milan@datajoin.net

http://datajoin.net

DataJoin

Synchronize data across multiple database types

[Document subtitle]

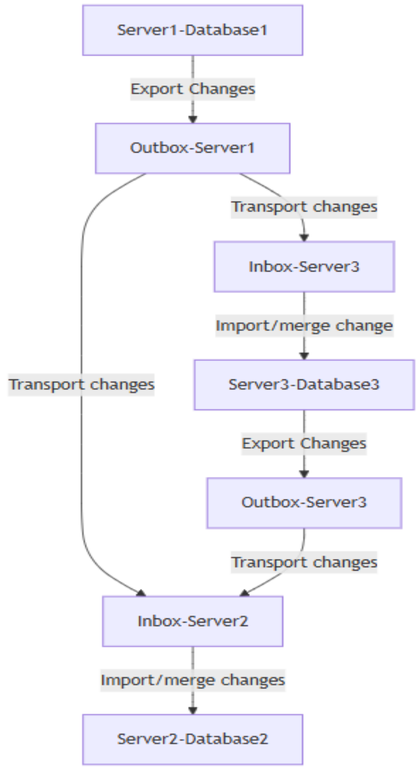
Milan Patel
5-2-2025

Overview

dj6_feature_2:overview
Problem: An application uses multiple brands of databases on various hardware sizes and operating systems. It needs to synchronize data for every insert, update, and delete between the databases. This data synchronization can be at attribute value level.

Solution: To achieve database synchronization, consider various technologies such as application API, database triggers, querying, ODBC/JDBC, and database-specific replication. There are three basic steps: 1. Change capture at attribute / column value level 2. Transport changes to target servers/databases 3. Import / merge change at target servers / databases.

DataJoin.net provides in-depth education and consulting on database synchronization.
https://github.com/milan888-design/synchronize-data



Data synchronization concepts

dj6_feature_2:concepts

Difference between integration, replication and synchronization

- Integration – data exchange between two different applications
- Replication – Data replicated within the same database type across various database installation
- Synchronization – Data update, delete and inserts are synchronized across different type of databases and across different size of servers. This is typically, within one application with one data model but across different machines (larger server, laptop, handheld)

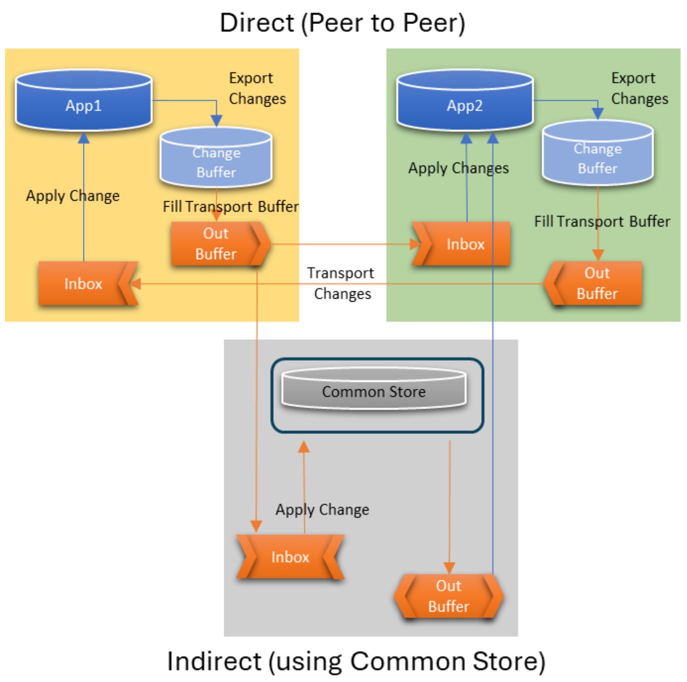
Components of synchronization – Capture change, transport change, import change.
Method within each component:

- Capture change – CRUD API, database trigger, database CDC (change data capture) feature.
  SQL database triggers are at column value level for update and row level for “delete” and ” insert”. CRUID API of an application can be at attribute/ column value level or it can be at object (row) level. However, CRUD API should be the same across servers for synchronization.
- Transport change – Queuing (Kafka, etc.), database to database exchange using ODBC, JDBC. Data security varies between querying and ODBC/JDBC, thus, it should be reconciled with data security policy of a company.
- Import change – Database update, insert and delete, API

Major challenges in data synchronization:

overwrite prevention- old data should not overwrite new data
echo prevention – same data must not go to multiple servers multiple times over and over again.
Server date time – Server date time must be same across server to timestamp change.

