Easy Career出品

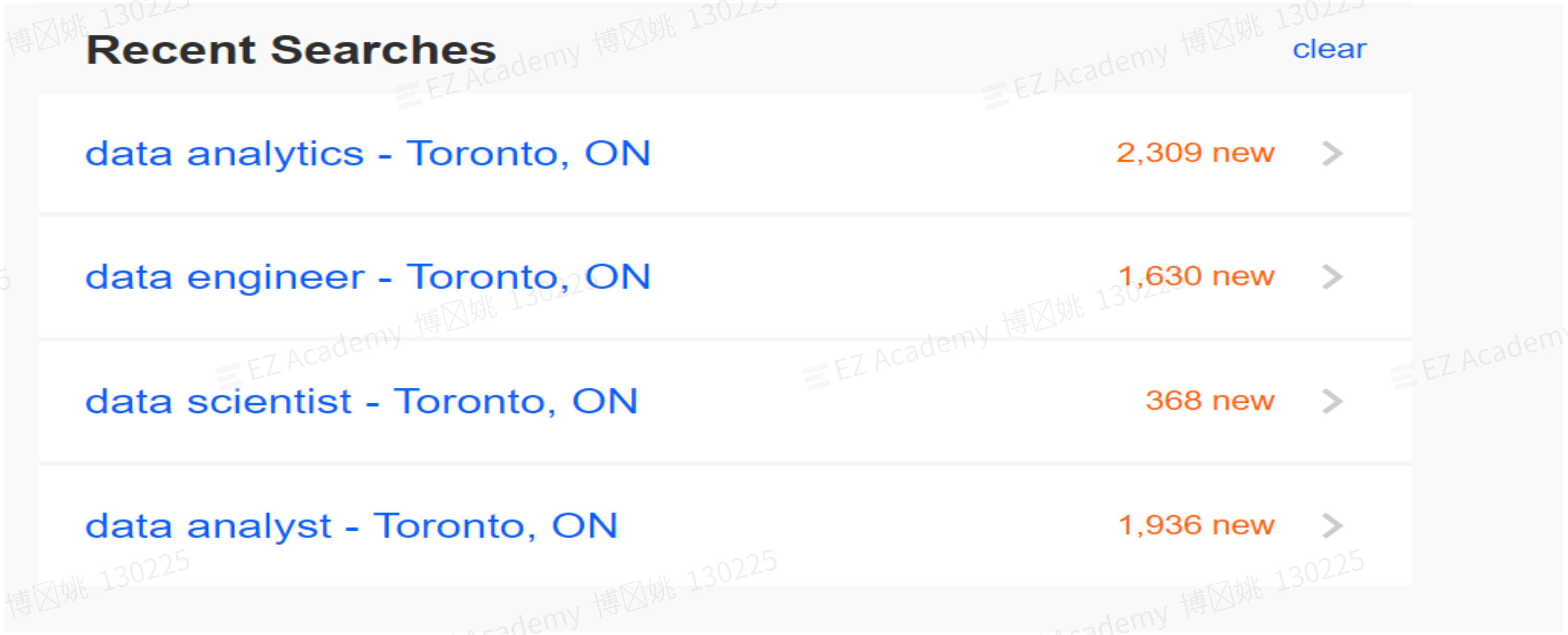
数据/商业分析求职必备技能培训

SQL for Business & Data Analytics

1. SQL和 Database介绍
2. SQL的重要性以及公司里的SQL使用场景

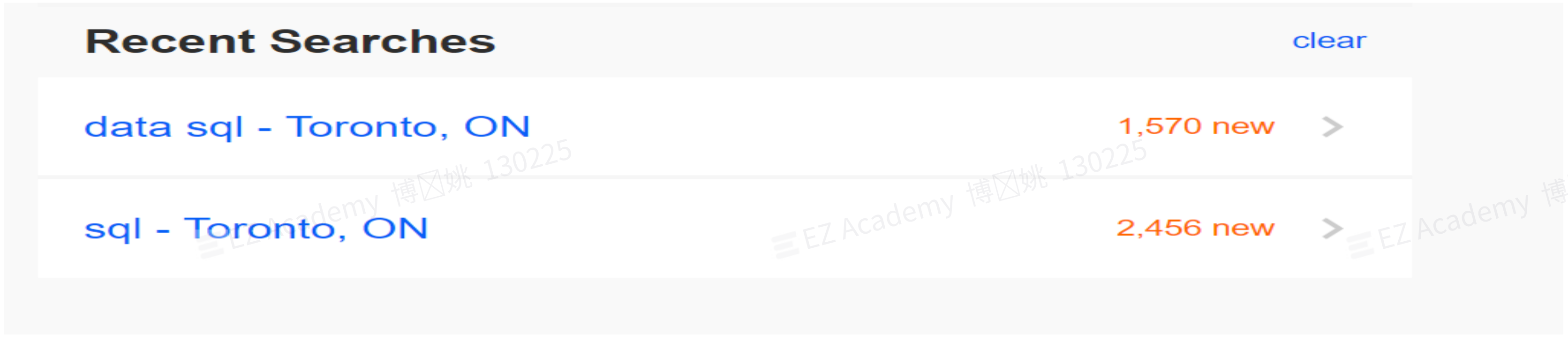
One of the most popular jobs in the era of Big Data

数据分析类工作依然毕业生的最佳求职方向之一!



Recent Searches clear

data analytics - Toronto, ON	2,309 new
data engineer - Toronto, ON	1,630 new
data scientist - Toronto, ON	368 new
data analyst - Toronto, ON	1,936 new



Recent Searches clear

data sql - Toronto, ON	1,570 new
sql - Toronto, ON	2,456 new

Data analytics skills have become essential for any job function in any industry

Demand in Canada is high across different industries

- Retail banking, e.g. credit card, mortgage business
- Insurance, e.g. SunLife, Manulife
- Retail, e.g. Walmart, Sobeys, Loblaw
- Technology, e.g. Amazon, Google, Facebook
- Government/Healthcare
- Investment banking/Asset management
- Consulting firms

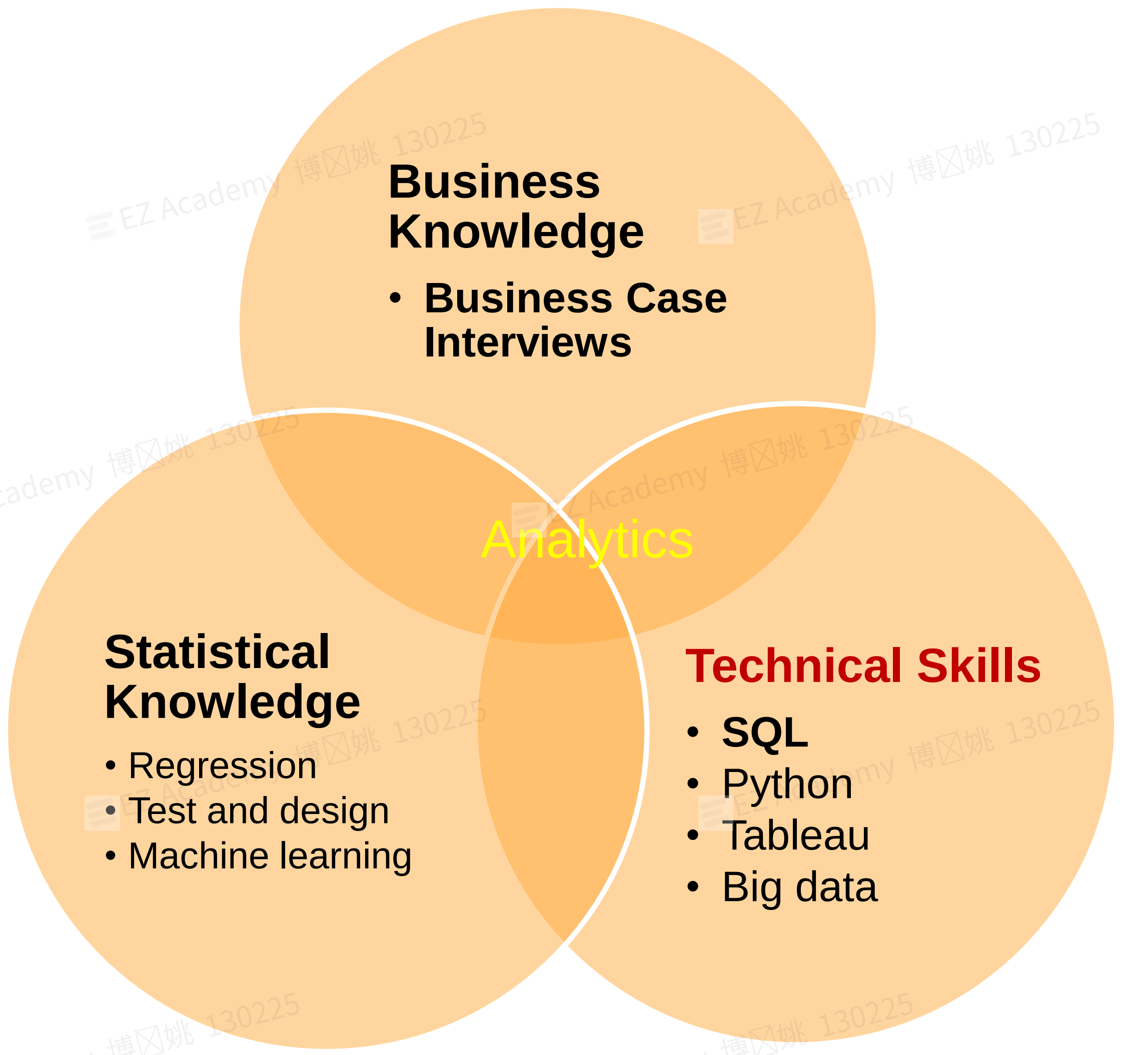
Data analytics skills have become essential for any job function in any industry

Different job functions require data analytics skills

- Data analytics/Data scientist/Data engineer
- Machine Learning
- Reporting
- Risk management
- Marketing
- Trading
- Financial modelling
- Accounting
- Investments
- Consulting
- HR

当下职场，就算不刻意找数据分析的工作，你的第一份工作也难免和数据分析打交道

Skills you need to develop from now on



What is SQL?

Structured Query Language (SQL) is a special-purpose programming language

SQL's purpose:

To manipulate sets of data; typically from a relational database ANSI and ISO standards

What is a Database?

Database:

A container to help organize data

A way to efficiently store and retrieve data

Relational:

- A way to describe data and the relationships between data entities

Why you must learn and master SQL?

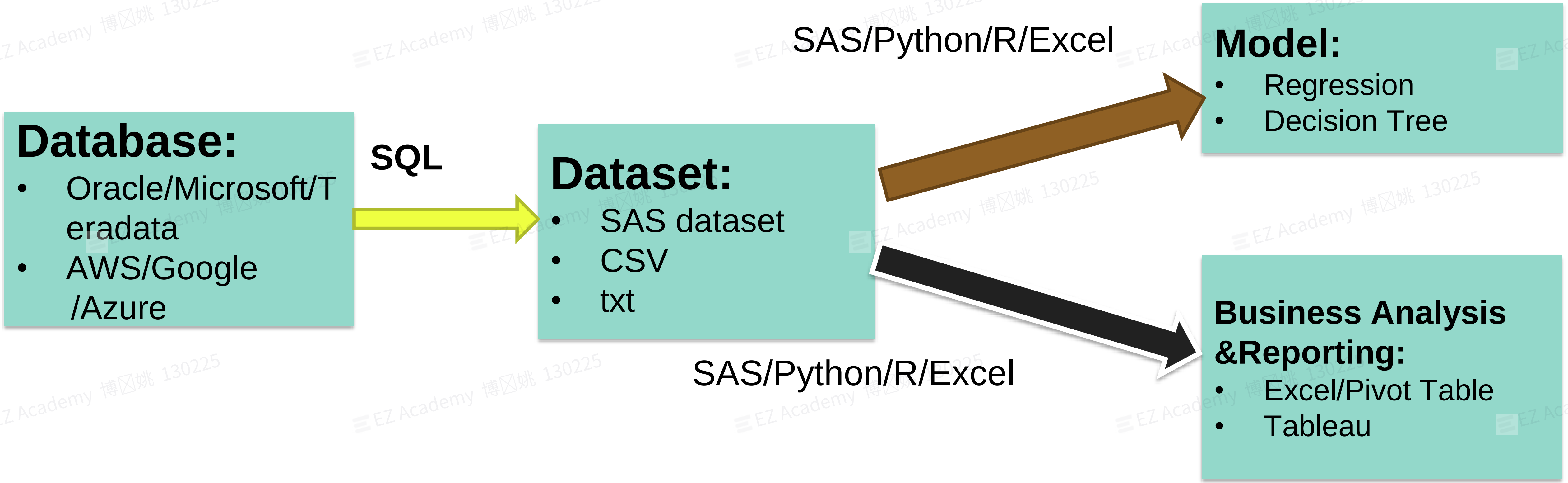
I. At least 80%-90% of data analytics work is to write SQL query

II. Almost every data-related job interviews will test SQL

III. Most of candidates still fail SQL interviews although they have learned it by themselves

Why you must learn and master SQL?

- **At least 80-90% of your daily work is to write SQL query and pull data from database**
- **在大部分公司，SQL是获取大量数据的唯一渠道**



Why you must learn and master SQL?

A typical interview process for data analytics jobs

Resume Screen → **Technical skills, work/project experience**

↓

Phone call interview by HR → **Behavior questions, job fit**

↓

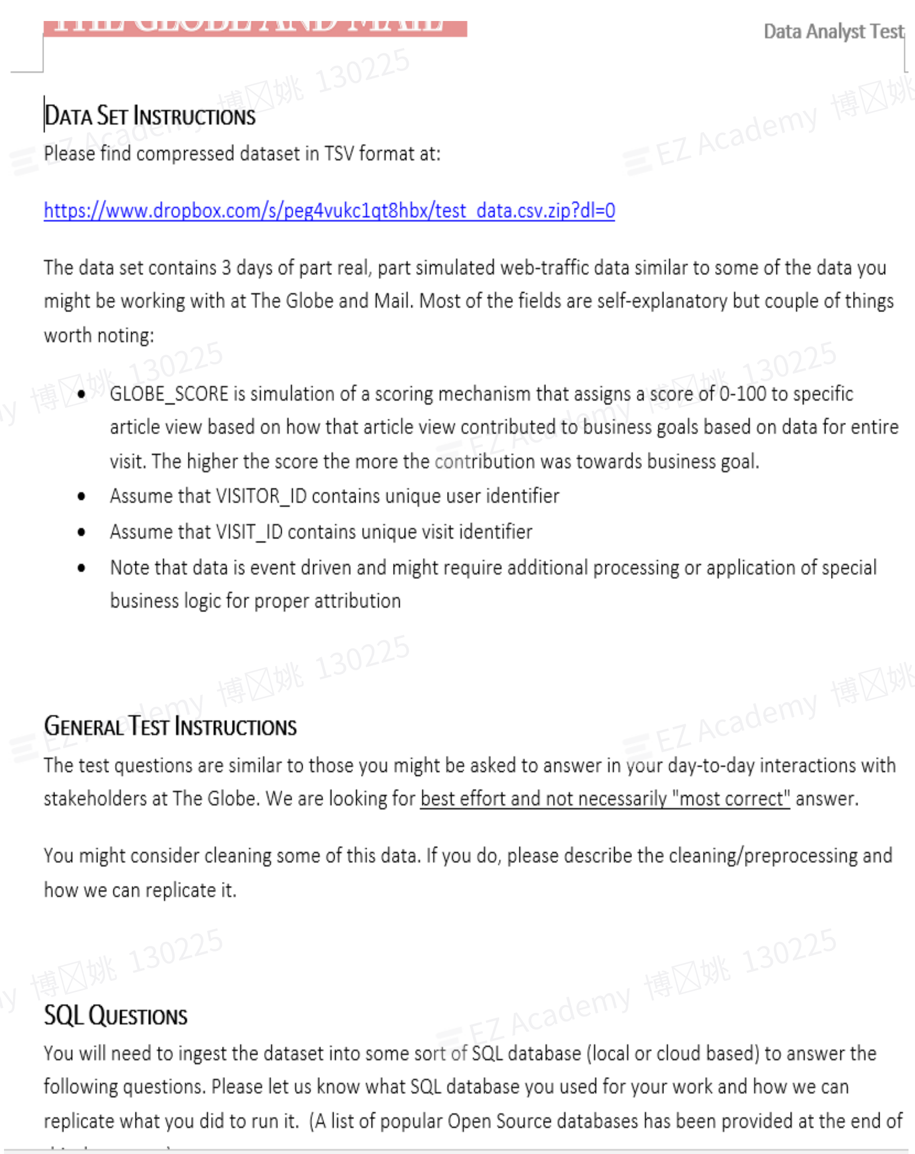
online test/homework → **Math/SQL/Python/R/Analytics challenges**

↓

On-site interview → **Business case interview/technical interview/behavior interview**

Why you must learn and master SQL?

