Study Unit 4
Persistent Stored Modules and Concurrency Control

Learning Outcomes

By the end of this unit, you should be able to:

1. Construct SQL/PSM statements to perform selection and loop with variables and cursors
2. Create table type for SQL/PSM modules to process table-valued parameters
3. Implement business rules via stored procedures and triggers
4. Explain and apply concurrency controls
5. Demonstrate the basic techniques of concurrency control and ACID

Overview

In this study unit, we look at how procedural code can be added to database so that business policies and procedures can be implemented. We also look at concurrency control so that one user work does not interfere with another in an unintended manner.

We start with the syntax of T-SQL, an implementation of SQL/PSM for procedural code. Next, we illustrate how T-SQL is used to implement three types of persistent stored modules: functions, stored procedures and triggers.

A user may issue individual SQL statements and invoke persistent stored modules. It is important for the DBMS to know which statements constitute a complete piece of work or a logical unit of work. A logical unit of work is also called a transaction. A transaction has ACID properties – atomic, consistent, isolated and durable. SQL/TCL or transaction control language is used to demarcate the boundary of a transaction.

When users access a database concurrently, the DBMS must isolate one transaction from another through a locking mechanism. We look at the types of locking mechanism as well as deadlock, a result of locking. We finally discuss the various isolation levels, the problems they solve and the implication on database throughput.

This study unit covers chapters 7 and 9 of the course text. It is estimated that the student will spend about 12 hours to read the course text chapters 7 and 9 in conjunction with the study notes, to work out the activities, and self-assessment questions in the study notes included at suitable junctures to test understanding of the contents covered. It is advisable to use the study notes to guide the reading of the chapter in the textbook, attempt the questions, and then check the text and/or other sources for the accuracy and completeness of your answers.

Chapter 1 SQL/Persistent Stored Modules

SQL/Persistent Stored Modules (SQL/PSM) is an extension of SQL. It adds standards procedural programming constructs such as control flow and variables, as well as constructs peculiar to the needs of database processing, such as cursors to access rows in resultset.

SQL/PSM gives us the flexibility to implement business procedures and constraints within the SQL framework so that we can use the facilities provided the DBMS. One such facility is concurrency control management, discussed in chapter 2 of this study unit.

1.1 SQL/PSM Programming Basic Syntax

We will use Transact-SQL (T-SQL) which is proprietary to Microsoft. T-SQL is the SQL/PSM extension for SQL Server. Another version of SQL/PSM is Procedural Language-SQL (PL-SQL) which is proprietary to Oracle Corporation. PL-SQL is best suited for Oracle databases.

Like SQL, T-SQL is not case-sensitive.

1.1.1 Variables

A variable stores a value of a specific data type.

- Declaring n variables

  Format:

  $\texttt{declare @varName}_1\ \texttt{datatype}_1\texttt{, …, @varName}_n\ \texttt{datatype}_n$

  The keyword `declare` starts a variable declaration.

  Example:

To declare two variables `@name` and `@qty` where `@name` is alphanumeric and `@qty` is a whole number, use the statement:

```
declare @name varchar(20),  @qty int
```

- Assigning a value to a variable
    - At declaration

      declare $@varName_1$ $datatype_1$ = $value_1$, …, $@varName_n$ $datatype_n$ = $value_n$

      Example:

      To assign the value `5B Pencil` to the variable `@name`, and the value `3` to `@qty`, use the statement:

      ```
      declare @name nvarchar(30) = '5B Pencil',  @qty int = 3
      ```

    - Through a `set` statement

      set *@varName* = *expression*

      Example:

      To assign the value `5B Pencil` to the variable `@name`, use the statement:

      ```
      set @name = '5B Pencil'
      ```

    - Through a `select` statement

      select $@varName_1$ = $expression_1$, …, $@varName_n$ = $expression_n$

      or

      select $@varName_1$ = $columnName_1$,…, $@varName_n$ = $columnName_n$ from *tableName* where *condition*

      Examples:

        - To assign the value `5B Pencil` to the variable `@name`, and the value `3` to the variable `@qty`, use the statement:

```
select @name = '5B Pencil', @qty = 3
```

- To assign the value in the column `name` to the variable `@name`, use the statement:

  ```
  select @name = name from Product where productId = 1
  ```

  Note that if the `select` statement returns more than one row, the variable to record the value of the last row in the resultset. Thus, if the `select` statement returns more one row, use a cursor to assign the variable to one value at a time, so that each value can be processed.

  Cursor is covered in Section 1.1.2.

- Accessing the content/value in a variable by the variable name
  - In an expression

    @varName operator expression

    Examples:

    - ```
      select @qty*unitPrice as subCost
      from Product where productId = 1
      ```
    - ```
      set @subCost = @qty * @unitPrice
      ```

  - In a `select/update/delete` statement.

    Examples:

    - To pick the unit cost for a product with the name found in the variable `@name`, use the statement:

      ```
      select unitPrice from Product where name = @name
      ```

    - To update the column name in the Product table to the value in the variable `@name`, use the statement:

      ```
      update Product set name = @name where productId = 1
      ```

    - To update the price to 2 for a product with the name that is the value in the variable `@name`, use the statement:

```
update Product set unitPrice = 2 where name = @name
```

- To delete the product with the name that is the value in the variable `@name`, use the statement:

```
delete from Product where name = @name
```

Activity 1

Run the statements to create and populate the table **`Product`** and then attempt parts a) to e).

```
create table Product(
  productId int not null,
  name varchar(20) not null,
  unitPrice numeric(5, 2) not null,
  quantityAvailable int not null,
  constraint ProductPK primary key(productId),
  constraint unitPriceCheck check(unitPrice < 500),
  constraint quantityAvailableCheck check(quantityAvailable >= 0)
)
insert into Product values
(1, '2B Pencil', 0.35, 2),
(2, '3B Pencil', 0.55, 10),
(3, 'Calculator', 27.85, 0)
```

a) Define two variables for `@productId` for the column `productId` and `@unitPrice` for the column `unitPrice`.

b) Assign these values 2 and 0.50 to the variables `@productId` and `unitPrice` respectively

c) Locate the row in the Product with `productId` the same as the value in `@productId`. Update the column `unitPrice` of that row to the value in `@unitPrice`.

d) Delete the row with the same `productId` as the value in `@productId`.

e) Insert a row with the value in `@productId` for `productId`, mechanical pencil for `name`, the sum of the value in `@unitPrice` and the value 2.35 for `unitPrice`, and the value 10 for `quantityAvailable`.

1.1.2 Cursors

A cursor is a database object. It is used to point to the rows in a resultset of a `select` statement, one row at a time. This allows each row to be individually processed.

- Declaring a Cursor

  ```
  declare cursorName cursor optionalCursorCharacteristics for
  select statement
  ```

  The `declare` statement assigns a cursor to operate on the resultset of the `select` statement.

  Some typical optional cursor characteristics include the following cursor types:

  - Forward only

    The declared cursor can move forward only through the resultset

  - Static

    The resultset contains rows for the `select` statement at the time the cursor is opened. The cursor can access only those rows.

  - Keyset

    At the time the cursor is opened, the key values are saved. The cursor can access only rows with the matching key values.

  - Dynamic

    The cursor can access all rows in the resultset, including updated and newly inserted rows.

  Cursor characteristics impact database throughput, as other users may be prevented from accessing the database records that a cursor is accessing. For example, a forward cursor releases the lock on the rows it has read, thus allowing other users to access them. We cover concurrency control in Chapter 2 of this study unit.

  If no cursor characteristic is specified, DBMS will assign default cursor characteristics.

Read

Kroenke, D and D. Auer. (2016). *Database Processing: Fundamentals, Design and Implementation Edition 14*. Pearson, 470-472.

Example:

To declare a cursor, `productCursor` for products costing less than $10, with cursor characteristics `forward only`, use the statement:

```
declare productCursor cursor forward only for
   select productId, name, unitPrice
   from Product where unitPrice < 10
```

The cursor is declared for the resultset of the `select` statement which returns products from the `Product` table that cost less than $10. Each row has these columns: product id, product name and product unit price.

- Opening a Cursor

  `open` *`cursorName`*

  Opening a cursor causes the `select` statement it is declared for to execute. The resultset of the `select` statement becomes available, and the cursor typically points to first row.

  Example:

  ```
  open productCursor
  ```

  This statement causes the `select` statement to execute:

  ```
  select productId, name, unitPrice from Product where unitPrice
  < 10
  ```

  The cursor points to the first row in the resultset.

- Fetching rows from a Cursor

  `fetch next from` *`cursorName`* `into @varName`$_1$`, …, @varName`$_n$

A cursor points to one row in the resultset. The row can be fetched into variables. If the `fetch` is successful, the fetch status, accessed through the function `@@fetch_status` returns 0.

After a fetch, the cursor moves to the next row. If the current fetch is for the last row of the resultset, then the fetch status for the next fetch will not be 0 as the next fetch will fail to get a row.

Example:

```
fetch next from productCursor into @productId, @name, @unitPrice
```

Besides `fetch next`, other possibilities include `fetch prior`, `fetch first` and `fetch last`.

- Closing a Cursor

  ```
  close cursorName
  ```

  Closing a cursor releases the resultset and locks that prevent other transactions from accessing the rows in the resultset. Locks and transactions are covered in chapter 2 of this study unit.

  After a cursor is closed, it cannot fetch any more rows, unless it is re-opened.

