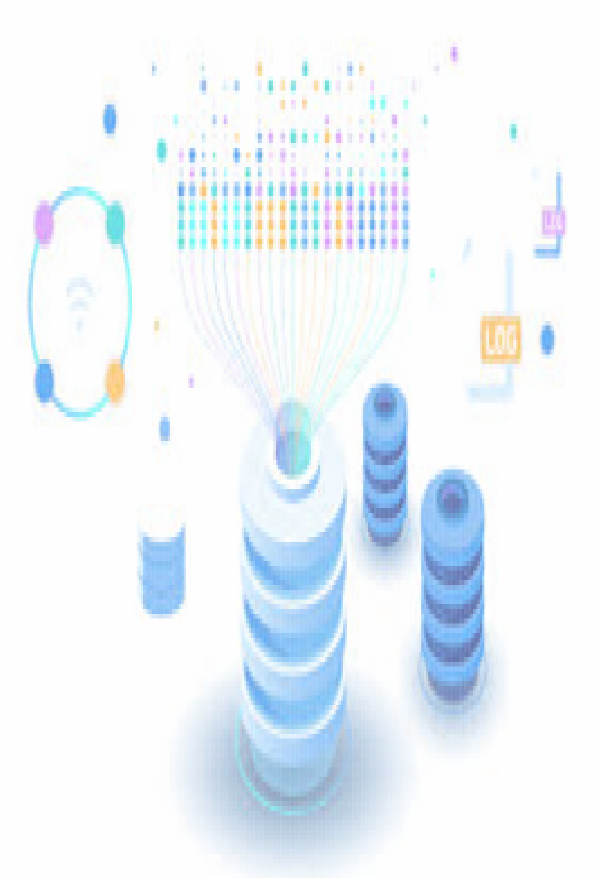


CHAPTER 7

PRACTICAL ISSUES OF DATABASE APPLICATION

OBJECTIVES

- Understand transactions and their properties (ACID)
- Apply transaction in database application programming
- Understand the roles of indexing techniques
- Implement indexes for query optimization
- Understand what views are for and how to use them
- Understand query execution plan for query optimization analysis

CONTENTS

1. Transaction in SQL
2. Indexes in SQL & Query optimization
3. Views

Introduction

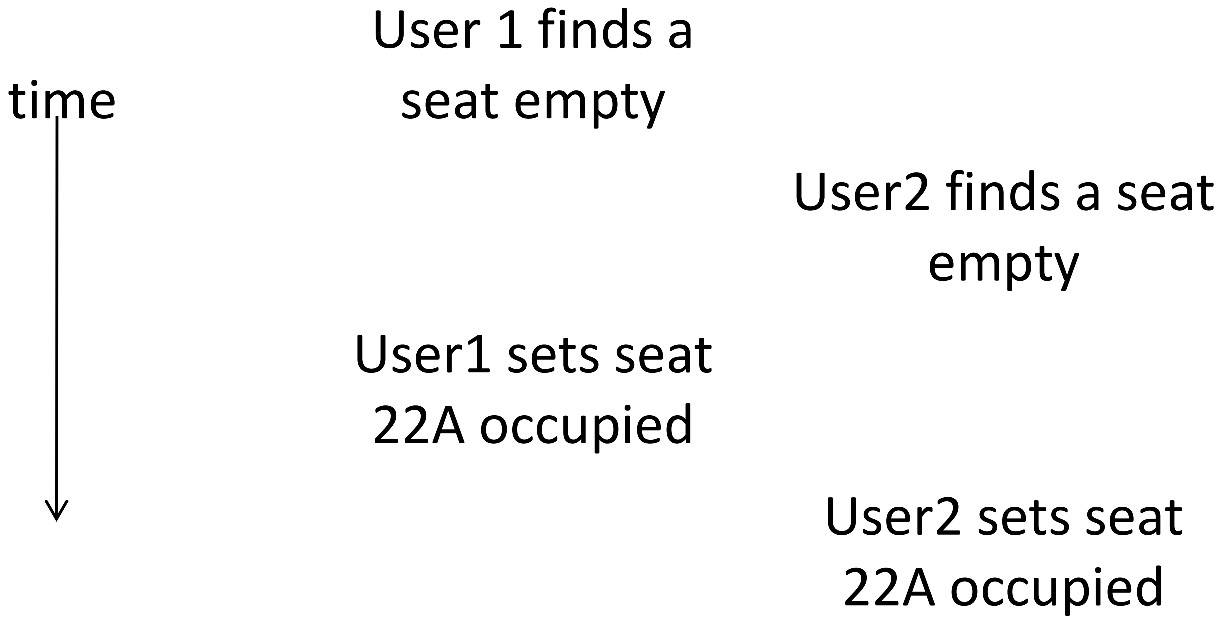
- DB User operates on database by querying or modifying the database
- Operations on database are executed one at a time
- Output of one operation is input of the next operation
- So, how the DBMS treats simultaneous operations?

Serializability (tuần tự)

- In applications, many operations per second may be performed on database
- These may operate on the same data
- We'll get unexpected results

Serializability

Example: Two users book the same seat of the flight



time

User 1 finds a seat empty

User2 finds a seat empty

User1 sets seat 22A occupied

User2 sets seat 22A occupied

Serializability

Transaction

- is a group of operations that need to be performed together.

- A certain transaction must be serializable with respect to other transactions, that is, the transactions run serially – one at a time, no overlap

Atomicity (tính nguyên tử)

A certain combinations of database operations need to be done atomically, that is, either they are all done or neither is done

Atomicity

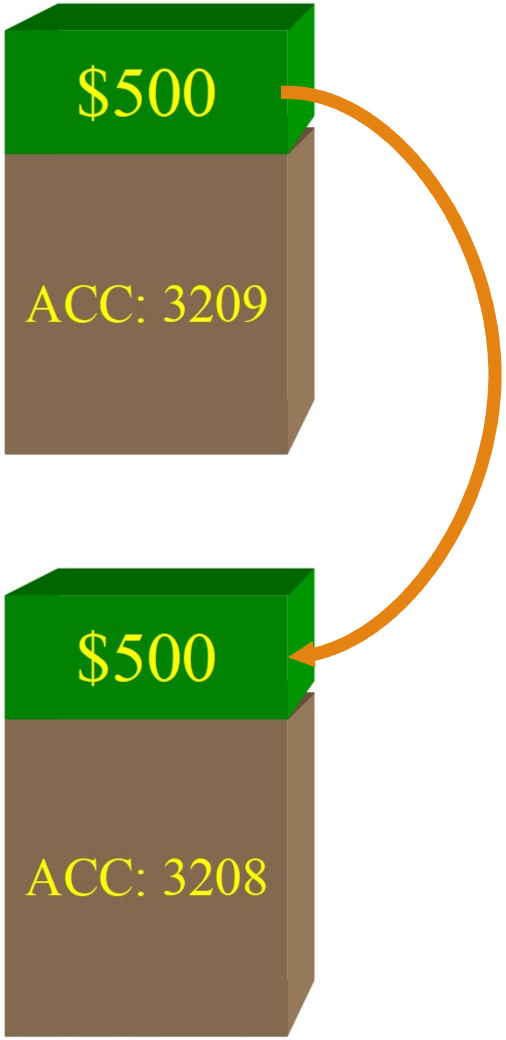
Example:

Transfer $500 from the account number 3209 to account number 3208 by two steps

- (1) Subtract $500 from account number 3209
- (2) Add $500 to account number 3208

What happen if there is a failure after step (1) but before step (2)?

Atomicity



A Banking Transaction

```
UPDATE savings_accounts
     SET balance = balance - 500
     WHERE account = 3209;
```

Decrement Savings Account

```
UPDATE checking_accounts
     SET balance = balance + 500
     WHERE account = 3208;
```

Increment Checking Account

```
INSERT INTO journal VALUES
     (journal_seq.NEXTVAL, '1B'
     3209, 3208, 500);
```

Record in Transaction Journal

```
COMMIT WORK;
```

End Transaction

Transaction Ends

Transactions

- Transaction is a collection of one or more operations on the database that must be executed atomically

- That is, either all operations are performed or none are

- In SQL, each statement is a transaction by itself

- SQL allows to group several statements into a single transaction

Transactions

Transaction begins by SQL command **START TRANSACTION**

Two ways to end a transaction

- The SQL statement **COMMIT** causes the transaction to end successfully
- The SQL statement **ROLLBACK** causes the transaction to abort, or terminate unsuccessfully

ACID properties of Transaction

- ❑**Atomicity** (nguyên tử): Giao dịch phải hoàn thành toàn bộ hoặc không thực hiện gì cả. Nếu một phần của giao dịch thất bại, toàn bộ giao dịch sẽ bị hủy và không có thay đổi nào được thực hiện trong cơ sở dữ liệu.
- ❑**Consistency** (nhất quán): Giao dịch phải đưa cơ sở dữ liệu từ một trạng thái hợp lệ này sang một trạng thái hợp lệ khác
- ❑**Isolation** (độc lập): Giao dịch phải được thực hiện một cách độc lập với các giao dịch khác
- ❑**Durability** (bền vững): Sau khi một giao dịch đã hoàn thành, các thay đổi của nó sẽ tồn tại ngay cả khi có sự cố xảy ra, như mất điện hoặc hệ thống gặp lỗi.

