talend | Connecting the Data-Driven Enterprise

Lab Guide

DI Advanced

Version 6.3

Talend Inc.
800 Bridge Parkway, Suite 200
Redwood City, CA 94065
United States
+1 (650) 539 3200

Welcome to Talend Training

Congratulations on choosing a Talend training module. Take a minute to review the following points to help you get the most from your experience.

Technical Difficulty

Instructor-Led

If you are following an instructor-led training (ILT) module, there will be periods for questions at regular intervals. However, if you need an answer in order to proceed with a particular lab, or if you encounter a situation with the software that prevents you from proceeding, don't hesitate to ask the instructor for assistance so it can be resolved quickly.

Self-Paced

If you are following a self-paced, on-demand training (ODT) module, and you need an answer in order to proceed with a particular lab, or you encounter a situation with the software that prevents you from proceeding with the training module, a Talend Support Engineer can provide assistance. Double-click the **Live Expert** icon on your desktop and follow the instructions to be placed in a queue. After a few minutes, a Support Engineer will contact you to determine your issue and help you on your way. Please be considerate of other students and only use this assistance if you are having difficulty with the training experience, not for general questions.

Exploring

Remember that you are interacting with an actual copy of the Talend software, not a simulation. Because of this, you may be tempted to perform tasks beyond the scope of the training module. Be aware that doing so can quickly derail your learning experience, leaving your project in a state that is not readily usable within the tutorial, or consuming your limited lab time before you have a chance to finish. For the best experience, stick to the tutorial steps! If you want to explore, feel free to do so with any time remaining after you've finished the tutorial (but note that you cannot receive assistance from Tech Support during such exploration).

Additional Resources

After completing this module, you may want to refer to the following additional resources to further clarify your understanding and refine and build upon the skills you have acquired:

- Talend product documentation (help.talend.com)
- Talend Forum (talendforge.org/)
- Documentation for the underlying technologies that Talend uses (such as Apache) and third-party applications that complement Talend products (such as MySQL Workbench)

This page intentionally left blank to ensure new chapters start on right (odd number) pages.

CONTENTS

LESSON 1 Connect to a Remote Repository

Introduction 12
Connecting to a Remote Repository 13
- Overview 13
- Objectives 13

Create a Remote Connection 14
- Overview 14
- Start Talend Administration Center Service 14
- Start Studio and Configure the Connection 14

Wrap-Up 18
- Next step 18

LESSON 2 SVN in Studio

Introduction 20
SVN in Studio 21
- Overview 21
- Objectives 21

Copying a Job to a Branch 22
- Overview 22
- Copy to Branch 22
- Switch Branches 24

Comparing Jobs 27
- Overview 27
- Compare a Job Across Branches 27

Resetting a Branch 30
- Overview 30

Wrap-Up 33
- Next step 33

LESSON 3 Remote Job Execution

Introduction 36
Remote Job Execution 37
- Overview 37
- Objectives 37

Creating and Running a Job Remotely 38
Overview 38
Start Server 38
Create and Test a Job Locally 38
Run the Job Remotely 40
Challenge 43
Overview 43
Second Remote Server 43
Next 43
Solution 44
Overview 44
Second Remote Server 44
Wrap-Up 45
Next step 45

LESSON 4 Resource Usage and Basic Debugging

Introduction 48
Resource Usage and Basic Debugging 49
Overview 49
Objectives 49
Using Memory Run to View Real Time Resource Usage 50
Overview 50
Build a Memory Intensive Job 50
Basic Run 53
Memory Run 53
Memory Run Violations 54
Resolution Options 54
Debugging Jobs Using Debug Run 57
Overview 57
Trace Debug 57
Wrap-Up 65
Next step 65

LESSON 5 Activity Monitoring Console (AMC)

Introduction 68
Activity Monitoring Console (AMC) 69
Overview 69
Objectives 69
Configuring Statistics and Logging 70
Overview 70
Configure the Project 70
Import and Configure a Job 71
Using the Activity Monitoring Console (AMC) 76
Overview 76

Accessing the AMC 76
Using the AMC 79
Monitor a Different Job 80
Challenge 84
Overview 84
Introduce an Error 84
Next 84
Solution 85
Overview 85
Introduce an Error 85
Next 85
Wrap-Up 86
Next step 86

LESSON 6 Parallel Execution

Introduction 88
Parallel Execution 89
Overview 89
Objectives 89
Writing Large Files 90
Overview 90
Create the Job 90
Enable Multi-threaded Execution of the JobParallel 91
Use the tParallelize Component 92
Writing to Databases 95
Overview 95
Create the Job 95
Retrieve Schema 97
Configure Parallel Execution 100
Automatic Parallelization 102
Overview 102
Generate data 102
Enrich and display data 103
First Run 105
Enable Parallelization 105
Change Parallelization Settings 107
Disable Parallelization 109
Partitioning 111
Overview 111
Generate Data 111
Partition and collect data 111
Sort Data 113
Finalize Job 114

Wrap-Up 116
Next step 116

LESSON 7 Joblets

Introduction 118
Joblets 119
Overview 119
Objectives 119
Creating a Joblet from an Existing Job 120
Overview 120
Run the Original Job 120
Refactor Mapping to Joblet 122
Creating a Joblet from Scratch 126
Overview 126
Creating a Joblet 126
Update the JoinData Job 127
Triggering Joblets 129
Overview 129
Create a Triggered Joblet 129
Create a Test Job 130
Wrap Up 133
Next step 133

LESSON 8 Unit Test

Introduction 136
Unit Test 137
Overview 137
Objectives 137
Creating a Unit Test 139
Build Standard Job 139
Capture Baseline Output 139
Create the Unit Test 139
Configure the Unit Test 141
Run the Test Case 143
Basic Run of the Unit Test 145
Wrap-Up 147
Next step 147

LESSON 9 Change Data Capture

Introduction 150
Change Data Capture 151
Overview 151
Objectives 151
Examining Databases 152

Overview 152
Retrieve Schema 152
Compare Data 158
Configure the CDC Database 161
Overview 161
Define Subscribers Table 161
Monitoring Changes 169
Overview 169
Build the Database Modification Job 169
Examine the CDC Database 176
Updating a Warehouse 180
Overview 180
Update the Warehouse 180
Run the Job 183
Challenge 186
Overview 186
Multiple Changes 186
Insert and Delete 186
Solutions 187
Overview 187
Multiple Changes 187
Insert then delete 187
Wrap-Up 190
Next step 190
Resetting the Databases 191
Overview 191
Reset Table Contents 191
Continue 192

APPENDIX Additional Information

This page intentionally left blank to ensure new chapters start on right (odd number) pages.

LESSON 1

Connect to a Remote Repository

This chapter discusses the following.

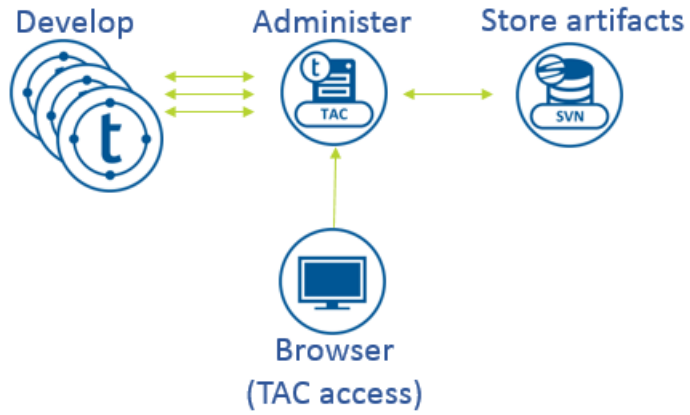
Introduction 12
Connecting to a Remote Repository 13
Create a Remote Connection 14
Wrap-Up 18

Introduction

Connecting to a Remote Repository

Overview

This lesson guides you through the process of starting Talend Studio with a connection to a remote, shared repository. Multiple developers may work on a single project, each running an individual instance of Talend Studio. By storing repository information, including Job designs and metadata in a central location, you avoid duplication and errors that might arise from out-of-sync assets. A central repository is critical for collaboration efforts.



The Subversion repository that you will connect to in this controlled training environment is actually stored locally within the machine on which the training environment is hosted, but the procedures are identical for a repository located on a different system.

Objectives

After completing this lesson, you will be able to:

- Start the required Windows services
- Configure Talend projects for local or remote connections
- Create and configure a connection to a remote Talend repository
- Start Talend Studio with a remote connection

The first step is to create a remote connection and start the software.

Create a Remote Connection

Overview

Talend Data Integration allows you to centralize Job designs and metadata in a common Repository to be accessed by multiple developers. In this exercise, you will start Talend Studio and connect to a remote Repository prepared in advance as part of the training environment.

NOTE: Note

Steps to create and configure a Repository are covered in the *Talend Data Integration Server Administration* training course.

Start Talend Administration Center Service

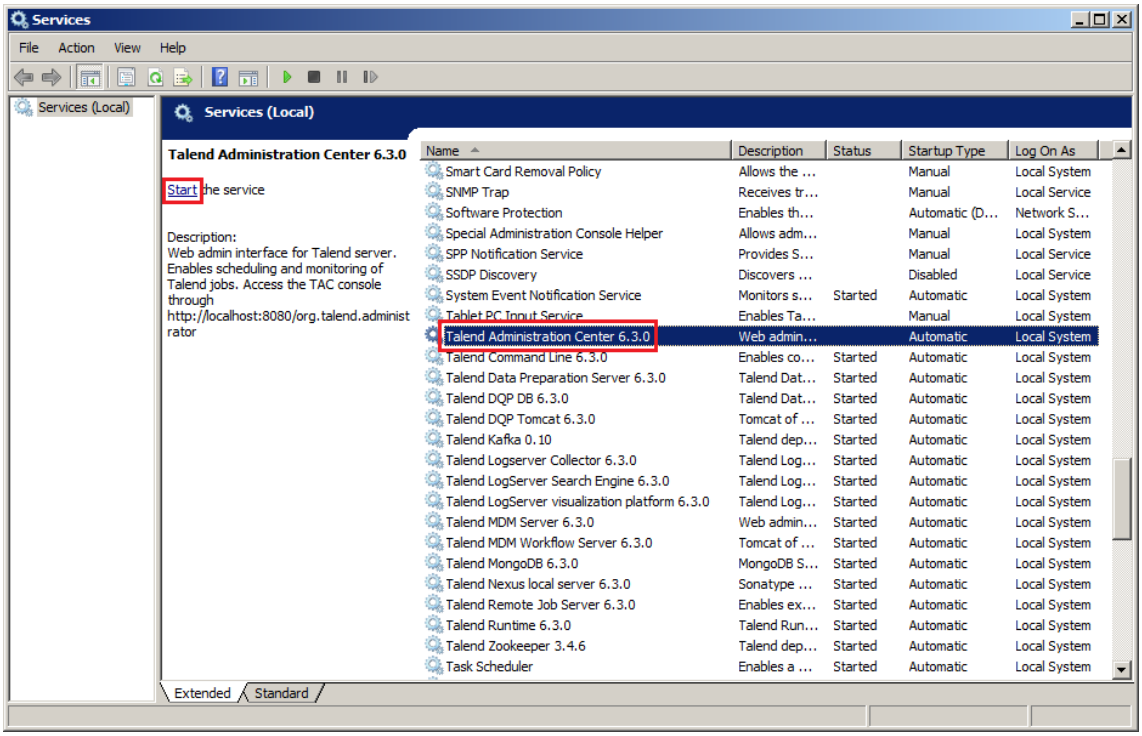
To create a remote connection, you need to start the Talend Administration Center (TAC) first.

1. ENABLE THE TALEND ADMINISTRATION CENTER

   Click the **Services** icon in the Windows task bar:

   In the **Services** window, scroll down and select **Talend Administration Center**. If the **Status** is not *Started*, then click the **Start** link.



Start Studio and Configure the Connection

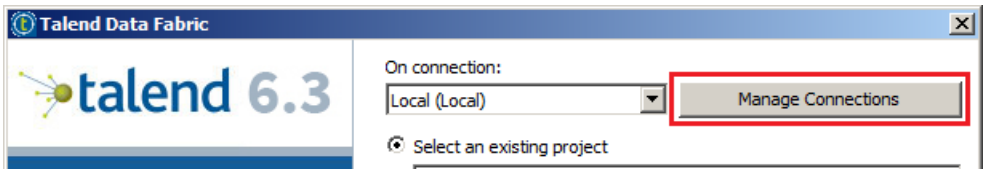
1. START TALEND STUDIO

   Start Talend Studio by double-clicking the following icon on your desktop:

After a few moments, the **Talend Data Fabric** window opens.

2. PREPARE A CONNECTION

Select the **Remote** connection, if available. If the **Remote** connection is not available, click **Manage Connections**.

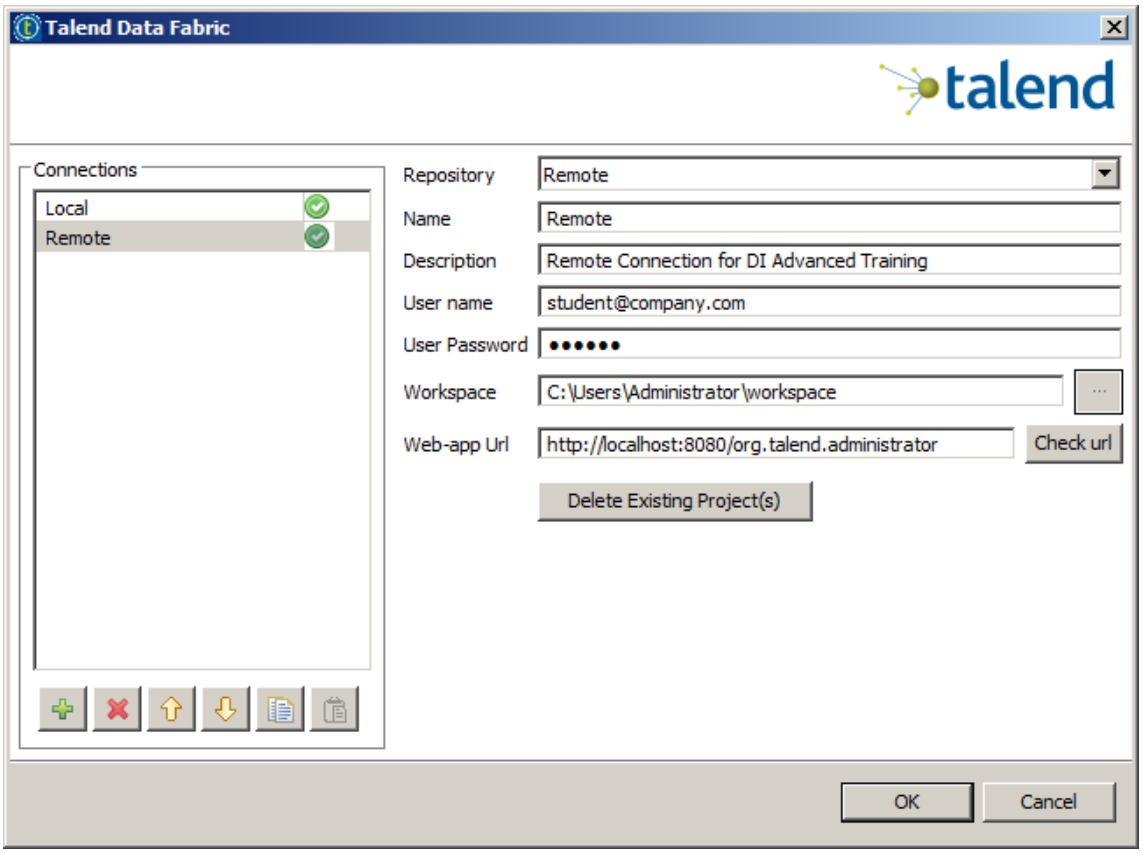


Here you create new and modify existing connections.

Create a new connection by clicking the **Add** button (). Fill out the fields as follows:

- Select **Remote** for **Repository**
- Enter *Remote* for **Name**
- Enter some descriptive text for the **Description**
- Enter *student@company.com* for **User name**
- Enter *Talend* for **User Password**
- Enter *http://localhost:8080/org.talend.administrator* for **Web-app Url**
- Leave **Workspace** set to its default value

Click **Check url** to verify your settings.

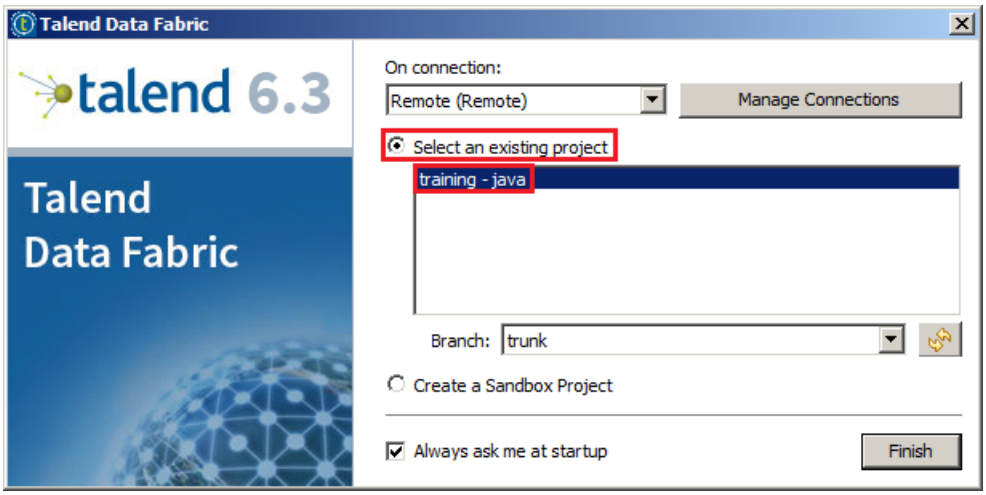


If the verification fails, double-check the values you entered. Otherwise click **OK** to continue.

3. SELECT A PROJECT

Back in the **Talend Data Fabric** window, choose the **Select an existing project** option. This option provides a list of projects available through the selected connection. These projects reside in a remote repository that is described by the TAC configuration and is already set up as part of the training environment.

From this list, select the **training** project and click **Finish**.



NOTE:

- If prompted to connect to the TalendForge community, select **Skip this Step**.
- If presented with the **Welcome** page, click **Start now!**

You have now completed this lesson. To recap your experience read the Wrap-up.

Wrap-Up

In this lesson, you created a new connection that allowed you to start Talend Studio and access an existing central repository as well as open up a project within that repository.

Using a centralized repository that allows for collaboration with other developers is one of the key differentiators when using a subscription based deployment as opposed to Talend Open Studio (TOS) for Data Integration.

Next step

Congratulations! You have successfully completed this lesson. To save your progress, click **Check your status with this unit** below. To go to the next lesson, on the next screen, click **Completed. Let's continue >**.

LESSON 2

SVN in Studio

This chapter discusses the following.

Introduction 20
SVN in Studio 21
Copying a Job to a Branch 22
Comparing Jobs 27
Resetting a Branch 30
Wrap-Up 33

Introduction

SVN in Studio

Overview

As your Talend projects move into production, you may find that you need to maintain the existing code while adding features for a new version of the project. Talend Data Integration supports the Subversion (SVN) source control system to help you maintain multiple versions of your Talend projects. In this lesson, you will examine another branch, copy a Job from one branch to another, and compare Jobs between branches.

Note that the branch has already been created for you. In a production environment, it is likely that branching would be an administrator's function.

Objectives

After completing this tutorial, you will be able to:

- Switch between branches in Talend Studio
- Copy a Job from one branch to another
- Compare the differences between two versions of the same Job

The first step is to copy a Job to a branch.

Copying a Job to a Branch

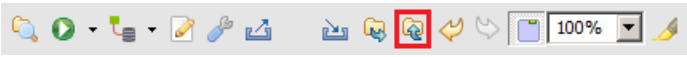
Overview

When an administrator creates a new branch for a project, all of the existing Jobs in the project are copied into the new branch. This is a typical scenario for when a project team is ready to start modifying an existing version for improvements. However, it is also possible to create a Job in a sandbox branch and then copy it to the trunk when it is ready, or copy it to another branch altogether. Here, you will explore working with a branch created for you already.

Copy to Branch

1. IMPORT A JOB

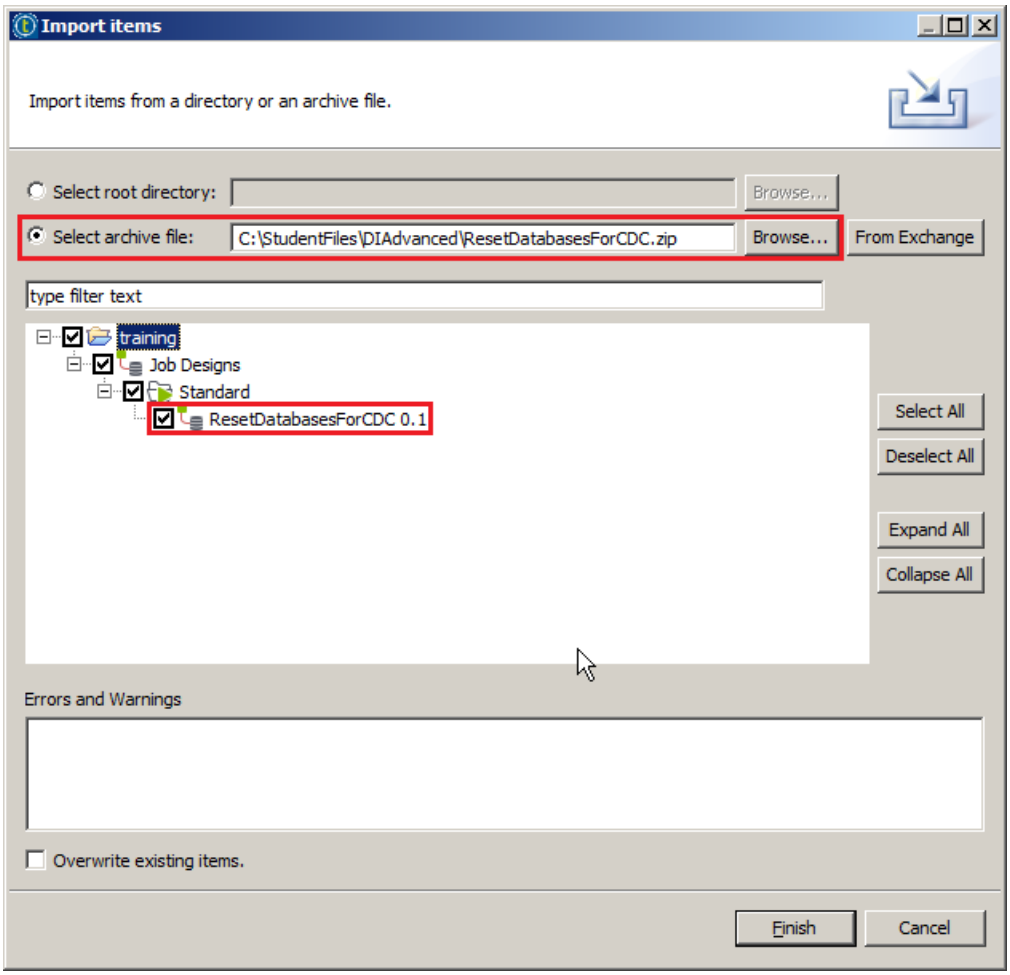
   Begin importing a Job by clicking the **Import Items** icon ( ) in the main tool bar.

   NOTE:
   You can also import a Job if you right-click on **Repository > Job Designs > Standard** and select **Import Items**.

   Choose **Select archive file** and specify a path of *C:\StudentFiles\DIAdvanced\ResetDatabasesForCDC.zip*.

   Select **training > Job Designs > Standard > ResetDatabasesForCDC** and click **Finish**.



