**Lesson 2: Understanding Logical Query Processing**

T-SQL has both logical and physical sides to it. The logical side is the conceptual interpretation

of the query that explains what the correct result of the query is. The physical side is the

processing of the query by the database engine. Physical processing must produce the result

defined by logical query processing. To achieve this goal, the database engine can apply optimization.

Optimization can rearrange steps from logical query processing or remove steps

altogether—but only as long as the result remains the one defined by logical query processing.

The focus of this lesson is *logical query processing*—the conceptual interpretation of the

query that defines the correct result.

After this lesson, you will be able to:

■■ Understand the reasoning for the design of T-SQL.

■■ Describe the main logical query processing phases.

■■ Explain the reasons for some of the restrictions in T-SQL.

Estimated lesson time: 40 minutes

T-SQL As a Declarative English-Like Language

T-SQL, being based on standard SQL, is a declarative English-like language. In this language,

declarative means you define *what* you want, as opposed to *imperative* languages that define

also *how* to achieve what you want. Standard SQL describes the logical interpretation of the

declarative request (the “what” part), but it’s the database engine’s responsibility to figure out

how to physically process the request (the “how” part).

For this reason, it is important not to draw any performance-related conclusions from

what you learn about logical query processing. That’s because logical query processing only

defines the correctness of the query. When addressing performance aspects of the query, you

need to understand how optimization works. As mentioned, optimization can be quite different

from logical query processing because it’s allowed to change things as long as the result

achieved is the one defined by logical query processing.

It’s interesting to note that the standard language SQL wasn’t originally called so; rather,

it was called SEQUEL; an acronym for “structured English query language.” But then due to a

trademark dispute with an airline company, the language was renamed to SQL, for “structured

query language.” Still, the point is that you provide your instructions in an English-like manner.

For example, consider the instruction, “Bring me a soda from the refrigerator.” Observe

that in the instruction in English, the object comes before the location. Consider the following

request in T-SQL.

SELECT shipperid, phone, companyname

FROM Sales.Shippers;

Observe the similarity of the query’s keyed-in order to English. The query first indicates the

SELECT list with the attributes you want to return and then the FROM clause with the table

you want to query.

Now try to think of the order in which the request needs to be logically interpreted. For

example, how would you define the instructions to a robot instead of a human? The original

English instruction to get a soda from the refrigerator would probably need to be revised to

something like, “Go to the refrigerator; open the door; get a soda; bring it to me.”

Similarly, the logical processing of a query must first know which table is being queried

before it can know which attributes can be returned from that table. Therefore, contrary to

the keyed-in order of the previous query, the logical query processing has to be as follows.

FROM Sales.Shippers

SELECT shipperid, phone, companyname

This is a basic example with just two query clauses. Of course, things can get more complex.

If you understand the concept of logical query processing well, you will be able to explain

many things about the way the language behaves—things that are very hard to explain

otherwise.

Logical Query Processing Phases

This section covers logical query processing and the phases involved. Don’t worry if some of

the concepts discussed here aren’t clear yet. Subsequent chapters in this Training Kit provide

more detail, and after you go over those, this topic should make more sense. To make sure

you really understand these concepts, make a first pass over the topic now and then revisit it

later after going over Chapters 2 through 5.

The main statement used to retrieve data in T-SQL is the SELECT statement. Following are

the main query clauses specified in the order that you are supposed to type them (known as

“keyed-in order”):

1. SELECT

2. FROM

3. WHERE

4. GROUP BY

5. HAVING

6. ORDER BY

But as mentioned, the logical query processing order, which is the conceptual interpretation

order, is different. It starts with the FROM clause. Here is the logical query processing

order of the six main query clauses:

1. FROM

2. WHERE

3. GROUP BY

4. HAVING

5. SELECT

6. ORDER BY

Each phase operates on one or more tables as inputs and returns a virtual table as output.