- Deallocating a Cursor

  ```
  deallocate cursorName
  ```

  Deallocating a cursor releases all resources that the cursor uses, including the cursor name. After the cursor is deallocate, the cursor name becomes undefined.

1.1.3 Table Type

A table type is a user data type. Once a table type is defined, it can be used as a data type for variable declaration.

- Format for defining a table type:

  ```
  create type tableType as table
  ```

```
(columnName1  datatype1,
…
columnNamen  datatypen
optionalPrimarykeyDefinition
)
```

Example:

To define the table type `productType` with columns `productId` and `qty`, use the statement:

```
create type productType as table (
   productId  int,
   qty int,
   primary key (productId )
)
```

- Declaring a variable with a table type

  Format:

  ```
  declare @tableVariable tableType
  ```

  Example:

  To declare a variable, `@products` of datatype `productType`, use the statement:

  ```
  declare @products productType
  ```

  Rows can be inserted into a table-typed variable, using the SQL `insert` statement.

  Example:

  We can operate on the variable, `@products` as if it is a table.

  - Inserting rows

    ```
    insert into @products values(1, 3), (2, 5)
    ```

  - Accessing rows through a `select` statement

    ```
    select * from @products
    ```

  - Declaring a cursor for its rows

    ```
    declare prodCursor cursor for
        select productId, qty from @products
    ```

We can declare table-valued parameters for functions and procedures. Table-typed parameters can receive rows of data, as illustrated in the example in Figure 4.2, in Section 1.3.

1.1.4 Control Flow

SQL/PSM provides control flow structures which are identical in behaviour as those in procedural programming languages.

- Conditional statement

  Format:

  ```
  if  condition_if
  begin
    statements_if
  end
  else if condition_elseif
  begin
    statements_elseif
  end
  else
  begin
    statements_else
  end
  ```

  The `else if` and `else` blocks are optional. The `begin` and `end` markers are necessary only when a statement block contains more than one statement. Otherwise, they are optional.

  Example:

  ```
  if  @rowcount  > 1
    print 'There are ' + cast(@rowcount as varchar(2))
          + ' products'
  else if @rowcount  = 1
    print 'There is 1 product'
  else
    print 'There is no product'
  ```

- Loop statement

  Format:

  ```
  while condition_while
  begin
    statement_while
  ```

```
end
```

Example:

```
declare productCursor Cursor for
   select productId, name, unitPrice, quantityAvailable
   from Product

open productCursor

fetch next from productCursor
   into @productId, @name, @unitPrice, @quantityAvailable
while @@fetch_Status = 0
begin
   if @quantityAvailable < @reorderPoint
      insert into SupplierOrderItem values (@productId, @name,
           @unitPrice, @reorderQuantity - @quantityAvailable)
   fetch next from productCursor
       into @productId, @name, @unitPrice, @quantityAvailable
end

close productCursor
deallocate productCursor
```

Statements such as `break` and `continue` can be used within a loop. The `break` statement causes the execution to leave the loop, and the `continue` statement causes the execution to abandon the current iteration.

- Raising exception

  Format:

  ```
  raiserror (message, severity, state)
  ```

  Example:

  ```
  raiserror (N'An input error', 11, 1)
  ```

  An exception can be raised using the `raiserror` statement.

  Severity levels from 0 to 10 are informational messages. Severity levels from 11 to 16 are user errors, that is, errors that user code can raise.

  Severity levels from 17 to 19 are non-fatal DBMS errors, and levels from 20 to 25 are fatal DBMS errors. Users should not raise exceptions with severity levels 17 to 25.

The state of the exception is often used to help the exception handling code differentiate the sources of the exception. Valid values for state are between 0 and 255.

- Handling exception

  Format:

```
begin try
   statements_try
end try
begin catch
   statements_catch
end catch
```

  Example:

```
begin try
   insert into Product values('P0001', 'pencil', 3),
end try
begin catch
   print 'The insert statement is not successful'
end catch
```

Read

Kroenke, D and D. Auer. (2016). *Database Processing: Fundamentals, Design and Implementation Edition 14*. Pearson, 371-372.

Activity 2

Assume the rows in the Product table in Activity 1:

productId	name	unitPrice	quantityAvailable
1	2B Pencil	0.35	2
2	3B Pencil	0.50	10
3	Calculator	27.85	0

Write statements to determine the number of products in these price ranges: under $5, between $5 and under $10 and $10 and above. Print the number of products in each price range.

A sample output is given here:

```
less than $5         - 2
$5 but less than $10 - 0
$10 or more          - 1
```

Note: The use of cursor is optional for this activity.

Activity 3

Improve the solution in Activity 2 by printing the product ids of products in each price category as well, as shown here:

```
less than $5         - 2
Products are 1, 2
$5 but less than $10 - 0
$10 or more          - 1
Product is 3
```

Hint: You should declare a table type so that you can declare variables to record the products in each price category.

1.2 Functions

A function can be written for some task which returns a result. Once a function is defined, it can be repeatedly called to perform the task it is defined for.

- Function definition:

```
create function functionName (
  @parameter1 datatype1 , …, @parametern datatypen,
  @parametern+m+1 tabletypen+m+1 READONLY, …,
  @parametern+m+p tabletypen+m+p READONLY)

  returns datatype or table or tableVar tableType
as
begin
  statements
  return value or return select statement or return
end
```

Note that the input parameters for the function are within the parenthesis after the function name. An input parameter can be a table-valued parameter. We will see an example of table-valued parameter in section 1.3 Stored Procedure.

The output parameter of a function can be scalar (single value), as in the example shown in Figure 4.1, or a table.

The function in Figure 4.1, `NameConcatenation` accepts a first name and a last name as parameters. It returns the full name which is a string, resulting from concatenating the first name and the last name.

```
CREATE FUNCTION dbo.NameConcatenation

-- These are the input parameters
       (
       @FirstName    CHAR(25),
       @LastName     CHAR(25)
       )
RETURNS VARCHAR(60)
AS
BEGIN
       -- This is the variable that will hold the value to be returned
       DECLARE @FullName VARCHAR(60);

       -- SQL statements to concatenate the names in the proper order
       SELECT @FullName = RTRIM(@LastName) + ', ' + RTRIM(@FirstName);

       -- Return the concatentate name
       RETURN @FullName;
END;
```

Figure 4.1 User-defined function to Concatenate FirstName and LastName

(Source: Kroenke, D and D. Auer. (2016). Database Processing: Fundamentals, Design and Implementation Edition 14. Pearson, Figure 7.21)

- Function call:

  `functionName (value`$_1$`, …, value`$_n$`)`

  Making a function call will get the statements in the function to execute. Without a function call, the statements do not get executed.

  The function calls `NameConcatenation` to perform the task concatenating `Alice` and `Tan` to produce `Tan, Alice`:

  ```
  dbo.NameConcatenation('Alice', 'Tan')
  ```

Activity 4

a) Define a function that returns the count of products within a price category. The function has 2 parameters: the lower limit and upper limit of a price range. The function should include products with unit price the same as the lower limit as well as products with unit price less than the upper limit. Assume that the unit prices of products are under $500.

b) Define a function that returns a table with only the column productId of products within a price category. The function has 2 parameters: the lower limit and upper limit of a price range. The function should include products with unit price the same as the lower limit as well as products with unit price less than the upper limit. Assume that the unit prices of products are under $500.

Write test scripts to test the functions.

Read

Kroenke, D and D. Auer. (2016). *Database Processing: Fundamentals, Design and Implementation Edition 14*. Pearson, 373-376.

1.3 Stored Procedures

A stored procedure is like a function in that it must be called for its statements to execute. However, it is optional for a stored procedure to return values.

Unlike a function which uses the `return` statement to return a value, a stored procedure uses one or more output parameters to return values. Output parameters are indicated with the keyword `OUTPUT`, or alternatively, `OUT`.

- Stored procedure definition:

```
create procedure procedureName
  @parameter1 datatype1 , …,
  @parametern datatypen,
  @parametern+1 datatypen+1 OUTPUT, … ,
  @parametern+m datatypen+m OUTPUT,
  @parametern+m+1 tabletypen+m+1 READONLY, …,
  @parametern+m+p tabletypen+m+p READONLY
as
begin
  statements
end
```

An example of a stored procedure definition is shown in Figure 4.2. The stored procedure uses the following tables and table type which are created first:

```
create table Customer (
custid char(8) not null primary key,
custname varchar(20),
address varchar(30)
)

create table CustomerOrder(
      orderNumber int identity(1,1) primary key,
      orderDate smallDateTime,
      custid char(8) not null foreign key references Customer
)

create table OrderItem(
      orderNumber int not null foreign key references
      CustomerOrder,
      productId int not null foreign key references Product,
      qty int not null,
      price numeric (5, 2) not null
      primary key (orderNumber, productId)
)

create type OrderProductType as table (
      productId int primary key,
      qty int
```

)

The stored procedure implements this business rule: only valid customers can make an order, and an order must have at least a product ordered. In addition, if the order quantity for a product exceeds the available quantity for the product, order quantity be adjusted to the available quantity, and available quantity will be used to fulfill the order. The available quantity for products should not become negative.

The stored procedure has 3 input parameters: `@orderDate` for the date the customer order is placed, `@custId` for the customer id, and `@orderItems`, a table-valued parameter for a list of items ordered.

There are two output parameters: `@totalCost` for the total cost payable for the customer order and `@orderNumber` for the order number of the newly recorded order.