You definitely will be tested for SQL skills during interviews



Data Analyst Test

DATA SET INSTRUCTIONS

Please find compressed dataset in TSV format at:

https://www.dropbox.com/s/peg4vukc1qt8hbx/test_data.csv.zip?dl=0

The data set contains 3 days of part real, part simulated web-traffic data similar to some of the data you might be working with at The Globe and Mail. Most of the fields are self-explanatory but couple of things worth noting:

- GLOBE_SCORE is simulation of a scoring mechanism that assigns a score of 0-100 to specific article view based on how that article view contributed to business goals based on data for entire visit. The higher the score the more the contribution was towards business goal.
- Assume that VISITOR_ID contains unique user identifier
- Assume that VISIT_ID contains unique visit identifier
- Note that data is event driven and might require additional processing or application of special business logic for proper attribution

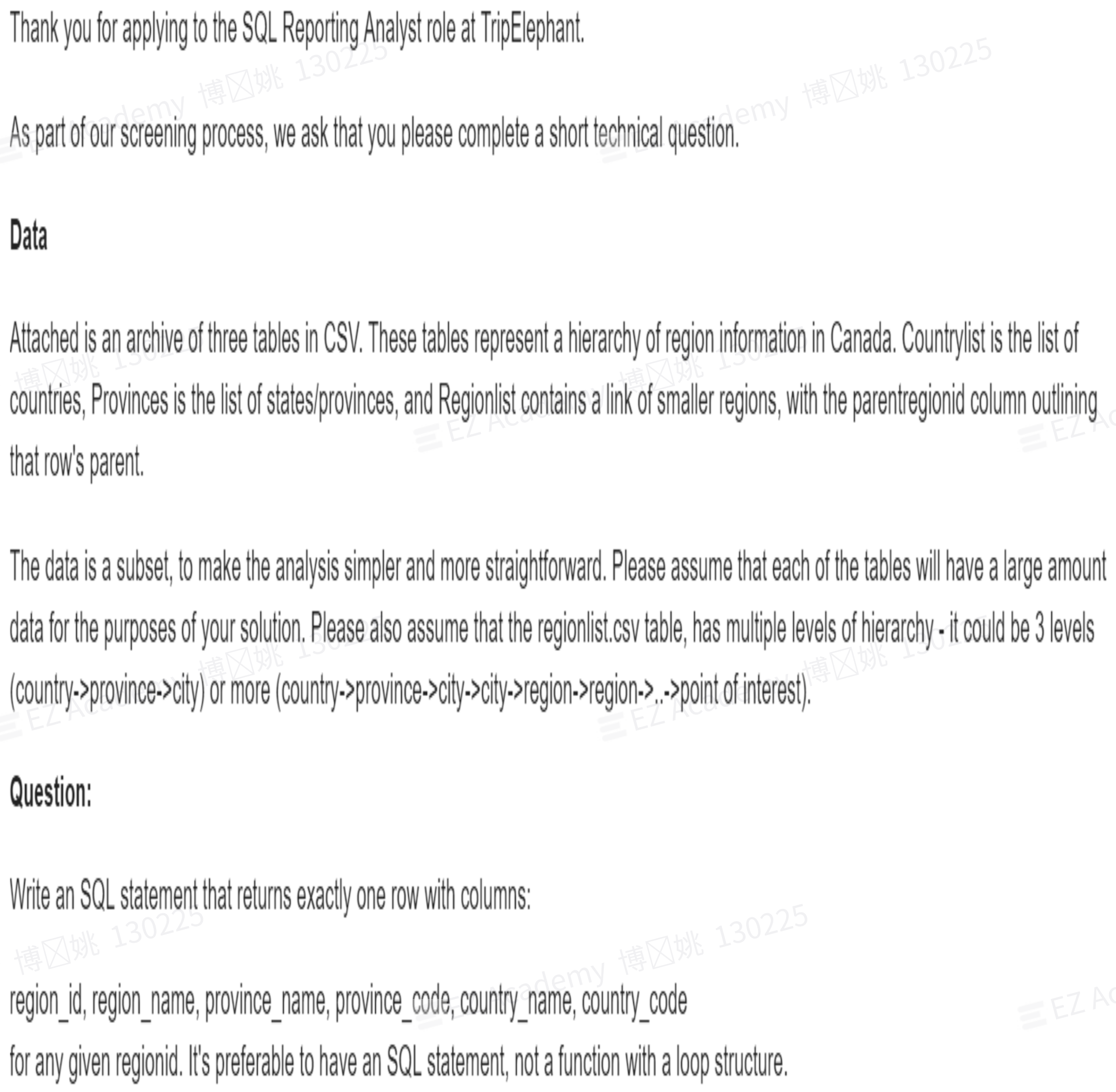
GENERAL TEST INSTRUCTIONS

The test questions are similar to those you might be asked to answer in your day-to-day interactions with stakeholders at The Globe. We are looking for best effort and not necessarily "most correct" answer.

You might consider cleaning some of this data. If you do, please describe the cleaning/preprocessing and how we can replicate it.

SQL QUESTIONS

You will need to ingest the dataset into some sort of SQL database (local or cloud based) to answer the following questions. Please let us know what SQL database you used for your work and how we can replicate what you did to run it. (A list of popular Open Source databases has been provided at the end of



Thank you for applying to the SQL Reporting Analyst role at TripElephant.

As part of our screening process, we ask that you please complete a short technical question.

Data

Attached is an archive of three tables in CSV. These tables represent a hierarchy of region information in Canada. Countrylist is the list of countries, Provinces is the list of states/provinces, and Regionlist contains a link of smaller regions, with the parentregionid column outlining that row's parent.

The data is a subset, to make the analysis simpler and more straightforward. Please assume that each of the tables will have a large amount data for the purposes of your solution. Please also assume that the regionlist.csv table, has multiple levels of hierarchy - it could be 3 levels (country->province->city) or more (country->province->city->city->region->region->..->point of interest).

Question:

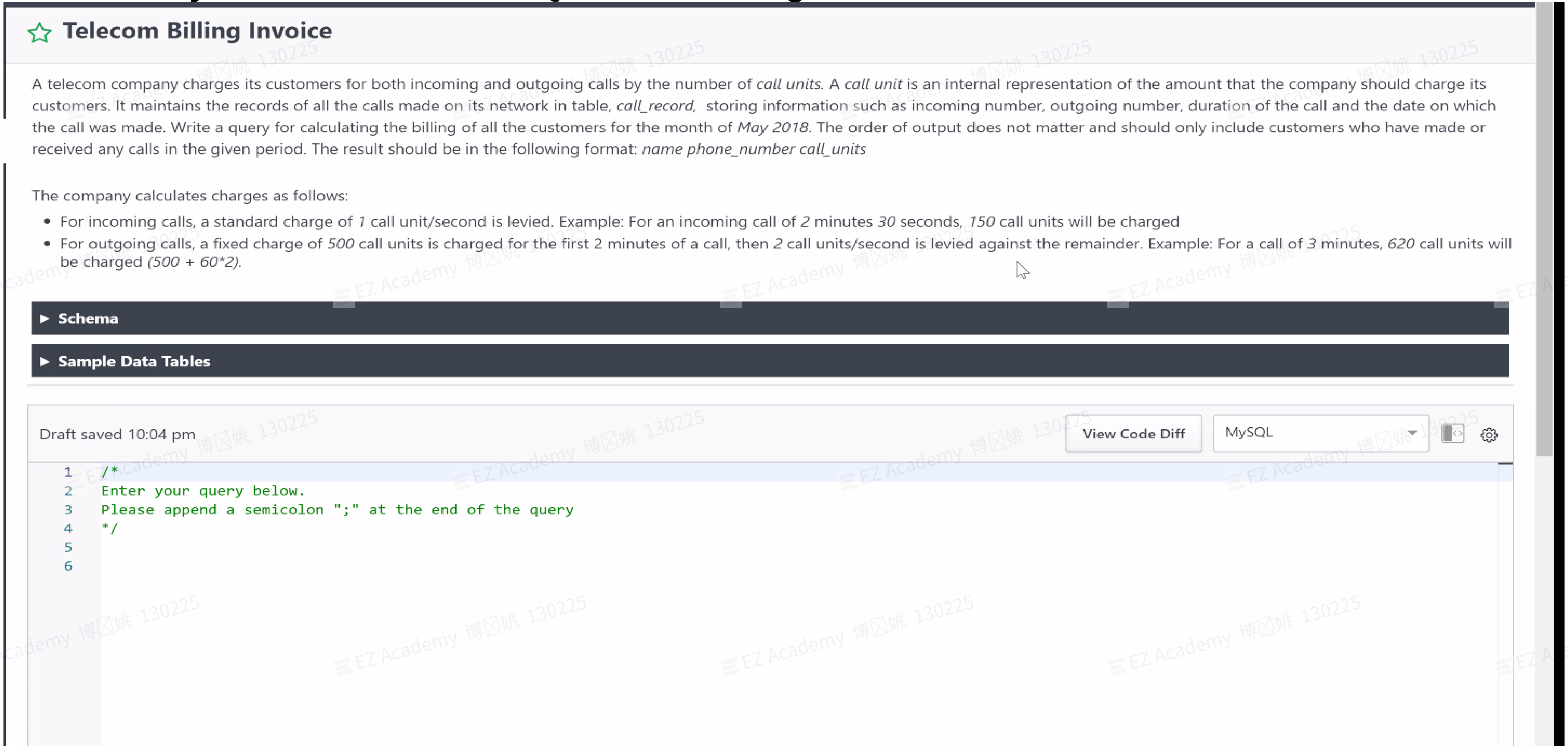
Write an SQL statement that returns exactly one row with columns:

region_id, region_name, province_name, province_code, country_name, country_code

for any given regionid. It's preferable to have an SQL statement, not a function with a loop structure.

Why you must learn and master SQL?

You definitely will be tested for SQL skills during interviews



Telecom Billing Invoice

A telecom company charges its customers for both incoming and outgoing calls by the number of *call units*. A *call unit* is an internal representation of the amount that the company should charge its customers. It maintains the records of all the calls made on its network in table, *call_record*, storing information such as incoming number, outgoing number, duration of the call and the date on which the call was made. Write a query for calculating the billing of all the customers for the month of *May 2018*. The order of output does not matter and should only include customers who have made or received any calls in the given period. The result should be in the following format: *name phone_number call_units*

The company calculates charges as follows:

- For incoming calls, a standard charge of *1* call unit/second is levied. Example: For an incoming call of *2* minutes *30* seconds, *150* call units will be charged
- For outgoing calls, a fixed charge of *500* call units is charged for the first 2 minutes of a call, then *2* call units/second is levied against the remainder. Example: For a call of *3* minutes, *620* call units will be charged *(500 + 60*2)*.

```
/*
Enter your query below.
Please append a semicolon ";" at the end of the query
*/
```

课程特色

SQL for Business & Data Analytics

- 零基础入门
- 真实面试题
- 结合商业实例，学习使用SQL解决问题
- 实战project体验数据分析工作场景，丰富简历

SQL 一定是面试中的必备！

Easy Career出品

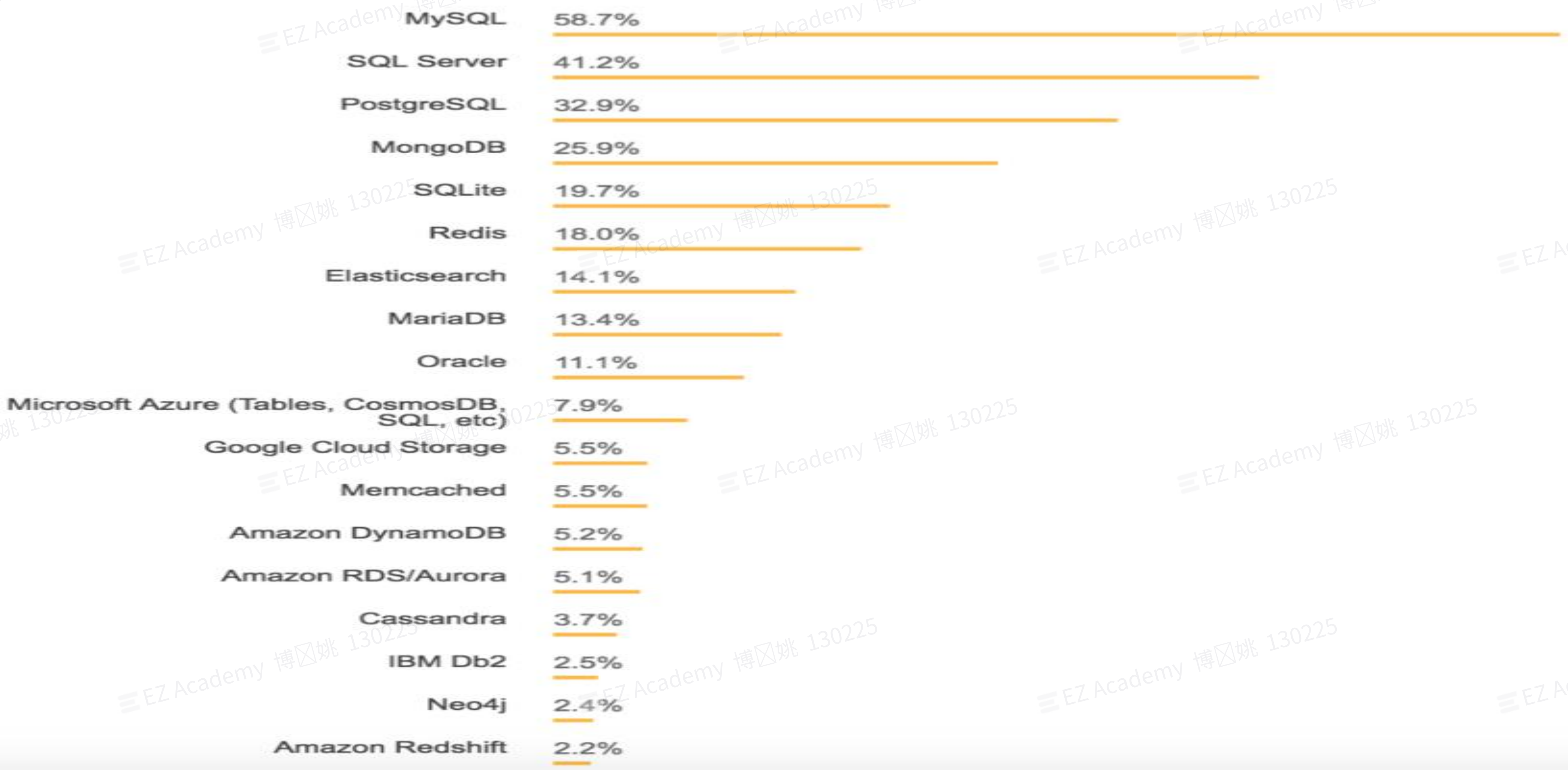
SQL for Business & Data Analytics

1. **SQL和 Database介绍 ☑**
2. **SQL的重要性以及公司里的SQL使用场景 ☑**
3. **SQL的安装和设置 ☑**

Different types of Relational Databases?

MySQL®
Microsoft® SQL Server®
PostgreSQL
Microsoft® Access®
IBM DB2
ORACLE® DATABASE
SQLite

Why MySQL?



StackOverflow_2018_survey

Who use MySQL?

facebook

LinkedIn

yelp

YouTube

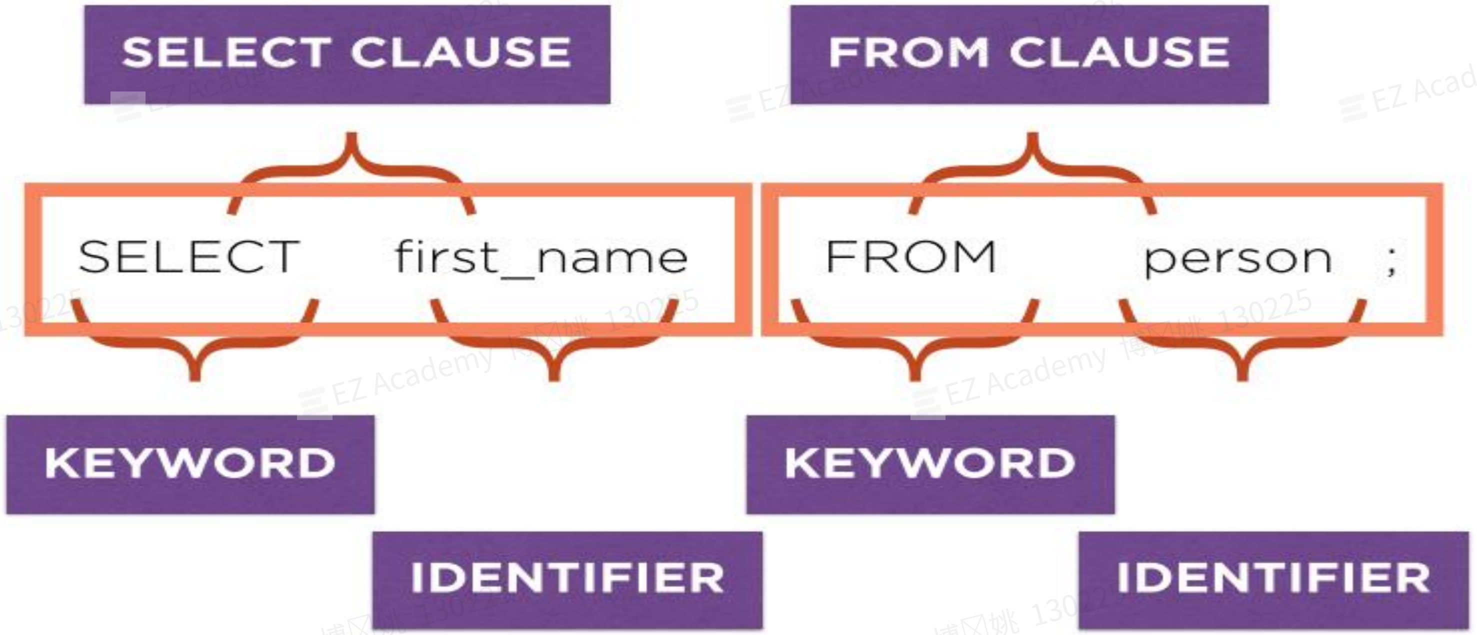
GitHub

twitter

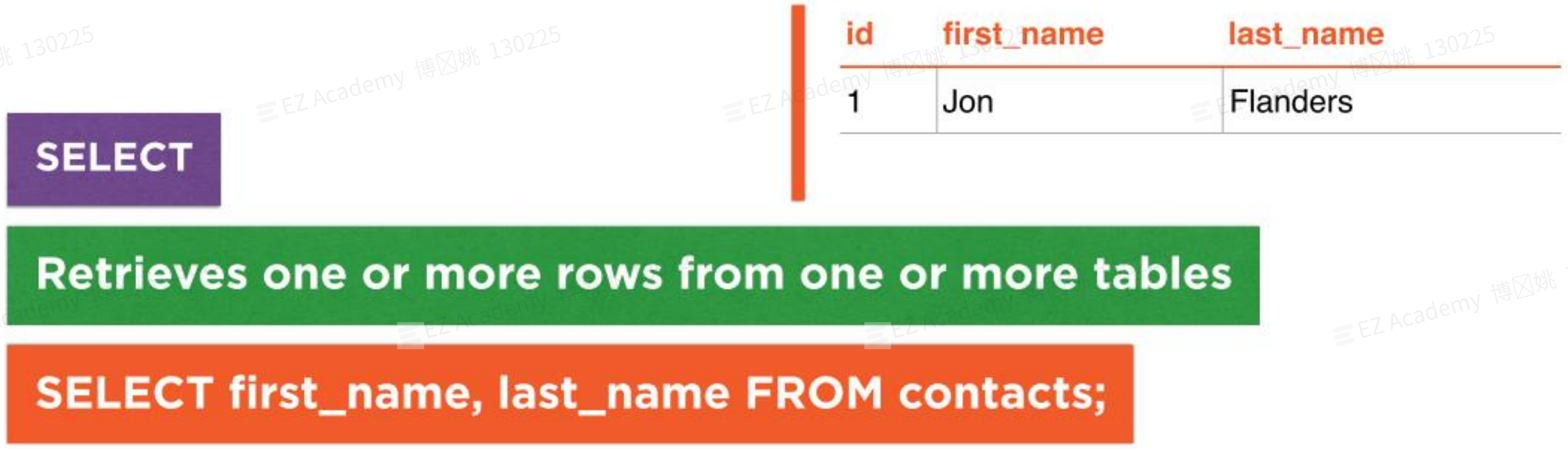
Basic SQL Syntax

SQL Statement

A SQL Statement is an expression that tells a database what you want it to do:



Basic SQL Commands



id	first_name	last_name
1	Jon	Flanders

SELECT

Retrieves one or more rows from one or more tables

```sql
SELECT first_name, last_name FROM contacts;
```

Basic SQL Commands

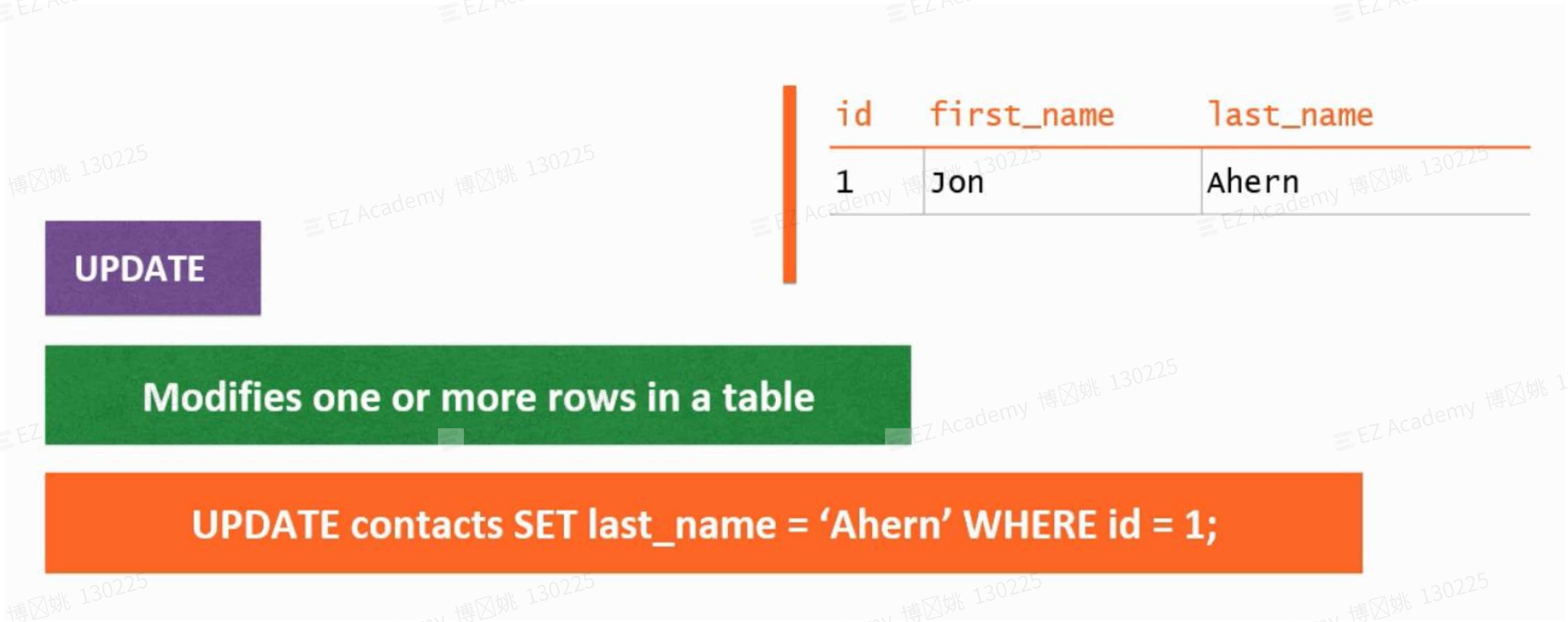
id	first_name	last_name
1	Jon	Flanders
2	Fritz	Onion

INSERT

Adds one or more rows into a table

```
INSERT INTO contacts (first_name, last_name)
VALUES ('Fritz','Onion');
```

Basic SQL Commands



id	first_name	last_name
1	Jon	Ahern

UPDATE

Modifies one or more rows in a table

```
UPDATE contacts SET last_name = ‘Ahern’ WHERE id = 1;
```

Basic SQL Commands

id	first_name	last_name
1	Jon	Flanders
2	Fritz	Onion

DELETE

Removes one or more rows from one table

```
DELETE FROM contacts where id = 2;
```

SELECT Statement

The SELECT Statement

- Most of the time it contains a list of columns from a table you want to query
- Then, a FROM clause is required
- After every column comes a comma
- *Except: no comma after the last column*

```
SELECT <COLUMN_NAME>, <COLUMN_NAME>  FROM  <TABLE_NAME> ;
```

SELECT prod_id, prod_name, prod_price FROM Products;

The SELECT Statement

- Use **Select *** to pulls all the columns from a table

    *SELECT * FROM Products;*

- It is a bad practice.

Use Limit to constrain the display of records

*SELECT * FROM Products* ***Limit 5****;*

Ways to Constrain the Number of Results

- **DISTINCT Qualifier**
- **WHERE Clause**

Distinct

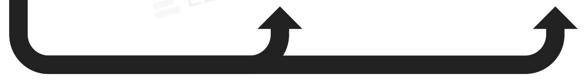
- Without DISTINCT:

  SELECT vend_id FROM Products;

- **With Distinct:**

  SELECT DISTINCT vend_id FROM Products;

SELECT DISTINCT vend_id, prod_price FROM Products;



You always put distinct before any column and it will apply to all the following columns

Use Comments

1. Use two hyphens

```
SELECT prod_name   -- this is a comment
FROM Products;
```

2. A # at the start of a line makes the entire line a comment

```
# This is a comment
SELECT prod_name
FROM Products;
```

3. /* starts a comments, and */ ends it. Anything between /* and */ is comment text.

```
/* SELECT prod_name, vend_id
FROM Products; */
SELECT prod_name
FROM Products;
```

Practice 1.1:

Use SCHEMAS(Database) **world** and table **country** to practice:

- Select all columns from table country and only display 5 records
- Only want to check code, name, region, population columns from table country;
- Want to check different values of region in table country
- Comment out one of the query you just wrote down

Sort Retrieved Data by Using Order By

Order By clause:

- ORDER BY takes the name of one or more columns by which to sort the output
- Position of ORDER BY Clause When specifying an ORDER BY clause, be sure that it is the **last clause** in your SELECT statement

SELECT prod_name FROM Products
***ORDER BY** prod_name;*

If it is not the last clause, an error will be generated.

Sort by Multiple Columns

- To sort by multiple columns, simply specify the column names **separated by commas** (just as you do when you are selecting multiple columns):

```
SELECT prod_id, prod_price, prod_name
FROM Products
ORDER BY prod_price, prod_name;
```

- It is important to understand that when you are sorting by multiple columns, the sort sequence is exactly as specified

Sort by Column Position

- ORDER BY also supports ordering specified by **relative column position**

```
SELECT prod_id, prod_price, prod_name
FROM Products
ORDER BY 2, 3;
```

- The relative positions of selected columns in the SELECT list are specified. ORDER BY 2 means sort by the second column in the SELECT list, the prod_price column

- Obviously, you cannot use this technique when sorting by columns that are not in the SELECT list

But you can mix and match actual column names and relative column positions in a single statement

Specifying Sort Direction

- By default, it is **ascending sort order**
- To sort by descending order, the keyword DESC must be specified and put after the column you want to order by descending

```
SELECT prod_id, prod_price, prod_name
FROM Products
ORDER BY prod_price DESC;
```

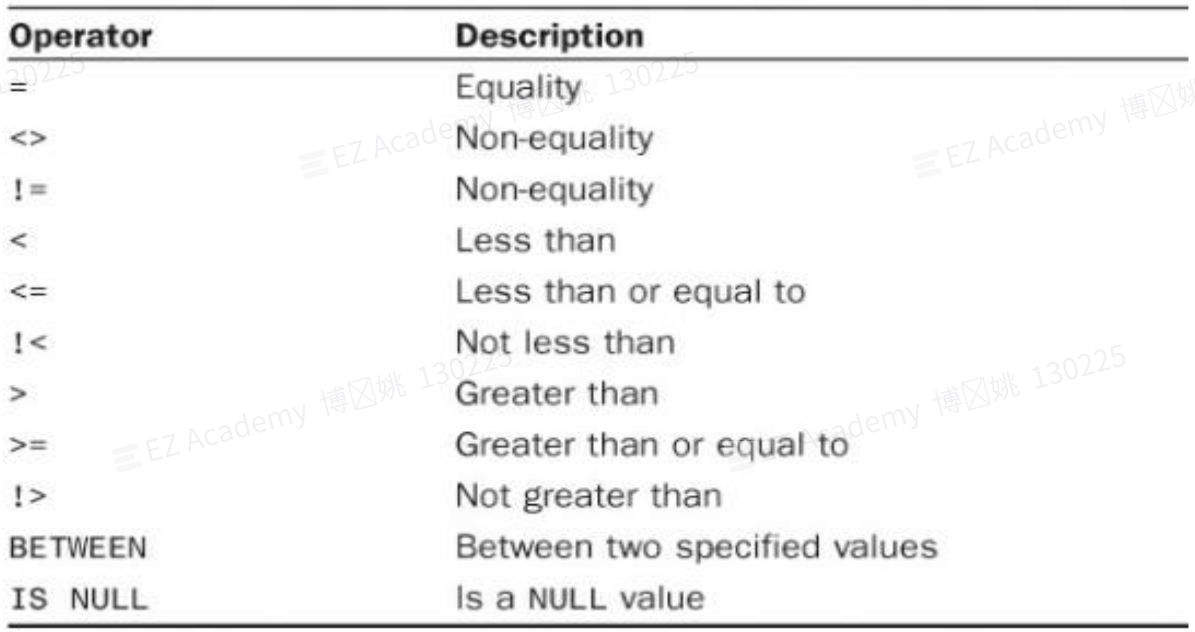
- If you want to sort descending on multiple columns, be sure each column has its **own** DESC keyword

Filter Data by using Where Clause

- Retrieving just the data you want involves specifying search criteria, also known as a filter condition
- The WHERE clause is specified right **after the table name** (the FROM clause) as follows:

  SELECT prod_name, prod_price
  FROM Products
  ***WHERE** prod_price = 3.49;*
- When using both ORDER BY and WHERE clauses, make sure that **ORDER BY comes after the WHERE**, otherwise an error will be generated

Where Clause Operators



Operator	Description
=	Equality
<>	Non-equality
!=	Non-equality
<	Less than
<=	Less than or equal to
!<	Not less than
>	Greater than
>=	Greater than or equal to
!>	Not greater than
BETWEEN	Between two specified values
IS NULL	Is a NULL value

- Some of the operators listed above are redundant

SELECT prod_name, prod_price
FROM Products
WHERE prod_price <= 10;

Where Clause Operators

- Check for Nonmatches:

  SELECT vend_id, prod_name
  FROM Products
  *WHERE vend_id <> **'DLL01';***

- **When to use Quotes**: the single quotes are used to delimit a string
- **Be careful - the value is case-sensitive!**

- Check for a Range of Values:

  SELECT prod_name, prod_price
  FROM Products
  WHERE prod_price BETWEEN 5 AND 10;

Where Clause Operators

- Check for No Value - When a column contains no value, it is said to contain a NULL value:

*SELECT * FROM Customers WHERE cust_email is NULL*

Advanced Data Filtering

Advanced Data Filtering

- Using the **AND** Operator - used in a WHERE clause to specify that only rows matching all the specified conditions should be retrieved.

```
SELECT prod_id, prod_price, prod_name
FROM Products
WHERE vend_id = 'DLL01' AND prod_price <= 4;
```

Advanced Data Filtering

- Using the **OR** Operator - The OR operator instructs the database management system software to retrieve rows that match either condition.

```
SELECT prod_name, prod_price
FROM Products
WHERE vend_id = 'DLL01' OR vend_id = 'BRS01';
```

Practice 1.2:

Use SCHEMAS(Database) **world** and table **country** to practice:

- Display the TOP 5 countries with the largest population
- Rank the country by descending region, and ascending surfaceArea
- List the countries with lifeExpectancy => 75 and rank by ascending population
- List countries became independent between 1980 and 1990;
- List countries in region Eastern Asia and indepYear is null;
- Select countries in Western Europe, with population less than 80000000 and surfacearea larger than 3000, and rank these countries by descending Code column

Advanced Data Filtering

- Combining AND and OR operators – understand order of evaluation
- Use table product to do the practice:

I need a list of all products costing $10 or more made by vendors DLL01 and BRS01

- SQL (like most languages) processes AND operators before OR operators.

```
SELECT prod_name, prod_price
FROM Products
WHERE vend_id = 'DLL01' OR vend_id = 'BRS01'
AND prod_price >= 10;
```

Advanced Data Filtering

- Combining AND and OR operators – using Parentheses in WHERE Clauses

```
SELECT prod_name, prod_price
FROM Products
WHERE (vend_id = 'DLL01' OR vend_id = 'BRS01’) AND prod_price >= 10;
```

Advanced Data Filtering

- Using the IN Operator - used to specify a range of conditions, any of which can be matched
- IN takes a comma-delimited list of valid values, all enclosed within parentheses

```
SELECT prod_name, prod_price, vend_id
FROM Products
WHERE vend_id  IN ('DLL01','BRS01’);
```

Use Wildcard Filtering

- Wildcard - Special characters used to match parts of a value
- Using the LIKE Operator - To use wildcards in search clauses, the LIKE operator must be used
- Wildcard searching can **only be used with text fields (strings)**, you can’t use wildcards to search fields of non-text datatypes

1. **The Percent Sign (%) Wildcard** – most frequently used wildcard - % means, ***match any number of occurrences of any character***

```
SELECT prod_id, prod_name
FROM Products
WHERE prod_name LIKE ‘Fish%’;

SELECT prod_id, prod_name
FROM Products
WHERE prod_name LIKE ‘%bean bag%’;
```

Use Wildcard Filtering

2. The Underscore (_) Wildcard – *match a single character*

SELECT prod_id, prod_name
FROM Products
WHERE prod_name LIKE '__ inch teddy bear’;

SELECT prod_id, prod_name
FROM Products
WHERE prod_name LIKE ‘% inch teddy bear';

Practice 2.1:

Use SCHEMAS(Database) **world** and table **country** to practice:

- List countries in region Eastern Africa or North America or Middle East order by region
- For all countries in region Eastern Asia , select the countries with population > 7000000 or lifeexpectancy > 75
- Identify countries with name beginning with ‘A’ and ending with ‘a’

Create Calculated Fields

Where calculated fields come in?

- Sometimes, the data stored in the table is not exactly what your application needs
- Rather than retrieve the data as it is, what you really want is to retrieve **converted, calculated, or reformatted data** directly from the database
- Calculated fields don’t actually exist in database tables. Rather, a calculated field is created on-the-fly **within a SQL SELECT statement**

Concatenating Fields

- Concatenating Fields - Joining values together (by appending them to each other) to form a single long value.