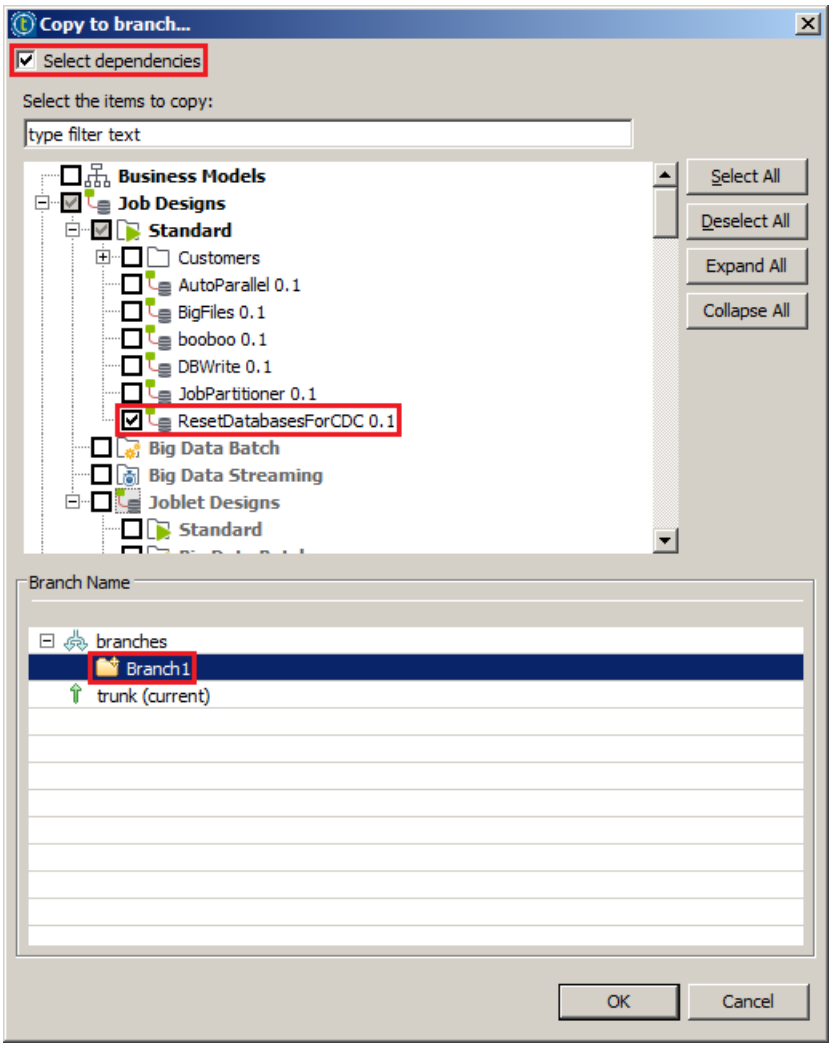
2. COPY THE JOB TO A BRANCH

   Right-click **Repository > Job Designs > Standard > ResetDatabasesForCDC** and select **Copy To Branch**.

   In the **Copy to branch** window that appears, pay attention to a few things:

   - The material selected is what you intend to copy (in this case, the **ResetDatabasesForCDC** job).
   - Select the **Select dependencies** check box to ensure that any additional resources your job requires are also copied to the branch. This is considered best practice.
   - Select the branch to which to copy, in this case **branches > Branch1**.

   Click **OK**.



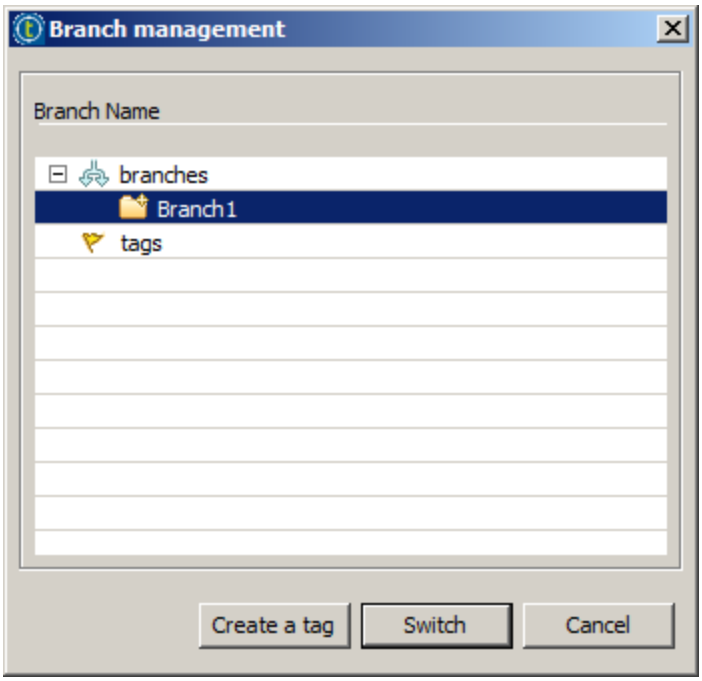
The operation takes a few moments. Once done, an identical copy of the Job resides in *Branch1*.

Switch Branches

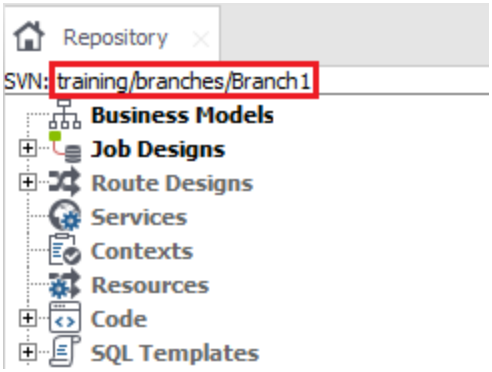
1. SWITCH REPOSITORY TO A DIFFERENT BRANCH

   Click the **Branch Management** icon ( ) at the top of the **Repository** view.

   In the **Branch Management** window that opens, select **branches > Branch1** and click **Switch**.



After a few moments, you are placed in the *Branch1* branch, as evidenced by the label above the **Repository**.



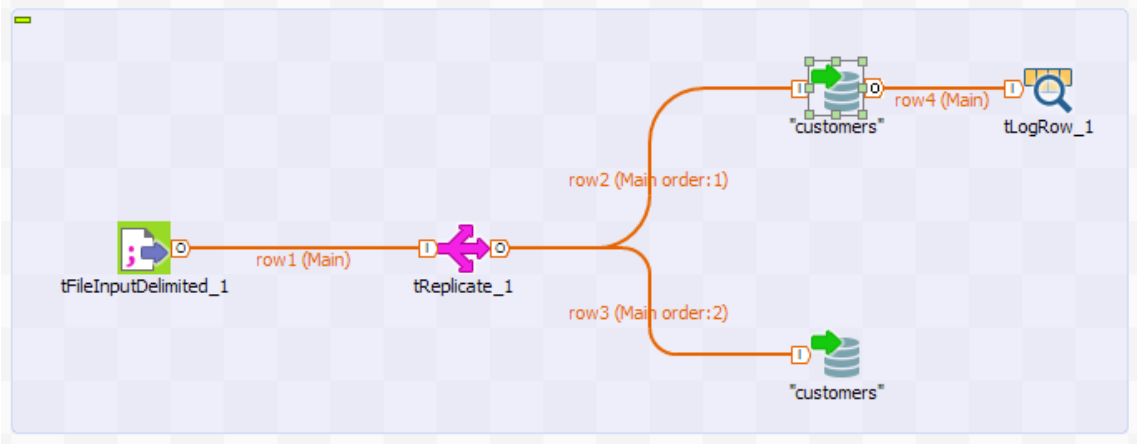
WARNING:
If for some reason the branch remains as *trunk* instead of switching to *Branch1*, follow the steps in Resetting the Branch before continuing.

2. OPEN A JOB

   Double-click the **Repository > Job Designs > Standard > ResetDatabasesForCDC** Job to open it.

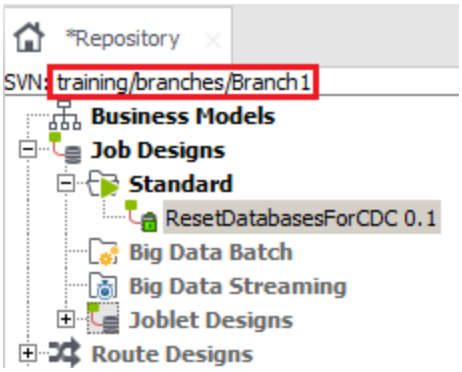
3. MODIFY THE JOB

   Add a **tLogRow** component and then connect it to either of the **customers** component using a **Main** row (for the purpose of this exercise, it does not matter to which one you connect). The Job should resemble the following figure.



Save the Job.

You now have the same Job in both branches, but with slight differences. Remember that the branch you are working on is reflected in the **Repository** view.



Now you can use Talend Studio to compare the two Jobs and display the differences.

Comparing Jobs

Overview

With a Job in both the trunk and a branch, you can now compare the Jobs to determine their differences.

Compare a Job Across Branches

1. SWITCH BRANCHES

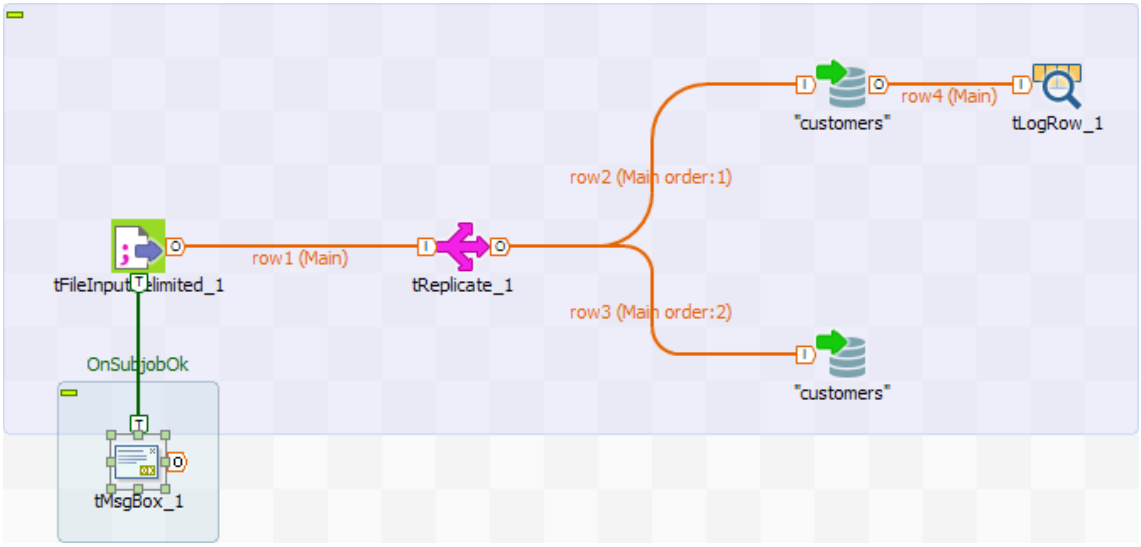
   In the **Repository**, switch to *Branch1* if you are not already there.

   TIP:

   Recall that to switch branches, you must click the **Branch Management** icon ( ) at the top of the **Repository**.

2. MODIFY A JOB

   Open the **ResetDatabasesForCDC** Job and make additional changes. For example, add a **tMsgBox** component below the **tFileInputDelimited** component and connect it with an **OnSubjobOK** trigger.



   When done, save the Job.

3. COMPARE THE JOB

   Right-click **Repository > Job Designs > Standard > ResetDatabasesForCDC** and select **Compare Job**.

   The **Compare Results** view will open, by default in rightmost area of the Perspective where the **Palette** view is located.

   TIP:

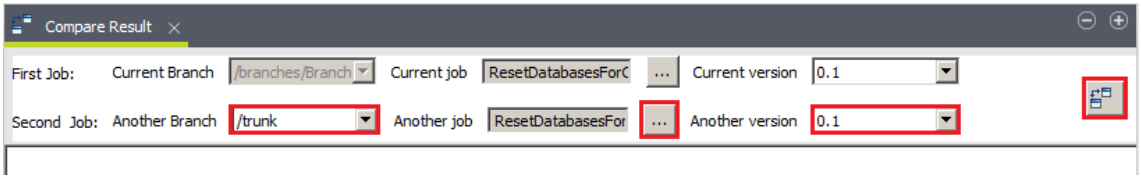
   Here are two ways to quickly increase the size of the **Compare Results** view to make it more useable. These techniques can be applied to any view in Talend Studio:

   - Drag and drop the **Compare Results** tab to another area of the Perspective that is larger, such as the area where the **Designer** is located.
   - Click the **Maximize** icon ( ) at the top-right corner of the view to increase the size to fill the available space. Click it again to restore the view to its original size.

In the **Compare Results** view, specify the Job against which you want to compare:

- Specify */trunk* for **Another Branch**
- Specify *ResetDatabasesForCDC* for **Another job**, using the button marked with an ellipsis
- Specify *0.1* for **Another version**

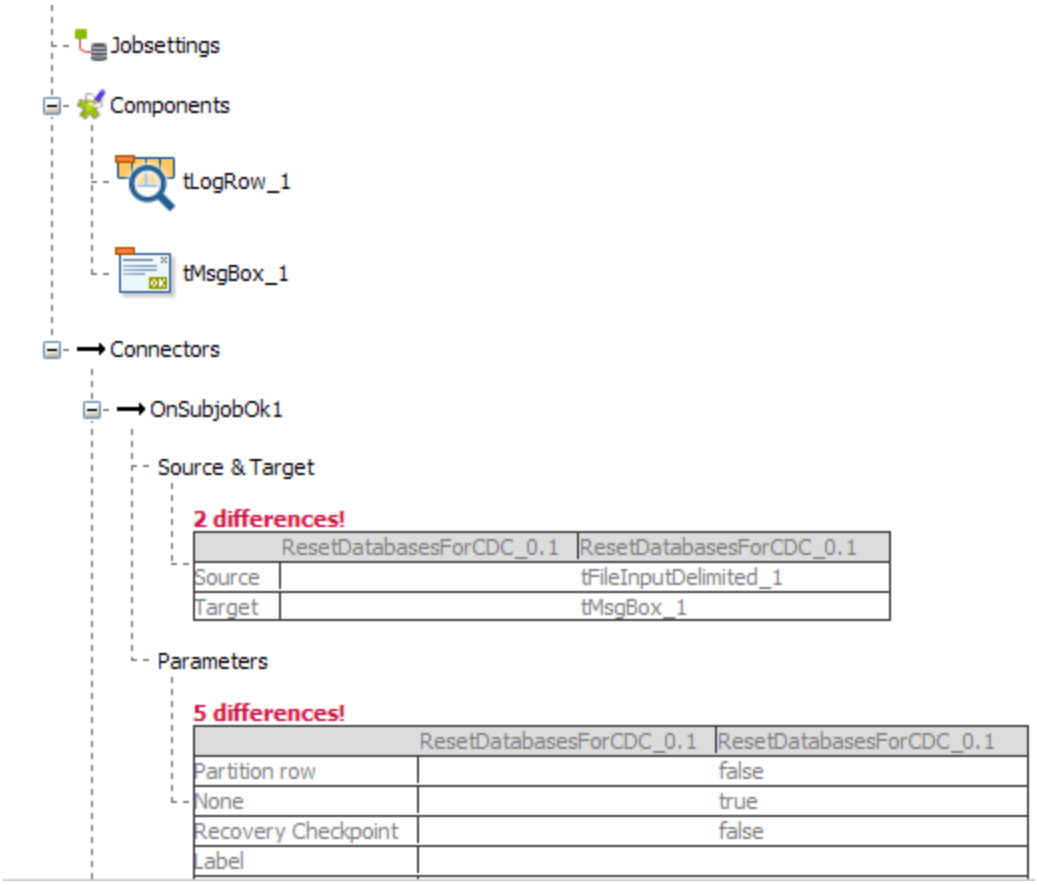
When done, click the **Compare** button ( ) on the far right.



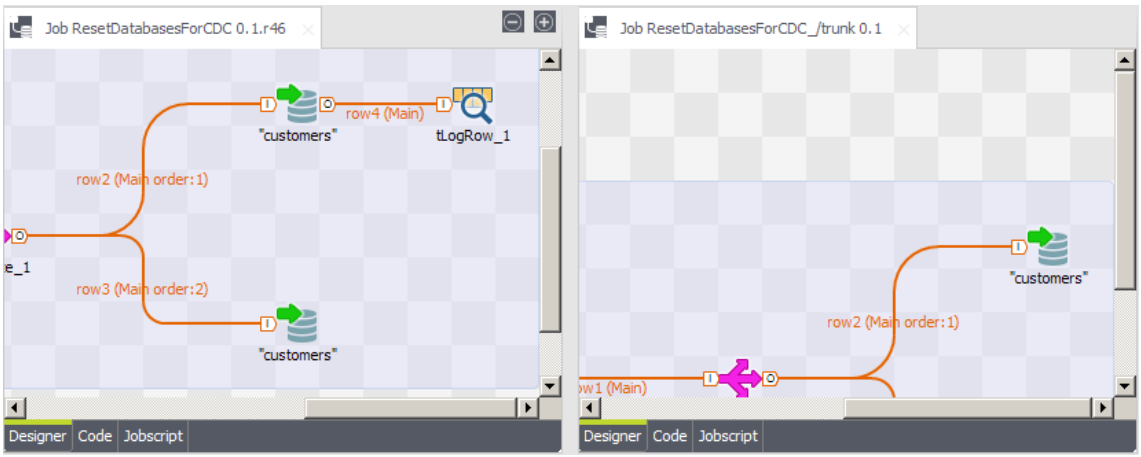
4. EXAMINE THE DIFFERENCES

The differences are displayed in the lower portion of the **Compare Results** view.

NOTE:
Your results will very likely look different than the example figure shown below.



Additionally, both Jobs open side-by-side in the **Designer**.



Examine the information shown in both views. One obvious difference shown is the **tLogRow** component added to one Job but not the other.

5. CLEAN UP

   Close the two Job views in the **Designer**, as well as the **Compare Results** view.

   Switch back to *trunk* using the **Branch Management** icon ( ) at the top of the **Repository**.

You now have an understanding of how to compare Jobs between branches. The same approach can be applied to compare Jobs in *trunk* to other branches, different versions of a Job in the same branch, or even variations of a Job in a branch.

You have now completed this lesson and it's time to Wrap-Up.

WARNING:
Skip the *Resetting the Branch* section since it is only needed if you had trouble earlier in the lesson when copying a branch.

Resetting a Branch

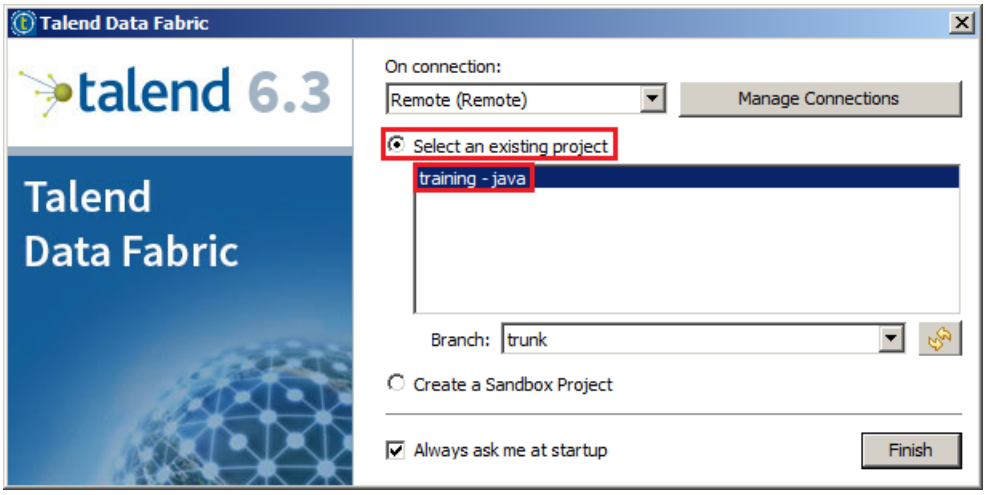
Overview

If you have difficulty switching branches, the most likely cause is that the Job copy did not complete correctly and a local copy of the project was created. Talend Studio does not permit both a local and remote copy of the same project. To recover, follow these steps.

WARNING:
If you don't need to recover from issues when copying the Job, please proceed to the Wrap-Up for this lesson.

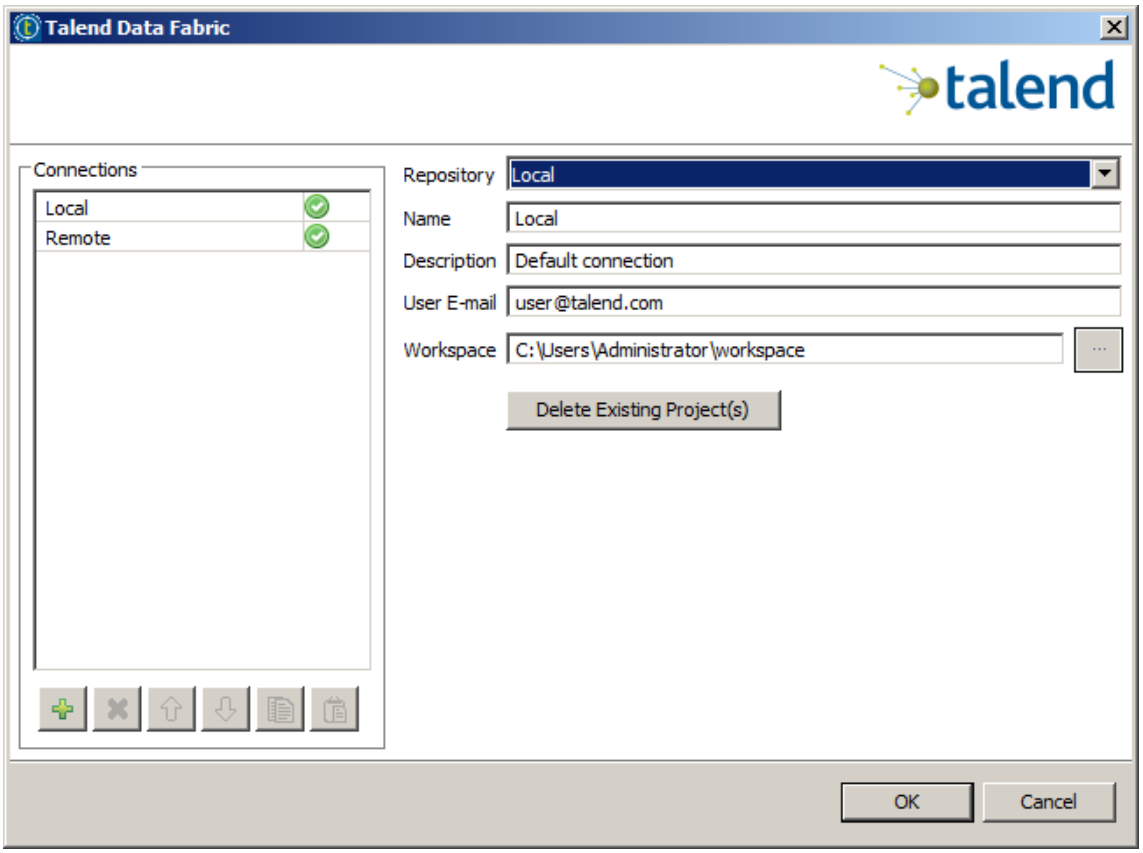
1. RESTART STUDIO

   From the main menubar, select **File > Switch Project or Workspace**. Talend Studio restarts. When it does, click **Manage Connections**.

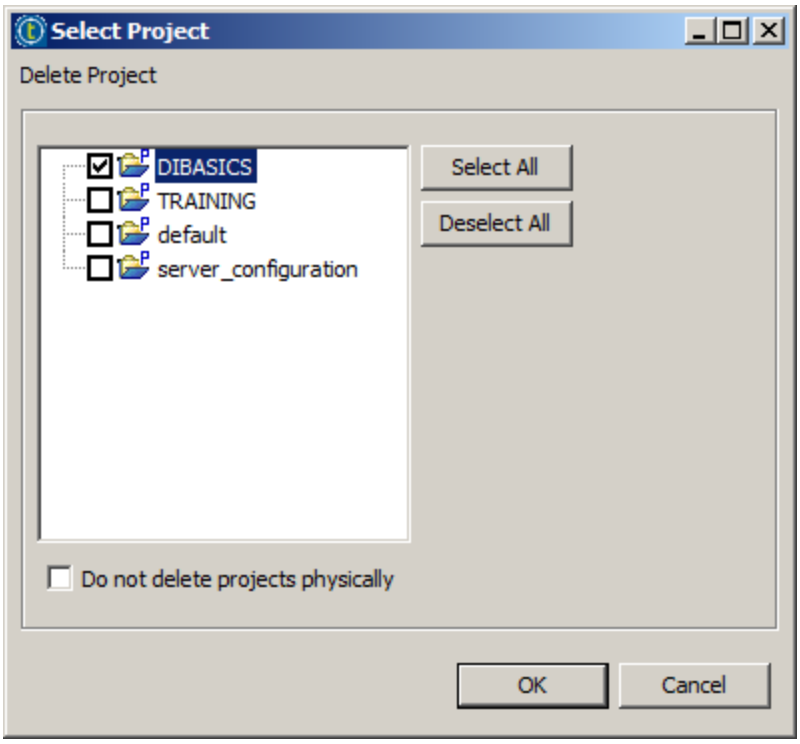


2. DELETE LOCAL PROJECT

   With the *Local* Repository selected (the default), click **Delete Existing Project(s)**.



Select **DIBASICS** in the list and click **OK** (this project is not needed for this course).



3. SELECT CONNECTION AND PROJECT

   Back in the **Talend Data Fabric** window, ensure that the *Remote* connection and *training* project is selected, then click **Finish**.

4. RESUME LESSON

   Once the Talend Studio has completed its start-up, please return to Copying a Job to a Branch and continue from where you left off.

Wrap-Up

In this lesson, you used Subversion (SVN), a third party source code revision control system that is integrated with the Talend Studio. You copied a Job from the trunk to an existing branch in SVN, switched Talend Studio to use that branch, made modifications to the Job and then compared Jobs between the trunk and a branch.

Next step

Congratulations! You have successfully completed this lesson. To save your progress, click **Check your status with this unit** below. To go to the next lesson, on the next screen, click **Completed. Let's continue >**.

This page intentionally left blank to ensure new chapters start on right (odd number) pages.