We will use a cursor to traverse through the list of items ordered. The product id and quantity of items ordered are processed one at a time in the loop.

```
create procedure makeOrder @orderDate smallDateTime, @custId char(8), @orderNumber int
   output, @totalCost numeric(6,2) output,@orderItems OrderProductType readonly as
begin
   set nocount on
   declare @productId int, @unitPrice numeric(5,2),
      @qtyAvailable int, @qtyOrdered int, @subtotal numeric(6,2)
   set @totalCost = 0; set @orderNumber = -1; set @totalCost = 0
   if not exists (select * from Customer where custid = @custId)
      print 'Error: Invalid customer id ' + cast(@custId as varchar(3))
   else
   begin
   begin transaction
      declare itemsCursor cursor for select productId, qty from @orderItems
      insert into CustomerOrder values (@orderDate, @custId)
      set @orderNumber = @@IDENTITY
      open itemsCursor
      fetch next from itemsCursor into @productId, @qtyOrdered
      while @@FETCH_STATUS = 0
      begin
         select @unitPrice = unitPrice, @qtyAvailable = quantityAvailable
         from Product where productId = @productId
         if @@ROWCOUNT = 0
            print 'Error: Product ' + cast(@productId as varchar(2)) +' is invalid'
         else
         begin
            if @qtyAvailable = 0
               print 'Error: Product ' + cast(@productId as varchar(2)) + ' is not
available'
            else
            begin
               if @qtyAvailable < @qtyOrdered
               begin
                  print 'Product ' + cast(@productId as varchar(2)) +
                     ' - Insuffient quantity! Ordered ' + cast(@qtyOrdered as varchar(2)) +
                     ' reduced to ' + cast(@qtyAvailable as varchar(2))
                  set @qtyOrdered = @qtyAvailable
               end
               update Product set quantityAvailable = @qtyAvailable - @qtyOrdered
                  where productId = @productId
               insert into OrderItem values
                  (@orderNumber, @productId, @qtyOrdered, @unitPrice)
               set @subtotal = @qtyOrdered * @unitPrice
               print 'Ordered ' +cast(@qtyOrdered as varchar(2)) + ' of Product ' +
                  cast(@productId as varchar(2)) + ' at $' + cast(@unitPrice as varchar(8))
                  + ' = $' + cast(@subtotal as varchar(9))
               set @totalCost = @totalCost + @subtotal
            end
         end
         fetch next from itemsCursor into @productId, @qtyOrdered
      end
      if exists (select * from OrderItem where orderNumber = @orderNumber)
      begin
         print 'Total = $' + cast(@totalCost as varchar(9))
         print 'Order added'
         commit
      end
      else
      begin
         print 'Order not added. No Order item'
         rollback
      end
      close itemsCursor
      deallocate itemsCursor
   end
end
```

Figure 4.2 Stored procedure to insert a customer order and the items ordered into tables CustomerOrder and OrderItem respectively

- Stored procedure call:

```
exec procedureName
  @parameter1 = value1, …,
  @parametern datatypen,
  @parametern+1 datatypen+1 OUTPUT, … ,
  @parametern+m datatypen+m OUTPUT,
  @parametern+m+1 tabletypen+m+1 READONLY, …,
  @parametern+m+p tabletypen+m+p READONLY
```

Example:

To call the stored procedure shown in Figure 4.2, we set up the input values to pass to the stored procedure, and declare variables to receive output values.

```
declare @toBuy OrderProductType
insert into @toBuy values (11, 3), (12, 2), (9, 1)
declare @orderNumber int, @tCost numeric(6,2),  @oDate
smalldatetime =  GETDATE()

exec makeOrder @orderDate = @oDate, @custId = 1,
      @orderNumber = @orderNumber, @totalCost = @tCost,
      @orderItems = @toBuy

print @totalCost
print @orderNumber

select * from CustomerOrder
select * from OrderItem
```

Execute the stored procedure with other test cases by replacing the `insert` statement with other statements such as this:

```
insert into @toBuy values (1, 4), (2, 3), (3, 1)
```

Read

Kroenke, D and D. Auer. (2016). *Database Processing: Fundamentals, Design and Implementation Edition 14*. Pearson, 382-388.

Activity 5

Convert the solution to Activity 3 using one or more stored procedures. Make the stored procedure call the functions defined in Activity 4.

Activity 6

Write a stored procedure that accepts a list of product ids as parameter.

For each product whose product id is in the list, delete the product if the column `quantityAvailable` is zero, and print a message that the delete is successful.

However, if `quantityAvailable` is not zero, print the product details and a message to indicate the `quantityAvailable` is not zero.

If a product id is not in the Product table, print a message to indicate that product id is invalid.

Write a test script to test the stored procedure.

Hint: You must use a cursor.

1.4 Triggers

A trigger is written for some task that must be performed in conjunction with an SQL DDL or SQL DML operation, e.g., when an insert is made on a customer table, a check must be made to ensure that the customer has not been blacklisted. This check is the task that must accompany the operation insert on Customer. The trigger should not allow the insert to succeed if the customer is blacklisted.

In other words, a trigger is written to perform some task for specific operation(s) such as DML operations on a table or a view. It contains statements that must be performed when the specified operation(s) are requested on the database object.

Unlike a function or stored procedure, a trigger is not called. Triggers differ from functions or stored procedures in these aspects:

- Access to pseudo-files, `inserted` and `deleted`

  While a function or stored procedure receives input values through parameters, a trigger does not receive input parameters as a trigger is not called.

  Instead, the DBMS supplies data to the trigger through the pseudo-files, called `inserted` and `deleted` in the case of SQL Server.

  When a trigger is defined for the insert and update operations, the pseudo-file, `inserted` contains new data. If the operations are delete and update, the pseudo-file, `deleted` contains old data.

  Note that the pseudo-files, `inserted` and `deleted` may contain zero or more rows of data. The number of rows in the pseudo-files depends on how many rows are affected when the user-requests for the operation.

- Types of triggers

  When the operation that trigger is defined for is requested, the trigger code gets executed. The trigger code may execute before, after or in place of the operation requested.

  - Before trigger

    The trigger executes before the requested operation is carried out.

- After trigger

  The trigger executes after the requested operation is carried out.

- Instead of trigger

  The trigger executes in place of the requested operation.

SQL Server supports only `after` and `instead of` triggers. An `after` trigger can also be defined using the keyword `for`.

- Trigger definition:

```
create trigger triggerName on tableOrViewName
after/for/instead of operations

  @parameter1 datatype1 , …, @parametern datatypen
as
begin
  statementsAccompanyingTheOperations
end
```

An example of a trigger is shown in Figure 4.3. This trigger implements a business rule that if a person is in a company's blacklist, he should not be registered as a customer. The person's id is checked against a blacklist. If the id is found in the blacklist, an error message is printed. Otherwise, the person is inserted into the Customer table.

The trigger can be defined as an `instead of` or an `after` trigger on the Customer table for the insert operation.

  - `instead of` trigger

    The trigger shown in Figure 4.3 is an `instead of` trigger which means whenever there is a request to insert into the Customer table, the trigger will execute in place of the `insert` request.

    For each customer to be inserted, the trigger first checks that he is not in the blacklist. If so, the trigger issues an insert request for that customer.

    The insert statement issued in the trigger will not fire itself (the trigger). Otherwise, the execution of the trigger will go into an infinite loop.

- `after/for` trigger

  The logic for the `after/for` trigger is different as the trigger code executes after the insert operation on the Customer table has been carried out. This means that some newly inserted customers could have previously been blacklisted.

  Thus, when the trigger executes, it checks whether any of the inserted customer have been blacklisted previously. If so, the trigger rollbacks, and it causes the insert operation to undo.

  The rollback will undo the insert of all newly inserted Customer rows, even non-blacklisted customers.

  Alternatively, instead of performing a rollback, the trigger can issue delete request to remove blacklisted customers.

The trigger `checkBlackListCustomer` assumes that the table `Blacklist` has been created. Therefore, create the table `Blacklist` before creating the trigger to avoid errors.

```
create table Blacklist (
custid char(8) not null primary key,
)
```

Populate the table `Blacklist.`

```
insert into BlackList values('s1234567'), ('s1234568')
```

```
create trigger checkBlackListCustomer on Customer
instead of insert
as
begin
   set nocount on
   declare @custid as char(8), @custname varchar(2), @address varchar(30)
   declare cCursor cursor for
      select custid, custname, address from inserted
   open cCursor
   fetch next from cCursor into @custid, @custname, @address
   while @@FETCH_STATUS = 0
   begin
      if exists (select * from blacklist where custid = @custid)
         print 'Error: This customer was previously blacklisted and cannot be inserted: '
            + @custid + ', ' + @custname + ', ' +  @address
      else
         begin
            insert into Customer values (@custid, @custname, @address)
            print 'Customer added: ' + @custid + ', ' + @custname + ', ' +  @address
         end
      fetch next from cCursor into @custid, @custname, @address
   end
   close cCursor
   deallocate cCursor
end
```

Figure 4.3 Trigger for insert into table Customer to check that customer is not blacklisted

- Activating a trigger

  Format:

  `Example:`

  ```
  insert into Customer values
  ('s1234567', 'Mary', '12 Tampines Road'),
  ('s1234566', 'Maria', '13 Tampines Road'),
  ('s1234569', 'Thomas', '21 Tampines Road'),
  ('s1234568', 'Tom', '25 Tampines Road')
  ```

  Executing the `insert` statement causes the trigger `checkBlackListCustomer` to be executed because that trigger has been defined for an insert operation on the table, customer. The DBMS will supply the following rows in the pseudo-file, `inserted` to the trigger.

  ```
  ('s1234567', 'Mary', '12 Tampines Road'),
  ('s1234566', 'Maria', '13 Tampines Road'),
  ('s1234569', 'Thomas', '21 Tampines Road'),
  ('s1234568', 'Tom', '25 Tampines Road')
  ```

Read

Kroenke, D and D. Auer. (2016). *Database Processing: Fundamentals, Design and Implementation Edition 14*. Pearson, 373-382.

