A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

EFOP-3.4.3-16-2016-00009

Advanced database management systems

Course notes

István Vassányi, PhD

vassanyi at almos dot uni-pannon.hu

University of Pannonia, Department of Electrical Engineering and Information Systems

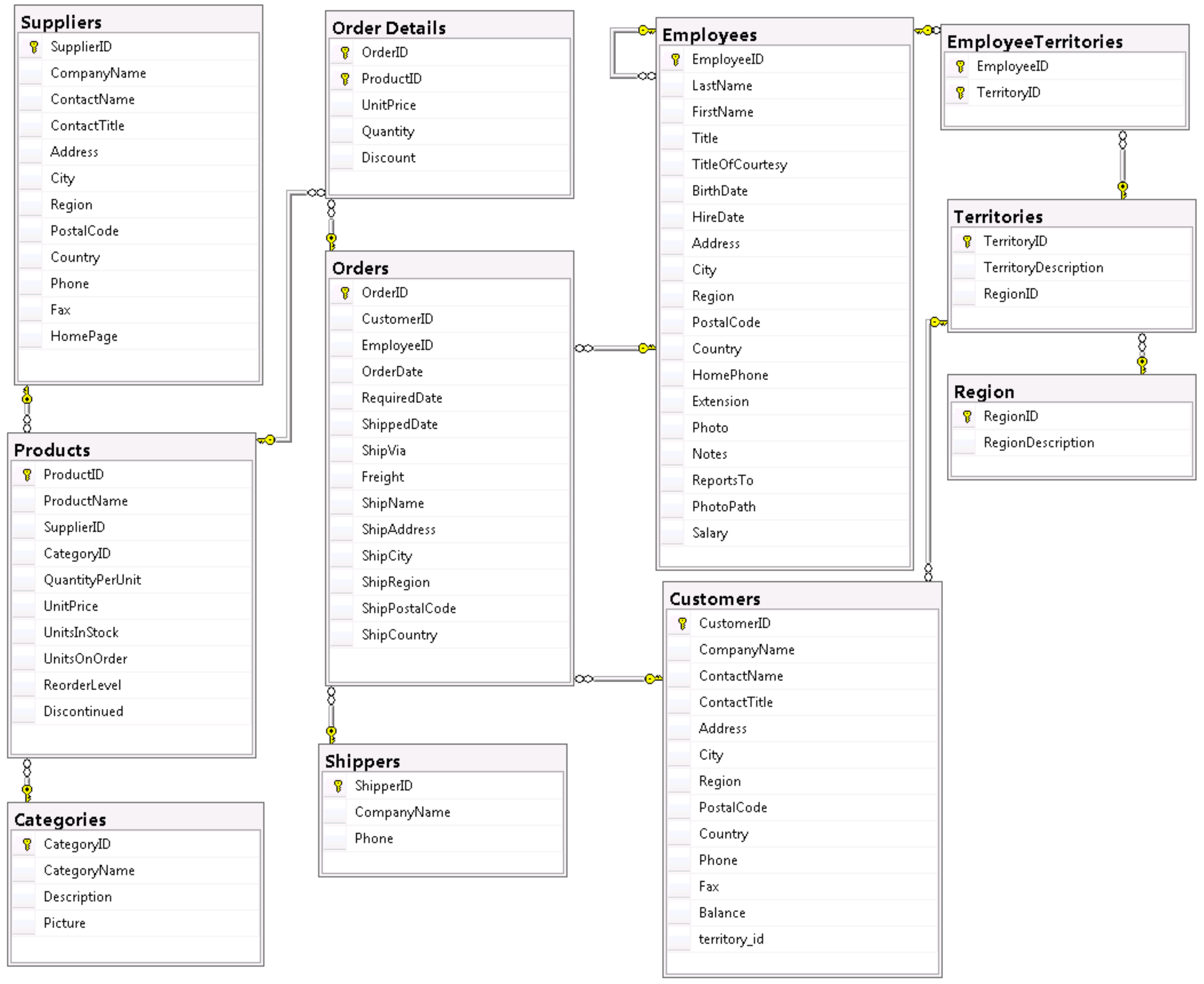
Veszprém, Hungary

June 2018

CONTENTS

1. Review of core database skills

Welcome. In most of the demos in this course, we'll use the Northwind sample relational database[1] and the SQL server 2016 technology from Microsoft. The Northwind database was designed to support a small company trading with consumables. It includes an inventory and tables for the administration of the orders. The table and filed names should be self-explanatory.



Modeling

- First we review the core relational modeling concepts for On-Line Transaction Processing (OLTP) databases, demonstrated on the Northwind database: Customers, Employees, Orders, OrderDetails, Products, Categories, Territories tables.
    - We start with a conceptual model (domain model or entity relationship model) that we derive from the use cases and our aim is to develop the logical database model

[1] You can download the database dump from https://www.microsoft.com/en-us/download/details.aspx?id=23654
In this course, we modified the original database by adding a foreign key territory_id to the Customers table and an extra field Salary to the Employees table, for the sake of some exercises.

- The relational model is the most widely used paradigm to support traditional business processes due to its simplicity
- Entities, attributes, instances, identifiers are implemented in the relational model as tables, fields, records, primary keys. Keys may be composed from multiple fields
- Only one value in any single cell—no redundancy and no inconsistency is allowed in third normal form (3NF). Characteristics of 3NF:
    - Each table has a primary key that may be composed of multiple fields, and on which all the other fields functionally depend;
    - In case of composite (multi-field) keys, all of the non-key fields depend on the whole key, and not just a part of it i.e. there are no partial dependencies;
    - The non-key fields depend on no other field(s) except the key, i.e. there are no transitive dependencies within a table.
- All tables are connected
- 1:N (one-to-many) relationships are implemented with foreign keys (e.g. Orders.EmployeeID)
- N:M (many-to-many) relationships are implemented with linking tables (e.g. EmployeeTerritories)
- 1:1 (one-to-one) relationship is not exemplified in the Northwind database
    - A normal 1:1 relationship could be a CompanyCar table if an employee may have at most one company car allocated
    - A specialization type 1:1 relationship could be an ExciseProducts table for excise goods with extra fields ExciseDutyAmount, RegBarCode etc.
- Linking tables usually have composite keys. We generate keys only if an external reference is needed.
- The relationship structure of an OLTP schema reveals the key transactions of the application that uses the database.
    - Snowflake or snowball structure, each snowflake supporting one or more transactions.
    - Base tables are at the leaves (e.g. Region, Customers, Categories)
    - Transactional tables or event-tables are in the middle (Order Details, EmployeeTerritories). These tables form the 'beating heart' of the information system.

Feature	**Base Tables**	**Transactional Tables**
Position in the schema	Leaf. Does not reference any other table	Centre. References directly or indirectly all tables
Size	Small	Large

Speed of change	Slow. Cold backups may be sufficient.	Fast. Hot backups are needed.

- Connection between a properly designed Graphical User Interface (GUI) and the relational schema
    - Hidden or read-only label: key
    - Editable text boxes: attributes (fields) that depend on the key
    - Dropdown/combo lists: references to base tables
    - Checkbox with an additional text box: specialization
    - Dropdown tables or lists: 1:N relationships
- Further reading on modeling:
    - https://www.safaribooksonline.com/library/view/relational-theory-for/9781449365431/ch01.html
    - http://www.blackwasp.co.uk/RelationalDBConcepts.aspx
    - https://www.tutorialspoint.com/ms_sql_server/index.htm
- PRACTICE: create and extend the sample database
    - Install MS SQL Server 2016 or later, start the database service and connect to it using MS Management Studio.
    - Run the northwind database create dump and review the tables with the GUI tools
    - Draw a logical database model diagram similar to the diagram above
    - Add the fields Employees.Salary and Customers.territory_id
    - Design and implement an extension to the database to model the following scenario. *We send our employees to regular training sessions where they learn various skills. Training sessions are organized by contracted third party companies. We have a list of required skills (like "grade B business presentation" or "accounting basics" etc.) for each employment category (like "sales manager", see Employees.Title) that they must learn within 10 years after the beginning of their employment. For each training, we store the duration (beginning and ending date), location, organizing company, skills taught, participants, their training status (like "enrolled", "started", "completed", "aborted") and their exam results separately for the various skills. With respect to the companies organizing the trainings, we store the fees paid by our company for the training sessions each year.*
    - (Add the new tables to the database diagram and enter some test data)
    - SOLUTION: train_tables.sql[2]

[2] For the solutions of the students' test problems please contact the author

Querying

- We review the basics of Structured Query Language (SQL) querying like selecting, grouping, joining. Example queries:
    - Value of each order
    - Minimum and maximum quantities sold for each product on a yearly basis
    - Which employee sold the most pieces of the most popular product in 1998?

```
--      Value of each order
select o.orderid, o.orderdate,
    str(sum((1-discount)*unitprice*quantity), 15, 2) as order_value,
    sum(quantity) as no_of_pieces,
    count(d.orderid) as no_of_items
from orders o inner join [order details] d on o.orderid=d.orderid
group by o.orderid, o.orderdate
order by sum((1-discount)*unitprice*quantity) desc

--      Quantities sold for each product on a yearly basis
select p.ProductID, p.ProductName, year(o.orderdate), SUM(quantity) as quantity
from orders o inner join [order details] d on o.orderid=d.orderid
inner join Products p on p.ProductID=d.ProductID
group by p.ProductID, p.ProductName, year(o.orderdate)
order by p.ProductName

--      Which employee sold the most pieces of the most popular product in 1998?
select top 1 u.titleofcourtesy+' '+u.lastname+' '+ u.firstname +' ('+u.title +')'  as name,
    sum(quantity) as pieces_sold,
    pr.productname as productname
from orders o inner join [order details] d on o.orderid=d.orderid
    inner join employees u on u.employeeid=o.employeeid
    inner join products pr on pr.productid=d.productid
where year(o.orderdate)=1998 and d.productid =
    (select top 1 p.productid
    from products p left outer join [order details] d on p.productid=d.productid
    group by p.productid
    order by count(*) desc)
group by u.employeeid, u.titleofcourtesy, u.title, u.lastname, u.firstname,
pr.ProductID,pr.productname
order by sum(quantity) desc
```

- For more examples and a systematic overview of SQL querying, see the Appendix
- Further reading on querying:
    - https://docs.microsoft.com/en-us/sql/t-sql/queries/queries
- PRACTICE: using the tables implemented in the first practice, implement the following queries
    - What are the missing skills for Mrs. Peacock?
    - Are there any sessions in the future that are still required for Peacock to attend?
    - What is the first and last training date and the average duration of trainings in days?
    - Which employee has the most skills with an exam result above ‘fail’?
    - What is the total fee paid for all training sessions in which our most skilled employee (see above) participated?

- Which required skill(s) have not yet been addressed by any training session?
- SOLUTION: train_solution.sql

Programming

- Besides SQL, procedural transactional logic can be implemented in the scripting language T-SQL, and it can be run and stored on the server side
  - Pros and cons for server side business logic
    - ✓ Simple architecture
    - ✓ Technological neutrality
    - ✓ Data safety
    - ✓ Manageability
    - ✓ Efficiency
    - ✓ Readable code
    - ✗ Low level
    - ✗ Poor software technological support
    - ✗ Expensive scalability
  - The bottom line is that the part of business logic that involves simple, set-based operations on *large volumes* of *structured* data are best implemented and managed on the database server in the form of stored procedures, functions, triggers and jobs. Procedurally sophisticated parts of the business logic that call for a high level, object-oriented programming environment, should be implemented on an application server.
  - The elements of server side programmability
    - Special SQL keywords for control flow: DECLARE, SET, BEGIN/END, IF/ELSE, WHILE/BREAK/CONTINUE, RETURN, WAITFOR/DELAY/TIME, GOTO
    - Error handling: TRY/CATCH/THROW/RAISERROR
    - Objects supporting programmability: CREATE PROCEDURE/FUNCTION/TRIGGER
    - Transactional support: BEGIN/COMMIT/ROLLBACK TRANSACTION
  - Below is a simple example of a T-SQL script and its stored procedure equivalent. The similar user defined function can be used in a SELECT statement.

```
--a simple script that demonstrates the elements of T-SQL
--we search for an emplyee, and if we find a single matching record,
--we increase the salary of the employee by 10%
set nocount on
declare @name nvarchar(20), @address nvarchar(max), @res_no int, @emp_id int
set @name='Fuller'
select @res_no=count(*) from Employees where LastName like @name + '%'
if @res_no=0 print 'No matching record.'
else if @res_no>1 print 'More than one matching record.'
else begin  --a single hit
	select @address=Country+', '+City+' '+Address, @emp_id=EmployeeID
		from Employees where LastName like @name
```

```
        print 'Employee ID: ' + cast(@emp_id as varchar(10)) + ', address: ' + @address
        update Employees set salary=1.1*salary where EmployeeID=@emp_id
        print 'Salary increased.'
end
go

--wrap it in a stored procedure
create procedure sp_increase_salary @name nvarchar(40)
as
set nocount on
declare @address nvarchar(max), @res_no int, @emp_id int
select @res_no=count(*) from Employees where LastName like @name + '%'
if @res_no=0 print 'No matching record.'
else if @res_no>1 print 'More than one matching record.'
else begin  --a single hit
        select @address=Country+', '+City+' '+Address, @emp_id=EmployeeID
               from Employees where LastName like @name
        print 'Employee ID: ' + cast(@emp_id as varchar(10)) + ', address: ' + @address
        update Employees set salary=1.1*salary where EmployeeID=@emp_id
        print 'Salary increased.'
end
go
--test
select Salary from Employees where LastName like 'Fuller%'
exec sp_increase_salary 'Fuller'
select Salary from Employees where LastName like 'Fuller%'

--a scalar valued function that returns the salary of a person or 0 if the person is not found
go
create function fn_salary (@name nvarchar(40)) returns money as
begin
        declare @salary money, @res_no int
        select @res_no=count(*) from Employees where LastName like @name + '%'
        if @res_no <> 1 set @salary=0
        else select @salary=Salary from Employees where LastName like @name  + '%'
        return @salary
end
go
--test
select [your user name].fn_salary('Fuller') as salary
```

- Note that a **stored procedure** can return multiple record sets is it contains multiple SELECT statements without variable assignment. Parameters passed by value as shown in the example above are INPUT type parameters. Stored procedures may also return scalar values in OUTPUT parameters (not shown in the example). Stored procedures may also call other stored procedures or functions, therefore they can be used to implement complex business logic on the DB server.
- A **user defined function** differs from a stored procedure in that it must have a single return value, the type of which may be scalar like money or table. The last statement of a function must be a RETURN. The advantage of user defined functions over stored procedures is that a function may be called from inside a SELECT statement like any other built-in SQL function like DATEDIFF etc., thus it can add a lot to the flexibility of static SQL queries.
- PRACTICE
    - Using the training queries, create a stored procedure that returns the missing skills for an employee name passed as a parameter. The stored procedure should return a table with

a single field containing the missing skills. If the employee cannot be identified, return an error message and no table.

  - Using the training queries, create a table-valued function that returns the missing skills for an employeeID, in the form of a table. Hint: use ‘returns table’ in the function specification.

- In order to demonstrate a more realistic business process, here is an example script for making a new Northwind order that contains a single order item. The scenario is that the company office receives an urgent order from a valued customer over the phone. Such a process is a typical business transaction.

```
--variables
declare @prod_name varchar(20), @quantity int, @cust_id nchar(5) --we receive the textual
customer id over the phone
declare @status_message nvarchar(100),  @status int --the result of the business process
declare @res_no int --No of hits
declare @prod_id int, @order_id int --IDs
declare @stock int --existing product stock
declare @cust_balance money --customers balance
declare @unitprice money --unit price of product

-- parameters
set @prod_name = 'boston'
set @quantity = 10
set @cust_id = 'AROUT'

begin try
       select @res_no = count(*) from products where productname like '%' + @prod_name + '%'
       if @res_no <> 1 begin
              set @status = 1
              set @status_message = 'ERROR: Ambiguous Product name.';
       end else begin
              -- if we find a single product, we look for the key and the stock
              select @prod_id = productID, @stock = unitsInStock from products where
productName like '%' + @prod_name + '%'
              -- is the stock sufficient?
              if @stock < @quantity begin
                     set @status = 2
                     set @status_message = 'ERROR: Stock is insufficient.'
              end else begin
              -- Does the customer have credit?
                     select @cust_balance = balance from customers where customerid =
@cust_id
                                          --if there is no hit, the @cust_balance is null
                                          --there cannot be more than one hit
                     select @unitprice = unitPrice from products where productID = @prod_id -
-no discount
                     if @cust_balance < @quantity*@unitprice or @cust_balance is null begin
                            set @status = 3
                            set @status_message = 'ERROR: Customer not found or balance
insufficient.'
                     end else begin
              -- no more checks, we start the transaction (3 steps)
              -- 1. decrease the balance
                     update customers set balance = balance-(@quantity*@unitprice) where
customerid=@cust_id
              -- 2. new record in the  Orders, Order Details
```

```
                              insert into orders (customerID, orderdate) values (@cust_id,
getdate()) --orderid: identity
                              set @order_id = @@identity  --result of the last identity insert
                              insert [order details] (orderid, productid, quantity, UnitPrice)
--here we make an error
                                     values(@order_id, @prod_id, @quantity, @unitprice) --here
we make an error
--                            insert [order details] (orderid, productid, quantity, UnitPrice,
Discount) --the correct line
--                                   values(@order_id, @prod_id, @quantity, @unitprice, 0) --
the correct line
               -- 3. update product stock
                              update products set unitsInStock = unitsInStock - @quantity where
productid = @prod_id
                              set @status = 0
                              set @status_message = cast(@order_id as varchar(20)) + ' order
processed successfully.'
                       end
               end
       end
       print @status
       print @status_message
end try
begin catch
       print 'OTHER ERROR: '+ ERROR_MESSAGE() + ' (' + cast(ERROR_NUMBER() as varchar(20)) +
')'
end catch
go

--we set parameters for testing
set nocount off
update products set unitsInStock = 900 where productid=40
update customers set balance=1000 where CustomerID='AROUT'
delete [Order Details] where OrderID in (select orderid from Orders where CustomerID='AROUT'
and EmployeeID is null)
delete Orders where CustomerID='AROUT' and EmployeeID is null
--we run the script and then check:
select * from Customers where CustomerID='AROUT'
select * from Products where productid=40
select top 3 * from Orders where CustomerID='arout' order by OrderDate desc

--Seems fine. However we neglected a NOT NULL constraint of the discount field:
--"OTHER ERROR: Cannot insert the value NULL into column 'Discount'"
--Even worse, we still decreased the balance of the customer!

--in a concurrent environment, other errors may manifest as well

--after correction, test the other two branches as well
```

- Further reading on programming:
    - https://docs.microsoft.com/en-us/sql/t-sql/language-elements/control-of-flow
- PRACTICE: using the tables and scripts implemented in the previous practices,
    - Write a script that checks whether an employee needs any of the skills offered by a training session, and if yes, **enroll the employee** for all such sessions.
    - Run the script in a stored procedure
    - SOLUTION: train_solution.sql

Cursors

Cursors can be used for problems for which the procedural row-by-row approach is more suitable than the set-based querying approach.

EXAMPLE for cursor syntax

```
declare @emp_id int, @emp_name nvarchar(50), @i int, @address nvarchar(60)
declare cursor_emp cursor for
    select employeeid, lastname, address from employees order by lastname
set @i=1
open cursor_emp
fetch next from cursor_emp into @emp_id, @emp_name, @address
while @@fetch_status = 0
begin
    print cast(@i as varchar(5)) + ' EMPLOYEE:'
    print 'ID: ' + cast(@emp_id as varchar(5)) + ', LASTNAME: ' + @emp_name + ', ADDRESS: ' +
@address
    set @i=@i+1
    fetch next from cursor_emp into @emp_id, @emp_name, @address
end
close cursor_emp
deallocate cursor_emp
go
--equivalent to this with a SELECT
select 'ID: ' + cast(employeeid as varchar(5)) + isnull(', LASTNAME: ' + lastname, '') +
isnull( ', ADDRESS: ' + address, '')
from employees order by lastname
--or, with a row number
select cast(row_number() over(order by lastname) as varchar(50))+
'. ügynök: ID: ' + cast(employeeid as varchar(5)) + isnull(', LASTNAME: ' + lastname, '') +
isnull( ', ADDRESS: ' + address, '')
from employees
```

PRACTICE: Implement a cursor that iterates the USA customers and prints the number of their respective orders row by row.

Transaction management

- The core transactional concepts
    - We define ‘**transaction**’ as a logically coherent sequence of operations in a business process. ‘Logically coherent’ means that the operations form a semantic unit. Transactions may be **nested** e.g. the transaction of buying a helicopter includes the transaction of the customer identifying herself and the transaction of paying the bill by bank transfer etc.
    - **Atomicity**, **consistency**, **isolation** and **durability** requirements for environments implementing transactions. We violated the atomicity and isolation requirement in our last order processing example.
    - There are **implicit** and **explicit** (programmed) **transactions**. Implicit transactions are all SQL DML statements.
    - Transactions in T-SQL are programmed with the BEGIN/COMMIT/ROLLBACK TRANSACTION statements. The transaction consists of all statements between the BEGIN TRANSACTION and a COMMIT TRANSACTION or ROLLBACK TRANSACTION statement. COMMIT closes the transaction and frees all the resources like table locks etc. that were

used by the server for transaction management. ROLLBACK does the same after undoing all changes performed by all the statements of the transaction. For this to be possible, the server uses a sophisticated logging mechanism called the **Write-Ahead Log** (WAL). If not truncated or backed up, the transactional log may grow bigger than the database itself.

- In MS SQL Server, if XACT_ABORT is ON and one of the transaction's statements causes an error, the server stops executing the transaction and performs an automatic ROLLBACK.

EXAMPLE

```
--simple demo for atomicity, with xact_abort on
set xact_abort off
delete t2
go
begin tran
       insert t2 (id, t1_id) values (10, 1)
       insert t2 (id, t1_id) values (11, 2) --foreign key constraint violation
       insert t2 (id, t1_id) values (12, 3)
commit tran
go
--"The INSERT statement conflicted with the FOREIGN KEY constraint ..." etc
select * from t2
id     t1_id
10     1
12     3
--atomicity was not preserved
set xact_abort on
delete t2
go
begin tran
       insert t2 (id, t1_id) values (10, 1)
       insert t2 (id, t1_id) values (11, 2) --foreign key constraint violation
       insert t2 (id, t1_id) values (12, 3)
commit tran
go
--"The INSERT statement conflicted with the FOREIGN KEY constraint ..." etc
select * from t2
id     t1_id
--atomicity was preserved
```

- **Nested transactions** technically mean multiple BEGIN TRANSACTION statements. A single ROLLBACK will roll back all transactions that have been begun, see example below

```
begin tran
       print @@trancount  --1
       begin tran
              print @@trancount  --2
       commit tran
       print @@trancount  --1
commit tran
print @@trancount  --0

begin tran
       print @@trancount  --1
       begin tran
              print @@trancount  --2
rollback tran
print @@trancount  --0
```

- It is a serious **programming error** not to close a transaction by either a COMMIT or a ROLLBACK. An unterminated transaction will continue consuming server resources and will eventually cripple the system. The **@@TRANCOUNT** global variable may be used to check whether the current connection has an unterminated transaction.

EXAMPLE: In order to correct the shortcomings of the example order processing script, we wrap it into a stored procedure, and add TRY/CATCH error handling and transactional support.

```
go
create procedure sp_new_order
@prod_name nvarchar(40), @quantity smallint, @cust_id nchar(5)
as
set nocount on
set xact_abort on
--variables
declare @status_message nvarchar(100),  @status int --the result of the business process
declare @res_no int --No of hits
declare @prod_id int, @order_id int --IDs
declare @stock int --existing product stock
declare @cust_balance money --customers balance
declare @unitprice money --unit price of product
begin tran
begin try
        select @res_no = count(*) from products where productname like '%' + @prod_name + '%'
        if @res_no <> 1 begin
                set @status = 1
                set @status_message = 'ERROR: Ambiguous Product name.';
        end else begin
                -- if we find a single product, we look for the key and the stock
                select @prod_id = productID, @stock = unitsInStock from products where
productName like '%' + @prod_name + '%'
                -- is the stock sufficient?
                if @stock < @quantity begin
                        set @status = 2
                        set @status_message = 'ERROR: Stock is insufficient.'
                end else begin
                -- Does the customer have credit?
                        select @cust_balance = balance from customers where customerid =
@cust_id
                                                --if there is no hit, the @cust_balance is null
                                                --there cannot be more than one hit
                        select @unitprice = unitPrice from products where productID = @prod_id -
-no discount
                        if @cust_balance < @quantity*@unitprice or @cust_balance is null begin
                                set @status = 3
                                set @status_message = 'ERROR: Customer not found or balance
insufficient.'
                        end else begin
                -- no more checks, we start the transaction (2 steps)
                -- 1. decrease the balance
                        print 'Processing order...'
                                update customers set balance = balance-(@quantity*@unitprice)
where customerid=@cust_id
                -- 2. new record in the  Orders, Order Details
                                insert into orders (customerID, orderdate) values (@cust_id,
getdate()) --orderid: identity
                                set @order_id = @@identity  --result of the last identity insert
                                insert [order details] (orderid, productid, quantity, UnitPrice)
values(@order_id, @prod_id, @quantity, @unitprice) --here we make an error
```

```
              --            insert [order details] (orderid, productid, quantity, UnitPrice, 
Discount) values(@order_id, @prod_id, @quantity, @unitprice, 0) --the correct line
                              set @status = 0
                              set @status_message = 'Order No. ' + cast(@order_id as 
varchar(20)) + ' processed successfully.'
                      end
              end
       end
       print 'Status: ' + cast(@status as varchar(50))
       print @status_message
       if @status = 0 commit tran else begin
              print 'Rolling back transaction'
              rollback tran
       end
end try
begin catch
       print 'OTHER ERROR: '+ ERROR_MESSAGE() + ' (' + cast(ERROR_NUMBER() as varchar(20)) + 
')'
       print 'Rolling back transaction'
       rollback tran
end catch
go

--test
--we set parameters for testing
set nocount off
update customers set balance=1000 where CustomerID='AROUT'
delete [Order Details] where OrderID in (select orderid from Orders where CustomerID='AROUT' 
and EmployeeID is null)
delete Orders where CustomerID='AROUT' and EmployeeID is null
--we run the stored proc
exec sp_new_order 'boston', 10, 'Arout'
--check the results:
select * from Customers where CustomerID='AROUT' --should be 816
select top 3 * from Orders o inner join [Order Details] od on o.OrderID=od.OrderID
       where CustomerID='arout' order by OrderDate desc --should see the new item
select @@trancount --must be 0
```

    - Test the above stored procedure for various errors: programming errors and logical errors like insufficient stock. Check the integrity of the database. Make sure that the transactional support prevents any serious errors.