LESSON 3

Remote Job Execution

This chapter discusses the following.

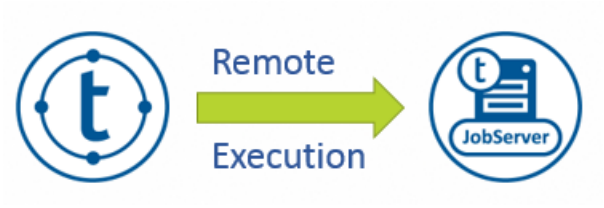
Introduction 36
Remote Job Execution 37
Creating and Running a Job Remotely 38
Challenge 43
Solution 44
Wrap-Up 45

Introduction

Remote Job Execution

Overview

Frequently the computer you use to create a Talend Job is not the computer that will run the Job in production. With Talend Data Integration, your administrator can install Talend Job Server software on different computers without installing any of the other applications:



In this lesson, you will create a simple Job and then run the Job on a remote Job Server from your local Talend Studio. Keep in mind that in the controlled training environment, the "remote" Job Server is actually running locally, but the process is exactly the same.

Objectives

After completing this tutorial, you will be able to:

- Configure Talend Studio to identify remote Job Servers
- Run a Job from Talend Studio on a remote Job Server

The first step is to create a sample Job.

Creating and Running a Job Remotely

Overview

In this lesson, you will create a Job and then run it on a Talend Job Server not from within the Talend Studio. The Job itself is trivial, because the focus of the exercise is the configuration and remote execution not the Job itself.

Start Server

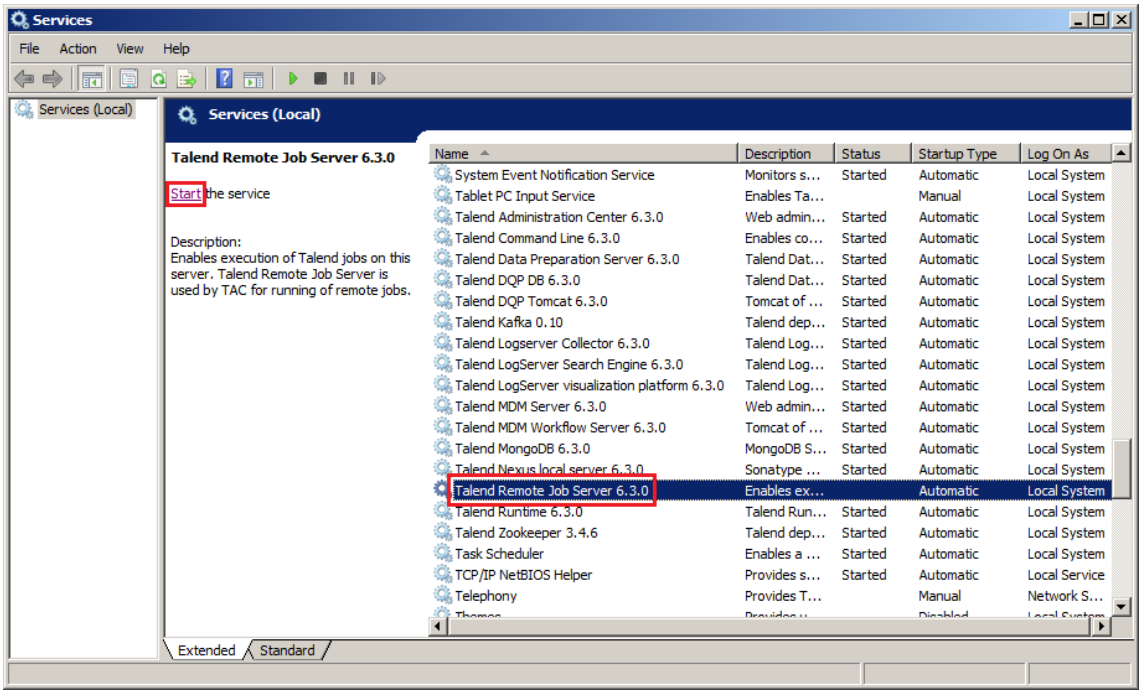
To run a Job remotely, you must start the **Talend Remote Job Server** first.

1. ENABLE THE TALEND REMOTE JOB SERVER

   Click the **Services** icon in the Windows task bar.

   In the **Services** window, scroll down and select **Talend Remote Job Server**. If the **Status** is not *Started*, then click the **Start** link.



Create and Test a Job Locally

With the **Talend Remote Job Server** running, you will now create a Job to run remotely.

WARNING:
Make sure the Repository is set to */trunk* before proceeding.

1. CREATE A JOB

   Create a new standard Job named *FileTouch*.

   Add a **tFileTouch** component.

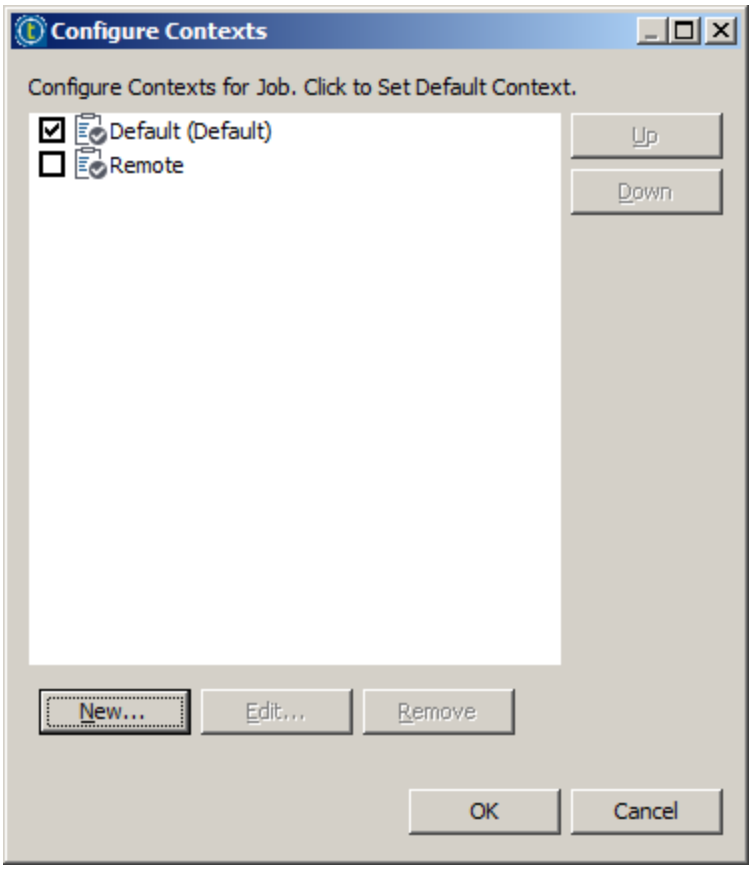
NOTE:
This component creates an empty file or updates the timestamp of an existing file. It is analogous to the Unix **touch** utility. If the file does not exist, it is created. If it does exist, only the modification time is updated; the actual content of the file is not modified.

2. CREATE A NEW CONTEXT

   In the **Contexts** view, click the **Configure Contexts** button at the far right (+) to add a second context.

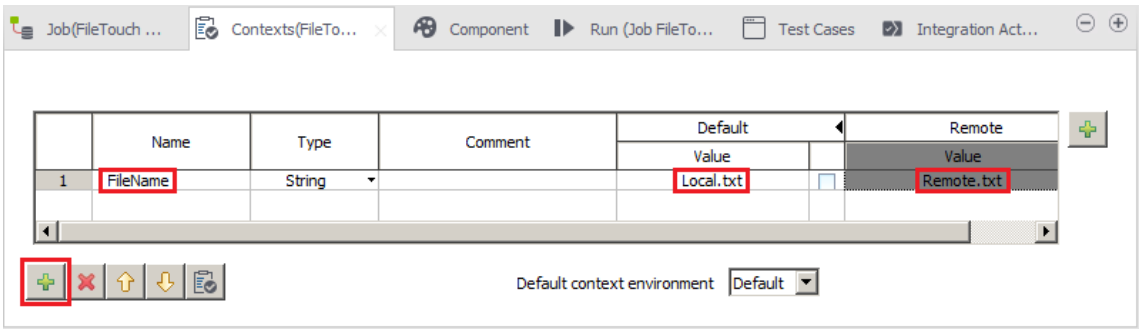
   In the **Configure Contexts** window that appears, click the **New** button to add a context called *Remote*.

   Click **OK** when done.



3. CREATE A NEW CONTEXT VARIABLE

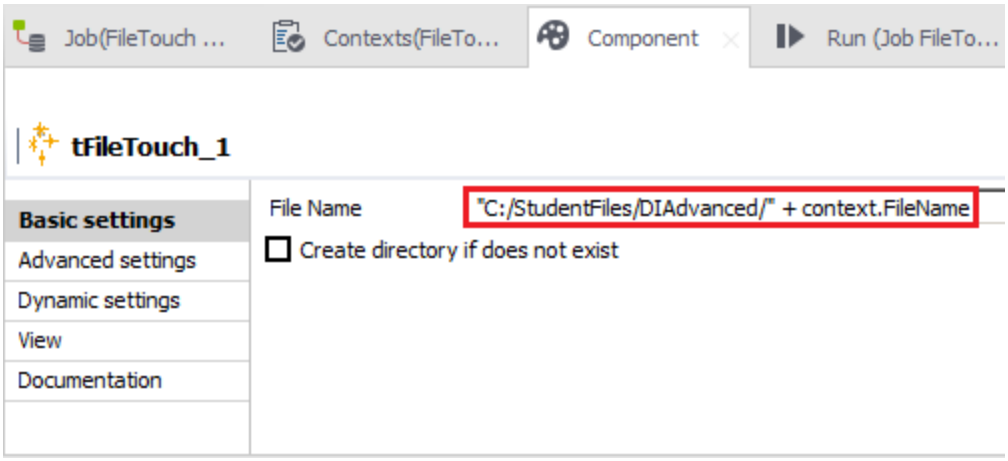
   Still in the **Contexts** view, click the button marked with a plus sign (+) below the table to create a new context variable named *FileName*. Set the **Default Value** to *Local.txt* and the Remote Value to *Remote.txt*.



NOTE:
Although context variables are not mandatory for running Jobs remotely, they allow you to adjust settings at run-time without having to modify any components in the Job.

4. CONFIGURE THE OUTPUT FILE NAME

   In the **Component** view of the **tFileTouch** component, set the **File Name** parameter to `"C:/StudentFiles/DIAdvanced/" + context.FileName`.



TIP:
- Don't forget the trailing slash and enclosing quotation marks in the first part of the expression
- When typing the names of context variables, remember that you can use **Ctrl + Space** as a shortcut

5. RUN THE JOB LOCALLY

   Run the Job using the **Default** context.

   Locate the file in the *C:\StudentFiles\DIAdvanced* folder. The empty file *Local.txt* should be present with a current time stamp.

Run the Job Remotely

Now you will configure the Studio to be aware of a remote host that runs the Talend Job Server. For simplicity in the training environment, the remote host is actually the same host you run the Studio on, however the procedure for configuring the Job Server is the same when target a host that is truly remote.

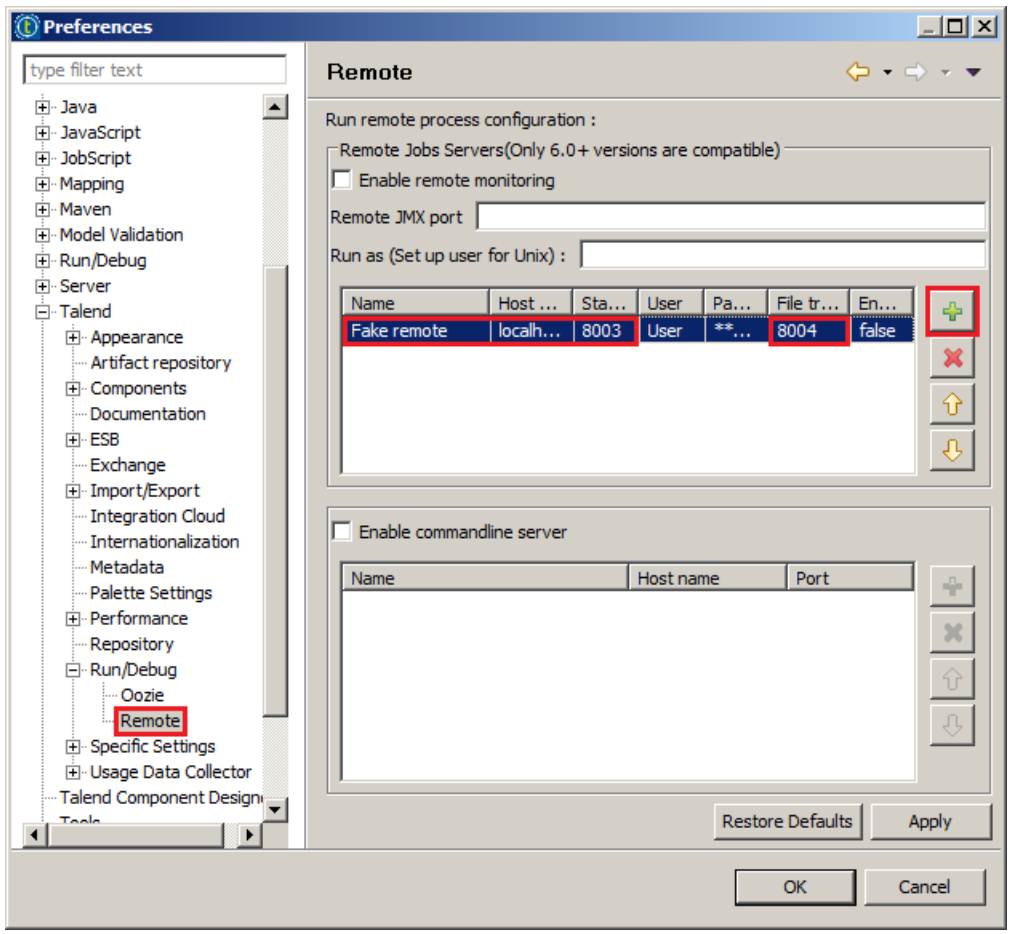
1. ADD A REMOTE JOB SERVER CONFIGURATION

   From the main menubar, select **Window > Preferences**. Then, in the **Preferences** window that opens, select **Talend > Run/Debug > Remote**.

   Click the **Add** button (), and change the following values:

   - Set **Name** to *Fake remote*
   - Set **Host name** to *localhost*
   - Set **Standard port** to *8003*
   - Set **File transfer port** to *8004*

   Click **Apply**, then click **OK**.



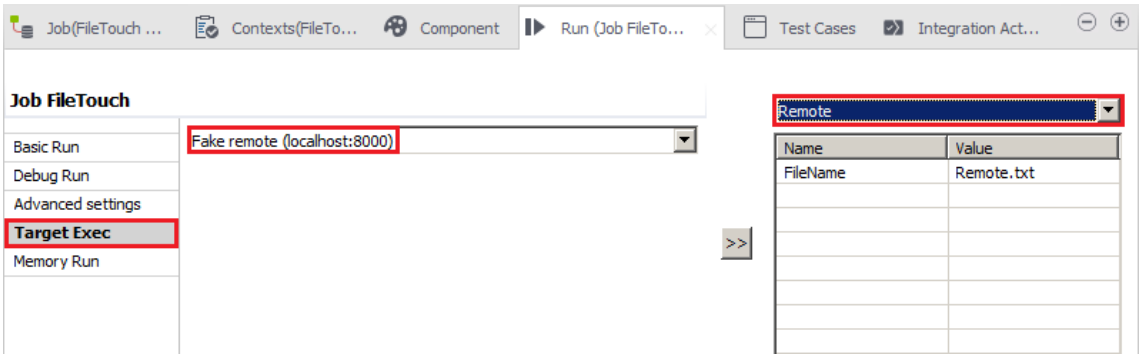
NOTE:
The settings for **Standard port** and **File transfer port** are defined in the *TalendJobServer.properties* file, whose location depends on the product installation. In the training environment, the directory containing this file is *C:\Talend\6.3\jobserver\agent\conf*.

2. RUN THE JOB REMOTELY

   In the **Run** view, click the **Target Exec** tab.

   Choose *Fake Remote* from the list of targets.

In the context list on the right, choose *Remote*.



Then, click the **Basic Run** tab and run the Job. Notice the console messages:

```
Checking ports...

Sending job 'FileTouch' to server (localhost:8001)...

File transfer completed.

Deploying job 'FileTouch' on server (127.0.0.1:8000)...

Running job 'FileTouch'...
Starting job FileTouch at 20:10 12/01/2017.


[statistics] connecting to socket on port 3734
[statistics] connected
[statistics] disconnected
Job FileTouch ended at 20:10 12/01/2017. [exit code=0]
```

In particular, notice the messages about sending the Job and connecting to the remote server.

Locate the file in the *C:\StudentFiles\DIAdvanced* folder. The empty file *Remote.txt* should be present with a current time stamp. The name is different in this case because of the change in context. When you run a Job remotely, you frequently need to use context variables to configure your Job to work properly on the remote computer.

3. CLEAN UP

   In preparation for the next exercise, change the **Target Exec** back to *Localhost*.

You have now completed this lesson and can move on to the Challenge.

Challenge

Overview

Complete this challenge to further explore the use of remote Job execution. See Solution for a possible solution to this exercise.

Second Remote Server

Modify the Job to add a second context variable that specifies the directory for the touched file. Make the directory value different for the default and remote contexts. Add a second remote Job Server and then run the Job on that server, specifying the remote context.

Next

You have now completed this lesson. It's time to Wrap-up.

Solution

Overview

This is a possible solution to the Challenge. Note that your solution may differ and still be valid.

Second Remote Server

1. In the **Context** view, add a second context variable named *Directory* to hold a value representing the directory containing the created file.

   Set the **Default Value** to *C:/StudentFiles/DIAdvanced/Local* and the **Remote Value** to *C:/StudentFiles/DIAdvanced/Remote*.

2. In the **Component** view for the **tFileTouch** component, configure the **File Name** to use the *Directory* context variable with the expression: `context.Directory + "/" + context.FileName`.

   Enable the **Create directory if does not exist** option.

3. In the **Preferences** window, add a second remote Job server. Set the **Name** to *Fake 2* and set the **Host name** to *localhost*. Keep all other fields as they are.

4. In the **Target Exec** tab of the **Run** view, choose the *Fake 2* target and select the *Remote* context. Run the Job and verify that the file *C:\StudentFiles\DIAdvanced\Remote\Remote.txt* is created.

You have now completed this lesson. To recap your experience read the Wrap-up.

Wrap-Up

In this lesson, you explored how to identify, configure and use remote Job Servers to execute Jobs from within Talend Studio on remote hosts. You used a basic component to touch files on both the local and remote Job Servers based on the context variable selected when the Job was run.

Next step

Congratulations! You have successfully completed this lesson. To save your progress, click **Check your status with this unit** below. To go to the next lesson, on the next screen, click **Completed. Let's continue >**.

This page intentionally left blank to ensure new chapters start on right (odd number) pages.

LESSON 4

Resource Usage and Basic Debugging

This chapter discusses the following.

Introduction 48
Resource Usage and Basic Debugging 49
Using Memory Run to View Real Time Resource Usage 50
Debugging Jobs Using Debug Run 57
Wrap-Up 65

Introduction

Resource Usage and Basic Debugging

Overview

Up to this point, you have used the most common functionality in the **Run** view:

- **Basic Run** tab: run a Job, select context, and see output on the console
- **Target Exec** tab: control whether the Job runs locally or remotely, select context
- **Advanced settings** tab: specify parameters, including the level of logging used and additional JVM options

In this lesson you will use additional functionality:

- **Memory Run** tab: observe memory and CPU usage of your JVM
- **Debug Run** tab: Job debugging capabilities for either java and non-java developers

These tools can be used to provide different levels of debugging functionality for your Jobs. Throughout this lesson you will run a basic Job that generates rows of client names with a sequential ID, it then sorts based on the first and last name.

Objectives

After completing this lesson, you will be able to:

- Run a Job and observe real-time CPU and memory heap usage
- Use debugging tools that do not require deep Java development skills

The first step is to perform a Memory Run on a memory intensive Job.

Using Memory Run to View Real Time Resource Usage

Overview

The **Memory Run** tab in the **Run** view allows you to see Job execution times, as well as monitor JVM memory and CPU usage while your Job is running. The display is graphical and simple to read, and displays current resource usage in real time. **Memory Run** can be a helpful indicator if your Job has a memory leak or is hogging the CPU, leading to reduced system performance.

In this lesson, you will use a Job that consumes a configurable amount of system resources. This will enable you to observe results under varying conditions.

Build a Memory Intensive Job

In order to view host CPU and JVM memory consumption in real-time, you need to build a Job that uses a lot of memory and is easily configurable to use more, or less, memory.

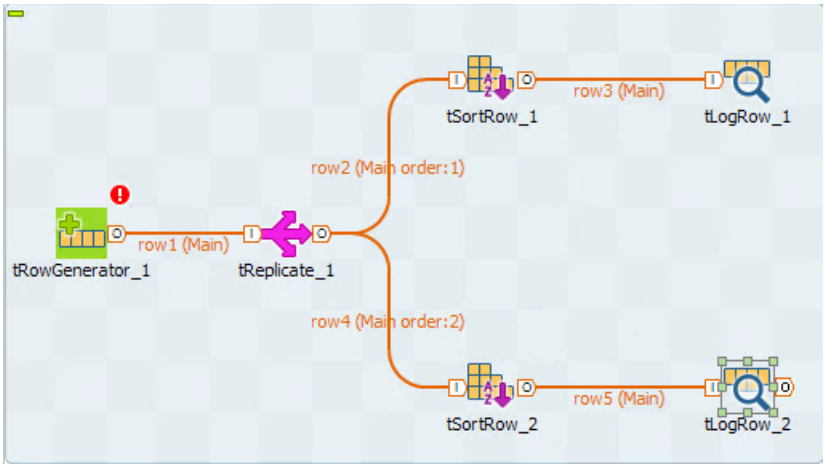
1. CREATE A JOB

   Create a new standard Job named *MemoryRun*.

2. POPULATE THE JOB

   Populate the Job as follows:

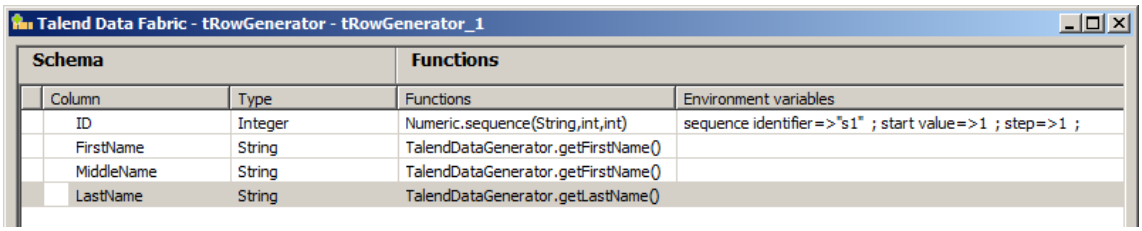
   - Place a **tRowGenerator** component in the left side of the **Designer**
   - Place a **tReplicate** component to the right of the **tRowGenerator**
   - Place two **tSortRow** components, on above and to the right of, and the other below and to the right of the **tReplicate**
   - Place two **tLogRow** components, each one to the right of a **tSortRow** component
   - Working from left to right, connect all the components together with a **Main** row. Note that the **tReplicate** component will have two output connectors, with one running to each **tSortRow**



3. CONFIGURE THE tRowGenerator COMPONENT

   Double-click the **tRowGenerator** component. Configure it with the following four columns:

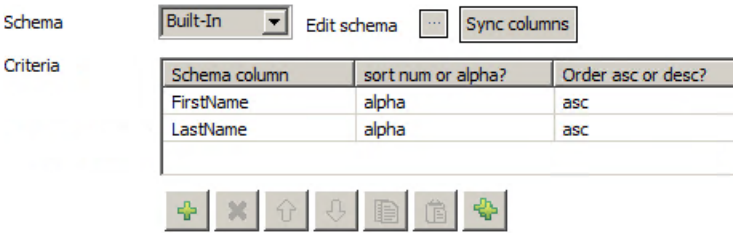
   - *ID* of type *Integer*. Use the `Numeric.sequence(String,int,int)` function so that the values for this column will take on unique values beginning at 1.
   - *FirstName* and *MiddleName*, both of type *String*. For each, use the `TalendDataGenerator.getFirstName()` function to generate values for these columns.
   - *LastName* of type *String*. Use the `TalendDataGenerator.getLastName()` function to generate values for this column.



When done, click **OK**. Click **Yes** when prompted to propagate the changes.

4. CONFIGURE THE tSortRow COMPONENTS

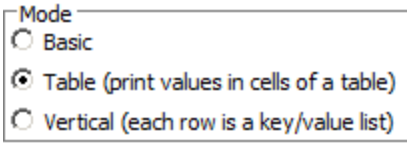
   In the **Component** view for the first (upper) **tSortRow**, configure the component to sort alphabetically in ascending order on the *FirstName*, then the *LastName*.



