gpa from students where student_id = NEW.student_id;
  select count(*) into number_of_classes from enrollments where student_id = NEW.student_id;

  update students set gpa = (((current_gpa * (number_of_classes-1)) + NEW.grade_received) /
number_of_classes) where students.student_id = NEW.student_id;
END //
```

```
DELIMITER ;
```

# Trigger Debugging

What would be some of the additional concerns with the TRIGGER we just created?

- we would need to cover INSERT statements which include the grade
- we would need to cover DELETES
- we would want to prevent anyone from directly modifying the gpa column in students (as the hacker did in our previous lecture), or eliminate it entirely

# Trigger Debugging

What are the tricks for debugging a trigger when its not doing what I expect it to do? With stored procedures we saw that debugging can be a pain, but there are some primitive ways of identifying what is going on internally. Triggers are arguably worse to debug, because you can not write data to the console while they are running.

There are a few options available to use:

- use a db client such as dbForge Studio which will provide some debug utilities
- create a debug table that you can write debug info into
- set up a stored procedure to write information into a dynamic log file (this has some big limitations though - for example, you cannot append to that output file):

```
CREATE PROCEDURE `DYN_LOG` (IN s VARCHAR(500))  
BEGIN  
SELECT s into outfile '/var/lib/mysql-files/data_dump';  
END
```

How can we know where to write our log files? MySQL will prevent you from writing to anything but a trusted location. You can identify the configured trusted location with:

```
SHOW VARIABLES LIKE "secure_file_priv";
```

# Database Security

Security is often overlooked and taken for granted - until something bad happens. When we set up access to AWS early in the semester, security was one of the first things we covered because it was important to protect your accounts as soon as they went active.

So far we haven't had much to worry about with our databases because

- they don't hold any sensitive data
- they should be well protected, assuming you have configured access to your server instances correctly

However, when you work on a system with other users, a system storing sensitive information, a system that could be the heart of your entire application - you need to ensure that it is protected.



<https://specials-images.forbesimg.com/imageserve/1061357610/960x0.jpg?fit=scale>

# Database Security

## Common Sources of Security Vulnerabilities

- **Technical:** flaws in your operating system, web browser, db client
- **Managerial:** when an organization does not place an emphasis on security and does not educate their employees on how to operate securely
- **Cultural:** leaving passwords written down on desks, sending them through email
- **Procedural:** when your system administrators don't require complex passwords, allow password sharing, don't enforce periodic password changes

Some of these security measures are annoying (like the MFA we mandated for this class), but a security breach can really lead to people getting hurt. Having a secure system is worth the additional security steps.

# Database Security: Guidelines

<https://dev.mysql.com/doc/refman/5.7/en/security-guidelines.html>

There are several recommendations here. Some that are worth highlighting:

- make sure all access is password protected. Not only the root user - all users.
- don't use weak passwords. Everyone should know by now what a strong password looks like.
- use a firewall to control port access to your server, as we've done in this course since the start
- don't send unencrypted data over the internet. Use SSL, use SSH for secure communications
- if you are storing passwords in your database, do not store them in a plaintext format
- understand that your MySQL database contains user data in the 'user' table of the 'mysql' database: access to this data should be restricted



# Database Security: Encryption

## Data Encryption:

As mentioned in the previous slide, you should not store passwords in an unencrypted format. There may be many other things you would want to protect as well. There used to be functions available such as ENCRYPT and PASSWORD which you may have heard of in the past, or may still see referenced now, but should no longer be used.

There is a lot of fun stuff to read about and play with here:

<https://dev.mysql.com/doc/refman/5.7/en/encryption-functions.html>

# Account Management

You have probably been using root access all semester long for your database access. This is okay since we've been dealing with disposable databases which are not shared with other users. In a real-world setup you'd want to have individual accounts created for every user, and you'd want their access privileges clearly and carefully defined.

There are three primary things to understand:

- CREATE USER
- GRANT
- REVOKE

# Account Management: CREATE USER

<https://dev.mysql.com/doc/refman/5.7/en/create-user.html>

```
CREATE USER [IF NOT EXISTS]
    user [auth_option] [, user [auth_option]] ...
    [REQUIRE {NONE | tls_option [[AND] tls_option] ...}]
    [WITH resource_option [resource_option] ...]
    [password_option | lock_option] ...