Activity 7

Write an instead of trigger for the delete operation for the Product table. Allow the delete to happen only if `quantityAvailable` is zero.

Write a test script to test the trigger.

Activity 8

Write an after trigger for the insert operation for the Product table. Allow the insert to happen only if `quantityAvailable` is not zero. Ifany `quantityAvailable` is zero, rollback and raise an error with the message that the insert operation does not allow `quantityAvailable` to be zero.

Note that the rollback will undo all the inserted rows for this operation.

Write a test script to test the trigger and handle exception, if any.

Chapter 2 Transactions and Concurrency Control

A database can allow multiple users to access the database concurrently. In order that the work of one user does not affect another in an unintended way, a DBMS must provide concurrency control.

Read

Kroenke, D and D. Auer. (2016). *Database Processing: Fundamentals, Design and Implementation Edition 14*. Pearson, 456-460.

2.1 Transaction

A transaction is a logical unit of work. The concept of transaction is important as it allows the DBMS to determine which database actions constitute one complete user's work. If any of the database actions of a transaction fails, the DMBS must undo the database actions already performed, to ensure that the database remains in a consistent state.

An example of a transaction is a fund transfer between two bank accounts. The fund transfer consists of two update statements: one update statement reduces the balance of the bank account to transfer from, and one update statement increases the balance of the bank account to transfer to. If any one of the two update statements fails, none of the update should be carried out. That is, either both update must succeed, and if any of the updates fails, the update that succeeds must be undone.

Refer to Figure 4.4a for the case when the database actions form one complete unit of work but the DBMS considers each database action independent, and treats each action as a separate unit of work. In this example, the customer order has been added, and the salesperson's commission has been updated. If the order details fail to insert, the earlier two actions are not undone. Thus, the database has become inconsistent. The customer is billed for an order with no record of what items are

ordered, and the salesperson is paid the commission for an order that will not be delivered to a customer.

To ensure the database is consistent, the effects of earlier database actions in the same transaction must be undone. SQL TCL (Transaction Control Language) can be used to demarcate the boundaries of a transaction, to indicate to the DBMS which actions form a logical unit of work..

- `begin transaction`

  `begin transaction` marks the start of a transaction

- `commit`

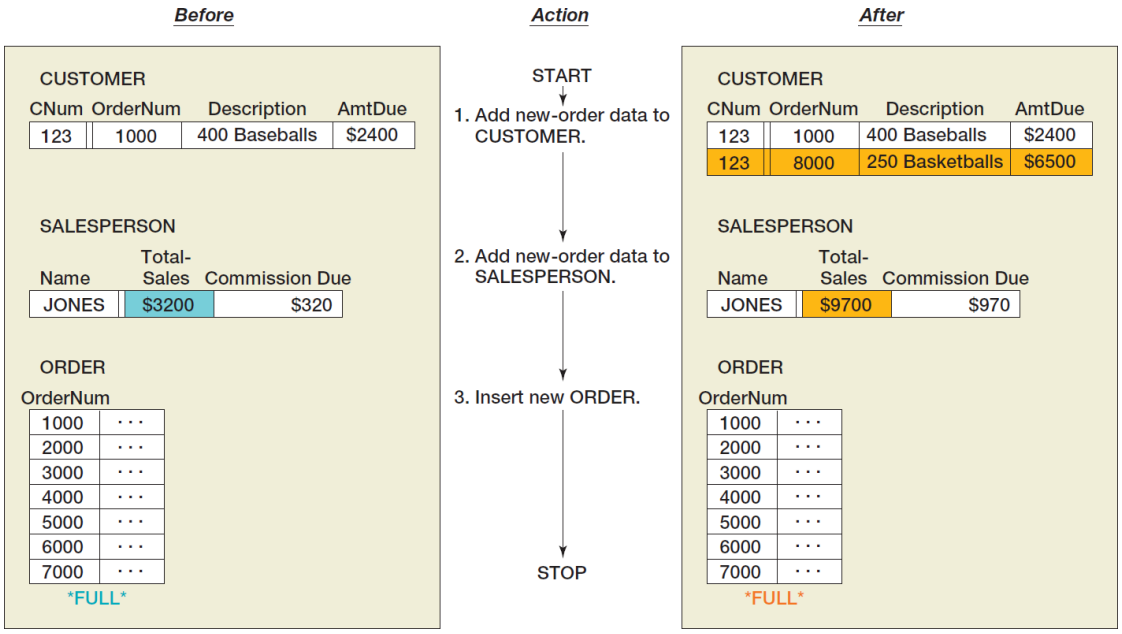
  `commit` signifies the transaction ended successfully, that is, all the database actions from the start of the transaction to `commit` are successfully carried out. Thus, all the effects of the database actions must be reflected on the database.

- `rollback`

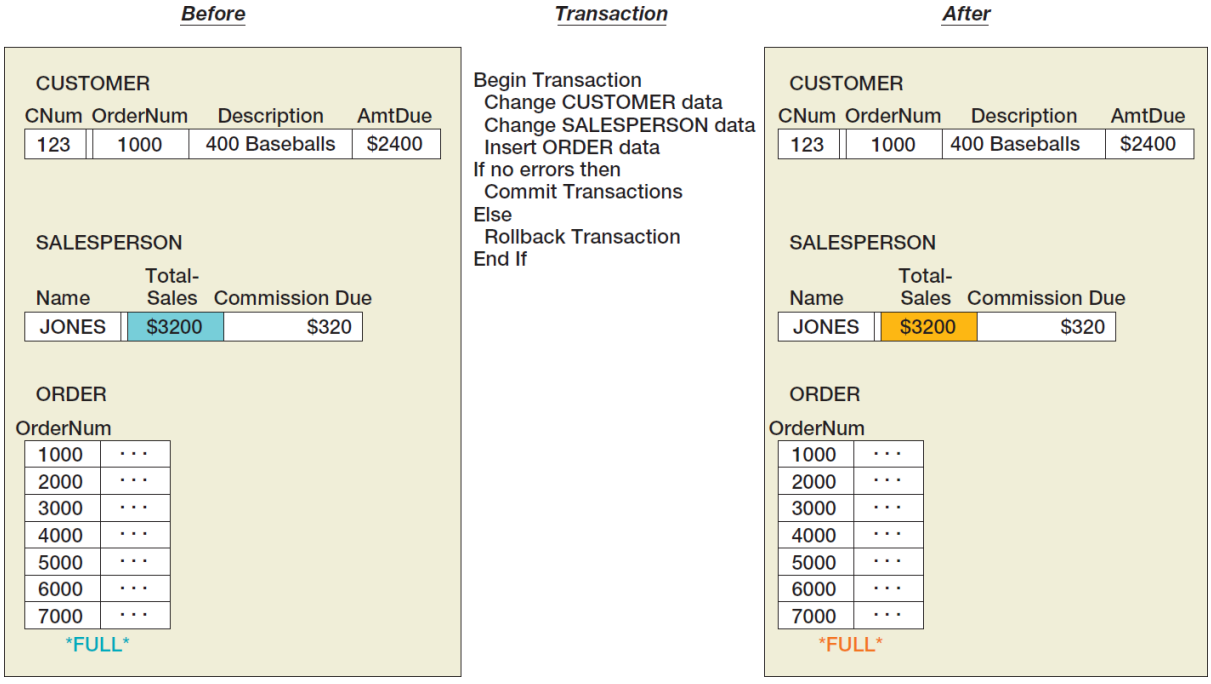
  `rollback` signifies that there is an error and so the subsequent database actions within the transaction cannot be carried out.

  All the database actions in the transaction that have been carried out must be undone to bring the database back to a consistent state.

In Figure 4.4b, database actions are placed in a single (atomic) transaction using SQL TCL `begin transaction`, and `commit` and `rollback` are used to demnarcate the end of the transaction. When one of the database actions fails, the DMBS knows where the start of the transaction is, and so, it can undo all earlier database actions within that transaction to bring the database back to a consistent state.



(a)



(b)

Figure 4.4 Transaction Processing Example

(Source: Kroenke, D and D. Auer. (2016). Database Processing: Fundamentals, Design and Implementation Edition 14. Pearson, Figure 9-3)

A transaction has ACID properties: Atomicity, Consistency, Isolation and Durability.

Read

Kroenke, D and D. Auer. (2016). *Database Processing: Fundamentals, Design and Implementation Edition 14*. Pearson, 460-462.

2.1.1 Atomicity

A transaction is atomic. This means that all the database actions within a transaction must be performed or none must be performed, as discussed in Section 2.1.

When all the database actions in a transaction are successfully performed, the transaction commits.

When one database action fails, the earlier actions must be undone so that the database remains consistent. The transaction must rollback.

If transaction is nested within another transaction, and if a database action of the outer transaction fails, then, because a transaction is atomic, the inner transaction will also be undone, even if the inner transaction has committed.

Activity 9

Reproduced from Question 9.8 of the course text.