   Similarly configure the second (lower) **tSortRow**, except sort first on *LastName*, then on *FirstName*.

5. CONFIGURE THE tLogRow COMPONENTS

   For both **tLogRow** components, in the **Component** view, set the **Mode** to *Table*; this is more readable than the default setting.

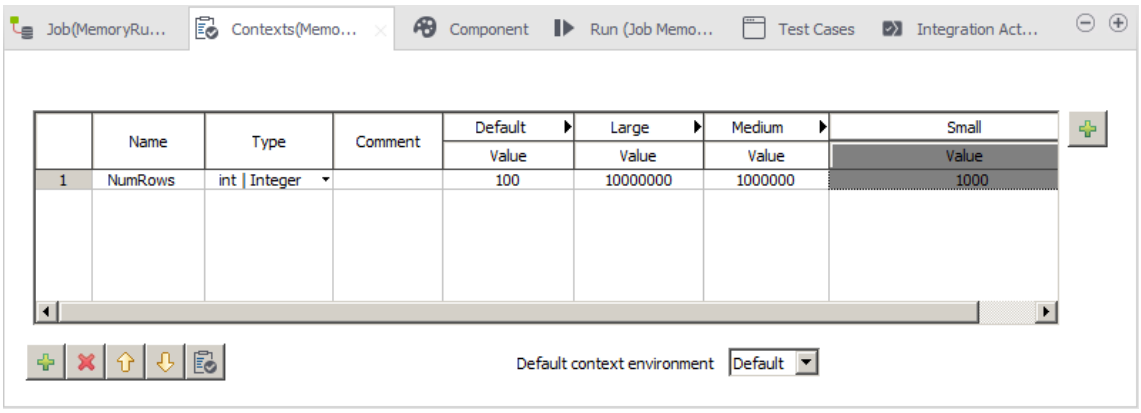


6. ADD A CONTEXT VARIABLE TO CONFIGURE RESOURCE USAGE

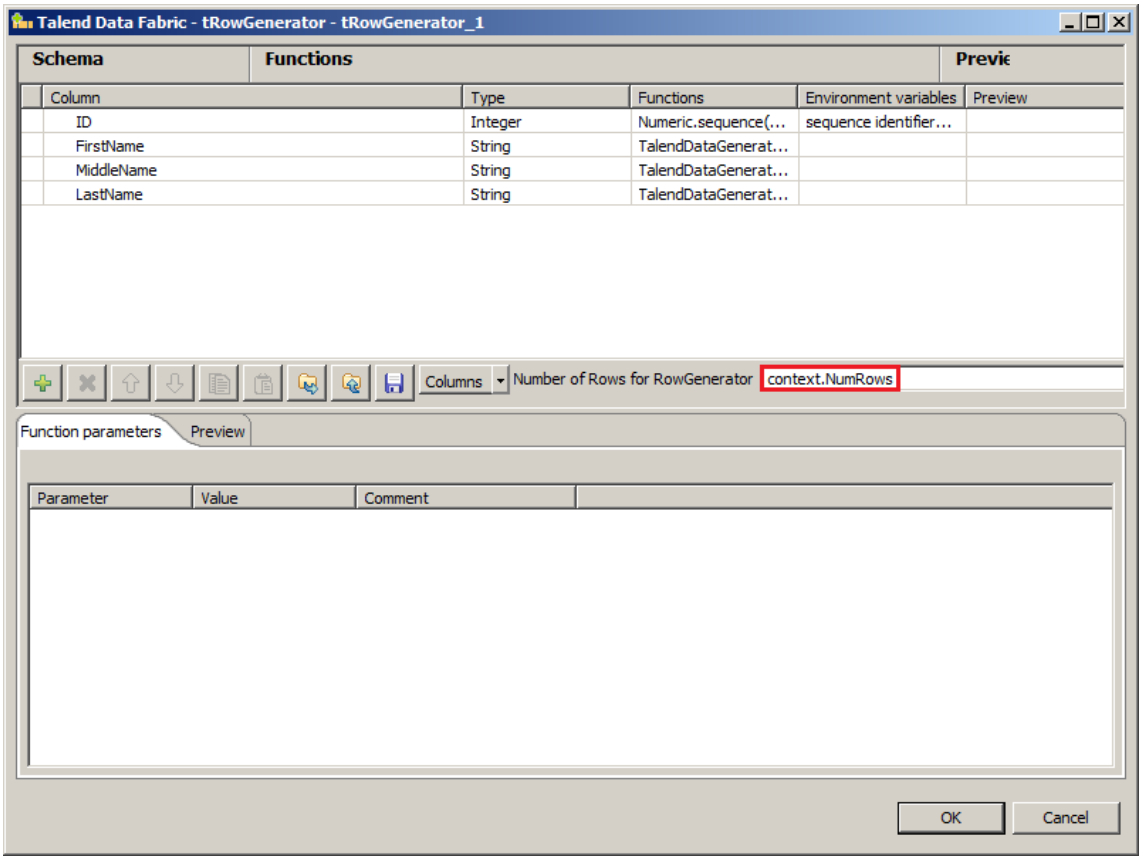
   In the **Contexts** view, add three new contexts named *Small*, *Medium*, and *Large*.

   Add a new context variable called *NumRows* of type *Integer*. Set the values as follows:

   - As a default value, use a value of *100*
   - For the *Large* context, use a value of *10,000,000*
   - For the *Medium* context, use a value of *1,000,000*
   - For the *Small* context, use a value of *1,000*



Double-click the **tRowGenerator** component again. In the **Number of Rows for RowGenerator** field, specify an expression of `context.NumRows`, then click **OK**.



The size of the data set is now configurable via a context variable whose value is determined by selecting a context at runtime.

This simple Job will generate as many rows of sample data you want, and sort one data flow by the first name, and the other by the last name. Depending on how many rows are generated it can be a memory and CPU intensive Job.

Basic Run

WARNING:
Before proceeding, make sure that the **Target Exec** is set to Localhost.

1. RUN THE JOB

   Select the *Small* context and run the Job. The generated names are sorted by *FirstName* (**tLogRow_1**) and *LastName* (**tLogRow_2**), with the output displayed in the console.

2. CLEAR THE CONSOLE

   Click the **Clear** button to clear the output from the console in preparation for the next section.

This example use case generates a set of data similar to a sample client list for application test purposes. The data could very easily have been read in from a file or database, or saved to a file or database as well.

Memory Run

Now you will run the Job, altering conditions so that it uses more system resources, and monitoring it real-time while it runs.

1. RUN THE JOB

   Switch to the **Memory Run** tab of the **Run** view. Select the *Medium* context and run the Job. Now the Job is using more resources as additional records are generated and processed. As time passes, memory and CPU usage is monitored and displayed.

   The details are displayed while the Job is still running and after it completes. The Job should approximately 30 seconds to process all rows. Start and end times are reported in the **Job execution information** pane to the right of the graphs.

2. EXAMINE THE RESULTS

Hover the mouse pointer over various areas of the memory usage graph (above) and the CPU usage graph (below). Additional details specific to that point of time are displayed.

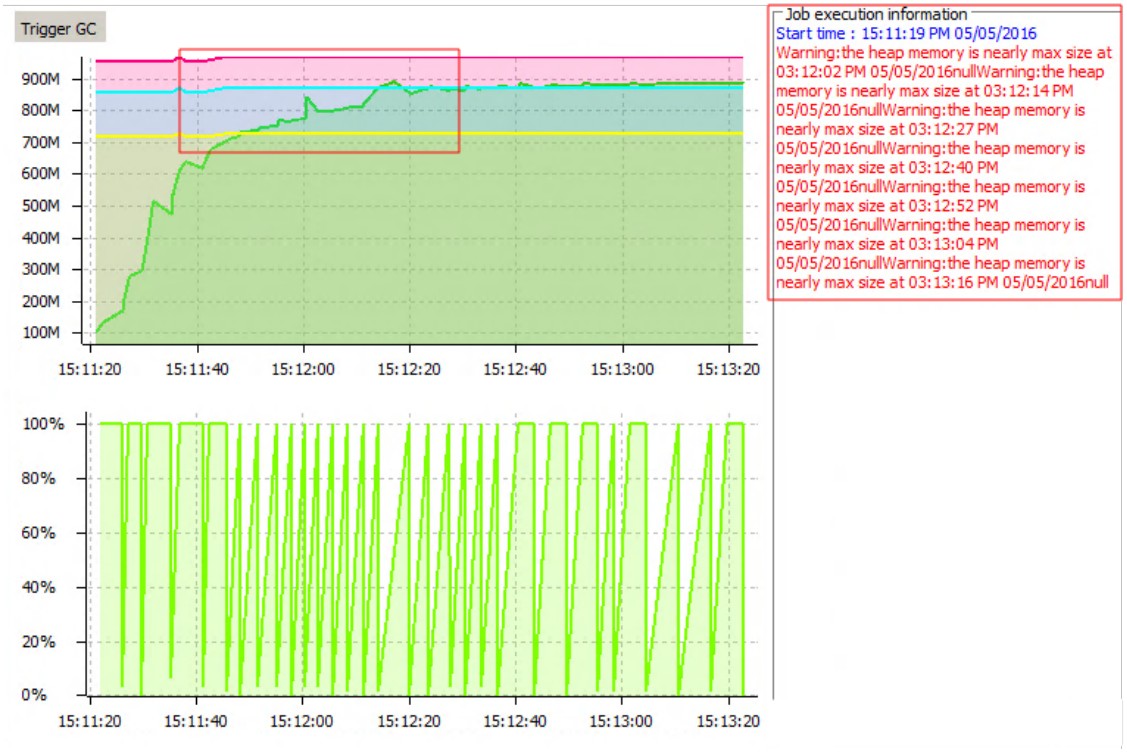
Memory Run Violations

Processing one million records used enough resources to monitor and graph, although even at 100% CPU utilization, no memory violations occurred. There are built-in thresholds that if crossed will trigger warnings in real-time.

1. RUN THE JOB

   Still in the **Memory Run** tab of the **Run** view, select the *Large* context and run the Job.

   Notice the CPU usage spikes continuously, and the memory usage continues to increase as time passes. Eventually warnings are displayed in the **Job execution information**.



   Although your results will vary somewhat, the Job should probably run for a couple of minutes, and after processing 8 or 9 million records, a fatal exception occurs, which can be seen on the console in the **Basic Run** tab.

Resolution Options

If you run low on resources (such as memory in the example), there are several things to consider that may assist in resolving the issue.

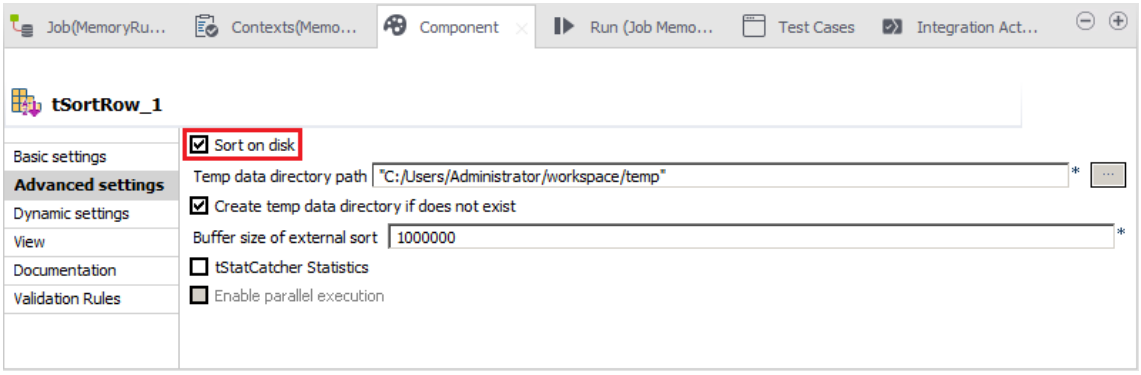
- Changes to the Job may be required.
- Changes to your JVM settings may be required. In the **Advanced settings** tab of the **Run** view, there are JVM options you can change to increase the amount of memory available to your Job. For example:
  - *-Xms512M*: initial memory pool size
  - *-Xmx2048M*: maximum memory pool size

  Other legal JVM command-line arguments can be added here as well.
- Often, specific components in your Job can be configured to use memory more efficiently.
- Ultimately, you may need to upgrade your system hardware to support your data sets and workload (additional cores or memory, for example).

In this section, you will overcome memory limitation issues by changing component configuration and tuning JVM resources. The goal is to leave the **tRowGenerator** processing 10 million records while enabling the Job to complete without errors or unreasonable delay.

1. SORT ON DISK

   For both **tSortRow** components, enable the **Sort on disk** option in the **Advanced settings** tab of the **Component** view.



2. RUN THE JOB

   Test these changes by performing another **Memory Run** with the *Large* context selected.

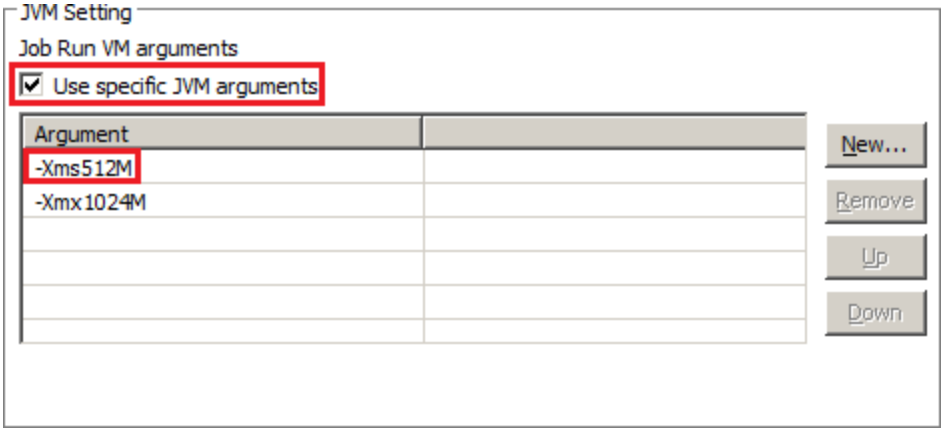
   As you would expect, the Job progresses slower as disk is relied upon to relieve pressure on system memory. The CPU is heavily used, and memory violations occur a few minutes into processing. Although **tReplicate** completes now, the upper flow slows drastically after a few minutes and about two million records. After approximately 5 minutes, processing is nearly halted at about 3.5 million records processed.

   Terminate the Job by clicking the **Kill** button, if it has not terminated on its own already.

3. INCREASE JVM INITIAL MEMORY

   In the **Advanced settings** tab of the **Run** view, select the **Use specific JVM arguments** option. Double-click the invidual arguments and set them as follows:

   - *-Xms512M*
   - *-Xmx1024M*



   This will double the size of the initial memory pool.

4. RUN THE JOB

   Return to the **Memory Run** tab and run the Job again, observing the results in real time. You should see something similar to the following:

- Approximately 2 minutes in, about 3 million records have processed on the upper flow. Memory usage begins to climb steadily as the CPU struggles and record processing slows dramatically.
- Approximately 4 minutes in, about 4 million records have processed on the upper flow. Multiple memory warnings get triggered, the CPU pegs (100% utilization) and the Job does not complete; a good indication that the Job is thrashing.

This change clearly relieved some initial congestion, but eventually resource availability limited the progression of the Job.

Kill the Job (if it has not terminated on its own yet).

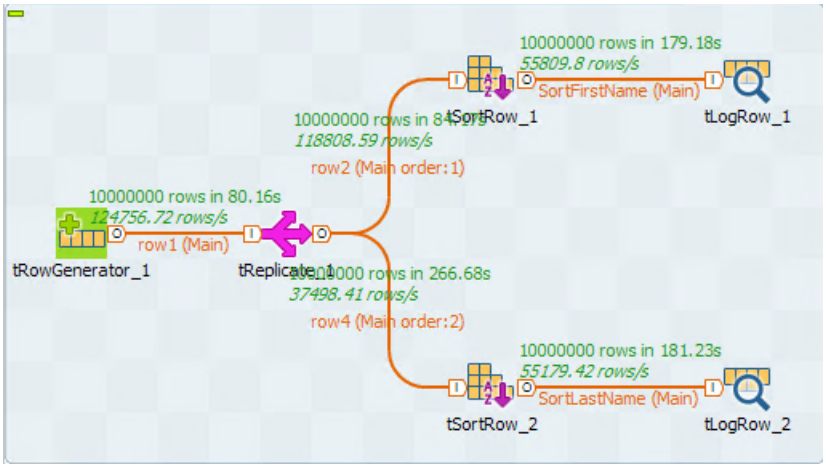
5. INCREASE JVM MAXIMUM MEMORY

   Increase the maximum pool size by modifying the second argument to *-Xmx8192M*. Leave the initial memory setting (*-Xms512M*) as is.

6. RUN THE JOB

   Run the Job again and observe the behavior. Again, your observations will differ slightly but should be similar to the following:

   - After just 2-3 minutes, 9 million records have processed through the upper **tSortRow** without memory climbing sharply. The CPU runs at 100% often, but memory holds.
   - After 3-4 minutes all 10 million records have processed through the upper flow.
   - After 1-2 additional minutes memory starts to climb, but no thresholds have been violated, hence no warnings issued. Progress has slowed, but **Memory Run** is still updating and reporting information in real-time, as the lower **tSortRow** flow processes records.
   - After about 8 minutes the Job completes without errors (although a warning or two is not uncommon).



NOTE:
Even after both the upper and lower flows have finished processing all 10 million records, the **Job execution information** will not show the end time stamp. The run console takes a while to log all of the output. When that completes, the end time stamp is issued.

Doubling memory allocation and observing the results is not a bad guideline, but it can end up wasting memory. Additional fine tuning could be applied. If you increase the memory size beyond available physical memory, your Job will not run because the JVM will fail to initialize.

The next step is to take a closer look at debugging tools.

Debugging Jobs Using Debug Run

Overview

There are several approaches to debugging Jobs:

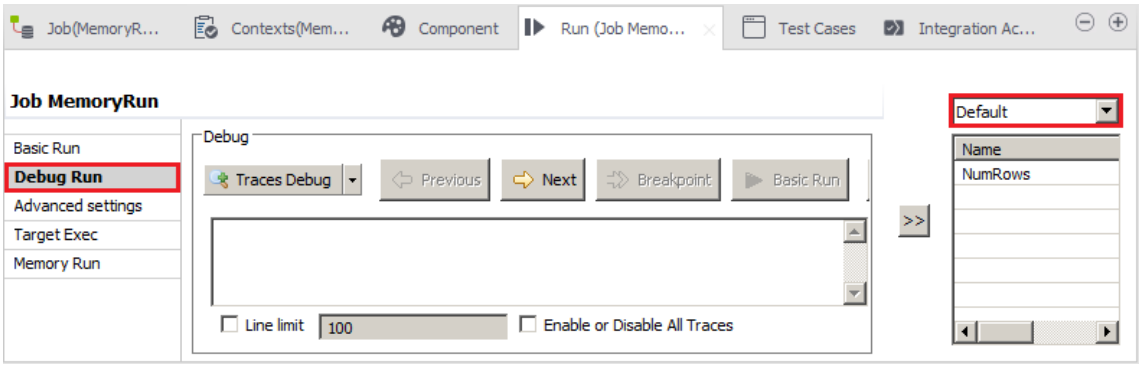
- The **log4jLevel** feature increases verbosity on the execution console so that you can get more information on the execution of your Jobs.
- **Trace Debug** is a full-fledged, real-time debugging tool that is beneficial regardless of whether or not you are an experienced Java developer. It allows you to set traces and breakpoints, then step through the execution of your Job while viewing the actual data row by row. It is a valuable debugging tool that does not require deep knowledge of Java.
- **Java Debug** is another real-time debugging tool that enables full source-level debugging of the Java code implementing your Job. While powerful, this tool is targeted at experienced Java developers.

In the *Data Integration Basics* course, you used the **log4jLevel** feature to get more information on the execution of your Jobs. In this lab, you will focus on the Trace Debug. Usage of the Java Debug tool requires deeper knowledge of Java and is beyond the scope of this course.

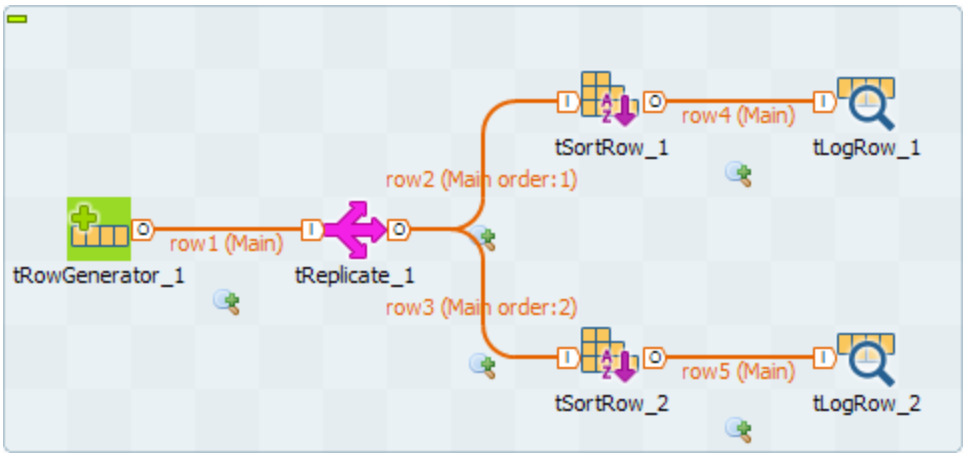
Trace Debug

1. PREPARE THE DEBUGGER

   In the **Run** view of the **MemoryRun** Job, switch to the **Debug Run** tab. Select the *Default* context to reduce the number of records that will be processed, as the intent of this exercise is to debug rather than push resource limits.

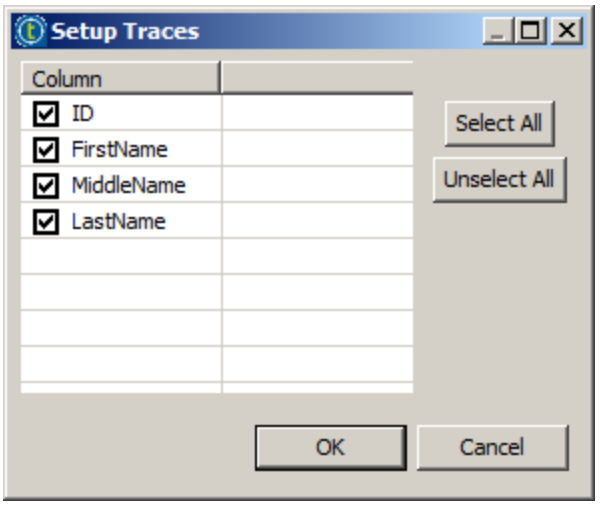


   By default, a trace is set up for each connection. These are signified by icons ( ) overlaid into the **Designer**.



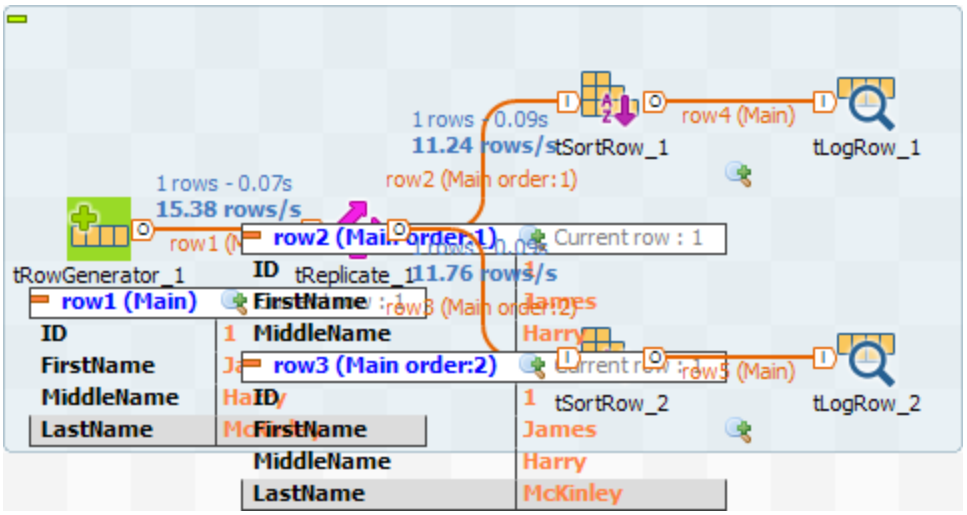
Also by default, each trace is configured to display all data columns. Verify this by double-clicking one of the trace icons ( ).

Click **OK** to dismiss the **Setup Traces** window.



2. STEP THROUGH THE DATA PROCESSING

   In the **Run** view, click the **Next** button to process the first row of data.