ACID properties of Transaction

Atomicity: a transaction is an atomic unit of processing; it should either be performed in its entirety or not performed at all.

- At the end of the transaction, either all statements of the transaction is successful or all statements of the transaction fail.
- If a partial transaction is written to the disk then the *Atomic* property is violated

Consistency: a transaction should be consistency preserving, meaning that if it is completely executed from beginning to end without interference from other transactions, it should take the database from one consistent state to another.

ACID properties of Transaction

Isolation: a transaction should appear as though it is being executed in isolation from other transactions, even though many transactions are executing concurrently. That is the execution of a transaction should not be interfered with by any other transactions executing concurrently.

Durability : the changes applied to the database by a committed transaction must persist in the database. These changes must not be lost because of any failure..

Read-Only Transactions

- A transaction can read or write some data into the database
- When a transaction only reads data and does not write data, the transaction may execute in parallel with other transactions
- Many read-only transactions access the same data to run in parallel, while they would not be allowed to run in parallel with a transaction that wrote the same data

Transactions

SQL statement set read-only to the next transaction

- SET TRANSACTION READ ONLY;

SQL statement set read/write to the next transaction

- SET TRANSACTION READ WRITE;

Basic Transaction Commands

BEGIN TRANSACTION: Marks the starting point of a transaction.

COMMIT: Saves all changes made by the transaction to the database.

ROLLBACK: Undoes all changes made by the transaction if an error occurs or if you want to cancel it.

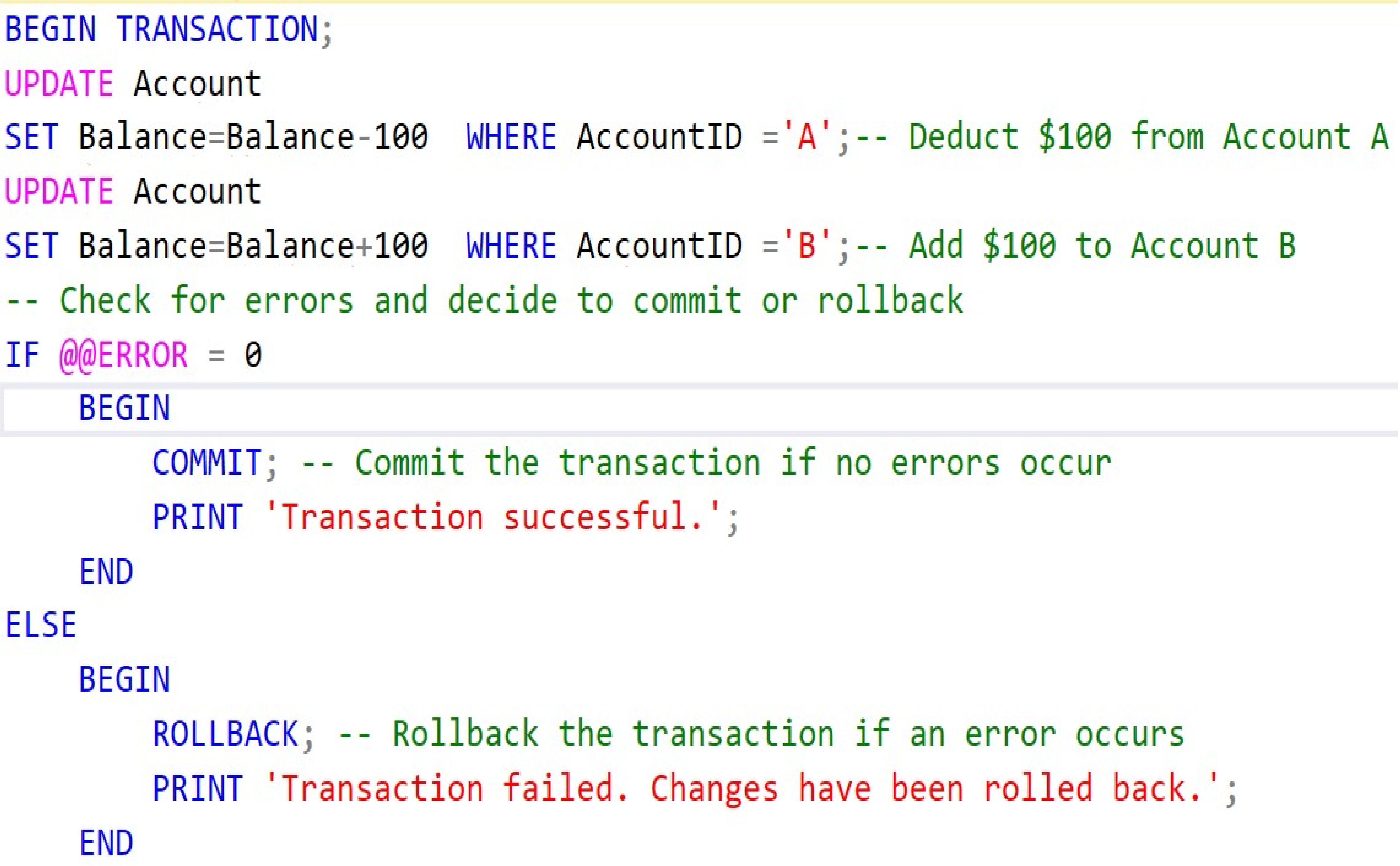
Dirty Reads

- **Dirty data**: data written by a transaction that has not yet committed

Example:

Let's say we are transferring money between two accounts. We want to ensure that either both the debit (ghi nợ) and credit(ghi có) actions are successfully completed, or none at all, to avoid data inconsistency (e.g., deducting money from one account without adding it to the other).

Transaction Example



```sql
BEGIN TRANSACTION;
UPDATE Account
SET Balance=Balance-100  WHERE AccountID ='A';-- Deduct $100 from Account A
UPDATE Account
SET Balance=Balance+100  WHERE AccountID ='B';-- Add $100 to Account B
-- Check for errors and decide to commit or rollback
IF @@ERROR = 0
    BEGIN
        COMMIT; -- Commit the transaction if no errors occur
        PRINT 'Transaction successful.';
    END
ELSE
    BEGIN
        ROLLBACK; -- Rollback the transaction if an error occurs
        PRINT 'Transaction failed. Changes have been rolled back.';
    END
```

Explanation:

BEGIN TRANSACTION: Starts a new transaction.