Synchronization components



dj6_feature_2:components

Synchronization solution components: Capture change

dj6_feature_2:components:capturechange

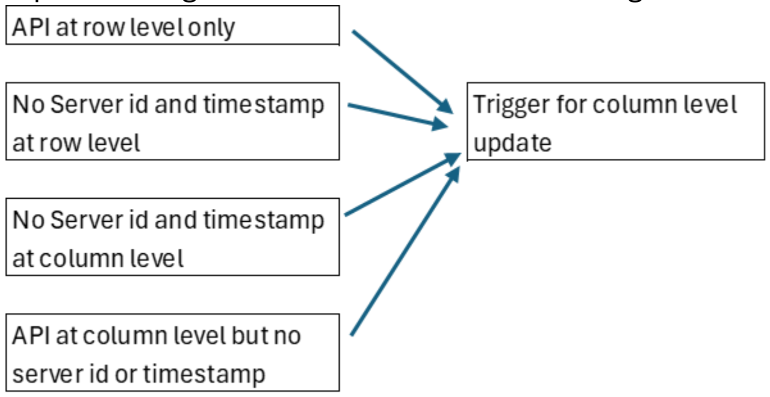
The scripts for "insert, update, delete" can be API or SQL code. The data is changed when these scripts are run. These scripts are stored in table called "Outbox" (refer section name Synchronization Code). If these scripts are run in the same sequence on the other servers with the same table, then, that table will have the same data. This is data synchronization. Change capture should capture changes that occur on local server as original data created, or changes occurring using UI (user interface) by end user. The changes data that is imported from "inbox" are not considered "local changes" since it came from other servers. All local changes should be marked with local server's server id. Thus, the changes in the "outbox" that will be sent to other servers will have local server id in the "outbox" row. Application API or other data change methods should make sure it uses local server id when change has occurred locally. All imported data from "inbox" will have server id of the server where that changed occurred.

CRUD API

API can be at database row level, or it can be at attribute level. API are for “update”, “delete” and “insert”.

Database trigger for column/ attribute level changes

API may not be at column level. Also, server id and timestamp may not be in tables. Thus, the column level trigger for update is the only way capture changes at column level. The following shows why database trigger has to be used since there is no other alternatives.


API at row level only
No Server id and timestamp at row level
No Server id and timestamp at column level
API at column level but no server id or timestamp
Trigger for column level update

Synchronization solution components: Transport change

dj6_feature_2:components:transportchange

Transport Program sends the data from “outbox” (change scripts of the local server) to “inbox” (refer section name Synchronization Code) to other servers. Program Parameter table to keep track of transport from outbox to other servers. One can run the transport program with any frequency. The frequency depends upon speed at which changes occurring and how fast changes need to be synchronized with other servers. If this program is running on local server and sending data to remote servers, then, it can be called “push”. If remote servers retrieving data from the local server, then, it can be called “pull”. If the changes are sent to another server that is not a common server, then, it can be called direct / peer to peer. If changes are sent to a common server, then, all servers pull the changes the common server, then, it is called indirect / common server. Transport program sends that data from “outbox” with local server id only. This is important to prevent “echo” (same changes sent to servers again and again).

Echo prevention as part of transport program
Echo is network of connected servers that exchange the same data again and again. Echo prevention is stopping first server data being sent to another server and that second server sends the same data back to first server.
Echo problem:
Server1>>>>record1 sent to >>>> server2 >>>> record1 sent to >>>> Server1

Proposed Solution:
Outbox of a server has changes that are truly originated (and not imported from other servers) at the local server. Thus, transport programs only pickup changes from the outbox that are truly originated locally. Transport program uses server id to select program.