- In order to ensure isolation, the server uses locks on rows (records), ranges or tables. An **Isolation Level** is a locking strategy enforced by the server. The main lock types on MS SQL Server are Read (shared), Write (exclusive) and Update. Below are the 4 ANSI standard isolation levels, though current database technologies support more than just these 4.
    - READ UNCOMMITTED: no locking
    - READ COMMITTED: locks removed after the completion of the SQL statement
    - REPEATABLE READ: locks that were granted for the transaction are kept until the end of the transaction
    - SERIALIZABLE: other transactions cannot insert records into a table for which a transaction has a row or range lock, phantom read is not possible

```
--simple demo for isolation: the webshop case
create table test_product(id int primary key, prod_name varchar(50) not null, sold 
varchar(50), buyer varchar(50))
```

```
insert test_product(id, prod_name, sold) values (1, 'car', 'for sale')
insert test_product(id, prod_name, sold) values (2, 'horse', 'for sale')
go
select * from test_product
update test_product set sold='for sale', buyer=null where id=2
go
set tran isolation level read committed --the default
go
begin tran
declare @sold varchar(50)
select @sold=sold from test_product where id=2
if @sold='for sale' begin
    waitfor delay '00:00:10' --now we are performing the bank transfer
    update test_product set sold='sold', buyer='My name' where id=2
    print 'sold successfully'
end else print 'product not available'
commit tran
go
--we run the above transaction concurrently in two query editors
--the second script:
set tran isolation level read committed
go
begin tran
declare @sold varchar(50)
select @sold=sold from test_product where id=2
if @sold='for sale' begin
    waitfor delay '00:00:10' --now we are performing the bank transfer
    update test_product set sold='sold', buyer='Your name' where id=2 --note the diff
    print 'sold successfully'
end else print 'product not available'
commit tran
go
--check what happens:
select * from test_product
id      prod_name       sold            buyer
1       car                     for sale        NULL
2       horse           sold            Your name
--The horse was sold successfully to two customers, but only Your name will receive it. Very
awkward.
update test_product set sold='for sale', buyer=null where id=2
--Now try the same with set tran isolation level repeatable read
--"Transaction (Process ID 53) was deadlocked on lock resources with another process and has
been chosen as the deadlock victim. Rerun the transaction."
--No logical error. Only one horse is sold.

--Conclusion: be careful to select the right isolation level.
```

- Further reading on transaction management:
  - https://docs.microsoft.com/en-us/sql/t-sql/language-elements/control-of-flow
- PRACTICE: Add transactional support to your own training management stored procedure and test it for various errors.

2. Loose coupling based on triggers and jobs

Problem scenario

The new orders are stored in the Orders and Orderitems tables by a third party management and trading application that has no open API, or for any other reason refuses to generate service level events. Therefore, in the current order processing workflow at the Northwind Traders Ltd Co. the trading department communicates with the Shipping and Logistics (SL) division via email (or any other manual messaging system) about the new or changed orders. Our company is responsible for IT support in SL management. The head of SL division prepares the detailed daily work plans every morning for the various units according to the emails received from the trading department. For this, she uses our software tool. Both the trading and the SL use the same Northwind SQL Server database.

We are asked to relieve the trading and SL staff from manually writing emails and manually entering data from emails into another application by automating the order processing workflow as much as possible.

Solution

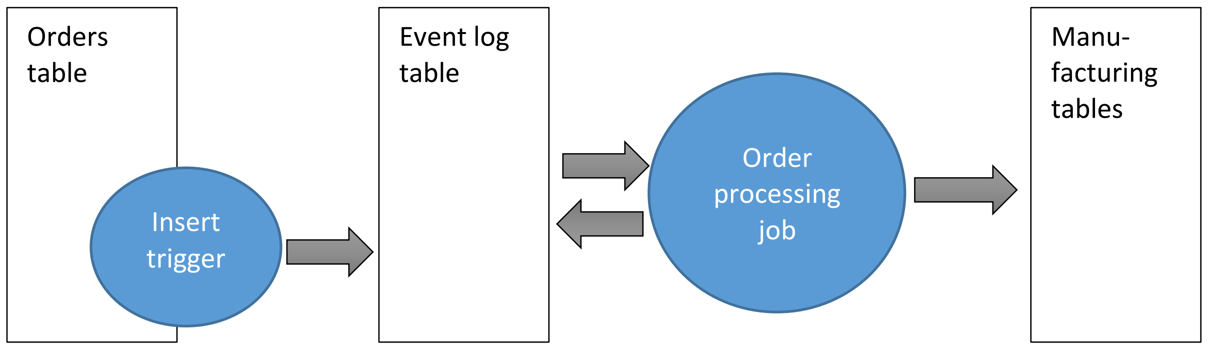
Since the trading system is a 'black box', we must rely on database level events. Every time an order is created or modified, we must run the required (rather complex) logic on the database that creates or changes the required records in the SL tables like Products. Thus both the writing and the processing of emails will be unnecessary.

It is, however, vital that our solution *should not at the least interfere* with the trading system. It cannot significantly slow down the order saving process, nor may any error that may occur while processing an order event on the SL side be propagated back to the trading system.

For this reason, we use the *loose coupling* concept. We only log the INSERT and UPDATE events on orders via a trigger in a special table, and process these events in batches executed by a scheduled job. The job also keeps track of the state and results of the processing of each event. Since the processing of the event is performed out-of-process, a processing error does not manifest as an error in the trading system.

*Note: a **trigger** is a special stored procedure that is invoked automatically by the database management system upon database events like table INSERT, UPDATE or DELETE.*

System overview:



A short overview on triggers

Triggers are special procedures stored on the server and run automatically when a pre-defined condition is satisfied. SQL Server supports the following types of triggers with respect to the trigger event:

- DML triggers (table level triggers) that are executed when a DELETE, INSERT or UPDATE action is performed on a table
- DDL triggers (database level triggers) that are fired when the schema of the database changes e.g. a table is created
- Logon triggers (server level triggers) that are fired after the authentication phase of logging in finishes

We focus now on DML triggers. The definition of the trigger includes the target table, the trigger event (DELETE, INSERT or UPDATE) and the mode of operation. SQL server supports the following operational modes:

- AFTER: fired after the successful execution of the specified SQL statement. This means that all eventual check and other constraints and cascade updates/deletes associated with the DML statement have executed successfully. We can place multiple triggers on the same object, even of the same type, like two INSERT triggers. In this case, the order of execution can be influenced by setting trigger properties.
- INSTEAD OF: the DML statement is not executed at all, only the trigger.

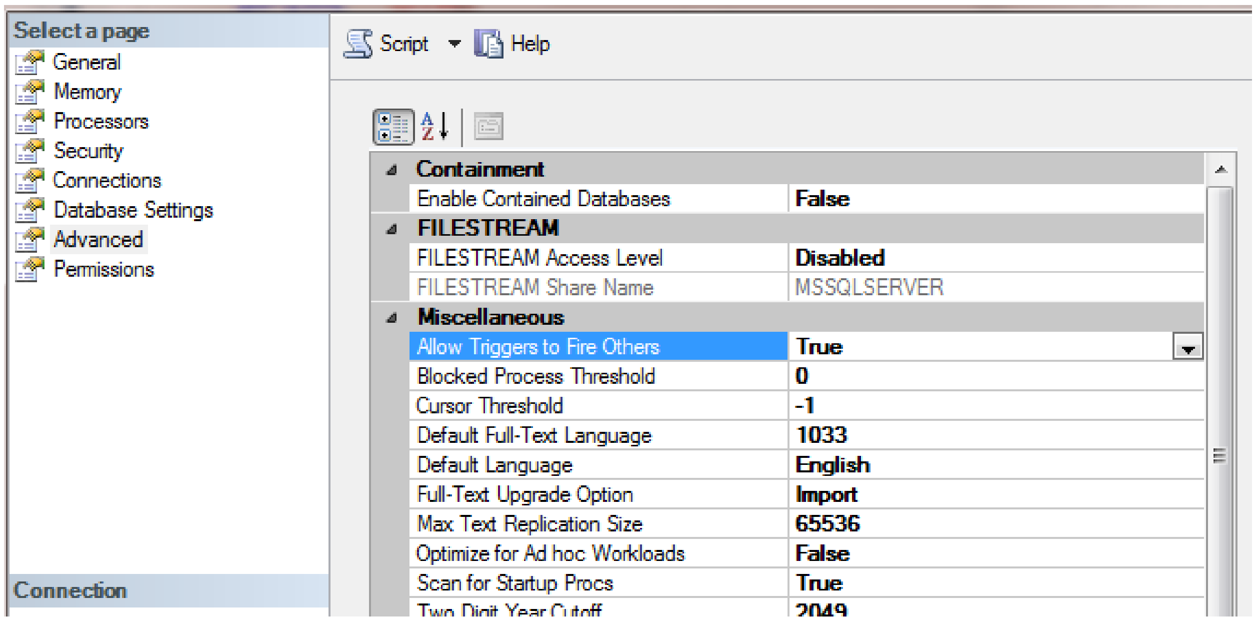
The records modified by the DML statement can be accessed by the trigger code via special logical tables. SQL server provides the following two logical tables:

- **'deleted'**: holds the records deleted form the table in case of a DELETE trigger or the *original* (old) records updated in case of an UPDATE trigger. An update is logically equivalent to a delete followed by an insert. The deleted table is empty in case of an INSERT trigger.
- **'inserted'**: holds the records inserted by an INSERT statement or the *new* records updated by an UPDATE trigger. The inserted table is empty in case of a DELETE trigger.

SQL server also supports the **update**([field name]) function available in INSERT or UPDATE triggers that returns true if the DML statement changed the specified field. The field cannot be a computed column.

If the trigger raises an error, the DML statement is rolled back.

A trigger may run code that invokes other triggers or even the same trigger in a recursive manner, up to 32 levels on MS SQL server. This feature is controlled via the nested triggers server option (see below).



Containment	
Enable Contained Databases	False
FILESTREAM	
FILESTREAM Access Level	Disabled
FILESTREAM Share Name	MSSQLSERVER
Miscellaneous	
Allow Triggers to Fire Others	True
Blocked Process Threshold	0
Cursor Threshold	-1
Default Full-Text Language	1033
Default Language	English
Full-Text Upgrade Option	Import
Max Text Replication Size	65536
Optimize for Ad hoc Workloads	False
Scan for Startup Procs	True
Two Digit Year Cutoff	2049

Cases when the use of a DML trigger is recommended

- Administration functions such as maintaining a log or keeping old values of changed records n backup tables.
- Enforcing data integrity rules that follow from the business logic and that are beyond the scope of simple primary key, foreign key or check constraints. Example in the Northwind database:
  - We do not send heavy packages overseas. Therefore we refuse orders with a freight over 200 that has a ShipCountry not equal to USA. (Can be implemented by an INSERT AFTER or INSERT INSTEAD OF trigger on the Orders table.) Such checks should of course be built into the client software, however, database level integrity enforcement can protect from application errors or hacking.
- Automating business workflow processes. Examples in the Northwind database:
  - We sent an automated email to the customer when the shipping date is decided i.e. when the ShippedDate field of an order is set (UPDATE trigger)
  - We automatically send an order to our gross supplier when the UnitsInStock of a Product drops below the ReorderLevel (UPDATE or INSERT trigger)
  - We automatically update the UnitsInStock field of the Products table when the quantity field in a corresponding Order Detail record changes

PRACTICE: write an update trigger for the Order Details. When the quantity changes, update the UnitsInStock of the product. You can assume that only one Order Details record is updated at a time.

PRACTICE: Assume that in the problem above more than Order Details records are updated at a time.

CAVEAT: the operation of triggers is ‘silent’, and severe problems may result from forgetting about them. For example, if the administrator restores the Order Items tables from a backup copy with an UPDATE statement without first disabling the trigger…

For more examples on SQL server triggers see http://sqlhints.com/2016/02/28/inserted-and-deleted-logical-tables-in-sql-server/

Tight coupling

In the example below we create a new insert trigger on the Orders table that runs long and throws an exception, thus disabling the order saving process. This is exactly what we do NOT want. We implement *loose coupling* instead of *tight coupling*.

```
drop trigger tr_demo_bad
go
create trigger tr_demo_bad on orders for insert as
declare @orderid int
select @orderid=OrderID from inserted
print 'New order ID: ' + cast(@orderid as varchar(50))
waitfor delay '00:00:10' --10 s
select 1/0 --we make an error
go
--test #1:  with both last lines commented out
insert  Orders (CustomerID, OrderDate) values ('AROUT', GETDATE())
--restore table
delete Orders where CustomerID='AROUT' and EmployeeID is null
--test #2: recreate the trigger, with the last lines commented out
insert  Orders (CustomerID, OrderDate) values ('AROUT', GETDATE())
--we have long to wait, but there is no error
--restore table
delete Orders where CustomerID='AROUT' and EmployeeID is null
--test #3: recreate the trigger, with all lines
insert  Orders (CustomerID, OrderDate) values ('AROUT', GETDATE())
--we have long to wait, then we have the message:
'New order ID: 11094
Msg 8134, Level 16, State 1, Procedure tr_demo_bad, Line 6 [Batch Start Line 276]
Divide by zero error encountered.
The statement has been terminated.'
select * from Orders where CustomerID='AROUT' and EmployeeID is null
--no such record, because
--the insert statemant has been rolled back -> we crashed the trading system
```

The loosely coupled system

The idea is that the trigger only saves the events into a log table. We then process the table with a stored procedure.

The log table and the trigger

The trigger uses the virtual tables *inserted* and *deleted*. This trigger can process multi-record INSERTs and UPDATEs.

```
--the log table
go
--drop table order_log
go
create table order_log (
	event_id int IDENTITY (1, 1) primary key ,
	event_type varchar(50) NOT NULL ,
	order_id int NOT NULL ,
	orderitem_id int NULL ,
	status int NOT NULL default(0),
	time_created datetime NOT NULL default(getdate()) ,
        time_process_begin datetime NULL ,
        time_process_end datetime NULL ,
        process_duration as datediff(second, time_process_begin, time_process_end)
)
go
drop trigger tr_log_order
go
create trigger tr_log_order ON Orders for insert, update as
declare @orderid int
select @orderid=orderid from inserted --there can be more then a single record in inserted
print 'OrderID of the LAST record: ' + cast(@orderid as varchar(50))
if update(orderid) begin --if the orderid has changed, then this is an INSERT
        print 'Warning: new order'
        insert order_log (event_type, order_id)  --status, time_created use default
                select 'new order', orderid from inserted
end else if update(shipaddress) or update(shipcity) begin --shipaddress or shipcity has
changed
        print 'Warning: address changed'
        insert order_log (event_type, order_id)
                select 'address changed', orderid from inserted
end else begin  --other change
        print 'Warning: other change'
        insert order_log (event_type, order_id)
                select 'other change', orderid from inserted
end
go

--test #1
insert  Orders (CustomerID, OrderDate) values ('AROUT', GETDATE())
select * from order_log
--we have one new record in the log table

--test #2
insert  Orders (CustomerID, OrderDate) values ('AROUT', GETDATE()), ('HANAR', GETDATE())
select * from order_log
--we have two new records in the log table

--test #3
update Orders set ShipVia = 3 where OrderID in (11097, 11096) --these are the IDs of test #2
select * from order_log
--we have two new records of the type 'other change'

--restore the tables
delete Orders where CustomerID in ('AROUT', 'HANAR') and EmployeeID is null
delete order_log
```

The stored procedure for processing new orders

We expect that the items of a new order are inserted subsequently after the order record is created.

```
--a simple stored procedure that processes a new order
--and returns 0 if all of its items could be committed to the inventory without error
--demonstrating also the use of output parameters
drop proc  sp_commit_new_order_to_inventory
go
create procedure sp_commit_new_order_to_inventory
@orderid int,
@result int output
as
begin try
        update products set unitsInStock = unitsInStock - od.quantity
```

```
	from products p inner join [Order Details] od on od.ProductID=p.ProductID
	where od.OrderID=@orderid
	set @result=0
end try
begin catch
	print '  Inventory error: '+ ERROR_MESSAGE() + ' (' + cast(ERROR_NUMBER() as
varchar(20)) + ')'
	set @result=1
end catch
go

--test
select * from order_log --11097
select * from Products where ProductID=10 --unitsinstock =31
select * from Products where ProductID=9 --unitsinstock =29
insert [Order Details]  (orderid, productid, quantity, UnitPrice, Discount)
values (11097, 9, 10, 30, 0),(11097, 10, 40, 30, 0)  --the second item will cause an error in
sp_commit_new_order_to_inventory
go
declare @res int
exec sp_commit_new_order_to_inventory 11097, @res output
print @res
exec sp_commit_new_order_to_inventory 11096, @res output
print @res
go
--check: no change in unitsinstock (OK)
select * from Products where ProductID=10 --unitsinstock =31
select * from Products where ProductID=9 --unitsinstock =29
```

The stored procedure for processing the event log

Since completely different actions are to be taken depending on the event type, we use a cursor to iterate the order log.

```
--stored procedure for processing the order_log
--drop proc sp_order_process
go
create proc sp_order_process as
declare @event_id int, @event_type varchar(50), @order_id int, @result int
declare cursor_events cursor forward_only static
	for
	select  event_id, event_type, order_id
	from order_log where status=0 --we only care for the unprocessed events

set xact_abort on
set nocount on
open cursor_events
fetch next from cursor_events into @event_id, @event_type, @order_id
while @@fetch_status = 0
begin
	print 'Processing event ID=' + cast(@event_id as varchar(10)) + ', Order ID=' +
cast(@order_id as varchar(10))
	update order_log set time_process_begin=getdate() where event_id=@event_id
	begin tran
	set @result = null
	if @event_type = 'new order' begin
		print '  Processing new order...'
		exec sp_commit_new_order_to_inventory @order_id, @result output
	end else if @event_type = 'address changed' begin
		print '  Processing address changed...'
```

```
               waitfor delay '00:00:01' --we only simulate the processing of other event types
               set @result=0
       end else if @event_type = 'other change' begin
               print '  Processing other change...'
               waitfor delay '00:00:01'
               set @result=0
       end else begin
               print '  Unknown event type...'
               waitfor delay '00:00:01'
               set @result=1
       end

       if @result=0 begin
               print 'Event processing OK'
               commit tran
       end else begin
               print 'Event processing failed'
               rollback tran
       end
       print ''
       update order_log set time_process_end=getdate(),
               status=case when @result=0 then 2 else 1 end
               where event_id=@event_id
       fetch next from cursor_events into @event_id, @event_type, @order_id
end
close cursor_events deallocate cursor_events
go

--teszt
update order_log set status=0
select *from orders where EmployeeID is null
select * from order_log
exec dbo.sp_order_process
select * from order_log

--we get:
Processing event ID=5, Order ID=11097
  Processing new order...
  Inventory error: The UPDATE statement conflicted with the CHECK constraint etc.
Event processing failed

Processing event ID=6, Order ID=11096
  Processing new order...
Event processing OK
```

The scheduled job that calls the event log processor

We implement the job using the SSMS GUI and check its operation in the Job Activity Monitor.

PRACTICE: create a loosely coupling solution that monitors the Products table and orders new supply from the associated Supplier when the UnitsinStock value falls below that specified in the ReorderLevel field.

3. Replication and log shipping

In the loose coupling case study we were in fact implementing a special form of *replication*.

Replication concepts and architecture

Replica means a copy of the original. In database technology, replication is used to automate the copying and merging of data from or to multiple sources. The components of the replication metaphor are as follows.

- The **publisher** is the entity (a database server) that has data to be shared. Such data is organized into **publications**. Each publication contains one or more **articles**. The articles can be tables or parts of tables, stored procedures or other database objects.
- The **subscriber** is the entity that subscribes to publications. It can be the same database server as the publisher or another server. Several subscribers, possibly on different servers, may subscribe for the same publication.
- The **subscription** may have various modalities with respect to the way and scheduling of copying. It can also include filtering the data or on-the-fly data transform steps. It can be a **push** or a **pull** subscription. The pull subscriptions are created at, and scheduled by, the subscriber.

The main types and application scenarios of replication are as follows.

- **Snapshot** replication. After an initial snapshot of the articles, the copied objects will be dropped and re-created on the subscriber each time the data is refreshed, regardless of whether there was any change in the publication. Applicable for reporting parts of an OLTP database off to a data warehouse or a reporting server, scheduled out of office hours e.g. sending data generated during the day overnight ('point in time reports'). Since several subscriptions may point to the same destination database, replication can be used as an ETL (extract-transform-load) mechanism if SQL Server technology is used by all publishers. CAVEAT: due to the drop/re-create mechanism, objects at the subscriber may be temporarily inaccessible. The latency of the data must also be tolerated. All other replication types below are initialized with a snapshot.
- **Transactional** replication. It moves only the data that has been changed, and it can be configured for near real time data synchronization. A primary key on the replicated tables is needed. Applicable when considerable latency is a problem and when we do not want to move unchanged data. An example is the off-site branches of a company that have their own local servers holding only parts of the central database relevant for their operation. Such an architecture improves site autonomy and robustness of the database system.
- **Merge** replication. In this scheme the subscribers may themselves generate changes to the data and there is a mechanism in place that distributes these changes to all parties and merges them into a consistent database. The merging process may also involve conflict resolution. In order to identify records across multiple servers, the tables must have a field of UNIQUEIDENTIFIER data type with ROWGUIDCOL[3] property. A typical scenario is the case of traveling businesspeople who are not always connected the central database. The changes they make on their local database is merged automatically with others' changes automatically.

The type of the replication is always determined by the publication.

[3] Works like an identity column without a seed and with globally unique values

Replication technology is based on scheduled jobs and, in the case of merge replication, triggers on published articles.

Replication is **not recommended** when an exact copy of a whole database is to be maintained on a remote server to improve availability and reliability, because log shipping and the Always-On technology of SQL server 2012 and later offer a simpler and more robust solution.

CAVEAT: though replication prepares multiple copies of the data, it is not a replacement for backups and disaster recovery planning.

There are three **server roles** in replication, the *publisher*, the *distributor* and the *subscriber*. All three roles may be taken by the same server instance when a local database is replicated into another local database, or by different instances. In more realistic setups, the distributor role is taken by another server to offload the publisher. The **distributor** is responsible for storing the changed data in a shared folder and a distribution database and forwarding the data to the subscribers. A publisher may have only one distributor, but a distributor may serve several subscribers.

In **bi-directional** or **updatable** replication the subscriber may be allowed to make changes on the publisher as well.

SQL server implements replication functionality with various **agents**. These agents are jobs running under the supervision of SQL Server Agent.

- **The Snapshot agent** generates the snapshot and stores it in the **snapshot folder** at the distributor. The agent uses the bcp (bulk copy) utility to copy the articles of the publication.
- **The Distribution agent**. In snapshot replication this agent applies the snapshot to the subscriber and in transactional replication it runs the transactions held in the **distribution database** on the subscriber. The distribution database is a system database on the distributor, therefore you can find it in the System Databases group. This agent runs at the subscriber for pull subscriptions and it runs at the publisher for push subscriptions.
- **The Log reader agent** reads the transaction log at the publisher and copies the relevant transactions from the log to the distribution database. It is used only in transactional replication. There is a separate agent for each database published.
- **The Queue reader agent** copies changes made by the subscribers to the publisher in an *updatable* or *bi-directional* transactional replication.
- **The Merge agent** merges incremental changes that occur at both the subscriber and the publisher in merge replication. Detecting changes is based on triggers. The merge agent is not used in transactional replication.

Except for a pull subscription, all agents run at the distributor.

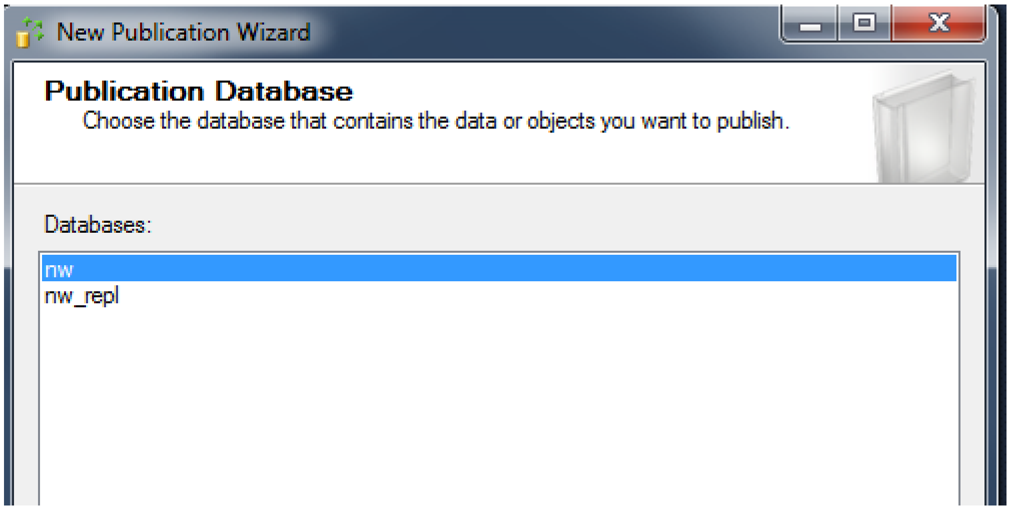
PRACTICE: snapshot replication

First of all configure the test environment. For the replication examples to work as expected, you need three ‘named’ MS SQL Server instances installed on the same server machine. They should be named Principal, Secondary and Third.