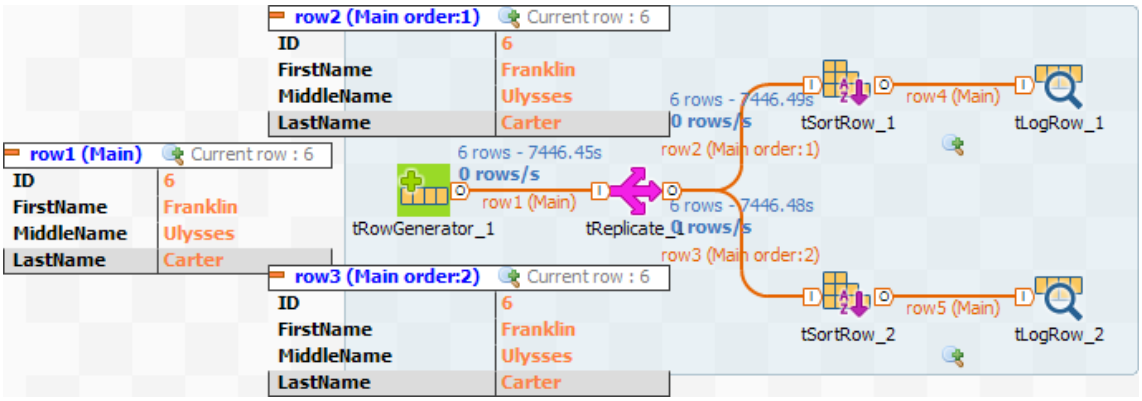


The *Main* row data is displayed for each trace as you step through execution of the Job. However, it's difficult to read as is, as many different traces are displayed and physically overlap on the screen. You will fix this in the next step.

3. REORGANIZE THE DISPLAY

   Drag the row data around so you can read it better. For example, drop rows from the lower flow beneath the Job, and the other row to the upper right.

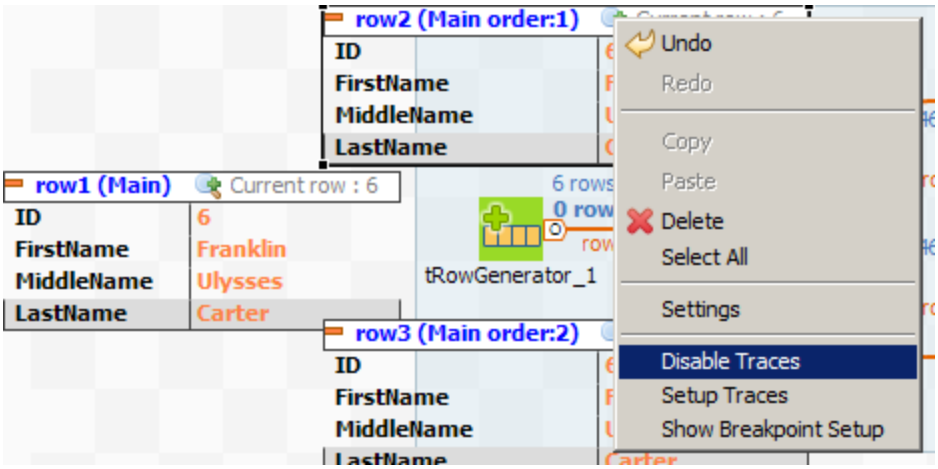
   Click the **Next** button several more times and observe the record data as it changes with each row processed. The data for each row should be easy to read.



4. DISABLE TRACES

   Imagine now that you are not interested in the upper flow (that is, the output sorted by first name). In this case, you want to reduce clutter by disabling the traces you are not interested in; that is, **row1**, **row2**, and **row4** (as shown in the figure above).

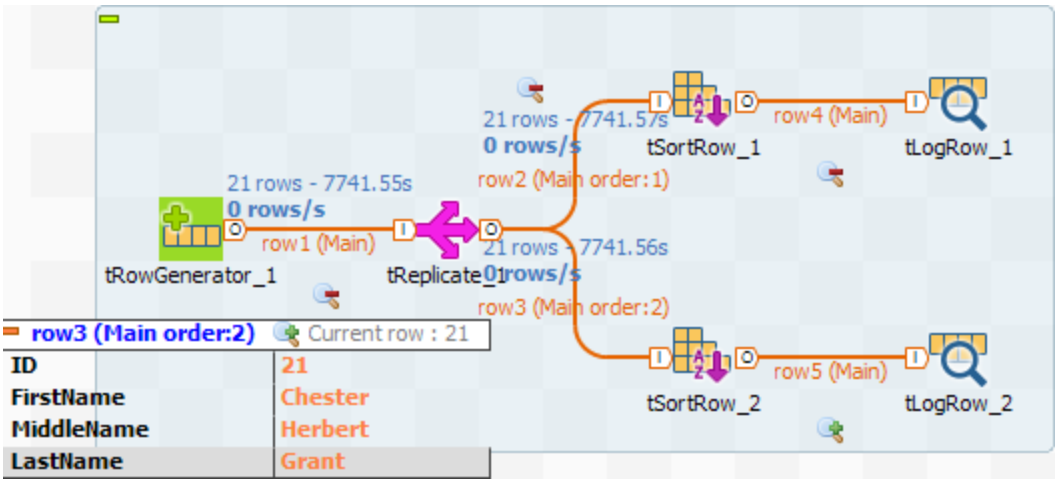
   Right-click on the **row2** trace icon ( ) between the upper **tReplicate** and upper **tSortRow** components and select **Disable Traces**.



The regular trace icon ( ) changes ( ), indicating that the trace is disabled for that flow.

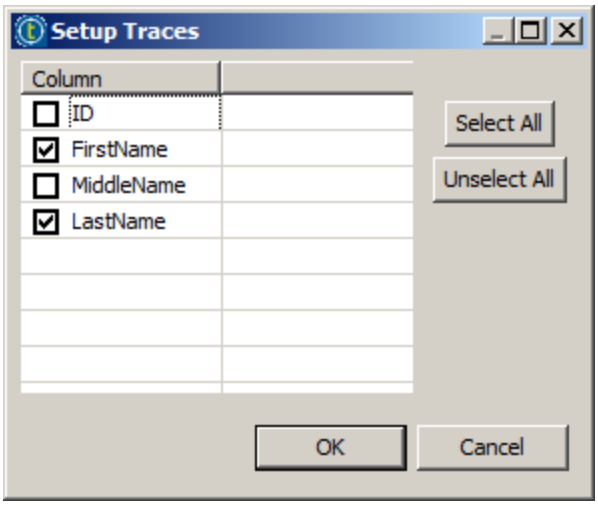
Repeat this for the trace between the upper **tSortRow** and **tLogRow** (**row4**) and also for the trace between the **tRowGenerator** and **tReplicate** (**row1**).

Click **Next** several times to step through the Job one row at a time. Only one record is displayed with each click of the **Next** button.

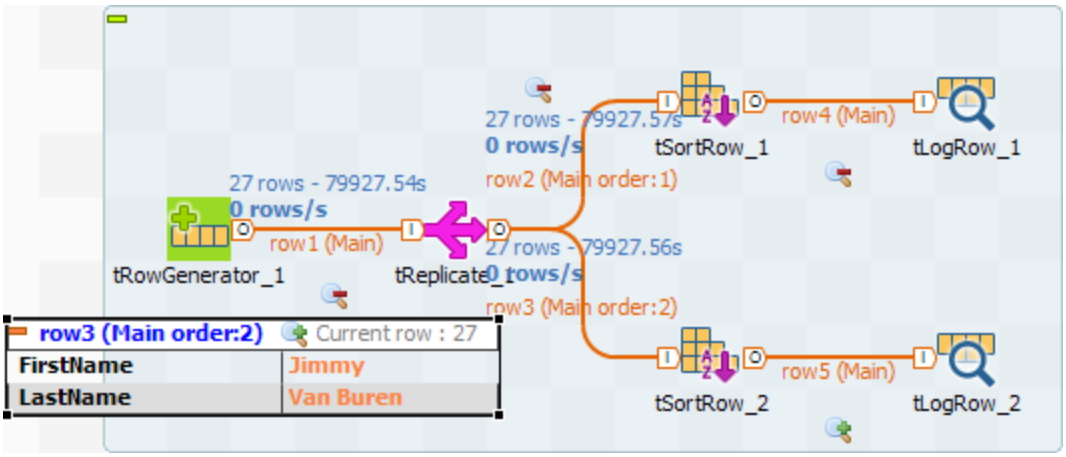


5. CONFIGURE DISPLAYED COLUMNS

   By default, all columns are displayed. In many cases, you may only be interested in a subset of the columns for debugging purposes. Change this now by double-clicking the trace icon ( ). In the **Setup Traces** window that appears, clear **ID** and **MiddleName**, so that only the **FirstName** and **LastName** columns will be displayed. Click **OK**.



In the Run view, click **Next** a few times. Now only the first and last names are displayed.



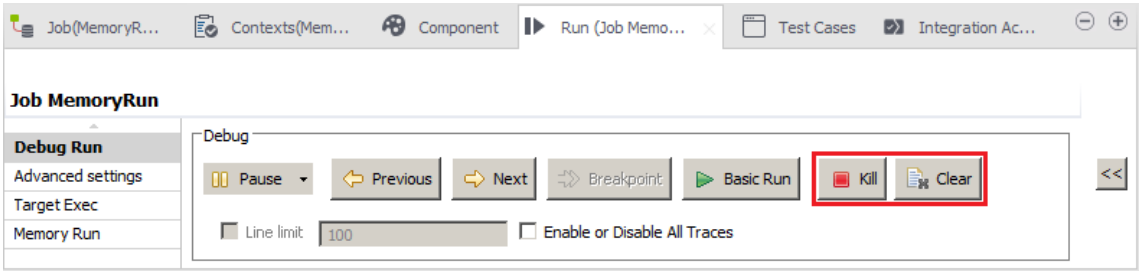
To turn a disabled trace back on, right-click the disabled trace icon ( ) and select **Trace Enable**. Similarly, you can double-click the trace to reconfigure what output is displayed as you step through the data.

Pause and Resume

Whenever you have a larger data set, single-stepping through the data as you have been doing is not feasible. In some cases, you might be aware of a problem that only occurs after, say, the 10,000th row. Using the pause and resume feature can help get to a suspected trouble spot in your data quickly.

1. RESTART THE DEBUGGER

   Stop the execution of the Job and clear the display by clicking the **Kill** and **Clear** buttons in the **Run** view.



2. START THE JOB EXECUTION

Notice that the **Traces Debug** button not only allows you to choose a debugging mode, but it is also clickable. Click it to start execution of the Job. When you do, a few things happen:

- The button transforms into a **Pause** button, allowing you to temporarily halt execution
- The Job begins running in debug mode, displaying records as they are processed

3. PAUSE AND RESUME EXECUTION

   Click the **Pause** button. Execution pauses, and the button transforms into a **Resume** button. Click **Resume** to continue execution.

   Cycle through clicking **Pause** and **Resume** several times as you view the records processed.

Conditional Breakpoints

In some debugging scenarios, you might be aware of a problem that only occurs with a particular record. For example, the 10,000th row, or perhaps a row where the last name is "Lincoln".

Using a more advanced feature like a conditional breakpoint can help zero in on troublesome areas, where you can then process data a row at a time while evaluating the data for troubleshooting purposes. This exercise shows you how.

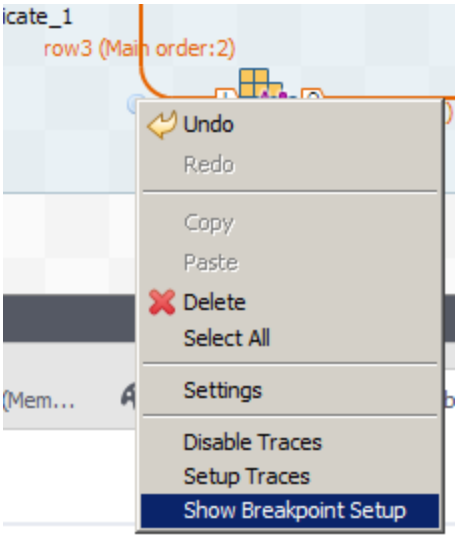
In this example, you will process the first 10 rows, then start stepping through a row at a time. The number of rows is arbitrary of course, and can be applied to any number of rows.

1. RESTART THE DEBUGGER

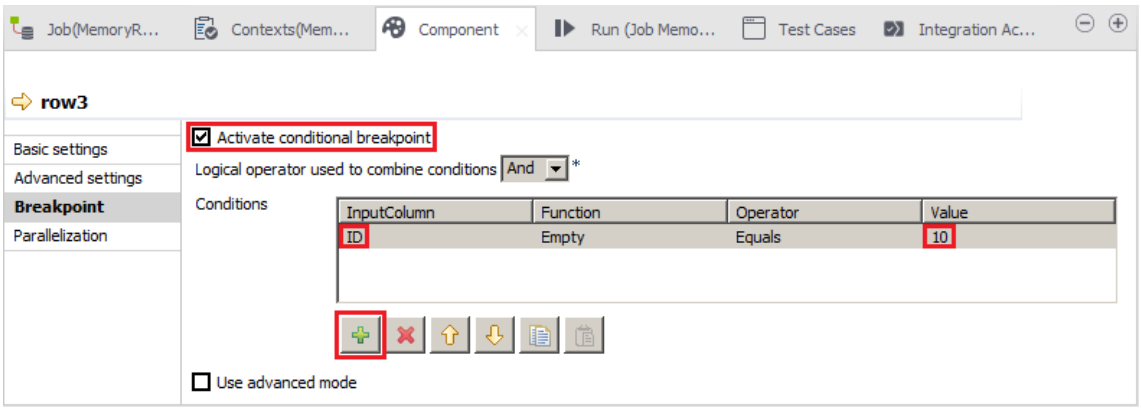
   Stop the execution of the Job and clear the display by clicking the **Kill** and **Clear** buttons in the **Run** view.

2. CONFIGURE A BREAKPOINT

   Right-click the trace between the **tReplicate** and lower **tSortRow** components and select **Show Breakpoint Setup**.



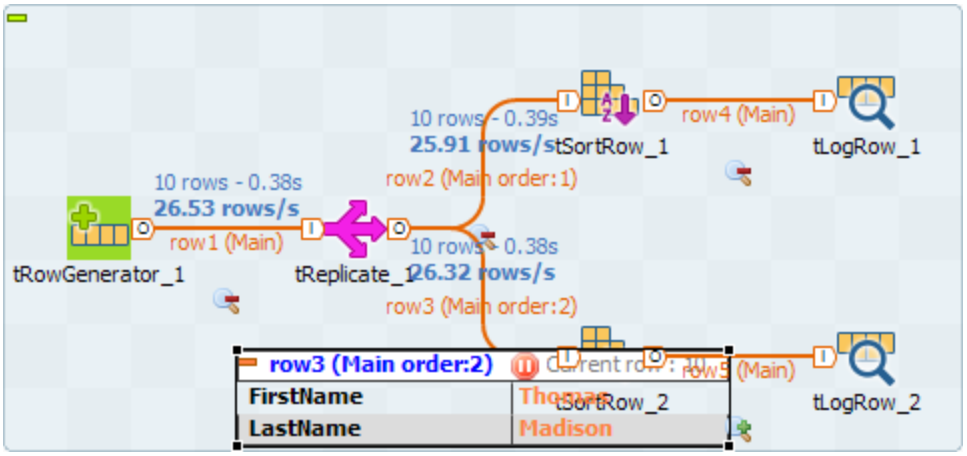
Then **Breakpoint** tab opens below. Select the **Activate conditional breakpoint** check box. Then, click the **Add** button ( ). Configure the condition such that the *ID* column equals *10*.



3. START THE DEBUGGER

   Return to the **Run** view. Notice the trace icon has changed from to to indicate that a conditional breakpoint is now active on that trace.

   Click the **Traces Debug** button to run the Job. The Job processes until the *ID* is 10, then pauses.



   You can now single-step through the process using **Next** and **Previous**.

   NOTE:
   You can only go back a maximum of four rows.

4. RESUME THE DEBUGGER

   Click **Breakpoint** to resume execution until the next breakpoint. Because there are no more breakpoints set up, the Job will run to completion.

Multiple Conditions

As a similar use case, perhaps you know there is a problem with a specific row where the name is "Warren Wilson". This exercise illustrates how to configure a breakpoint to locate such a record.

1. RESTART THE DEBUGGER

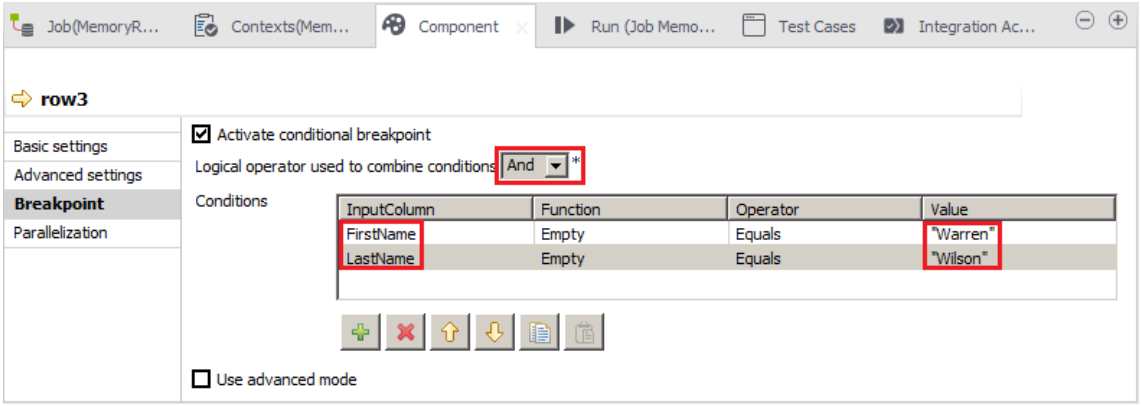
   Stop the execution of the Job and clear the display by clicking the **Kill** and **Clear** buttons in the **Run** view.

2. RECONFIGURE THE BREAKPOINT

As you did in the previous exercise, right-click the trace between the **tReplicate** and lower **tSortRow** components again and select **Show Breakpoint Setup**.

Remove the existing condition by selecting it and clicking the **Remove selected items** button ( ).

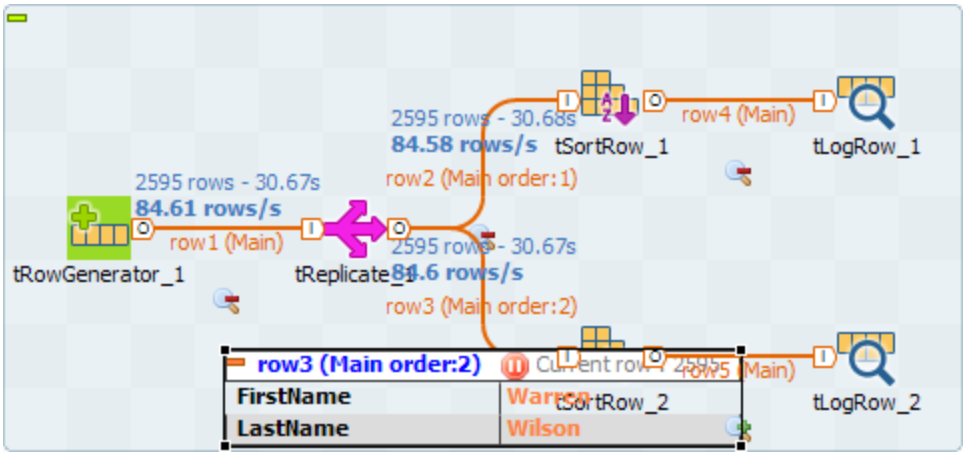
Add two new conditions where *FirstName* equals *"Warren"* and *LastName* equals *"Wilson"*. Leave the **Logical Operator used to combine conditions** set to *And*.



6. START THE DEBUGGER

Return to the **Run** view. This time, select the **Large** context (this increases the amount of random records generated and hence increases the chance of a match against the breakpoint condition).

As before, click the **Traces Debug** button to run the Job. Rows should be processed until the *FirstName* is Warren and *LastName* is Wilson.



As before, you can now single-step through the process using **Next** and **Previous**.

3. RESUME THE DEBUGGER

Click **Breakpoint** to resume execution until the next breakpoint. Because of the size of the data set, the likelihood of encountering another record with the same name is fairly high, which will trigger the breakpoint again.

You have finished this Lesson and now it's time to Wrap-Up.

Wrap-Up

In this lesson, you built a basic Job that is easy to reconfigure so that it uses relatively little or a massive amount of system resources. You used the **Memory Run** tab in the **Run** view to monitor host CPU and JVM memory usage in real-time as your Job ran. You explored several options available to you in the Studio for addressing issues tied to limited resources, such as utilizing too much memory. You learned how to use the **Debug Run** tab as well. Most of the time was spent using **Traces Debug**, where you set and configured various Traces and breakpoints as you stepped through processing rows of client data. These features are very practical and can help java developers and non-developers alike as they troubleshoot issues with Jobs in the Studio.

Next step

Congratulations! You have successfully completed this lesson. To save your progress, click **Check your status with this unit** below. To go to the next lesson, on the next screen, click **Completed. Let's continue >**.

This page intentionally left blank to ensure new chapters start on right (odd number) pages.

LESSON 5

Activity Monitoring Console (AMC)

This chapter discusses the following.

Introduction 68
Activity Monitoring Console (AMC) 69
Configuring Statistics and Logging 70
Using the Activity Monitoring Console (AMC) 76
Challenge 84
Solution 85
Wrap-Up 86

Introduction

Activity Monitoring Console (AMC)

Overview

The Talend Activity Monitoring Console (AMC) is an application that allows you to log information and view statistics about your Jobs. In this lesson, you will configure a Project in Talend Studio to capture information about your Jobs and then access AMC from Talend Studio to view that information. Note that you can also access AMC from the Talend Administration Console (TAC), a topic covered in the *Data Integration Administration* course.

Objectives

After completing this lesson, you will be able to:

- Configure a Talend project to capture statistics and logs
- Configure a Talend Job to capture statistics and logs
- Access the Talend Activity Monitoring Console from within Talend Studio
- List the kinds of information available in AMC

The first step is to configure the project to capture statistics and logs.

Configuring Statistics and Logging

Overview

The Activity Monitoring Console (AMC) allows you to view historical information about Job execution. The AMC uses either files or databases to store that information.

If you want to maintain information for a particular Job, you first need to configure the Job and the Project to store information in the AMC database. In this lesson, you will configure it to use three tables in a MySQL database.

NOTE:
Each deployment varies depending on specific requirements. AMC is often used for development and test environments only. It is typically not configured for production systems, due to the desire for:

- **IT simplicity:** AMC requires additional ports to be open
- **Performance:** AMC increases database activity

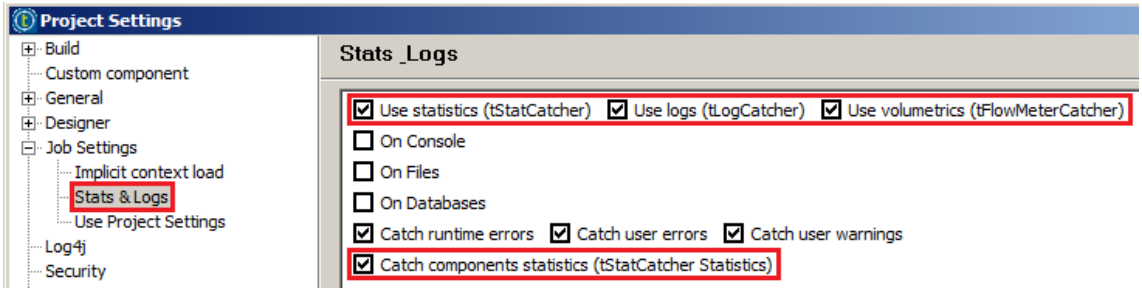
Configure the Project

1. ENABLE CAPTURE OF STATISTICS

   From the main menubar, select **File > Edit Project properties**.

   In the **Project Settings** window that appears, select **Job Settings > Stats & Logs**. In the **Stats Logs** pane, select **Use statistics**, **Use logs**, and **Use volumetrics**. Selecting these three enables the capture of statistics, logs, and volume information relating to the respective catcher components.

   Select **Catch components statistics**. This will capture all events generated by Jobs, including statistics about individual components rather than just the Job as a whole.



2. CONFIGURE THE DATABASE STORAGE

   Now select **On Databases**. This option captures the information in a database rather than in a file or as console output.

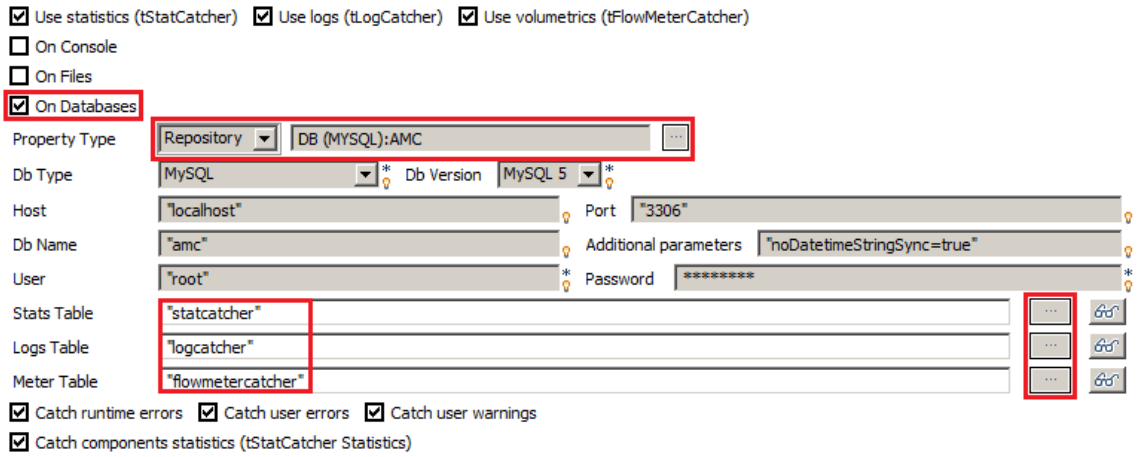
   Set **Property Type** to *Repository* and click the button marked with an ellipsis to choose the *AMC* database connection.

