Item: 1 (Ref:1Z0-061.8.7.2)

Evaluate the following SELECT statement:

```
SELECT DISTINCT emp_id
FROM emp e JOIN emp_hist h
ON e.emp_id = h.emp_id;
```

Which SELECT statement will return the same result as the given statement?

- ```
  SELECT emp_id
  FROM emp
  UNION
  SELECT emp_id
  FROM emp_hist;
  ```
- ```
  SELECT e.emp_id
  FROM emp e, emp_hist h;
  WHERE e.emp_id <> h.emp_id;
  ```
- ```
  SELECT emp_id
  FROM emp
  MINUS
  SELECT emp_id
  FROM emp_hist;
  ```
- ```
  SELECT emp_id
  FROM emp
  INTERSECT
  SELECT emp_id
  FROM emp_hist;
  ```

Answer:

```
SELECT emp_id
FROM emp
INTERSECT
SELECT emp_id
FROM emp_hist;
```

Explanation:

The following SELECT statement will return the same result as the statement given in this scenario:

```
SELECT emp_id
FROM emp
INTERSECT
SELECT emp_id
FROM emp_hist;
```

The given JOIN statement returns only the emp_id values that are common to the emp and emp_hist tables. The INTERSECT operator in the correct option also returns only the emp_id values that are common to both tables.

The following SELECT statement is incorrect because it uses the UNION operator and will return all the rows in both tables, eliminating any duplicates:

```
SELECT emp_id
FROM emp
UNION
SELECT emp_id
FROM emp_hist;
```

The statement does not display only the values that are common to both tables; it displays all values in both tables.

The following SELECT statement is incorrect because it returns the emp_id values that are unique for each table:

```
SELECT e.emp_id
FROM emp e, emp_hist h;
WHERE e.emp_id <> h.emp_id;
```

The statement does not display the values that are common to both tables; it displays the `emp_id` values that are unique in each table.

The following `SELECT` statement is incorrect because it returns the `emp_id` values that are in the `emp` table, but are not in the `emp_hist` table:

```
SELECT emp_id
FROM emp
MINUS
SELECT emp_id
FROM emp_hist;
```

The statement does not display the values that are common to both tables; it displays the `emp_id` values that are in the `emp` table and are not in the `emp_hist` table.

Item: 2 (Ref:1Z0-061.8.2.2)

Examine the structures of the `PLAYER` and `TEAM` tables:

```
PLAYER
-------------
PLAYER_ID NUMBER PK
LAST_NAME VARCHAR2(30)
FIRST_NAME VARCHAR2(25)
TEAM_ID NUMBER
MGR_ID NUMBER
SIGNING_BONUS NUMBER(9,2)

TEAM
---------
TEAM_ID NUMBER
TEAM_NAME VARCHAR2(30)
```

Which situation would require a subquery to return the desired result?

- ◯ a list of all players who are also managers
- ◯ a list of all teams that have more than 11 players
- ◯ a list of all players, including their signing bonus amounts and their manager names
- ◯ a list of all players who have a larger signing bonus than their manager
- ◯ a list of all players who received a signing bonus that was lower than the average bonus

Answer:

a list of all players who received a signing bonus that was lower than the average bonus

---

Explanation:

With the given tables, a subquery would be required to produce a list of all players who received a signing bonus that was lower than the average. To produce a list of players who received a signing bonus that was lower than the average, you could use:

```
SELECT last_name, first_name
FROM player
WHERE signing_bonus < (SELECT AVG(signing_bonus)
FROM player);
```

To produce a list of all players that are also managers, you could use a self join to join the `PLAYER` table to itself.

To produce a list of all teams with more than 11 players, you could query the `PLAYER` table grouping the records by `TEAM_ID` and using the `COUNT` function to count the distinct values of `PLAYER_ID`. Then, after grouping the data you could use a `HAVING` clause to return only those teams having more than 11 players.

To produce a list of all players, including their manager names and signing bonuses, you could use a self join to join the `PLAYER` table to itself. This would allow you to display each player's manager.

To produce a list of all players who have a larger signing bonus than their manager, you would join the `PLAYER` table to itself using a self join and then compare the signing bonuses for the player and the manager using a `WHERE` clause.

Subqueries are often used in a `WHERE` clause of a SQL statement to return values for an unknown conditional value. The inner query executes first and returns the results to the outer query for use in the outer query's `WHERE` clause.

Item: 3 (Ref:1Z0-061.8.4.3)

Evaluate this `SELECT` statement:

```
SELECT first_name, last_name
FROM physician
WHERE physician_id NOT IN (SELECT physician_id
FROM physician
WHERE license_no = 17852);
```

Which one of the following `SELECT` statements would achieve the same result?

- ```
  SELECT first_name, last_name
  FROM physician
  WHERE physician_id = 17852;
  ```
- ```
  SELECT first_name, last_name
  FROM physician_id
  WHERE license_no <> 17852
  AND license_no IS NOT NULL;
  ```
- ```
  SELECT first_name, last_name
  FROM physician
  WHERE physician_id IN (SELECT physician_id
  FROM physician
  WHERE license_no = 17852);
  ```
- ```
  SELECT first_name, last_name
  FROM physician
  WHERE physician_id != ALL (SELECT physician_id
  FROM physician
  WHERE license_no = 17852);
  ```

Answer:

```
SELECT first_name, last_name
FROM physician
WHERE physician_id != ALL (SELECT physician_id
FROM physician
WHERE license_no = 17852);
```

Explanation:

The following `SELECT` statement will return the same result as the given `SELECT` statement:

```
SELECT first_name, last_name
FROM physician
WHERE physician_id != ALL (SELECT physician_id
FROM physician
WHERE license_no = 17852);
```

In the scenario, the given `SELECT` statement uses the `NOT IN` operator to display all physicians who do not have a license number of 17852. The same results can be achieved using the `!=` and `ALL` operators. If null values are likely to be returned by the inner query, you should not use either of these operators. If one of the values returned by the inner query is null, the entire query will not return any rows, because all the conditions that compare a `NULL` value yield a null result.