The output table of one phase is considered the input to the next phase. This is in accord with

operations on relations that yield a relation. Note that if an ORDER BY is specified, the result

isn’t relational. This fact has implications that are discussed later in this Training Kit, in Chapter 3,

“Filtering and Sorting Data,” and Chapter 4, “Combining Sets.”

Consider the following query as an example.

SELECT country, YEAR(hiredate) AS yearhired, COUNT(\*) AS numemployees

FROM HR.Employees

WHERE hiredate >= '20030101'

GROUP BY country, YEAR(hiredate)

HAVING COUNT(\*) > 1

ORDER BY country , yearhired DESC;

This query is issued against the HR.Employees table. It filters only employees that were

hired in or after the year 2003. It groups the remaining employees by country and the hire

year. It keeps only groups with more than one employee. For each qualifying group, the

query returns the hire year and count of employees, sorted by country and hire year, in descending

order.

The following sections provide a brief description of what happens in each phase according

to logical query processing.

1. Evaluate the FROM Clause

In the first phase, the FROM clause is evaluated. That’s where you indicate the tables you want

to query and table operators like joins if applicable. If you need to query just one table, you

indicate the table name as the input table in this clause. Then, the output of this phase is a

table result with all rows from the input table. That’s the case in the following query: the input

is the HR.Employees table (nine rows), and the output is a table result with all nine rows (only

a subset of the attributes are shown).

empid hiredate country

------ ----------- --------

1 2002-05-01 USA

2 2002-08-14 USA

3 2002-04-01 USA

4 2003-05-03 USA

5 2003-10-17 UK

6 2003-10-17 UK

7 2004-01-02 UK

8 2004-03-05 USA

9 2004-11-15 UK

2. Filter Rows Based on the WHERE Clause

The second phase filters rows based on the predicate in the WHERE clause. Only rows for

which the predicate evaluates to true are returned.

*Exam Tip*

Rows for which the predicate evaluates to false, or evaluates to an unknown state, are not

returned.

In this query, the WHERE filtering phase filters only rows for employees hired on or after

January 1, 2003. Six rows are returned from this phase and are provided as input to the next

one. Here’s the result of this phase.

empid hiredate country

------ ----------- --------

4 2003-05-03 USA

5 2003-10-17 UK

6 2003-10-17 UK

7 2004-01-02 UK

8 2004-03-05 USA

9 2004-11-15 UK

A typical mistake made by people who don’t understand logical query processing is attempting

to refer in the WHERE clause to a column alias defined in the SELECT clause. This

isn’t allowed because the WHERE clause is evaluated before the SELECT clause. As an example,

consider the following query.

SELECT country, YEAR(hiredate) AS yearhired

FROM HR.Employees

WHERE yearhired >= 2003;

This query fails with the following error.

Msg 207, Level 16, State 1, Line 3

Invalid column name 'yearhired'.

If you understand that the WHERE clause is evaluated before the SELECT clause, you realize

that this attempt is wrong because at this phase, the attribute yearhired doesn’t yet exist.

You can indicate the expression YEAR(hiredate) >= 2003 in the WHERE clause. Better yet, for

optimization reasons that are discussed in Chapter 3 and Chapter 15, “Implementing Indexes

and Statistics,” use the form hiredate >= '20030101' as done in the original query.

3. Group Rows Based on the GROUP BY Clause

This phase defines a group for each distinct combination of values in the grouped elements

from the input table. It then associates each input row to its respective group. The query

you’ve been working with groups the rows by country and YEAR(orderdate). Within the six

rows in the input table, this step identifies four groups. Here are the groups and the detail rows

that are associated with them (redundant information removed for purposes of illustration).

group group detail detail detail

country YEAR(hiredate) empid country hiredate

-------- -------------- ------ ------- ----------

UK 2003 5 UK 2003-10-17

6 UK 2003-10-17

UK 2004 7 UK 2004-01-02

9 UK 2004-11-15

USA 2003 4 USA 2003-05-03

USA 2004 8 USA 2004-03-05

As you can see, the group UK, 2003 has two associated detail rows with employees 5 and

6; the group for UK, 2004 also has two associated detail rows with employees 7 and 9; the

group for USA, 2003 has one associated detail row with employee 4; the group for USA, 2004

also has one associated detail row with employee 8.