Scenario: we want to replicate the orders of the American customers to another database on the same server (Principal) to refresh the reporting data warehouse overnight. We choose snapshot replication.

Creating the publication

1. Connect to the Principal server and create a new database called nw. Select the FULL recovery mode. Run the create dump of the Northwind database.
2. Create another empty database called nw_repl, also on the Principal server.
3. Start the New publication wizard and select nw as the publication database:



New Publication Wizard

Publication Database

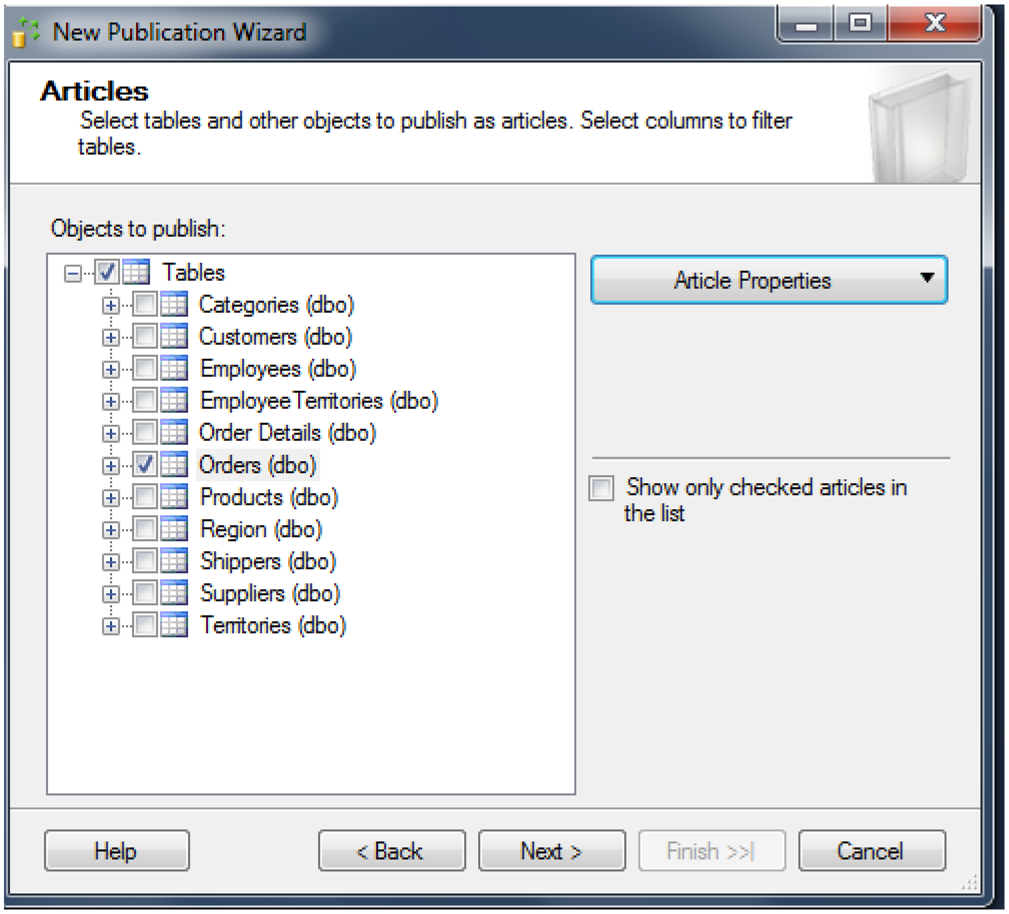
Choose the database that contains the data or objects you want to publish.

Databases:

nw

nw_repl

4. On the next panel, select Snapshot publication, then select the Orders table as the single article of the publication:



Articles

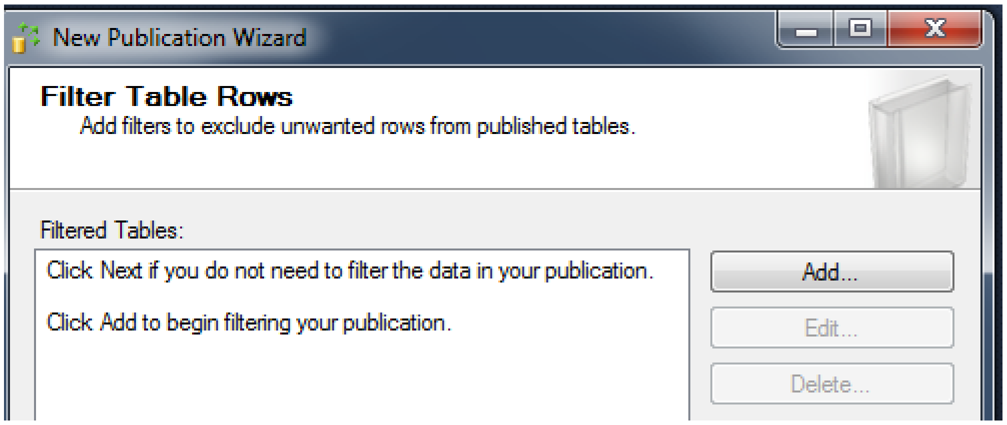
Select tables and other objects to publish as articles. Select columns to filter tables.

Objects to publish:

- [✓] Tables
  - [ ] Categories (dbo)
  - [ ] Customers (dbo)
  - [ ] Employees (dbo)
  - [ ] EmployeeTerritories (dbo)
  - [ ] Order Details (dbo)
  - [✓] Orders (dbo)
  - [ ] Products (dbo)
  - [ ] Region (dbo)
  - [ ] Shippers (dbo)
  - [ ] Suppliers (dbo)
  - [ ] Territories (dbo)

[ ] Show only checked articles in the list

5. On the Filter table dialog, choose Add



Filter Table Rows

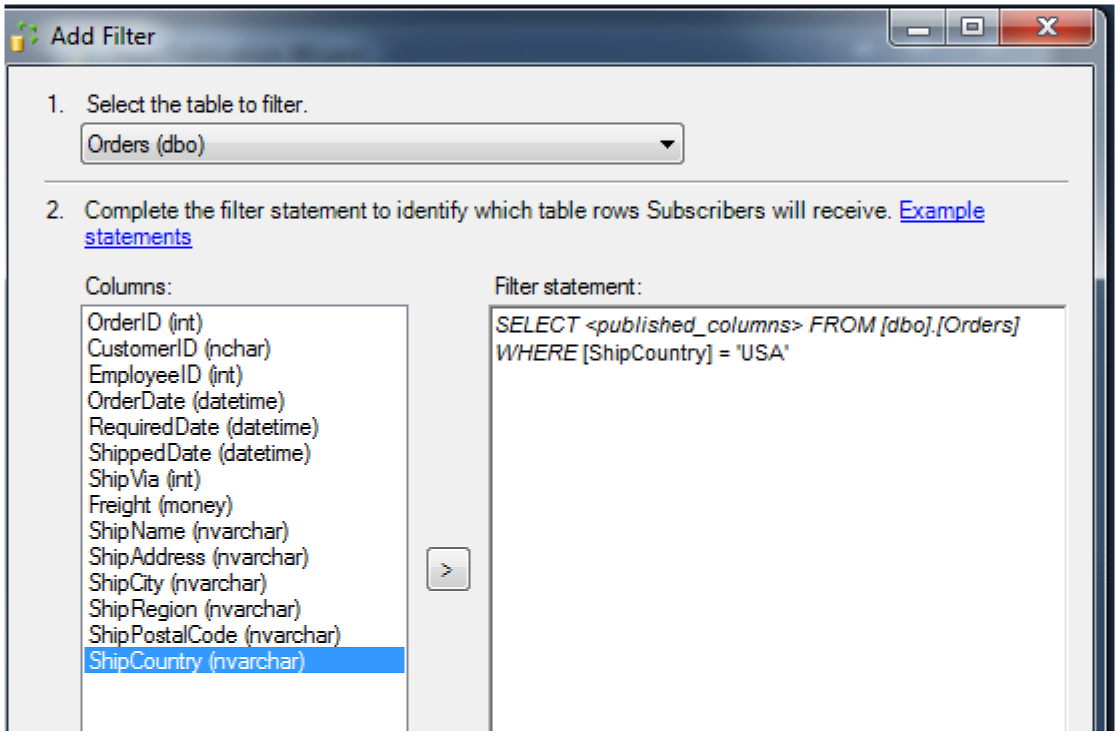
Add filters to exclude unwanted rows from published tables.

Filtered Tables:

Click Next if you do not need to filter the data in your publication.

Click Add to begin filtering your publication.

6. Complete the filter statement to filter out American customers



1. Select the table to filter.

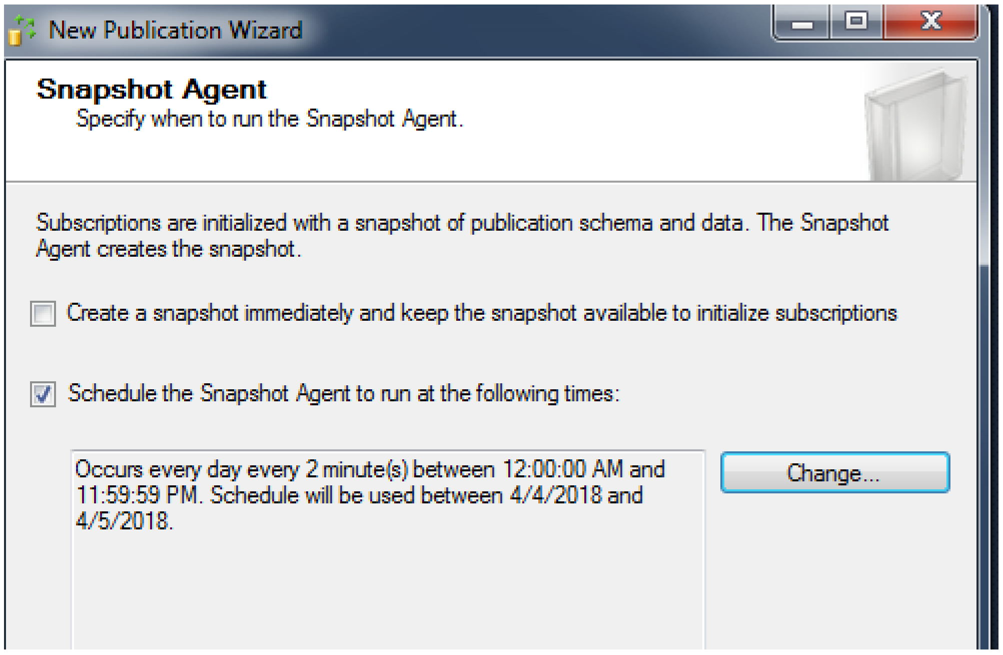
Orders (dbo)

2. Complete the filter statement to identify which table rows Subscribers will receive.

OrderID (int)
CustomerID (nchar)
EmployeeID (int)
OrderDate (datetime)
RequiredDate (datetime)
ShippedDate (datetime)
ShipVia (int)
Freight (money)
ShipName (nvarchar)
ShipAddress (nvarchar)
ShipCity (nvarchar)
ShipRegion (nvarchar)
ShipPostalCode (nvarchar)
ShipCountry (nvarchar)

```
SELECT <published_columns> FROM [dbo].[Orders]
WHERE [ShipCountry] = 'USA'
```

7. Specify that the snapshot agent should run every two minutes by selecting Change on the next panel. Note: *We use this short time interval* ***only for demo*** *purposes. The agent runs the* bcp *utility which places a lock on the whole table until it finishes the copying in order to guarantee data consistency. This means the blocking of all other transactions that may wish to modify the table. Snapshot generation in production systems should be scheduled considering performance implications.*



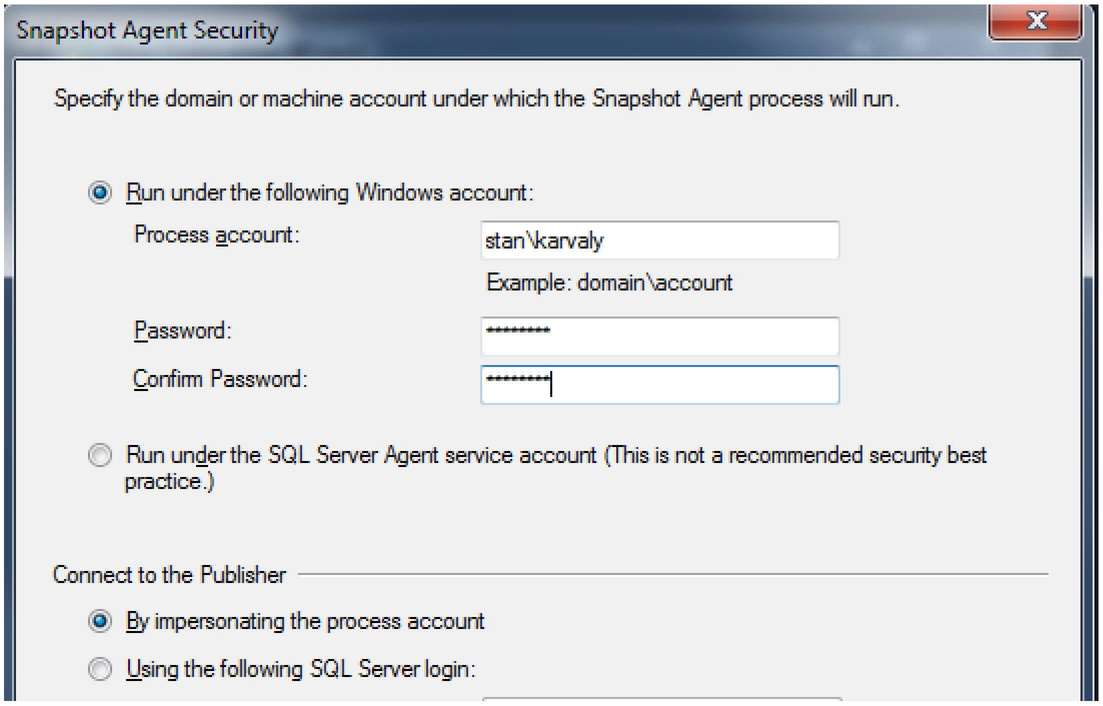
Subscriptions are initialized with a snapshot of publication schema and data. The Snapshot Agent creates the snapshot.

☐ Create a snapshot immediately and keep the snapshot available to initialize subscriptions

☑ Schedule the Snapshot Agent to run at the following times:

Occurs every day every 2 minute(s) between 12:00:00 AM and 11:59:59 PM. Schedule will be used between 4/4/2018 and 4/5/2018.

8. We now have to specify agent security. On the Security settings tab, set your own user credentials and impersonate process account. This is the simplest way of ensuring that the snapshot agent will have write permission to the snapshot folder. **Note**: you might wonder why the SQL Server Agent service or the SQLSERVER service uses a low privilege non-administrator Windows account. The reason for this is that in this way an attacker who has successfully cracked the DBMS has less chance to corrupt the whole server.


Snapshot Agent Security

Specify the domain or machine account under which the Snapshot Agent process will run.

(●) Run under the following Windows account:

Process account: stan\karvaly

Example: domain\account

Password: *********

Confirm Password: *********

( ) Run under the SQL Server Agent service account (This is not a recommended security best practice.)

Connect to the Publisher

(●) By impersonating the process account

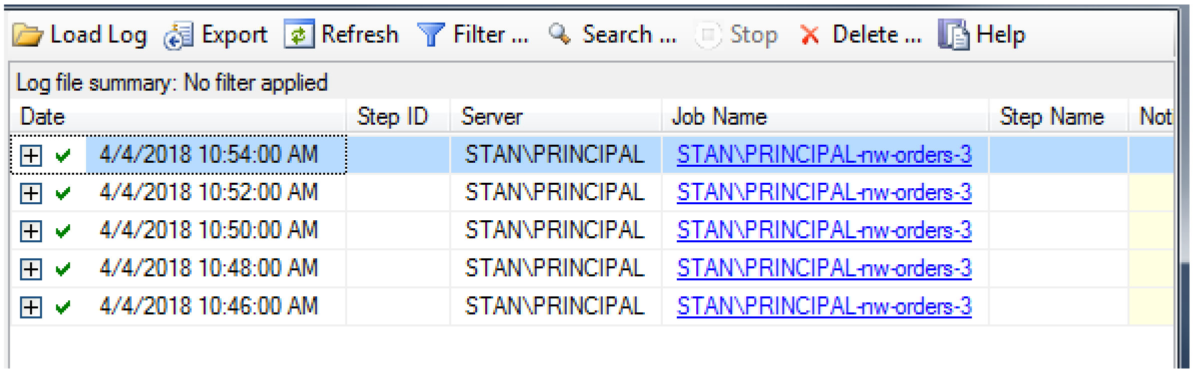
( ) Using the following SQL Server login:

9. On the next panel, select create the publication and name it ‘orders’.

Checking the publication

The new publication appears under Local publications. The snapshot folders are created at C:\Program Files\Microsoft SQL Server\MSSQL12.PRINCIPAL\MSSQL\repldata\unc\STAN$PRINCIPAL_NW_ORDERS, but no actual snapshot was not generated because no subscriptions needed initialization yet.

Check the new job that appears under SQL Server Agent jobs. The job history shows that the agent is run periodically as configured.

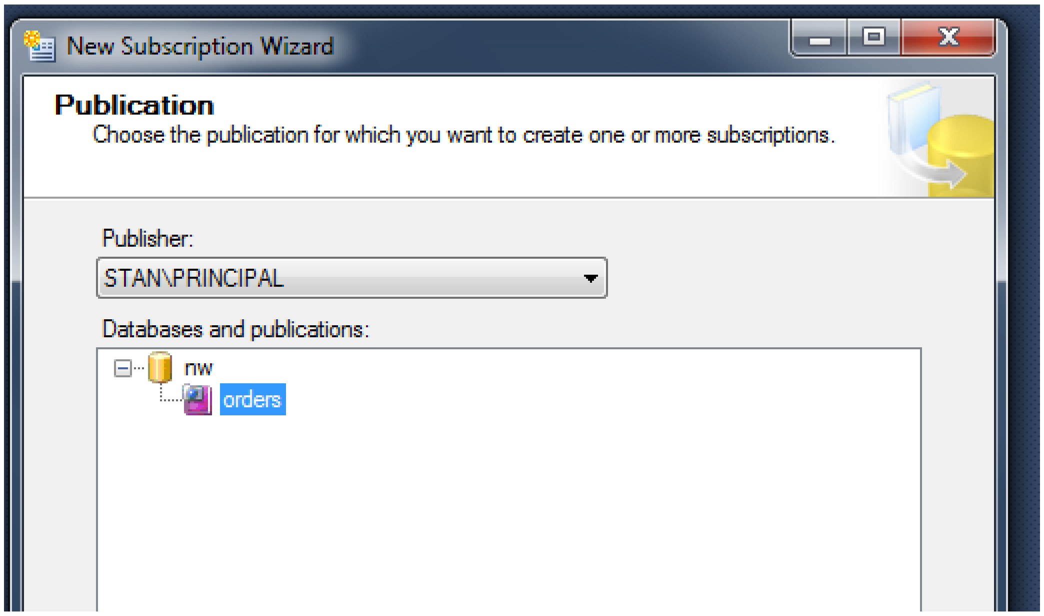


Log file summary: No filter applied

Date	Step ID	Server	Job Name	Step Name	Not
✓ 4/4/2018 10:54:00 AM		STAN\PRINCIPAL	STAN\PRINCIPAL-nw-orders-3		
✓ 4/4/2018 10:52:00 AM		STAN\PRINCIPAL	STAN\PRINCIPAL-nw-orders-3		
✓ 4/4/2018 10:50:00 AM		STAN\PRINCIPAL	STAN\PRINCIPAL-nw-orders-3		
✓ 4/4/2018 10:48:00 AM		STAN\PRINCIPAL	STAN\PRINCIPAL-nw-orders-3		
✓ 4/4/2018 10:46:00 AM		STAN\PRINCIPAL	STAN\PRINCIPAL-nw-orders-3		

Creating a push subscription

1. Start the new subscription wizard from the pop-up menu on the orders publication. Select the orders publication as the source



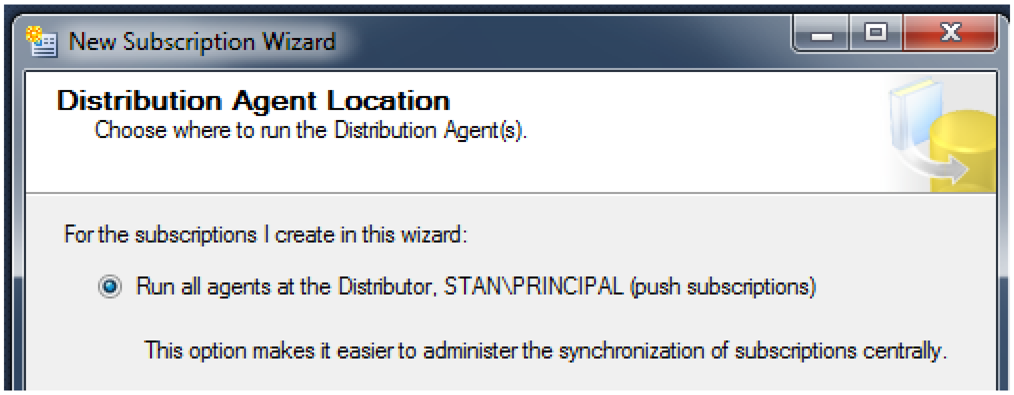
Publication

Choose the publication for which you want to create one or more subscriptions.

Publisher: STAN\PRINCIPAL

Databases and publications:
- nw
  - orders

2. On the next panel, choose Run all agents at the distributor for push subscription



Distribution Agent Location

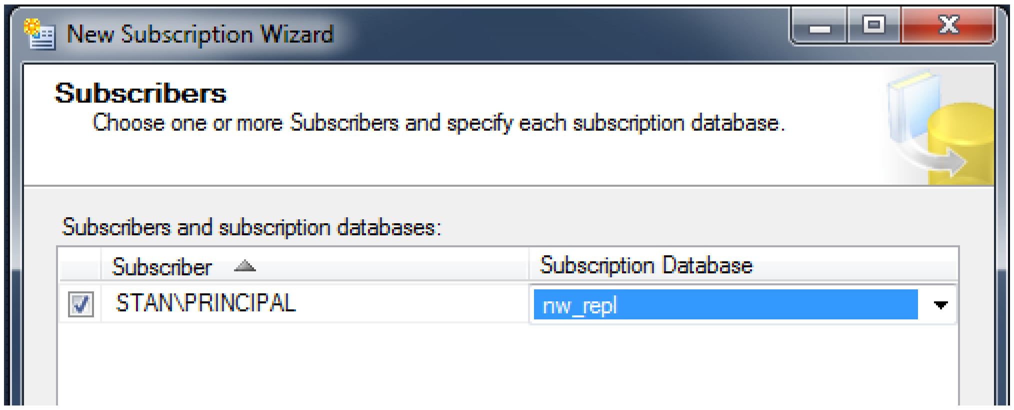
Choose where to run the Distribution Agent(s).

For the subscriptions I create in this wizard:

(●) Run all agents at the Distributor, STAN\PRINCIPAL (push subscriptions)

This option makes it easier to administer the synchronization of subscriptions centrally.

3. Specify the same server as the subscriber and the nw_repl as the subscription database



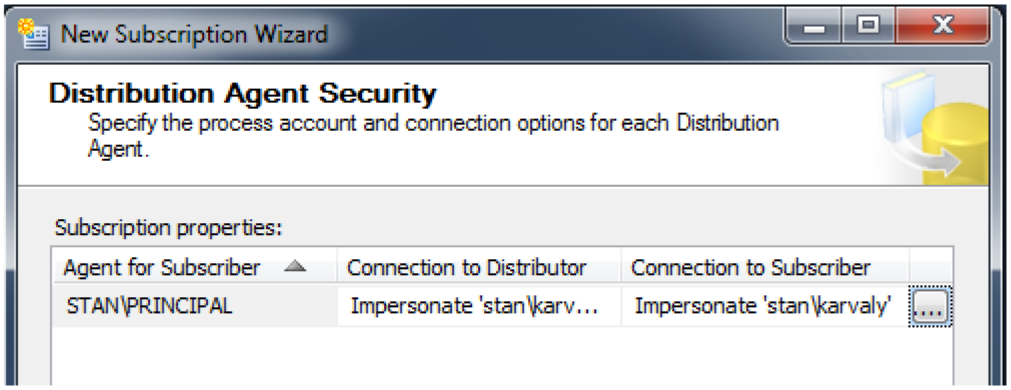
Subscribers

Choose one or more Subscribers and specify each subscription database.