The `SELECT` statement that includes `WHERE physician_id = 17852` as the query condition is incorrect because it will return only the physician that has an identifier of 17852, and this is not what you desired.

The `SELECT` statement that includes the `PHYSICIAN_ID` table in the `FROM` clause is incorrect. `PHYSICIAN_ID` is a column in the `PHYSICIAN` table, not a table itself.

The `SELECT` statement that uses a subquery with the `IN` operator returns the opposite of what you needed. The inner query returns the identifier for the physician with a license number of 17852, and then the outer query returns this physician's name. You needed all the physicians who did not have a license number of 17852. Therefore, this option is incorrect.

Item: 4 (Ref:1Z0-061.8.7.4)

Evaluate the `SELECT` statements in the following SQL compound queries:

```
SELECT emp_id
FROM emp
INTERSECT
SELECT emp_id
FROM emp_hist;

SELECT emp_id
FROM emp_hist
INTERSECT
SELECT emp_id
FROM emp;
```

Which statement is TRUE regarding these SQL compound queries?

- ◯ The results of the compound queries will be identical.
- ◯ The first compound query will return more results than the second.
- ◯ The second compound query will return more results than the first.
- ◯ The second compound query will return a syntax error.

Answer:

The results of the compound queries will be identical.

Explanation:

The results of the two queries will be identical because reversing the order of the `SELECT` statements with an `INTERSECT` operator will not alter the result set.

The first compound query will return the same number of results as the second compound query.

Reversing the order of the `SELECT` statements in a compound query using the `INTERSECT` operator will not affect the result set.

The second compound query does not contain a syntax error. The query uses the appropriate `SELECT` statement syntax, which is as follows:

```
{ query_block
| subquery { UNION [ALL] | INTERSECT | MINUS } subquery
[ { UNION [ALL] | INTERSECT | MINUS } subquery ]...
| ( subquery )
} [ order_by_clause ]
```

Item: 5 (Ref:1Z0-061.8.4.4)

The `employee` table contains these columns:

```
EMPLOYEE_ID NUMBER NOT NULL
EMP_LNAME VARCHAR2(20) NOT NULL
EMP_FNAME VARCHAR2(10) NOT NULL
DEPT_ID NUMBER
SALARY NUMBER(9,2)
```

A user needs to retrieve information on employees who have the same department ID and salary as an employee ID that the user will enter. You want the query results to include employees who do not have a salary, but not the employee that the user entered.

Which statement will return the desired result?

○
```
SELECT *
FROM employee
WHERE (department, salary) NOT IN

(SELECT department, salary)
FROM employee
WHERE employee_id = &1);
```

○
```
SELECT *
FROM employee
WHERE (dept_id, salary) IN
(SELECT dept_id, NVL(salary, 0)
FROM employee
WHERE employee_id = &1);
```

○
```
SELECT *
FROM employee
WHERE (dept_id, NVL(salary, 0)) IN

(SELECT dept_id, NVL(salary, 0)
FROM employee
WHERE employee_id = &&1)
AND employee_id <> &&1;
```

○
```
SELECT *
FROM employee
WHERE (dept_id, salary) IN
(SELECT dept_id, salary)
FROM employee
WHERE employee_id = &1
AND salary IS NULL);
```

Answer:

```
SELECT *
FROM employee
WHERE (dept_id, NVL(salary, 0)) IN

(SELECT dept_id, NVL(salary, 0)
FROM employee
WHERE employee_id = &&1)
AND employee_id <> &&1;
```

Explanation:

The following query will retrieve the desired result:

```
SELECT *
FROM employee
WHERE (dept_id, NVL(salary, 0)) IN (SELECT dept_id, NVL(salary, 0)
FROM employee
WHERE employee_id = &&1)
AND employee_id <> &&1;
```

When this statement executes, the inner query is processed first. The inner query returns the `dept_id` and `salary` values for the `employee_id` entered. These values are passed to the outer query, which produces a list of employees having the same department and salary. If a `NULL` value is returned in a subquery, the entire query will return no rows. To ensure that the subquery does not return a `NULL` value for the `salary` column, the `NVL` function is used. Because a value of zero is returned from the subquery if the `salary` value is `NULL`, the query result will include employees that do not have a salary. In addition, the `employee_id <> &&1` condition in the outer query `WHERE` clause will exclude the employee entered from the list.

The `SELECT` statement that references the `department` column is invalid because this is not a valid column name in the `employee` table.

The `SELECT` statement that is similar to the correct statement but includes only one condition in the outer query `WHERE` clause is incorrect. Although this query will return the correct list of employees, it will also return the employee that was entered, and this is not what you desired.

The `SELECT` statement that uses `salary IS NULL` in the inner query `WHERE` clause is incorrect. This inner query will return a `NULL` value, and when a subquery returns a `NULL` value, the entire result is null. Therefore, this query will return no rows.

Item: 6 (Ref:1Z0-061.8.2.1)

Which construct can be used to return data based on an unknown condition?

- ◯ a subquery
- ◯ a `GROUP BY` clause
- ◯ an `ORDER BY` clause
- ◯ a `WHERE` clause with an `OR` condition

Answer:

a subquery

Explanation:

A subquery can be used to return data based on an unknown condition. Often when the condition for a query cannot be stated directly, the query can be broken into two smaller queries to return the desired result. The subquery, or inner query, returns a value that is used by the main, or outer, query.

A `GROUP BY` clause creates groups of data so that aggregate calculations, such as sums and averages, can be performed on the group. An `ORDER BY` clause sorts the results of a query based on a specified sort order. A `WHERE` clause, whether it includes a logical conditional operator or not, defines a condition that must be met for rows to be returned. None of these constructs will allow you to return data based on an unknown condition.