Server1>>>>record1 sent to >>>> server2 >>>> DO NOT SEND record1 to >>>> Server1

It is possible to use a time stamp to keep track of what was the last row transported or imported. However, sequence numbers are better to know which was the last row that was transported or imported. Refer section "Synchronization Code" for tables named as ds_change_inbox and ds_change_outbox. Both tables are identical. ds_change_outbox can be sending changes to multiple other servers. ds_change_inbox can be receiving changes from multiple servers.

A program to transport changes should be configured between two servers databases. This program must process the changes and mark the last sequence number of the ds_change_outbox for specific iteration of transport, then, for the next iteration, use the sequence number to pick up another set of changes.
Here is the example,
iteration 1: picked up changes from last sequence number (or greater than) sequence to current max sequence number 10 (and below). Mark 10 as last sequence number.
Iteration 2: use last sequence number 10 (or greater than) to current max sequence number 25 (and below).

The logic is the same for ds_change_inbox to import changes in app tables.

Synchronization solution components: Import change

dj6_feature_2:components:importchange
"Inbox" has changes from other servers are stored and ready to be imported in app tables. "Inbox" has rows with changes from multiple other servers. Import program with Parameter table to track imports from inbox to application tables. Local server can decide which servers' data to be imported. Local server can decide to pull data from common store only and not the data sent by "peers". Local server can decide

the frequence and time of import. The import program has to make sure that the most recent (latest) data in local server is not overwritten by data coming from other servers. Thus, timestamp is necessary for data in “inbox”. Also, it is necessary that the time clock on servers is the same. “Insert” and “Delete” data operations do not require to check timestamp of the row in the app table. Only the “update” data operation type needs to make sure date is compared of two rows (or at attribute level) in the same table with same primary key. An application can have many tables, this, there is need to create a common view that includes all the tables so that import program uses this view to check if there is more recent row in the app table.

Advantages of transporting data to common server and importing data from common server – All server are guaranty to have the same data. In some cases, peer-to-peer data exchange can result in situations where data are not perfectly synchronized. For example, server 1 has value1 and server 2 has value 2, then, both servers are trying to synchronize, but it is possible that values are switched but they are not synchronized. Thus, server 1 will have value 2 and server 2 will have value 1. That is not correct, since synchronization means both have value 1 or value 2 depending upon which one has latest timestamp.

It is possible to use a time stamp to keep track of what was the last row transported or imported. However, sequence numbers are better to know which was the last row that was transported or imported. Refer section “Synchronization Code” for tables named as ds_change_inbox and ds_change_outbox. Both tables are identical. ds_change_outbox can be sending changes to multiple other servers. ds_change_inbox can be receiving changes from multiple servers.

A program to import changes should be configured to import changes from “Inbox” from all servers for all application tables. This program has to process the changes and mark the last sequence number of the ds_change_inbox for specific iteration of import, then, for the next iteration, use the sequence number to pick up another set of changes.
Here is the example,
iteration 1: picked up changes from last sequence number (or greater than) sequence to current max sequence number 10 (and below). Mark 10 as last sequence number.
Iteration 2: use last sequence number 10 (or greater than) to current max sequence number 25 (and below).

Overwrite prevention as part of import program

It is possible to overwrite most recent value (latest value) with old value. This can happen when a disconnected server with old data is connected with a server that has the most recent data.

Overwrite problem:
Server1 has record1 with datetime 25june>>>>
program imports record1 with datetime 24june and overwrite record1 with 25june. (newer or latest record is overwritten by older record)

Proposed solution:
Server1 has record1 with datetime 25june>>>> import program compares datetime of record1 in app table and datetime of record1 being imported.
If the incoming record1 datetime is higher than record1 datetime in app table, then, overwrite (update table set...)

Clocks of different servers are synchronized (set to the same universal time). This is the only way to know which record is the latest.