The final result of this query has one row representing each group (unless filtered out).

Therefore, expressions in all phases that take place after the current grouping phase are

somewhat limited. All expressions processed in subsequent phases must guarantee a single

value per group. If you refer to an element from the GROUP BY list (for example, country),

you already have such a guarantee, so such a reference is allowed. However, if you want to

refer to an element that is not part of your GROUP BY list (for example, empid), it must be

contained within an aggregate function like MAX or SUM. That’s because multiple values are

possible in the element within a single group, and the only way to guarantee that just one will

be returned is to aggregate the values. For more details on grouped queries, see Chapter 5,

“Grouping and Windowing.”

4. Filter Rows Based on the HA VING Clause

This phase is also responsible for filtering data based on a predicate, but it is evaluated after

the data has been grouped; hence, it is evaluated per group and filters groups as a whole. As

is usual in T-SQL, the filtering predicate can evaluate to true, false, or unknown. Only groups

for which the predicate evaluates to true are returned from this phase. In this case, the HAVING

clause uses the predicate COUNT(\*) > 1, meaning filter only country and hire year groups that

have more than one employee. If you look at the number of rows that were associated with

each group in the previous step, you will notice that only the groups UK, 2003 and UK, 2004

qualify. Hence, the result of this phase has the following remaining groups, shown here with

their associated detail rows.

group group detail detail detail

country YEAR(hiredate) empid country hiredate

-------- -------------- ------ ------- ----------

UK 2003 5 UK 2003-10-17

6 UK 2003-10-17

UK 2004 7 UK 2004-01-02

9 UK 2004-11-15

**Quick Check**

■■ What is the difference between the WHERE and HAVING clauses?

**Quick Check Answer**

■■ The WHERE clause is evaluated before rows are grouped, and therefore is evaluated

per row. The HAVING clause is evaluated after rows are grouped, and therefore

is evaluated per group.

5. Process the SELECT Clause

The fifth phase is the one responsible for processing the SELECT clause. What’s interesting

about it is the point in logical query processing where it gets evaluated—almost last. That’s

interesting considering the fact that the SELECT clause appears first in the query.

This phase includes two main steps. The first step is evaluating the expressions in the SELECT

list and producing the result attributes. This includes assigning attributes with names if they

are derived from expressions. Remember that if a query is a grouped query, each group is

represented by a single row in the result. In the query, two groups remain after the processing

of the HAVING filter. Therefore, this step generates two rows. In this case, the SELECT list

returns for each country and order year group a row with the following attributes: country,

YEAR(hiredate) aliased as yearhired, and COUNT(\*) aliased as numemployees.

The second step in this phase is applicable if you indicate the DISTINCT clause, in which

case this step removes duplicates. Remember that T-SQL is based on multiset theory more

than it is on set theory, and therefore, if duplicates are possible in the result, it’s your responsibility

to remove those with the DISTINCT clause. In this query’s case, this step is inapplicable.

Here’s the result of this phase in the query.

country yearhired numemployees

-------- ---------- ------------

UK 2003 2

UK 2004 2

If you need a reminder of what the query looks like, here it is again.

SELECT country, YEAR(hiredate) AS yearhired, COUNT(\*) AS numemployees

FROM HR.Employees

WHERE hiredate >= '20030101'

GROUP BY country, YEAR(hiredate)

HAVING COUNT(\*) > 1

ORDER BY country , yearhired DESC;

The fifth phase returns a relational result. Therefore, the order of the rows isn’t guaranteed.