Define an atomic transaction and explain why atomicity is important.

2.1.2 Consistency

A database is consistent when none of the database constraints are violated.

If, at the start of a transaction and when a transaction completes, the database continues to be consistent, we say that the transaction is consistent. A consistent transaction does not leave a database in an inconsistent state.

2.1.3 Isolation

A transaction is isolated if it is not affected by the database actions of other ongoing transactions in unintended ways.

Consider the scenario in Figure 4.5. Two concurrent transactions, after reading the current value of a shared resource, attempt to update the value of the shared resource. The update action by user A is lost when user B performs an update. User B has affected user A in an unintended way.

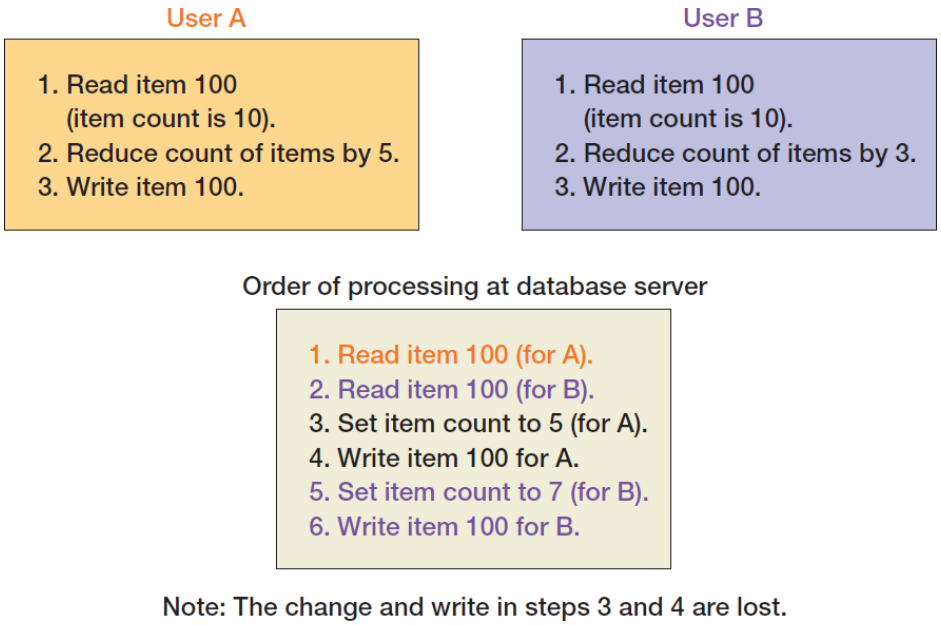


Figure 4.5 Lost Update Problem

(Source: Kroenke, D and D. Auer. (2016). Database Processing: Fundamentals, Design and Implementation Edition 14. Pearson, Figure 9-5)

To prevent other transactions from affecting a transaction in unintended ways, we isolate the transactions. Isolation requires the use of locks.

If user A had locked item 100, it would have prevented other users, such as user B from accessing item 100. User B would not have been able to read and update item 100 until user A is done. Thus, a lock on item 100 can isolate user A from user B, and prevent the lost update problem.

Isolation requires that a transaction get a lock on a resource before using it. If the resource is already locked by another transaction, the requesting transaction waits until the resource is released. If there is no lock on a required resource yet, the transaction gets the lock on the resource and can use it.

Refer to Figure 4.6 on how locks can prevent lost updates. This example uses explicit locks, that is, the lock request for Item 100 is explicitly requested. SQL Server does not require user to request for locks explicitly. Instead transactions specify the isolation level they require, and the DBMS will issue locks according to the requested isolation level.

Isolation levels are covered in Section 2.2.5 of this study unit.

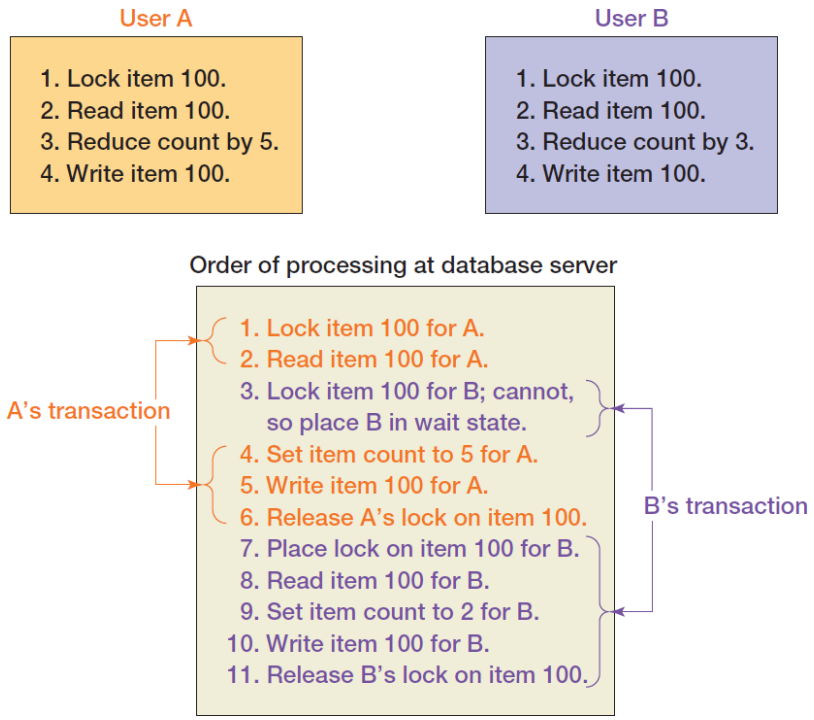


Figure 4.6 Lost Concurrent Processing with Explicit Locks

(Source: Kroenke, D and D. Auer. (2016). Database Processing: Fundamentals, Design and Implementation Edition 14. Pearson, Figure 9-5)

Read

Kroenke, D and D. Auer. (2016). *Database Processing: Fundamentals, Design and Implementation Edition 14*. Pearson, 462.

Introducing locks in transactions can cause deadlocks. A deadlock happens when one transaction, t_A requests for a resource held by another transaction, t_B , and t_B is in a wait state because it is waiting for a resource that t_A has locked. The lock requests result in a cycle of transactions waiting for one another to release resources they need.

In general, a deadlock may involve two or more transactions, with each transaction waiting for a resource held by another transaction, resulting in a deadly embrace.

Read

Kroenke, D and D. Auer. (2016). *Database Processing: Fundamentals, Design and Implementation Edition 14*. Pearson, 463-464.

2.1.4 Durability

A committed transaction is durable, that is, the changes it makes must persist in the database in secondary storage.

The tables in a database are stored in data and index files, partitioned in pages in secondary storage. When a transaction makes a request to manipulate data in a page, the DBMS first checks for the required page in the DBMS buffers. If the page is not in the buffers yet, it will be read into buffer frame.

When a transaction modifies data, the changes to the database are made on the buffers first. Dirtied buffers are written back to the secondary storage at some point,

so that the changes are permanently recorded on the secondary storage. The DBMS guarantees that the database actions of a committed transaction are reflected on the secondary storage.

We will look at how a DBMS ensure that transactions are durable even in an event of a system crash when we cover database recovery in Study Unit 5.

2.2 Concurrency Control

Concurrency control is about taking measure to ensure that no transaction affects another in an unintended manner. The key to concurrency control is isolation through locks. There are different types of lock and lock granularity as well as different locking techniques.

2.2.1 Shared Locks and Mutually Exclusive Locks

A read lock is a shared lock. When a transaction needs to read data, it requests for a read lock. If the data is not locked or if it has been locked with a read lock, the request is successful. Otherwise, the transaction waits until the lock is released and given to it. A read lock is a shared lock. It allows one or more transactions to read the same data.

A write lock is a mutually exclusive lock. When a transaction needs to change the data, it requests for a write lock. If the data is not locked already by a write lock or a read lock, the request is successful. Otherwise, the transaction waits until the lock is released and given to it.

Once data is write-locked, no other transaction can get a read or write lock for it until the write lock is released. A write lock allows only one transaction to change the data.

When a write lock is released, the data can be locked by one of the waiting transactions. Either a read lock or write lock will apply, depending on which lock request is fulfilled next.

A read lock will allow all waiting transactions for read lock to be released from the wait. A write lock will allow only one waiting (the chosen) transaction to be released from the wait.

Locks lowers throughput but they isolates transactions.

Activity 10

Reproduced from Question 9.13 of the course text.

Explain the difference between an exclusive lock and a shared lock.

2.2.2 Lock Granularity

A lock request can be fulfilled by locking a row, a page or a table, depending on the isolation level the transaction is running under and types of cursors it uses.

A smaller lock granularity such as row locks means that the DBMS must manage more locks. However, having a smaller lock granularity improves database throughput.

A bigger lock granularity such as a table lock is easier to manage but the database throughput will be lowered as more transactions are unable to get resources.

2.2.3 Optimistic and Pessimistic Locking

Optimistic locking is an attempt to hold a write lock for the shortest time possible, so that another transaction can use the resource as soon as possible. It can be applied when data collision is rare, i.e., when it is rare to have two or more transaction making changes (update or delete) to the same resource.

Refer to Figure 4.7 for an example of optimistic locking. Note that a transaction does not lock the Product row (or page or table, depending on the lock granularity) until it is about to update the row. Also, once the `update` statement has executed, the lock is released.