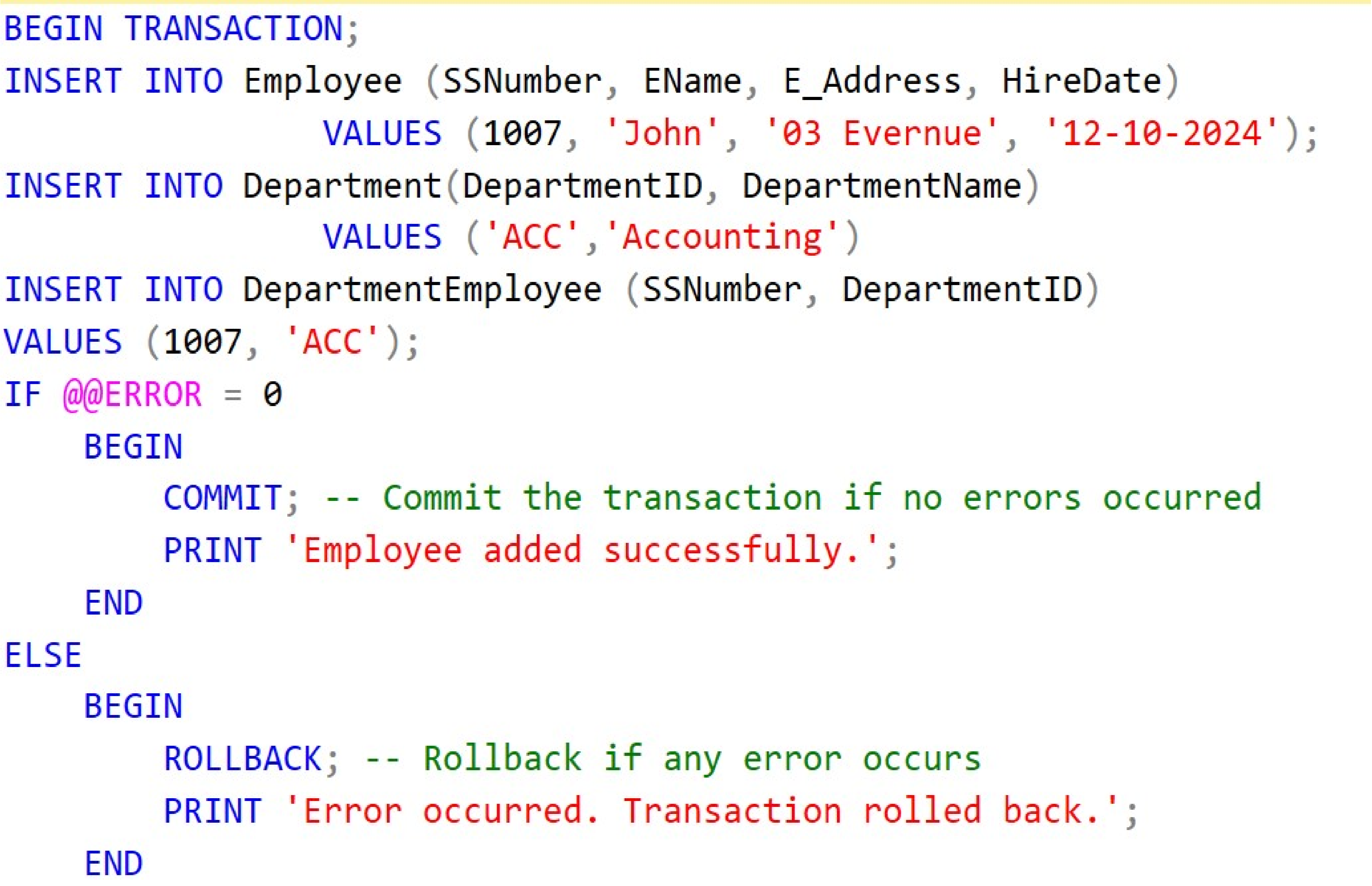
UPDATE statements: Execute the debit and credit operations.

@@ERROR: System function that returns 0 if the last operation was successful, or an error code if it failed.

COMMIT: Confirms and saves the transaction's changes to the database.

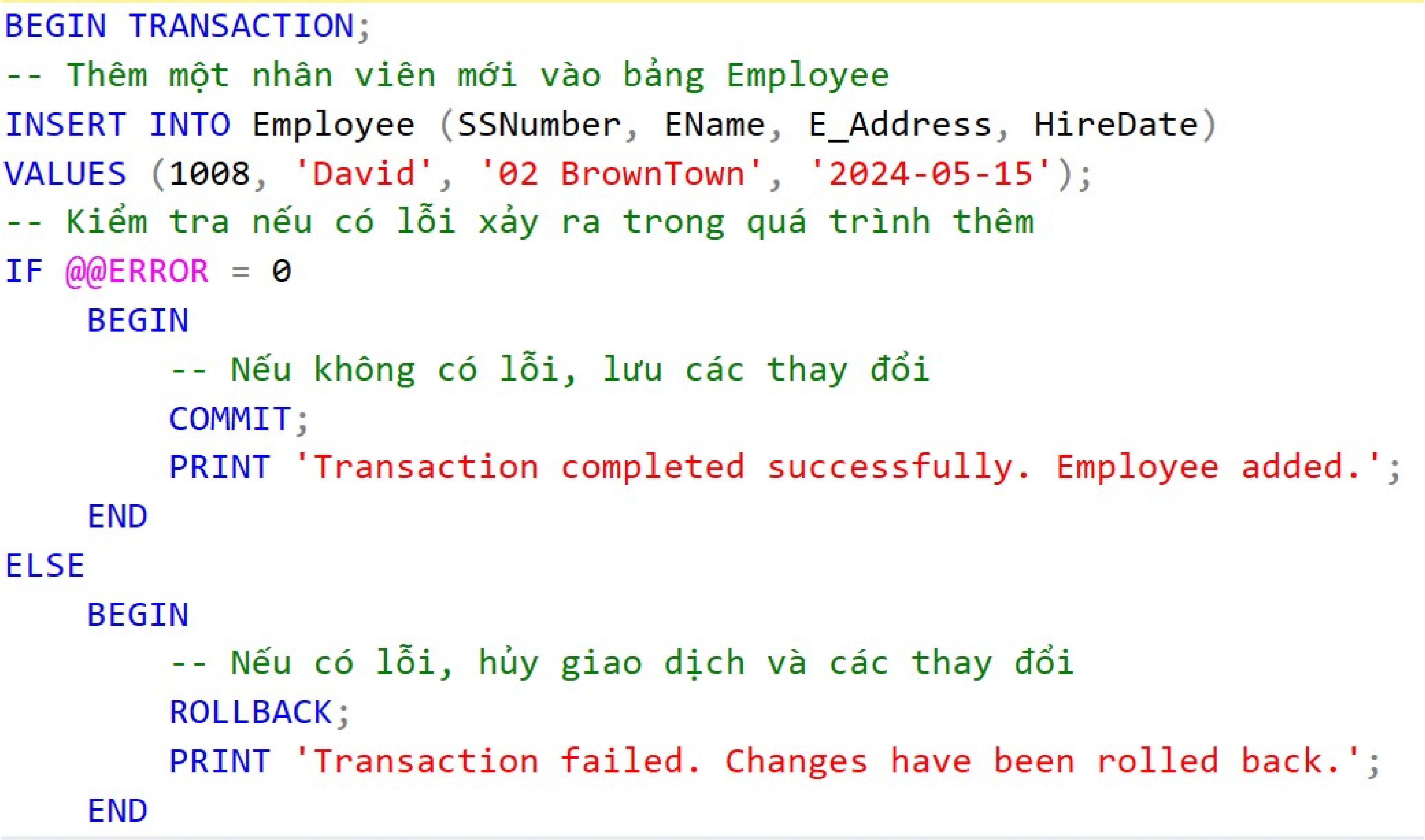
ROLLBACK: If any part of the transaction fails, it undoes all the changes made in the transaction, ensuring data integrity.

Example 1:



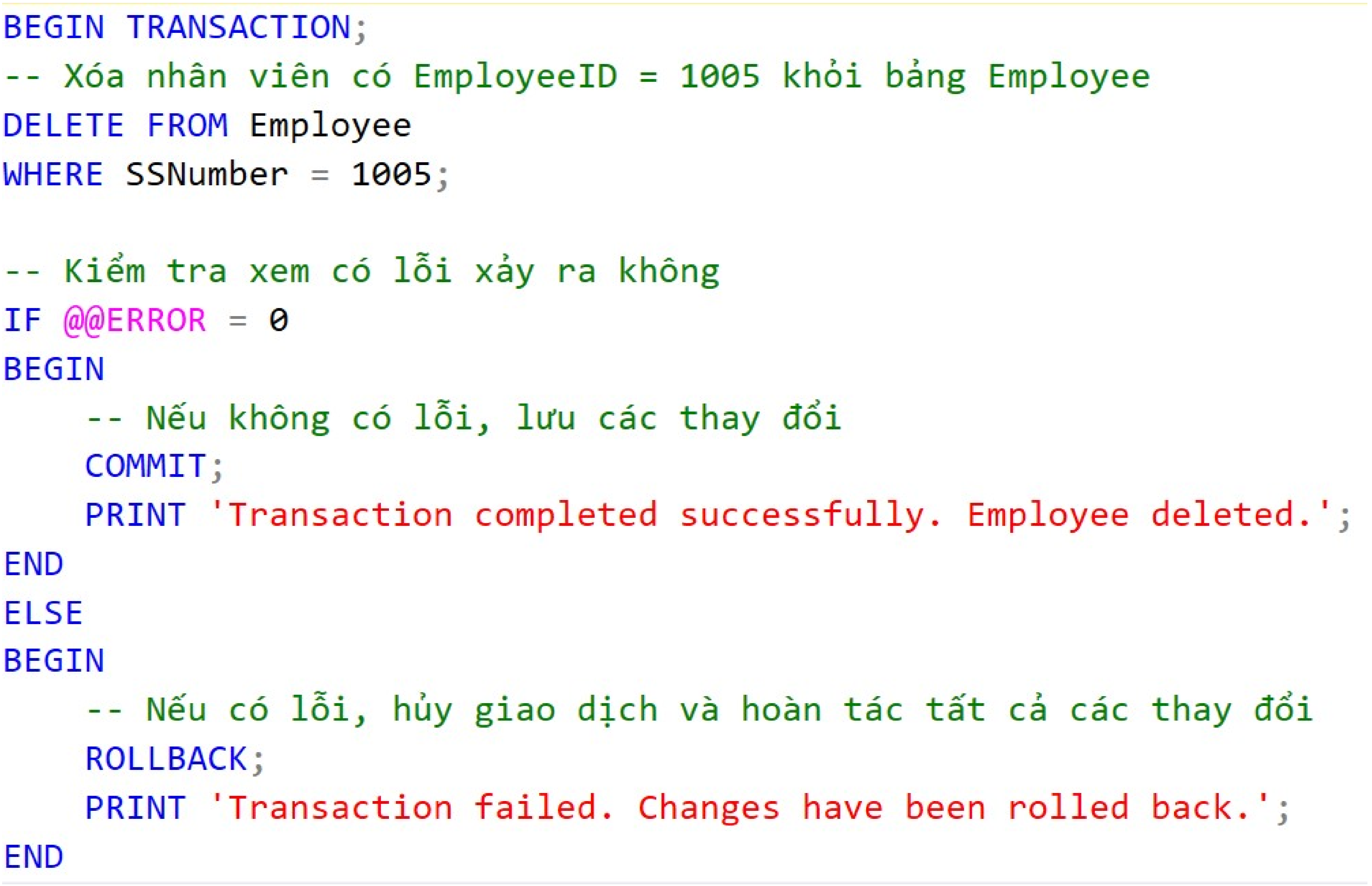
```sql
BEGIN TRANSACTION;
INSERT INTO Employee (SSNumber, EName, E_Address, HireDate)
                VALUES (1007, 'John', '03 Evernue', '12-10-2024');
INSERT INTO Department(DepartmentID, DepartmentName)
                VALUES ('ACC','Accounting')
INSERT INTO DepartmentEmployee (SSNumber, DepartmentID)
VALUES (1007, 'ACC');
IF @@ERROR = 0
    BEGIN
        COMMIT; -- Commit the transaction if no errors occurred
        PRINT 'Employee added successfully.';
    END
ELSE
    BEGIN
        ROLLBACK; -- Rollback if any error occurs
        PRINT 'Error occurred. Transaction rolled back.';
    END
```

Example 2:



```sql
BEGIN TRANSACTION;
-- Thêm một nhân viên mới vào bảng Employee
INSERT INTO Employee (SSNumber, EName, E_Address, HireDate)
VALUES (1008, 'David', '02 BrownTown', '2024-05-15');
-- Kiểm tra nếu có lỗi xảy ra trong quá trình thêm
IF @@ERROR = 0
    BEGIN
        -- Nếu không có lỗi, lưu các thay đổi
        COMMIT;
        PRINT 'Transaction completed successfully. Employee added.';
    END
ELSE
    BEGIN
        -- Nếu có lỗi, hủy giao dịch và các thay đổi
        ROLLBACK;
        PRINT 'Transaction failed. Changes have been rolled back.';
    END
```

Example 3:



```
BEGIN TRANSACTION;
-- Xóa nhân viên có EmployeeID = 1005 khỏi bảng Employee
DELETE FROM Employee
WHERE SSNumber = 1005;

-- Kiểm tra xem có lỗi xảy ra không
IF @@ERROR = 0
BEGIN
    -- Nếu không có lỗi, lưu các thay đổi
    COMMIT;
    PRINT 'Transaction completed successfully. Employee deleted.';
END
ELSE
BEGIN
    -- Nếu có lỗi, hủy giao dịch và hoàn tác tất cả các thay đổi
    ROLLBACK;
    PRINT 'Transaction failed. Changes have been rolled back.';
END
```

Index overview

In SQL Server, an index is a database object that **improves the speed of data retrieval** operations on a table by providing a more efficient path to locate rows.

It is similar to an index in a book, which allows you to **quickly find specific information** without having to read through the entire content.

Index overview

An index on attribute A is a data structure that makes it efficient to find those tuples that have a fixed value for attribute A.

Can dramatically speed up certain operations:

- Find all R tuples where R.A = v
- Find all R and S tuples where R.A = S.B
- Find all R tuples where R.A > v (sometimes, depending on index type)

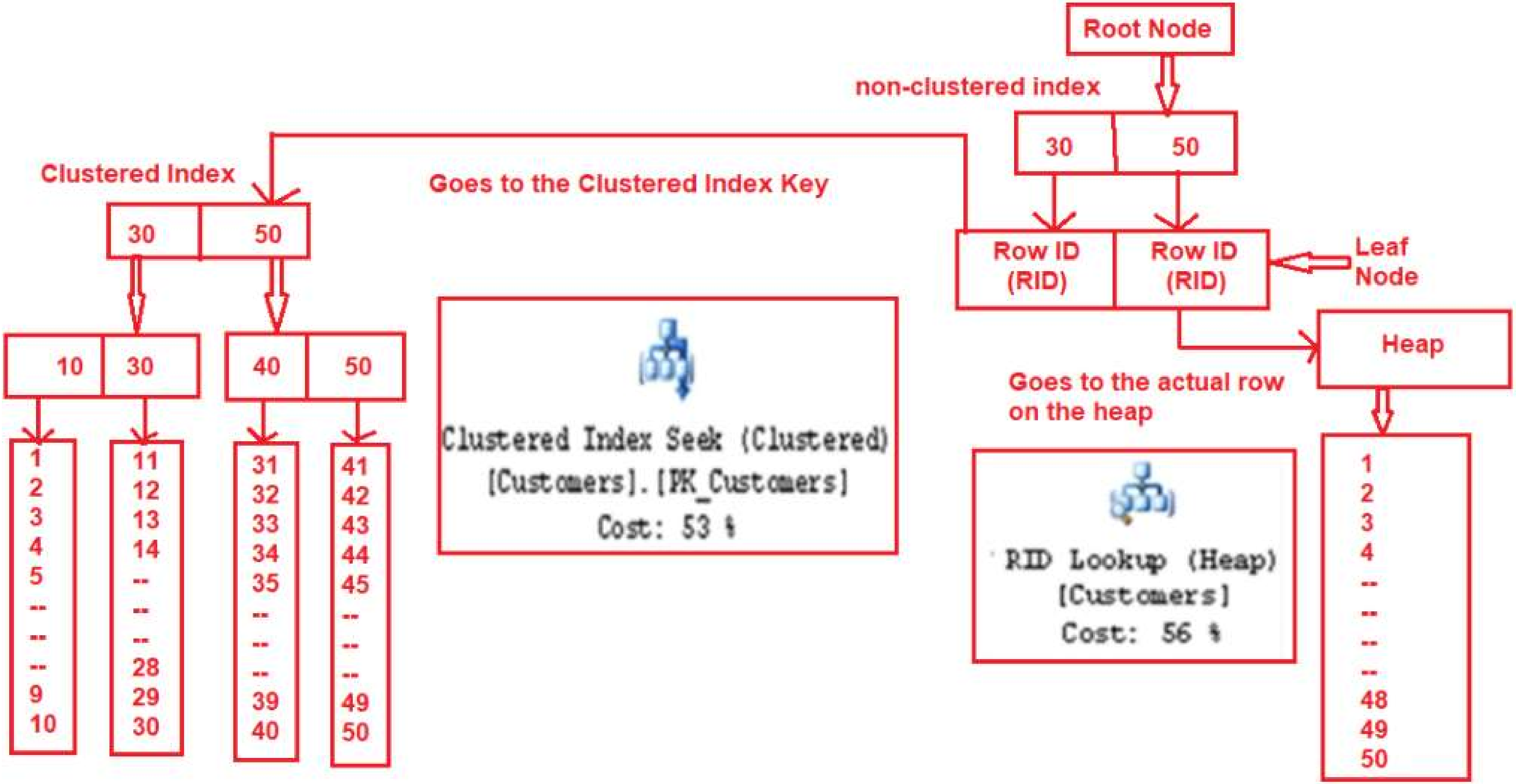
Example

SELECT * FROM Student WHERE name = 'Mary’

- Without index: Scan all Student tuples
- With index: Go "directly" to tuples with name='Mary'

Indexes are built on single attributes or combinations of attributes.