Subscribers and subscription databases:

	Subscriber	Subscription Database
✓	STAN\PRINCIPAL	nw_repl

4. Set the security of the distribution agent the same way as the snapshot agent



New Subscription Wizard

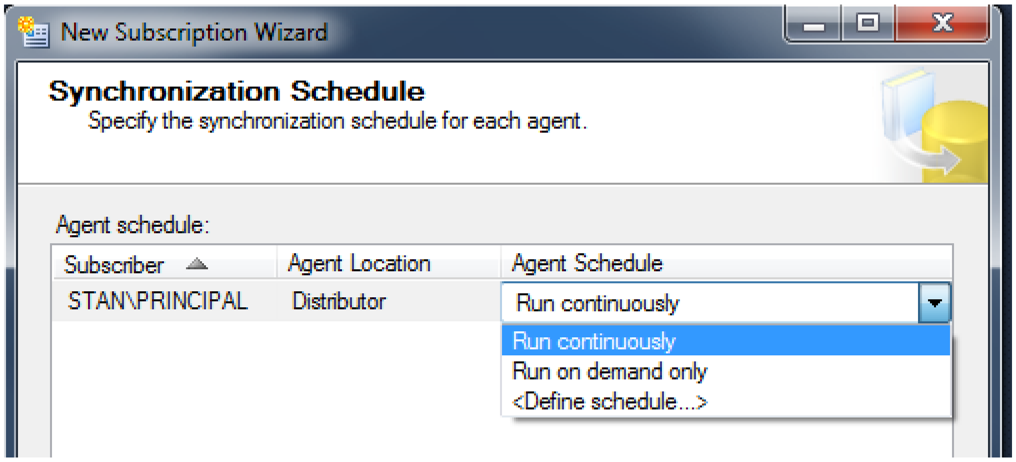
Distribution Agent Security

Specify the process account and connection options for each Distribution Agent.

Subscription properties:

Agent for Subscriber	Connection to Distributor	Connection to Subscriber
STAN\PRINCIPAL	Impersonate 'stan\karv...	Impersonate 'stan\karvaly'

5. For the schedule, select Run continuously to provide minimal latency



New Subscription Wizard

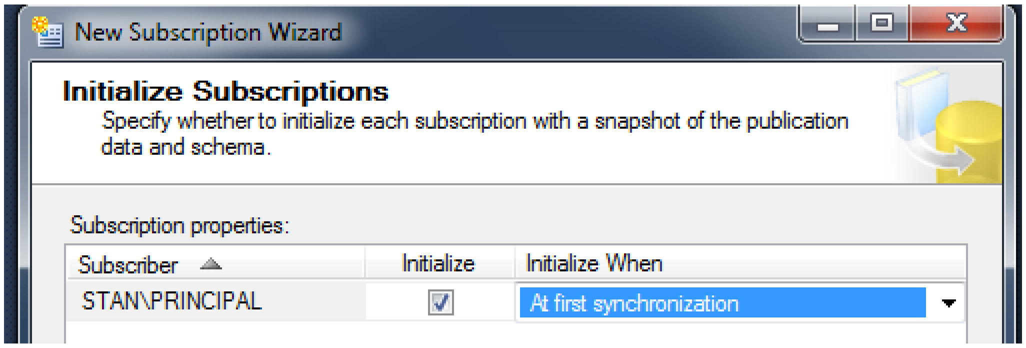
Synchronization Schedule

Specify the synchronization schedule for each agent.

Agent schedule:

Subscriber	Agent Location	Agent Schedule
STAN\PRINCIPAL	Distributor	Run continuously

6. On the next panel, select initialization. This will generate the first snapshot in the distribution folder.



New Subscription Wizard

Initialize Subscriptions

Specify whether to initialize each subscription with a snapshot of the publication data and schema.

Subscription properties:

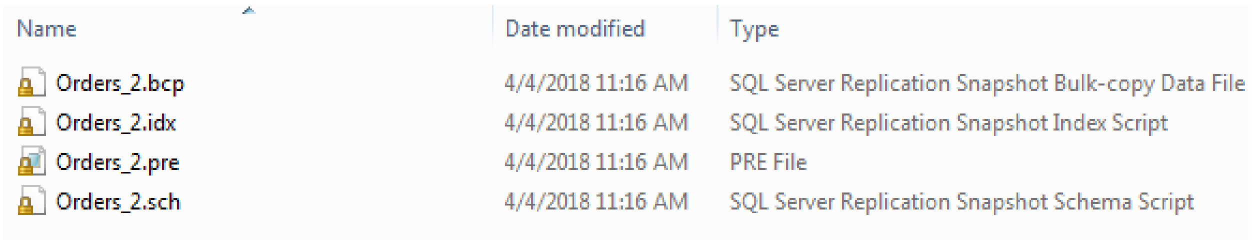
Subscriber	Initialize	Initialize When
STAN\PRINCIPAL	✓	At first synchronization

7. Finish creating the new subscription

Note: you can change the properties of subscriptions and publications later if you select Properties from their pop-up menu.

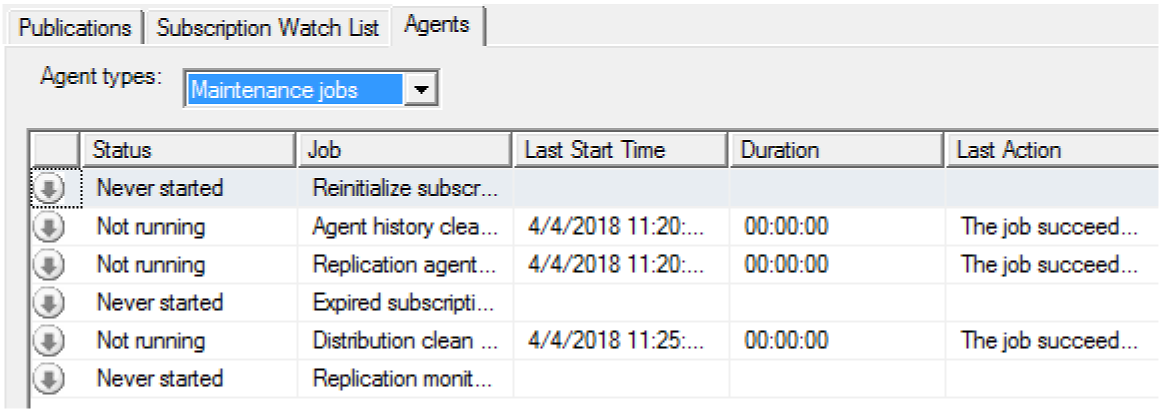
Checking the subscription

1. Locate the new Orders table in the nw_repl database and check that it contains the USA orders
2. Check the contents of the snapshot folder. **Bulk copy** is a fast method SQL server uses to insert data directly into database files.



Name	Date modified	Type
Orders_2.bcp	4/4/2018 11:16 AM	SQL Server Replication Snapshot Bulk-copy Data File
Orders_2.idx	4/4/2018 11:16 AM	SQL Server Replication Snapshot Index Script
Orders_2.pre	4/4/2018 11:16 AM	PRE File
Orders_2.sch	4/4/2018 11:16 AM	SQL Server Replication Snapshot Schema Script

3. Start the Replication monitor from the pop-up menu of the new subscription and check the publication and the subscription status. You can also review the active replication agents here:



Agent types: Maintenance jobs

	Status	Job	Last Start Time	Duration	Last Action
	Never started	Reinitialize subscr...			
	Not running	Agent history clea...	4/4/2018 11:20:...	00:00:00	The job succeed...
	Not running	Replication agent...	4/4/2018 11:20:...	00:00:00	The job succeed...
	Never started	Expired subscripti...			
	Not running	Distribution clean ...	4/4/2018 11:25:...	00:00:00	The job succeed...
	Never started	Replication monit...			

8. Open the Job activity monitor. The distribution agent appears as a new job in the list, with a status of Executing all the time
9. Change the first USA record of the orders table at the publisher with an UPDATE statement. The change appears in the replicated table shortly (ca. 30 seconds) after the snapshot agent is run the next time.
10. Finally, delete the subscription and the publication. This can be accomplished by selecting Generate scripts form the popup menu of the Replication group, specifying 'To drop…' and running the script in an editor. Alternatively, you can delete the objects one-by-one manually.
11. The replicated tables at the subscriber will not be deleted by deleting the subscription, so delete the Orders table manually from the nw_repl database.

PRACTICE: create a push snapshot publication into the nw_repl table that copies those employees from the Employees table whose title is Sales representative. Verify the correct operation, then delete all related objects.

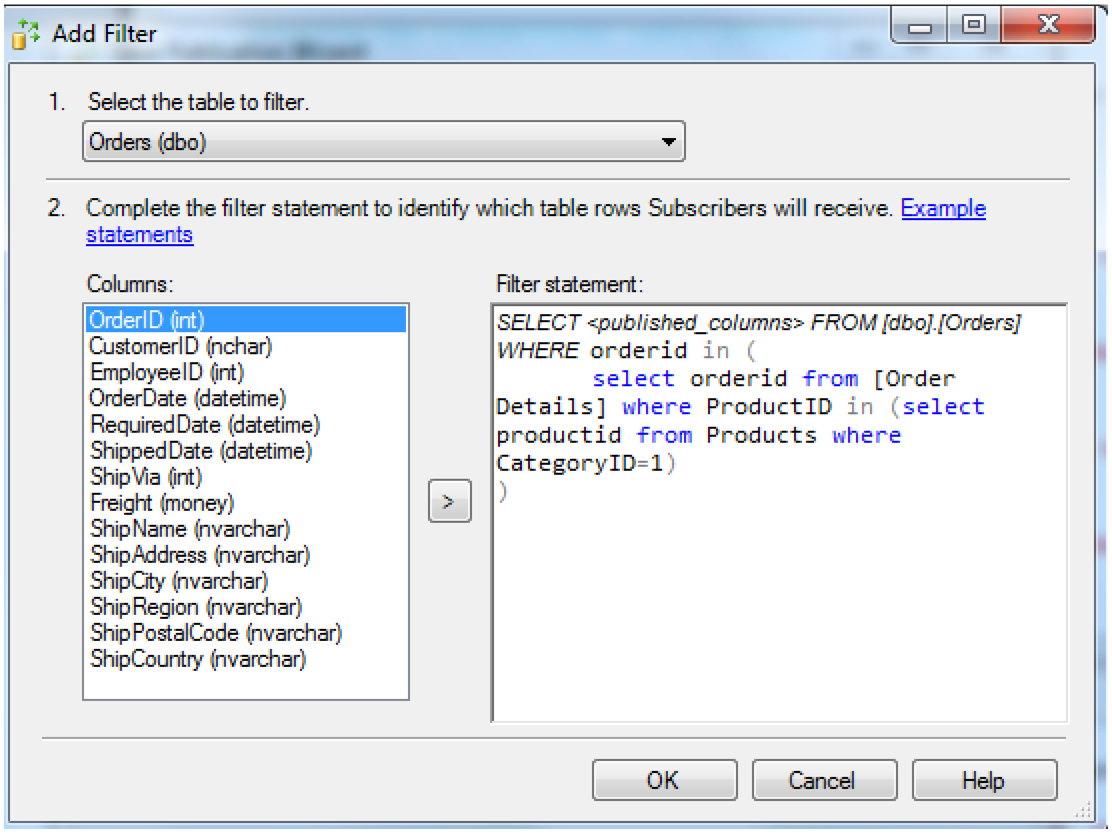
PRACTICE: transactional replication

Scenario: we wish to create a near-real time (scheduled) loose coupling between the central Northwind database and an off-site division that deals only with the products of the category 'Beverages' (CategoryID=1). We replicate only orders and order items that are beverages via transactional replication.

First we implement this demo on a single server (Principal). The filter condition above can be defined as follows:

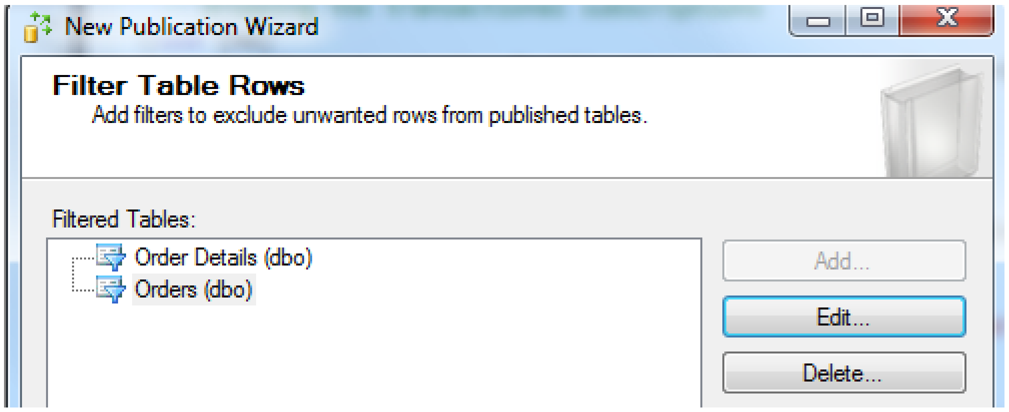
```
select * from [Order Details] where ProductID in (select productid from Products where
CategoryID=1)

select * from orders where orderid in (
        select orderid from [Order Details] where ProductID in (select productid from Products
where CategoryID=1)
)
```

1. Define the type of the publication as transactional on the Publication type dialog panel
2. Select the Orders and Order Details tables on the Articles panel
3. On the Filer table panel, add the filter to the two tables one by one by copying the WHERE part from the above queries. For the Orders table:



```
SELECT <published_columns> FROM [dbo].[Orders]
WHERE orderid in (
        select orderid from [Order
Details] where ProductID in (select
productid from Products where
CategoryID=1)
)
```

4. You should have the filters defined for both tables:



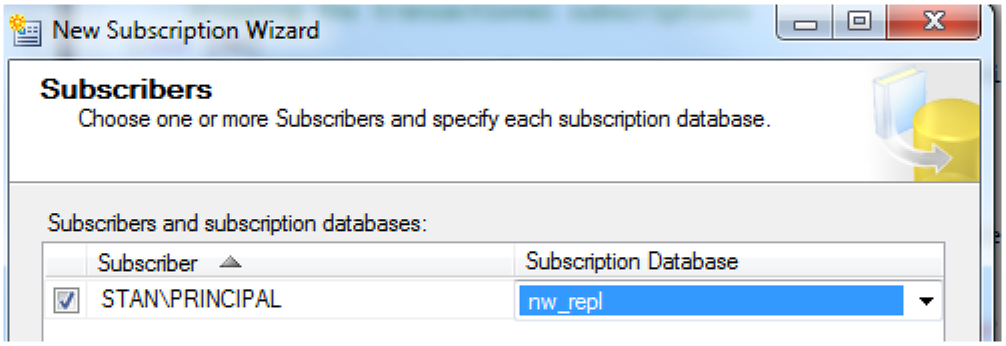
Filter Table Rows

Add filters to exclude unwanted rows from published tables.

Filtered Tables:

- Order Details (dbo)
- Orders (dbo)

5. On the next panel, select Create a snapshot immediately
6. On the panels that follow set the security of the agents the same way as in the previous demo
7. Name the publication nw_trans and finish creating the publication
8. Select New subscription in the pop-up menu of the publication
9. Select the Run all agents at the Distributor on the next panel (push subscription)
10. Select the nw_repl database as the subscription database:



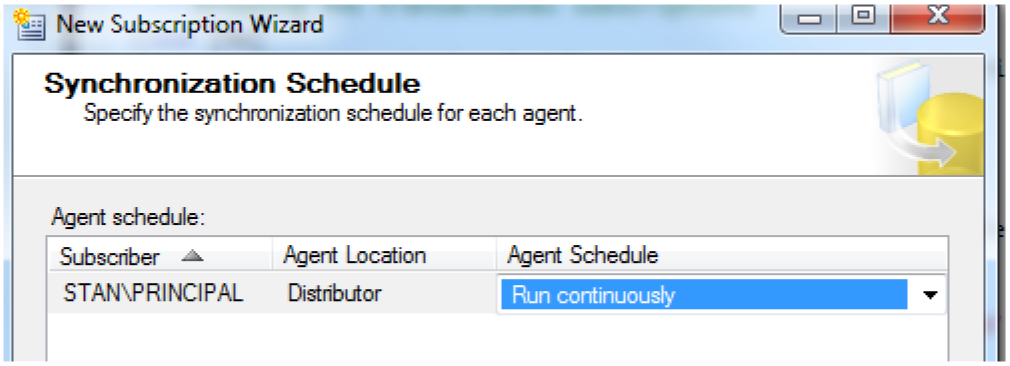
Subscribers

Choose one or more Subscribers and specify each subscription database.

Subscribers and subscription databases:

	Subscriber	Subscription Database
✓	STAN\PRINCIPAL	nw_repl

11. Set the security of the distribution agent the same way as in the previous demo
12. Specify Run continuously for the schedule of the Log reader and Distribution agents:



Synchronization Schedule

Specify the synchronization schedule for each agent.

Agent schedule:

Subscriber	Agent Location	Agent Schedule
STAN\PRINCIPAL	Distributor	Run continuously

13. Test the correct operation of the transactional replication. Update the employeeID in first record of the Orders table in the nw database and then select the same record in the nw_repl database. You should see the changed value within 10 seconds.

14. Check the operation of the replication agents
15. Clean up the replication by deleting all replication objects

PRACTICE: implement the loose coupling scenario for order event processing using transactional replication on the Products, Orders and Order details tables. We suppose that the logistics division has its own database, possibly running on another server.

1. Add a new field named status to the Orders table in the nw database with a default value of 0
2. Replicate the three tables to the nw_repl database
3. Since the subscriber can change the replicated records and since the trading application using the Orders table will *never* update the status field, we will use this field on the subscriber to log the processing state of order records in a way similar to the case study solution:
   a. New orders will have a status of 0 by default
   b. In order to mark changed orders, we can use an update trigger on the publisher that changes the status of the already existing record to 1
   c. The job at the subscriber processes order records with a status 0 or 1 and sets the status to 2 on success

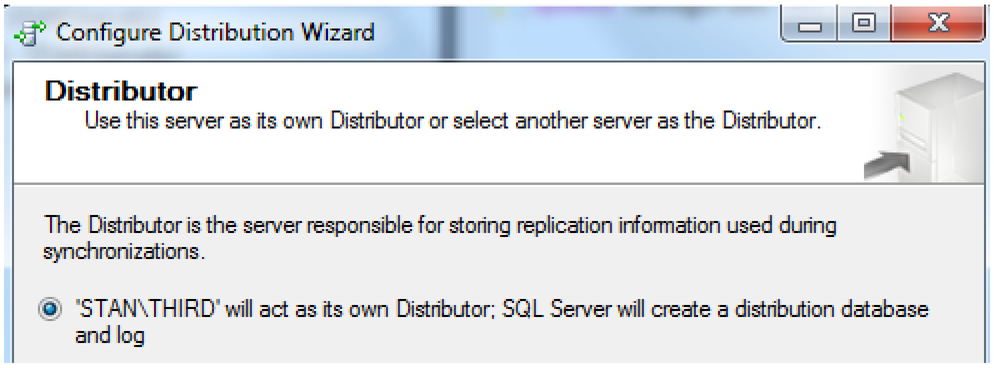
In this way we avoid using an extra log table.

Replication between separate servers

In the next demo we implement the same transactional replication in a more realistic scenario using the Principal as the publisher and the Third server as the distributor and subscriber, respectively. In an even more realistic scenario, they would be not only separate server instances, but they would also reside on separate server machines. We cannot, however, implement such a scenario in the lab.

Configuring the distributor

1. In the pop-up menu of Replication on the Third instance select Configure Distribution, and accept the first choice. This will create the distribution database on Third.



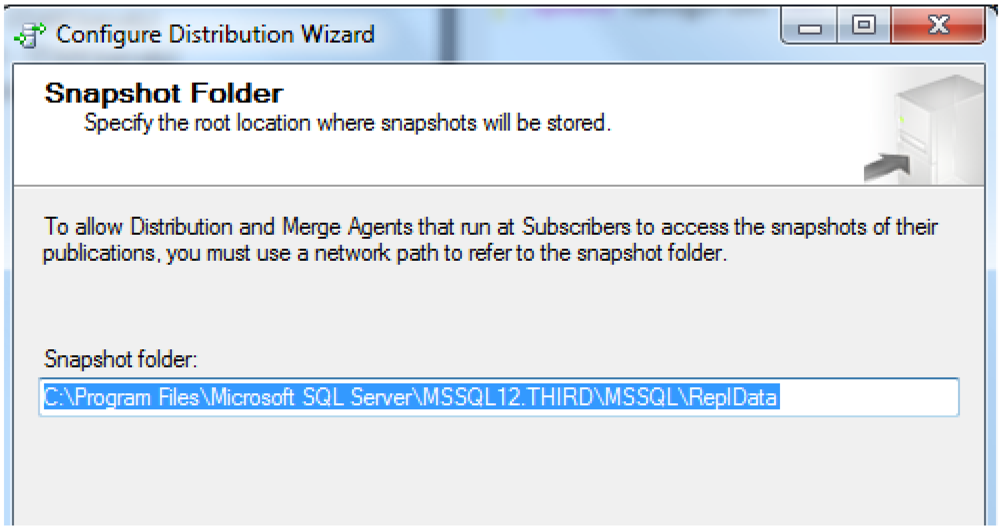
Distributor

Use this server as its own Distributor or select another server as the Distributor.

The Distributor is the server responsible for storing replication information used during synchronizations.

[✓ 'STAN\THIRD' will act as its own Distributor; SQL Server will create a distribution database and log]

2. On the next pane accept the location of the snapshot folder. Since the Third instance will also be the subscriber, we do not create a network share.



Snapshot Folder

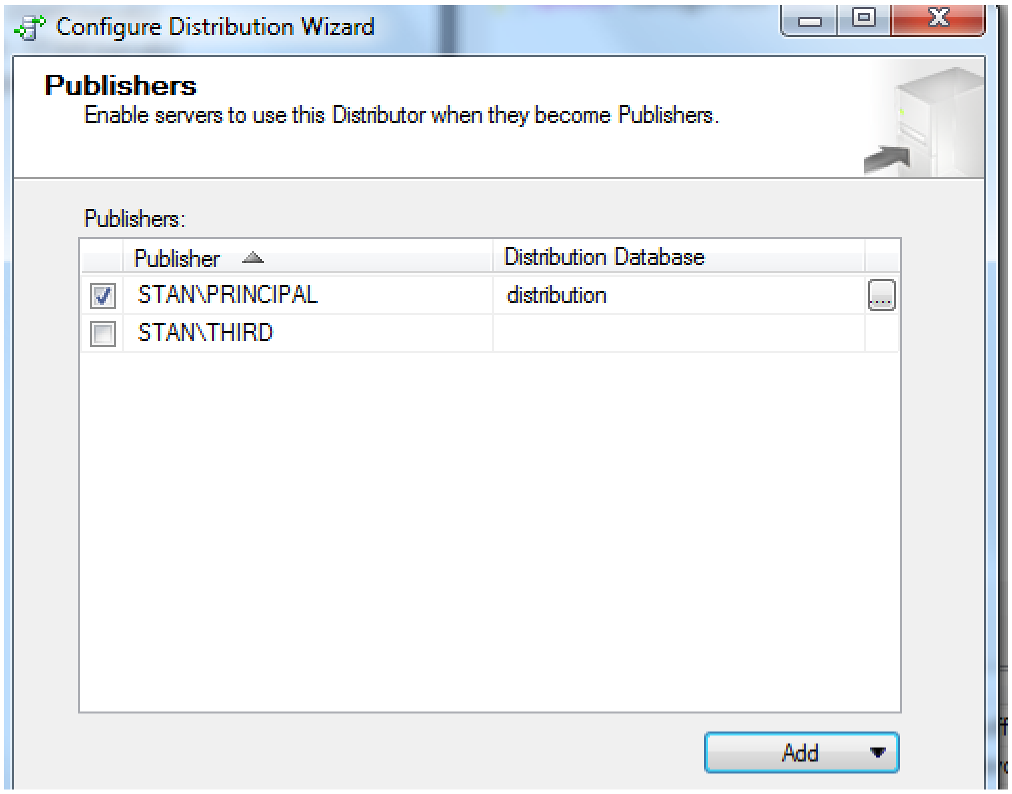
Specify the root location where snapshots will be stored.

To allow Distribution and Merge Agents that run at Subscribers to access the snapshots of their publications, you must use a network path to refer to the snapshot folder.

Snapshot folder:

C:\Program Files\Microsoft SQL Server\MSSQL12.THIRD\MSSQL\ReplData

3. On the next pane accept the defaults for the location and name of the distribution database.
4. We then have to specify which publisher servers are allowed to use this distribution database. On the next pane deselect Third (since Third will not be publishing) and by pressing Add, add the Principal instance:



Publishers

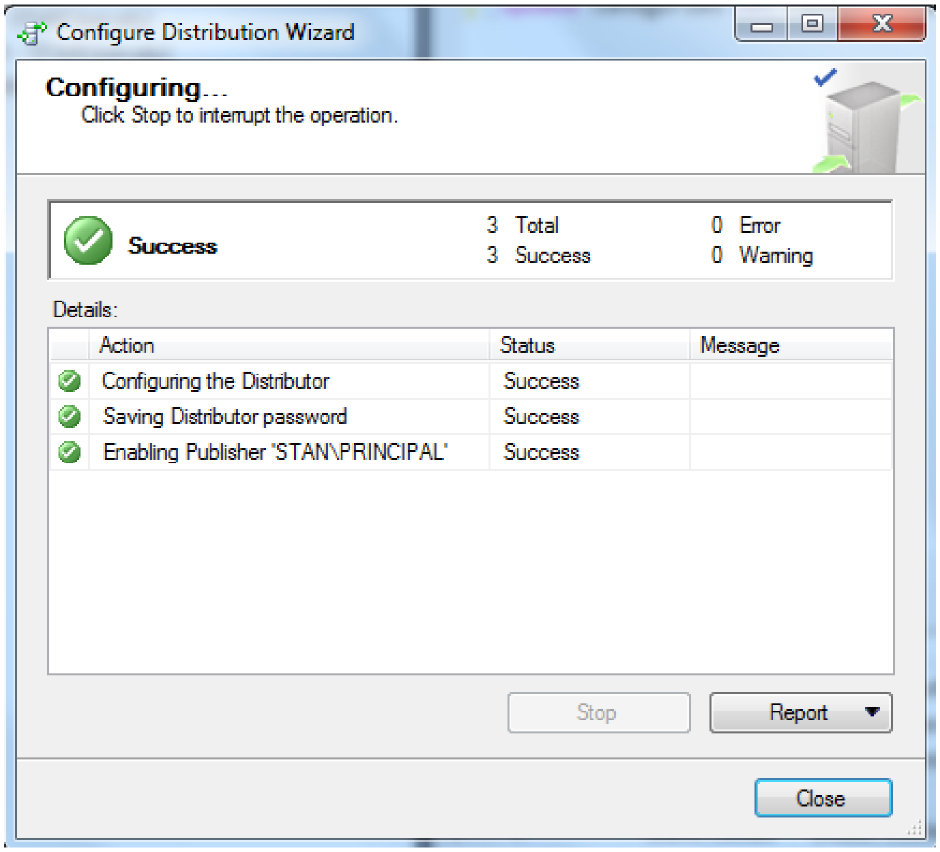
Enable servers to use this Distributor when they become Publishers.