Item: 7 (Ref:1Z0-061.8.4.2)

Evaluate this `SELECT` statement:

```
SELECT s.student_name, s.grade_point_avg, s.major_id, m.gpa_avg
FROM student s, (SELECT major_id, AVG(grade_point_avg) gpa_avg
FROM student
GROUP BY major_id) m
WHERE s.major_id = m.major_id AND s.grade_point_avg > m.gpa_avg;
```

What will be the result of executing this `SELECT` statement?

- ◯ The names of all students with a grade point average that is higher than the average grade point average in their major will be displayed.
- ◯ The names of all students with a grade point average that is higher than the average grade point average of all students will be displayed.
- ◯ The names of all students, grouped by each major, with a grade point average that is higher than the average grade point average of all students in each major will be displayed.
- ◯ A syntax error will occur because of ambiguous table aliases.
- ◯ A syntax error will be returned because the `FROM` clause cannot contain a subquery.

Answer:

The names of all students with a grade point average that is higher than the average grade point average in their major will be displayed.

Explanation:

The names of all students with a grade point average that is higher than the average grade point average in their major will be displayed. You can use a subquery in the `FROM` clause of a `SELECT` statement to define a data source for the `SELECT` statement. This is helpful if you need to view aggregate values but need to include columns in the select list that are not grouped. In this scenario, the subquery returns the `major_id` values and the average grade point average of students with each major. This result is then used as a data source for the main query. The `WHERE` clause of the main query joins this result table to the `STUDENT` table using `major_id` and ensures that you only return rows where the grade point average is higher than the average grade point average of the student's major.

The option stating that the names of all students with a grade point average that is higher than the average grade point average of all students will be displayed is incorrect. Because the inner query groups the records by major and the outer query `WHERE` clause joins on major, each student's grade point average will be compared to the average grade point average for the same major.

The option stating that the names of all students grouped by major are displayed is incorrect. The outer query does not contain a `GROUP BY` clause, so records returned are not grouped.

The option stating that an error will occur because of ambiguous table aliases is incorrect. Although the table alias m is used twice, this is acceptable. The main query uses the table alias for the entire subquery.

The option stating that an error will occur because the `FROM` clause cannot contain a subquery is incorrect because a table, view, or subquery is allowed in the `FROM` clause of a `SELECT` statement.

Item: 8 (Ref:1Z0-061.8.8.2)

Evaluate this `SELECT` statement:

```
SELECT product_id, category_id
FROM product
ORDER BY 2
INTERSECT
SELECT product_id, category_id
FROM product_history;
```

Which of the following results is TRUE regarding this `SELECT` statement?

- ◯ It will return the results sorted ascending by the `category_id` values returned by the first query.
- ◯ It will return the results sorted ascending by the `category_id` values returned by both queries.
- ◯ It will return the results sorted ascending by both the `product_id` and `category_id` values returned by the first query.
- ◯ It will return the results sorted ascending by both the `product_id` and `category_id` values returned by both queries.
- ◯ It will return an error.

Answer:

It will return an error.

Explanation:

This statement will return an error. Only one `ORDER BY` clause is allowed in a compound query, and it must be placed at the end of the compound query. The correct statement would be as follows:

```
SELECT product_id, category_id
FROM product
INTERSECT
SELECT product_id, category_id
FROM product_history
ORDER BY 2;
```

The `ORDER BY` clause in a compound query will only recognize and sort the columns in the first `SELECT` list. If this statement was corrected, the clause `ORDER BY 2` will sort the results based on the `category_id` values retrieved by the first query in the compound query.

Item: 9 (Ref:1Z0-061.8.2.3)

Examine the structures of the CUSTOMER and CURR_ORDER tables:

```
CUSTOMER
--------------------
CUSTOMER_ID  NUMBER(5)
NAME         VARCHAR2(25)
CREDIT_LIMIT   NUMBER(8,2)
ACCT_OPEN_DATE DATE

CURR_ORDER
-------------------------
ORDER_ID    NUMBER(5)
CUSTOMER_ID NUMBER(5)
ORDER_DATE  DATE
TOTAL       NUMBER(8,2)
```

Which scenario would require a subquery to return the desired results?

- ◯ You need to display the names of all the customers who placed an order today.
- ◯ You need to determine the number of orders placed this year by the customer with CUSTOMER_ID value 30450.
- ◯ You need to determine the average credit limit of all the customers who opened an account this year.
- ◯ You need to determine which customers have placed orders with amount totals larger than the average order amount.

Answer:

You need to determine which customers have placed orders with amount totals larger than the average order amount.

Explanation:

With the given tables, a subquery would be required to determine which customers have placed orders with amount totals larger than the average order amount. To return the desired result, an inner query would return the average order amount. Then, the outer query would use this value in the WHERE clause to restrict the rows returned to only those customers who had placed orders with total amounts larger than the average order amount.

You could display the names of all the customers who placed an order today by using a join operator or a subquery, but a subquery is not required. This could be accomplished simply by joining the CUSTOMER and CURR_ORDER tables using the CUSTOMER_ID column and including WHERE order_date = sysdate to restrict the rows to only those customers placing orders today.

To determine the number of orders placed this year by the customer with CUSTOMER_ID value 30450, you could query the CURR_ORDER table restricting the rows using a WHERE clause and include the COUNT aggregate function in the select list.

To determine the average credit limit of all the customers who opened an account this year, you could join the CUSTOMER and CURR_ORDER tables and restrict the rows using a WHERE clause. Then, use the AVG aggregate function in the select list to calculate the average credit limit for each group of rows.

Subqueries are often used in a WHERE clause of a SQL statement to return values for an unknown conditional value. The inner query executes first and returns the results to the outer query for use in the outer query's WHERE clause.

Item: 10 (Ref:1Z0-061.8.6.1)

Which set operator would you use to display the employee IDs of employees hired after January 10, 2007 in the `employee` table and employee IDs of employees who have held more than one position in the `emp_hist` table, eliminating any duplicate IDs?