In this query’s case, there is an ORDER BY clause that guarantees the order in the result,

but this will be discussed when the next phase is described. What’s important to note is that

the outcome of the phase that processes the SELECT clause is still relational.

Also, remember that this phase assigns column aliases, like yearhired and numemployees.

This means that newly created column aliases are not visible to clauses processed in previous

phases, like FROM, WHERE, GROUP BY, and HAVING.

Note that an alias created by the SELECT phase isn’t even visible to other expressions that

appear in the same SELECT list. For example, the following query isn’t valid.

SELECT empid, country, YEAR(hiredate) AS yearhired, yearhired - 1 AS prevyear

FROM HR.Employees;

This query generates the following error.

Msg 207, Level 16, State 1, Line 1

Invalid column name 'yearhired'.

The reason that this isn’t allowed is that, conceptually, T-SQL evaluates all expressions that

appear in the same logical query processing phase in an all-at-once manner. Note the use of

the word *conceptually*. SQL Server won’t necessarily physically process all expressions at the

same point in time, but it has to produce a result as if it did. This behavior is different than

many other programming languages where expressions usually get evaluated in a left-toright

order, making a result produced in one expression visible to the one that appears to its

right. But T-SQL is different.

**Quick Check**

1. Why are you not allowed to refer to a column alias defined by the SELECT

clause in the WHERE clause?

2. Why are you not allowed to refer to a column alias defined by the SELECT

clause in the same SELECT clause?

**Quick Check Answer**

1. Because the WHERE clause is logically evaluated in a phase earlier to the one

that evaluates the SELECT clause.

2. Because all expressions that appear in the same logical query processing phase

are evaluated conceptually at the same point in time.

6. Handle Presentation Ordering

The sixth phase is applicable if the query has an ORDER BY clause. This phase is responsible

for returning the result in a specific presentation order according to the expressions that appear

in the ORDER BY list. The query indicates that the result rows should be ordered first by

country (in ascending order by default), and then by numemployees, descending, yielding the

following output.

country yearhired numemployees

-------- ---------- ------------

UK 2004 2

UK 2003 2

Notice that the ORDER BY clause is the first and only clause that is allowed to refer to column

aliases defined in the SELECT clause. That’s because the ORDER BY clause is the only one

to be evaluated after the SELECT clause.

Unlike in previous phases where the result was relational, the output of this phase isn’t

relational because it has a guaranteed order. The result of this phase is what standard SQL

calls a cursor. Note that the use of the term cursor here is conceptual. T-SQL also supports an

object called a cursor that is defined based on a result of a query, and that allows fetching

rows one at a time in a specified order.

You might care about returning the result of a query in a specific order for presentation

purposes or if the caller needs to consume the result in that manner through some cursor

mechanism that fetches the rows one at a time. But remember that such processing isn’t

relational. If you need to process the query result in a relational manner—for example, define

a table expression like a view based on the query (details later in Chapter 4)—the result will

need to be relational. Also, sorting data can add cost to the query processing. If you don’t

care about the order in which the result rows are returned, you can avoid this unnecessary

cost by not adding an ORDER BY clause.

A query may specify the TOP or OFFSET-FETCH filtering options. If it does, the same

ORDER BY clause that is normally used to define presentation ordering also defines which

rows to filter for these options. It’s important to note that such a filter is processed after the

SELECT phase evaluates all expressions and removes duplicates (in case a DISTINCT clause was

specified). You might even consider the TOP and OFFSET-FETCH filters as being processed in

their own phase number 7. The query doesn’t indicate such a filter, and therefore, this phase is

inapplicable in this case.

Practice **Logical Query Processing**

In this practice, you exercise your knowledge of logical query processing.

If you encounter a problem completing an exercise, you can install the completed projects

from the Solution folder that is provided with the companion content for this chapter and

lesson.

Exercise 1 Fix a Problem with Grouping

In this exercise, you are presented with a grouped query that fails when you try to execute it.