   Use the buttons marked with ellipses below to specify the tables as follows:

   - Specify *"statcatcher"* for **Stats Table**
   - Specify *"logcatcher"* for **Logs Table**
   - Specify *"flowmetercatcher"* for **Meter Table**

   These are the tables in the preconfigured AMC database that will store the three different kinds of Job information.

When this configuration is complete, click **OK**.



Import and Configure a Job

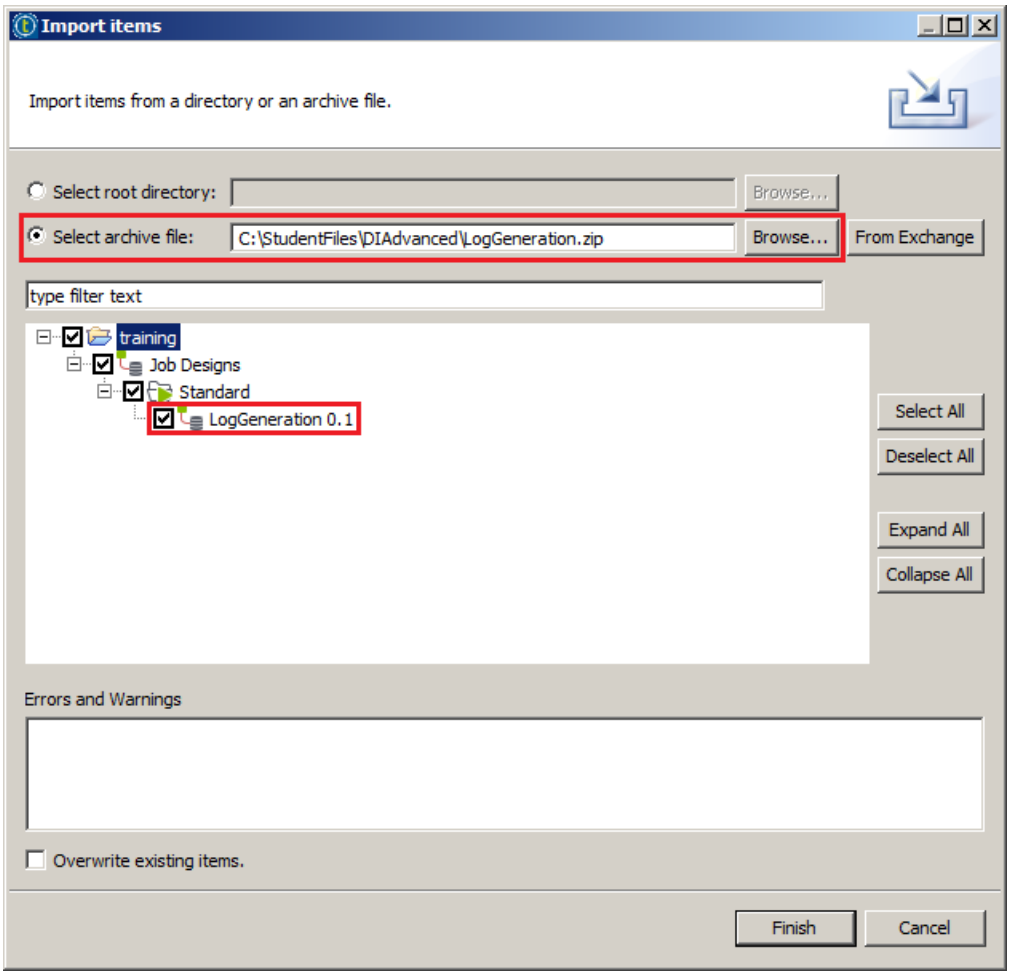
The *LogGeneration* Job was designed to generate a range of information that you can examine with the AMC. In this exercise you will import and configure it.

1. IMPORT THE JOB

   Right-click on **Repository > Job Designs** and select Import items.

   Click **Select archive file** and specify the path *C:\StudentFiles\DIAdvanced\LogGeneration.zip*.

   Then, select **training > Job Designs > Standard > LogGeneration 0.1** and click **Finish**.

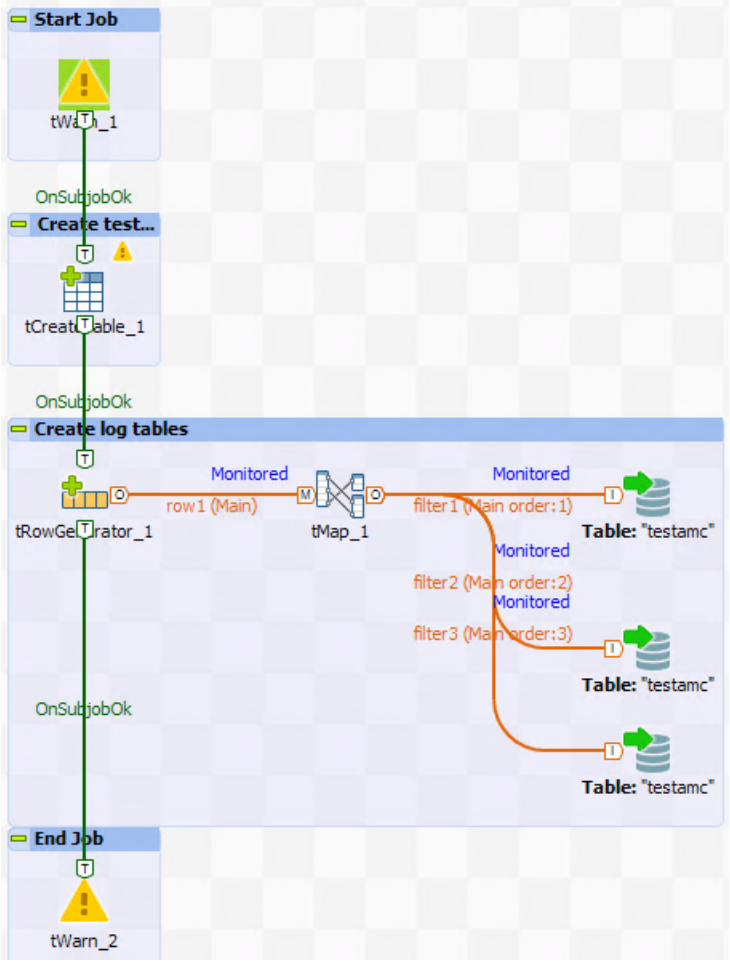


The Job appears in the **Repository**.

2. OPEN THE JOB

   Open the Job and, in the **Update Detection** window that appears, click **OK**.

   Take a moment to examine the Job.

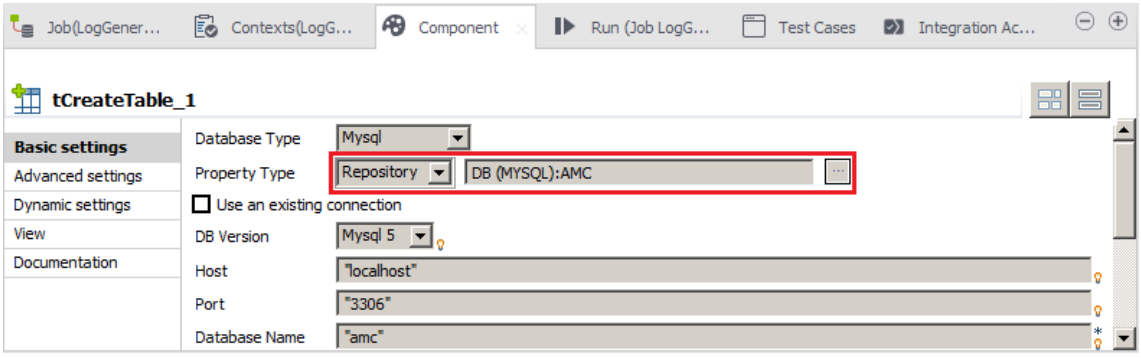


The Job issues a warning, creates a database table called *testamc* in the *amc* database, generates 1000 rows of random data, filters the data before writing it to the database, and then issues a final warning when the Job is complete.

3. RECONFIGURE COMPONENTS

Before you can run this Job and have it store logging information properly, you need to make a few minor modifications. Begin by double-clicking the **tCreateTable** component to open the **Component** view.

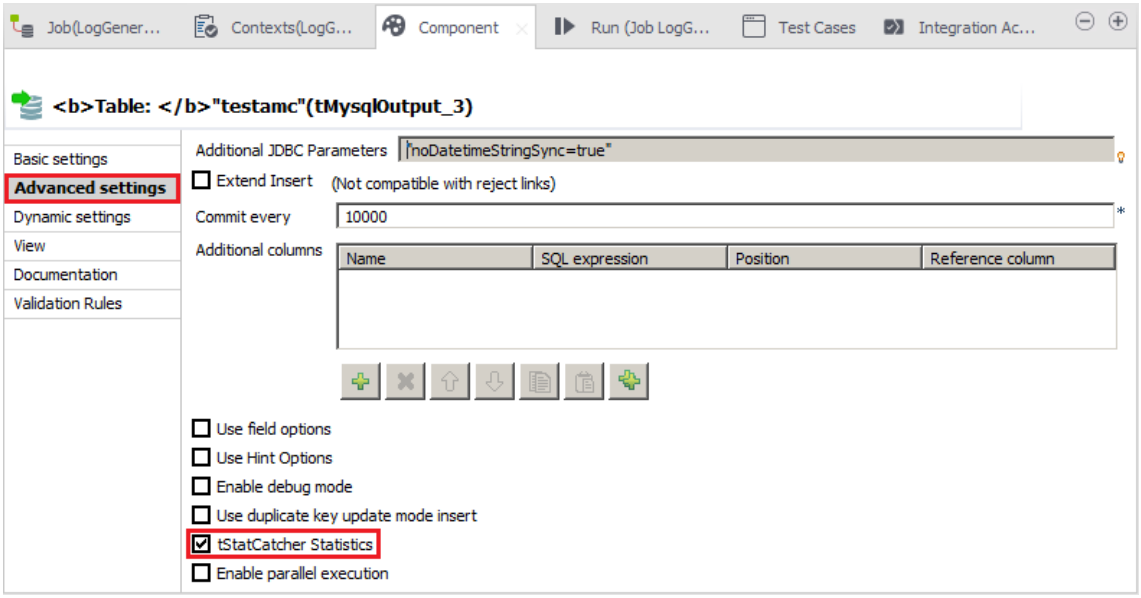
Change the **Property Type** to *Repository* and use the button marked with an ellipsis to choose the *AMC* database connection.



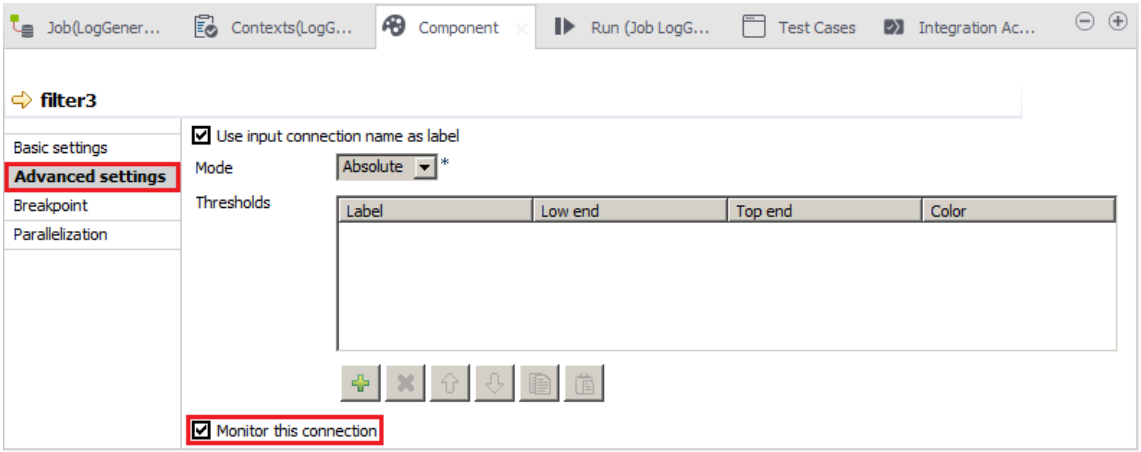
Make the same change for each of the three **tMysqlOutput** components (labeled **Table: "testamc"**).

4. INSPECT MONITORING PROPERTIES

With one of the **tMysqlOutput** components selected from the previous step, click the **Advanced settings** tab in the **Component** view. Notice that the **tStatCatcher Statistics** check box is selected, allowing information about this component to be collected.



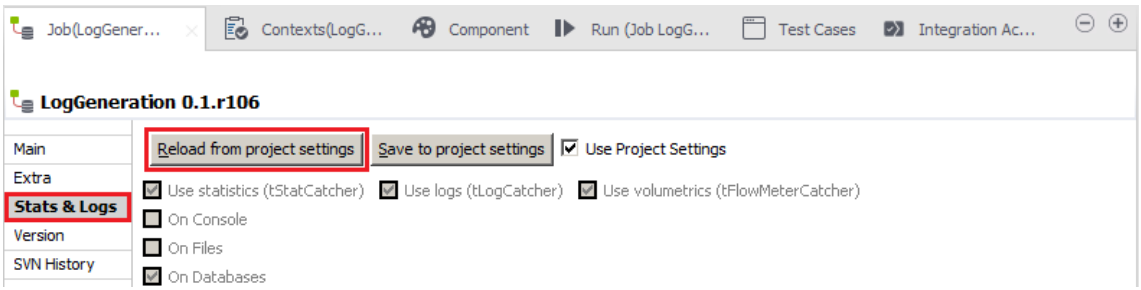
Now, double-click the output flow connecting the **tMap** component to that **tMysqlOutput** component. Click the **Advanced settings** tab in the **Component** view. Notice that the **Monitor this connection** check box is selected, and also the **Monitored** label in the **Designer**. This indicates that the number of rows of data flowing through this connection is tracked when the Job runs, storing the flow meter information for the AMC.



5. RECONFIGURE JOB PARAMETERS

Although you already enabled statistics and log monitoring for the project as a whole, as well as several individual components within the Job, you now you need to enable it for this specific Job.

Open the **Job** view and click the **Stats & Logs** tab. Click the **Reload from project settings** button, and click **OK** in the warning window that appears.



Now this Job uses the same tracking settings that you configured for the project, so that the AMC information will be stored in the three different database tables: *statcatcher*, *logcatcher*, and *flowmetercatcher*.

6. RUN THE JOB

   In the **Basic Run** tab of the **Run** view, run the Job at least five times to generate enough tracking information for you to examine.

Now that you have run a Job configured to track statistics and logging information, you can examine that information with the AMC.

Using the Activity Monitoring Console (AMC)

Overview

You have created a Job that generated tracking information and stored it in a database. Now you will configure Talend Studio so that you can visualize that information using the AMC.

NOTE:
You can also access the AMC from the Talend Administration Center (TAC). This is explored in the *Data Integration Administration* course.

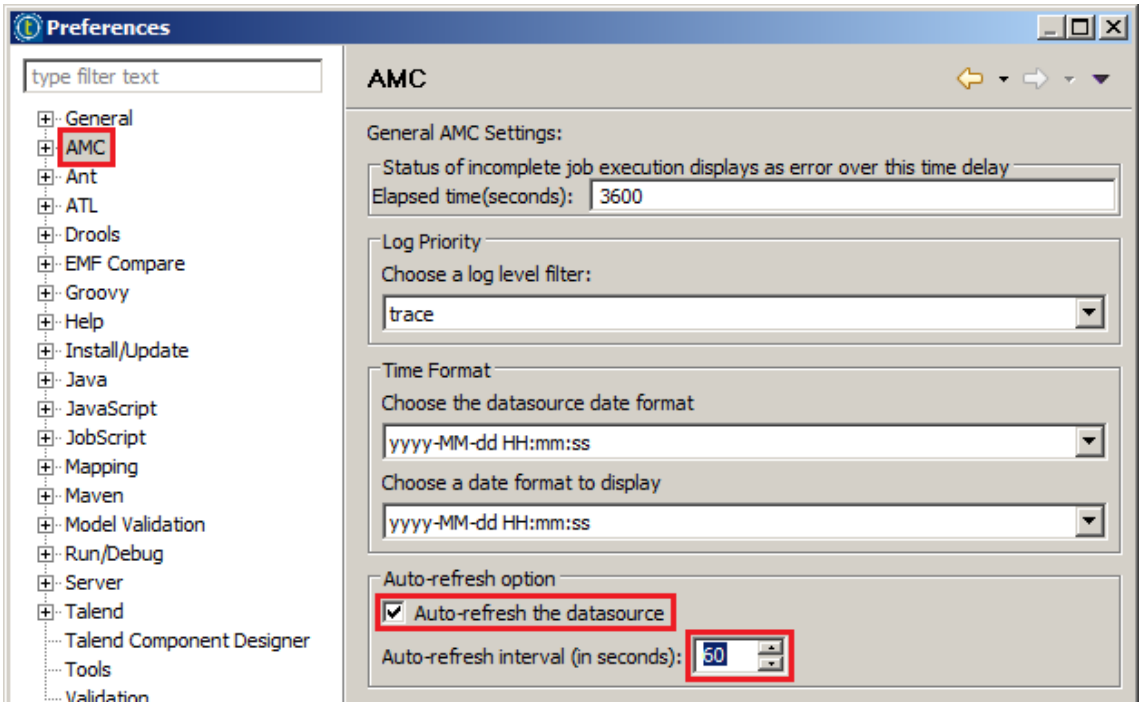
Accessing the AMC

1. CONFIGURE THE AMC

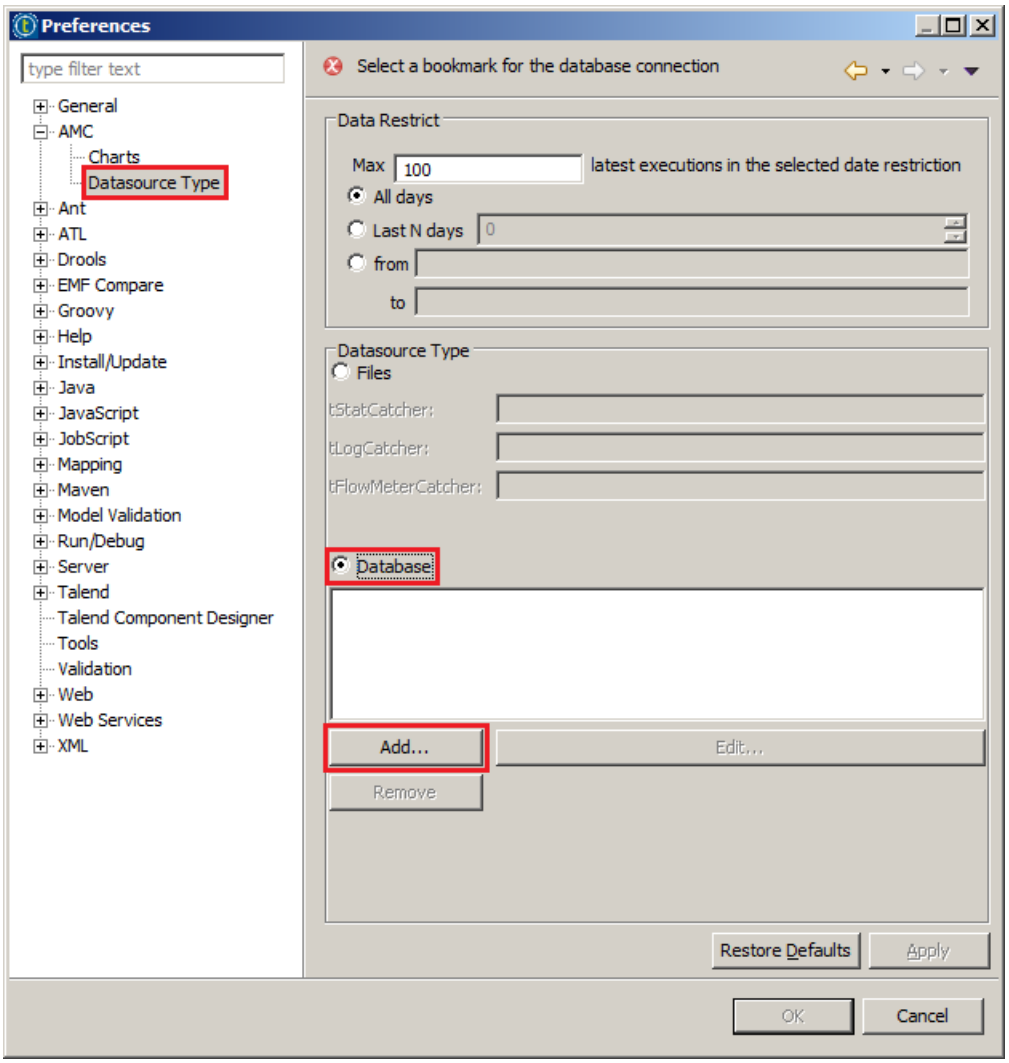
   From the main menubar, select **Window > Preferences**.

   In the **Preferences** window that appears, select **AMC**.

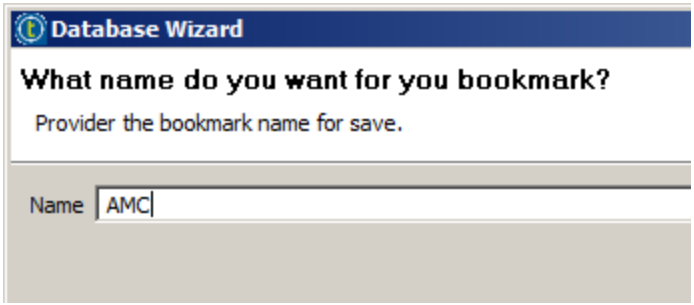
   Select **Auto-refresh the datasource** and set the **Auto-refresh interval** to *60*. Now the AMC will automatically check the database for new information once every minute.



   Now you will configure the database from which AMC will obtain its data. On the left-hand size of the window, select **AMC > Datasource Type**. Select **Database**, then click **Add**.



In the **Database Wizard** window that appears, enter *AMC* for the **Name**, then click **Next**.

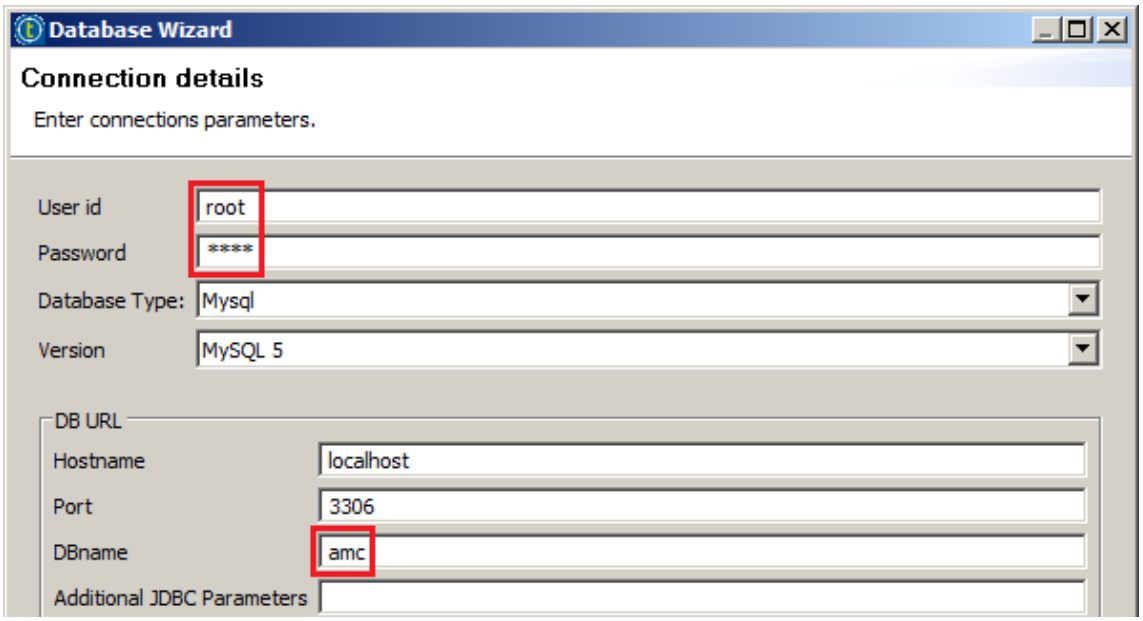


Now enter the AMC database connection details:

- *root* for both the **User id** and **Password**
- *amc* for the **DBName**

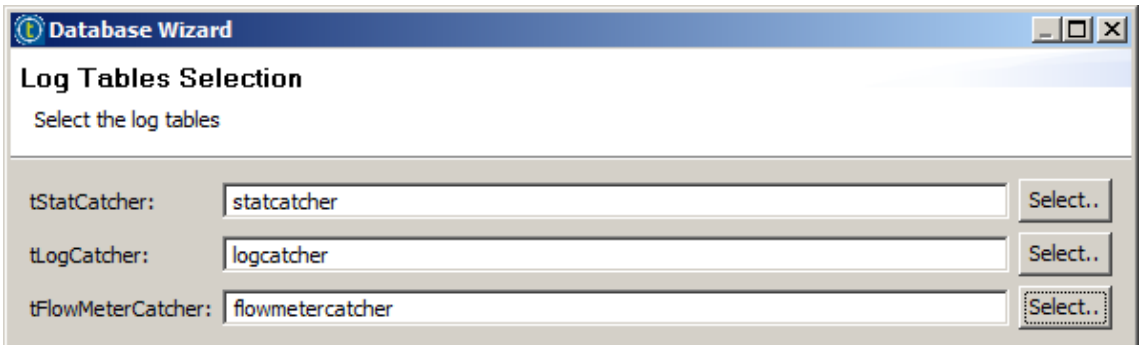
NOTE:
Recall that this specifies the same database that the Job targets when storing information. Because you can store tracking information for each Job and Project separately, you need to identify the source for the AMC to read.