Type of indexes



Clustered Index

30 | 50

10 | 30

40 | 50

1 2 3 4 5 -- -- -- 9 10

11 12 13 14 -- -- -- 28 29 30

31 32 33 34 35 -- -- -- 39 40

41 42 43 44 45 -- -- -- 49 50

Goes to the Clustered Index Key

Clustered Index Seek (Clustered)
[Customers].[PK_Customers]
Cost: 53 %

Root Node

non-clustered index

30 | 50

Row ID (RID) | Row ID (RID)

Leaf Node

Heap

Goes to the actual row on the heap

1 2 3 4 -- -- -- -- 48 49 50

RID Lookup (Heap)
[Customers]
Cost: 56 %

Type of indexes

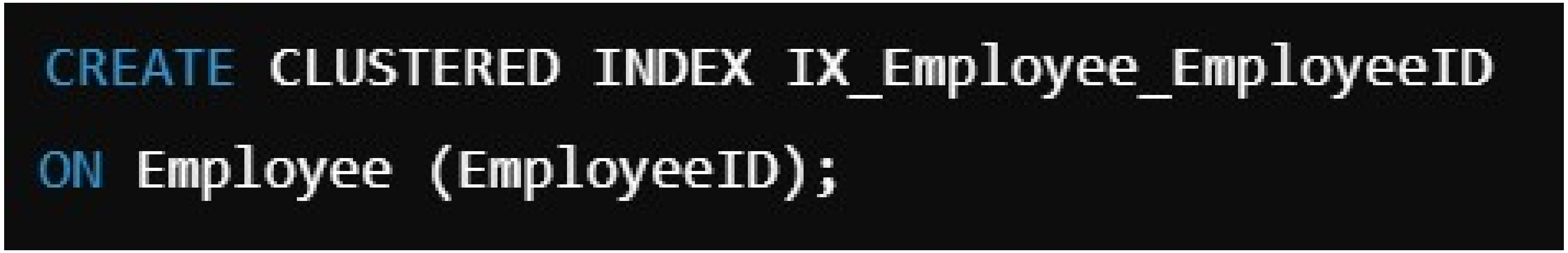
1) Clustered Index:

□ A clustered index **sorts** and **stores** the data rows of the table or view based on the index key.

□ There can only be one clustered index per table, because the data rows themselves are stored in order.

□ The clustered index determines the physical order of the data in the table.

□ For example, if a table has a clustered index on a column like EmployeeID, the rows will be physically sorted in the order of EmployeeID.

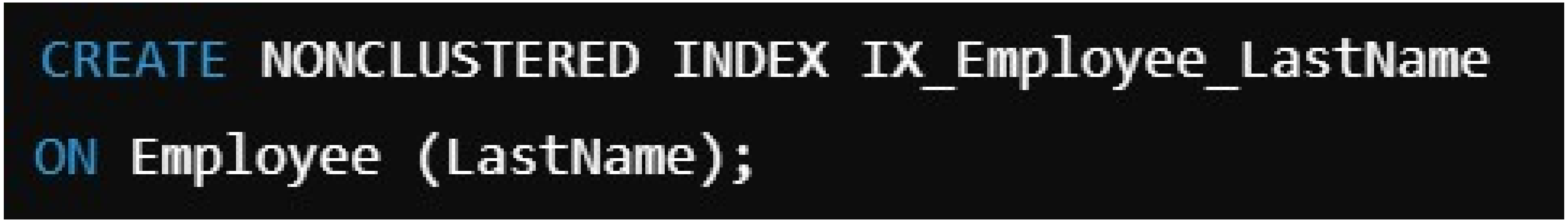


```
CREATE CLUSTERED INDEX IX_Employee_EmployeeID
ON Employee (EmployeeID);
```

Type of indexes

2) Non-Clustered Index:

- □ Contains a sorted copy of the indexed columns and pointers to the actual rows in the table.
- □ The data is not physically ordered by this index, but the index provides a fast way to find rows.
- □ A table can have multiple non-clustered indexes.



```
CREATE NONCLUSTERED INDEX IX_Employee_LastName
ON Employee (LastName);
```

Indexes implementation on SQL

```
CREATE CLUSTERED INDEX index_name ON
dbo.Tablename(ColumnName1,ColumnName2...)
```

```
CREATE NONCLUSTERED INDEX index_name
ON dbo.Tablename(ColumnName1,
ColumnName2...)
```

```
DROP INDEX index_name
```

//demo required

Index design guidelines

- Choosing which indexes to create is a difficult and very important design issue. The decision depends on size of tables, data distributions, and most importantly query/update load.
  - Table Size: enough large. Why?
  - Column Types:
    - Small size (INT, BIGINT). Should create clustered index on Unique and NOT NULL field.
    - Identity field (automatically increment)
    - Static field
  - Number of indexes
  - Storage Location of Indexes
  - Index Types
  - Query design

3. VIEWS

Relations

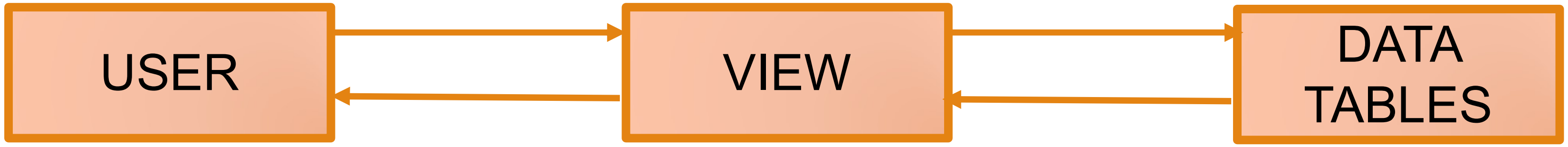
- **Actual exist** in database **in some physical** organization
- Defined with a **CREATE TABLE** statement
- Exist indefinitely and not to change unless explicit request

Views

- **Do not exist physically**
- Defined by an expression like a query
- **Can be queried** and can even be modified

Views

- A view just a relation, but we store a definition rather than a set of tuples



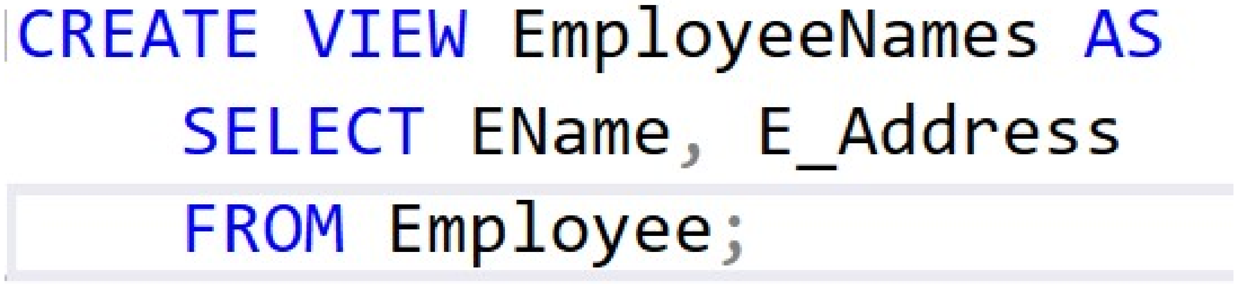
- Syntax:

```
CREATE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

Views

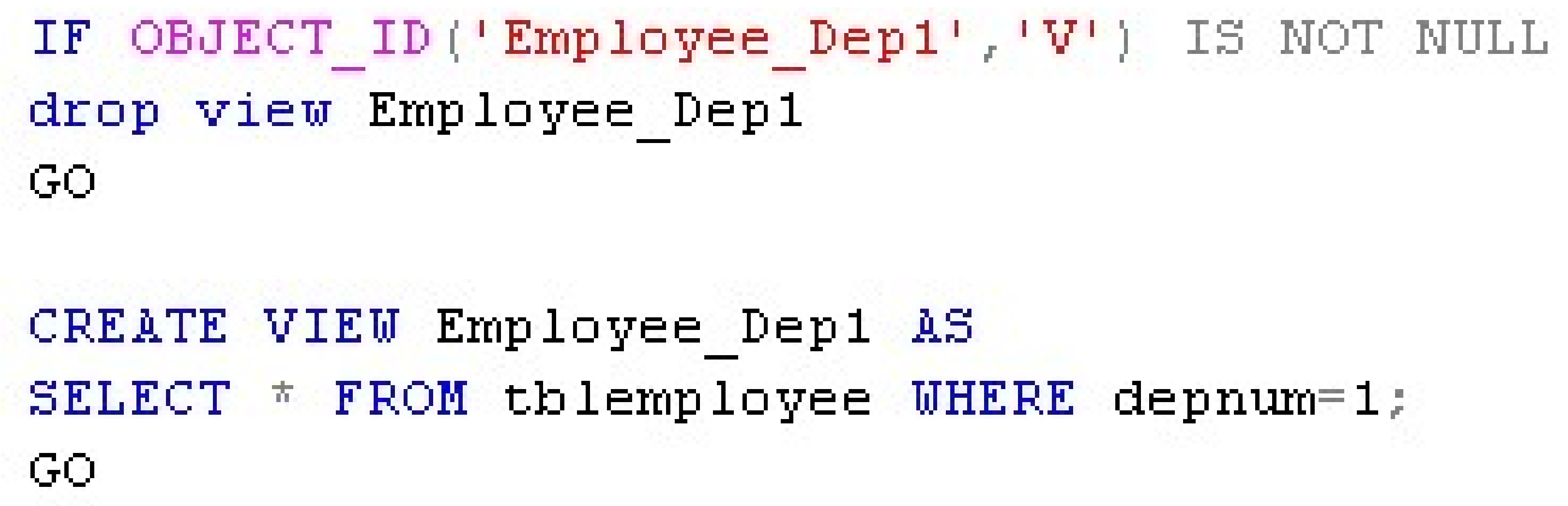
Types of views in SQL Server:

- Simple view or Updatable views: single table, can INSERT, UPDATE, DELETE through view.
- Complex view or non-updatable views: multi tables