Publishers:

	Publisher	Distribution Database
☑	STAN\PRINCIPAL	distribution
☐	STAN\THIRD	

5. On the next pane you must specify a password that the Publishers using this distribution database will need to use. Specify the same password that you use for login.

6. At the end of the process, you have configured the distributor successfully:


Configure Distribution Wizard

Configuring...
Click Stop to interrupt the operation.

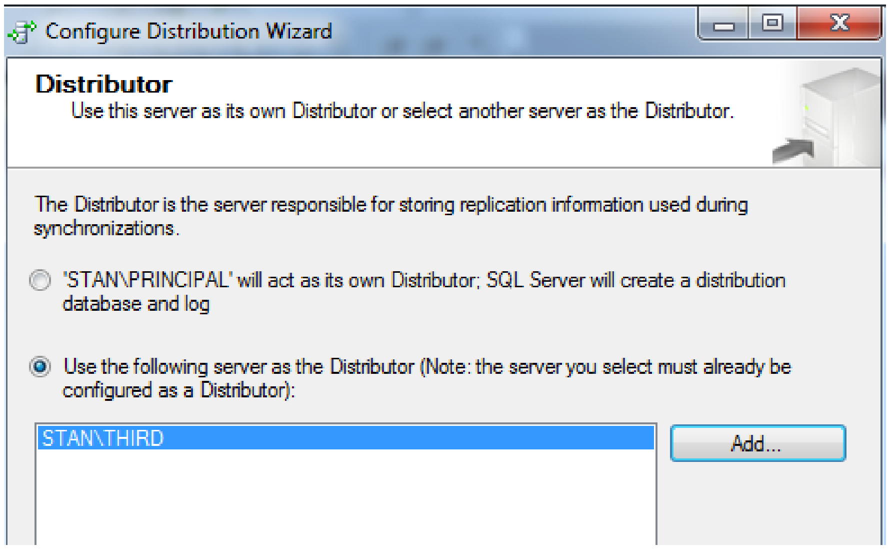
Success — 3 Total, 3 Success, 0 Error, 0 Warning

Details:

	Action	Status	Message
✓	Configuring the Distributor	Success	
✓	Saving Distributor password	Success	
✓	Enabling Publisher 'STAN\PRINCIPAL'	Success	

Configuring the publisher

1. In order to configure the Principal as publisher, we first disable it as distributor. Remember that so far the Principal acted as its own distributor. In the pop-up menu of Replication select Disable Distribution and Publishing. When the Principal has been disabled as a distributor, the pop-up menu changes. Select now Configure distribution and specify the Third instance as the distributor of Principal:


Configure Distribution Wizard

Distributor
Use this server as its own Distributor or select another server as the Distributor.

The Distributor is the server responsible for storing replication information used during synchronizations.

○ 'STAN\PRINCIPAL' will act as its own Distributor; SQL Server will create a distribution database and log

[✓ ● Use the following server as the Distributor (Note: the server you select must already be configured as a Distributor):]

STAN\THIRD

2. On the next pane, enter the same password as before.
3. The distribution is now set up.

Adding the publication and the subscription

1. Go on creating the transactional publication on Principal in the usual way.
2. Create a new subscription on the Third instance. Select Principal as the publisher and select the publication that you created in the previous step. You can have the replication database created on Third by the wizard.
3. Open a query editor on the Principal and update a record in a table that is part of the publication.
4. Open a query editor on the Third and verify that the change is propagated to the replicated table within 10 seconds.
5. Delete the publication and the subscription.

Merge replication

Scenario: our sales employees are traveling and making new orders for their clients. Occasionally they also need to change previous orders when, for example, a shipping address changes. It may also happen that two employees modify the same order. The employees are not continuously connected to the internet. We must design a replication solution that merges all such changes with each other and with the central Northwind database.

In **merge replication**, the duty of the log reader agent is taken by triggers, tables, and views which are created automatically in each subscriber database and also in the publisher database. The triggers log changes in special metadata tables named 'MSmerge_*' that are created in the same database as the replicated table. There are three triggers named 'MSmerge_[ins, upd, del]_*' created for each table. There are also database level schema triggers that log the changes in the schema of the replicated tables.

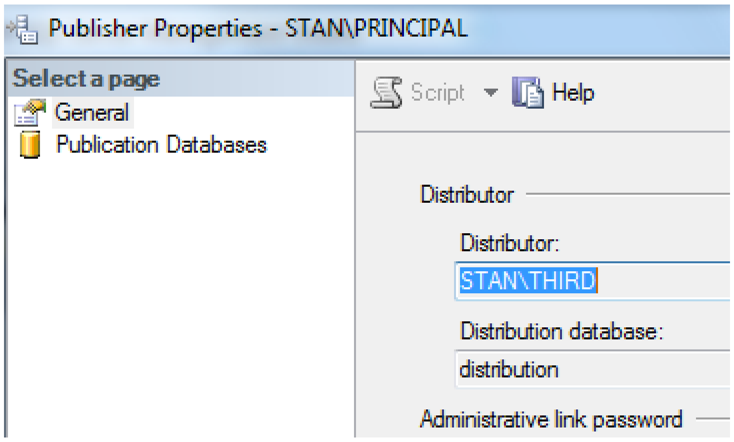
Merge replication starts with an initial snapshot created by a snapshot agent. By default, the snapshot agent is configured to run every 14 days. Then the duty of the merge agent is similar to the distribution agent in transactional replication with the difference that merge replication can be configured bi-directional. This means that the agent applies changes on the subscriber and publisher sides as well. There is a separate merge agent for each subscription. The merge agent runs on the distributor in case of a push distribution and on the subscriber in case of a pull subscription.

In order to support the bi-directional data synchronization, articles in merge replication must have a uniqueidentifier (GUID) type column that is similar to the identity data type but it produces globally unique IDs[4]. If such a column does not exist, it is added automatically to the tables—which can potentially crash the legacy applications already using the table.

[4] Use this data type like this: CREATE TABLE test (my_guid uniqueidentifier DEFAULT NEWSEQUENTIALID() ROWGUIDCOL,… etc

The publication

In order to develop the merge solution, first configure the Third instance as the distributor of the Principal (see the previous section). After that, this is what you should see on the Principal:


Publisher Properties - STAN\PRINCIPAL

Distributor

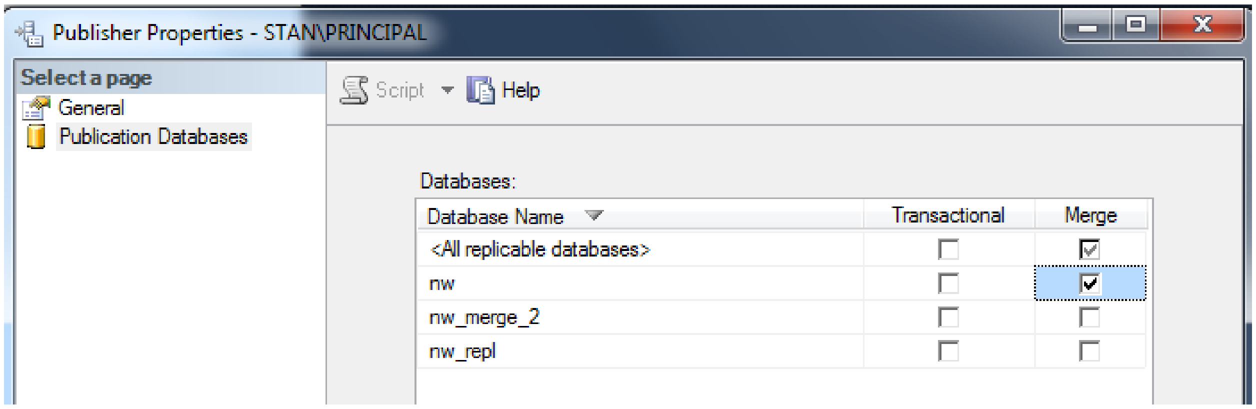
Distributor: STAN\THIRD

Distribution database: distribution

Administrative link password

Now add the publication on Principal as follows.

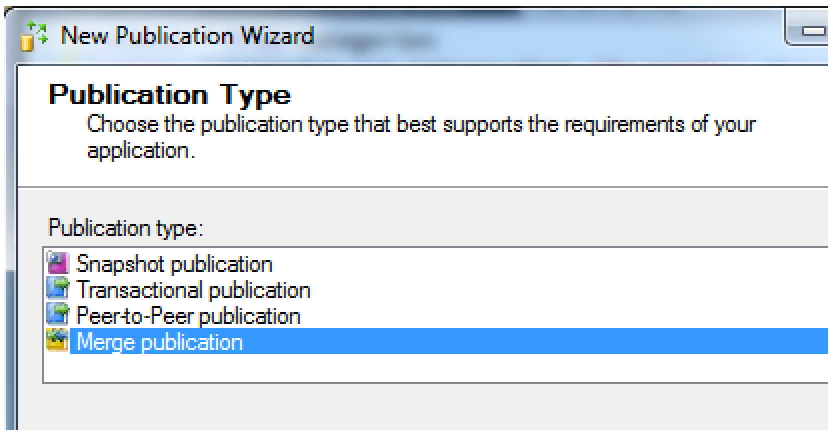
1. Enable the nw database for merge replication. Select Publisher properties from the pop-up menu of Replication:


Publisher Properties - STAN\PRINCIPAL

Databases:

Database Name	Transactional	Merge
<All replicable databases>	☐	☑
nw	☐	☑
nw_merge_2	☐	☐
nw_repl	☐	☐

2. Select the nw database for the publication database and select the publication type:


New Publication Wizard

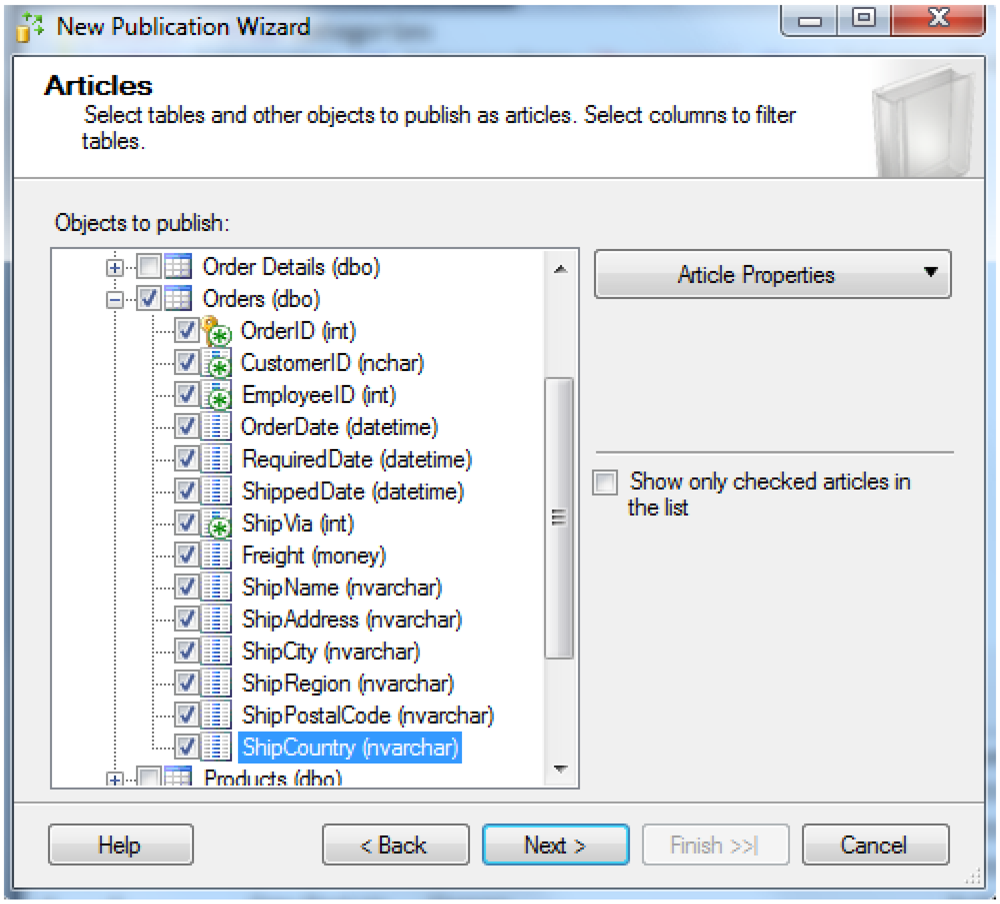
Publication Type

Choose the publication type that best supports the requirements of your application.

Publication type:

- Snapshot publication
- Transactional publication
- Peer-to-Peer publication
- Merge publication

3. Accept the default subscriber types and select the Orders table as the single article of the publication.


New Publication Wizard

Articles

Select tables and other objects to publish as articles. Select columns to filter tables.

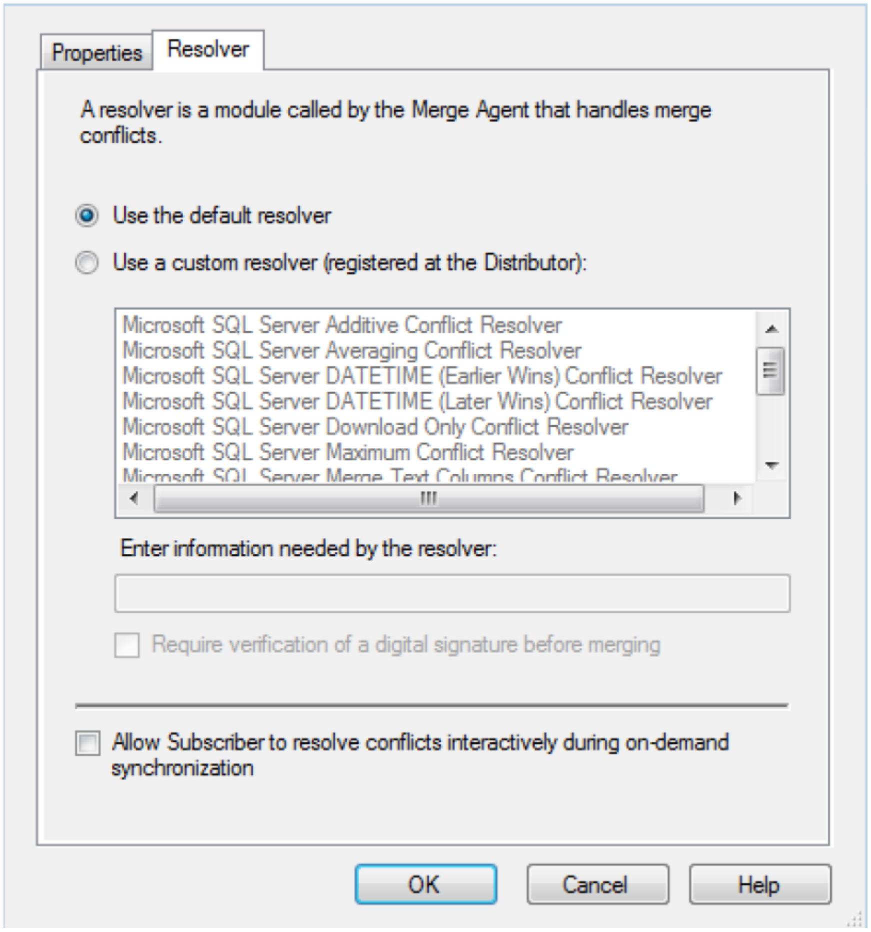
Objects to publish:

- [ ] Order Details (dbo)
- [✓] Orders (dbo)
  - [✓] OrderID (int)
  - [✓] CustomerID (nchar)
  - [✓] EmployeeID (int)
  - [✓] OrderDate (datetime)
  - [✓] RequiredDate (datetime)
  - [✓] ShippedDate (datetime)
  - [✓] ShipVia (int)
  - [✓] Freight (money)
  - [✓] ShipName (nvarchar)
  - [✓] ShipAddress (nvarchar)
  - [✓] ShipCity (nvarchar)
  - [✓] ShipRegion (nvarchar)
  - [✓] ShipPostalCode (nvarchar)
  - [✓] ShipCountry (nvarchar)
- [ ] Products (dbo)

Article Properties

[ ] Show only checked articles in the list

4. You can set various properties for each article by selecting Article Properties, including the resolver to be used for various conflict types. You could also add your own resolver. However, we do not go into the details of conflict resolution in this course.



A resolver is a module called by the Merge Agent that handles merge conflicts.

(●) Use the default resolver

( ) Use a custom resolver (registered at the Distributor):

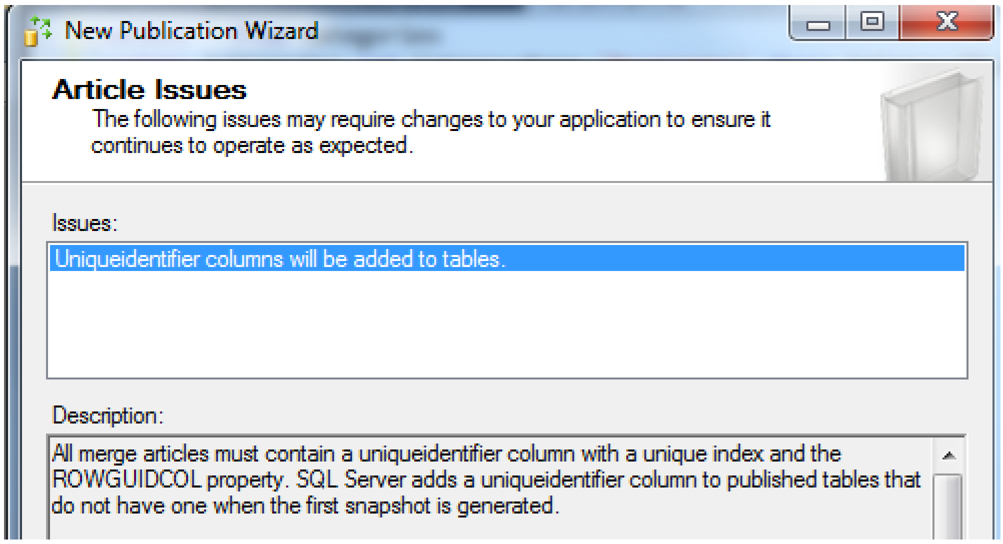
Microsoft SQL Server Additive Conflict Resolver
Microsoft SQL Server Averaging Conflict Resolver
Microsoft SQL Server DATETIME (Earlier Wins) Conflict Resolver
Microsoft SQL Server DATETIME (Later Wins) Conflict Resolver
Microsoft SQL Server Download Only Conflict Resolver
Microsoft SQL Server Maximum Conflict Resolver
Microsoft SQL Server Merge Text Columns Conflict Resolver

Enter information needed by the resolver:

[ ] Require verification of a digital signature before merging

[ ] Allow Subscriber to resolve conflicts interactively during on-demand synchronization

5. The next panel warns that a new GUID will be added to the table. This does not change the primary and foreign key constraints of the table.



Article Issues

The following issues may require changes to your application to ensure it continues to operate as expected.

Issues:

Uniqueidentifier columns will be added to tables.

Description:

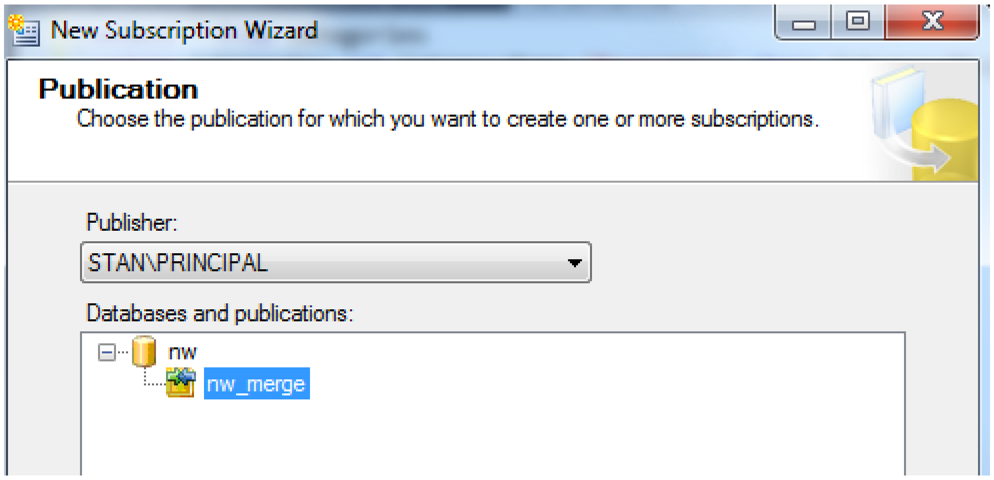
All merge articles must contain a uniqueidentifier column with a unique index and the ROWGUIDCOL property. SQL Server adds a uniqueidentifier column to published tables that do not have one when the first snapshot is generated.

6. For the snapshot agent scheduling, accept the defaults, then define the agent security in the usual way.
7. Name the publication nw_merge and create it.

The subscription

We add two subscribers to see the merge process in action.

1. On the Third instance, select New subscription and specify the Principal as the publisher:



New Subscription Wizard

Publication

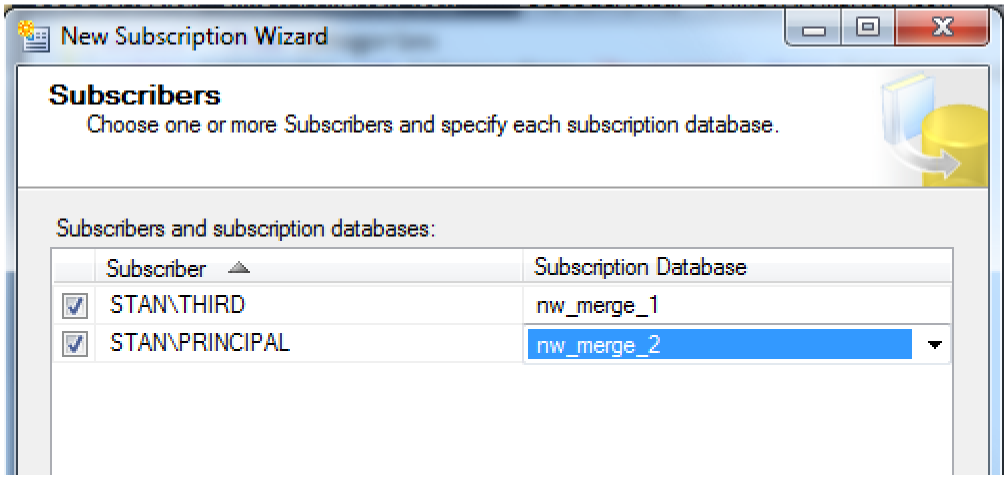
Choose the publication for which you want to create one or more subscriptions.

Publisher: STAN\PRINCIPAL

Databases and publications:

nw
nw_merge

2. On the next pane, select pull subscription. This is in line with the usual expectation that the subscribers will want to schedule their synchronization.
3. Add the two subscribers, one on the Third instance and one on the Principal, with newly created databases nw_merge_1-2.



New Subscription Wizard

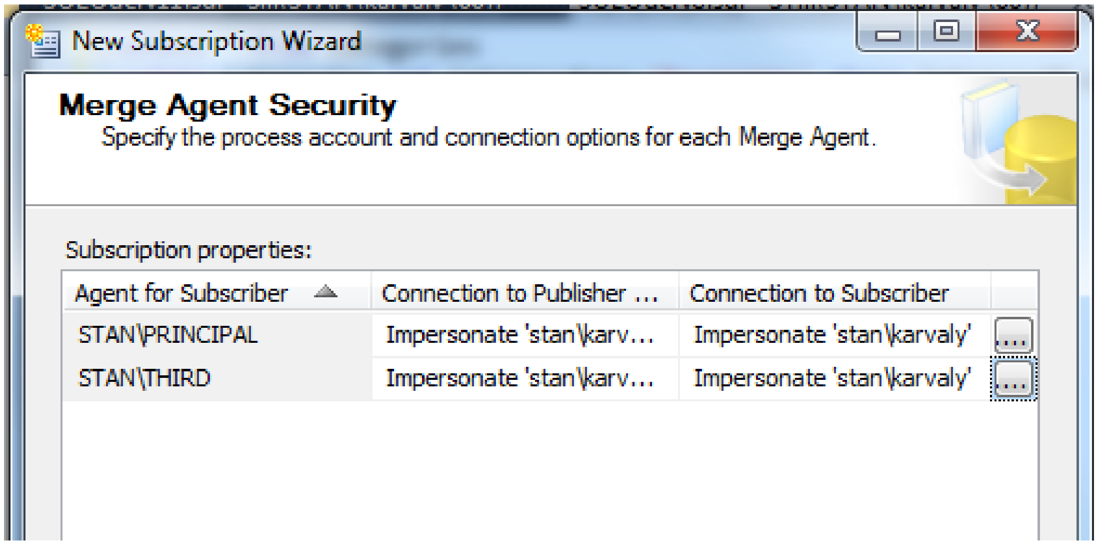
Subscribers

Choose one or more Subscribers and specify each subscription database.