When done, click the **Check** button below to verify the connection. If the check does not succeed, verify the information you entered. Otherwise, click **Next** to proceed.



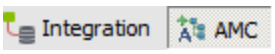
Finally, use the Select buttons to specify the tables:

- *statcatcher* for **tStatCatcher**
- *logcatcher* for **tLogCatcher**
- *flowmetercatcher* for **tFLowMeterCatcher**



When done, click **Finish**.

Back in the **Preferences** window, click **OK**. The **AMC** Perspective opens. You can verify this by observing the state of the perspective switcher at the top part of the main window:

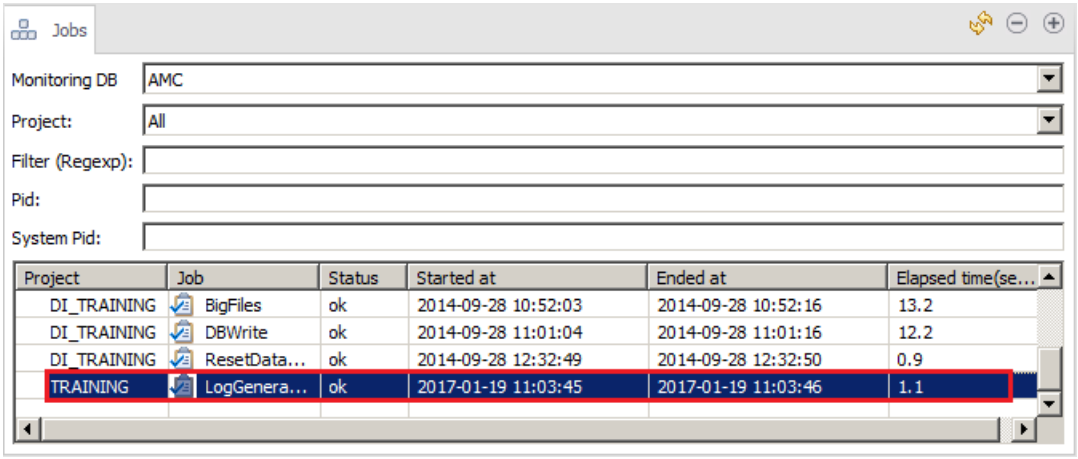


Using the AMC

1. SELECT THE JOB

   Ensure that you are in the **AMC** Perspective before continuing. Turn your attention to the **Jobs** view in the upper left portion of the window. This view shows a list of Jobs with stored information. Selecting one of these Jobs displays additional information specific to that Job.

   Locate the most recent instance of the **LogGeneration** Job and select it to display the tracking information for the Job you just ran.

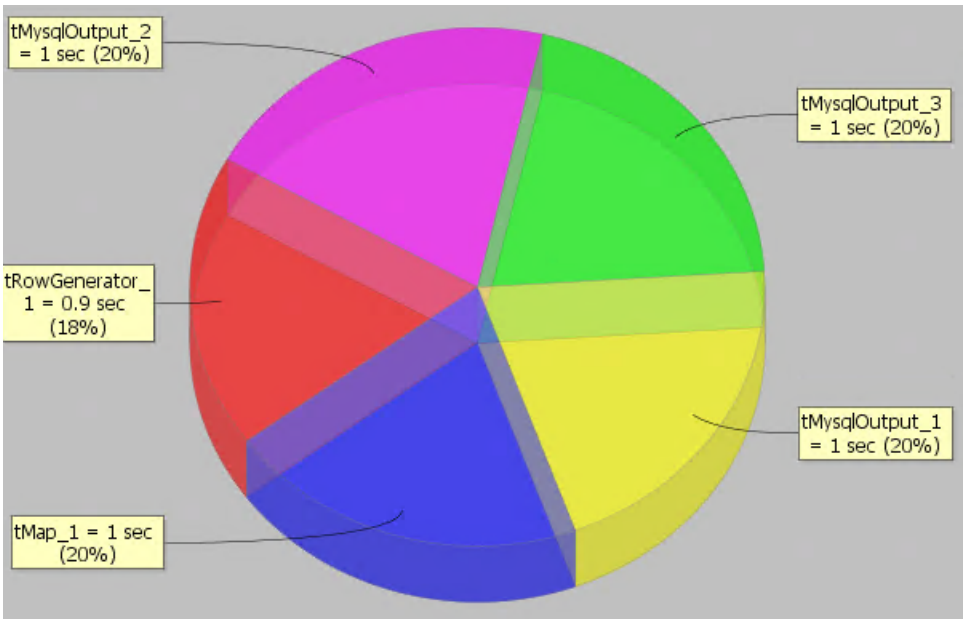


2. EXPLORING THE AMC PERSPECTIVE

   Explore the information available in the AMC, examining each of the tabs and being sure to click and expand each of the Job runs listed in the **LogGeneration detailed history** tab. Several views shows helpful information at a glance, such as:

   - The **Job** view provides a status column as well as a line graph of execution times in the **Main chart** tab
   - The **History** view lists the executions of the job with outcomes of each
   - The **Detailed history** view enables you to see the amount of time spent in each component. The **Main chart** tab provides a pie chart graphing these statistics.

   NOTE:
   The **Catch components statistics** option that you enabled earlier is what allows you to see these per-component statistics.



As you can see, the AMC makes information available that you can use to compare a detailed history of Job execution during development to compare variations in design, configuration, and performance. Recall that the auto refresh was set to 60 seconds earlier, so the graph in the **Main chart** tab will get updated each minute.

Monitor a Different Job

Now you will quickly go over several configuration items that will help you understand the impact on AMC behavior.

1. OPEN A DIFFERENT JOB

   Switch to the **Integration** perspective.

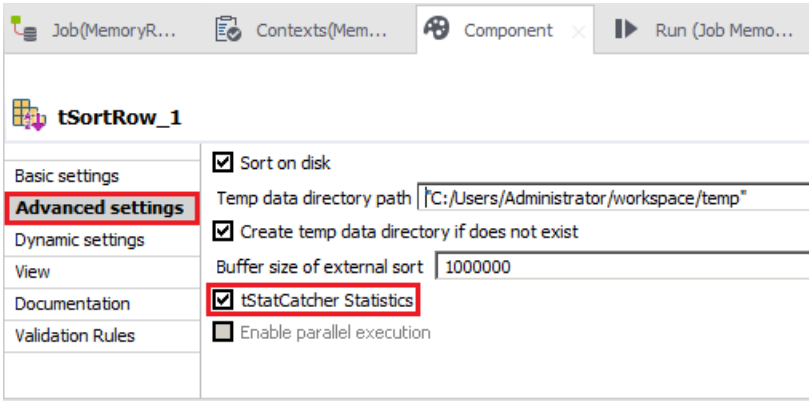
   Open the **MemoryRun** Job.

2. RECONFIGURE JOB PARAMETERS

   Open the **Job** view and click the **Stats & Logs** tab. Click the **Reload from project settings** button, and click **OK** in the warning window that appears.

3. RECONFIGURE COMPONENTS

   Double-click the **tSortRow_1** component to open the **Component** view. Click the **Advanced settings** tab and select the **tStatCatcher Statistics** check box.



Repeat this for the other **tSortRow** component as well as the two **tLogRow** components.

With this option enabled, these four components will be monitored in the **Detailed history** pie chart.

4. RUN THE JOB

   Select the *Small* context and run the Job several times.

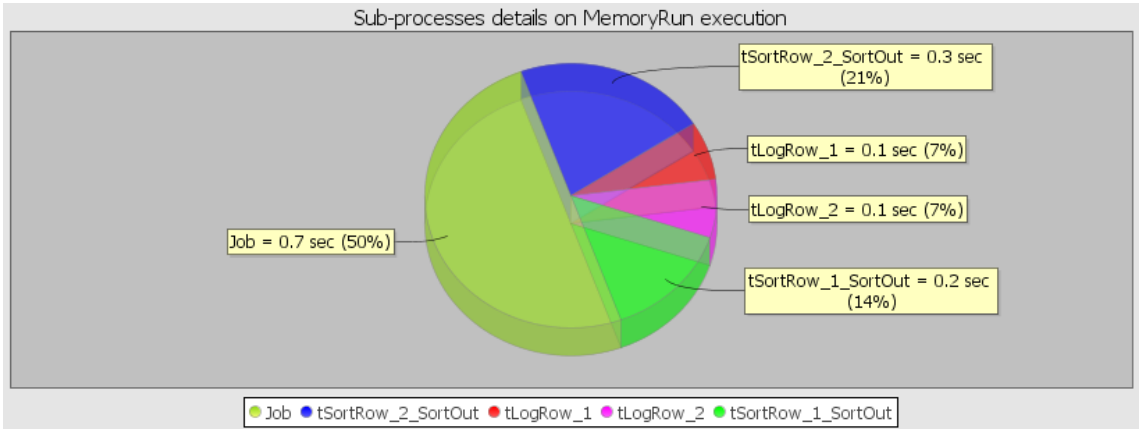
5. EXPLORE THE RESULTS

   Switch to the **AMC** perspective, and select the **MemoryRun** Job in the **Job** view.

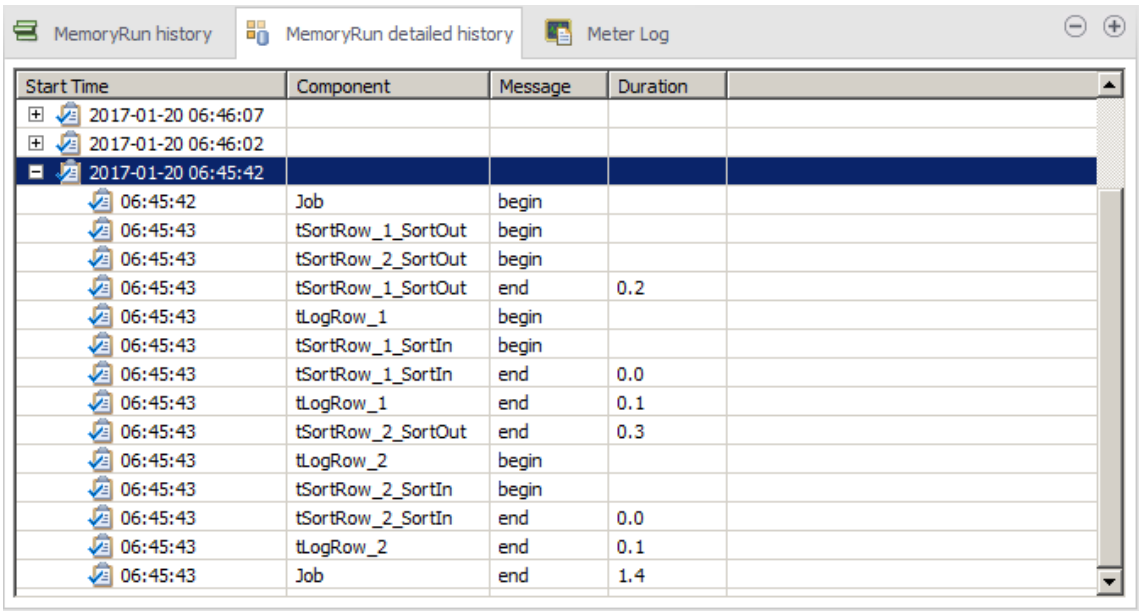
> TIP:
>
> You may have to click the refresh icon ( ) at the top of the **Jobs** view to avoid waiting for the Perspective to update automatically.

The **Main chart** shows the execution time of the Jobs you just ran. This execution time depends on the number of rows processed.

Switch to the **Detailed history** tab. Notice the pie chart includes only components for which the **tStatCatcher Statistics** check box has been selected. The chart shows components with a significant duration, while components with an insignificant duration are included in the **Job** slice with the remaining unmonitored components.



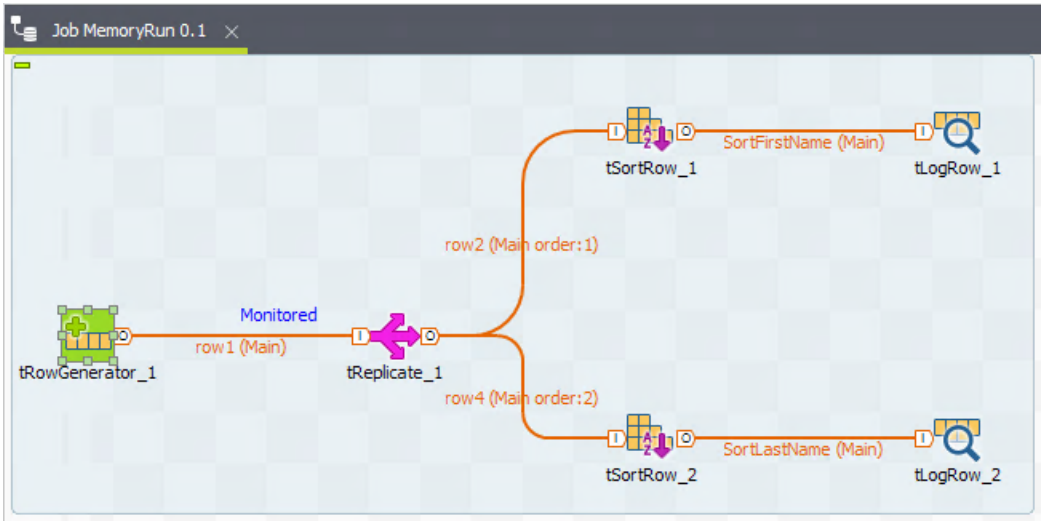
View exact durations by expanding the Job in the **Detailed history** tab.



Start Time	Component	Message	Duration
2017-01-20 06:46:07			
2017-01-20 06:46:02			
2017-01-20 06:45:42			
06:45:42	Job	begin	
06:45:43	tSortRow_1_SortOut	begin	
06:45:43	tSortRow_2_SortOut	begin	
06:45:43	tSortRow_1_SortOut	end	0.2
06:45:43	tLogRow_1	begin	
06:45:43	tSortRow_1_SortIn	begin	
06:45:43	tSortRow_1_SortIn	end	0.0
06:45:43	tLogRow_1	end	0.1
06:45:43	tSortRow_2_SortOut	end	0.3
06:45:43	tLogRow_2	begin	
06:45:43	tSortRow_2_SortIn	begin	
06:45:43	tSortRow_2_SortIn	end	0.0
06:45:43	tLogRow_2	end	0.1
06:45:43	Job	end	1.4

6. MONITOR A CONNECTION

   Switch back to the **Integration** perspective. Select the output flow of the **tRowGenerator** component, then open the **Component** view and click on the **Advanced settings** tab. Select the **Monitor this connection** check box.



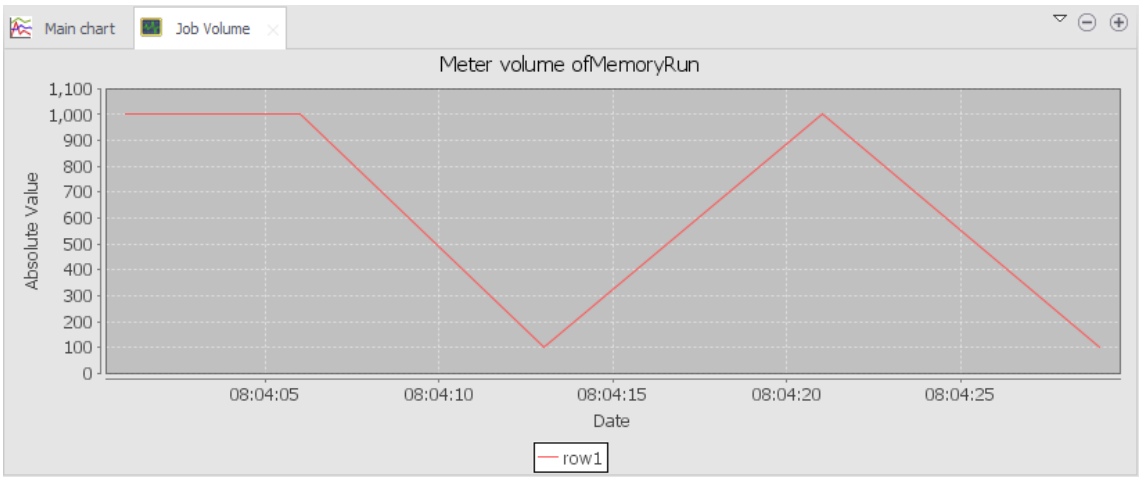
   Notice that the flow is labeled **Monitored** in the **Designer** now.

7. RUN THE JOB

   Run the Job several times, switching between the *Small* and *Default* contexts randomly.

8. EXPLORE THE RESULTS

   Switch back to the **AMC** perspective. the flow is now monitored in the **Job Volume** chart. The values displayed depend on the number of rows you have setup; that is, the contexts you selected for each run.



You have now completed this lesson and can move on to the Challenge.

Challenge

Overview

Complete this exercise to further explore the use of the AMC. See Solution for a possible solution to the exercise.

Introduce an Error

Introduce an error into the Job. For example, configure a **tSort** component to sort the first or last name by *date*, instead of *alpha*. Run the Job under these conditions and see what additional information is available from the AMC.

Next

You have now completed this lesson. It's time to Wrap-up.

Solution

Overview

This is a possible solution to the Challenge. Note that your solution may differ and still be valid.

Introduce an Error

1. Having introduced an error (such as the one suggested with the **tSort** configuration), run the Job once using the *Default* or *Small* context.
2. Switch to the **AMC** perspective and refresh the perspective by clicking the **Refresh** icon () just above the **Jobs** view. In the **Jobs** view, select the instance of the Job you just ran.
3. Notice the information in the **Execution logged events** and **Error report** tabs.

Next

You have now completed this lesson. It's time to Wrap-up.

Wrap-Up

In this lesson, you configured the project settings in Talend Studio to enable capturing statistics and logging information. You then imported a Job, configured it to use the Project settings, and ran the Job several times to generate tracking information. You then configured the AMC perspective and explored the information available within it. The use of several views in the AMC perspective displays historical data tied to Job execution in the Studio. This historical data complements the real-time resource consumption of memory and CPU examined earlier in the **Memory Run** view.

Next step

Congratulations! You have successfully completed this lesson. To save your progress, click **Check your status with this unit** below. To go to the next lesson, on the next screen, click **Completed. Let's continue >**.

LESSON 6

Parallel Execution

This chapter discusses the following.

Introduction 88
Parallel Execution 89
Writing Large Files 90
Writing to Databases 95
Automatic Parallelization 102
Partitioning 111
Wrap-Up 116

Introduction

Parallel Execution

Overview

Talend Data Integration provides several different mechanisms to take advantage of parallel execution to speed up certain kinds of Jobs. For example:

- SubJobs within a Job can be executed in parallel using multiple threads
- Specialized components can be employed to control parallel execution
- Parallel execution capabilities of certain database components can be leveraged

In this lesson, you will create Jobs that write large amounts of data to files and a database, and compare the execution times when run in sequence or in parallel. You will also perform multi-threading using dedicated components as well as the automatic multi-threading feature provided by the Studio.

Objectives

After completing this tutorial, you will be able to:

- Configure a Job to use multi-threaded execution
- Configure an individual component to use parallel execution
- Use a Talend component to run subJobs in parallel
- Use Talend components to split your data across multiple threads for multi-threaded execution

The first step is to create a Job that consumes a significant amount of resources.

Writing Large Files

Overview

To demonstrate the different methods of executing Talend Jobs in parallel, you will be using a Job that writes a million rows of random data to multiple files.

WARNING:
Ensure that you switch back to the **Integration** perspective before starting this lesson.

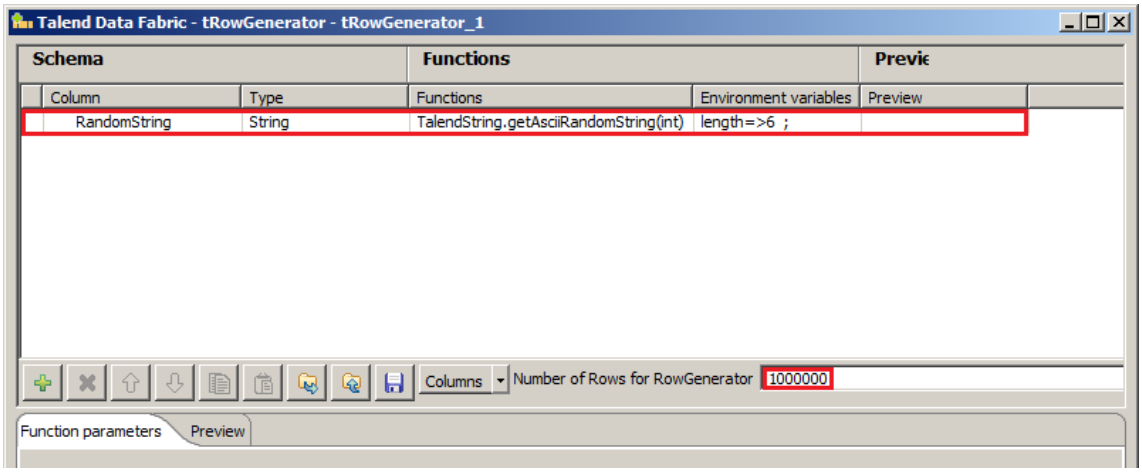
Create the Job

1. CREATE A NEW JOB

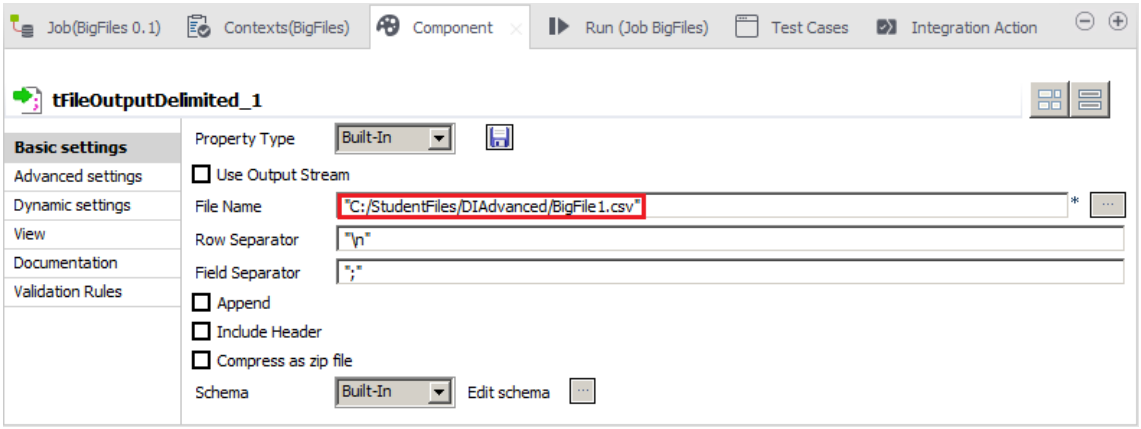
   Create a new standard Job named *BigFiles*.

2. ADD THE FIRST SUBJOB

   Add a **tRowGenerator** component and configure it to create 1,000,000 rows, each with a single column containing a randomly generated string.



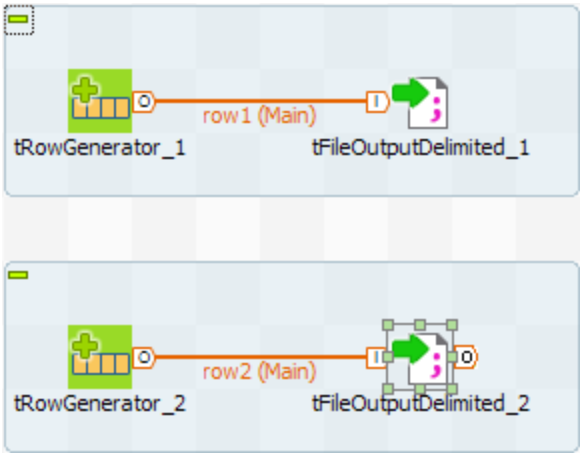
   Add a **tFileOutputDelimited** component. Configure it to write to the file *"C:/StudentFiles/DIAdvanced/BigFile1.csv"*.