  - MySQL uses CONCAT() to concatenate strings:

    SELECT ***concat(****vend_name, ‘(‘ , vend_country, ‘)’****)***
    FROM Vendors
    ORDER BY vend_name;

Concatenating Fields

- Use **Aliases** to name the new calculated column

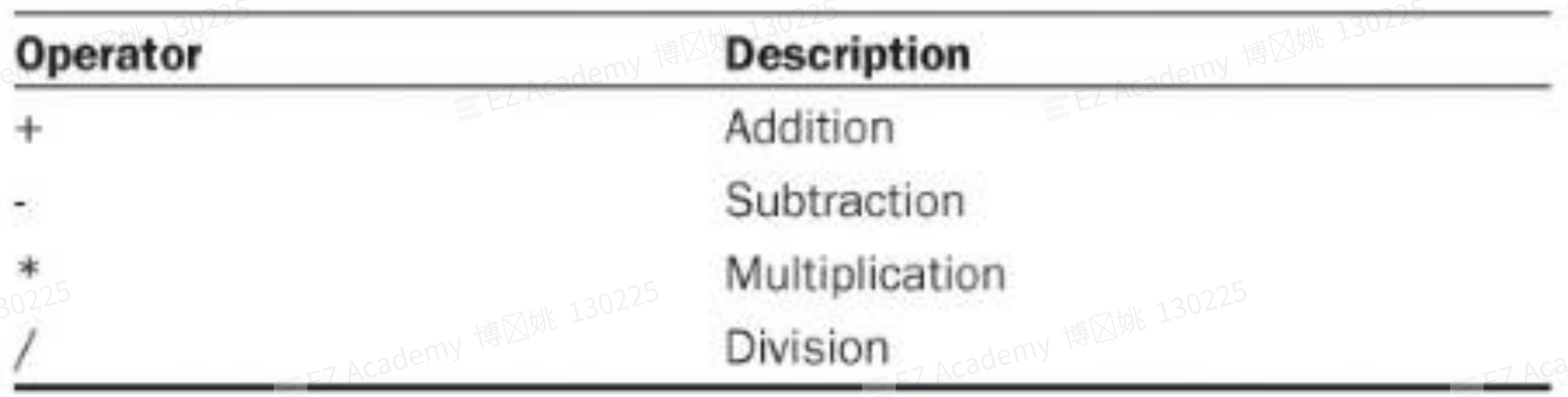
  SELECT Concat(vend_name, ' (', vend_country, ')')
  AS vend_title
  FROM Vendors
  ORDER BY vend_name;

Performing Mathematical Calculations

- Another frequent use for calculated fields is performing mathematical calculations on retrieved data.

```
SELECT prod_id, quantity, item_price, quantity*item_price AS total_sales
FROM OrderItems
WHERE order_num = 20008;
```

- SQL Mathematical Operators:

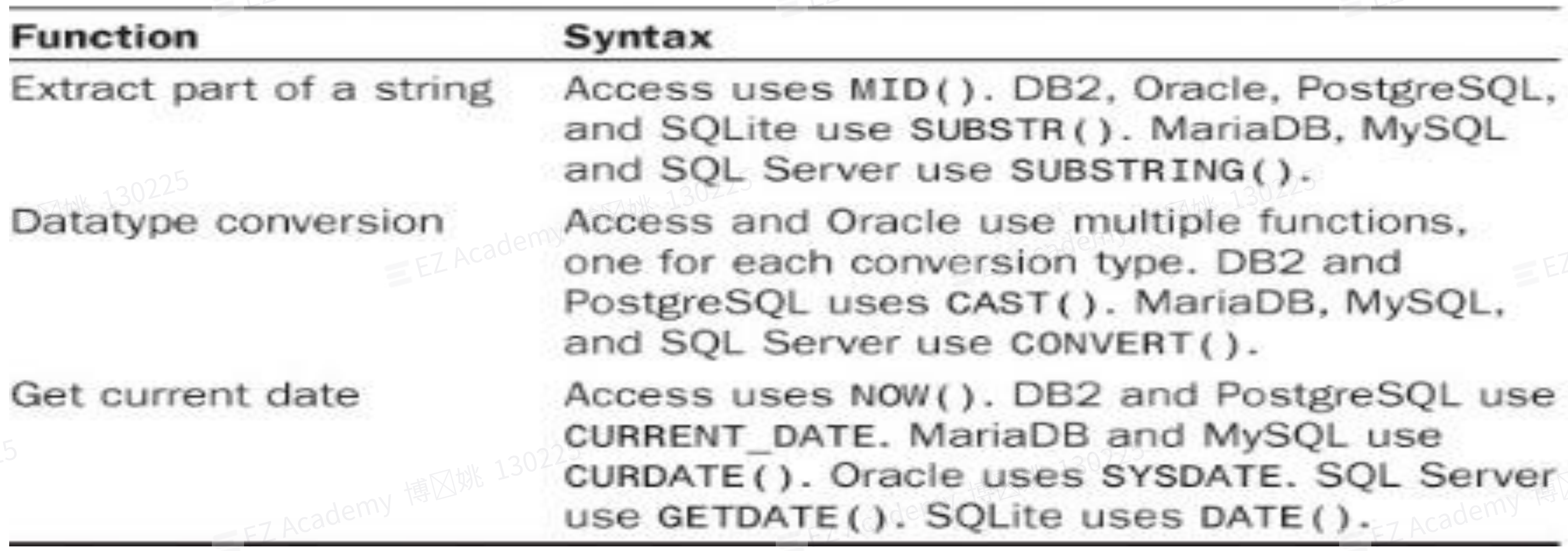


Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division

Use Data Manipulation Functions

Understand Functions

- Functions perform calculations **on columns!**
- Unlike SQL statements, which for the most part are supported by all DBMSs equally, functions tend to be very DBMS(database management system) specific:
  - DBMS Function Differences:



Function	Syntax
Extract part of a string	Access uses `MID()`. DB2, Oracle, PostgreSQL, and SQLite use `SUBSTR()`. MariaDB, MySQL and SQL Server use `SUBSTRING()`.
Datatype conversion	Access and Oracle use multiple functions, one for each conversion type. DB2 and PostgreSQL uses `CAST()`. MariaDB, MySQL, and SQL Server use `CONVERT()`.
Get current date	Access uses `NOW()`. DB2 and PostgreSQL use `CURRENT_DATE`. MariaDB and MySQL use `CURDATE()`. Oracle uses `SYSDATE`. SQL Server use `GETDATE()`. SQLite uses `DATE()`.

Different Types of Functions

- **String functions** are used to manipulate strings of text; for example, trimming or padding values and **converting values to upper and lowercase**

*SELECT vend_name, **UPPER(**vend_name**)** AS vend_name_uppercase FROM Vendors ORDER BY vend_name;*

*SELECT vend_name, **SUBSTRING(**vend_name,1,4**)** AS first_4_letters_of_vend_name FROM Vendors ORDER BY vend_name;*

SUBSTRING(str,pos,len) Return the substring as specified

Different Types of Functions

- **Date and time functions** are used to manipulate date and time values and to extract specific components from these values; for example, extracting date part from a date value, and checking date validity:

SELECT order_num
FROM Orders
*WHERE **YEAR(**order_date**)** = 2012;*

YEAR() Return the year

SELECT order_num, order_date,
***NOW()** as currentdateandtime*
FROM Orders;

NOW() Return the current date and time

SELECT order_num, order_date, NOW() as currentdateandtime, curdate() as current_dt, **DATEDIFF(**curdate(), order_date**)** as dategap
FROM Orders;

DATEDIFF() Subtract two dates

Different Types of Functions

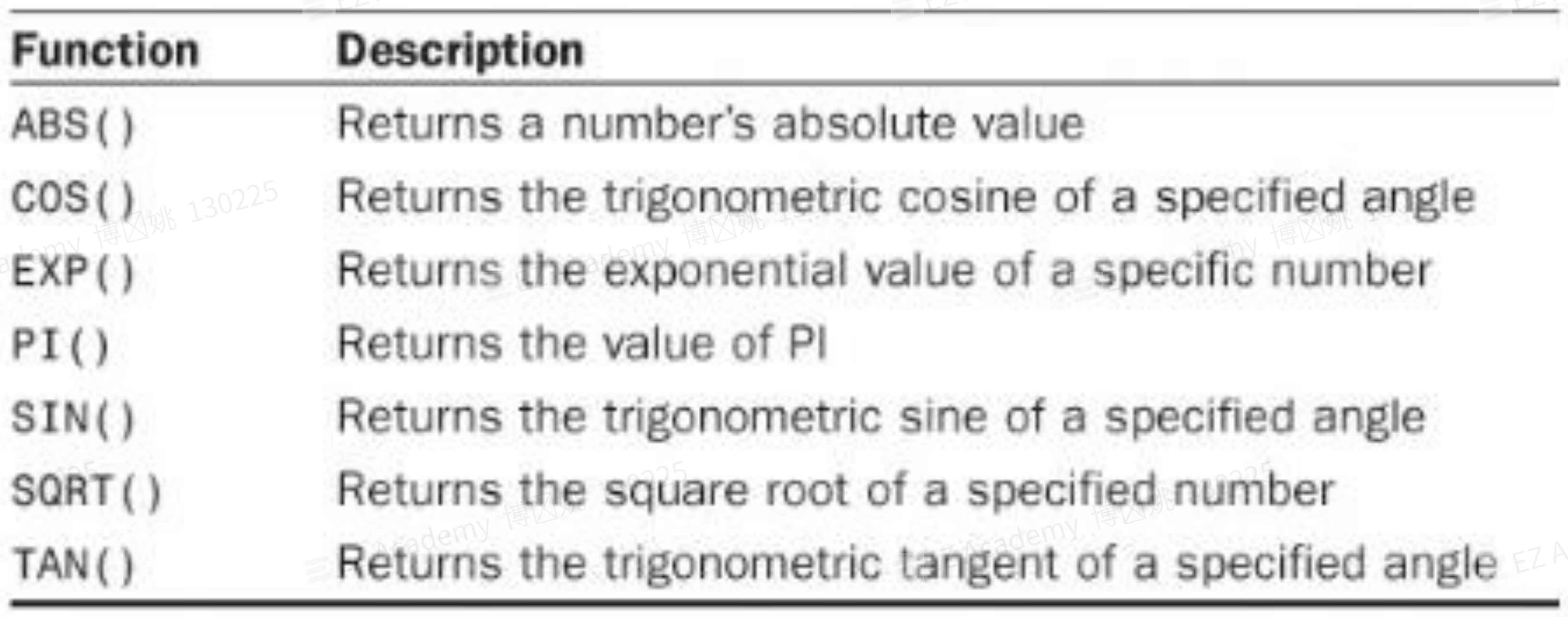
- **CASE** expression is a very commonly used **control flow function**
- CASE expression can be used to conditionally enter into some logic based on the status of a condition being satisfied
- It is usually used to create a new column
- It is better to make sure each condition is mutually exclusive

```
CASE WHEN [value=compare_value] THEN result
          [WHEN [value=compare_value] THEN result ...]
             [ELSE result] END (AS new_column)
```

SELECT prod_price,
case when** prod_price < 6 then 'low price' **else** 'high price' **end
from products;

Different Types of Functions

- **Numeric functions** are used to perform mathematical operations on numeric data
  - for example, returning absolute numbers and **performing algebraic calculations**
  - Commonly used Numeric Manipulation Functions:



Function	Description
ABS()	Returns a number's absolute value
COS()	Returns the trigonometric cosine of a specified angle
EXP()	Returns the exponential value of a specific number
PI()	Returns the value of PI
SIN()	Returns the trigonometric sine of a specified angle
SQRT()	Returns the square root of a specified number
TAN()	Returns the trigonometric tangent of a specified angle

Practice 2.2:

Use SCHEMAS(Database) **world** and table **country** to practice:

- Use population/surfacearea to get pop_density and rank pop_density by descending order
- *Bonus: how can I get countries with pop_density > 1000*
- *Create a column called Population_size to segment the country by population size:*
  - *If population < 1 million, then 'small'; if 1 million <= population < 10 million, then 'medium',*

    if 10 million <= population < 100 million, then 'large; if population >= 100 million, then 'Extra large';

  Then show country name and population_size
- In the table, we found a column called as Code which should be country code, and another column called as Code2. I want to know whether Code2 is just the first 2 letters of Code. Please write query to confirm this. *Hint: use substring to get the first 2 letters of Code, and compare with Code2, if they match with each other, then we can confirm*

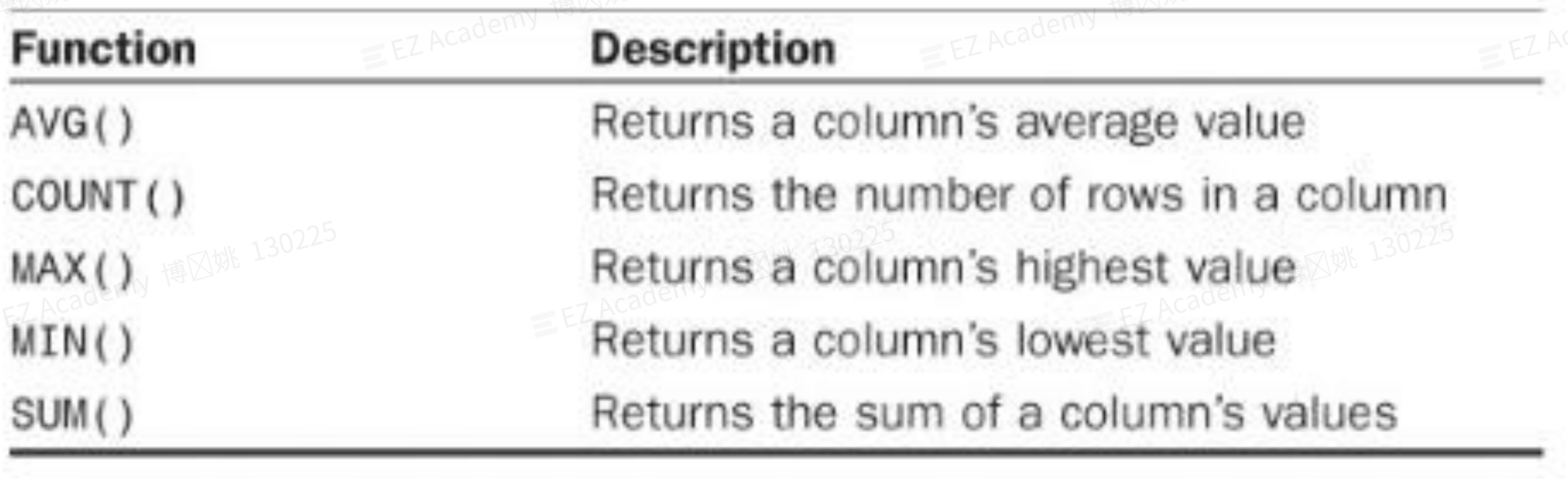
Summarize Data by using Aggregate Functions

Use Aggregate Functions

- Examples of Aggregate Functions usages:
  - Determining the number of rows in a table (or the number of rows that meet some condition or contain a specific value)
  - Obtaining the sum of a set of rows in a table.
  - Finding the highest, lowest, and average values in a table column (either for all rows or for specific rows)

Use Aggregate Functions

- Aggregate Functions – Functions that operate on a set of rows to calculate and return a single value:



Function	Description
AVG()	Returns a column's average value
COUNT()	Returns the number of rows in a column
MAX()	Returns a column's highest value
MIN()	Returns a column's lowest value
SUM()	Returns the sum of a column's values

Use Aggregate Functions

- The **AVG()** Function - can be used to return the average value of any column:

  SELECT AVG(prod_price) AS avg_price
  FROM Products;

- AVG() can also be used to determine the average value of specific columns or rows:

  SELECT AVG(prod_price) AS avg_price
  FROM Products
  ***WHERE** vend_id = 'DLL01';*

> - AVG() only be used on an individual Column
> - Column rows containing NULL values are ignored by the AVG() function

Use Aggregate Functions

- The **COUNT()** Function - can determine the number of rows in a table or the number of rows that match a specific criterion
  - Use **COUNT(*)** to count the number of rows in a table, whether columns contain values or NULL values:

    SELECT COUNT(*) AS num_cust
    FROM Customers;

- Column rows with NULL values in them are ignored by the COUNT() function if a column name is specified, but not if the asterisk (*) is used.

  - Use **COUNT(column)** to count the number of rows that have values in a specific column, ignoring NULL values:

    SELECT COUNT(cust_email) AS num_cust
    FROM Customers;

Use Aggregate Functions

- The **MAX()** Function - returns the highest value in a specified column

```
SELECT MAX(prod_price) AS max_price
FROM Products;
```

- The **MIN()** Function - returns the lowest value in a specified column

```
SELECT MIN(prod_price) AS min_price
FROM Products;
```

- Column rows with NULL values in them are ignored by the MAX() and MIN() function

Use Aggregate Functions

- The **SUM()** Function - to return the sum (total) of the values in a specific column

```
SELECT SUM(quantity) AS items_ordered
FROM OrderItems
WHERE order_num = 20005;
```

- **SUM()** can also be used to calculated values:

```
SELECT SUM(item_price*quantity) AS total_sales
FROM OrderItems
WHERE order_num = 20005;
```

- All the aggregate functions can be used to perform calculations on multiple columns using the standard mathematical operators, as shown in the example
- Column rows with NULL values in them are ignored by the SUM() function

Aggregates on Distinct Values

- The five aggregate functions can all be used in two ways:
  - To perform calculations on all rows, specify the ALL argument or specify no argument at all (because ALL is the default behavior)
  - To only include unique values, specify the **DISTINCT** argument

```
SELECT count(DISTINCT prod_price) AS count_price
FROM Products
WHERE vend_id = 'DLL01';
```

- DISTINCT may only be used with COUNT() if a column name is specified. DISTINCT may not be used with COUNT(*)
- Most of time, DISTINCT is used together with Count()

Combine Aggregate Functions

- SELECT statements may contain as few or as many aggregate functions as needed:

```
SELECT COUNT(*) AS num_items,
MIN(prod_price) AS price_min,
MAX(prod_price) AS price_max,
AVG(prod_price) AS price_avg
FROM Products;
```

*GPA is important or not for finding your **first** job**?***

It is important:

- *if you want to get hired through campus recruiting (but not necessary)*
- *If you want to do consulting firms/investment banking*

However, it is not that important for over 90% jobs in the market:

- *Most jobs don't have GPA requirements*
- *Keep in mind you are not only competing with fresh graduates but also with professionals*
- *Higher GPA won't guarantee you pass resume screen, but Hard skills and relevant experience can*
- *Of course, it can not be too bad*

Group Data

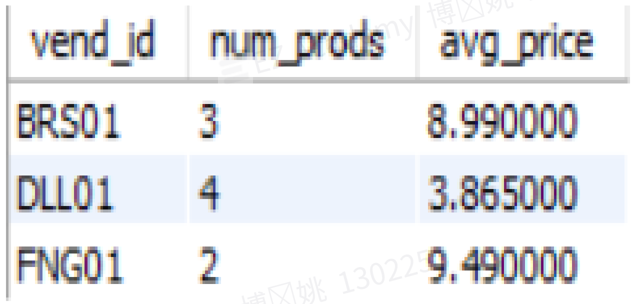
Create Groups

- Groups are created using the GROUP BY clause in your SELECT statement

```
SELECT vend_id,
        COUNT(*) AS num_prod
        FROM Products
        GROUP BY vend_id
        ORDER by num_prods;
```

With Group By, the Count() aggregates the number of records for each **unique vend_id**

```
SELECT vend_id,
        COUNT(*) AS num_prod ,
        AVG(prod_price) as avg_price
        FROM Products
        GROUP BY vend_id
        ORDER by num_prods;
```



vend_id	num_prods	avg_price
BRS01	3	8.990000
DLL01	4	3.865000
FNG01	2	9.490000

Create Groups

- Sometimes, we call non-aggregated columns as segmentation variables

*SELECT **order_num,prod_id**,*
sum(quantity)
FROM orderitems
*GROUP BY **order_num,prod_id**;*

You can have more than one non-aggregated columns in SELECT statement, but make sure they are also in Group BY

- **Aside from the aggregate calculation statements, every column in your SELECT statement must be present in the GROUP BY clause!!!**
- If the grouping column contains a row with a NULL value, NULL will be returned as a group. If there are multiple rows with NULL values, they'll all be grouped together
- The GROUP BY clause must come after any WHERE clause and before any ORDER BY clause

Filter Groups

- To filter Groups, you have to use HAVING clause, instead of WHERE clause
- The only difference is that WHERE filters rows and HAVING filters groups

```
SELECT cust_id, COUNT(*) AS orders
FROM Orders
GROUP BY cust_id
HAVING COUNT(*) >= 2;
```

- We can use them together: WHERE filters before data is grouped, and HAVING filters after data is grouped