However, the `update` statement may not get to update the row for Pencil if the quantity has been changed between the `select` statement and the `update` statement. If the quantity has been changed, there will be no row that will match the `where` clause of the `update` statement, and so the `update` statement will fail to find a row to update.

This happens when some transaction has updated quantity. The value of `newQuantity` obtained through the `set` statement is thus invalid. The transaction must be repeated to reread quantity, recompute `newQuantity`, and finally re-update the Product table.

Note that transaction is repeated until the update finds a row to update, that is, until `newQuantity` is valid.

```
/* *** EXAMPLE CODE - DO NOT RUN *** */
/* *** SQL-Code-Example-CH09-01 *** */

SELECT   PRODUCT.Name, PRODUCT.Quantity
FROM     PRODUCT
WHERE    PRODUCT.Name = 'Pencil';

Set NewQuantity = PRODUCT.Quantity - 5;

{process transaction - take exception action if NewQuantity < 0, etc.

Assuming all is OK: }

LOCK     PRODUCT;

UPDATE   PRODUCT
SET      PRODUCT.Quantity = NewQuantity
WHERE    PRODUCT.Name = 'Pencil'
     AND PRODUCT.Quantity = OldQuantity;

UNLOCK   PRODUCT;

{check to see if update was successful;
if not, repeat transaction}
```

Figure 4.7 Optimistic Locking

(Source: Kroenke, D and D. Auer. (2016). Database Processing: Fundamentals, Design and Implementation Edition 14. Pearson, Figure 9-8)

Figure 4.8 shows pessimistic locking. This technique assumes that there is a high probability that some transaction will change the value of quantity between the `select` statement and the `update` statement. Therefore, a lock is requested when the transaction starts and release when the transaction ends.

The transaction holds the lock, and no other transaction can update quantity to invalidate the `set` statement. Therefore, the `update` statement will always be able to update the row for Pencil.

As the lock is held for a longer period, the throughput is lowered.

```
/* *** EXAMPLE CODE - DO NOT RUN *** */
/* *** SQL-Code-Example-CH09-02 *** */

LOCK     PRODUCT;

SELECT   PRODUCT.Name, PRODUCT.Quantity
FROM     PRODUCT
WHERE    PRODUCT.Name = 'Pencil';

Set NewQuantity = PRODUCT.Quantity - 5;

{process transaction - take exception action if NewQuantity < 0, etc.

Assuming all is OK: }

UPDATE   PRODUCT
SET      PRODUCT.Quantity = NewQuantity
WHERE    PRODUCT.Name = 'Pencil';

UNLOCK   PRODUCT;

{no need to check if update was successful}
```

Figure 4.8 Pessimistic Locking

(Source: Kroenke, D and D. Auer. (2016). Database Processing: Fundamentals, Design and Implementation Edition 14. Pearson, Figure 9-8)

Read

Kroenke, D and D. Auer. (2016). *Database Processing: Fundamentals, Design and Implementation Edition 14*. Pearson, 465-466.

2.2.4 Statement Level and Transaction Level Consistency

A transaction may request for an isolation level which will offer consistency either at statement level or at transaction level.

- Statement level consistency

When a statement in a transaction executes with statement level consistency, other transactions are prevented from updating or deleting the data that the statement is using.

This enables the statement to apply consistently on the data as there is no other write action to change the data that the statement is processing

- Transaction level consistency

  When the transaction executes at transaction level consistency, the data that the transaction uses cannot be updated or deleted by other transactions, until the transaction completes.

  This enables the transaction to apply consistently on the data throughout the transaction as there is no other write action to change the data that the transaction is processing.

Both statement level consistency and transaction level consistency require locks to be applied.

Read

Kroenke, D and D. Auer. (2016). *Database Processing: Fundamentals, Design and Implementation Edition 14*. Pearson, 466-469.

2.2.5 Isolation Level

Refer to Figure 4.9 for the definition of problems that may result when transaction do not have the required isolation levels.

Data Read Problem Type	Definition
Dirty Read	The transaction reads a row that has been changed, but the change has *not* been committed. If the change is rolled back, the transaction has incorrect data.
Nonrepeatable Read	The transaction rereads data that has been changed, and finds updates or deletions due to committed transactions.
Phantom Read	The transaction rereads data and finds new rows inserted by a committed transaction.

Figure 4.9 Summary of Data Read Problems

(Source: Kroenke, D and D. Auer. (2016). Database Processing: Fundamentals, Design and Implementation Edition 14. Pearson, Figure 9-11)

These are four levels of isolation in SQL-92:

- read uncommitted

  This is the lowest level of isolation with the highest throughput. This level allows a transaction to read data updated or inserted by a transaction that has not been committed.

  If the transaction that provides the data rollbacks, a dirty read problem arises as the read data is no longer valid data. That may mean that the decisions made by transactions that perform dirty reads will be incorrect. Therefore, when the transaction that provides the data rollbacks, transactions that perform the dirty read may have to be rollbacked as well.

  This level does not apply lock.

- read committed

  This level allows only committed data to be read. Thus, at this level, the dirty read problem is eliminated.

  Read committed isolation level uses a shared read lock, which is obtained when there is no write lock on the data. At this level, the read lock can be released once the required data is read.

As the shared read lock may be released between the reads, other transactions can obtain a write lock to update or delete the data. Therefore, at this isolation level, two reads on the same table may not be repeatable. This is the non-repeatable read problem.

- Repeatable read

  At this level, the shared read lock is not released until the end of the transaction. As no write lock can be granted and so, no data can be updated or deleted as long as the read lock is not released, this level eliminates the non-repeatable read problem.

  However, new data rows can still be inserted, resulting in the phantom read problem.

- Serializable

  This is the highest level of isolation with the lowest throughput. At this level, the resultant database that we get from interleaving the database actions of all concurrent serializable transactions can always be derived as if we are executing each transaction to completion, one transaction after another, in some specific serial order, that is, we can always find a serial schedule when executing serializable transactions.

  Serializable transaction requires 2-phase locking or 2PL. In 2PL, once a transaction releases a lock, the transaction cannot request for more locks. It can also keep releasing locks. Thus, 2PL has a growing phase followed by a shrinking phase.

  2PL guarantees a serial schedule but it can result in dirty reads as locks are released at any point after the last lock request, before the transaction is ready to commit. Furthermore, if the transaction later rollbacks, it can cause cascading aborts or cascading rollbacks.

Activity 11

Reproduced from Question 9.14 of the course text.

Explain two-phased locking.

- Strict 2PL is a special form of 2PL whereby all locks are released together, only when transactions commit or rollback. Strick 2PL does not result in cascading aborts. At this level, the dirty read problem, the non-repeatable problem and the phantom read problem are all eliminated. Many DBMSs use strict 2PL to implement the serializable isolation level.

Refer to Figure 4.10 for a summary of the four transaction isolation levels.

		Isolation Level			
		Read Uncommitted	Read Committed	Repeatable Read	Serializable
Problem Type	Dirty Read	Possible	Not Possible	Not Possible	Not Possible
	Nonrepeatable Read	Possible	Possible	Not Possible	Not Possible
	Phantom Read	Possible	Possible	Possible	Not Possible

Figure 4.10 Summary of Transaction Isolation Levels

(Source: Kroenke, D and D. Auer. (2016). Database Processing: Fundamentals, Design and Implementation Edition 14. Pearson, Figure 9-12)

SQL Server does not allow locks to be explicitly requested. Instead, transactions specify the isolation level they require before the start of a transaction.

Example:

```
Set transaction isolation level read committed
```

```
Begin transaction
   SomeStatements
Commit
```

Besides specifying isolation level, a transaction can also specify the types of cursors it uses to process a resultset. The type of cursors determines the lock granularity as well as the duration of time the lock is held. Refer to the comments column in Figure 4.11 for how cursor types affect isolation of cursor operations that occur between opening and closing a cursor..

CursorType	Description	Comments
Forward only	Application can only move forward through the recordset.	Changes made by other cursors in this transaction or in other transactions will be visible only if they occur on rows ahead of the cursor.
Static	Application sees the data as they were at the time the cursor was opened.	Changes made by this cursor are visible. Changes from other sources are not visible. Backward and forward scrolling allowed.
Keyset	When the cursor is opened, a primary key value is saved for each row in the recordset. When the application accesses a row, the key is used to fetch the current values for the row.	Updates from any source are visible. Inserts from sources outside this cursor are not visible (there is no key for them in the keyset). Inserts from this cursor appear at the bottom of the recordset. Deletions from any source are visible. Changes in row order are not visible. If the isolation level is read-uncommitted, then uncommitted updates and deletions are visible; otherwise only committed updates and deletions are visible.
Dynamic	Changes of any type and from any source are visible.	All inserts, updates, deletions, and changes in recordset order are visible. If the isolation level is dirty read, then uncommitted changes are visible. Otherwise, only committed changes are visible.

Figure 4.11 Summary of SQL Cursor Types

(Source: Kroenke, D and D. Auer. (2016). Database Processing: Fundamentals, Design and Implementation Edition 14. Pearson, Figure 9-13)

Read

Kroenke, D and D. Auer. (2016). *Database Processing: Fundamentals, Design and Implementation Edition 14*. Pearson, 469-470.

2.2.6 Deadlock Detection and Resolution

Locks are introduced to isolate one transaction from one another. However, locks can result in deadlocks. When lock requests are cyclic, as seen in the example in Figure 4.12, a deadlock arises. Both user A and user B will be in the wait state indefinitely.