```
CREATE VIEW EmployeeNames AS
    SELECT EName, E_Address
    FROM Employee;
```

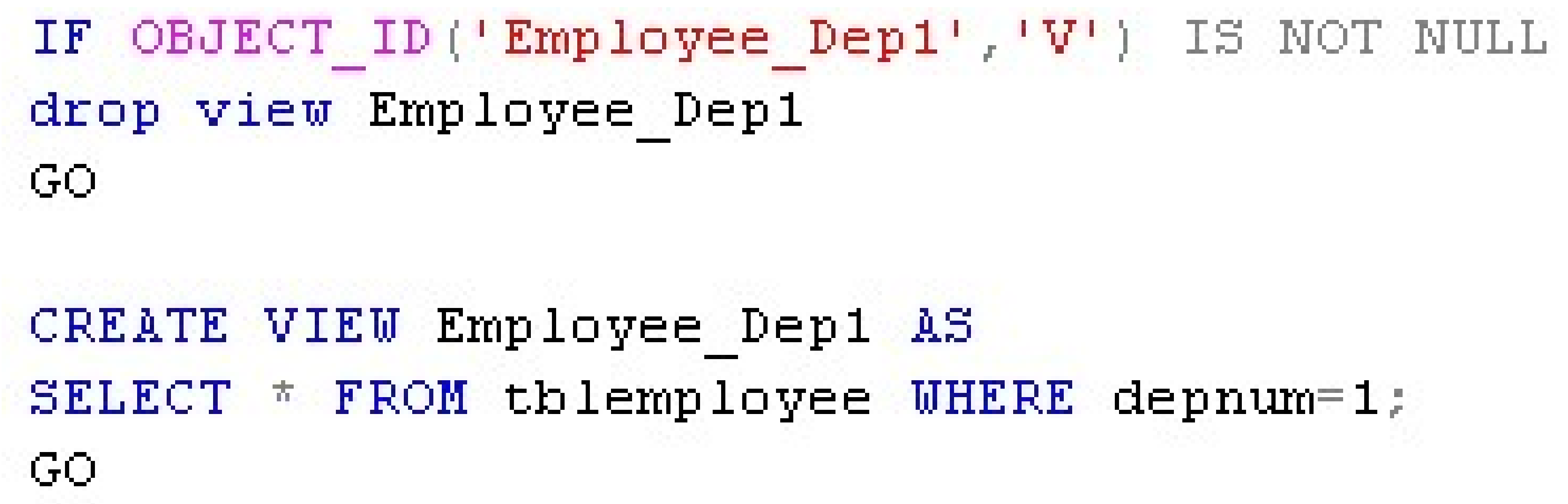
Example 1



```
IF OBJECT_ID('Employee_Dep1','V') IS NOT NULL
drop view Employee_Dep1
GO

CREATE VIEW Employee_Dep1 AS
SELECT * FROM tblemployee WHERE depnum=1;
GO
```

Example 2



```
IF OBJECT_ID('Employee_Dep1','V') IS NOT NULL
drop view Employee_Dep1
GO

CREATE VIEW Employee_Dep1 AS
SELECT * FROM tblemployee WHERE depnum=1;
GO
```

QUERYING VIEWS

Find all dependents of the employees who work in department 1

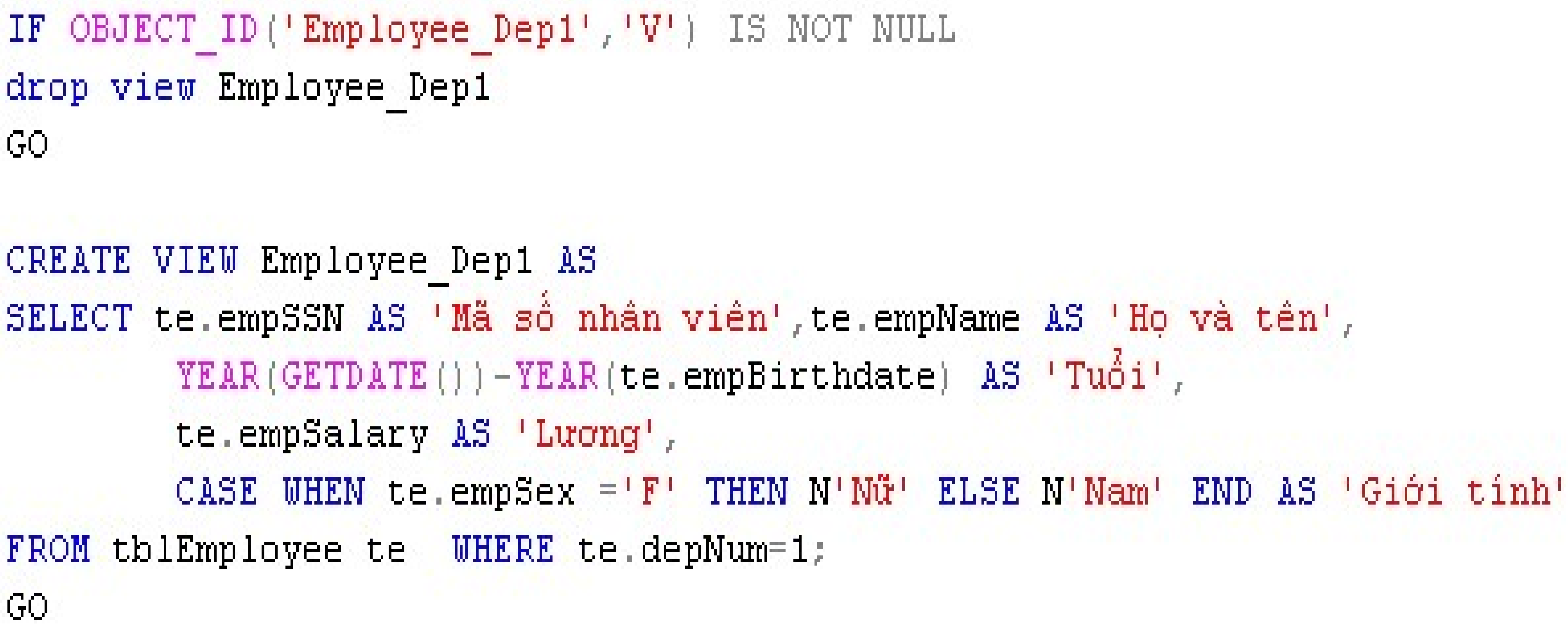
```
SELECT *
FROM Employee_Dep1 ed1, tblDependent d
WHERE ed1.empSSN=d.empSSN


SELECT *
FROM  (SELECT *
       FROM tblEmployee
       WHERE depNum=1
      ) ed1, tblDependent d
WHERE ed1.empSSN=d.empSSN
```

RENAMING ATTRIBUTES

Example 3:

- Create view for all employees of Department number 1, including: SSN, Fullname, Age, Salary, Sex