Subscribers and subscription databases:

	Subscriber	Subscription Database
✓	STAN\THIRD	nw_merge_1
✓	STAN\PRINCIPAL	nw_merge_2

4. Set the security on the agents in the usual way.



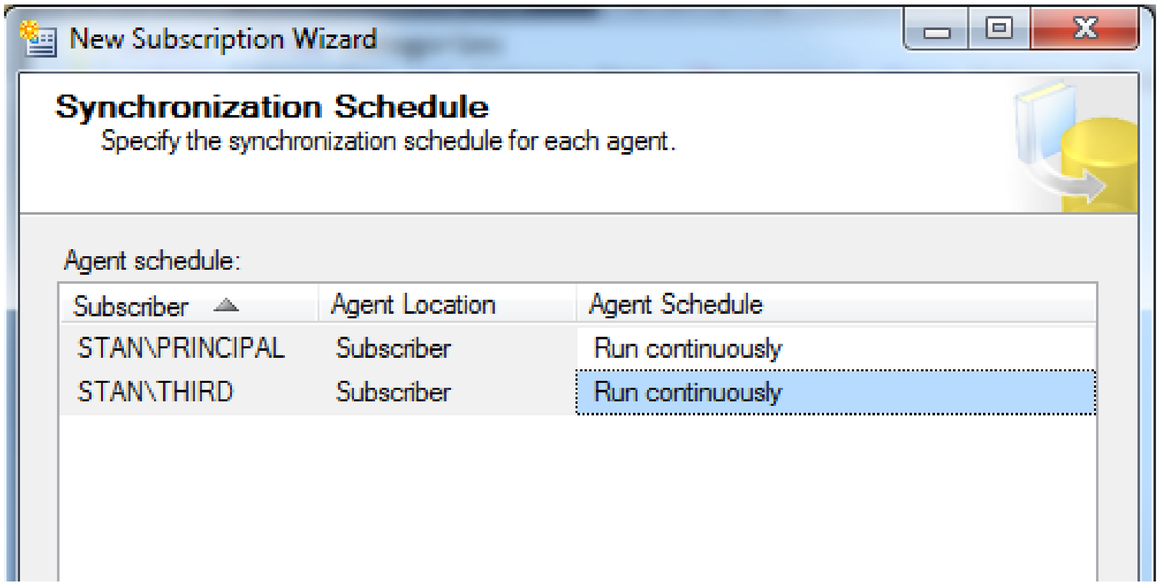
Merge Agent Security

Specify the process account and connection options for each Merge Agent.

Subscription properties:

Agent for Subscriber	Connection to Publisher ...	Connection to Subscriber
STAN\PRINCIPAL	Impersonate 'stan\karv...	Impersonate 'stan\karvaly'
STAN\THIRD	Impersonate 'stan\karv...	Impersonate 'stan\karvaly'

5. We set the synch schedule for both merge agents to Run continuously. *Note: this setting is only for test and demo purposes. In a real life scenario, the merge process would start either at specified intervals or on demand, when a specific event, e.g. a VPN connection occurs on the subscriber.*



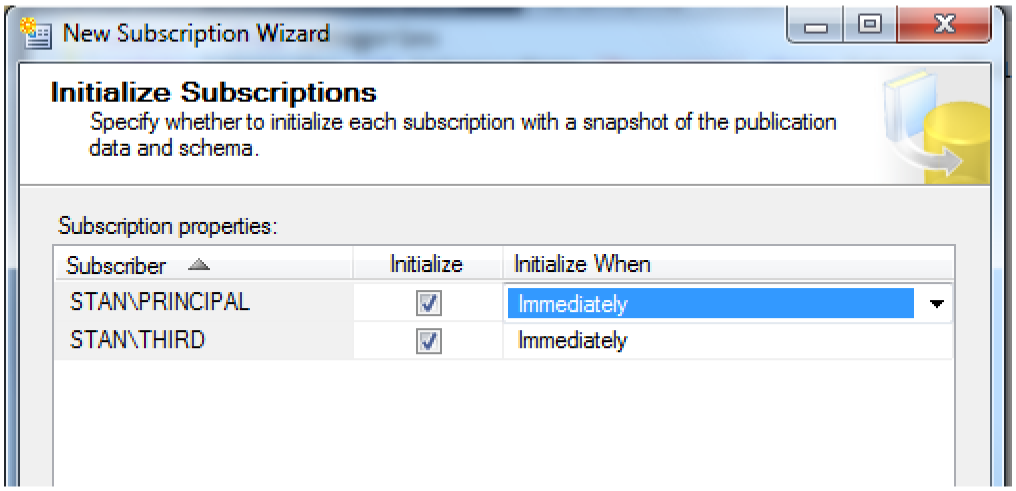
Synchronization Schedule

Specify the synchronization schedule for each agent.

Agent schedule:

Subscriber	Agent Location	Agent Schedule
STAN\PRINCIPAL	Subscriber	Run continuously
STAN\THIRD	Subscriber	Run continuously

6. Initialize the subscriptions immediately.



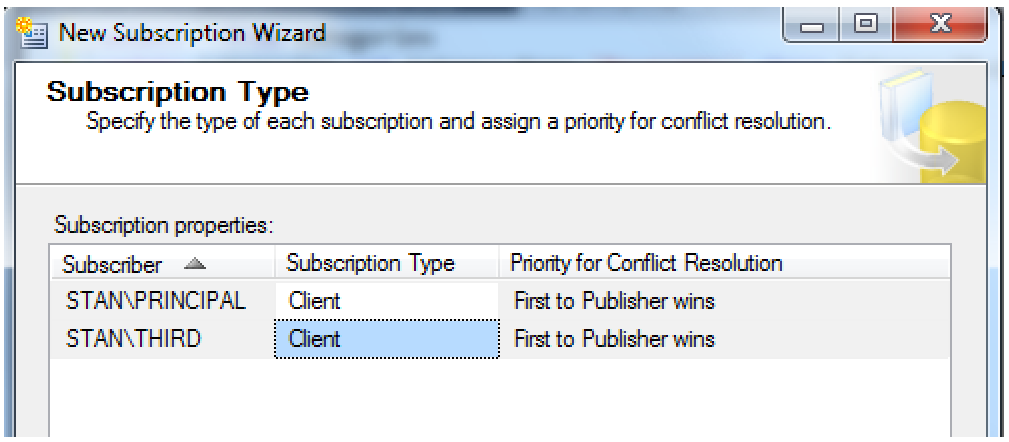
Initialize Subscriptions

Specify whether to initialize each subscription with a snapshot of the publication data and schema.

Subscription properties:

Subscriber	Initialize	Initialize When
STAN\PRINCIPAL	✓	Immediately
STAN\THIRD	✓	Immediately

7. In merge publication, the subscribers may be allowed to republish the publication to which they are subscribed (Server type subscription), thus creating a hierarchical subscribing architecture. Since we do not want to create a hierarchy of subscribers, we select Client type—no republication.



Subscription Type

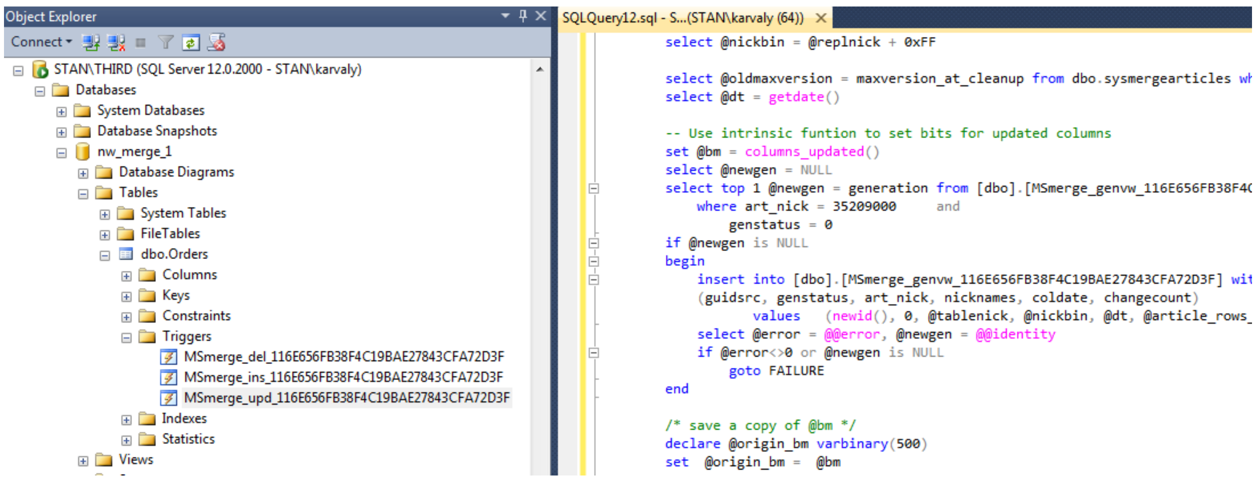
Specify the type of each subscription and assign a priority for conflict resolution.

Subscription properties:

Subscriber	Subscription Type	Priority for Conflict Resolution
STAN\PRINCIPAL	Client	First to Publisher wins
STAN\THIRD	Client	First to Publisher wins

8. Finalize and start the replication.

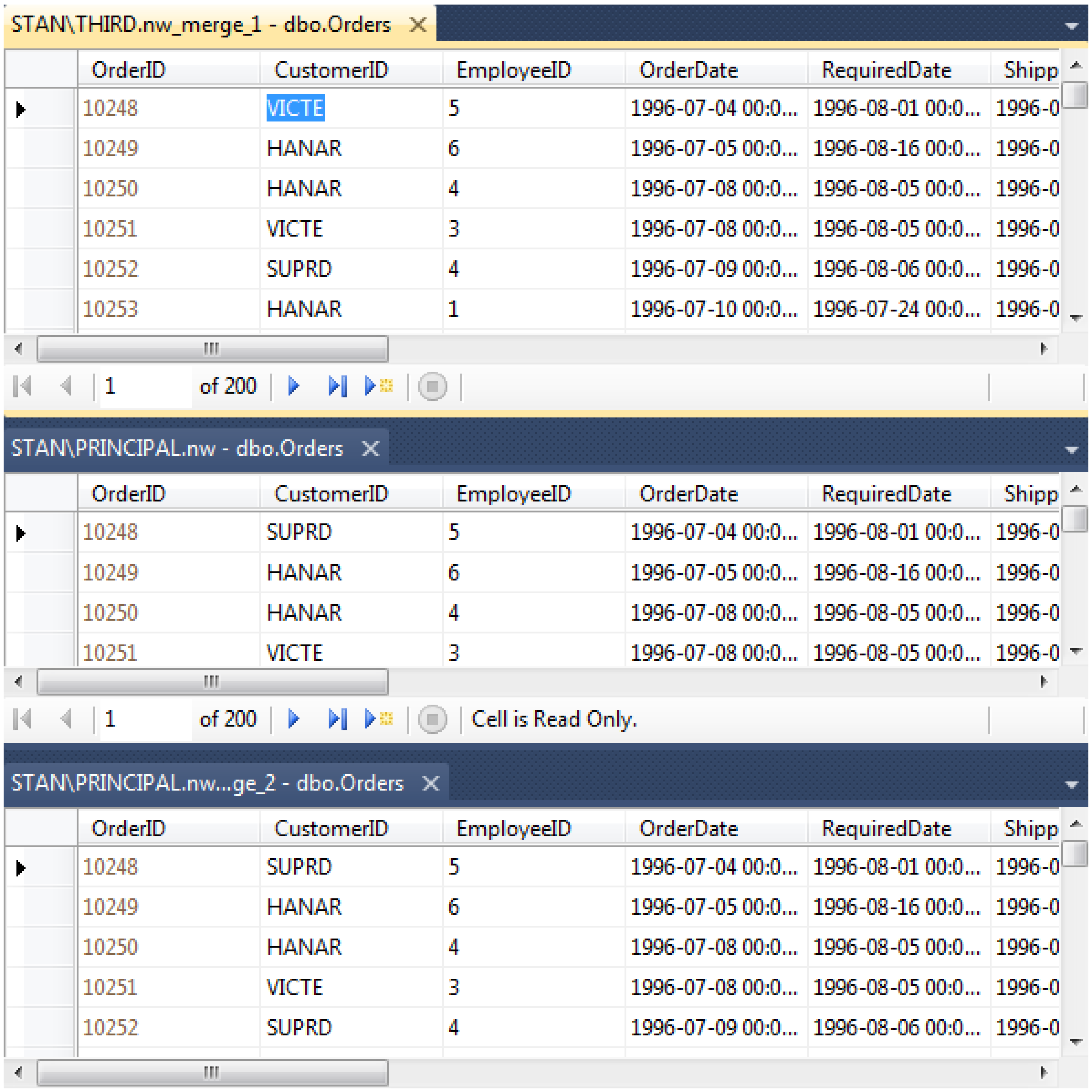
9. In the object browser, verify that the triggers have been created:



```
STAN\THIRD (SQL Server 12.0.2000 - STAN\karvaly)
  Databases
    System Databases
    Database Snapshots
    nw_merge_1
      Database Diagrams
      Tables
        System Tables
        FileTables
        dbo.Orders
          Columns
          Keys
          Constraints
          Triggers
            MSmerge_del_116E656FB38F4C19BAE27843CFA72D3F
            MSmerge_ins_116E656FB38F4C19BAE27843CFA72D3F
            MSmerge_upd_116E656FB38F4C19BAE27843CFA72D3F
          Indexes
          Statistics
      Views
```

```
select @nickbin = @replnick + 0xFF

select @oldmaxversion = maxversion_at_cleanup from dbo.sysmergearticles wh
select @dt = getdate()

-- Use intrinsic funtion to set bits for updated columns
set @bm = columns_updated()
select @newgen = NULL
select top 1 @newgen = generation from [dbo].[MSmerge_genvw_116E656FB38F4C
    where art_nick = 35209000     and
        genstatus = 0
if @newgen is NULL
begin
    insert into [dbo].[MSmerge_genvw_116E656FB38F4C19BAE27843CFA72D3F] wit
    (guidsrc, genstatus, art_nick, nicknames, coldate, changecount)
           values   (newid(), 0, @tablenick, @nickbin, @dt, @article_rows_
    select @error = @@error, @newgen = @@identity
    if @error<>0 or @newgen is NULL
        goto FAILURE
end

/* save a copy of @bm */
declare @origin_bm varbinary(500)
set  @origin_bm =  @bm
```

10. Verify that the changed values are propagated from any subscriber or the publisher to all the other by editing the Orders table in three separate editors. Be patient: It can take up to 2 minutes for the synchronization to complete. Note the first-come-first-served style default conflict resolution when the two subscribers update a record ‘simultaneously’.



STAN\THIRD.nw_merge_1 - dbo.Orders

OrderID	CustomerID	EmployeeID	OrderDate	RequiredDate	Shipp
10248	VICTE	5	1996-07-04 00:0...	1996-08-01 00:0...	1996-0
10249	HANAR	6	1996-07-05 00:0...	1996-08-16 00:0...	1996-0
10250	HANAR	4	1996-07-08 00:0...	1996-08-05 00:0...	1996-0
10251	VICTE	3	1996-07-08 00:0...	1996-08-05 00:0...	1996-0
10252	SUPRD	4	1996-07-09 00:0...	1996-08-06 00:0...	1996-0
10253	HANAR	1	1996-07-10 00:0...	1996-07-24 00:0...	1996-0

1 of 200

STAN\PRINCIPAL.nw - dbo.Orders

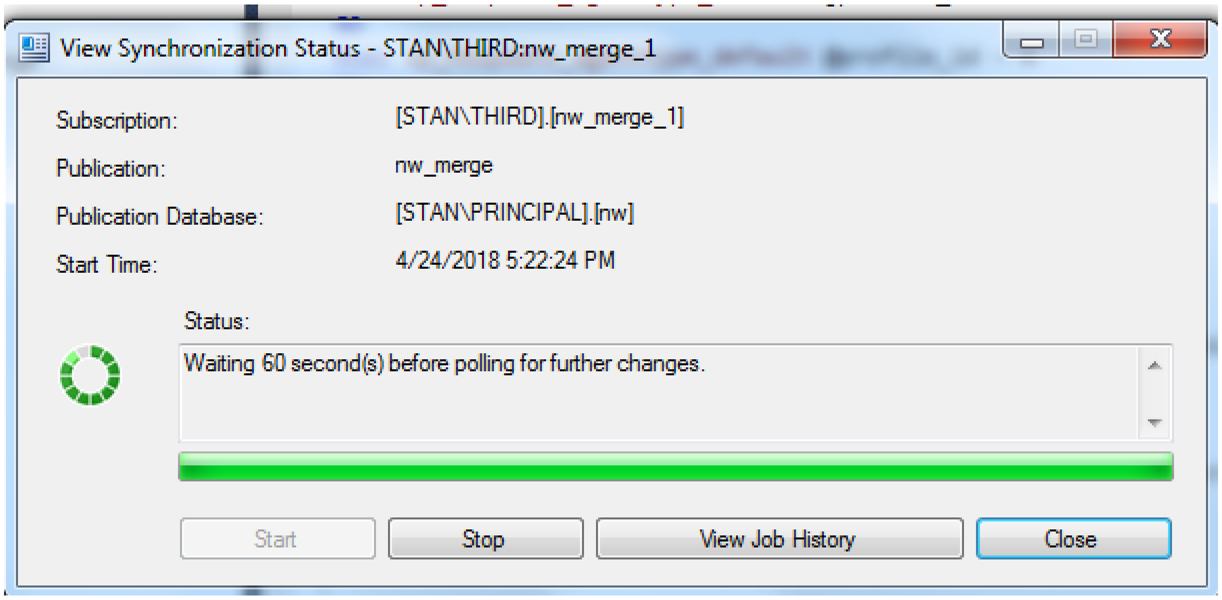
OrderID	CustomerID	EmployeeID	OrderDate	RequiredDate	Shipp
10248	SUPRD	5	1996-07-04 00:0...	1996-08-01 00:0...	1996-0
10249	HANAR	6	1996-07-05 00:0...	1996-08-16 00:0...	1996-0
10250	HANAR	4	1996-07-08 00:0...	1996-08-05 00:0...	1996-0
10251	VICTE	3	1996-07-08 00:0...	1996-08-05 00:0...	1996-0

1 of 200 Cell is Read Only.

STAN\PRINCIPAL.nw...ge_2 - dbo.Orders

OrderID	CustomerID	EmployeeID	OrderDate	RequiredDate	Shipp
10248	SUPRD	5	1996-07-04 00:0...	1996-08-01 00:0...	1996-0
10249	HANAR	6	1996-07-05 00:0...	1996-08-16 00:0...	1996-0
10250	HANAR	4	1996-07-08 00:0...	1996-08-05 00:0...	1996-0
10251	VICTE	3	1996-07-08 00:0...	1996-08-05 00:0...	1996-0
10252	SUPRD	4	1996-07-09 00:0...	1996-08-06 00:0...	1996-0

11. You can review the current state of the merge process by selecting View synch status from the pop-up menu of the subscription. You can also start the process manually.


View Synchronization Status - STAN\THIRD:nw_merge_1

Subscription:	[STAN\THIRD].[nw_merge_1]
Publication:	nw_merge
Publication Database:	[STAN\PRINCIPAL].[nw]
Start Time:	4/24/2018 5:22:24 PM

Status:

Waiting 60 second(s) before polling for further changes.

12. Generate and review replication scripts on both instances by selecting Generate scripts from the pop-up menu of the Replication group.

13. Delete all replication objects.

Log shipping

While the primary application area of replication is business process management, log shipping is designed to support **disaster recovery** of a database, by creating and synchronizing an exact, usually read-only copy of the database also known as a ‘warm’ standby database. A log can be shipped from the primary sever to multiple secondary servers.

The three steps involved in log shipping are as follows.

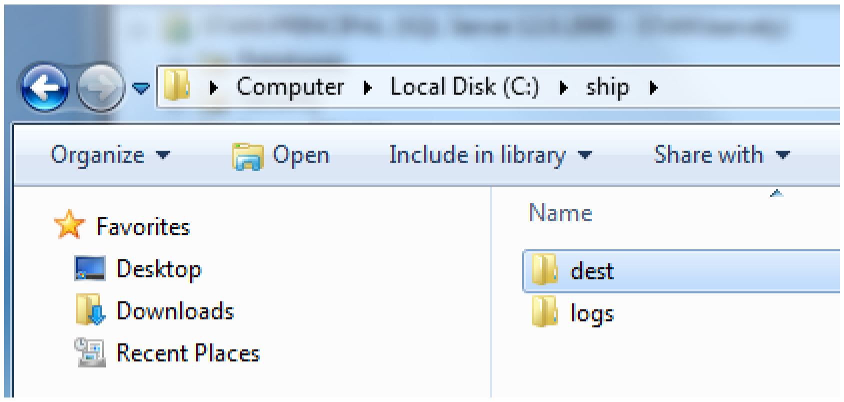
1. A backup job running on the primary server backs up the database transaction log to the local server
2. A copy job running on the secondary server copies the log to a configurable destination (e.g. a network file server)
3. A restore job running on the secondary server restores the backup to the secondary database

An alert job may also be running monitoring whether each step is performed as expected and on time.

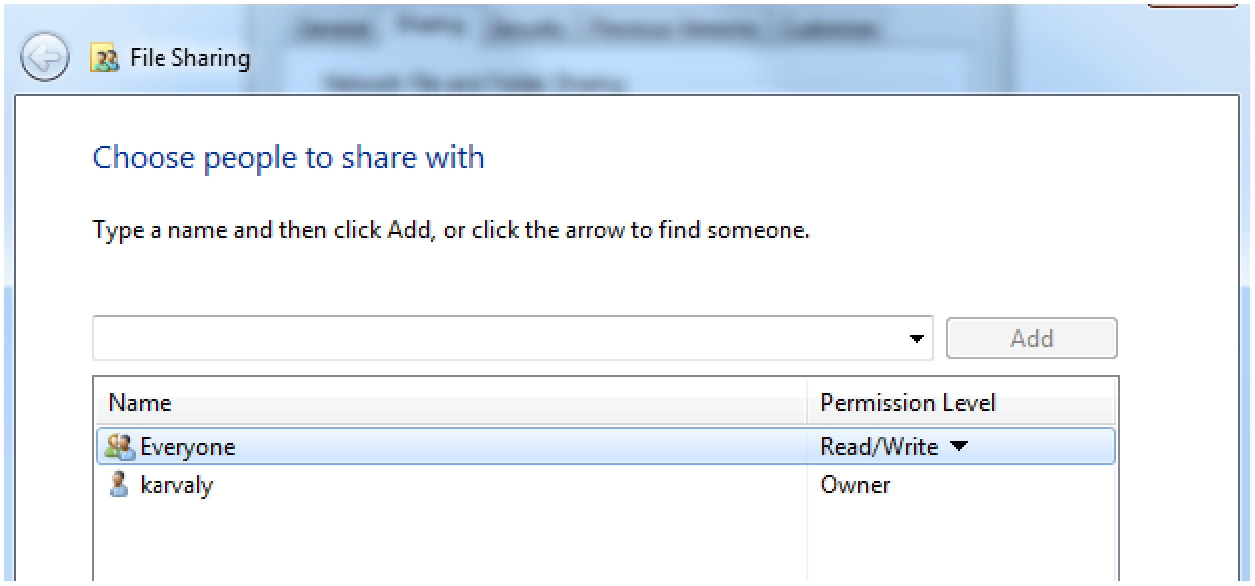
Depending on the scheduling of the jobs, there is a latency between the two databases. This latency can be exploited when the primary database is modified by mistake.

In our scenario, we create a warm backup of the nw database as follows. The primary and secondary servers run normally on different machines, but in our demo we will use the same machine and two server instances, the Primary and the Secondary.

1. Create a folder for the storing of the logs and another where the copy will be placed.



2. Set sharing for *both* folders to shared to everyone (Properties/Sharing/Share). Type Everyone and set Read/Write.

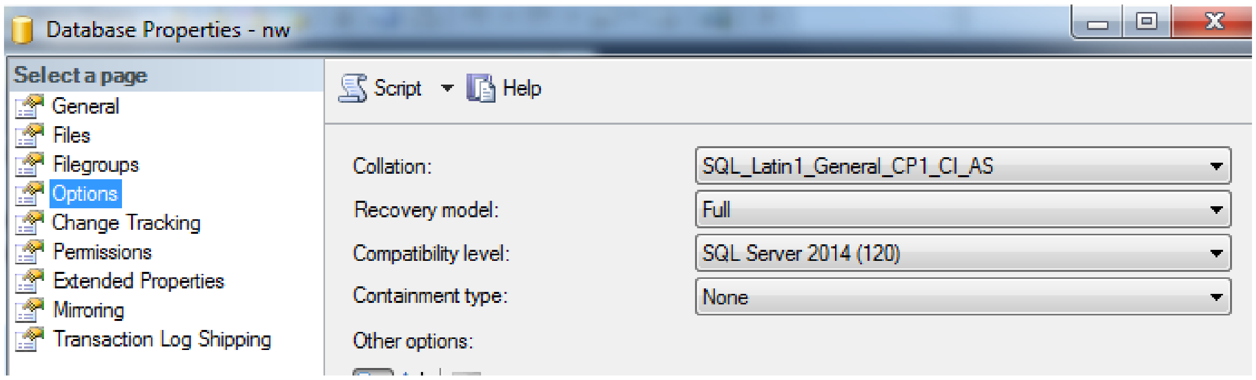


Choose people to share with

Type a name and then click Add, or click the arrow to find someone.

Name	Permission Level
Everyone	Read/Write
karvaly	Owner

3. In the pop-up menu of the nw database on the Primary server, select Properties and verify that the recovery model of the nw database is set to Full.