user:
    (see Section 6.2.4, “Specifying Account Names”)

auth_option: {
    IDENTIFIED BY 'auth_string'
  | IDENTIFIED WITH auth_plugin
  | IDENTIFIED WITH auth_plugin BY 'auth_string'
  | IDENTIFIED WITH auth_plugin AS 'auth_string'
  | IDENTIFIED BY PASSWORD 'auth_string'
}

tls_option: {
    SSL
  | X509
  | CIPHER 'cipher'
  | ISSUER 'issuer'
  | SUBJECT 'subject'
}

resource_option: {
    MAX_QUERIES_PER_HOUR count
  | MAX_UPDATES_PER_HOUR count
  | MAX_CONNECTIONS_PER_HOUR count
  | MAX_USER_CONNECTIONS count
}

password_option: {
    PASSWORD EXPIRE
  | PASSWORD EXPIRE DEFAULT
  | PASSWORD EXPIRE NEVER
  | PASSWORD EXPIRE INTERVAL N DAY
}

lock_option: {
    ACCOUNT LOCK
  | ACCOUNT UNLOCK
}
```

The create user statement is pretty long, but we don't need to cover all of the options listed. Let's focus on what you really need to know.

# Account Management

The basic syntax you need to understand is as follows:

```
CREATE USER [IF NOT EXISTS] account_name  
IDENTIFIED BY 'password';
```

Example:

```
CREATE USER IF NOT EXISTS greg@localhost IDENTIFIED BY 'dsf7!8634#kJHa';
```

# Account Management

When you attempt to create a password, it can be rejected for being too weak. And I'm not going to show you how to change that setting to allow something weaker - if you want to know how to do that you'll have to look it up yourself :-)

Passwords must contain a combination of upper/lower case letters, symbols and numbers to be able to pass. Standard dictionary words may be rejected.

If you want to measure the strength of a password you can use the function `VALIDATE_PASSWORD_STRENGTH()`

# Account Management

Who am I?

Once you start working with multiple accounts you may occasionally forget who you are (or rather, who you've connected as). In linux there is a command `whoami` which will tell you, and in MySQL your options are:

```
SELECT USER();  
SELECT CURRENT_USER();  
\s
```



<https://philosophersteve.files.wordpress.com/2018/05/question-mark1.jpg?w=810&h=580&crop=1>

@GregMisicko

# Account Management

Users are going to forget passwords. As long as you still have a user with the privileges to do so (don't lose your root password!) you can reset it:

```
ALTER USER 'greg'@'localhost'  
  IDENTIFIED BY 'new_password' PASSWORD EXPIRE;
```

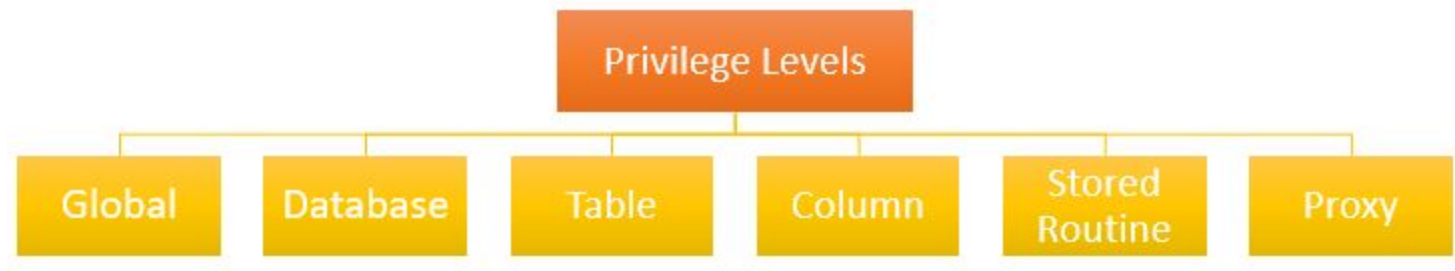
Using PASSWORD EXPIRE is optional, but is recommended as it forces the user to set a new password as soon as they log in.

There are many options for password expiry, etc available to work with:

<https://dev.mysql.com/doc/refman/8.0/en/alter-user.html>

# Account Management: Privileges

MySQL supports the following main privilege levels:



<https://www.mysqltutorial.org/mysql-grant.aspx>

@GregMisicko



# Account Management: Privileges

**Global:** Access to everything, although it is possible to partially revoke access to users who have been granted global access.

**Database, Table, Column:** You can define how much access you wish to grant, and define how granular you want to be by setting privileges at the Database, Table or even the column level.

**Stored Routine:** refers to access for your stored procedures, and stored functions

**Proxy:** Allows you to perform operations as if you were another user. For example, if I had two users: greg and root, and wanted to give root privileges to greg, I could do so by setting up greg as a proxy for root.

# Account Management: Privileges

We can give privileges to a user with GRANT, and we can take privileges away using REVOKE.

To see which privileges a user currently has we use:

```
SHOW GRANTS FOR <user>;
```

For example:

```
SHOW GRANTS FOR greg@localhost;
```

```
+-----+
| Grants for greg@localhost |
+-----+
| GRANT USAGE ON *.* TO `greg`@`localhost` |
+-----+
1 row in set (0.00 sec)
```

The `*.*` as you can probably guess, is a “wildcard” . “wildcard”

The first wildcard refers to a database and the second refers to a table; the wildcards indicate we are granting access to all of them. What is not clear from the results above is what this user actually has access to (the answer is: nothing). “GRANT USAGE ON” means that the user does not have any access privileges defined.

# Account Management: Privileges

We could set this users access to allow everything using a GRANT statement that applies to all databases, and all tables:

```
GRANT ALL ON *.* TO greg@localhost;
```

which now results in a long list of privileges being displayed the next time we check.

However, we don't want this user to have root access. We'll roll back access to nothing by using:

```
REVOKE ALL ON *.* FROM greg@localhost;
```

And let's grant full privileges to this user on the bike\_stores db:

```
GRANT ALL ON bike_stores.* TO greg@localhost;
```

This user now has access to all privileges and all tables in the bike\_stores database.

# Account Management: Privileges

We can GRANT read-only privileges to our user, meaning this user will be capable of retrieving information through a SELECT statement, but will not be able to INSERT, UPDATE, ALTER, DELETE.

```
GRANT SELECT ON school.* TO greg@localhost;
```

We've now granted read-only access to this user for the entire school database.

Let's give the ability to perform INSERT operations on one specific table:

```
GRANT INSERT ON school.courses TO greg@localhost;
```

And finally let's give the ability to UPDATE only one specific column of one table:

```
GRANT UPDATE (grade_received) ON enrollments TO greg@localhost;
```

# Account Management: Roles

Being able to GRANT and REVOKE specific privileges to users is extremely powerful, and an essential part of keeping your database secure.

When determining which privileges to grant to your users, it is good practice to employ the **principle of least privilege**, which means you should grant the least amount of privileges a user requires - no more. The idea behind this is that you are better off being cautious in how much access you grant to a user, and if they require more you can always grant it to them later.

Managing privileges for users on a per-user basis is going to be difficult with a large number of users. It's also unnecessary to manage privileges on a per-user basis in most cases, since it is much more likely you'll want to set up privileges for groups of people based on their responsibilities.

A ROLE is a collection of privileges that multiple users can belong to. Examples of roles you might want to create could be things like: administrators, developers, testers, etc.

# Account Management: Roles

**Mandatory Roles** can also be identify privileges which will be applied to all users in your database. Obviously, be careful to minimize what you grant to mandatory roles.

Mandatory Roles are not specified at the command line, but can be configured in the MySQL server configuration (my.cnf)

You can grant multiple roles to users. The combination of privileges from those multiple roles will be the privileges the user ends up with.

# Account Management: Roles

We can create one or more roles:

```
CREATE ROLE 'administrator', 'school_developer', 'school_marketer';
```

GRANT privileges to them:

```
GRANT ALL ON school.* TO 'school_developer';
```

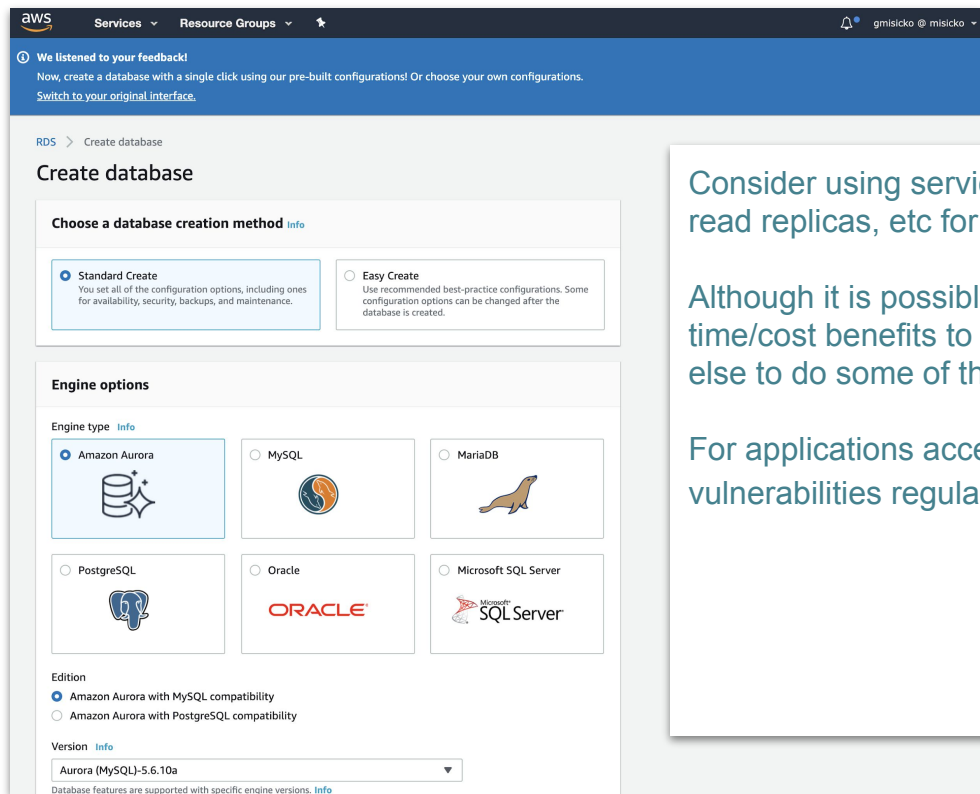
Associate users with those roles:

```
GRANT 'school_developer' TO 'maya'@'localhost';
```

And activate the role for those users:

```
SET DEFAULT ROLE school_developer TO 'maya'@'localhost';
```

# Database Security: Advanced Features



The screenshot shows the AWS Management Console for the 'Create database' page. At the top, there's a blue banner with a message: 'We listened to your feedback! Now, create a database with a single click using our pre-built configurations! Or choose your own configurations. Switch to your original interface.' Below this, the breadcrumb 'RDS > Create database' is visible. The main heading is 'Create database'. Under 'Choose a database creation method', there are two radio buttons: 'Standard Create' (selected) and 'Easy Create'. The 'Standard Create' option has a description: 'You set all of the configuration options, including ones for availability, security, backups, and maintenance.' The 'Easy Create' option has a description: 'Use recommended best-practice configurations. Some configuration options can be changed after the database is created.' Below this is the 'Engine options' section. It has a sub-heading 'Engine type' and an 'Info' link. There are six engine options, each with a radio button and a logo: 'Amazon Aurora' (selected), 'MySQL', 'MariaDB', 'PostgreSQL', 'Oracle', and 'Microsoft SQL Server'. Below the engine options is the 'Edition' section with two radio buttons: 'Amazon Aurora with MySQL compatibility' (selected) and 'Amazon Aurora with PostgreSQL compatibility'. At the bottom, there's a 'Version' section with a dropdown menu showing 'Aurora (MySQL)-5.6.10a' and an 'Info' link. A small note at the very bottom says 'Database features are supported with specific engine versions. Info'.

**Create database**

**Choose a database creation method** [Info](#)

☒ **Standard Create**  
You set all of the configuration options, including ones for availability, security, backups, and maintenance.

☐ **Easy Create**  
Use recommended best-practice configurations. Some configuration options can be changed after the database is created.

**Engine options**

**Engine type** [Info](#)

☒ **Amazon Aurora**

☐ **MySQL**

☐ **MariaDB**

☐ **PostgreSQL**

☐ **Oracle**

☐ **Microsoft SQL Server**

**Edition**

☒ **Amazon Aurora with MySQL compatibility**

☐ **Amazon Aurora with PostgreSQL compatibility**

**Version** [Info](#)

Aurora (MySQL)-5.6.10a

Database features are supported with specific engine versions. [Info](#)

Consider using services which will provide encryption, backups, firewalls, read replicas, etc for you.

Although it is possible to do these things manually, it is worth weighing the time/cost benefits to determine whether or not it's beneficial to pay someone else to do some of those tasks for you.

For applications accessing your database: scan them for security vulnerabilities regularly