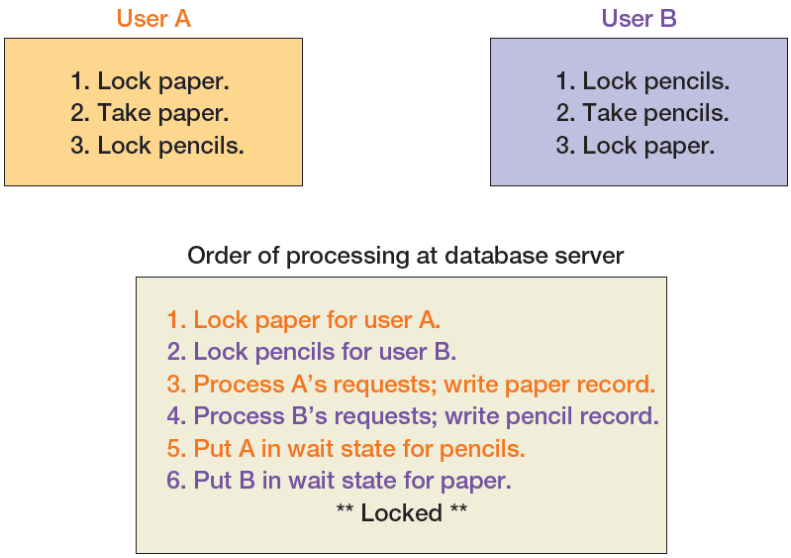


Figure 4.12 Deadlock example

(Source: Kroenke, D and D. Auer. (2016). Database Processing: Fundamentals, Design and Implementation Edition 14. Pearson, Figure 9-6)

A DBMS can identify whether a deadlock has occurred as it tracks both the fulfilled and unfulfilled resource requests. When the DBMS detects a deadlock, it resolves the deadlock by selecting a victim to break the wait cycle.

The victim is terminated and those database actions that have been carried out for the victim are rollbacked. The resources locked by the victim are released and assigned to requesting transactions.

Read

Kroenke, D and D. Auer. (2016). *Database Processing: Fundamentals, Design and Implementation Edition 14*. Pearson, 464.

Activity 12

Reproduced from Question 9.17 of the course text.

What is deadlock? How can it be avoided? How can it be resolved once it occurs?

Summary

SQL/PSM (SQL Persistent Store Module) is an extension of SQL that allows us to write procedural code. SQL/PSM provides standard programming constructs such as variables, type definition, selection and loop control structures, and allows modules to raise and handle exceptions.

SQL/PSM also provides constructs to define table type to allow a module to receive a table as parameter. In addition, it also provides a structure where rows in a resultset can be accessed using cursor operations.

SQL/PSM can be used to define functions, stored procedure and trigger. A function can accept input parameters and must return either a scalar or a table value. A stored procedure can have both input and output parameters. Both functions and stored procedures must be called. A trigger is some code that accompanies an SQL DDL or DML operation. The code may execute before, after or in place of the requested SQL DDL or DML operation.

A transaction has ACID properties. It is atomic. If any of the statements in a transaction fails, all the actions performed must be undone, so that the database remains consistent. A transaction is consistent if at the end of the transaction, remains consistent. A transaction is isolated at the level it requests. Isolation means that other transactions do not affect it in an unintended manner. A transaction is durable. Committed transactions are guaranteed to persist.

SQL/TCL or transaction control language is used to demarcate the boundary of a transaction with `begin transaction`, `commit` and `rollback`. The transaction boundaries help the DBMS know which statements constitute a transaction, that is, complete piece of work or a logical unit of work.

The DBMS isolates one transaction from other transactions by locking the resources it is using. Locks can be shared or mutually exclusive, using pessimistic or optimistic locking, and requested explicitly or implicitly. A result locking is deadlock. Deadlock is resolved by killing one transaction and then releasing its resources.

References

Book

Author(s)	Year	Book Title	Edition	Publisher
Kroenke, D., & Auer, D. J.	2016	*Database Processing – Fundamentals, Design, and Implementation*	14	Pearson

Quiz

1. Which assignment statement is incorrect?

   a. `declare @var1 int = 3`

   *b. `set @var1 int = 3`

   c. `set @var1 = 3`

   d. None of the statements is incorrect

2. Which is the correct sequence when using a cursor?

   i. open cursor

   ii. fetch from cursor

   iii. deallocate cursor

   iv. close cursor

   v. declare cursor

   a. i., ii., iii., iv., v.

   b. i., v., ii., iii., iv.

   c. v., i., ii., iii., iv.

   *d. v., i., ii., iv., iii.

3. Which statement about stored procedures and triggers is false?

   a. Both stored procedures and triggers can get input though from difference sources.

   b. A stored procedure must be called but a trigger gets executed when the operation it is defined for is requested.

   *c. Both stored procedures and triggers do not produce output.

   d. None of the statements is false.

4. What is the purpose of concurrency control?

   a. Ensure that all transactions complete successfully

   b. Maximise database throughput

   c. Ensure that there is no deadlock

*d. None of the statements is incorrect

5. Which isolation level does not use locks?
   *a. read uncommitted
   b. read committed
   c. repeatable read
   d. serializable

6. What type of locks should be used if we assume that a conflict will occur?
   a. explicit locks
   b. implicit locks
   c. optimistic locks
   *d. pessimistic locks

Formative Assessment

1. Which statement(s) defines two variables?

a. set @a =1; set @b = 2

Incorrect. The statements assign values to the variable. Refer to Section 1.1.1 of this study guide.

b. declare @a = 1, @b = 1

Incorrect. The syntax is incorrect. Refer to Section 1.1.1 of this study guide.

c. declare @a int = 1, @b int = 2

Correct! The statement defines two variables. In addition, the variables are also assigned values. Refer to Section 1.1.1 of this study guide.

d. set @a int =1; set @b int = 2

Incorrect. The syntax is incorrect. Refer to Section 1.1.1 of this study guide.

2. Assume that several customer details are in a tabled-value parameter. Which scenario requires a cursor?

a. Displaying the count of number of orders each customer in the tabled-value parameter has made. Assume that there is an Order table.

Incorrect. Only a select statement with outer join and a group by clause is necessary. Refer to Study Unit 3 of this study guide.

b. Updating customer table with the tabled-value parameter

Incorrect. A bulk update syntax can be used. Refer to Study Unit 3 of this study guide.

c. Displaying the customer details of customer whose customer ids are in a tabled-value parameter.

Incorrect. Only a select statement with where clause with in operator is necessary. Refer to Study Unit 3 of this study guide.

d. Inserting customers into different tables based on the first character of customer id.

Correct. A cursor is needed to allow each row to be individually inserted into the different tables. Refer to Section 1.1.2 of this study guide.

3. Which modules cannot raise errors?

a. function

Incorrect. A function should raise an error if it detects a business rule violation. Refer to Section 1.1.4 of this study guide.

b. stored procedure

Incorrect. A stored procedure should raise an error if it detects a business rule violation. Refer to Section 1.1.4 of this study guide.

c. trigger

Incorrect. A trigger should raise an error if it detects a business rule violation. Refer to Section 1.1.4 of this study guide.`

d. None of the above

Correct! Any module that detects a business rule violation should raise error. Refer to Section 1.1.4 of this study guide.

4. Which property of transaction is being described by the statement 'A committed transaction is persisted'?

a. durable

Correct! The statement refers to the DBMS ensuring that changes made by a committed transaction are reflected in the database. Refer to Section 2.1 of this study guide.

b. atomic

Incorrect. Atomicity refers to the database actions of a transaction must all succeed for none of them must succeed. Refer to Section 2.1 of this study guide.

c. isolated

Incorrect. Isolation refers to the DBMS ensuring that transactions do not affect one another in an unintended manner. Refer to Section 2.1 of this study guide.

d. consistent

Incorrect. Consistency refers to a transaction that does not leave the database in an inconsistent state. Refer to Section 2.1 of this study guide..

5. Name the problem that refers to one transaction's change to the database is overwritten by another transaction's change, causes the first change to be lost.

a. dirty read problem

Incorrect. A dirty read problem arises when non-committed data is read. Refer to Section 2.2 of this study guide.

b. lost update

Correct! A lost update problem arises when one transaction overwrites the change of another transaction. Refer to Section 2.1 of this study guide.

c. non-repeatable read.

Incorrect. A non-repeatable read occurs when between two reads of one transaction, another transaction makes a change to the read data. Refer to Section 2.2 of this study guide.

d. phantom read problem

Incorrect. A phantom read occurs when between two reads of one transaction, another transaction inserts data. Refer to Section 2.2 of this study guide.

6. The situation that occurs when two users are each waiting for a resource that the other person has locked is known as a(n) _____.

a. deadlock

Correct! Deadlock is a result of cyclic wait. Refer to Section 2.2.6 of this study guide.

b. shared lock

Incorrect. A shared lock allows one or more transactions to read the data resource. Refer to Section 2.2.1 of this study guide.

c. pessimistic locking

Incorrect. Pessimistic locking refers to locking the resources at the start of the transaction and releasing when the resource is not needed. Refer to Section 2.2.3 of this study guide.

d. mutually exclusive lock

Incorrect. A mutually exclusive lock allows only one transaction to hold the data resource at any time. Refer to Section 2.2.1 of this study guide.