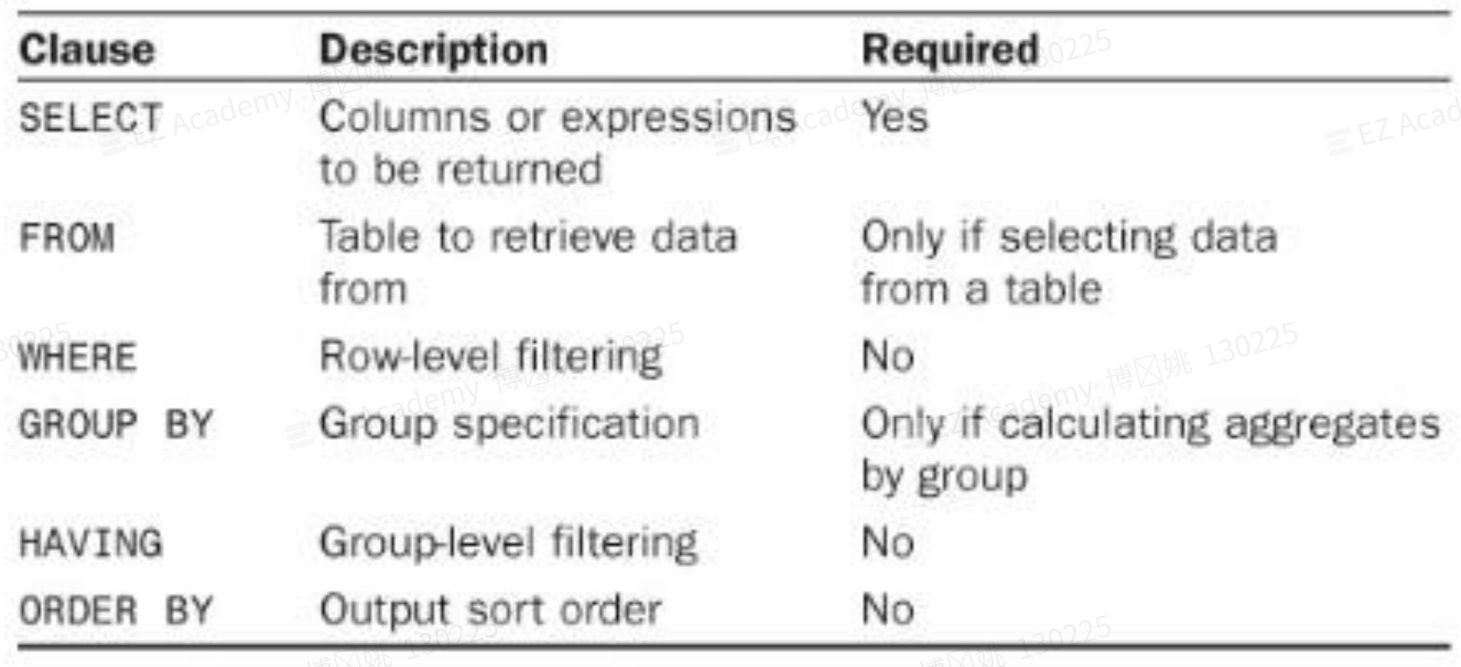
```
SELECT vend_id, COUNT(*) AS num_prods
FROM Products
WHERE prod_price >= 4
GROUP BY vend_id
HAVING COUNT(*) >= 2;
```

- **Rows that are eliminated by a WHERE clause will not be included in the group**
- **Use HAVING only in conjunction with GROUP BY clauses.**
- **Use WHERE for standard row-level filtering**

SELECT Statement Ordering

- Order matters in SQL!

SELECT Statement and Sequence of each Clause:



Clause	Description	Required
SELECT	Columns or expressions to be returned	Yes
FROM	Table to retrieve data from	Only if selecting data from a table
WHERE	Row-level filtering	No
GROUP BY	Group specification	Only if calculating aggregates by group
HAVING	Group-level filtering	No
ORDER BY	Output sort order	No

Subqueries

Understand Subqueries

- SQL also enables you to create subqueries - queries that are embedded into other queries

Example: Now suppose you wanted a list of all the customers who ordered item RGAN01:

1. Retrieve the order numbers of all orders containing item RGAN01

```
SELECT order_num
FROM OrderItems
WHERE prod_id = 'RGAN01';
```

Understand Subqueries

Example: Now suppose you wanted a list of all the customers who ordered item RGAN01:

1. Retrieve the order numbers of all orders containing item RGAN01
2. Retrieve the customer ID of all the customers who have orders listed in the order numbers returned in the previous step

```
SELECT cust_id
FROM Orders
WHERE order_num IN (20007,20008);
```

Understand Subqueries

Example: Now suppose you wanted a list of all the customers who ordered item RGAN01:

1. Retrieve the order numbers of all orders containing item RGAN01
2. Retrieve the customer ID of all the customers who have orders listed in the order numbers returned in the previous step
3. Retrieve the customer information for all the customer IDs returned in the previous step

```
SELECT cust_name, cust_contact
FROM Customers
WHERE cust_id IN ('1000000004','1000000005');
```

Understand Subqueries

Example: Now suppose you wanted a list of all the customers who ordered item RGAN01:

```
SELECT cust_name, cust_contact
FROM Customers
WHERE cust_id IN (SELECT cust_id
                  FROM Orders
                  WHERE order_num IN (SELECT order_num FROM OrderItems
                                      WHERE prod_id = 'RGAN01'));
```

- **Subquery SELECT statements can only retrieve a single column. Attempting to retrieve multiple columns will return an error**

Practice 3

Use SCHEMAS(Database) **world** to practice:

1. Create a report showing sum of population and average life expectancy for each continent, and make sure your result doesn’t include any continent with total population less than 1000000;

2. Create a column called Population_size to segment the country by population size and calculate the average lifeexpectancy for each segment

    1. *If population < 1 million, then ‘small’; if 1 million <= population < 10 million, then ‘medium’, if 10 million <= population < 100 million, then ‘large; if population >= 100 million, then ‘Extra large’;*

    your final result should show the population size segment and the average life expectancy for each segment

3. Using table countrylanguage, to get the number of countries speaking each distinct language, then rank the language by how many countries and by descending order

4. Calculate the average population for each region and **exclude** the region whose average population is fewer than the average population of all the countries in the country table. The final result should have 2 columns – region and average population. Hint: use subquery to get the overall average population first

Resume

On Average How Long Does a Recruiter Look at a Resume?

30-55 Seconds

How do I get started?

1. Job Descriptions
2. Performance Reviews
3. Function / Industry Research

Key Tips

- Results oriented
- Start with strong action verbs
- Focus on accomplishments
- Always update your Resume based on what jobs you are applying!!!! (change title, experience, accomplishments)
- In the past tense
- Try not to be more than 2 lines
- Quantify experience whenever possible
- Utilize industry keywords (learn from job descriptions)

Accomplishment Statements

<u>**Basic statement:**</u>

- Conducted training programs in several provinces and reduced customer complaints

<u>**More specific:**</u>

- Conducted more than 45 service technician training programs throughout Ontario and Quebec, reducing customer complaints

<u>**Even better**</u>

- Reduced customer complaints by 22% in a 6-month period by conducting more than 45 service technician training programs throughout Ontario and Quebec in English and French

Join Tables

Create a Join

- A join is a mechanism used to associate tables within a SELECT statement

```
SELECT vend_name, prod_name, prod_price
FROM Vendors, Products
WHERE Vendors.vend_id = Products.vend_id;
```

- You also need **Join Condition** to tell the database how to join – in this example, the WHERE clause acts as a filter to only include rows that match the specified filter condition
- Without **Join Condition**, Cartesian Product will be generated - The number of rows retrieved will be the number of rows in the first table multiplied by the number of rows in the second table

- you must use the fully qualified column name (table and column separated by a period) whenever there is a possible ambiguity about which column you are referring to

Cartesian Product

One

X	A
1	a
2	b
4	d

Two

X	B
2	x
3	y
5	v

```
Proc sql;
        select *
From one, two
```

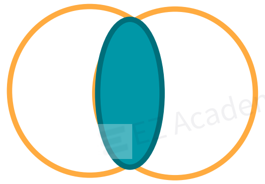
X	A	X	B
1	a	2	x
1	a	3	y
1	a	5	v
2	b	2	x
2	b	3	y
2	b	5	v
4	d	7	x
4	d	3	y
4	d	5	v

- When NOT including a WHERE, the SQL returns **the Cartesian Product**
- The number of rows in the Cartesian product of tables One and Two **= 3 x 3 = 9**

In all types of joins, SQL generates a Cartesian Product first, and then eliminates rows that do not meet any subsettting criteria that you have specified!

Inner Join

Inner join – Only returns the rows that match across all tables



Three

X	A
1	a1
1	b2
2	b1
2	b2
4	d

Four

X	B
2	x1
2	x2
3	y
5	v

```
Proc sql;
        select *
From three, four
Where three.x=four.x;
```

X	A	X	B
2	b1	2	x1
2	b1	2	x2
2	b2	2	x1
2	b2	2	x2

Inner Join

- Inner Join has **2 different syntaxes**:

  SELECT vend_name, prod_name, prod_price

  FROM Vendors, Products

  ***WHERE** Vendors.vend_id = Products.vend_id;*

- A different one:

  SELECT vend_name, prod_name, prod_price

  *FROM Vendors **(INNER) JOIN** Products*

  ***ON** Vendors.vend_id = Products.vend_id;*

Join Multiple Tables

- SQL imposes no limit to the number of tables that may be joined in a SELECT statement.

```
SELECT prod_name, vend_name, prod_price, quantity
FROM OrderItems, Products, Vendors
WHERE Products.vend_id = Vendors.vend_id
AND OrderItems.prod_id = Products.prod_id
AND order_num = 20007;
```

- Be careful not to join tables unnecessarily. The more tables you join the more performance will degrade

Let us revisit the example from last class

- Now suppose you wanted the information (e.g. customer name, contact) of all the customers who ordered item RGAN01

```
SELECT cust_name, cust_contact
FROM Customers
WHERE cust_id IN (SELECT cust_id
                  FROM Orders
                  WHERE order_num IN (SELECT order_num FROM OrderItems
                                      WHERE prod_id = 'RGAN01'));
```

- Using Joins is more efficient than subqueries:

```
SELECT cust_name, cust_contact
FROM Customers, Orders, OrderItems
WHERE Customers.cust_id = Orders.cust_id
AND OrderItems.order_num = Orders.order_num
AND prod_id = 'RGAN01';
```

Create Advanced Joins

Use Table Aliases

- In addition to using aliases for column names and calculated fields, SQL also enables you to alias table names. There are two primary reasons to do this:
  - To shorten the SQL syntax
  - To enable multiple uses of the same table within a single SELECT statement

```
SELECT C.cust_name, C.cust_contact as customer_contact
FROM Customers AS C, Orders AS O, OrderItems AS OI
WHERE C.cust_id = O.cust_id
AND OI.order_num = O.order_num
AND prod_id = 'RGAN01';
```

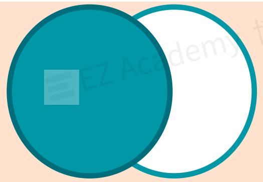
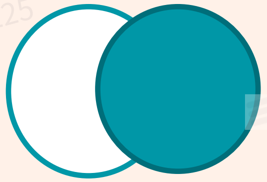
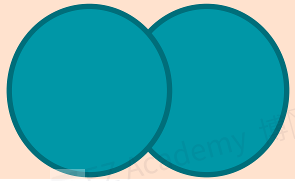
Practice 4.1

Use SCHEMAS(Database) **world** to practice:

- Use table Country and Language to find the language used in each country. I want all columns from Country table and language column from Language table
  - Use 2 types of Inner Join syntax to solve this: WHERE/Join On
- Use table Country and City to find each country's capital city name. Hint: check the capital column in table Country, and find which column you should use in table City to join these 2 tables.
- Use table Country and City to find each country's capital city name, the population in the capital city and the percentage of capital city's population in the whole country. Hint: you need to use calculated field, and you may want to rename the column, so that the final table won't have 2 columns with the same name

Use Different Join Types – Outer Joins

- Outer Joins - the join includes table rows that have no associated rows in the related table

Type of Outer Joins	Output
Left 	All matching rows plus nonmatching rows from the first table(the left table) specified in the FROM clause
Right 	All matching rows plus nonmatching rows from the second table(the right table) specified in the FROM clause
Full Outer 	*All matching rows plus nonmatching rows in both tables*

Different Outer Joins – Left Join

One

X	A
1	a
2	**b**
4	d

Two

X	B
2	**x**
3	y
5	v

```
Proc sql;
Select * from
                one
left join
                two
on one.x=two.x
```

X	A	X	B
1	a	.	
2	**b**	**2**	**x**
4	d	.	

Different Outer Joins – Right Join

One

X	A
1	a
2	**b**
4	d

Two

X	B
2	**x**
3	y
5	v

```
Proc sql;
Select * from
            one
right join
            two
on one.x=two.x
```

X	A	X	B
2	b	2	x
.		3	y
.		5	v

Different Outer Joins – Full Outer Join

One

X	A
1	a
2	**b**
4	d

Two

X	B
2	**x**
3	y
5	v

```
Proc sql;
Select * from
              one
full outer join
              two
on one.x=two.x
```

Let us do it together:

X	A	X	B
1	**a**	**.**	
2	**b**	2	x
4	d	.	
		3	y
		5	v

The FULL OUTER JOIN syntax is not supported by MySQL

Use Different Join Types – Outer Joins

- Outer Joins - the join includes table rows that have no associated rows in the related table

The following SELECT statement is a simple inner join. It retrieves a list of all customers and their orders:

```
SELECT Customers.cust_id, Orders.order_num
FROM Customers INNER JOIN Orders
ON Customers.cust_id = Orders.cust_id;
```

- Outer join syntax is similar. To retrieve a list of all **customers including those who have placed no orders, you can do the following:**

```
SELECT Customers.cust_id, Orders.order_num
FROM Customers LEFT OUTER JOIN Orders
ON Customers.cust_id = Orders.cust_id;
```

Left Outer Joins

- Left Join **Syntax**

```
SELECT Customers.cust_id, Orders.order_num
FROM Customers LEFT OUTER JOIN Orders
ON Customers.cust_id = Orders.cust_id;
```

- A right outer join can be turned into a left outer join simply by reversing the order of the tables in the FROM clause
- You can still use WHERE clause after ON condition to do filtering

Use Joins with Aggregate Functions

- Aggregate functions can be used with JOINs

Let us retrieve a list of all customers and the number of orders that each has placed:

```
SELECT Customers.cust_id, COUNT(Orders.order_num) AS num_ord
FROM Customers INNER JOIN Orders
 ON Customers.cust_id = Orders.cust_id
GROUP BY Customers.cust_id;

SELECT Customers.cust_id, COUNT(Orders.order_num) AS num_ord
FROM Customers LEFT OUTER JOIN Orders
 ON Customers.cust_id = Orders.cust_id
GROUP BY Customers.cust_id;
```

Summary for JOINS

- Pay careful attention to the type of join being used. More often than not, you'll want an inner join, but there are often valid uses for outer joins, too
- Check your DBMS documentation for the exact join syntax it supports
- Make sure you use the correct join condition (regardless of the syntax being used), or you'll return incorrect data
- Make sure you always provide a join condition, or you'll end up with the Cartesian product
- You may include multiple tables in a join and even have different join types for each.

Combine Queries

Use Union

- There are basically two scenarios in which you'd use union queries:
  - To return similarly structured data from different tables in a single query
  - To perform multiple queries against a single table returning the data as one query

- Using UNION is simple enough. All you do is specify each SELECT statement and place the keyword UNION between each

  For example, you need a report on all your customers in Illinois, Indiana, and Michigan. You also want to include all Fun4ALL locations, regardless of state:

```
select cust_name, cust_contact, cust_email
from customers where cust_state in ('IL', 'IN','MI’)
UNION
select cust_name, cust_contact, cust_email
 from customers where cust_name = 'Fun4All’;
```

Use Union ALL

- The UNION automatically removes any duplicate rows from the query result set – this is the default behavior of UNION. Use UNION ALL to include all occurrences of all matches

```
select cust_name, cust_contact, cust_email
from customerswhere cust_state in ('IL', 'IN','MI')
UNION ALL
select cust_name, cust_contact, cust_email
 from customerswhere cust_name = 'Fun4All'
```

Sort Combined Results

- When combining queries with a UNION **only one Order BY** can be used. It must occur **after the final SELECT statement.**

```
select cust_name, cust_contact, cust_email
from customerswhere cust_state in ('IL', 'IN','MI’)
UNION ALL
select cust_name, cust_contact, cust_email
 from customerswhere cust_name = 'Fun4All’
ORDER BY cust_name,cust_contact;
```

Practice 4.2

Use SCHEMAS(Database) **world** to practice:

- For all questions in Practice 4.1, change to LEFT JOIN and check the differences
- Use table Country and Countrylanguage to find the **official** language used in each country. Hint: use column **Isofficial** and **WHERE** filter
- Count the number of different languages used in each country. I only need columns: country name, number of languages used. Hint: do not forget GROUP BY
- Some countries may have more than one types of official languages. Count the number of different **official** languages used in each country. I only need columns: country name, number of languages used
- Multiple table joins – show me the information as below:
  - country name
  - Different languages used in the country
  - for each language, how many people speak as column ‘language_pop’
  - official language or not
  - capital city name

Insert Data

Insert Into

- Insert is used to insert rows to a database table and it can be used in several ways:
  - Insert a single complete row
  - Insert a single partial row
  - Insert the result of a query

INSERT INTO customers
VALUES
('1000000006','Toy Land','123 Any Street', 'New York','NY','11111','USA',NULL,NULL);

- **The safer way to write the INSERT statement is as follows:**

INSERT INTO customers
(cust_id,cust_name,cust_address,cust_city,cust_state,cust_zip,cust_country,cust_contact,cust_email)
VALUES
('1000000007','Toy Land','123 Any Street', 'New York','NY','11111','USA',NULL,NULL);

Insert Retrieved Data

- **INSERT SELECT:**

INSERT INTO *customers*

(cust_id,cust_name,cust_address,cust_city,cust_state,cust_zip,cust_country,cust_contact,cust_email)

SELECT

cust_id,cust_name,cust_address,cust_city,cust_state,cust_zip,cust_country,cust_contact,cust_email

FROM *CustNew;*

Update and Delete Data

Update Data

- To update (modify) data in a table the UPDATE statement is used. UPDATE can be used in two ways:
  - To update specific rows in a table
  - To update all rows in a table

```
UPDATE customers
SET cust_email = 'kim@gmail.com’
WHERE cust_id = '1000000005’;
```