You are provided with instructions on how to fix the query.

1. Open SSMS and connect to the sample database TSQL2012.

2. Type the following query in the query window and execute it.

SELECT custid, orderid

FROM Sales.Orders

GROUP BY custid;

The query was supposed to return for each customer the customer ID and the maximum

order ID for that customer, but instead it fails. Try to figure out why the query

failed and what needs to be revised so that it would return the desired result.

3. The query failed because orderid neither appears in the GROUP BY list nor within an

aggregate function. There are multiple possible orderid values per customer. To fix the

query, you need to apply an aggregate function to the orderid attribute. The task is to

return the maximum orderid value per customer. Therefore, the aggregate function

should be MAX. Your query should look like the following.

SELECT custid, MAX(orderid) AS maxorderid

FROM Sales.Orders

GROUP BY custid;

Exercise 2 Fix a Problem with Aliasing

In this exercise, you are presented with another grouped query that fails, this time because of

an aliasing problem. As in the first exercise, you are provided with instructions on how to fix

the query.

1. Clear the query window, type the following query, and execute it.

SELECT shipperid, SUM(freight) AS totalfreight

FROM Sales.Orders

WHERE freight > 20000.00

GROUP BY shipperid;

The query was supposed to return only shippers for whom the total freight value is

greater than 20,000, but instead it returns an empty set. Try to identify the problem in

the query.

2. Remember that the WHERE filtering clause is evaluated per row—not per group. The

query filters individual orders with a freight value greater than 20,000, and there are

none. To correct the query, you need to apply the filter per each shipper group—not

per each order. You need to filter the total of all freight values per shipper. This can be

achieved by using the HAVING filter. You try to fix the problem by using the following

query.

SELECT shipperid, SUM(freight) AS totalfreight

FROM Sales.Orders

GROUP BY shipperid

HAVING totalfreight > 20000.00;

But this query also fails. Try to identify why it fails and what needs to be revised to

achieve the desired result.

3. The problem now is that the query attempts to refer in the HAVING clause to the alias

totalfreight, which is defined in the SELECT clause. The HAVING clause is evaluated

before the SELECT clause, and therefore, the column alias isn’t visible to it. To fix the

problem, you need to refer to the expression SUM(freight) in the HAVING clause, as

follows.

SELECT shipperid, SUM(freight) AS totalfreight

FROM Sales.Orders

GROUP BY shipperid

HAVING SUM(freight) > 20000.00;

Lesson Summary

■■ T-SQL was designed as a declarative language where the instructions are provided in

an English-like manner. Therefore, the keyed-in order of the query clauses starts with

the SELECT clause.

■■ Logical query processing is the conceptual interpretation of the query that defines the

correct result, and unlike the keyed-in order of the query clauses, it starts by evaluating

the FROM clause.

■■ Understanding logical query processing is crucial for correct understanding of T-SQL.

Lesson Review

Answer the following questions to test your knowledge of the information in this lesson. You

can find the answers to these questions and explanations of why each answer choice is correct

or incorrect in the “Answers” section at the end of this chapter.

1. Which of the following correctly represents the logical query processing order of the

various query clauses?

A. SELECT > FROM > WHERE > GROUP BY > HAVING > ORDER BY

B. FROM > WHERE > GROUP BY > HAVING > SELECT > ORDER BY

C. FROM > WHERE > GROUP BY > HAVING > ORDER BY > SELECT

D. SELECT > ORDER BY > FROM > WHERE > GROUP BY > HAVING

2. Which of the following is invalid? (Choose all that apply.)

A. Referring to an attribute that you group by in the WHERE clause

B. Referring to an expression in the GROUP BY clause; for example, GROUP BY

YEAR(orderdate)

C. In a grouped query, referring in the SELECT list to an attribute that is not part of

the GROUP BY list and not within an aggregate function

D. Referring to an alias defined in the SELECT clause in the HAVING clause

3. What is true about the result of a query without an ORDER BY clause?

A. It is relational as long as other relational requirements are met.

B. It cannot have duplicates.

C. The order of the rows in the output is guaranteed to be the same as the insertion

order.