Collation: SQL_Latin1_General_CP1_CI_AS

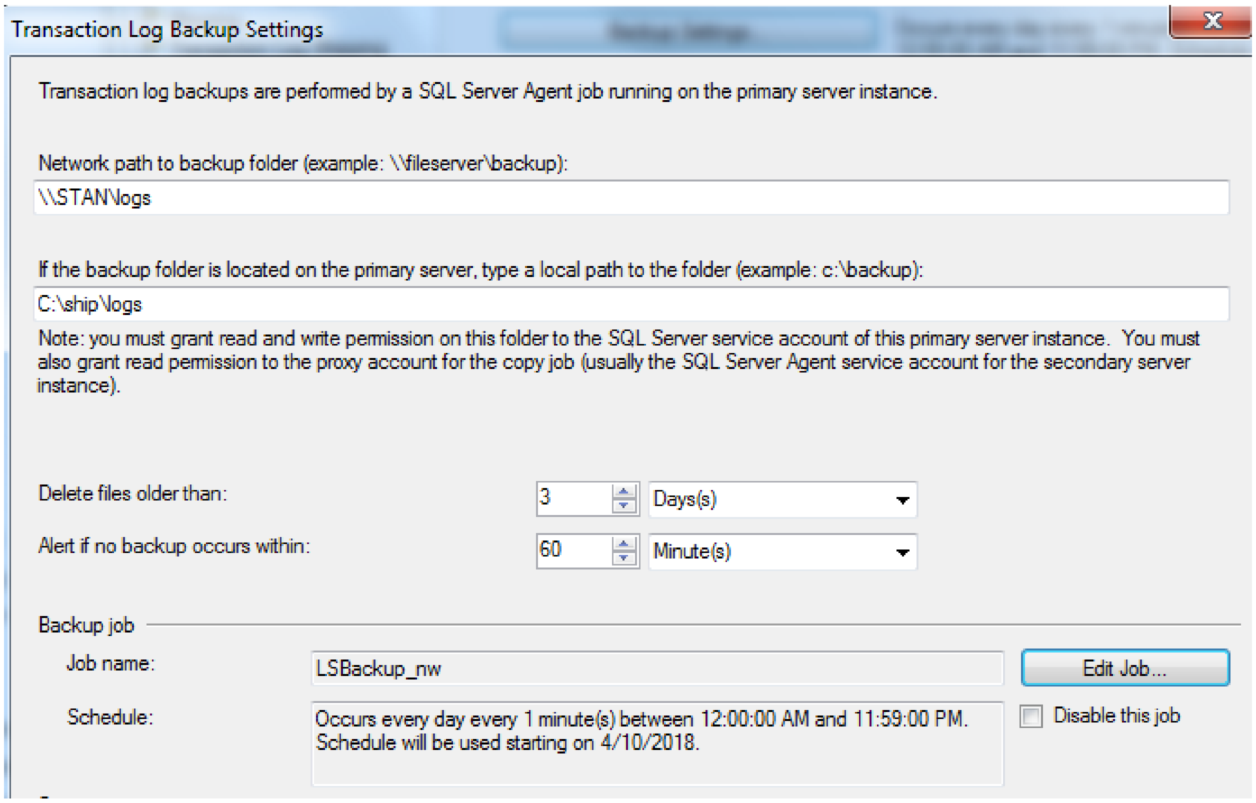
Recovery model: Full

Compatibility level: SQL Server 2014 (120)

Containment type: None

Other options:

4. In the Transaction log shipping group, enable the database for shipping, and select Backup settings. Type \\STAN\logs as the network path of the backup folder and C:\ship\logs as the local folder path.



Transaction log backups are performed by a SQL Server Agent job running on the primary server instance.

Network path to backup folder (example: \\fileserver\backup):
\\STAN\logs

If the backup folder is located on the primary server, type a local path to the folder (example: c:\backup):
C:\ship\logs

Note: you must grant read and write permission on this folder to the SQL Server service account of this primary server instance. You must also grant read permission to the proxy account for the copy job (usually the SQL Server Agent service account for the secondary server instance).

Delete files older than: 3 Days(s)

Alert if no backup occurs within: 60 Minute(s)

Backup job

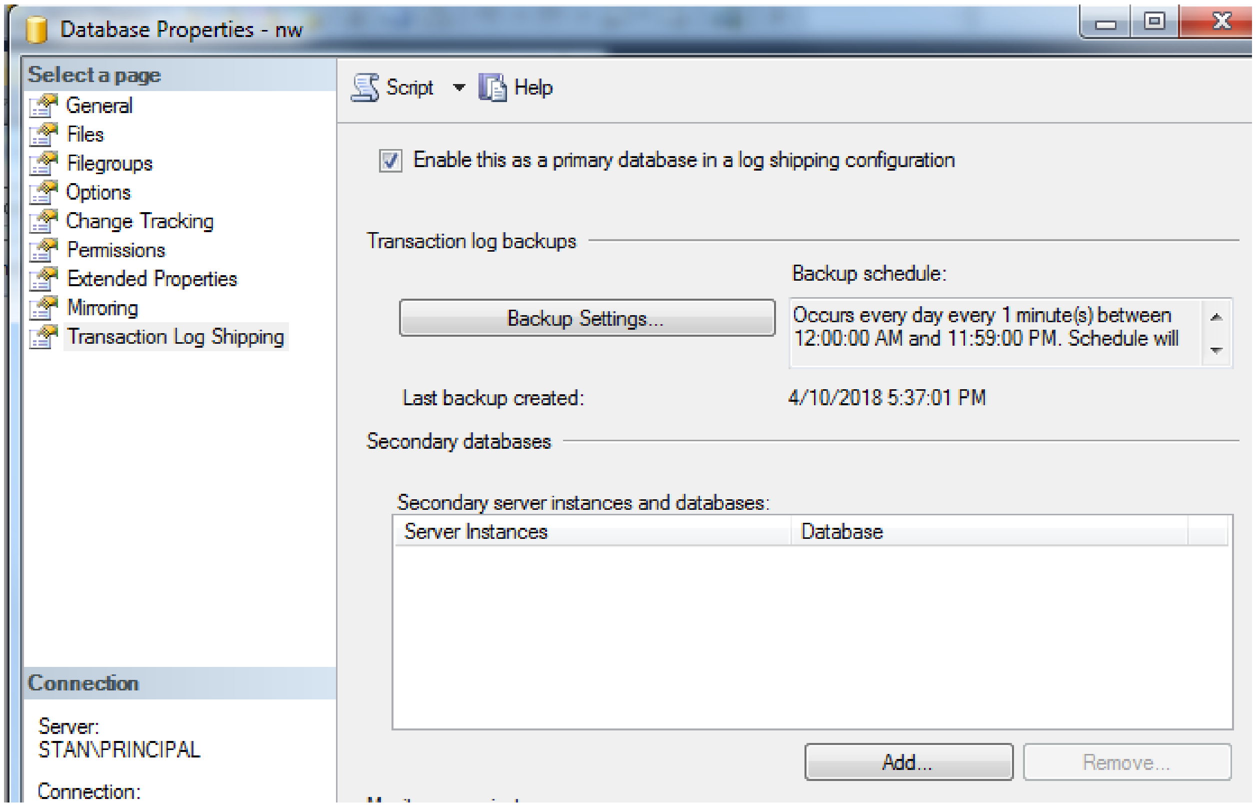
Job name: LSBackup_nw

Schedule: Occurs every day every 1 minute(s) between 12:00:00 AM and 11:59:00 PM. Schedule will be used starting on 4/10/2018.

[ ] Disable this job

5. Select Edit job and set a schedule that runs the job every minute. *Note: this setting is only for demo purposes.*

6. Return to the Database properties panel and select Add in the Secondary databases section:



Database Properties - nw

Server:
STAN\PRINCIPAL

Connection:

☑ Enable this as a primary database in a log shipping configuration

Transaction log backups

Backup schedule:

Occurs every day every 1 minute(s) between 12:00:00 AM and 11:59:00 PM. Schedule will

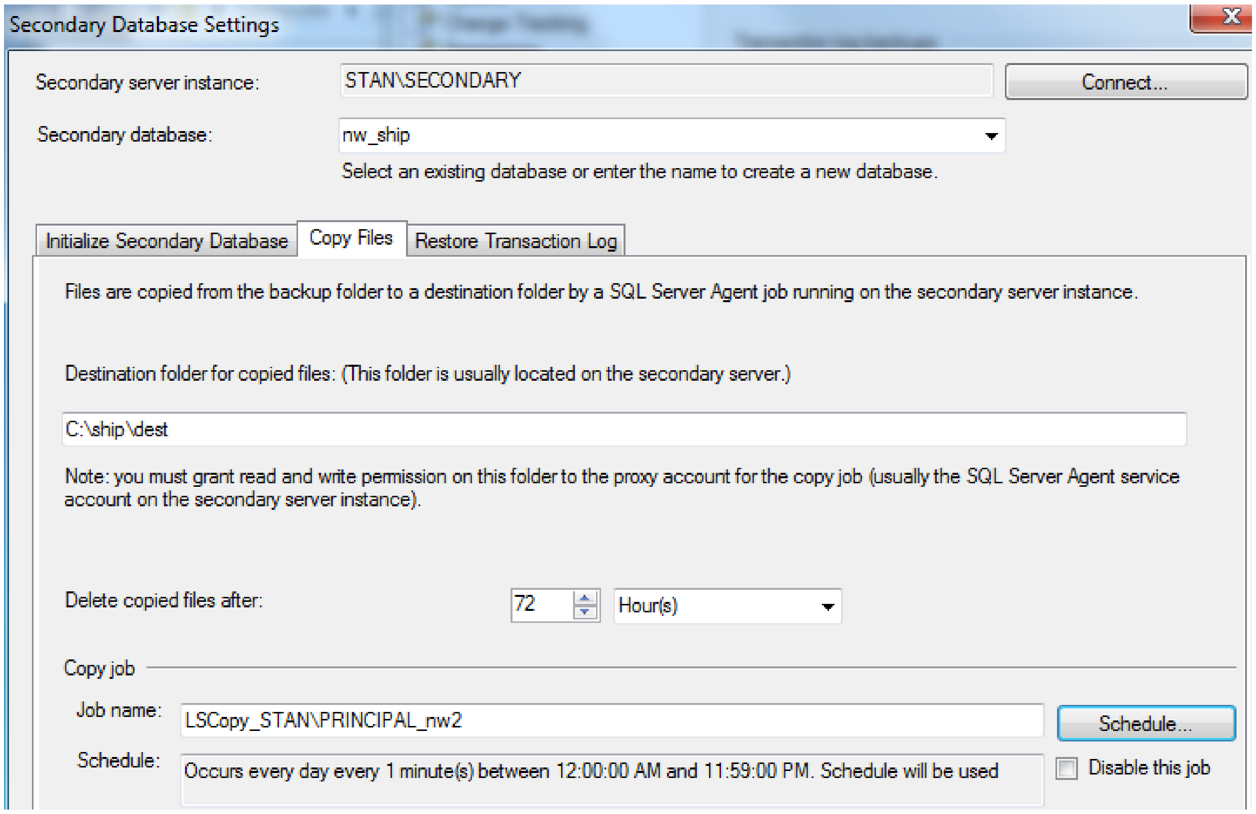
Last backup created: 4/10/2018 5:37:01 PM

Secondary databases

Secondary server instances and databases:

Server Instances	Database

7. On the Secondary database settings panel, type the new database name nw_ship. Accept the default for initialization. In the Copy files tab, enter C:\ship\dest as the destination path. Set the schedule for every 1 minute. *Note: this setting is only for demo purposes*.



Secondary server instance: STAN\SECONDARY

Secondary database: nw_ship

Select an existing database or enter the name to create a new database.

Files are copied from the backup folder to a destination folder by a SQL Server Agent job running on the secondary server instance.

Destination folder for copied files: (This folder is usually located on the secondary server.)

C:\ship\dest

Note: you must grant read and write permission on this folder to the proxy account for the copy job (usually the SQL Server Agent service account on the secondary server instance).

Delete copied files after: 72 Hour(s)

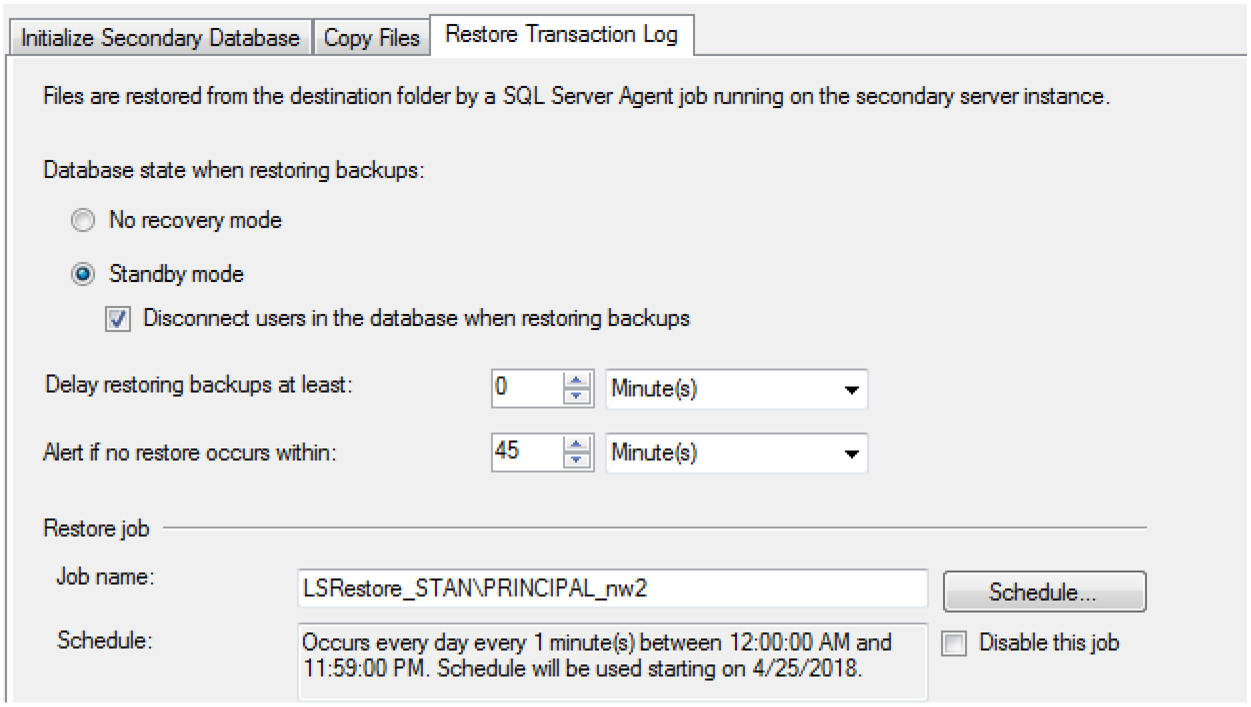
Copy job

Job name: LSCopy_STAN\PRINCIPAL_nw2

Schedule: Occurs every day every 1 minute(s) between 12:00:00 AM and 11:59:00 PM. Schedule will be used

☐ Disable this job

8. On the Restore tab, select Standby mode and Disconnect users and also specify a restore schedule of every 1 minute. *Note: this setting is only for demo purposes*.



Files are restored from the destination folder by a SQL Server Agent job running on the secondary server instance.

Database state when restoring backups:

○ No recovery mode

● Standby mode

☑ Disconnect users in the database when restoring backups

Delay restoring backups at least: 0 Minute(s)

Alert if no restore occurs within: 45 Minute(s)

Restore job

Job name: LSRestore_STAN\PRINCIPAL_nw2

Schedule: Occurs every day every 1 minute(s) between 12:00:00 AM and 11:59:00 PM. Schedule will be used starting on 4/25/2018.

☐ Disable this job

9. Finalize and run the configuration.

10. Verify the correct operation by connecting to both databases. You may need to wait 3 minutes to see the changes in the secondary database.

11. Disable the log shipping on both servers. Since the shipped database on Secondary is in warm standby state, you must execute the following commands before you can drop it:

```
use master
alter database nw_ship set single_user with rollback immediate
restore database nw_ship with recovery
```

4. Database administration and maintenance

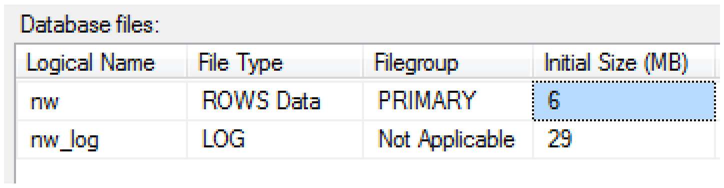
The most common tasks of relational database administration are as follows.

- Database file management
- Maintaining database performance—solving issues like file fragmentation and re-computing table statistics
- User and security management i.e. login roles, privileges, policies, and database encryption (not detailed here)
- Configuring alerts for critical conditions
- Implementing a backup strategy

These elements may be implemented separately, or may be combined in a single database maintenance plan.

Database files

Each database must have at least two files associated with it, one containing the database objects (mdf file) and the other containing the transaction log (ldf file). It is the log file that is most critical with respect to failover recovery, so this file should be stored on redundant media, e.g. on a RAID array.



Database files:

Logical Name	File Type	Filegroup	Initial Size (MB)
nw	ROWS Data	PRIMARY	6
nw_log	LOG	Not Applicable	29

In order to avoid fragmentation, the files are best placed on volumes that no other programs use. The initial size of the database can be estimated based on the planned application. Too small amounts of auto-grow may also lead to fragmentation.

In order to check database integrity it is a good practice to run DBCC CHECKDB ('DB_name') WITH NO_INFOMSGS, ALL_ERRORMSGS, as often as a full backup is created. Any output means a corruption of the database.

Database performance

Select the right recovery mode. Use the Full mode only if it is really needed by the application. Make sure that the following database options are set:

- Auto update statistics = TRUE
- Auto create statistics = TRUE
- Auto shrink = FALSE (if possible, do not use table or database shrinking at all)
- Page verify = CHECKSUM

Tuning a database for an application is beyond the scope of this course.

Alerts

It is a good practice to set up an alert for all severity 24 errors. Another classic critical condition is when the volume holding the database files is running low on free space.

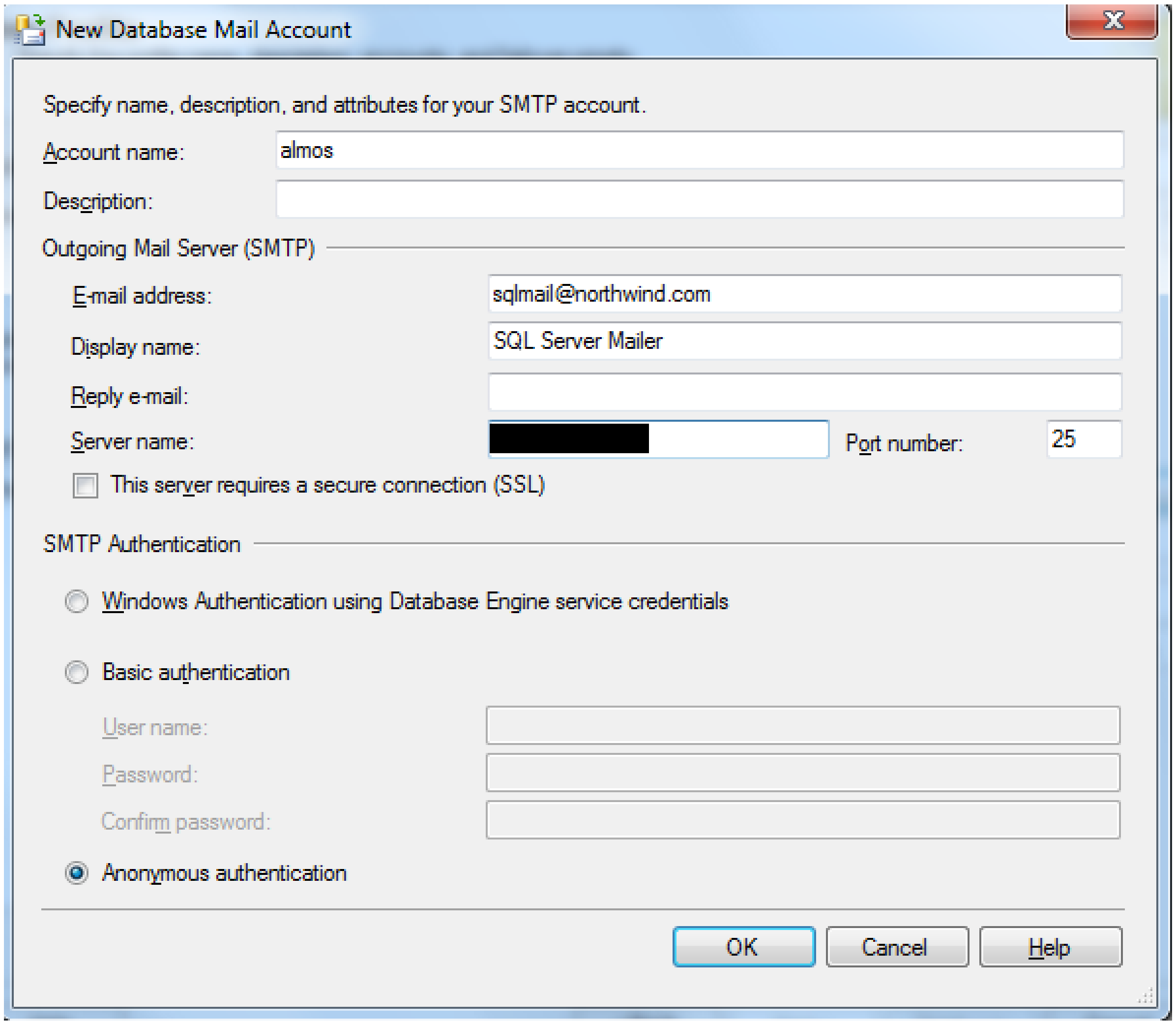
PRACTICE: In this demo, we check the current size of the nw database in the C:\Program Files\Microsoft SQL Server\MSSQL12.PRINCIPAL\MSSQL\DATA folder: 6.4 MB. For the demo, we set up an alert that fires when the size hits 150% of the current size (in our case 9 MB)

The process has 4 steps:

1. Set up a database mail profile
2. Enable the mail profile in SQL server agent that will run the alert
3. Create an operator (a person who will receive the alert and resolve the issue)
4. Set up and test the alert

Setting up database mail

Select Server -> Management node -> Database mail ->Configure Database mail, then in the New profile panel, type principal_mail for profile name, then click Add SMTP account. Enter the name of your SMTP server.


New Database Mail Account

Specify name, description, and attributes for your SMTP account.

Account name: almos

Description:

Outgoing Mail Server (SMTP)

E-mail address: sqlmail@northwind.com

Display name: SQL Server Mailer

Reply e-mail:

Server name: [redacted] Port number: 25

☐ This server requires a secure connection (SSL)

SMTP Authentication

○ Windows Authentication using Database Engine service credentials

○ Basic authentication

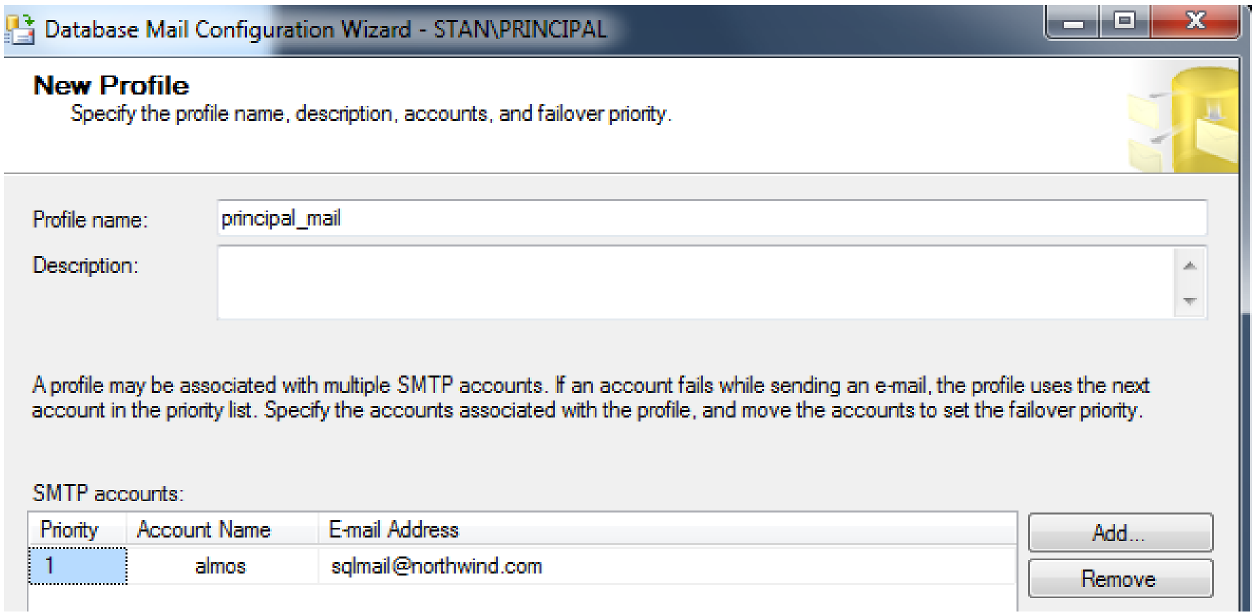
User name:

Password:

Confirm password:

● Anonymous authentication

Then you must specify the name of the new mail profile that will use the SMTP account:



Database Mail Configuration Wizard - STAN\PRINCIPAL

New Profile

Specify the profile name, description, accounts, and failover priority.

