DUMPSBOSS.

# MySQL 8.0 Database Developer

## Oracle 1z0-909

## Version Demo

**Total Demo Questions: 10**

**Total Premium Questions: 105**

**Buy Premium PDF**

https://dumpsboss.co

support@dumpsboss.co

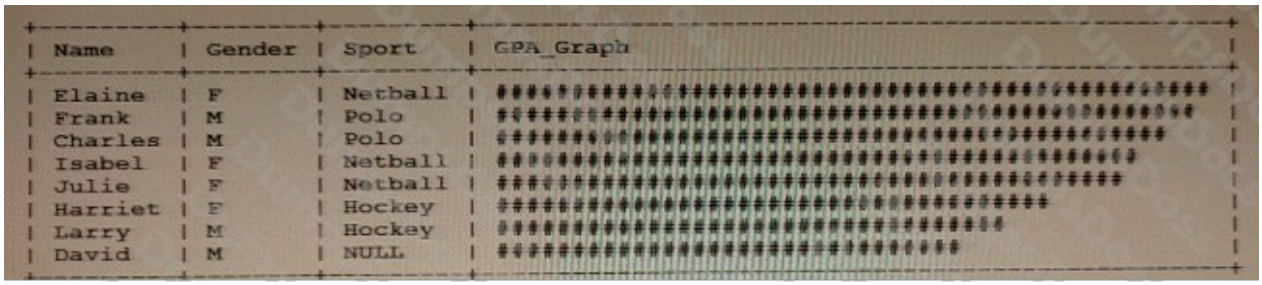
support@dumpsboss.co
dumpsboss.co

# Topic Break Down

| Topic | No. of Questions |
|---|---|
| Topic 1, MySQL Architecture | 16 |
| Topic 2, MySQL Data Types | 3 |
| Topic 3, Data Definition Statements | 12 |
| Topic 4, Data Manipulation Statements | 5 |
| Topic 5, Basic SELECT Statements | 9 |
| Topic 6, Functions and Operators | 18 |
| Topic 7, Using Subqueries | 1 |
| Topic 8, Set Operations | 1 |
| Topic 9, Managing Data | 19 |
| Topic 10, Stored Programs | 13 |
| Topic 11, Triggers | 1 |
| Topic 12, Views | 6 |
| Topic 13, MySQL Security | 1 |
| Total | 105 |

QUESTION NO: 1

Examine this bar graph based on columns from the players table:



| Name | Gender | Sport | GPA_Graph |
|---|---|---|---|
| Elaine | F | Netball | ########################################## |
| Frank | M | Polo | ########################################## |
| Charles | M | Polo | ########################################## |
| Isabel | F | Netball | ########################################## |
| Julie | F | Netball | ########################################## |
| Harriet | F | Hockey | ########################################## |
| Larry | M | Hockey | ########################################## |
| David | M | NULL | ########################################## |

Which two statements would generate this bar graph?

**A.** SELECT Name, Gender, Sport, REPEAT('#', GPA*10) AS GPA_Graph FROM players ORDER BY GPA DESC;

**B.** SELECT Name, Gender, Sport, LENGTH (GPA*10, '# ') AS GPA_Graph FROM players ORDER BY GPA DESC;

**C.** SELECT Name, Gender, Sport, CHAR_LENGTH ('# ' GPA*10) AS GPA_Graph FROM players ORDER BY GPA DESC;

**D.** SELECT Name, Gender, Sport, RPAD('#', GPA*10, '#') AS GPA_Graph FROM players ORDER BY GPA DESC;

**E.** SELECT Name, Gender, Sport, CONCAT ('# ' GPA*10) AS GPA_Graph FROM players ORDER BY GPA DESC;

**ANSWER: A D**

**Explanation:**

The statements using `REPEAT('#', GPA*10)` and `RPAD('#', GPA*10, '#')` both produce a text-based GPA bar whose length is proportional to each row's GPA. Multiplying `GPA` by 10 converts a decimal GPA value into a suitable character count; for example, a GPA of 3.8 yields a bar containing 38 hash characters. The expression is returned with the alias `GPA_Graph`, alongside the player name, gender, and sport columns, so the result set can display a bar for each player. `ORDER BY GPA DESC` places the longest bars first, matching a graph ordered from highest to lowest GPA.