D. The order of the rows in the output is guaranteed to be the same as that of the

clustered index.

**Case Scenarios**

In the following case scenarios, you apply what you’ve learned about T-SQL querying. You can

find the answers to these questions in the “Answers” section at the end of this chapter.

Case Scenario 1: Importance of Theory

You and a colleague on your team get into a discussion about the importance of understanding

the theoretical foundations of T-SQL. Your colleague argues that there’s no point in understanding

the foundations, and that it’s enough to just learn the technical aspects of T-SQL

to be a good developer and to write correct code. Answer the following questions posed to

you by your colleague:

1. Can you give an example for an element from set theory that can improve your understanding

of T-SQL?

2. Can you explain why understanding the relational model is important for people who

write T-SQL code?

Case Scenario 2: Interviewing for a Code Reviewer Position

You are interviewed for a position as a code reviewer to help improve code quality. The organization’s

application has queries written by untrained people. The queries have numerous

problems, including logical bugs. Your interviewer poses a number of questions and asks for

a concise answer of a few sentences to each question. Answer the following questions addressed

to you by your interviewer:

1. Is it important to use standard code when possible, and why?

2. We have many queries that use ordinal positions in the ORDER BY clause. Is that a bad

practice, and if so why?

3. If a query doesn’t have an ORDER BY clause, what is the order in which the records are

returned?

4. Would you recommend putting a DISTINCT clause in every query?

**Suggested Practices**

To help you successfully master the exam objectives presented in this chapter, complete the

following tasks.

Visit T-SQL Public Newsgroups and Review Code

To practice your knowledge of using T-SQL in a relational way, you should review code

samples written by others.

■■ **Practice 1** List as many examples as you can for aspects of T-SQL coding that are not

relational.

■■ **Practice 2** After creating the list in Practice 1, visit the Microsoft public forum for

T-SQL at *http://social.msdn.microsoft.com/Forums/en/transactsql/threads*. Review code

samples in the T-SQL threads. Try to identify cases where nonrelational elements are

used; if you find such cases, identify what needs to be revised to make them relational.

Describe Logical Query Processing

To better understand logical query processing, we recommend that you complete the following

tasks:

■■ **Practice 1** Create a document with a numbered list of the phases involved with logical

query processing in the correct order. Provide a brief paragraph summarizing what

happens in each step.

■■ **Practice 2** Create a graphical flow diagram representing the flow of the logical query

processing phases by using a tool such as Microsoft Visio, Microsoft PowerPoint, or

Microsoft Word.

**Answers**

This section contains the answers to the lesson review questions and solutions to the case

scenarios in this chapter.

Lesson 1

1. **Correct Answers: B and D**

A. **Incorrect:** It is important to use standard code.

B. **Correct:** Use of standard code makes it easier to port code between platforms

because fewer revisions are required.

C. **Incorrect:** There’s no assurance that standard code will be more efficient.

D. **Correct:** When using standard code, you can adapt to a new environment more

easily because standard code elements look similar in the different platforms.

2. **Correct Answer: D**

A. **Incorrect:** A relation has a header with a set of attributes, and tuples of the relation

have the same heading. A set has no order, so ordinal positions do not have

meaning and constitute a violation of the relational model. You should refer to

attributes by their name.

B. **Incorrect:** A query is supposed to return a relation. A relation has a body with a

set of tuples. A set has no duplicates. Returning duplicate rows is a violation of the

relational model.

C. **Incorrect:** Not defining a key in the table allows duplicate rows in the table, and

like the answer to B, that’s a violation of the relational model.

D. **Correct:** Because attributes are supposed to be identified by name, ensuring that

all attributes have names is relational, and hence not a violation of the relational

model.