Create view for last update date covering all tables
Ds_change_inbox has insert, update and delete statements for multiple tables within a database. Thus, it is necessary to create a "view" across all tables in the database with record pk and time stamp for the record. This assumes that application tables have time stamp at record level and also, it is not necessary to have the update time stamp at column value level. It is possible to change log table at attribute/column value level. This requires a different design depending upon application API at record level or at attribute level.

```
CREATE VIEW view_recordlevel_update_datetime
 AS
SELECT
'test_app1' as object_database
,'sales_order' as object_Table
,'order_id' object_pk_attribute
,order_id as object_id
,update_datetime update_datetime
  FROM sales_order
  UNION
SELECT
'test_app1' as object_database
,'purchase_order' as object_Table
,'purchase_id' object_pk_attribute
,purchase_id as object_id
,update_datetime update_datetime
  FROM purchase_order
```

Synchronization Code

dj6_feature_2:components:code
Testapp1 and testapp2 are in PostgreSQL databases. These two databases can be in different database brands such as SQLite or SQL Server or others.
User table create and data insert part of the script. You may create database depending upon the database brand. For simplification, a table name sales_order is used for synchronization.
Follow the steps below to test the synchronization between two applications testapp1 and testapp2.

Set up two databases with two test apps.

dj6_feature_2:components:code:setuptwoapps
After the steps below are the examples of objects named in the steps such as sales_order, etc.

Setup testapp1
Step 1: create testapp1 database (use admin UI)
Step 2: create sales_order table (CREATE TABLE sales_order.sql)
Step 3: create view name view_recordlevel_update_datetime (ds_create_view_recordlevel_update_datetime.sql)
Step 4: create ds_program_parameter table (CREATE TABLE ds_program_para.sql)

Setup testapp2
Step 5: create testapp2 database (use admin UI)
Step 6: create sales_order table (do not enter rows in this table. Let the synchronization bring the data from testapp1
Step 7: create view name view_recordlevel_update_datetime (ds_create_view_recordlevel_update_datetime.sql)

Data operations for synchronization test

dj6_feature_2:components:code:test
(ds_test_insert_update_delete__from_testapp1_to_testapp2.sql)

"insert" data operation synchronization test steps

Step 1: insert one row in sales order_table
Step 2: insert Step1 data operation row in ds_outbox table

Step 3: run transport program between testapp1 and testapp2. This will transport insert operation row from outbox of testapp1 to inbox of testapp2. (program id=1000 for ds_change_transport.py)
Step 4: run import program for testapp2. (program id=1001 for ds_change_import.py). This will import a new row in sales_order table. The same row that was in testapp1.
Step 5: check that there is row in testapp2 database in sales_order table using SELECT * from sales_order;

"update" data operation synchronization test steps

Step 1: update a value in a column in sales_order_table
Step 2: insert Step1 data operation row in ds_outbox table
Step 3: run transport program between testapp1 and testapp2. This will transport update operation row from outbox of testapp1 to inbox of testapp2. (program id=1000 for ds_change_transport.py)
Step 4: run import program for testapp2. (program id=1001 for ds_change_import.py). This will update the same column value in sales_order table.
Step 5: check that there is row in testapp2 database in sales_order table using SELECT * from sales_order;

"delete" data operation synchronization test steps

Step 1: delete a row in sales order_table
Step 2: insert Step1 data operation row in ds_outbox table
Step 3: run transport program between testapp1 and testapp2. This will transport update operation row from outbox of testapp1 to inbox of testapp2. (program id=1000 for ds_change_transport.py)
Step 4: run import program for testapp2. (program id=1001 for ds_change_import.py). This will delete the same row in sales_order table.
Step 5: check that there is row in testapp2 database in sales_order table using SELECT * from sales_order;

Two programs, ds_change_transport.py and ds_change_import.py can be in recurring running mode at specified interval.

Table: sales_order

order_id	order_type	description	product_type_name	customer_id	quantity	order_date	update_datetime	updated_by_user	updated_by_server
ord1	retail	new order	laptop	c1	200	4/25/2025	4/25/2025 10:30:00 AM	mtp	server1

ord2	wholesale	old order	desktop	c1	100	5/25/2025	6/25/2025 10:30:00 AM	mtp	server1

Data operation 1 : insert row in sales_order

```
--insert in sales_order table
INSERT INTO public.sales_order(
	order_id, order_type, description, product_type_name, customer_id, quantity, order_date, update_datetime, updated_by_user, updated_by_server)
	VALUES ('ord2','wholesale','old order','desktop','c1','100','5/25/2025','06/25/2025  10:30:00 AM','mtp','server1');
```

Data operation 2: update one column in sales_order

```
--update description column in sales_order table
UPDATE sales_order set description='old order change1' where order_id='ord2'
```

Data operation 3: delete row in sales_order

```
--delete record in sales_order
delete from sales_order where order_id='ord2'
```

The above operations are used for synchronization. The following changes are occurring in table name sales_order. These changes are put in ds_change_outbox by inserting rows in that table with the following in column name object_operation_command, type of change (U for update, I for insert, and D for delete) is in object_operation column. The object_operation_command is altered to make sure single quote isreplaced with ~ (another way is to use escape character). Application API would have to do the same “insert” in the “outbox”.