- ◯ `UNION`
- ◯ `UNION ALL`
- ◯ `INTERSECT`
- ◯ `MINUS`

Answer:

`UNION`

Explanation:

You should use the `UNION` operator to display the employee IDs of employees hired after January 10, 2007 in the `employee` table and employee IDs of employees who have held more than one position in the `emp_hist` table, eliminating any duplicate IDs. The following SQL statement will achieve this result set:

```
SELECT emp_id
FROM employee
WHERE hire_date > TO_DATE('10-JAN-2007')
UNION
SELECT emp_id
FROM emp_hist;
```

Set operators allow the results of two or more queries to be combined into a single result set. SQL statements that include set operators are known as compound queries. The set operators are:

- `UNION` - returns the result sets from all the queries after eliminating any duplicate records
- `UNION ALL` - returns the result sets from all the queries in a statement including the duplicate records
- `INTERSECT` - returns only the common result sets that are retrieved by all the queries
- `MINUS` - returns only the results that are returned by the first query and not by the second query.

Set operators of equal precedence are evaluated from left to right unless parentheses force the order of evaluation. When using a set operator, the columns in the `SELECT` list in each query must be the same in number and data type.

Item: 11 (Ref:1Z0-061.8.4.6)

You need to create a report to display the names of customers with a credit limit greater than the average credit limit of all customers.

Which `SELECT` statement should you use?

- ```
  SELECT last_name, first_name
  FROM customer
  WHERE credit_limit > AVG(credit_limit);
  ```
- ```
  SELECT last_name, first_name, AVG(credit_limit)
  FROM customer
  GROUP BY AVG(credit_limit);
  ```
- ```
  SELECT last_name, first_name, AVG(credit_limit)
  FROM customer
  GROUP BY AVG(credit_limit)
  HAVING credit_limit > AVG(credit_limit);
  ```
- ```
  SELECT last_name, first_name
  FROM customer
  WHERE credit_limit > (SELECT AVG(credit_limit)

  FROM customer);
  ```
- ```
  SELECT last_name, first_name
  FROM customer
  WHERE credit_limit = (SELECT AVG(credit_limit)

  FROM customer);
  ```

Answer:

```
SELECT last_name, first_name
FROM customer
WHERE credit_limit > (SELECT AVG(credit_limit)

FROM customer);
```

Explanation:

You should use the following `SELECT` statement:

```
SELECT last_name, first_name
FROM customer
WHERE credit_limit > (SELECT AVG(credit_limit)
FROM customer);
```

To return the names of all customers with a credit limit greater than the average credit limit of all customers, you must use the statement that uses a subquery and compares the credit limit to the subquery values using the greater than operator (>). In this scenario, the inner query returns the average credit limit of all customers. The outer query takes this average credit limit value and uses this value to display all the customers who have a credit limit greater than this amount.

The statement that includes `WHERE credit_limit > AVG(credit_limit)` for the query condition is incorrect. Aggregate, or group, functions cannot be used in a `WHERE` clause.

Neither of the statements that group the result by `AVG(credit_limit)` is correct because group functions are not allowed in a `GROUP BY` clause.

The statement that includes a subquery and compares the credit limit to the subquery values using the equality operator (=) will return only those customers who have a credit limit equal to the average credit limit of all customers, and this is not what you desired.
```

Item: 12 (Ref:1Z0-061.8.8.3)

Evaluate this `SELECT` statement:

```
SELECT emp_id "Employee", dept_id "Department"
FROM emp
INTERSECT
SELECT emp_id employee, dept_id department
FROM emp
WHERE dept_id >100
MINUS
SELECT emp_id "Employee", dept_id "Department"
FROM emp
WHERE dept_id <> 200
ORDER BY 2;
```

Which of the following statements is true?

- ◯ The statement will return the results sorted by the `dept_id` values in the first query.
- ◯ The statement will return the results sorted by the `dept_id` values in the second query.
- ◯ The statement will return the results sorted by the `dept_id` values in the third query.
- ◯ The statement will return an error.

Answer:

The statement will return the results sorted by the `dept_id` values in the first query.

Explanation:

The statement will return the results sorted by the `dept_id` values in the first query. Only one `ORDER BY` clause is allowed in a compound query, and it must be placed at the end of the compound query. The `ORDER BY` clause only recognizes and sorts the columns in the first `SELECT` list in a compound query.

For this statement, the clause `ORDER BY 2` will sort the results based on the `dept_id` values retrieved by the first query in the compound query.

Item: 13 (Ref:1Z0-061.8.1.1)

Which two statements regarding the valid use of single-row and multiple-row subqueries are true? (Choose two.)

- [ ] Single-row subqueries can only be used in a `WHERE` clause.
- [ ] Multiple-row subqueries can be used with the `LIKE` operator.
- [ ] Single-row operators can only be used with single-row subqueries.
- [ ] Single- and multiple-row subqueries can be used with the `BETWEEN` operator.
- [ ] Multiple-row subqueries can be used with both single-row and multiple-row operators.
- [ ] Multiple-row subqueries can be used in a `WHERE` clause and the `INTO` portion of an `INSERT` statement.

Answer:

Single-row operators can only be used with single-row subqueries.

Multiple-row subqueries can be used in a `WHERE` clause and the `INTO` portion of an `INSERT` statement.

Explanation:

The following two statements regarding the valid use of single-row and multiple-row subqueries are true:

- Single-row operators can only be used with single-row subqueries.
- Multiple-row subqueries can be used in a `WHERE` clause and the `INTO` portion of an `INSERT` statement.

A single-row subquery is a subquery that returns only one row from the inner `SELECT` statement. Single-row subqueries can only be used with single-row operators, such as =, > , >=, <, <=, or <>. When a single-row operator is used, then the subquery must be a single-row subquery that returns only one row. If you attempt to use a single-row operator with a subquery that returns multiple rows, an error occurs. Multiple-row subqueries can be used in a `WHERE` clause and the `INTO` portion of an `INSERT` statement. When used in a `WHERE` clause, the multiple-row subquery must use a multiple-row operator, such as `IN`, `ANY`, or `ALL`. When used in the `INTO` portion of an `INSERT` statement, all rows returned by the multiple-row subquery are inserted into the specified table.

The option stating that single-row subqueries can only be used in a `WHERE` clause is incorrect. Single-row subqueries can be used any place that a single scalar value can be used.

The option stating that multiple-row subqueries can be used with the `LIKE` operator is incorrect. The `LIKE` operator accepts a single value and can only be used with single-row queries.

The option stating that single- and multiple-row subqueries can be used with the `BETWEEN` operator is incorrect. The `BETWEEN` operator accepts two values. Either one or both of these values may come from single-row subqueries. However, the values may not come from multiple-row subqueries.

The option stating that multiple-row subqueries can be used with both single-row and multiple-row operators is incorrect because they can only use multiple-row operators.

Item: 14 (Ref:1Z0-061.8.4.1)

Examine the data from the DONATION table.

DONATION (PLEDGE_ID is the primary key)

PLEDGE_ID	DONOR_ID	PLEDGE_DT	AMOUNT_PLEDGED	AMOUNT_PAID	PAYMENT_DATE
1	1	10-SEP-2011	1000	1000	02-OCT-2001
2	1	22-FEB-2011	1000		
3	2	08-OCT-2001	10	10	28-OCT-2001
4	2	10-DEC-2001	50		
5	3	02-NOV-2001	10000	9000	28-DEC-2001
6	3	05-JAN-2002	1000	1000	31-JAN-2002
7	4	09-NOV-2001	2100	2100	15-DEC-2001
8	5	09-DEC-2001	110	110	29-DEC-2001

This statement fails when executed:

```
SELECT amount_pledged, amount_paid
FROM donation
WHERE donor_id =
(SELECT donor_id
FROM donation
WHERE amount_pledged = 1000.00
OR pledge_dt = '05-JAN-2002');
```

Which two changes could correct the problem? (Choose two. Each correct answer is a separate solution.)

- [ ] Remove the subquery WHERE clause.
- [ ] Change the outer query WHERE clause to WHERE donor_id IN.
- [ ] Change the outer query WHERE clause to WHERE donor_id LIKE.
- [ ] Include the donor_id column in the select list of the outer query.
- [ ] Remove the single quotes around the date value in the inner query WHERE clause.
- [ ] Change the subquery WHERE clause to WHERE amount_pledged = 1000.00 AND pledge_dt = '05-JAN-2002'.

Answer:

Change the outer query WHERE clause to WHERE donor_id IN.

Change the subquery WHERE clause to WHERE amount_pledged = 1000.00 AND pledge_dt = '05-JAN-2002'.

Explanation:

This statement fails because the subquery returns multiple rows, which cannot be compared to a single value using the equality operator (=) in the outer query. To correct the problem, you could change the outer query WHERE clause to WHERE donor_id IN. Changing the outer query WHERE clause to use the IN operator would allow the inner query to return multiple rows without generating an error.

Alternatively, you could change the subquery WHERE clause to WHERE amount_pledged = 1000.00 AND pledge_dt = '05-JAN-2002'. Based on the given data, this change would cause only one row to be returned and would also eliminate the error.

Removing the subquery WHERE clause, changing the outer query WHERE clause to WHERE donor_id LIKE, or including the donor_id column in the select list of the outer query would not correct the problem. The inner query would still return multiple rows and produce an error.

Removing the single quotes around the date value in the inner query WHERE clause will not correct the problem. When dates values are used in a WHERE clause, they must be enclosed in single quotation marks.

Item: 15 (Ref:1Z0-061.8.4.5)

Examine the structure of the `employee` table.

EMPLOYEE

EMPLOYEE_ID	NUMBER	NOT NULL, Primary Key
EMP_LNAME	VARCHAR2(25)	
EMP_FNAME	VARCHAR2(25)	
DEPT_ID	NUMBER	Foreign key to DEPT_ID column of DEPARTMENT table
JOB_ID	NUMBER	Foreign key to JOB_ID column of JOB table
MGR_ID	NUMBER	References EMPLOYEE_ID column
SALARY	NUMBER(9,2)	
HIRE_DATE	DATE	
DOB	DATE	

You want to generate a list of employees are in department 30, have been promoted from clerk to associate by querying the `employee` and `employee_hist` tables. The `employee_hist` table has the same structure as the `employee` table. The `job_id` value for clerks is 1 and the `job_id` value for associates is 6.

Which query should you use?

○
```
SELECT employee_id, emp_lname, emp_fname, dept_id
FROM employee
WHERE (employee_id, dept_id) IN
(SELECT employee_id, dept_id
FROM employee_hist
WHERE dept_id = 30 AND job_id = 1)
AND job_id = 6;
```

○
```
SELECT employee_id, emp_lname, emp_fname, dept_id
FROM employee
WHERE (employee_id) IN
(SELECT employee_id
FROM employee_hist
WHERE dept_id = 30 AND job_id = 1);
```

○
```
SELECT employee_id, emp_lname, emp_fname, dept_id
FROM employee
WHERE (employee_id, dept_id) =
(SELECT employee_id, dept_id
FROM employee_hist
WHERE dept_id = 30 AND job_id = 6);
```

○
```
SELECT employee_id, emp_lname, emp_fname, dept_id
FROM employee
WHERE (employee_id, dept_id) IN
(SELECT employee_id, dept_id
FROM employee
WHERE dept_id = 30)
AND job_id = 6;
```

○
```
SELECT employee_id, emp_lname, emp_fname, dept_id
FROM employee_hist
WHERE (employee_id, dept_id) =
(SELECT employee_id, dept_id
FROM employee_hist WHERE dept_id = 30
AND job_id = 1)
AND job_id = 6;
```

Answer:

```
SELECT employee_id, emp_lname, emp_fname, dept_id
FROM employee
WHERE (employee_id, dept_id) IN
(SELECT employee_id, dept_id
```

```
FROM employee_hist
WHERE dept_id = 30 AND job_id = 1)
AND job_id = 6;
```

Explanation:

You should use the following query:

```
SELECT employee_id, emp_lname, emp_fname, dept_id
FROM employee
WHERE (employee_id, dept_id) IN
(SELECT employee_id, dept_id
FROM employee_hist
WHERE dept_id = 30 AND job_id = 1)
AND job_id = 6;
```

A multi-column subquery is used to retrieve the employee IDs and department IDs of employees who are clerks (`job_id = 1`) and who work in department 30 from the `employee_hist` table. The `IN` operator is used to compare the list of employee IDs retrieved from the subquery with the employee IDs in the `employee` table (the outer query). This retrieved list of employees is further qualified with the use of the `AND` operator, eliminating any employees from the list that are not currently associates (`job_id = 6`). The result is a list of employees in department 30 who were promoted from clerk to associate.

The `SELECT` statement that returns a single column in the subquery is incorrect because this statement will return a list of all employees who were previously clerks in department 30.

The `SELECT` statement that includes `WHERE dept_id = 30 AND job_id = 6` as the condition for the inner query is incorrect because this statement returns all employees who at any time have been associates in department 30.

Both of the statements that use the same table in the inner and outer query are incorrect. To produce a report of promotions, both tables must be included in the query. The `employee` table must be queried to check for each employee's current job, and the `employee_hist` table must be queried to determine each employee's previous position.

Item: 16 (Ref:1Z0-061.8.8.1)

Which two `SELECT` statements have valid `ORDER BY` clauses? (Choose two. Each correct answer is a separate solution.)

- [ ]
```
SELECT product_id
FROM current_product
MINUS
SELECT product_id
FROM product_develop
WHERE develop_cost > 5.60
ORDER BY develop_cost;
```

- [ ]
```
SELECT product_id, category_id
FROM current_product
MINUS
SELECT product_id, category_id
FROM product_develop
WHERE resource_code = 10
ORDER BY 1,2;
```

- [ ]
```
SELECT product_id Product, category_id Category
FROM current_product
MINUS
SELECT product_id, category_id
FROM product_develop
WHERE resource_code = 10
ORDER BY Category, Product;
```

- [ ]
```
SELECT product_id, category_id
FROM current_product
ORDER BY 1,2
MINUS
SELECT product_id, category_id
FROM product_develop
WHERE resource_code = 10;
```

- [ ]
```
SELECT product_id Product, category_id Category
FROM current_product
ORDER BY Category, Product
MINUS
SELECT product_id, category_id
FROM product_develop
WHERE resource_code = 10;
```

Answer:

```
SELECT product_id, category_id
FROM current_product
MINUS
SELECT product_id, category_id
FROM product_develop
WHERE resource_code = 10
ORDER BY 1,2;
```

```
SELECT product_id Product, category_id Category
FROM current_product
MINUS
SELECT product_id, category_id
FROM product_develop
WHERE resource_code = 10
ORDER BY Category, Product;
```

Explanation:

This `SELECT` statement has a valid `ORDER BY` clause:

```
SELECT product_id, category_id
FROM current_product
MINUS
SELECT product_id, category_id
```

```
FROM product_develop
WHERE resource_code = 10
ORDER BY 1,2;
```

The results of the compound query will be sorted by the `product_id` values and then by the `category_id` values returned by the first query in the compound statement. The `ORDER BY` clause uses the column positions in the first `SELECT` list.

This `SELECT` statement also has a valid `ORDER BY` clause:

```
SELECT product_id Product, category_id Category
FROM current_product
MINUS
SELECT product_id, category_id
FROM product_develop
WHERE resource_code = 10
ORDER BY Category, Product;
```

The results of the compound query will be sorted by the `category_id` values and then by the `product_id` values returned by the first query in the compound statement. The `ORDER BY` clause uses the column aliases defined in the `SELECT` list.

The `ORDER BY` clause in the following `SELECT` statement is incorrect because you can use columns in only the first `SELECT` list in a compound query:

```
SELECT product_id
FROM current_product
MINUS
SELECT product_id
FROM product_develop
WHERE develop_cost > 5.60
ORDER BY develop_cost;
```

You cannot sort the results by the `develop_cost` values because this column is not included in the first `SELECT` list in this compound query.

The `ORDER BY` clause in the following `SELECT` statement is incorrect because the `ORDER BY` clause must be placed at the end of a compound query:

```
SELECT product_id, category_id
FROM current_product
ORDER BY 1,2
MINUS
SELECT product_id, category_id
FROM product_develop
WHERE resource_code = 10;
```

The `ORDER BY` clause uses the column positions in the first `SELECT` list. Moving this `ORDER BY` clause to the end of the compound statement would allow this statement to execute.

The `ORDER BY` clause in the following `SELECT` statement is incorrect because the `ORDER BY` clause must be placed at the end of a compound query:

```
SELECT product_id Product, category_id Category
FROM current_product
ORDER BY Category, Product
MINUS
SELECT product_id, category_id
FROM product_develop
WHERE resource_code = 10;
```

The `ORDER BY` clause uses the column aliases defined in the first `SELECT` list. Moving this `ORDER BY` clause to the end of the compound statement would allow this statement to execute.

Item: 17 (Ref:1Z0-061.8.1.2)

How many values could a subquery used with the <> operator return?

- ◯ only one
- ◯ up to two
- ◯ up to ten
- ◯ unlimited

Answer:

only one

Explanation:

The not equal (<>) operator is a single-row operator. Single-row operators can only be used with single-row subqueries, or inner queries, that return only one row. Attempting to use the not equal (<>) operator with a query that returns more than one row will generate an error.

All other options that indicate more than one value could be returned are incorrect. If a subquery returns more than one row, then it can only be used with a multiple-row operator. Multiple-row operators include the `IN`, `ANY`, and `ALL` operators.

Item: 18 (Ref:1Z0-061.8.7.1)

Click the **Exhibit(s)** button to examine the structures of the `CURRENT_PRODUCTS` , `LINE_ITEM` , and `PRODUCT_DEVELOP` tables.

You need to display the product IDs for current products that were released before January 1, 2008 and were sold after July 10, 2008.

Which two `SELECT` statements will return the desired results? (Choose two. Each correct answer is a separate solution.)

☐
```
SELECT product_id
FROM current_products
INTERSECT
SELECT product_id
FROM product_develop
WHERE release_date < TO_DATE('01-JAN-2008')
MINUS
SELECT product_id
FROM line_item
WHERE sale_date <= TO_DATE('10-JUL-2008');
```

☐
```
SELECT product_id
FROM current_products
INTERSECT
SELECT product_id
FROM product_develop
WHERE release_date < TO_DATE('01-JAN-2008')
INTERSECT
SELECT product_id
FROM line_item
WHERE sale_date > TO_DATE('10-JUL-2008');
```

☐
```
SELECT product_id
FROM current_products
UNION
SELECT product_id
FROM product_develop
WHERE release_date < TO_DATE('01-JAN-2008')
UNION
SELECT product_id
FROM line_item
WHERE sale_date >= TO_DATE('10-JUL-2008');
```

☐
```
SELECT product_id
FROM current_products
UNION
SELECT product_id
FROM product_develop
WHERE release_date < TO_DATE('01-JAN-2008')
MINUS
SELECT product_id
FROM line_item
WHERE sale_date >= TO_DATE('10-JUL-2008');
```

☐
```
SELECT product_id
FROM current_products
UNION
SELECT product_id
FROM product_develop
WHERE release_date < TO_DATE('01-JAN-2008')
UNION
SELECT product_id
FROM line_item
WHERE sale_date <= TO_DATE('10-JUL-2008');
```

Answer:

```
SELECT product_id
FROM current_products
INTERSECT
SELECT product_id
FROM product_develop
```

```
WHERE release_date < TO_DATE('01-JAN-2008')
MINUS
SELECT product_id
FROM line_item
WHERE sale_date <= TO_DATE('10-JUL-2008');
```

```
SELECT product_id
FROM current_products
INTERSECT
SELECT product_id
FROM product_develop
WHERE release_date < TO_DATE('01-JAN-2008')
INTERSECT
SELECT product_id
FROM line_item
WHERE sale_date > TO_DATE('10-JUL-2008');
```

CURRENT_PRODUCTS

PRODUCT_ID	NUMBER(9)	Primary Key
CATEGORY_ID	NUMBER(9)	Foreign Key (CATEGORY)
DESCRIPTION	VARCHAR2(50)	
COST	NUMBER(9,2)	
PRICE	NUMBER(9,2)	

LINE_ITEM

LINE_ITEM_ID	NUMBER(9)	Primary Key
ORDER_ID	NUMBER(9)	Foreign Key (ORDER)
PRODUCT_ID	NUMBER(9)	Foreign Key (CURRENT_PRODUCT)
CUSTOMER_ID	NUMBER(9)	Foreign Key (CATEGORY)
PRICE	NUMBER(9,2)	
SALE_DATE	DATE	

PRODUCT_DEVELOP

PRODUCT_ID	NUMBER(9)	Primary Key
CATEGORY_ID	NUMBER(9)	Foreign Key (CATEGORY)
DESCRIPTION	VARCHAR2(50)	
COST	NUMBER(9,2)	
PRICE	NUMBER(9,2)	
START_PRODUCTION_DATE	DATE	
RELEASE_DATE	DATE	

Explanation:

The following `SELECT` statement will display the `product_id` values for current products that were released before January 1, 2008 and sold after July 10th, 2008:

```
SELECT product_id
FROM current_products
INTERSECT
SELECT product_id
FROM product_develop
WHERE release_date < TO_DATE('01-JAN-2008')
MINUS
SELECT product_id
FROM line_item
WHERE sale_date <= TO_DATE('10-JUL-2008');
```

The first query returns all the `product_id` values from the `CURRENT_PRODUCTS` table. The second query returns only the `product_id` values from the `PRODUCT_DEVELOP` table that released before January 10, 2008. The `INTERSECT` operator returns only the `product_id` values that are common in the first and second queries. The final query pulls only the `product_id` values from the `line_item` table with a `sale_date` value that is less than or equal to July 10, 2008. The `MINUS` operator returns only the

product_id values that are in the INTERSECT result set but are not in the third query result set. The final result set of this compound query is the product_id values that are current products released before January 1, 2008 and sold after July 10, 2008.

The following SELECT statement will also display the product_id values for current products that were released before January 1, 2008 and sold after July 10, 2008:

```
SELECT product_id
FROM current_products
INTERSECT
SELECT product_id
FROM product_develop
WHERE release_date < TO_DATE('01-JAN-2008')
INTERSECT
SELECT product_id
FROM line_item
WHERE sale_date > TO_DATE('10-JUL-2008');
```

The first query returns all the product_id values from the CURRENT_PRODUCTS table. The second query returns only the product_id values from the PRODUCT_DEVELOP table that released before January 10, 2008. The INTERSECT operator returns only the product_id values that are common in the first and second queries. The final query pulls only the product_id values from the line_item table with a sale_date value that is greater than July 10, 2008. The second INTERSECT operator returns only the product_id values that are in both the first INTERSECT result set and in the third query result set. The final result set of this compound query is the product_id values that are current products released before January 1, 2008 and sold after July 10, 2008.

This SELECT statement does not return the desired result set:

```
SELECT product_id
FROM current_products
UNION
SELECT product_id
FROM product_develop
WHERE release_date < TO_DATE('01-JAN-2008')
UNION
SELECT product_id
FROM line_item
WHERE sale_date >= TO_DATE('10-JUL-2008');
```

The first query returns all the product_id values from the CURRENT_PRODUCTS table. The second query returns only the product_id values from the PRODUCT_DEVELOP table that released before January 10, 2008. The first UNION operator returns all the product_id values from both of the first two queries, eliminating any duplicates. The final query pulls only the product_id values from the line_item table with a sale_date value that is on or after July 10, 2008. The second UNION operator returns all the product_id values from both the first UNION result set and the third query result set, eliminating any duplicates. The final result set of this compound query is the product_id values that are current products, all products that were released before January 1, 2008 not just current products, and all products that were released before January 1, 2008 and were sold on or after July 10, 2008.

This SELECT statement does not return the desired result set:

```
SELECT product_id
FROM current_products
UNION
SELECT product_id
FROM product_develop
WHERE release_date < TO_DATE('01-JAN-2008')
MINUS
SELECT product_id
FROM line_item
WHERE sale_date >= TO_DATE('10-JUL-2008');
```

The first statement returns all the product_id values from the CURRENT_PRODUCTS table. The second query returns only the product_id values from the PRODUCT_DEVELOP table that released before January 10, 2008. The first UNION operator returns all the product_id values from the first two queries, eliminating any duplicates. The final query pulls only the product_id values from the line_item table with a sale_date value that is on or after July 10, 2008. The MINUS operator returns all the product_id values from the first UNION result set that are not present in the third query result set. The final result set of this compound query is the product_id values that are current products except products that sold on or after July 10, 2008, all products that were released before January 1, 2008 (not just current products), and no products sold on or after July 10, 2008.

This SELECT statement does not return the desired result set:

```
SELECT product_id
```

```
FROM current_products
UNION
SELECT product_id
FROM product_develop
WHERE release_date < TO_DATE('01-JAN-2008')
UNION
SELECT product_id
FROM line_item
WHERE sale_date <= TO_DATE('10-JUL-2008');
```

The first statement returns all the `product_id` values from the `CURRENT_PRODUCTS` table. The second query returns only the `product_id` values from the `PRODUCT_DEVELOP` table that released before January 10, 2008. The first `UNION` operator returns all the `product_id` values from both of the first two queries, eliminating any duplicates. The final query pulls only the `product_id` values from the `line_item` table with a `sale_date` value that is on or before July 10, 2008. The second `UNION` operator returns all the `product_id` values from both the first `UNION` result set and the third query result set. The final result set of this compound query is the `product_id` values that are current products, all products that were released before January 1, 2008 (not just current products), and products that were released before January 1, 2008 and were sold on or before July 10, 2008.

Item: 19 (Ref:1Z0-061.8.4.7)

Evaluate this SQL statement:

```
SELECT product_id, product_name, price
FROM product
WHERE supplier_id IN (SELECT supplier_id
FROM product
WHERE price > 120 OR qty_in_stock > 100);
```

Which values will be displayed?

- ◯ the `product_id`, `product_name`, and `price` of products that are priced greater than $120.00 and have a `qty_in_stock` value greater than 100
- ◯ the `product_id`, `product_name`, and `price` of products that are priced greater than $120.00 or that have a `qty_in_stock` value greater than 100
- ◯ the `product_id`, `product_name`, and `price` of products that are priced greater than $120.00 or that have a `qty_in_stock` value greater than 100, and that have a supplier
- ◯ the `product_id`, `product_name`, and `price` of products supplied by a supplier with products that are priced greater than $120.00 or with products that have a `qty_in_stock` value greater than 100

Answer:

the `product_id`, `product_name`, and `price` of products supplied by a supplier with products that are priced greater than $120.00 or with products that have a `qty_in_stock` value greater than 100

Explanation:

The subquery will return the `supplier_id` values of products that have a `price` value greater than $120.00 or have a `qty_in_stock` value greater than 100. The main query will return the `product_id`, `product_name`, and `price` values for all products with `supplier_id` values equal to those returned by the subquery.

The query does not return the `product_id`, `product_name`, and `price` of products that are priced greater than $120.00 and have a `qty_in_stock` value greater than 100. To accomplish this, you would use the following query:

```
SELECT product_id, product_name, price
FROM product
WHERE price > 120 AND qty_in_stock > 100;
```

The query does not return the `product_id`, `product_name`, and `price` of products that are priced greater than $120.00 or that have a `qty_in_stock` value greater than 100. To accomplish this, you would use the following query:

```
SELECT product_id, product_name, price
FROM product
WHERE price > 120 OR qty_in_stock > 100;
```

The query does not return the `product_id`, `product_name`, and `price` of products that are priced greater than $120.00 or that have a `qty_in_stock` value greater than 100, and that have a supplier. To accomplish this, you would use the following query:

```
SELECT product_id, product_name, price
FROM product
WHERE (price > 120 OR qty_in_stock > 100)
AND supplier_id IS NOT NULL;
```

Item: 20 (Ref:1Z0-061.8.7.3)

Evaluate this compound query statement:

```
SELECT emp_id, last_name || ', '|| first_name
FROM emp
INTERSECT
SELECT emp_id
FROM emp_hist;
```

Which statement is TRUE regarding the given `SELECT` statement?

- ◯ Duplicate `emp_id` values will be included.
- ◯ The output will be sorted by the `emp_id` values in ascending order.
- ◯ The results will contain the distinct `emp_id` values return by either query.
- ◯ The statement will return an error.

Answer:

The statement will return an error.

Explanation:

This compound query will return an error. The number and data types of the columns in the `SELECT` list of both queries must be the same in number and data type. In this compound query, the first `SELECT` list in the compound query has two columns listed. The second `SELECT` list in the compound query only has one column listed. The names of the columns are not required to match, but the number and data type must match. Removing the `last_name || ', '|| first_name` concatenated columns from the first `SELECT` list would allow the compound query to execute.

The remaining options are incorrect because the compound query will return an error.