```
IF OBJECT_ID('Employee_Dep1','V') IS NOT NULL
drop view Employee_Dep1
GO

CREATE VIEW Employee_Dep1 AS
SELECT te.empSSN AS 'Mã số nhân viên',te.empName AS 'Họ và tên',
        YEAR(GETDATE())-YEAR(te.empBirthdate) AS 'Tuổi',
        te.empSalary AS 'Lương',
        CASE WHEN te.empSex ='F' THEN N'Nữ' ELSE N'Nam' END AS 'Giới tính'
FROM tblEmployee te  WHERE te.depNum=1;
GO
```

MODIFYING VIEWS

With *updatable views*, the modification is translated into an equivalent modification on a base table

The modification can be done to the base table

VIEW REMOVAL

As we know, Employee_Dep1 is associated to tplEmployee relation

DROP VIEW Employee_Dep1;

- Delete the definition of the view
- Does not effect on tplEmployee relation

DROP TABLE tplEmployee;

- Delete tplEmployee relation
- Make the view Employee_Dep1 unusable

UPDATABLE VIEWS

We can modify views that are defined by selecting some attributes from one relation R, and

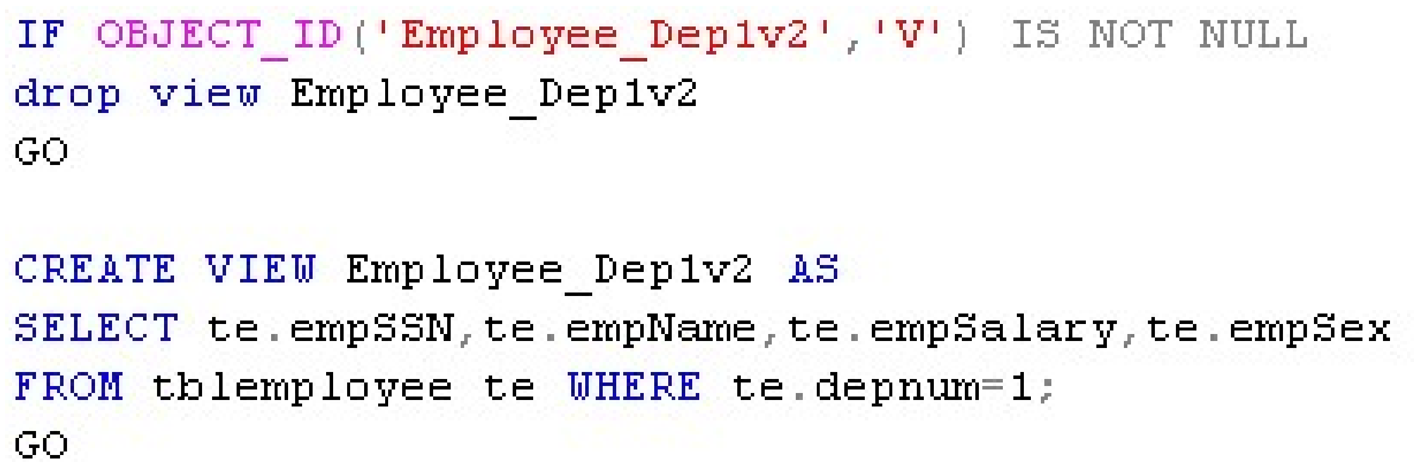
- Sub query started by SELECT, not SELECT DISTINCT
- The WHERE clause must not involve R in a sub query
- The FROM clause can only consist of one occurrence of R and no other relation
- The list in the SELECT clause must include enough attributes that for every tuple inserted into the view, we can fill the other attributes out with NULL values or the proper default

Updatable Views

Example 4:

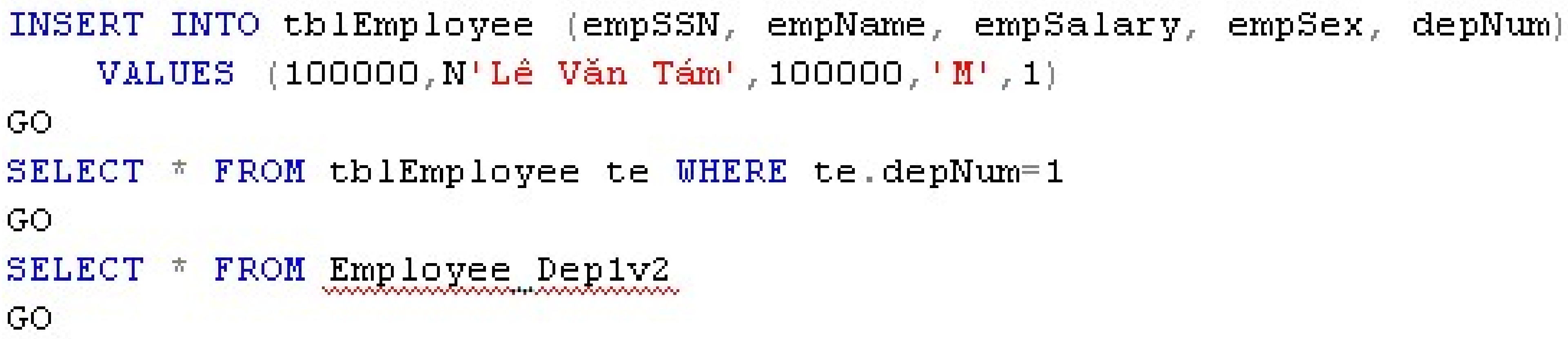
- Create view from table Employee
- Do changes on Employee and review created view
- Do changes on created view and review Employee

UPDATE ON TABLE EFFECTS ON VIEW



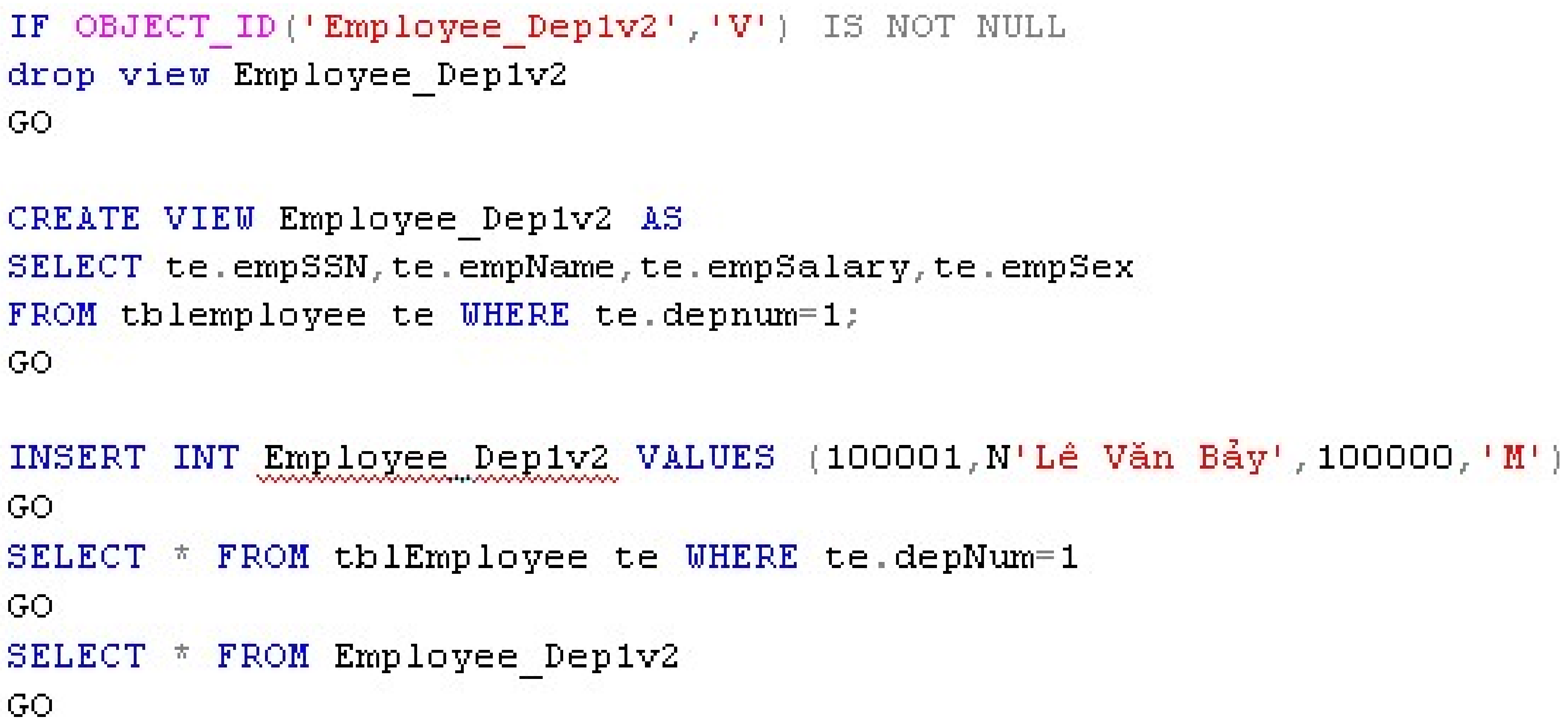
```
IF OBJECT_ID('Employee_Dep1v2','V') IS NOT NULL
drop view Employee_Dep1v2
GO