  - Without WHERE clause, the database will update all rows – be careful!
  - Update multiple columns:

```
UPDATE customers
SET cust_email = 'kim@gmail.com’,
        cust_contact = 'Sam Roberts'
WHERE cust_id = '1000000005’;
```

Delete Data

- To delete (remove) data in a table the DELETE statement is used. DELETE can be used in two ways:
  - To delete specific rows from a table
  - To delete all rows from a table

    DELETE FROM customers
    WHERE cust_id = '1000000005';
  - The DELETE statement deletes rows from the table, but never deletes the table itself
  - If you omit the WHERE clause, the DELETE will delete every row

Create Table

Create Table

- You can create one table from another by adding a SELECT statement at the end of the CREATE TABLE statement:

```
CREATE TABLE new_c AS
        SELECT * FROM customers;
```

Drop Table

Drop Table

- DROP TABLE removes one or more tables. You must have the DROP privilege for each table
- Be careful with this statement! It removes the table definition and all table data

```
DROP TABLE new_c;
```

- Use IF EXISTS to prevent an error from occurring for tables that do not exist

```
DROP TABLE IF EXISTS new_c ;
```

Derived Table

Derived Table

- A derived table is an expression that generates a table within the scope of a query FROM clause. For example, a subquery in a SELECT statement FROM clause is a derived table:

```
SELECT a.vend_id, b.vend_city  FROM
        (SELECT vend_id, COUNT(*) AS num_prods
                        FROM Products WHERE prod_price >= 4
                        GROUP BY vend_id    order by num_prods
                        Having num_prods >=2
                                                ) AS A
        LEFT JOIN
                        vendors as B
                                on a.vend_id=b.vend_id;
```

- The [AS] table_name clause is mandatory because every table in a FROM clause must have a name

Window Functions

Window Functions

- MySQL **8.0.2** introduces **SQL window functions**, or **analytic functions** as they are also sometimes called
- Similar to Group By aggregation, window functions perform some calculation on a set of rows, e.g. COUNT or SUM; However:
  - Group By aggregation collapses query rows in a single result row
  - A window function produces a result for each query row

- The general syntax:

```
WINDOW_FUNCTION_NAME(expression)
        OVER (
                [PARTITION_DEFINTION]
                [ORDER_DEFINITION]
                        )
```

Window Functions Examples

- Group By:

```
SELECT country, SUM(profit) AS
country_profit
    FROM sales
    GROUP BY country
    ORDER BY country;
```

```
+---------+----------------+
| country | country_profit |
+---------+----------------+
| Finland |           1610 |
| India   |           1350 |
| USA     |           4575 |
+---------+----------------+
```

- Window Functions

```
SELECT year, country, product, profit,
     SUM(profit) OVER() AS total_profit,
     SUM(profit) OVER(PARTITION BY country) AS country_profit
FROM sales
    ORDER BY country, year, product, profit;
```

```
+------+---------+------------+--------+--------------+----------------+
| year | country | product    | profit | total_profit | country_profit |
+------+---------+------------+--------+--------------+----------------+
| 2000 | Finland | Computer   |   1500 |         7535 |           1610 |
| 2000 | Finland | Phone      |    100 |         7535 |           1610 |
| 2001 | Finland | Phone      |     10 |         7535 |           1610 |
| 2000 | India   | Calculator |     75 |         7535 |           1350 |
| 2000 | India   | Calculator |     75 |         7535 |           1350 |
| 2000 | India   | Computer   |   1200 |         7535 |           1350 |
| 2000 | USA     | Calculator |     75 |         7535 |           4575 |
| 2000 | USA     | Computer   |   1500 |         7535 |           4575 |
| 2001 | USA     | Calculator |     50 |         7535 |           4575 |
| 2001 | USA     | Computer   |   1200 |         7535 |           4575 |
| 2001 | USA     | Computer   |   1500 |         7535 |           4575 |
| 2001 | USA     | TV         |    100 |         7535 |           4575 |
| 2001 | USA     | TV         |    150 |         7535 |           4575 |
+------+---------+------------+--------+--------------+----------------+
```

Window Functions Examples

- Window Functions

```
SELECT year, country, product, profit,
    SUM(profit) OVER() AS total_profit,
    SUM(profit) OVER(PARTITION BY country) AS country_profit FROM sales
  ORDER BY country, year, product, profit;
+------+---------+------------+--------+--------------+----------------+
| year | country | product    | profit | total_profit | country_profit |
+------+---------+------------+--------+--------------+----------------+
| 2000 | Finland | Computer   |   1500 |         7535 |           1610 |
| 2000 | Finland | Phone      |    100 |         7535 |           1610 |
| 2001 | Finland | Phone      |     10 |         7535 |           1610 |
| 2000 | India   | Calculator |     75 |         7535 |           1350 |
| 2000 | India   | Calculator |     75 |         7535 |           1350 |
| 2000 | India   | Computer   |   1200 |         7535 |           1350 |
| 2000 | USA     | Calculator |     75 |         7535 |           4575 |
| 2000 | USA     | Computer   |   1500 |         7535 |           4575 |
| 2001 | USA     | Calculator |     50 |         7535 |           4575 |
| 2001 | USA     | Computer   |   1200 |         7535 |           4575 |
| 2001 | USA     | Computer   |   1500 |         7535 |           4575 |
| 2001 | USA     | TV         |    100 |         7535 |           4575 |
| 2001 | USA     | TV         |    150 |         7535 |           4575 |
+------+---------+------------+--------+--------------+----------------+
```

Each window operation in the query is signified by inclusion of an OVER clause that specifies how to partition query rows into groups for processing by the window function

The first OVER clause is empty, which treats the entire set of query rows as a single partition. The window function thus produces a global sum, but does so for each row

The second OVER clause partitions rows by country, producing a sum per partition (per country). The function produces this sum for each partition row

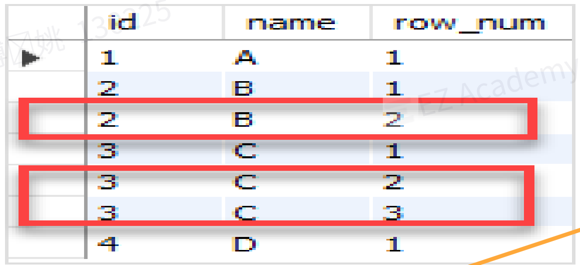
Window Functions Examples

- Note that window functions are performed on the result set after all JOIN, WHERE, GROUP BY, and HAVING clauses and before the ORDER BY
- The most commonly used Window Functions:

ROW_NUMBER()

Returns the number of the current row within its partition. Rows numbers range from 1 to the number of partition rows

SELECT id, name, ROW_NUMBER() OVER (PARTITION BY id, name ORDER BY name desc) AS row_num FROM t;



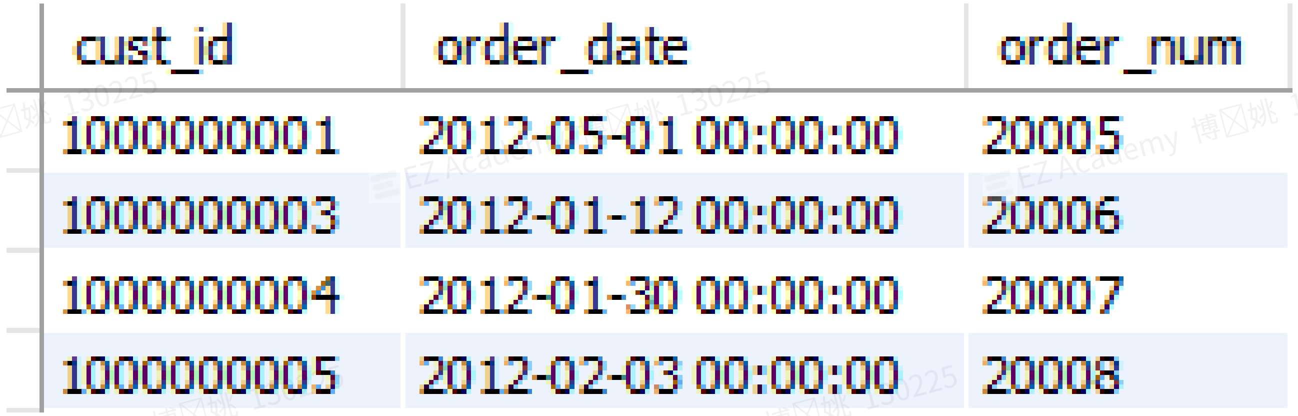
id	name	row_num
1	A	1
2	B	1
2	B	2
3	C	1
3	C	2
3	C	3
4	D	1

Select * from a
(SELECT id,
name,
ROW_NUMBER() OVER (PARTITION BY id, name ORDER BY name desc) AS row_num FROM t) as a
where a.row_num =1

As you can see from the output, the unique rows are the ones whose the row number equals one. Then, you can use *row_num=1* to **dedupe.**

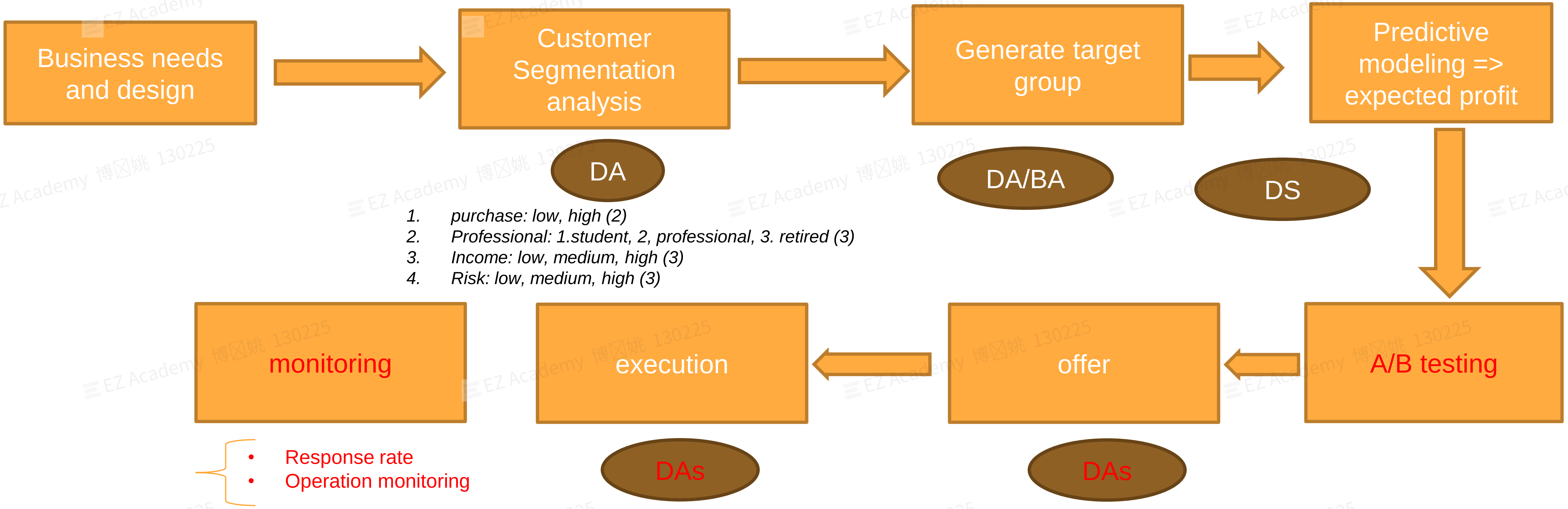
Window Functions Practice

- Use **row_number()** function to only output the **most recent order** for each customer ID in table Orders ; the output should look like this:



cust_id	order_date	order_num
1000000001	2012-05-01 00:00:00	20005
1000000003	2012-01-12 00:00:00	20006
1000000004	2012-01-30 00:00:00	20007
1000000005	2012-02-03 00:00:00	20008

Marketing Campaign


Business needs and design
Customer Segmentation analysis
DA
Generate target group
DA/BA
Predictive modeling => expected profit
DS
A/B testing
offer
DAs
execution
DAs
monitoring

1. purchase: low, high (2)
2. Professional: 1.student, 2, professional, 3. retired (3)
3. Income: low, medium, high (3)
4. Risk: low, medium, high (3)

- Response rate
- Operation monitoring

MySQL CTE(common table expression)

What is a common table expression (CTE)?

- A common table expression is a named temporary result set that **exists only within the execution scope of a single SQL statement** e.g., SELECT, INSERT, UPDATE, or DELETE.
  - ➢ Like a derive table, a CTE is not stored
  - ➢ Unlike a derive table, a CTE can be self-referencing; also, CTE has a better performance than derive tables

- The general syntax:

```
            WITH cte_name (column_list) AS (
    query
)
SELECT * FROM cte_name;
```

** Use schema classicmodels for CTE examples*

CTE example

```
WITH customers_in_usa AS (
    SELECT
        customerName, state
    FROM
        customers
    WHERE
        country = 'USA'
) SELECT
    customerName
 FROM
    customers_in_usa
 WHERE
    state = 'CA'
 ORDER BY customerName;
```

VS

```
SELECT  customerName from customers
where  state = 'CA' and country = 'USA'
order by customerName;
```

CTE example

```
WITH topsales2003 AS (
    SELECT
        salesRepEmployeeNumber as employeeNumber,
        SUM(quantityOrdered * priceEach) sales
    FROM
        orders
            INNER JOIN
        orderdetails USING (orderNumber)
            INNER JOIN
        customers USING (customerNumber)
    WHERE
        YEAR(shippedDate) = 2003
            AND status = 'Shipped'
    GROUP BY salesRepEmployeeNumber
    ORDER BY sales DESC
    LIMIT 5
)
SELECT
    employeeNumber,
    firstName,
    lastName,
    sales
FROM
    employees
        JOIN
    topsales2003 USING (employeeNumber);
```

If the join condition uses the equality operator (=) ***and the column names in both tables used for matching are the same****, and you can use the USING clause*

CTE example

```
WITH salesrep AS (
    SELECT
        employeeNumber,
        CONCAT(firstName, ' ', lastName) AS
salesrepName
    FROM
        employees
    WHERE
        jobTitle = 'Sales Rep'
),
customer_salesrep AS (
    SELECT
        customerName, salesrepName
    FROM
        customers
            INNER JOIN
        salesrep ON employeeNumber =
salesrepEmployeeNumber
)
SELECT
    *
FROM
    customer_salesrep
ORDER BY customerName;
```

In this example, we have two CTEs in the same query

MySQL recursive CTE

- A recursive common table expression (CTE) is a CTE that has a subquery which refers to the CTE name itself
    - Recursion: the process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function.

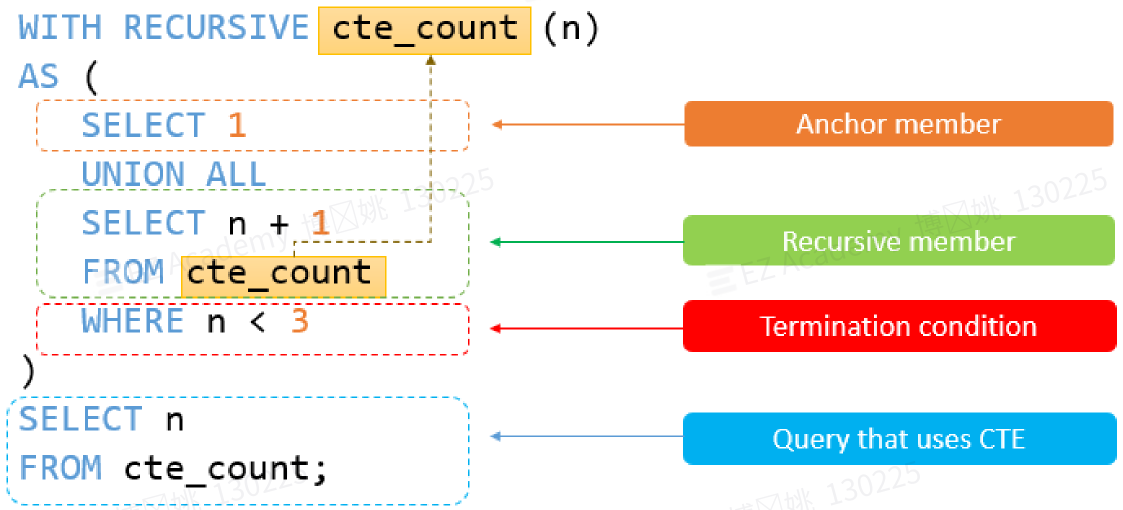
```
WITH RECURSIVE cte_name AS (
    initial_query  -- anchor member
    UNION ALL
    recursive_query -- recursive
member that references to the CTE
name
)
SELECT * FROM cte_name;
```

A recursive CTE consists of three main parts:

1. *An initial query that forms the base result set of the CTE structure. The initial query part is referred to as an anchor member.*
2. *A recursive query part is a query that references to the CTE name; therefore, it is called a recursive member. The recursive member is joined with the anchor member by a UNION ALL or UNION DISTINCT operator.*
3. *A termination condition that ensures the recursion stops when the recursive member returns no row.*

MySQL recursive CTE

- A recursive common table expression (CTE) is a CTE that has a subquery which refers to the CTE name itself
    - Recursion: the process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function.



```
WITH RECURSIVE cte_count (n)
AS (
   SELECT 1
   UNION ALL
   SELECT n + 1
   FROM cte_count
   WHERE n < 3
)
SELECT n
FROM cte_count;
```

The recursive member must not contain the following constructs:

- *Aggregate functions e.g., MAX, MIN, SUM, AVG, COUNT, etc.*
- *GROUP BY clause*
- *ORDER BY clause*
- *LIMIT clause*
- *DISTINCT*

In addition, the recursive member can only reference the CTE name once and in its FROM clause, not in any subquery

Use recursive CTE to traverse the hierarchical data

Apply the recursive CTE to query the whole organization structure in the top-down manner

```
WITH RECURSIVE employee_paths AS
  ( SELECT employeeNumber,
           reportsTo managerNumber,
           officeCode,
           1 as lvl
   FROM employees
   WHERE reportsTo IS NULL
     UNION ALL
     SELECT e.employeeNumber,
            e.reportsTo,
            e.officeCode,
            ep.lvl+1
     FROM employees e
     INNER JOIN employee_paths ep ON ep.employeeNumber =
e.reportsTo )
SELECT employeeNumber,
       managerNumber,
       lvl,
       city
FROM employee_paths ep
INNER JOIN offices o USING (officeCode)
ORDER BY lvl, city;
```