After each of the following changes, run transport change program between testapp1 and testapp2.

“insert” data operation row in ds_change_outbox

```
INSERT INTO public.ds_change_outbox(
	object_database, object_table, object_pk_attribute, object_attribute, object_id, object_value, object_operation, object_operation_command, updated_by_server_id, update_datetime, updated_by_user_id, object_id_old, object_value_old, note1)
	VALUES ('testapp1', 'sales_order', 'order_id', 'description', 'ord2', 'old order', 'I', 'INSERT INTO public.sales_order(order_id, order_type, description, product_type_name, customer_id, quantity, order_date, update_datetime, updated_by_user, updated_by_server) VALUES
```

(~ord2~,~wholesale~,~old order~,~desktop~,~c1~,~100~,~5/25/2025~,~06/25/2025 10:30:00 AM~,~mtp~,~server1~);', 'server1', '06/25/2025 10:30:00 AM', 'mtp', 'tbd', 'tbd', 'tbd');

"update" data operation row in ds_change_outbox

INSERT INTO public.ds_change_outbox(
	object_database, object_table, object_pk_attribute, object_attribute, object_id, object_value, object_operation, object_operation_command, updated_by_server_id, update_datetime, updated_by_user_id, object_id_old, object_value_old, note1)
	VALUES ('testapp1', 'sales_order', 'order_id', 'description', 'ord2', 'old order', 'U', 'UPDATE sales_order set description=~old order change~ where order_id=~ord2~', 'server1', '06/25/2025 10:30:00 AM', 'mtp', 'tbd', 'tbd', 'tbd');

"delete" data operation row in ds_change_outbox

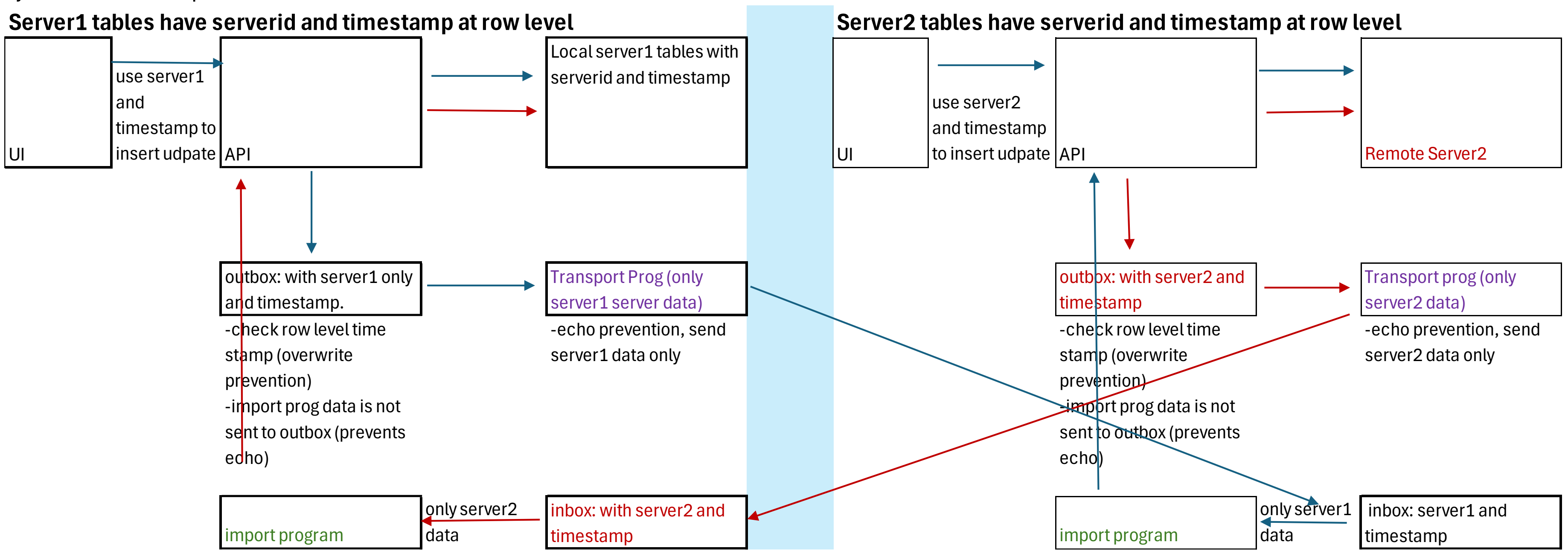
INSERT INTO public.ds_change_outbox(
	object_database, object_table, object_pk_attribute, object_attribute, object_id, object_value, object_operation, object_operation_command, updated_by_server_id, update_datetime, updated_by_user_id, object_id_old, object_value_old, note1)
	VALUES ('testapp1', 'sales_order', 'order_id', 'description', 'ord2', 'old order', 'D', 'delete from sales_order where order_id=~ord2~', 'server1', '06/25/2025 10:30:00 AM', 'mtp', 'tbd', 'tbd', 'tbd');