Profile name: principal_mail

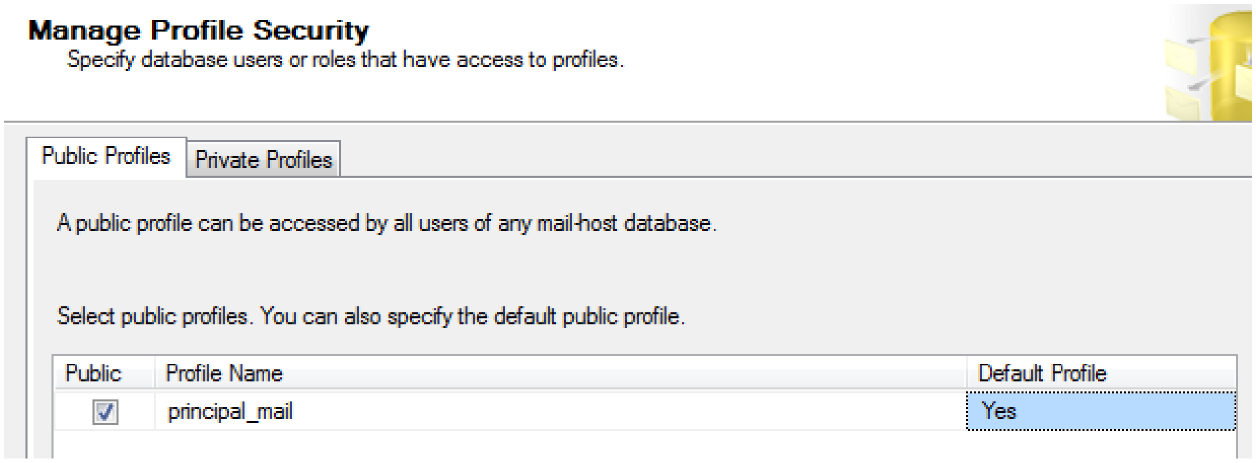
Description:

A profile may be associated with multiple SMTP accounts. If an account fails while sending an e-mail, the profile uses the next account in the priority list. Specify the accounts associated with the profile, and move the accounts to set the failover priority.

SMTP accounts:

Priority	Account Name	E-mail Address
1	almos	sqlmail@northwind.com

By making the profile public, every user can use it for sending emails:



Manage Profile Security

Specify database users or roles that have access to profiles.

A public profile can be accessed by all users of any mail-host database.

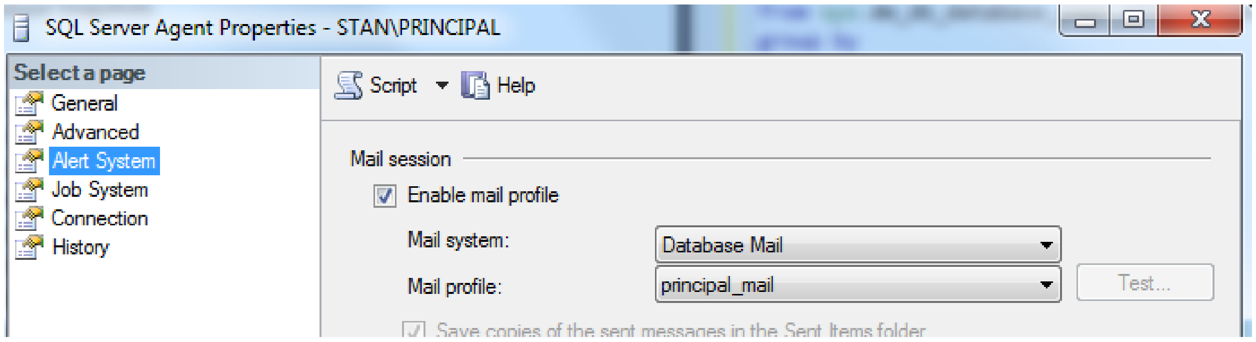
Select public profiles. You can also specify the default public profile.

Public	Profile Name	Default Profile
✓	principal_mail	Yes

Verify that the profile works by Database mail -> Send test E-mail. Check that the email is received as expected.

Enabling the mail profile in SQL server agent

Select Properties from the SQL server agent node popup menu:



SQL Server Agent Properties - STAN\PRINCIPAL

Mail session

✓ Enable mail profile

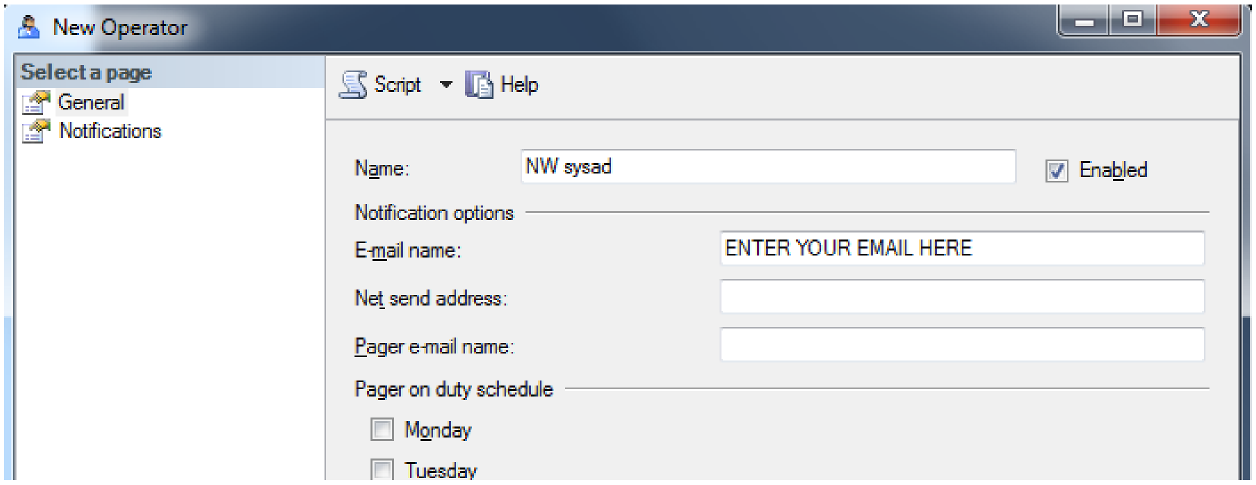
Mail system: Database Mail

Mail profile: principal_mail

✓ Save copies of the sent messages in the Sent Items folder

Creating an operator

Select Operators from the SQL server agent node and add a new operator. In the E-mail name text box enter your own email address.



New Operator

Select a page: General, Notifications

Name: NW sysad   ☑ Enabled

Notification options

E-mail name: ENTER YOUR EMAIL HERE

Net send address:

Pager e-mail name:

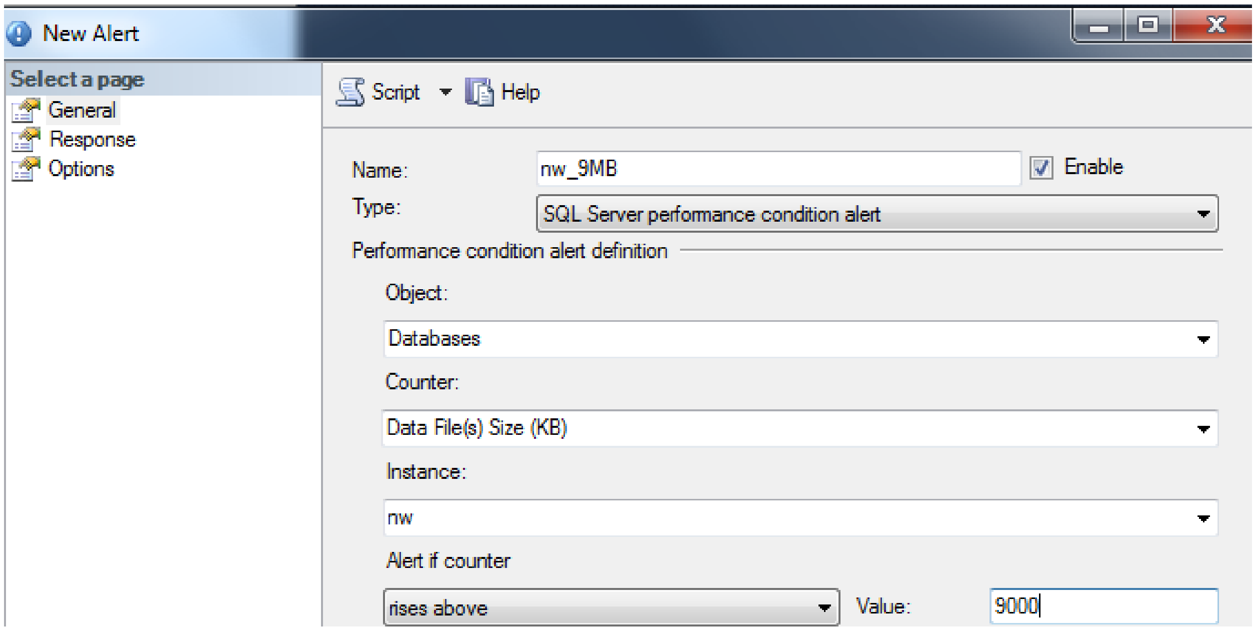
Pager on duty schedule

☐ Monday

☐ Tuesday

Adding the alert

Select Alerts from the SQL server agent node and add a new alert:



New Alert

Select a page: General, Response, Options

Name: nw_9MB   ☑ Enable

Type: SQL Server performance condition alert

Performance condition alert definition

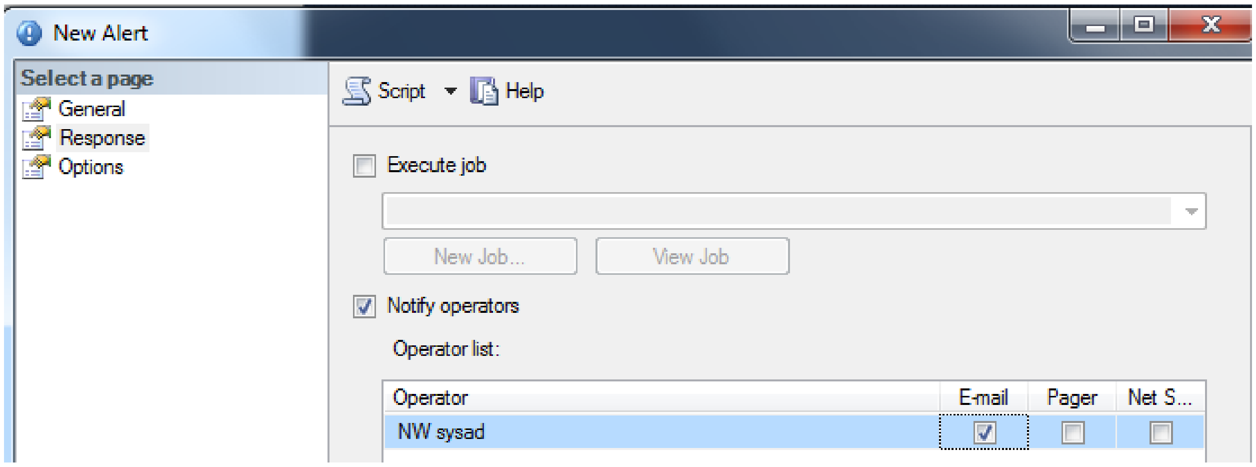
Object: Databases

Counter: Data File(s) Size (KB)

Instance: nw

Alert if counter: rises above   Value: 9000

Set the response to Notify operator (NW system administrator) by email:



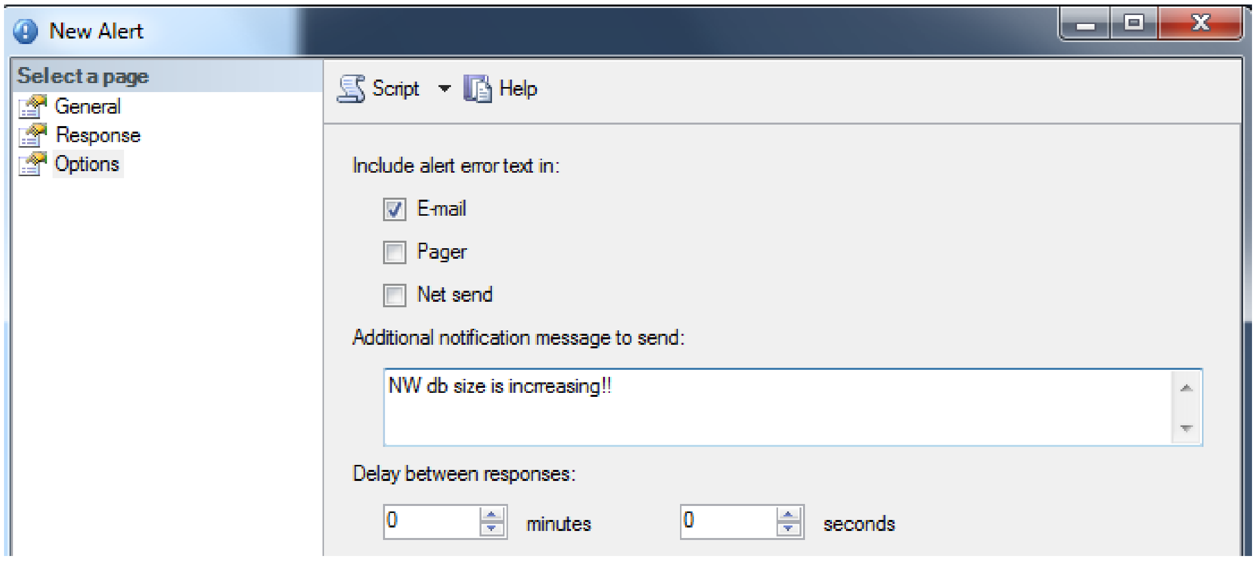
Execute job [ ]

Notify operators [✓]

Operator list:

Operator	E-mail	Pager	Net S...
NW sysad	✓		

Add a custom message. If you set the Delay between responses to 0, the response will be repeated continuously as long as the alert condition is satisfied. *We only use this setting for demo purposes.*



Include alert error text in:

E-mail [✓]

Pager [ ]

Net send [ ]

Additional notification message to send:

NW db size is increasing!!

Delay between responses:

0 minutes 0 seconds

Test the alert with the following script.

```
select  object_name(object_id) as 'tablename',
       count(*) as 'totalpages',
       sum(Case when is_allocated=0 then 1 else 0 end) as 'unusedPages',
       sum(Case when is_allocated=1 then 1 else 0 end) as 'usedPages'
from sys.dm_db_database_page_allocations(db_id(),null,null,null,'DETAILED')
group by
object_name(object_id)
--we will create a big table
go
create table big_table (a char(4000))
declare @i int=0
while @i<1000 begin
       insert big_table values ('a')
       set @i=@i+1
end
--big_table has 500 pages -> alert is fired
```

Check the alert history and your mailbox. The alert mails are coming repeatedly. Then drop the big_table and disable or delete the alert.

Backups

Taking regular backups is a common way to support disaster recovery. Backup files are to be stored on a media different form that of the database and log files. The three most common strategies

#1. The minimum: use full backups in SIMPLE recovery mode, e.g. on a daily basis in off-load periods. The backups should be stored and re-written in a round-robin fashion. In this way the data loss can be limited to one day. The simple recovery mode means that he inactive parts of the log are regularly truncated.

#2. Use full backups and differential backups in SIMPLE recovery mode, e.g. a daily full backup and differential backups every 2 hours. In this way the data loss can be limited to two hours.

#3. If the application requires that the possibility of data loss be minimized, the database must be set to FULL recovery mode and the second strategy must be combined with transaction log backups. Full recovery mode means that the log may grow substantially. An example strategy is a daily full backup, differential backups every 2 hours, and a transaction log backup every 20 minutes. If the data file is lost, all the committed transactions will be preserved.

Backup jobs can be integrated in a database maintenance plan.

PRACTICE: We simulate a disk error. We create a full backup of the nw database in file device, stop the SQL Server Principal service, delete the database file, start the service again and restore from backup, using the WITH REPLACE option. Check the contents.

Maintenance plans

Before the plan can be created, we must enable the use of extended stored procedures for SQL Server agent:

```
use master
sp_configure 'show advanced options', 1
go
reconfigure
go
sp_configure 'Agent XPs', 1
go
reconfigure
go
```

PRACTICE: Implement the backup strategy #3 in a new maintenance plan by setting the right schedule for each task. Also include index re-computing and integrity checking as tasks to be executed on a daily basis.

5. APPENDIX: SQL examples for self-learning

```
use NORTHWIND
select * from employees
select lastname, birthdate from employees

--the name of those customers who are located in London
select companyname, city
from customers
--where city LIKE 'L%' and (city LIKE '%b%' or city LIKE '%n%') --partial matching
where city IN ('London', 'Lander')
where city ='London' or city ='Lander'

where city IN ('London')
where city = 'London'

--who is the youngest employee? What is her name?
--1/2) the maximal birthdate
--aggregate functions: max, min, avg, std, sum, count
select max(birthdate) as max_year, min(birthdate) as min_year
--, lastname --would be an error
from employees

--2/2) embed this query
select lastname, birthdate from employees
where birthdate = ('1966-01-27 00:00:00.000')

select lastname, birthdate as "birth date" from employees
where birthdate = (
        select max(birthdate) as max_year
        from employees
)

--PROBLEM: find the ShipAddress of the first order
select orderdate, shipaddress from orders
where orderdate = (
        select min(orderdate) as min_date
        from orders
)

--ship addresses of the youngest employee
--joining tables
select distinct lastname, shipaddress
from orders o inner join employees e on o.employeeid=e.employeeid
where e.employeeid = (
        select employeeid from employees
        where birthdate = (
                select max(birthdate) as max_year
                from employees
        )
)
order by shipaddress --desc


--which products were ordered form the youngest employee
--note: always start with the FROM part of the query
select distinct p.productname, e.lastname
from orders o inner join employees e on o.employeeid=e.employeeid
        inner join [order details] od on od.orderid=o.orderid
        inner join products p on p.productid=od.productid
```

```
where e.employeeid=9  --she is the youngest
order by productname

--PROBLEM: which are the ship cities of products with CategoryID=1?
select distinct o.shipcity
from orders o inner join [order details] od on od.orderid=o.orderid
       inner join products p on p.productid=od.productid
where p.categoryid = 1  --our search conditon
order by shipcity


--No. of orders per employee?
--1/5) GROUPING
select employeeid, count(*)
from orders
group by employeeid

--a note aside: how to test for null?
select * from orders where employeeid is null
delete from orders where employeeid is null

--2/5) DO NOT DO THIS:
select e.lastname, count(*)
from orders o inner join employees e on o.employeeid=e.employeeid
group by e.lastname
--results in logical error if there are
--2 persons with the same lastname!!!

--3/5)
select e.lastname, e.firstname, count(*)
from orders o inner join employees e on o.employeeid=e.employeeid
group by e.employeeid, e.lastname, e.firstname
--this query misses the agent with no orders

--PROBLEM: list the number of products in each Category (we need the CategoryName also)
3. select c.categoryid, c.categoryname, count(*) as no_prod
1. from products p inner join categories c on p.categoryid=c.categoryid
2. group by c.categoryid, c.categoryname
4. order by no_prod desc

--4/5)
select e.employeeid, e.lastname, e.firstname, count(*)
from employees e left outer join orders o on o.employeeid=e.employeeid
group by e.employeeid, e.lastname, e.firstname
--problem: we have a fake count of 1 for the idle agent

--5/5)
select e.employeeid, e.lastname, e.firstname, count(o.orderid) as no_ord
from employees e left outer join orders o on o.employeeid=e.employeeid
group by e.employeeid, e.lastname, e.firstname
order by no_ord desc
--all problems solved

--who is whose boss
select e.lastname, boss.lastname as boss, bboss.lastname as boss_of_boss
from employees e left outer join employees boss on e.reportsto=boss.employeeid
       left outer join employees bboss on boss.reportsto=bboss.employeeid


--No. of orders per employee?
```

```
select e.lastname, count(orderid)
--count(*) would produce an order for Lamer who has no order at all
from employees e left outer join orders o on
--from employees e inner join orders o on
e.employeeid = o.employeeid
group by e.employeeid, e.lastname
order by count(*) desc

--who has no orders?
select e.*
from employees e left outer join orders o on
e.employeeid = o.employeeid
where o.orderid is null

--which is the biggest order?
--arithmetics

4. select o.orderid,
    cast(o.orderdate as varchar(50)) as order_date,
    str(sum((1-discount)*unitprice*quantity), 15, 2) as order_total,
    sum(quantity) as no_of_units,
    count(d.orderid) as no_of_items
1. from orders o inner join [order details] d on o.orderid=d.orderid
2. where...
3. group by o.orderid, o.orderdate
--order by o.orderdate
5. order by sum((1-discount)*unitprice*quantity) desc

--in order to order by date:  group by o.orderid, o.orderdate

--who's the most successful agent? with how many orders?
-- observe: having
-- count distinct
-- formatting numbers

select  u.titleofcourtesy+' '+u.lastname+' '+ u.firstname +' ('+u.title +')'  as name,
--select u.lastname as name,
str(sum((1-discount)*unitprice*quantity), 15, 2) as cash_income,
count(distinct o.orderid) as no_of_orders, count(productid) as no_of_items
from orders o inner join [order details] d on o.orderid=d.orderid
    inner join employees u on u.employeeid=o.employeeid
group by u.employeeid, u.titleofcourtesy, u.title, u.lastname, u.firstname

--having count(o.orderid)>200 -if we are only interested in agents with more than 200 orders
order by cash_income
--sum((1-discount)*unitprice*quantity) desc

--why do we have only 9?

select count(*) from employees

--it should be 10!
--we would also need those with 0 order
-- isnull function

select  isnull(u.titleofcourtesy, '')+' '+isnull(u.lastname, '')+' '+ isnull(u.firstname, '')
+' ('+isnull(u.title, '') +')'  as name,
isnull(str(sum((1-discount)*unitprice*quantity), 15, 2), 'N/A') as cash_income,
count(distinct o.orderid) as no_of_orders, COUNT(d.productid) as no_of_items
from employees u left outer join
    (orders o inner join [order details] d on o.orderid=d.orderid)
```

```
on u.employeeid=o.employeeid
--where u.titleofcourtesy='Mr.' -if we are only interested in men
group by u.employeeid, u.titleofcourtesy, u.title, u.lastname, u.firstname
order by sum((1-discount)*unitprice*quantity) desc

--which is the most popular product?
-- top 1

select top 1 p.productid, p.productname, count(*) as no_app,
    sum(quantity) as total_pieces
from products p left outer join [order details] d on p.productid=d.productid
group by p.productid, p.productname
order by no_app desc

--which agent sold the most of the most popular product?
--first version
 select top 1 u.titleofcourtesy+' '+u.lastname+' '+ u.firstname +' ('+u.title +')'  as name,
    sum(quantity) as no_pieces_sold
 from orders o inner join [order details] d on o.orderid=d.orderid
    inner join employees u on u.employeeid=o.employeeid
 where d.productid = 59 --we know this already
 group by u.employeeid, u.titleofcourtesy, u.title, u.lastname, u.firstname
-- having....
 order by sum(quantity) desc

/**************************************************************************
PROBLEM

--which agent sold the most of the most popular product, and what is the name of that product?
--in the pubs_access database: which is the most frequnted publisher of the author with the
most publications?

***************************************************************************/

--MULTI LEVEL GROUPING
--datetime fUNCTIONS
select 2
select getdate() --datetime data type
select DATEDIFF(s,'2013-10-10 12:13:50.370', '2013-10-10 14:16:50.370')
select DATEADD(s, 1000, '2013-10-10 14:16:50.370')
select YEAR(getdate()), MONTH(getdate())

--ORDERS BY MONTH AND AGENT
select e.employeeid, lastname, year(orderdate) as year, month(orderdate) as month,
count(orderid) as no_of_orders
from employees e left outer join orders o on e.employeeid=o.employeeid
group by e.employeeid, lastname, year(orderdate), month(orderdate)
order by lastname, year, month

--the same in another way:
select e.employeeid, lastname,
cast(year(orderdate) as varchar(4)) +'_'+  cast(month(orderdate) as char(2)) as month,
count(orderid) as no_of_orders
from employees e left outer join orders o on e.employeeid=o.employeeid
group by e.employeeid, lastname, cast(year(orderdate) as varchar(4)) +'_'+
cast(month(orderdate) as char(2))
order by lastname, month

--select case

select e.employeeid, lastname,
case
    when month(orderdate) < 10 then cast(year(orderdate) as varchar(4)) +'_0'+
cast(month(orderdate) as char(2))
    when month(orderdate) >= 10 then cast(year(orderdate) as varchar(4)) +'_'+
cast(month(orderdate) as char(2))
    else 'N.A'
end as month,
count(orderid) as no_of_orders
from employees e left outer join orders o on e.employeeid=o.employeeid
group by e.employeeid, lastname,
case
    when month(orderdate) < 10 then cast(year(orderdate) as varchar(4)) +'_0'+
cast(month(orderdate) as char(2))
    when month(orderdate) >= 10 then cast(year(orderdate) as varchar(4)) +'_'+
cast(month(orderdate) as char(2))
    else 'N.A'
end --a function serves better for this purpose
order by lastname, month

--using temp tables

select GETDATE() as ido into #uj_tabla
select * from #uj_tabla
drop table #uj_tabla

select * into #uj_tabla from employees


--drop table #tt

select e.employeeid, lastname, year(orderdate) as ev, month(orderdate) as month,
count(orderid) as no_of_orders
into #tt
from employees e left outer join orders o on e.employeeid=o.employeeid
group by e.employeeid, lastname, year(orderdate), month(orderdate)
order by lastname, month

select * from #tt

--Warning: Null value is eliminated by an aggregate or other SET operation.
--reason: an aggregate function(max,sum,avg..) exists on null values

select * from #tt

select lastname, str(avg(cast(rend_szam as float)), 15, 2) as avg_no_of_orders
--select lastname, avg(rend_szam) as avg_no_of_orders
from #tt group by employeeid, lastname
order by atlagos_rend_szam desc

--another solution for the same problem with an embedded query

select forras.lastname, str(avg(cast(forras.rend_szam as float)), 15, 2) as avg_no_of_orders
from (
       select e.employeeid, lastname, year(orderdate) as ev, month(orderdate) as month,
count(orderid) as no_of_orders
       from employees e left outer join orders o on e.employeeid=o.employeeid
       group by e.employeeid, lastname, year(orderdate), month(orderdate)
) as f -using an alias is compulsory
group by employeeid, lastname
order by avg_no_of_orders desc
```

```sql
--HOMEWORK PROBLEMS:

--AVG monthly number of orders for all products?

--Who had more than double order total compared to his boss?
```