MySQL Stored Procedures

Introduction to MySQL Stored Procedures

- By definition, a stored procedure is a segment of declarative SQL statements stored inside the MySQL Server for execution later
- Once you save the stored procedure, you can invoke it by using the **CALL** statement
    - ➢ A stored procedure can have parameters so you can pass values to it and get the result back
    - ➢ A stored procedure may contain control flow statements such as IF, CASE, and LOOP that allow you to implement the code in the procedural way
    - ➢ A stored procedure can call other stored procedures or stored functions

Basic syntax of the CREATE PROCEDURE statement:

```
CREATE PROCEDURE procedure_name(parameter_list)
BEGIN
   statements;
END //
```

$\Rightarrow$

```
CALL stored_procedure_name(argument_list);
```

** Use schema classicmodels*

Introduction to MySQL Stored Procedures

```
DELIMITER $$

CREATE PROCEDURE GetCustomers()
BEGIN
        SELECT
                customerName,
                city,
                state,
                postalCode,
                country
        FROM
                customers
        ORDER BY customerName;
END$$
DELIMITER ;
```

```
CALL GetCustomers();
```

<u>Stored Procedures:</u>

Advantage	***Disadvantage***
• *Reduce network traffic*	• *Resource usages increase when there are too many stored procedures*
• *Reduce the efforts of duplicating the same logic*	• *It's difficult to debug stored procedures*
• *Make database more secure*	• *Need maintenance*

** Use schema classicmodels*

Create MySQL Stored Procedures

- Usually, we uses the delimiter (;) to separate statements and executes each statement separately;
- However, **typically, a stored procedure contains multiple statements separated by semicolons (;)**
- **To compile the whole stored procedure as a single compound statement, you need to temporarily change the delimiter from the semicolon (;) to another delimiter such as $$ or //:**

```
DELIMITER $$

CREATE PROCEDURE sp_name()
BEGIN
  -- statements
END $$

DELIMITER ;
```

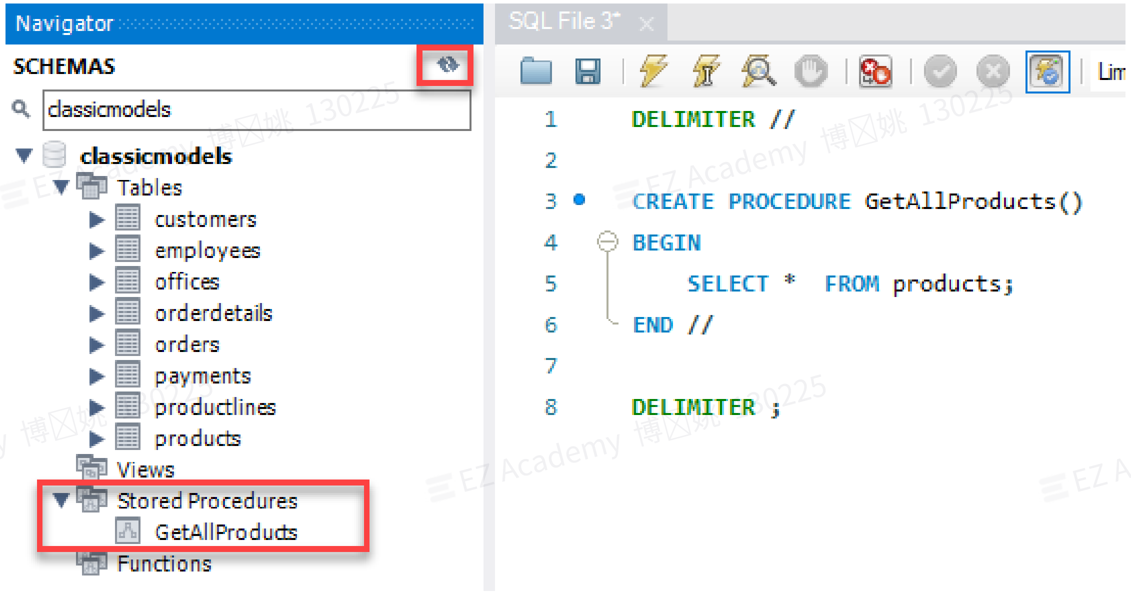
- *First, change the default delimiter to $$.*
- *Second, use (;) in the body of the stored procedure and $$ after the END keyword to end the stored procedure.*
- *Third, change the default delimiter back to a semicolon (;)*

Create MySQL Stored Procedures

Creates a new stored procedure that wraps the query which returns all products:

```
DELIMITER //

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

DELIMITER ;
```

1. *To create a new stored procedure, you use the CREATE PROCEDURE statement*
2. *Execute the statements, MySQL will create the stored procedure and save it in the server.*



```
DELIMITER //

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

DELIMITER ;
```

```
CALL GetAllProducts();
```

Drop Procedures

- The DROP PROCEDURE statement deletes a stored procedure created by the CREATE PROCEDURE statement

```
DROP PROCEDURE [IF EXISTS]
stored_procedure_name;
```

→

- *use IF EXISTS option to conditionally drop the stored procedure if it exists.*

Stored Procedures Practice

1. create a new stored procedure that returns employee and office information including columns:
   - firstName
   - lastName
   - city
   - state
   - country
2. Call the procedure and delete it

MySQL Stored Procedure Parameters

- The parameters make the stored procedure more useful and reusable
- A parameter in a stored procedure has one of three modes: IN,OUT, or INOUT.

1. IN parameters: when you define an IN parameter in a stored procedure, the calling program **must** pass an argument to the stored procedure.

2. OUT parameters: the value of an OUT parameter can be changed inside the stored procedure and its new value is passed back to the calling program; the stored procedure cannot access the initial value of the OUT parameter when it starts.

3. INOUT parameters: an INOUT parameter is a combination of IN and OUT parameters. It means that 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.

Syntax:

```
[IN | OUT | INOUT] parameter_name datatype[(length)]
```

MySQL Stored Procedure Parameters

1. IN parameters: when you define an IN parameter in a stored procedure, the calling program **must** pass an argument to the stored procedure.

```
DELIMITER //

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

DELIMITER ;
```

1. *The countryName is the IN parameter of the stored procedure*
2. *Suppose that you want to find offices locating in the USA, you need to pass an argument (USA) to the stored procedure as shown in the following query:*

```
CALL GetOfficeByCountry('USA');
```

MySQL Stored Procedure Parameters

2. OUT parameters: the value of an OUT parameter can be changed inside the stored procedure and its new value is passed back to the calling program; the stored procedure cannot access the initial value of the OUT parameter when it starts.

```
DELIMITER $$

CREATE PROCEDURE GetOrderCountByStatus (
        IN  orderStatus VARCHAR(25),
        OUT total INT
)
BEGIN
        SELECT COUNT(orderNumber)
        INTO total
        FROM orders
        WHERE status = orderStatus;
END$$

DELIMITER ;
```

1. *The orderStatus: is the IN parameter specifies the status of orders to return*
2. *The total: is the OUT parameter that stores the number of orders in a specific status*
3. *To find the number of orders that already shipped, you call GetOrderCountByStatus and pass the order status as of Shipped, and also pass a session variable (@total) to receive the return value.*

$\Downarrow$

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

MySQL Stored Procedure Parameters

3. INOUT parameters: an INOUT parameter is a combination of IN and OUT parameters. It means that 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.

```
DELIMITER $$

CREATE PROCEDURE SetCounter(
             INOUT counter INT,
    IN inc INT
)
BEGIN
             SET counter = counter + inc;
END$$

DELIMITER ;
```

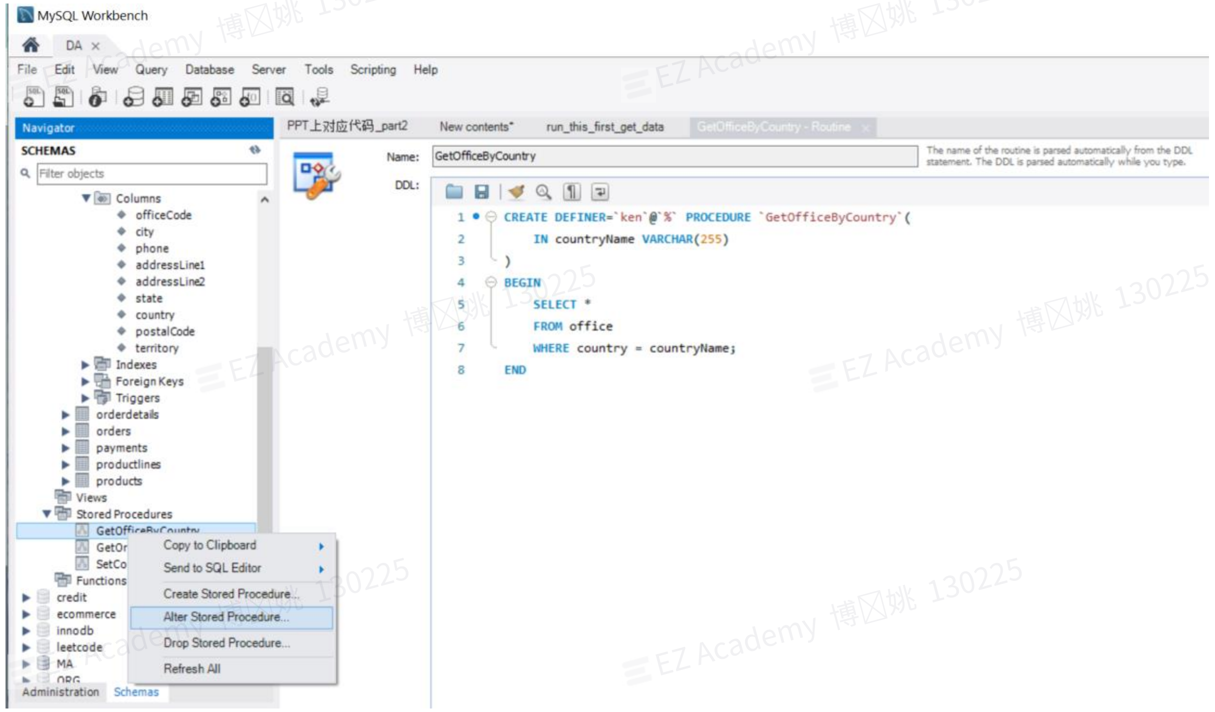
1. *the stored procedure SetCounter() accepts one INOUT parameter (counter) and one IN parameter (inc)*
2. *It increases the counter (counter) by the value of specified by the inc parameter.*

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

Alter Stored Procedures

- MySQL does not have any statement that allows you to directly modify the parameters and body of the stored procedure => To make such changes, you must drop ad re-create the stored procedure using the DROP PROCEDURE and CREATE PROCEDURE statements

  - MySQL Workbench provides a tool to change a stored procedure quickly



Name: GetOfficeByCountry

The name of the routine is parsed automatically from the DDL statement. The DDL is parsed automatically while you type.

DDL:

```
CREATE DEFINER=`ken`@`%` PROCEDURE `GetOfficeByCountry`(
    IN countryName VARCHAR(255)
)
BEGIN
    SELECT *
    FROM office
    WHERE country = countryName;
END
```

Stored Procedure Variables

- A **variable in stored procedure** is a named data object whose value can change during the stored procedure execution
- We use variables in stored procedures to hold immediate results. **These variables are local to the stored procedure**

  ➢ To declare local variables, use the DECLARE statement:

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

  ➢ **The DECLARE statement** is used to define various items **local to a program!**

  - DECLARE is permitted only inside a BEGIN ... END compound statement and must be at its start, before any other statements.

```
DELIMITER $$

CREATE PROCEDURE GetTotalOrder()
BEGIN
              DECLARE totalOrder INT DEFAULT 0;

   SELECT COUNT(*)
   INTO totalOrder
   FROM orders;

   SELECT totalOrder;
END$$

DELIMITER ;
```

Stored Procedure Variables

- **SET statement** enables you to assign values to different types of variables:
  - User-defined variables
  - Stored procedure and function parameters, and stored program local variables
  - System variables.

```
SET variable_name = value;
```

Variables can be set directly with the SET statement

Stored Procedure Variables

- **SET statement** enables you to assign values to different types of variables:
  - **User-Defined Variable** Assignment:
    - User-defined variables are created locally within a session and exist only within the context of that session
    - A user-defined variable is written as @var_name and is assigned an expression value as follows:

```
SET @var_name = expr;
```

Only user-defined variables need @

Stored Procedure Variables

- **SET statement** enables you to assign values to different types of variables:
  - **Parameter and Local Variable** Assignment:
    - SET applies to parameters and local variables in the context of the stored object within which they are defined:

```
DELIMITER $$

CREATE PROCEDURE SetCounter( IN inc INT)
          BEGIN
                    DECLARE counter INT DEFAULT 0;
                    SET counter = counter + inc;
END$$

DELIMITER ;
```

List Stored Procedures

- The SHOW PROCEDURE STATUS statement shows all characteristic of stored procedures including stored procedure names. It returns stored procedures that you have a privilege to access.
  - The following statement shows all stored procedure in the current MySQL server:

```
SHOW PROCEDURE STATUS;
```

  - If you just want to show stored procedures in a particular database

```
SHOW PROCEDURE STATUS WHERE db = 'classicmodels';
```

MySQL IF Statement in Stored Procedures

1. IF-THEN statement

```
IF condition THEN
  statements;
END IF;
```

An example:

```
DELIMITER $$

CREATE PROCEDURE GetCustomerLevel(
   IN  pCustomerNumber INT,
   OUT pCustomerLevel  VARCHAR(20))
BEGIN
   DECLARE credit DECIMAL(10,2) DEFAULT 0;

   SELECT creditLimit
   INTO credit
   FROM customers
   WHERE customerNumber = pCustomerNumber;

   IF credit > 50000 THEN
      SET pCustomerLevel = 'PLATINUM';
   END IF;
END$$

DELIMITER ;
```

MySQL supports the IF, CASE, ITERATE, LEAVE LOOP, WHILE, and REPEAT constructs for flow control within stored programs!

```
CALL GetCustomerLevel(141, @pCustomerLevel);
SELECT @pCustomerLevel;
```

MySQL IF Statement in Stored Procedures

2. IF-THEN-ELSE statement

```
IF condition THEN
  statements;
ELSE
  else-statements;
END IF;
```

An example:

```
DROP PROCEDURE IF EXISTS GetCustomerLevel;
DELIMITER $$

CREATE PROCEDURE GetCustomerLevel(
    IN  pCustomerNumber INT,
    OUT pCustomerLevel  VARCHAR(20))
BEGIN
    DECLARE credit DECIMAL DEFAULT 0;

    SELECT creditLimit
    INTO credit
    FROM customers
    WHERE customerNumber = pCustomerNumber;

    IF credit > 50000 THEN
        SET pCustomerLevel = 'PLATINUM';
    ELSE
        SET pCustomerLevel = 'NOT PLATINUM';
    END IF;
END$$

DELIMITER ;
```

```
CALL GetCustomerLevel(447, @pCustomerLevel);
SELECT @pCustomerLevel;
```

MySQL IF Statement in Stored Procedures

3. IF-THEN-ELSEIF-ELSE statement

```
IF condition THEN
   statements;
ELSEIF elseif-condition THEN
   elseif-statements;
...
ELSE
   else-statements;
END IF;
```

An example:

```
DROP PROCEDURE IF EXISTS GetCustomerLevel;
DELIMITER $$

CREATE PROCEDURE GetCustomerLevel(
    IN  pCustomerNumber INT,
    OUT pCustomerLevel  VARCHAR(20))
BEGIN
    DECLARE credit DECIMAL DEFAULT 0;

    SELECT creditLimit
    INTO credit
    FROM customers
    WHERE customerNumber = pCustomerNumber;

    IF credit > 50000 THEN
        SET pCustomerLevel = 'PLATINUM';
    ELSEIF credit <= 50000 AND credit > 10000 THEN
        SET pCustomerLevel = 'GOLD';
    ELSE
        SET pCustomerLevel = 'SILVER';
    END IF;
END $$

DELIMITER ;
```

```
CALL GetCustomerLevel(447, @pCustomerLevel);
SELECT @pCustomerLevel;
```

MySQL CASE Statement

- Besides the IF statement, MySQL provides an alternative conditional statement called the **CASE statement for constructing conditional statements in stored procedures**
- The following is the basic syntax of the simple CASE statement:

```
CASE case_value
  WHEN when_value1 THEN
statements
  WHEN when_value2 THEN
statements
  ...
  [ELSE else-statements]
END CASE;
```

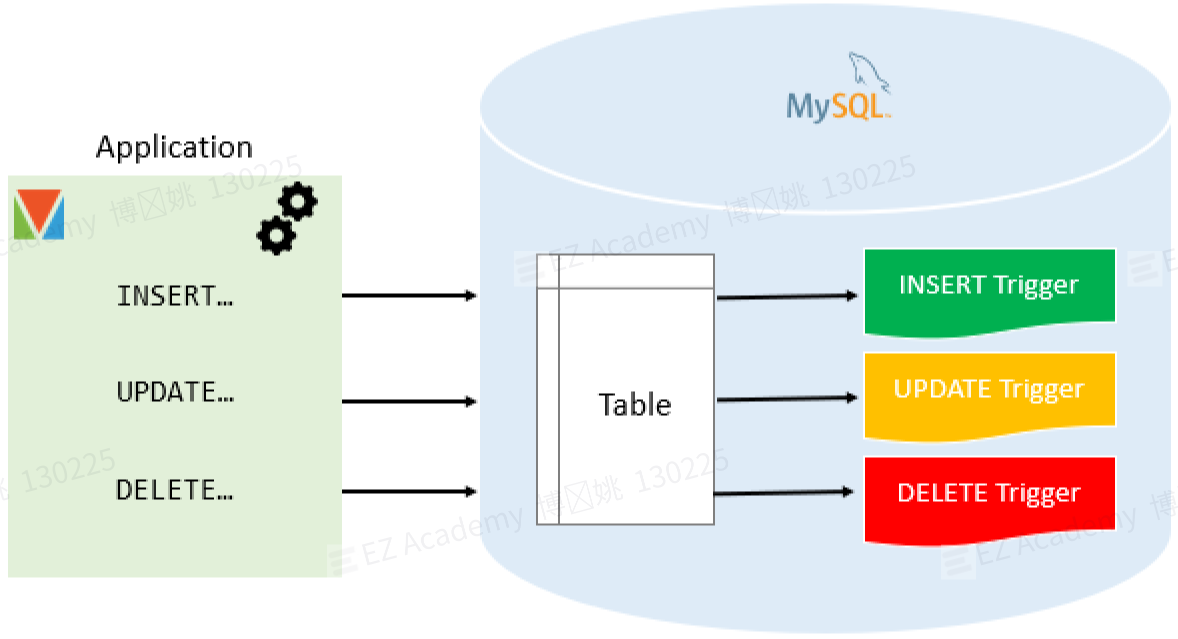
An example:

```
DELIMITER $$

CREATE PROCEDURE GetCustomerShipping(
                 IN  pCustomerNUmber INT,
                 OUT pShipping       VARCHAR(50)
)
BEGIN
   DECLARE customerCountry VARCHAR(100);

SELECT
   country
INTO customerCountry FROM
   customers
WHERE
   customerNumber = pCustomerNUmber;

   CASE customerCountry
                                    WHEN  'USA' THEN
                                      SET pShipping = '2-day Shipping';
                                    WHEN 'Canada' THEN
                                      SET pShipping = '3-day Shipping';
                                    ELSE
                                      SET pShipping = '5-day Shipping';
                 END CASE;
END$$

DELIMITER ;
```