Solutions or Suggested Answers

Activity 1

```
create table Product(
      productId int not null,
      name varchar(20) not null,
      unitPrice numeric(5, 2) not null,
      quantityAvailable int not null,
      constraint ProductPK primary key(productId),
      constraint unitPriceCheck check(unitPrice < 500),
      constraint quantityAvailableCheck check(quantityAvailable >= 0)
)

insert into product values
(1, '2B Pencil', 0.35, 2),
(2, '3B Pencil', 0.50, 10),
(3, 'Calculator', 27.85, 0)


--(a)
declare @productid int, @unitPrice numeric(6, 2)

--(b)
set @productid = 2
set @unitPrice = 0.50

--(c)
update Product
set unitPrice = @unitPrice
where productId = @productid
select * from Product

--(d)
declare @productid int = 2, @unitPrice numeric(6, 2) = 0.50
set @productid = 2
set @unitPrice = 0.50
delete from Product where productId = @productid

--(e)
declare @productid int = 2, @unitPrice numeric(6, 2) = 0.50
insert into Product values
      (@productid, 'mechanical pencil', @unitPrice + 2.35, 10)
```

Activity 2

```
set nocount on
declare @productid int, @unitPrice numeric(6, 2),
```

```
      @quantityAvailable int, @count1 int = 0, @count2 int = 0,
      @count3 int = 0

select @count1 = count(*)
      from Product where unitPrice < 5
select @count2 = count(*)
      from Product where unitPrice >= 5 and unitPrice < 10
select @count3 = count(*)
      from Product where unitPrice >= 10

print 'less than $5         - ' + cast(@count1 as varchar(2))
print '$5 but less than $10 - ' + cast(@count2 as varchar(2))
print '$10 or more          - ' + cast(@count3 as varchar(2))
```

Activity 3

```
create type pType as table
(productId int)


set nocount on
declare @productid int, @unitPrice numeric(6, 2),
      @quantityAvailable int
declare @p1 pType, @p2 pType, @p3 pType

insert into @p1 select productid
      from Product where unitPrice < 5
insert into @p2 select productid
      from Product where unitPrice >= 5 and unitPrice < 10
insert into @p3 select productid
      from Product where unitPrice >= 10

DECLARE @result varchar(1000), @countrow int

select @countrow = count(*) from @p1
print 'less than $5         - ' + cast(@countrow as varchar(2))
if exists (select * from @p1)
begin
      SELECT @result = ISNULL(@result, '') +
            cast(productId as varchar(2)) + ', '
      FROM @p1
      set @result = substring(@result, 0, len(@result))
      if @countrow = 1
            print 'Product is ' + @result
      else
            print 'Products are ' + @result
end

select @countrow = count(*) from @p2
```

```
print '$5 but less than $10 - ' + cast(@countrow as varchar(2))

if exists (select * from @p2)
begin
      set @result = ''
      SELECT @result = ISNULL(@result, '') +
            cast(productId as varchar(2)) + ', ' FROM @p2
      set @result = substring(@result, 0, len(@result))
      if @countrow = 1
            print 'Product is ' + @result
      else
            print 'Products are ' + @result
end

select @countrow = count(*) from @p3
print '$10 or more          - ' + cast(@countrow  as varchar(2))
if exists (select * from @p3)
begin
      set @result = ''
      SELECT @result = ISNULL(@result, '') +
            cast(productId as varchar(2)) + ', ' FROM @p3
      set @result = substring(@result, 0, len(@result))
      if @countrow = 1
            print 'Product is ' + @result
      else
            print 'Products are ' + @result
end
```

Activity 4

```
-- (a)
create function countInPriceCategory (@lower int, @upper int)
      returns int
as
begin
      declare @count int
      select @count = count(*) from Product
      where unitPrice >= @lower and unitPrice < @upper
  return @count
end
```

Test Script

```
set nocount on
print 'less than $5         - ' +
      cast(dbo.countInPriceCategory(0, 5) as varchar(2))
print '$5 but less than $10 - ' +
      cast(dbo.countInPriceCategory(5, 10) as varchar(2))
print '$10 or more          - ' +
      cast(dbo.countInPriceCategory(10, 500) as varchar(2))
```

```
-- (b)
create function productsInPriceCategory (@lower int, @upper int)
      returns table
as
      return select productId
            from Product
            where unitPrice >= @lower and unitPrice < @upper
```

Test Script

```
declare @p ptype
insert into @p
      select productId from dbo.productsInPriceCategory(0, 5)
select * from @p
```

Activity 5

```
create procedure printProductInCategory
      @message varchar(25), @lower int, @upper int
as
begin
      DECLARE @result varchar(1000), @countrow int, @p ptype

      insert into @p select productId
            from dbo.productsInPriceCategory(@lower, @upper)
      set @countrow = dbo.countInPriceCategory(@lower, @upper)
      print @message + cast(@countrow as varchar(2))

      if exists (select * from @p)
      begin
            SELECT @result = ISNULL(@result, '') +
                  cast(productId as varchar(2)) + ', ' FROM @p
            set @result = substring(@result, 0, len(@result))
            if  @countrow = 1
                  print 'Product is ' + @result
            else
                  print 'Products are ' + @result
      end
end

create procedure listByPriceCategory as
begin
      set nocount on
      exec printProductInCategory 'less than $5         - ', 0, 5
      exec printProductInCategory '$5 but less than $10 - ', 5, 10
      exec printProductInCategory '$10 or more          - ', 10, 500
end
```

Test Script

```
exec listByPriceCategory
```

Activity 6

```
create procedure deleteProducts @pids ptype readonly as
begin
      set nocount on
      declare @pid int, @name varchar(20), @unitPrice numeric(5,2),
            @quantityAvailable int
      declare pCursor cursor for select productId from @pids
      open pCursor
      fetch next from pCursor into @pid
      while @@FETCH_STATUS = 0
      begin
            if not exists (select * from Product
                              where productId = @pid)
                  print cast(@pid as varchar(2)) +
                        ' is an invalid product id'
            else
            begin
                  select @quantityAvailable = quantityAvailable,
                        @name = name, @unitPrice = unitPrice
                  from Product where productId = @pid
                  if @quantityAvailable > 0
                        print cast(@pid as varchar(2)) + ', ' + @name
                              + ', $' + cast(@unitPrice as varchar(7))
                              + ', ' +
                              cast(@quantityAvailable as varchar(2))
                              + ' cannot be deleted. Available quantity
                              is non zero'
                  else
                        begin
                              delete from Product
                                    where productId = @pid
                              print cast(@pid as varchar(2)) + ', ' +
                                    @name + ', $' +
                                    cast(@unitPrice as varchar(7)) +
                                    ', ' + cast(@quantityAvailable as
                                    varchar(2))
                                    + ' is successfully deleted.
Available quantity is zero'
                         end
            end
            fetch next from pCursor into @pid
      end
      close pCursor
      deallocate pCursor
end
```

Test Script

```
declare @p ptype
insert into @p values (1), (3), (5)
exec deleteProducts @p
```

Activity 7

```
create trigger deleteProductCheck on Product
instead of delete
as
begin
	declare @pid int, @name varchar(20), @unitPrice numeric(5,2),
		@quantityAvailable int
	declare pCursor cursor for
		select * from deleted
	open pCursor
	fetch next from pcursor into
		@pid, @name, @unitPrice, @quantityAvailable
	while @@FETCH_STATUS = 0
	begin
		if @quantityAvailable > 0
			print cast(@pid as varchar(2)) + ', ' + @name +
				', $' + cast(@unitPrice as varchar(7)) + ', '
				+ cast(@quantityAvailable as varchar(2))
				+ ' cannot be deleted. Available quantity is
non zero'
		else
			begin
				delete from Product where productId = @pid
				print cast(@pid as varchar(2)) + ', ' + @name
					+ ', $' +
					cast(@unitPrice as varchar(7)) + ', ' +
					cast(@quantityAvailable as varchar(2))
					+ ' is successfully deleted. Available
quantity is zero'
			end
		fetch next from pcursor into
			@pid, @name, @unitPrice, @quantityAvailable
	end
	close pCursor
	deallocate pCursor
end
```

Test Script

```
delete from Product
where productId in (1, 5, 18, 3)
```

Activity 8

```
create trigger insertProductCheck on Product
for insert
as
begin
      if exists (select * from inserted where quantityAvailable = 0)
      begin
            rollback
            raiserror( N'Error: Inserted product must have positive
quantity svailable', 16, 1)
      end
      else
            print 'Insert operation is successful'
end
```

Test Script

```
begin try
      insert into product values
      (10, '2B Pencil', 0.35, 2),(11, '2B Pencil', 0.35, 0)
end try
begin catch
      print error_message()
end catch
```

Activity 9

An atomic transaction is a series of actions to be taken on the database so that either all of them are performed successfully or none of them are performed at all, in which case the database remains unchanged. Such a transaction is sometimes called a Logical Unit of Work because it is performed as a unit.

Activity 10

An exclusive lock locks the item from access of any type. No other transaction can read or change the data. A shared lock locks the item from change but not from read. That is, other transactions can read the item as long as they do not attempt to alter it.

Activity 11

Transactions are allowed to obtain locks as necessary, but once the first lock is released, no other lock can be obtained. Transactions thus have a growing phase, in which the locks are obtained, and a shrinking phase, in which the locks are released.

Activity 12

Deadlock occurs when User1 locks a resource needed by User2 and User2 locks a resource needed by User1. Each is waiting for a resource that the other person has locked. One way of avoiding a deadlock is to allow users to issue all lock requests at one time. Users must lock all the resources they want at once. When deadlock occurs, the normal solution is to rollback one of the transactions to remove its changes from the database.