CREATE VIEW Employee_Dep1v2 AS
SELECT te.empSSN,te.empName,te.empSalary,te.empSex
FROM tblemployee te WHERE te.depnum=1;
GO
```



```
INSERT INTO tblEmployee (empSSN, empName, empSalary, empSex, depNum)
    VALUES (100000,N'Lê Văn Tám',100000,'M',1)
GO
SELECT * FROM tblEmployee te WHERE te.depNum=1
GO
SELECT * FROM Employee_Dep1v2
GO
```

Update on view effects on table with unexpected result

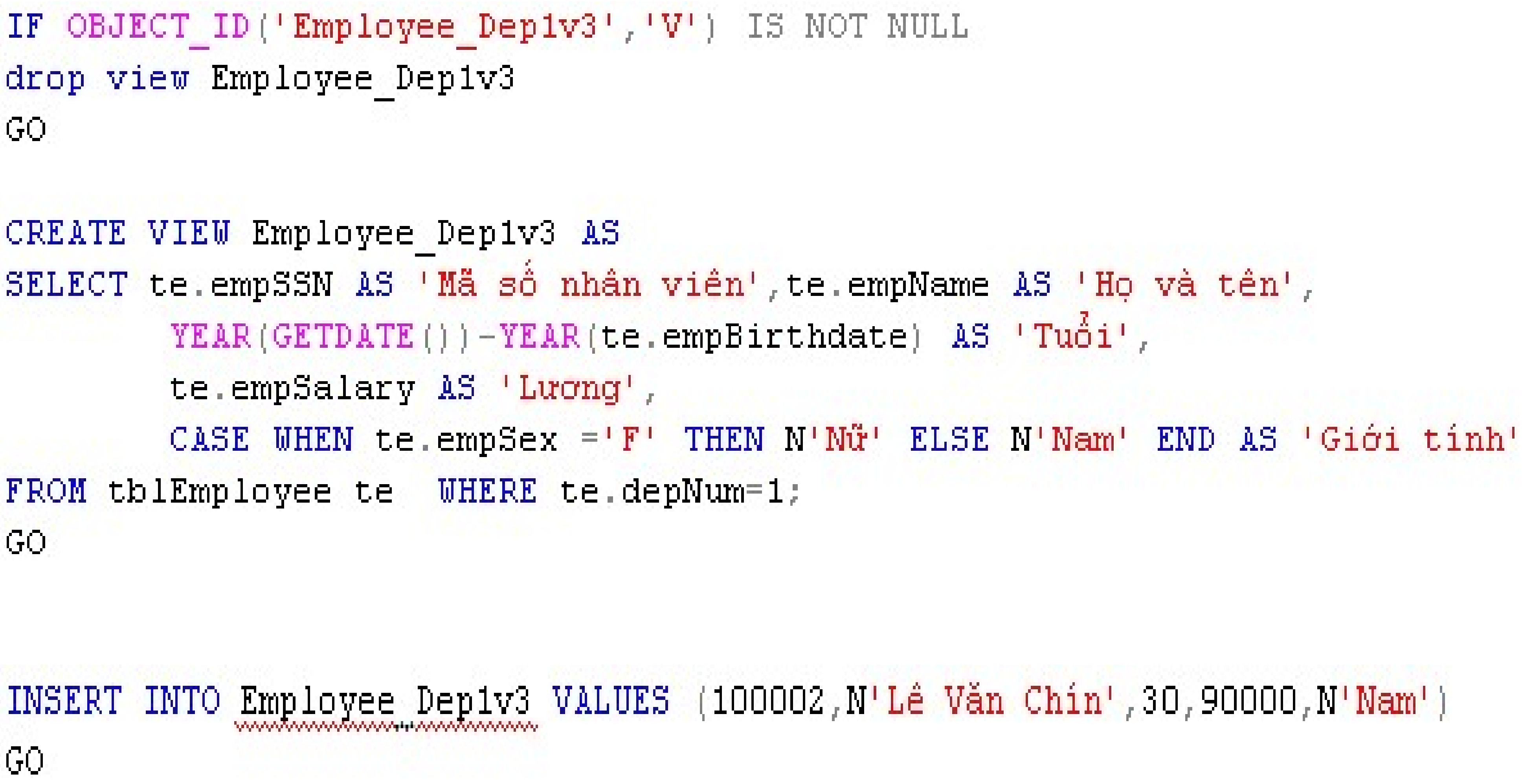


```
IF OBJECT_ID('Employee_Dep1v2','V') IS NOT NULL
drop view Employee_Dep1v2
GO

CREATE VIEW Employee_Dep1v2 AS
SELECT te.empSSN,te.empName,te.empSalary,te.empSex
FROM tblemployee te WHERE te.depnum=1;
GO

INSERT INT Employee_Dep1v2 VALUES (100001,N'Lê Văn Bảy',100000,'M')
GO
SELECT * FROM tblEmployee te WHERE te.depNum=1
GO
SELECT * FROM Employee_Dep1v2
GO
```

Update on view raises error on table

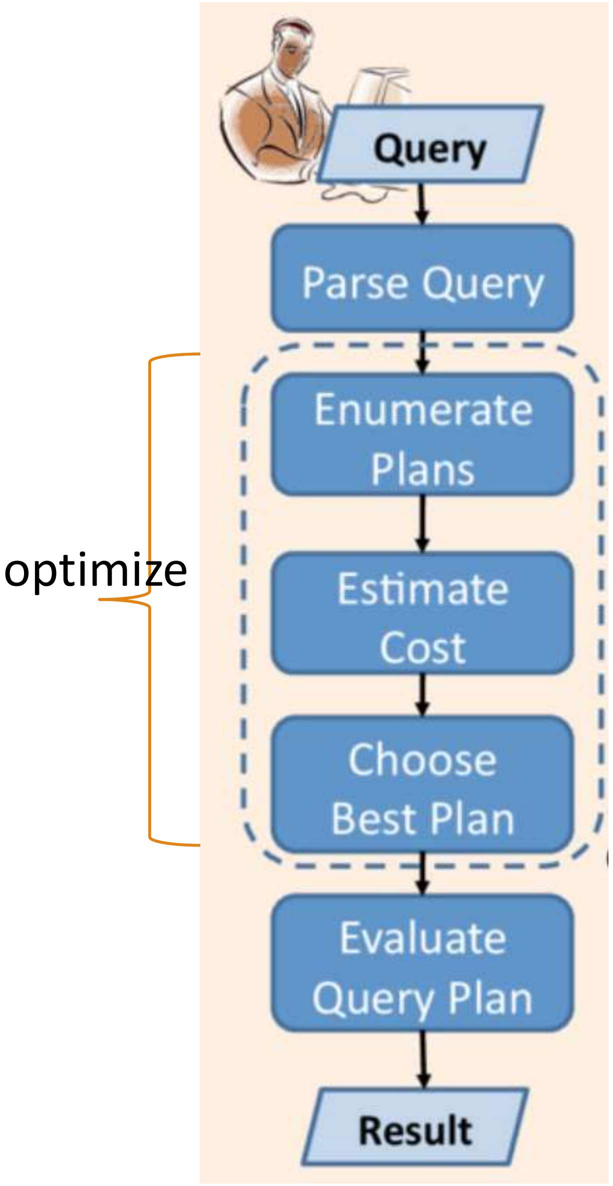


```
IF OBJECT_ID('Employee_Dep1v3','V') IS NOT NULL
drop view Employee_Dep1v3
GO

CREATE VIEW Employee_Dep1v3 AS
SELECT te.empSSN AS 'Mã số nhân viên',te.empName AS 'Họ và tên',
        YEAR(GETDATE())-YEAR(te.empBirthdate) AS 'Tuổi',
        te.empSalary AS 'Lương',
        CASE WHEN te.empSex ='F' THEN N'Nữ' ELSE N'Nam' END AS 'Giới tính'
FROM tblEmployee te  WHERE te.depNum=1;
GO


INSERT INTO Employee_Dep1v3 VALUES (100002,N'Lê Văn Chín',30,90000,N'Nam')
GO
```

Query optimization



Query

Parse Query

Enumerate Plans

Estimate Cost

Choose Best Plan

Evaluate Query Plan

Result

optimize

- In practice:

1. Define the requirements: Who? What? Where? When? Why?
2. SELECT fields instead of using SELECT *
3. Avoid SELECT DISTINCT
4. Indexing
5. Create joins with INNER JOIN (not WHERE)
6. To check the existence of records, use EXISTS() rather than COUNT()
7. Ignore linked subqueries
8. Use of temp table
9. Don’t run queries in a loop
10. Limit your working data set size