MySQL Triggers

MySQL Triggers

- In MySQL, a trigger is a stored program invoked automatically in response to an event such as insert, update, or delete that occurs in the associated table
- MySQL supports triggers that are invoked in response to the INSERT, UPDATE or DELETE event
- **A row-level trigger is activated for each row that is inserted, updated, or deleted** *(For example, if a table has 100 rows inserted, updated, or deleted, the trigger is automatically invoked 100 times for the 100 rows affected) => MySQL supports only row-level triggers.*



Triggers:

Advantage	***Disadvantage***
• *Provide another way to check the integrity of data.*	• *Triggers can be difficult to troubleshoot because they execute automatically in the database*
• *Triggers give an alternative way to run scheduled tasks*	• *It may increase the overhead of the MySQL Server.*
• *Triggers can be useful for auditing the data changes in tables*	

Create Triggers

Basic syntax of the CREATE TRIGGER statement:

```
CREATE TRIGGER trigger_name
{BEFORE | AFTER} {INSERT | UPDATE| DELETE }
ON table_name FOR EACH ROW
trigger_body;
```

- *First, specify the name of the trigger after the CREATE TRIGGER keywords (trigger name must be unique within a database)*
- *Next, specify the trigger action time which can be either* ***BEFORE*** *or* ***AFTER*** *which indicates that the trigger is invoked before or after each row is modified*
- *Then, specify the operation that activates the trigger, which can be INSERT, UPDATE, or DELETE*
- *After that, specify the name of the table to which the trigger belongs after the ON keyword*
- *Finally, specify the statement to execute when the trigger activates. If you want to execute multiple statements, you use the BEGIN END compound statement*
- *To distinguish between the value of the columns BEFORE and AFTER the modify, you use the NEW and OLD modifiers:*

The availability of the OLD and NEW modifiers for different types of modifies:

Tigger Event	OLD	NEW
INSERT	NO	YES
UPDATE	YES	YES
DELETE	YES	NO

Create Triggers

Create a trigger in MySQL to log the changes of the employees table:

- First, create a new table named employees_audit to keep the changes to the employees table:

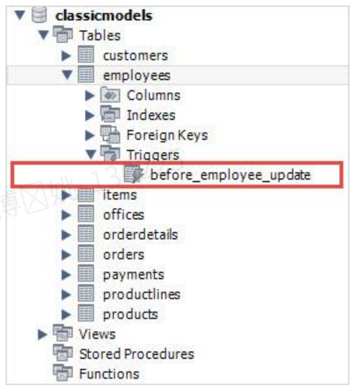
```
CREATE TABLE employees_audit (
    id INT AUTO_INCREMENT PRIMARY
KEY,
    employeeNumber INT NOT NULL,
    lastname VARCHAR(50) NOT NULL,
    changedat DATETIME DEFAULT NULL,
    action VARCHAR(50) DEFAULT NULL
);
```

- *Next, create a BEFORE UPDATE trigger that is invoked before a change is made to the employees table*
- *Inside the body of the trigger, we used the OLD keyword to access values of the columns employeeNumber and lastname of the row affected by the trigger*

```
CREATE TRIGGER before_employee_update
    BEFORE UPDATE ON employees
    FOR EACH ROW
 INSERT INTO employees_audit
 SET action = 'update',
     employeeNumber = OLD.employeeNumber,
     lastname = OLD.lastname,
     changedat = NOW();
```

- *Then, show all triggers in the current database by using the SHOW TRIGGERS statement:*

```
SHOW TRIGGERS;
```



```
▼ classicmodels
  ▼ Tables
    ▶ customers
    ▼ employees
      ▶ Columns
      ▶ Indexes
      ▶ Foreign Keys
      ▼ Triggers
          before_employee_update
    ▶ items
    ▶ offices
    ▶ orderdetails
    ▶ orders
    ▶ payments
    ▶ productlines
    ▶ products
  ▶ Views
    Stored Procedures
    Functions
```

** Use schema classicmodels*

Create Triggers

Create a trigger in MySQL to log the changes of the employees table:

- After creating the trigger, let's update a row in the employees table:

```
UPDATE employees
SET
    lastName = 'Phan'
WHERE
    employeeNumber = 1056;
```

- Finally, query the employees_audit table to check if the trigger was fired by the UPDATE statement:

```
SELECT * FROM employees_audit;
```

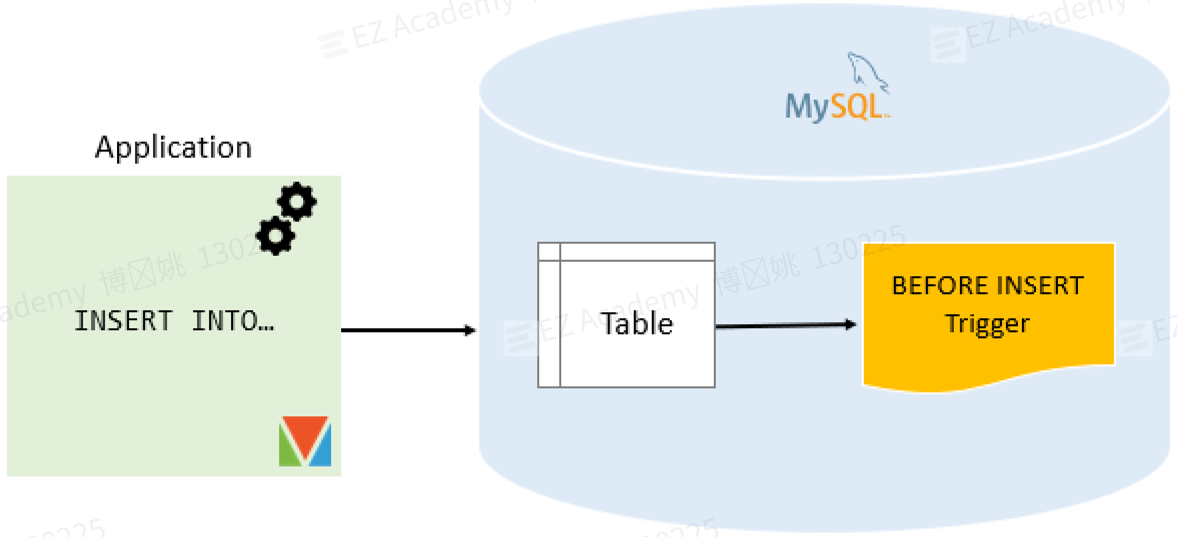
Drop Triggers

- The DROP TRIGGER statement deletes a trigger from the database
- If you drop a table, MySQL will automatically drop all triggers associated with the table.

```
DROP TRIGGER [IF EXISTS]
[schema_name.]trigger_name;
```

MySQL BEFORE INSERT Triggers

- BEFORE INSERT triggers are automatically fired before an insert event occurs on the table



*In a BEFORE INSERT trigger, **you can access and change the NEW values.** However, you cannot access the OLD values because OLD values obviously do not exist.*

MySQL BEFORE INSERT Triggers

Create a BEFORE INSERT trigger to maintain a summary table from another table:

- First, create a new table called WorkCenters:

```
DROP TABLE IF EXISTS
WorkCenters;

CREATE TABLE WorkCenters (
    id INT AUTO_INCREMENT
PRIMARY KEY,
    name VARCHAR(100) NOT
NULL,
    capacity INT NOT NULL
);
```

$\Rightarrow$

- Second, create another table called WorkCenterStats that stores the summary of the capacity of the work centers:

```
DROP TABLE IF EXISTS
WorkCenterStats;

CREATE TABLE
WorkCenterStats(
    totalCapacity INT NOT NULL
);
```

MySQL BEFORE INSERT Triggers

Create a BEFORE INSERT trigger to maintain a summary table from another table:

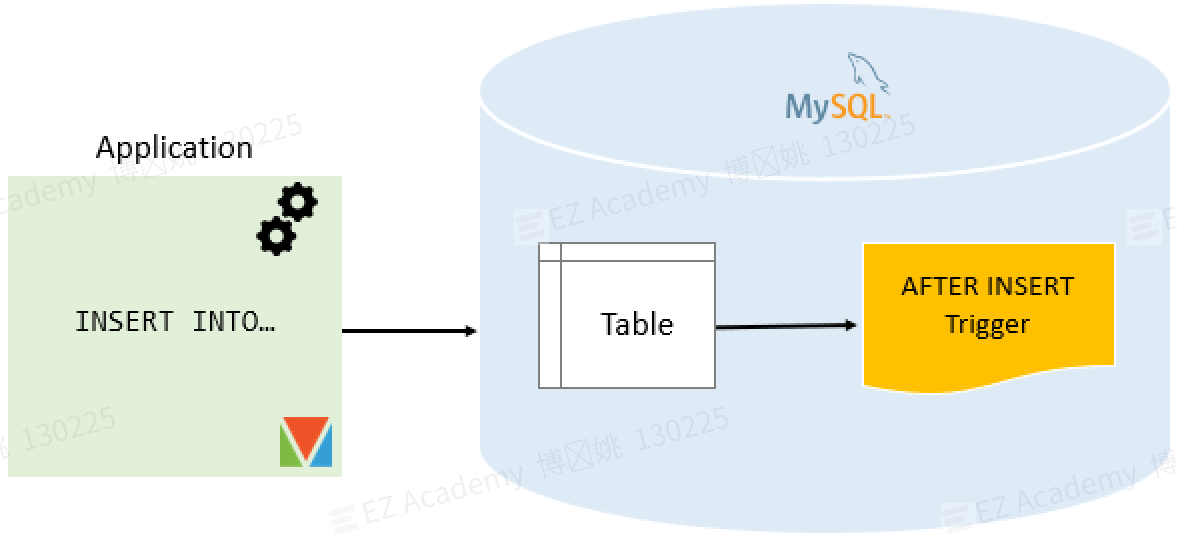
- The following trigger updates the total capacity in the WorkCenterStats table before a new work center is inserted into the WorkCenter table

```
DELIMITER $$

CREATE TRIGGER before_workcenters_insert
BEFORE INSERT
ON WorkCenters FOR EACH ROW
BEGIN
   DECLARE rowcount INT;

   SELECT COUNT(*)
   INTO rowcount
   FROM WorkCenterStats;

   IF rowcount > 0 THEN
      UPDATE WorkCenterStats
      SET totalCapacity = totalCapacity + new.capacity;
   ELSE
      INSERT INTO WorkCenterStats(totalCapacity)
      VALUES(new.capacity);
   END IF;

END $$

DELIMITER ;
```

MySQL AFTER INSERT Triggers

- AFTER INSERT triggers are automatically invoked after an insert event occurs on the table.

```
CREATE TRIGGER trigger_name
    AFTER INSERT
    ON table_name FOR EACH ROW
        trigger_body;
```



*In an AFTER INSERT trigger, **you can access the NEW values but you cannot change them.** Also, you cannot access the OLD values because there is no OLD on INSERT triggers.*

MySQL AFTER INSERT Triggers

Create an AFTER INSERT trigger to insert data into a table after inserting data into another table:

- First, create a new table called members:

```
DROP TABLE IF EXISTS
members;

CREATE TABLE members (
   id INT AUTO_INCREMENT,
   name VARCHAR(100) NOT
NULL,
   email VARCHAR(255),
   birthDate DATE,
   PRIMARY KEY (id)
);
```

- Second, create another table called reminders that stores reminder messages to members:

```
DROP TABLE IF EXISTS
reminders;

CREATE TABLE reminders (
   id INT AUTO_INCREMENT,
   memberId INT,
   message VARCHAR(255) NOT
NULL,
   PRIMARY KEY (id , memberId)
);
```

MySQL AFTER INSERT Triggers

Create an AFTER INSERT trigger to insert data into a table after inserting data into another table:

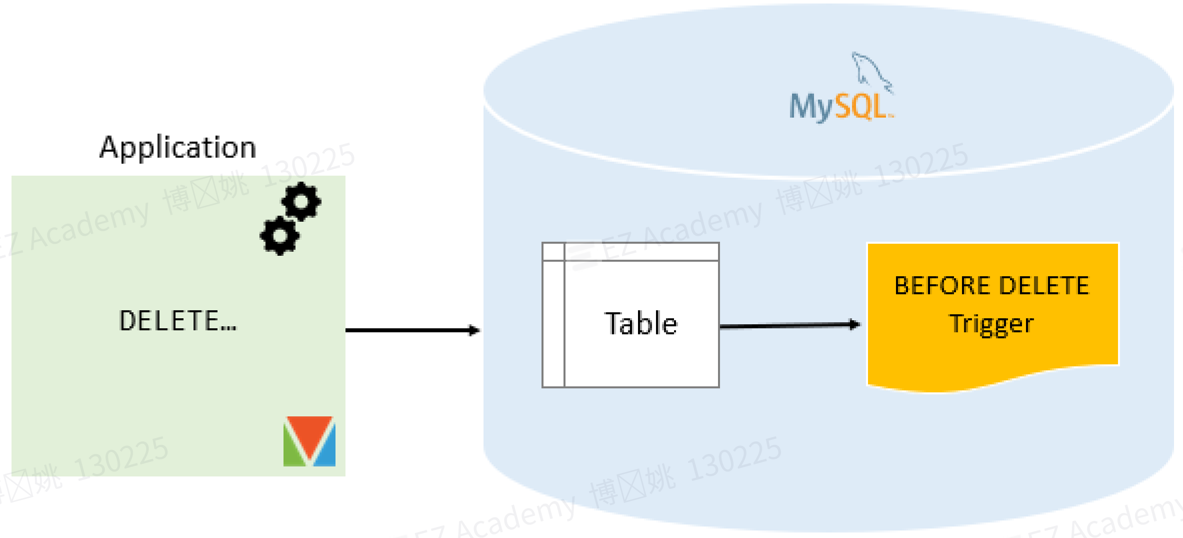
- The following statement creates an AFTER INSERT trigger that inserts a reminder into the reminders table if the birth date of the member is NULL

```
DELIMITER $$

CREATE TRIGGER after_members_insert
AFTER INSERT
ON members FOR EACH ROW
BEGIN
    IF NEW.birthDate IS NULL THEN
        INSERT INTO reminders(memberId, message)
        VALUES(new.id,CONCAT('Hi ', NEW.name, ', please
update your date of birth.'));
    END IF;
END$$

DELIMITER ;
```

MySQL BEFORE DELETE Triggers

- BEFORE DELETE triggers are fired automatically before a delete event occurs in a table

```
CREATE TRIGGER trigger_name
    BEFORE DELETE
    ON table_name FOR EACH ROW
trigger_body;
```



In a BEFORE DELETE trigger, you can access the OLD row but cannot update it. Also, there is no NEW row in the BEFORE DELETE trigger

MySQL BEFORE DELETE Triggers

Create a BEFORE DELETE trigger to add deleted rows into an archive table

- First, create a new table called Salaries that stores salary information of employees:

```
DROP TABLE IF EXISTS Salaries;

CREATE TABLE Salaries (
    employeeNumber INT PRIMARY
KEY,
    validFrom DATE NOT NULL,
    amount DEC(12 , 2 ) NOT NULL
DEFAULT 0
);
```

- Second, insert some rows into the Salaries table:

```
INSERT INTO
Salaries(employeeNumber,validFr
om,amount)
VALUES
    (1002,'2000-01-01',50000),
    (1056,'2000-01-01',60000),
    (1076,'2000-01-01',70000);
```

MySQL BEFORE DELETE Triggers

Create a BEFORE DELETE trigger to add deleted rows into an archive table

- Third, create a table that stores the deleted salary:

```
DROP TABLE IF EXISTS SalaryArchives;

CREATE TABLE SalaryArchives (
   id INT PRIMARY KEY AUTO_INCREMENT,
   employeeNumber INT,
   validFrom DATE NOT NULL,
   amount DEC(12 , 2 ) NOT NULL DEFAULT 0,
   deletedAt TIMESTAMP DEFAULT NOW()
);
```

- The following BEFORE DELETE trigger inserts a new row into the SalaryArchives table before a row from the Salaries table is deleted:

```
DELIMITER $$

CREATE TRIGGER before_salaries_delete
BEFORE DELETE
ON Salaries FOR EACH ROW
BEGIN
    INSERT INTO
SalaryArchives(employeeNumber,validFrom,amount)

VALUES(OLD.employeeNumber,OLD.validFrom,OLD.amount);
END$$

DELIMITER ;
```

Create Multiple Triggers

After MySQL 5.7.2, we can create multiple triggers for a given table that have the same event and action time. These triggers will activate sequentially when an event occurs.

```
DELIMITER $$

CREATE TRIGGER trigger_name
{BEFORE|AFTER}{INSERT|UPDATE|DELETE}
ON table_name FOR EACH ROW
{FOLLOWS|PRECEDES} existing_trigger_name
BEGIN
    -- statements
END$$

DELIMITER ;
```

- The **FOLLOWS** allows the new trigger to activate after an existing trigger.
- The **PRECEDES** allows the new trigger to activate before an existing trigger.

- MySQL CTE(common table expression) ✓
- MySQL Stored Procedures ✓
- MySQL Triggers ✓

Easy Career出品

Thank You