Table: Ds_program_parameter

id	program type	description	active_flag	status	fromserver	frommdb	fromtable	fromconnectionstring	fromoutboxpointer	toserver	todb	totable
1000	outbox to inbox transport	server1:testapp1 outbox transport to server1:testapp2 inbox	Y		server1	testapp1	ds_change_outbox	db_connectionstring1 = 'postgresql://postgres:pass@localhost:5432/'	1	server1	testapp2	ds_change_inbox
1001	inbox to apptable import	server1:testapp2 import from inbox to apptable	Y	NULL	server1	testapp2	ds_change_inbox	db_connectionstring1 = 'postgresql://postgres:pass@localhost:5432/'	1	server1	testapp2	apptable

Flow chart: How changes from testapp1 and testapp2 are synchronized

dj6_feature_2:components:code:flowchart


Server1 tables have serverid and timestamp at row level
UI
use server1 and timestamp to insert udpate
API
Local server1 tables with serverid and timestamp
outbox: with server1 only and timestamp.
-check row level time stamp (overwrite prevention)
-import prog data is not sent to outbox (prevents echo)
Transport Prog (only server1 server data)
-echo prevention, send server1 data only
import program
only server2 data
inbox: with server2 and timestamp
Server2 tables have serverid and timestamp at row level
UI
use server2 and timestamp to insert udpate
API
Remote Server2
outbox: with server2 and timestamp
-check row level time stamp (overwrite prevention)
-import prog data is not sent to outbox (prevents echo)
Transport prog (only server2 data)
-echo prevention, send server2 data only
import program
only server1 data
inbox: server1 and timestamp

ds_change_transport.py

dj6_feature_2:components:code:ds_change_tranport.py

This is simply moving data from outbox of testapp1 to inbox of testapp2

Block 0: read ds_program_parameter table for specific id for last pointer of testapp1 outbox and destination testapp2 info.
BLOCK A: Define database connection based on source and destination (testapp1 and testapp2)
BLOCK B: read the last or maximum id number from ds_change_outbox table
BLOCK C: Retrieve data between last point and maximum pointer from ds_change_outbox and writing data to ds_change_inbox of testapp2.
BLOCK D: updating program parameter with status and latest pointers

ds_import_changes.py

dj6_feature_2:components:code:ds_change_import.py

This is simply moving data from inbox of testapp2 to testapp2 app tables

Block 0: read ds_program_parameter table for specific id for last pointer of testapp2 inbox and destination testapp2 info.
BLOCK A: Define database connection (testapp2)
BLOCK B: read the last or maximum id number from ds_change_inbox table
BLOCK C: Retrieve data between last point and maximum pointer from ds_change_inbox and perform data operations (insert, update, delete) in testapp2 tables.
BLOCK D: updating program parameter with status and latest pointers