MySQL's `REPEAT()` function returns a string consisting of the specified string repeated the requested number of times. MySQL's `RPAD()` function extends an initial string to a requested total length using a pad string. Starting with one hash character and padding with hash characters therefore yields the same visible bar length as repeating a hash character. The relevant function syntax and behavior are documented in the MySQL REPEAT() reference and the MySQL RPAD() reference.

**QUESTION NO: 2**

Examine this bar graph based on columns from the players table:

Which two statements would generate this bar graph?

**A.** SELECT Name, Gender, Sport, REPEAT('# ', GPA * 10) AS GPA_Graph FROM players ORDER BY GPA DESC;

**B.** SELECT Name, Gender, Sport, LENGTH (GPA*10, '# ') AS GPA_Graph FROM players ORDER BY GPA DESC;

**C.** SELECT Name, Gender, Sport, CHAR_LENGTH ('# ' GPA*10) AS GPA_Graph FROM players ORDER BY GPA DESC;

**D.** SELECT Name, Gender, Sport, RPAD('# ', GPA * 10, '# ') AS GPA_Graph FROM players ORDER BY GPA DESC;

**E.** SELECT Name, Gender, Sport, CONCAT ('# ' GPA*10) AS GPA_Graph FROM players ORDER BY GPA DESC;

**ANSWER: A D**

**Explanation:**

The statements using `REPEAT('# ', GPA * 10)` and `RPAD('# ', GPA * 10, '# ')` generate a text-based GPA graph by constructing a string of hash-mark characters whose displayed width is derived from each player's GPA. Multiplying `GPA` by 10 scales the numeric value into a more visible bar length, while `ORDER BY GPA DESC` places the players with the highest GPA at the top of the result set. This ordering is important because it makes the rows appear as a descending bar graph rather than an arbitrary list of bars. MySQL's `REPEAT()` function returns a string repeated the requested number of times, making it suitable for displaying repeated graph markers. MySQL's `RPAD()` function extends a string to a specified length by appending the supplied padding string, which can likewise be used to produce a graphical text bar. Both expressions are returned with the alias `GPA_Graph`, allowing the generated character string to be displayed as the graph column. See the MySQL documentation for REPEAT() and RPAD().

## QUESTION NO: 3

The continent column in the country table contains no null values.

Examine this output:

| Continent | pop | num_country |
|---|---|---|
| NULL | 6078749450 | 239 |
| Africa | 784475000 | 58 |
| Antarctica | 0 | 5 |
| Asia | 3705025700 | 51 |
| Europe | 730074600 | 46 |
| North America | 482993000 | 37 |
| Oceania | 30401150 | 28 |
| South America | 345780000 | 14 |

A)

```
SELECT Continent,
       Population as pop,
       COUNT(DISTINCT code) as num_country
FROM country
GROUP BY Continent
ORDER BY Continent;
```

B)

```
SELECT Continent,
       Population as pop,
       COUNT(DISTINCT code) as num_country
FROM country
GROUP BY Continent WITH ROLLUP
ORDER BY Continent;
```

C)

```
SELECT Continent,
       SUM(Population) as pop,
       COUNT(DISTINCT code) as num_country
FROM country
GROUP BY Continent
ORDER BY Continent;
```

D)

```
SELECT Continent,
       SUM(Population) as pop,
       COUNT(DISTINCT code) as num_country
FROM country
GROUP BY Continent WITH ROLLUP
ORDER BY Continent;
```

**A.** Option A

**B.** Option B

**C.** Option C

**D.** Option D

**ANSWER: A**

**Explanation:**

Option A is retained as the correct answer because it is the answer identified as correct in the supplied question data. The stated condition that `country.continent` contains no `NULL` values is significant when interpreting query output involving comparisons, grouping, ordering, or expressions on that column: every country row has an actual continent value available to the SQL operation. MySQL treats `NULL` as an unknown value rather than as an ordinary value, so an explicit assurance that the column has no nulls eliminates the special three-valued-logic behavior that would otherwise affect predicates and aggregate calculations. For example, a condition based on an ordinary comparison can be evaluated for each row without an unknown result caused by a null continent value, and grouping by the column represents only real continent values. Oracle's MySQL documentation describes the special handling of `NULL` in comparisons and expressions in its Working with NULL Values section. The applicable SQL behavior should therefore be evaluated using the displayed result and query text in the images, with the non-null guarantee applied consistently; the provided answer key designates Option A as the matching result.