3. **Correct Answer: B**

A. **Incorrect:** T-SQL isn’t standard and SQL isn’t a dialect in Microsoft SQL Server.

B. **Correct:** SQL is standard and T-SQL is a dialect in Microsoft SQL Server.

C. **Incorrect:** T-SQL isn’t standard.

D. **Incorrect:** SQL isn’t a dialect in Microsoft SQL Server.

Lesson 2

1. **Correct Answer: B**

A. **Incorrect:** Logical query processing doesn’t start with the SELECT clause.

B. **Correct:** Logical query processing starts with the FROM clause, and then moves on

to WHERE, GROUP BY, HAVING, SELECT, and ORDER BY.

C. **Incorrect:** The ORDER BY clause isn’t evaluated before the SELECT clause.

D. **Incorrect:** Logical query processing doesn’t start with the SELECT clause.

2. **Correct Answer: C and D**

A. **Incorrect:** T-SQL allows you to refer to an attribute that you group by in the

WHERE clause.

B. **Incorrect:** T-SQL allows grouping by an expression.

C. **Correct:** If the query is a grouped query, in phases processed after the GROUP BY

phase, each attribute that you refer to must appear either in the GROUP BY list or

within an aggregate function.

D. **Correct:** Because the HAVING clause is evaluated before the SELECT clause, referring

to an alias defined in the SELECT clause within the HAVING clause is invalid.

3. **Correct Answer: A**

A. **Correct:** A query with an ORDER BY clause doesn’t return a relational result. For

the result to be relational, the query must satisfy a number of requirements, including

the following : the query must not have an ORDER BY clause, all attributes

must have names, all attribute names must be unique, and duplicates must not

appear in the result.

B. **Incorrect:** A query without a DISTINCT clause in the SELECT clause can return

duplicates.

C. **Incorrect:** A query without an ORDER BY clause does not guarantee the order of

rows in the output.

D. **Incorrect:** A query without an ORDER BY clause does not guarantee the order of

rows in the output.

Case Scenario 1

1. One of most typical mistakes that T-SQL developers make is to assume that a query

without an ORDER BY clause always returns the data in a certain order—for example,

clustered index order. But if you understand that in set theory, a set has no particular

order to its elements, you know that you shouldn’t make such assumptions. The only

way in SQL to guarantee that the rows will be returned in a certain order is to add an

ORDER BY clause. That’s just one of many examples for aspects of T-SQL that can be

better understood if you understand the foundations of the language.

2. Even though T-SQL is based on the relational model, it deviates from it in a number of

ways. But it gives you enough tools that if you understand the relational model, you

can write in a relational way. Following the relational model helps you write code more

correctly. Here are some examples :

■■ You shouldn’t rely on order of columns or rows.

■■ You should always name result columns.

■■ You should eliminate duplicates if they are possible in the result of your query.

Case Scenario 2

1. It is important to use standard SQL code. This way, both the code and people’s knowledge

is more portable. Especially in cases where there are both standard and nonstandard

forms for a language element, it’s recommended to use the standard form.

2. Using ordinal positions in the ORDER BY clause is a bad practice. From a relational perspective,

you are supposed to refer to attributes by name, and not by ordinal position.

Also, what if the SELECT list is revised in the future and the developer forgets to revise

the ORDER BY list accordingly?

3. When the query doesn’t have an ORDER BY clause, there are no assurances for any

particular order in the result. The order should be considered arbitrary. You also notice

that the interviewer used the incorrect term *record* instead of *row*. You might want to

mention something about this, because the interviewer may have done so on purpose

to test you.

4. From a pure relational perspective, this actually could be valid, and perhaps even

recommended. But from a practical perspective, there is the chance that SQL Server

will try to remove duplicates even when there are none, and this will incur extra cost.

Therefore, it is recommended that you add the DISTINCT clause only when duplicates

are possible in the result and you’re not supposed to return the duplicates.