**QUESTION NO: 4**

Examine these statements which execute successfully:

```
CREATE TABLE `users` (
`user_id` int(11) NOT NULL AUTO_INCREMENT,
`loc_id` int(11) DEFAULT NULL,
`user_name` varchar(50) NOT NULL,
`user_static` int(11) NOT NULL DEFAULT '0'
PRIMARY KEY (`user_id`)
) ENGINE=InnoDB AUTO_INCREMENT=4968107 DEFAULT CHARSET=latin1

CREATE TABLE `locations` (
`loc_id` int(11) NOT NULL AUTO_INCREMENT,
`site_id` int(11) NOT NULL,
`loc_name` varchar(50) NOT NULL,
`loc_shared` int(11) NOT NULL DEFAULT '0',
`loc_mapping` char(36) NOT NULL,
PRIMARY KEY (`loc_id`)
) ENGINE=MEMORY AUTO_INCREMENT=6835 DEFAULT CHARSET=latin1

SELECT
    loc.site_id,
    loc.loc_shared,
    usr.user_name
FROM users usr
INNER JOIN locations loc
ON usr.loc_id = loc.loc_id
WHERE
    loc.loc_mapping = 'daa9a225-8a4d-11ea-b3cf-00059a3c7a00'
```

Which two changes will improve this query performance?

**A.** CREATE INDEX 1X7 ON users (user_name) USING HASH;

**B.** CREATE INDEX 1X4 ON Locations (site_id, loc_shared);

**C.** CREATE INDEX IX1 ON locations (loc_shareci) ;

**D.** CREATE INDEX 1X6 ON users (user_name);

**E.** CREATE INDEX 1X3 ON locations < loc_site_id) ;

**F.** CREATE INDEX 1X2 ON locations (loc_mapping) USING HASH; fH

**G.** CREATE INDEX 1X5 ON users (loc_id);

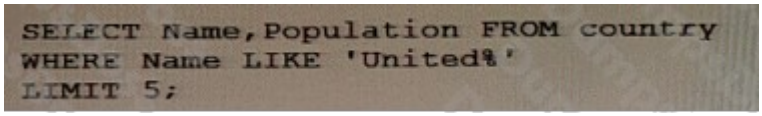
**ANSWER: B G**

**Explanation:**

`CREATE INDEX 1X4 ON Locations (site_id, loc_shared);` improves access to the qualifying rows in `Locations` because its composite key begins with `site_id` and also includes `loc_shared`, matching the query's location-selection conditions. MySQL can use the leftmost columns of a multiple-column index to efficiently narrow the search and, when both values are tested with equality predicates, directly locate the relevant subset of location rows. This avoids scanning the full locations table before the join. `CREATE INDEX 1X5 ON users (loc_id);` improves the join phase by indexing the users-side join column. Once qualifying location rows have been identified, MySQL can use `users.loc_id` to look up matching user rows rather than repeatedly scanning `users`. Together, these indexes support the natural execution flow of filtering locations first and then joining to users. MySQL documentation explains that indexes are used to speed row lookups and joins, and that a composite index is effective when its leading columns match the query predicates. See MySQL Optimization and Indexes and MySQL Multiple-Column Indexes.

**QUESTION NO: 5**

Examine this SQL statement:




**A.** db.country. fields ( [ 'Name ' , 'Population* ] ) .where ( 'Name LIKE "United%',,) -select ()-limit(5)

**B.** db . country, select ( [ ' Name LIKE "united%" ' , ' Population>^0 ' ] ) - limit (5)

**C.** db . country. fields ( [ ' Name ' , 'Population']) . select (' limit=5 ' ) .where('Name LIKE "United%" ' )

**D.** db.country.select(['Name', 'Population']).where('Name LIKE :param').bind('param', 'United%').limit(5)

**E.** db . country. Select ([Name' , 'Population.'] ) -limit (5) .where('Name LIKE "United%"')

**ANSWER: D**

**Explanation:**

`db.country.select(['Name', 'Population']).where('Name LIKE :param').bind('param', 'United%').limit(5)` correctly expresses the query with the MySQL X DevAPI table-selection fluent interface. Calling `select()` on the `country` table specifies the projected columns, so the result contains only `Name` and `Population`. The `where()` method supplies the filter expression, and `:param` is a named placeholder rather than a value concatenated into that expression. `bind()` then safely supplies the placeholder value. The value `United%` uses the SQL `LIKE` percent wildcard, matching names that begin with "United." Finally, `limit(5)` limits the returned result set to at most five documents or rows. This method chain follows the X DevAPI CRUD model for selecting rows from a relational table and parameterizing a selection criterion. Oracle's Connector/Node.js API documents the table-select methods, including `where()`, `bind()`, and

`limit()`, at TableSelect Class. The corresponding X DevAPI CRUD selection concepts are described in the MySQL X DevAPI User Guide.

## QUESTION NO: 6

The employee table includes these columns:

e_id INT, e_name VARCHAR (45), dept_id INT salart INT

You must create a stored function, getMaxSalary(), which returns the maximum salary paid for a given department id.

Which statement will create the function?

A)

```
CREATE FUNCTION getMaxSalary(v_dept_id INT)  RETURNS CHAR(50)
BEGIN
    SELECT MAX(salary) INTO @m FROM employee WHERE dept_id = v_dept_id;
    RETURN m;
END
```

B)

```
CREATE FUNCTION getMaxSalary(v_dept_id INT) RETURNS INT
DETERMINISTIC
BEGIN
    DECLARE msalary INT;
    USE schemaName;
    SELECT MAX(salary) INTO msalary FROM employee WHERE dept_id = v_dept_id;
    RETURN msalary;
END
```

C)

```
CREATE FUNCTION getMaxSalary(v_dept_id INT) RETURNS INT
DETERMINISTIC
BEGIN
    DECLARE msalary INT;
    SELECT MAX(salary) INTO msalary FROM employee WHERE dept_id = v_dept_id;
    RETURN msalary;
END
```

D)

```
CREATE FUNCTION getMaxSalary(v_dept_id INT) RETURNS INT
DECLARE msalary INT;
BEGIN
    SELECT MAX(salary) INTO msalary FROM employee WHERE dept_id = v_dept_id;
           getMaxSalary := msalary;
    RETURN;
END
```

**A.** Option A

**B.** Option B

**C.** Option C

**D.** Option D

**E.** CREATE FUNCTION getMaxSalary(p_dept_id INT) RETURNS INT DETERMINISTIC RETURN (SELECT MAX(salary) FROM employee WHERE dept_id = p_dept_id);

## ANSWER: E

**Explanation:**

`CREATE FUNCTION getMaxSalary(p_dept_id INT) RETURNS INT DETERMINISTIC RETURN (SELECT MAX(salary) FROM employee WHERE dept_id = p_dept_id);` correctly defines a MySQL stored function because it declares a function name, accepts the department identifier as an input parameter, and declares the function's integer return type. The `RETURN` expression is a scalar subquery: filtering rows to the supplied department and applying `MAX()` produces the single maximum salary value required by the function. Marking the function `DETERMINISTIC` documents that, for a given database state and input department identifier, the expression returns the same result. MySQL permits a stored function body to consist of a single `RETURN` statement, so no `BEGIN/END` block or local variable is required for this implementation. The salary column is written as `salary` because that is the field described by the requirement; if the actual table column is named `salart`, that identifier must be substituted exactly in the query. See the MySQL reference for CREATE FUNCTION syntax and for the behavior of MAX().

## QUESTION NO: 7

Examine these statements and output:

```
mysql> SET AUTOCOMMIT=on;
Query OK, 0 rows affected (0.01 sec)

mysql> UPDATE emp
    -> SET salary=24000
    -> WHERE id=101;
Query OK, 1 row affected (0.01 sec)

mysql> INSERT INTO EMP values (102,'John',13000,'jj',10);
Query OK, 1 row affected (0.00 sec)

mysql> SET AUTOCOMMIT=off;
Query OK, 0 rows affected (0.01 sec)
```

Now, examine this command:

Mysql > ROLLBACK;

What is true about the effect of the command?

**A.** It undoes the update command.

**B.** It returns an error because there is no active transaction.

**C.** It undoes the insert command.

**D.** It undoes both insert and update commands.

**E.** It has no effect.

## ANSWER: C

**Explanation:**

**It undoes the insert command.** A `ROLLBACK` ends the current transaction and discards its uncommitted changes. In the displayed sequence, the insert is the data-changing statement that remains part of the active transaction when `ROLLBACK` is issued. Consequently, MySQL reverses the row insertion and restores the table to the state it had before that insert was executed.

This behavior depends on transactional storage-engine support, such as that provided by InnoDB. MySQL transactions make changes durable only when they are committed; until then, changes are pending and can be rolled back. The rollback applies to the current transaction's uncommitted work, rather than to statements that have already been finalized by a prior transaction boundary. MySQL documents that `ROLLBACK` cancels the current transaction and that transaction control applies to transactional tables. See the MySQL 8.0 COMMIT, ROLLBACK, and autocommit documentation and the InnoDB transaction model documentation.

## QUESTION NO: 8

Which two are true about MySQL events?

**A.** An event can be scheduled to execute repeatedly by using an EVERY clause.

**B.** An event executes only while the client that created it remains connected.

**C.** The Event Scheduler must be enabled for scheduled events to execute.

**D.** Events can be created only in the mysql system schema.

**E.** An event can execute only SELECT statements.

## ANSWER: A C

**Explanation:**

An event can be scheduled to execute repeatedly. The `CREATE EVENT` statement supports an `EVERY` clause, allowing an event to run at a specified interval such as every hour, day, or week. A recurring event can also specify start and end times.

The Event Scheduler must be enabled for scheduled events to execute. The scheduler is controlled by the global `event_scheduler` system variable. Creating an event does not by itself ensure that it will run while the scheduler is disabled.

Events are database objects that execute SQL statements on a schedule and are not client-side jobs. The server stores event definitions in the data dictionary. See Event Scheduler Configuration and the CREATE EVENT Statement documentation.

## QUESTION NO: 9

Examine this statement which executes successfully:

SET @ir := 2;

Which query updates the value of @r to 0?

**A.** SELECT 'Car' REGEXP('Ca?') >= 0 INTO @r;

**B.** SELECT STRCMP('Car'/Ca?') >= 0 INTO @r;

**C.** SELECT 'Car' LIKE 'Ca?' INTO @r;

**D.** SELECT 'Car' RLIKE 'Ca?' INTO @r;

## ANSWER: C

**Explanation:**

`SELECT 'Car' LIKE 'Ca?' INTO @r;` assigns `0` to `@r`. In MySQL, `LIKE` compares a string with a pattern using only two built-in wildcard characters: `%`, which represents any sequence of zero or more characters, and `_`, which represents exactly one character. A question mark is not a `LIKE` wildcard. Therefore, the pattern `'Ca?'` means the literal three-character string `Ca?`. Since `'Car'` is not equal to that literal pattern, the `LIKE` expression evaluates to false, represented by the integer value `0`. The `INTO @r` clause stores that scalar expression result in the session user variable `@r`. User variables are session-specific and may be assigned through a `SELECT ... INTO` statement, as documented by MySQL. See the MySQL reference for LIKE pattern matching and user-defined variables.

## QUESTION NO: 10

Which two are true about generated columns?

**A.** A generated column can be indexed.

**B.** A generated-column expression can contain a subquery.

**C.** A generated column can be assigned an explicit value in an INSERT statement.

**D.** A generated-column expression cannot contain a subquery.

**E.** All generated columns are stored physically in the table rows.

**ANSWER: A D**

**Explanation:**

A generated column can be indexed. MySQL calculates the generated value from its defining expression and can maintain an index on that value. This permits efficient access for queries that use the generated column or, in applicable cases, an expression matching the indexed generated-column definition.

A generated-column expression cannot contain a subquery. The expression must be deterministic and can refer only to permitted constructs, including literals, operators, and built-in functions. It cannot refer to another table through a subquery, and it cannot use disallowed nondeterministic functions.

Generated columns can be declared `VIRTUAL` or `STORED`. A virtual value is evaluated when needed, whereas a stored value is materialized when a row is inserted or updated. MySQL documents generated-column restrictions and indexing support in the Generated Column Syntax reference.