Connect the two components with a **Main** row.

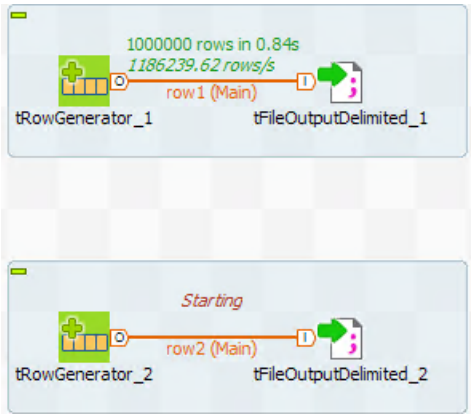
3. ADD A SECOND SUBJOB

   Paste a copy the entire subJob created in the previous step. Configure the **tFileOutputDelimited** component in the second subJob to write to a file named *BigFile2.csv*.



4. RUN THE JOB

   Run the Job. By watching the **Designer** closely while the Job executes, you can see that the first subJob executes first to entirety, followed by the second.



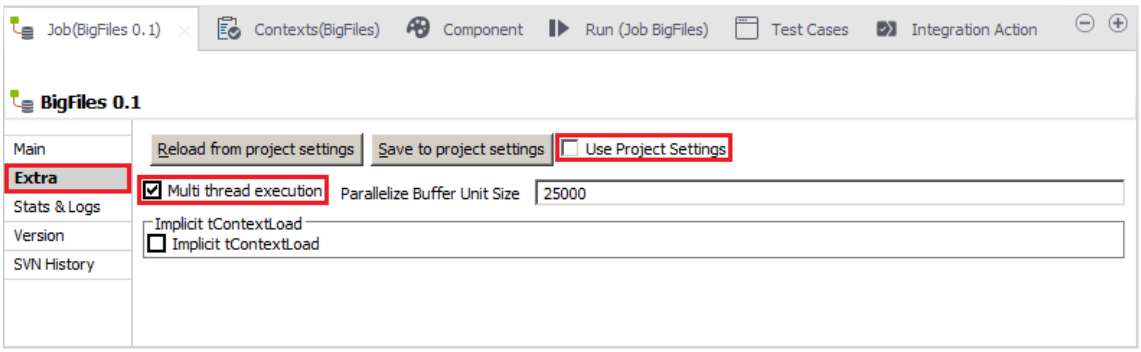
   By default, the subJobs execute in the order in which they were created.

Enable Multi-threaded Execution of the JobParallel

1. CONFIGURE SUBJOBS TO RUN IN PARALLEL

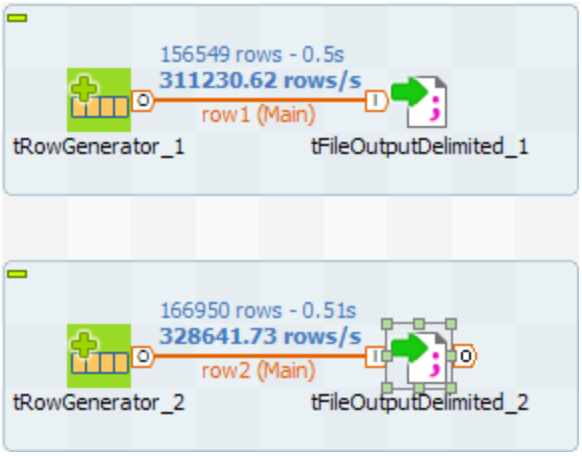
   Open the **Job** view. Click the **Extra** tab.

   Clear the **Use Project Settings** check box, and select the **Multi thread execution** check box.



2. RUN THE JOB

   Run the Job again. This time, you can see that both subJobs run simultaneously.



WARNING:
Enabling this feature on a single-core system may actually decrease performance.

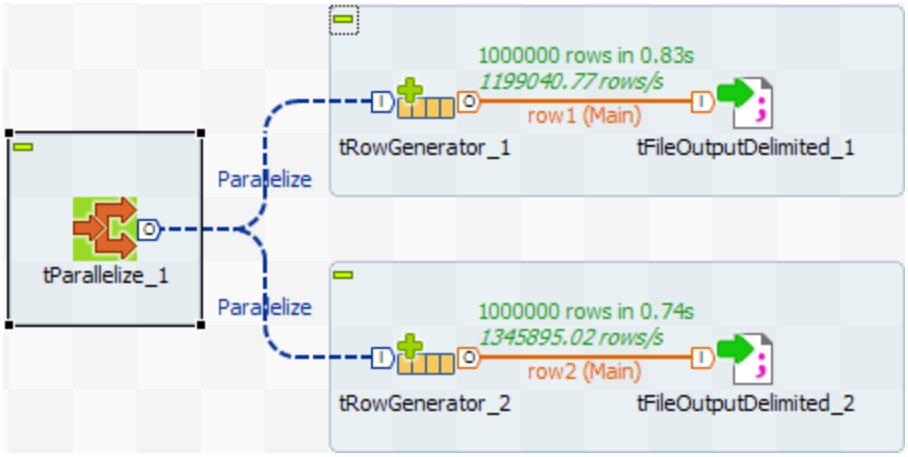
3. RESTORE THE PREVIOUS CONFIGURATION

   Open the **Job** view. Click the **Extra** tab. Clear the **Multi thread execution** check box in preparation for the next section. Run the Job to verify that the subJobs are executing sequentially as before.

Use the tParallelize Component

1. ADD A tParallelize COMPONENT

   Add a **tParallelize** component to the left of the two subJobs. Connect it to each subJob with a **Parallelize** trigger.



2. RUN THE JOB

   Run the Job. Again the subJobs run in parallel, but the difference this time is that the parallel execution is orchestrated by a component, rather than a Job-wide setting. The key advantage of this approach is that it gives you control over which parts of your Job execute in parallel. As an example, some subJobs can be run in parallel while other subJobs can be synchronized to run when other subJobs complete.

3. ADD A THIRD SUBJOB

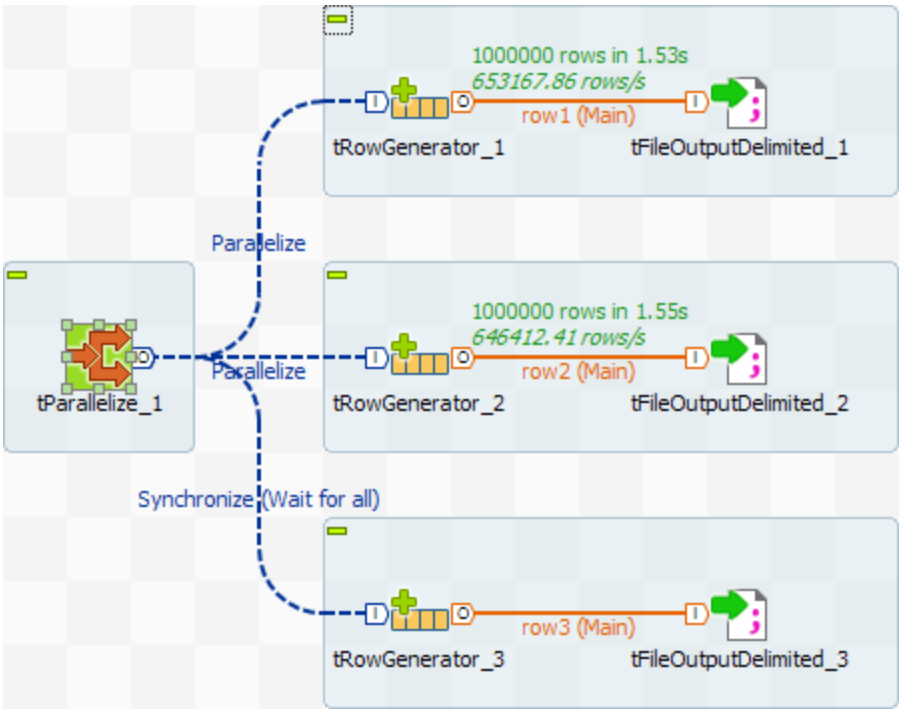
   Paste a copy and of the second subJob below the other two. Configure the **tFileOutputDelimited** component to write to a file named *BigFile3.csv*.

4. SYNCHRONIZE THE NEW SUBJOB

   Connect the **tParallelize** component to the **tRowGenerator** component of the third subjob with a **Synchronize** trigger. Now the first two subjobs are configured to run in at the same time and the third subjob is configured to run after the completion of the other two subjobs.

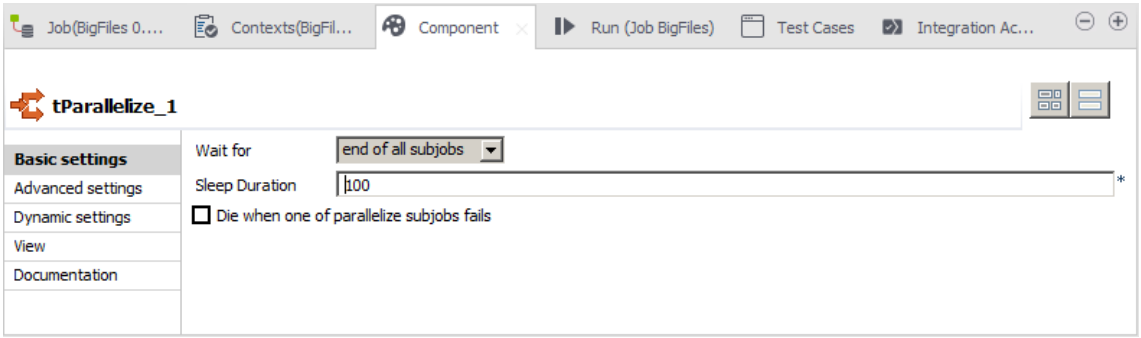
5. RUN THE JOB

   Run the Job again. Notice that no rows are processed for the third subJob until the first two subJobs complete executing, which they do in parallel.



6. EXPLORE tParallelize PROPERTIES

   Double-click the **tParallelize** component to open the **Component** view:



   Notice that you can specify:

   - whether the synchronization subJob executes after the first parallel subJob finishes, or after all subJobs complete (this is the default)
   - the polling interval
   - whether or not the entire Job should exit if one of the subJobs encounters an error

Now that you have seen how to perform parallel execution with subJobs writing to files, you can move on to apply parallel execution to working with databases.

Writing to Databases

Overview

In the previous exercise, you ran subJobs in parallel. Many Talend components can be configured to execute in parallel, particularly those that write to databases. For a complete list, see the *Talend Component Reference Guide*.

In this exercise, you will create a Job that writes a large number of rows to a database and then compare execution times without and then with parallel execution.

Create the Job

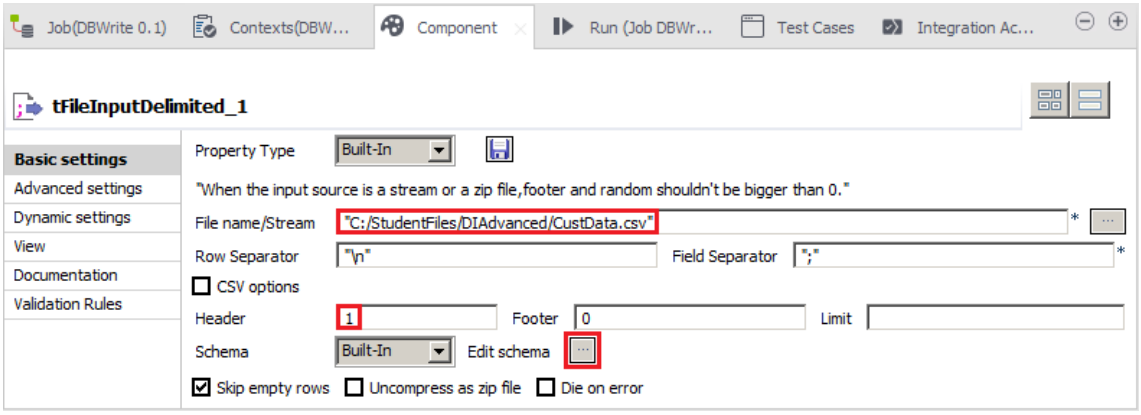
1. CREATE A NEW JOB

   Create a new standard Job named *DBWrite*.

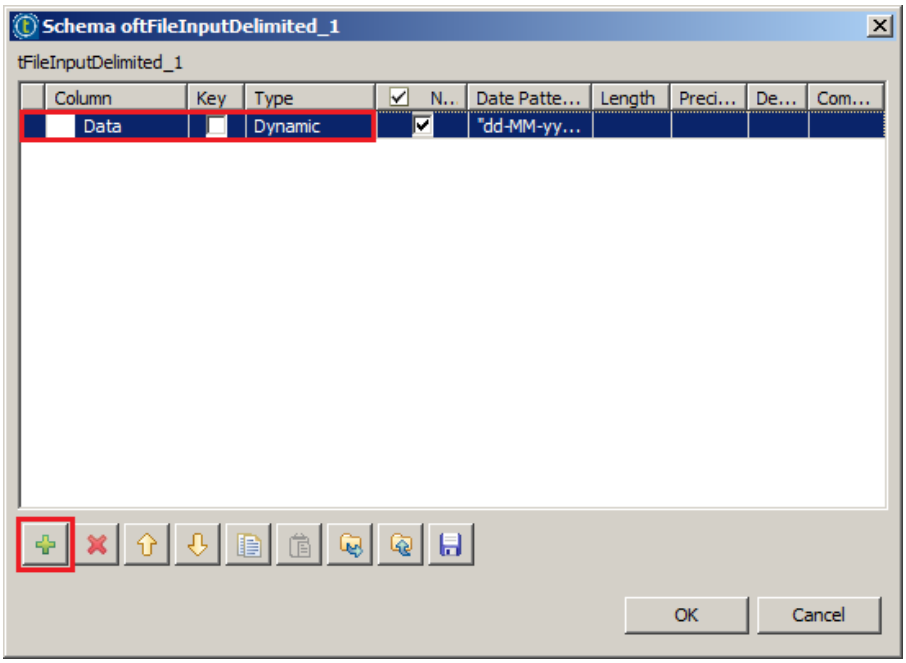
2. POPULATE THE JOB

   Add a **tFileInputDelimited** component to the design workspace. Configure it to read the file *"C:/StudentFiles/DIAdvanced/CustData.csv"*, and enter *1* into the **Header** field.

   Click the **Edit schema** button to configure a schema for the input file.



   Add a single column named *Data* and set the type to *Dynamic*. Click **OK** when done.

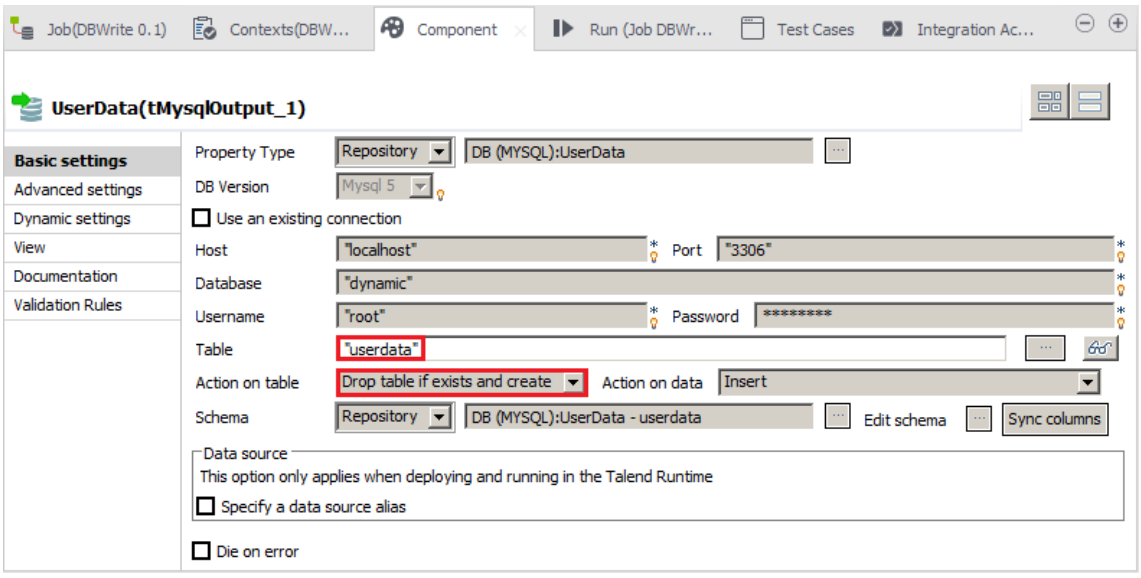


NOTE:
Dynamic is an opaque data type that allows data to be passed through without knowing the actual columns in the file or database. It will capture all columns not explicitly named.

Drag the connection metadata **Repository > Metadata > Db Connections > UserData** to the **Designer**, selecting **tMysqlOutput** as the component.

Connect the two components with a **Main** Row.

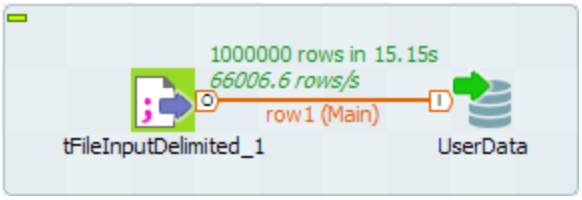
Configure the **tMysqlOutput** component, entering *userdata* into the **Table** box and selecting *Drop table if exists and create* for the **Action on table**.



3. RUN THE JOB

   Run the Job and then make a note of the total time the Job took to execute.

   On our test system, this Job took approximately 15.1 seconds to execute to completion.



Retrieve Schema

In this section, you are going to use a database inspection tool called **MySQL Workbench** to examine database schemas.

1. OPEN MYSQL WORKBENCH

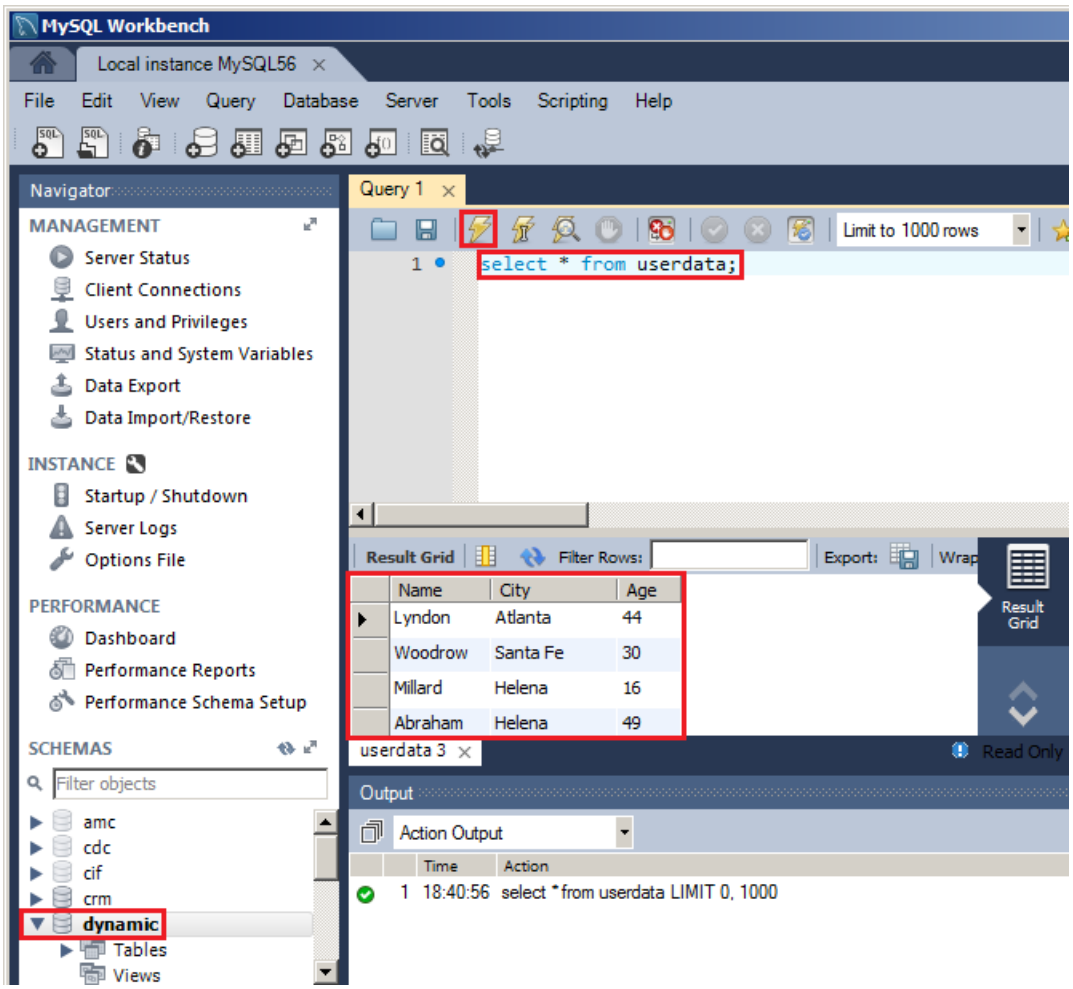
   From the Windows **Start** menu, select **All Programs > MySQL > MySQL Workbench 6.3 CE**. The welcome window appears briefly, and then the **MySQL Workbench** window appears. Double-click **Local instance MySQL56** to open the connection.

   TIP:
   If prompted for authentication, enter *root* for both **User** and **Password**.

2. EXAMINE A SCHEMA USING MYSQL WORKBENCH

   In the **MySQL Workbench** window, double-click **SCHEMAS > dynamic** to select the database to query.

   Enter the statement *select * from userdata;* into the **Query** box and click the **Execute** icon ( ) to run it.



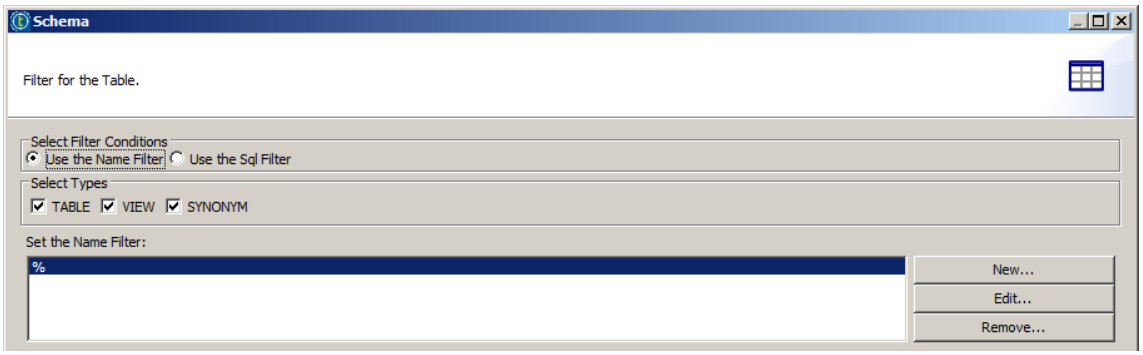
Notice in the output below that the database table has three columns: *Name*, *City*, and *Age*. The table is constructed correctly from the column definitions in the input file even though you did not specify the schema anywhere in your Job.

3. LOAD THE SCHEMA IN STUDIO

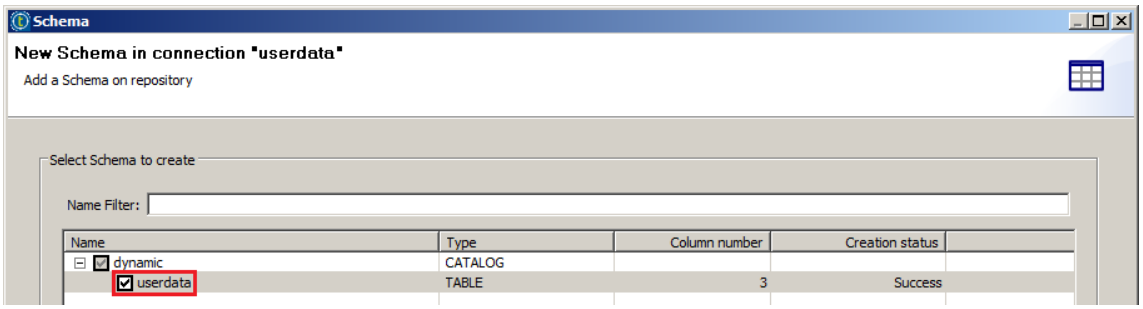
   Although the connection metadata specifies connection information and a database name (that is, *userdata*), there is no metadata in the repository describing the table format. You can verify this by expanding **Repository > Metadata > Db Connections > UserData**. Notice that the **Table schemas** folder is empty.

   Load the schema by right-clicking **Repository >Metadata > Db Connections > UserData** and selecting **Retrieve Schema**.

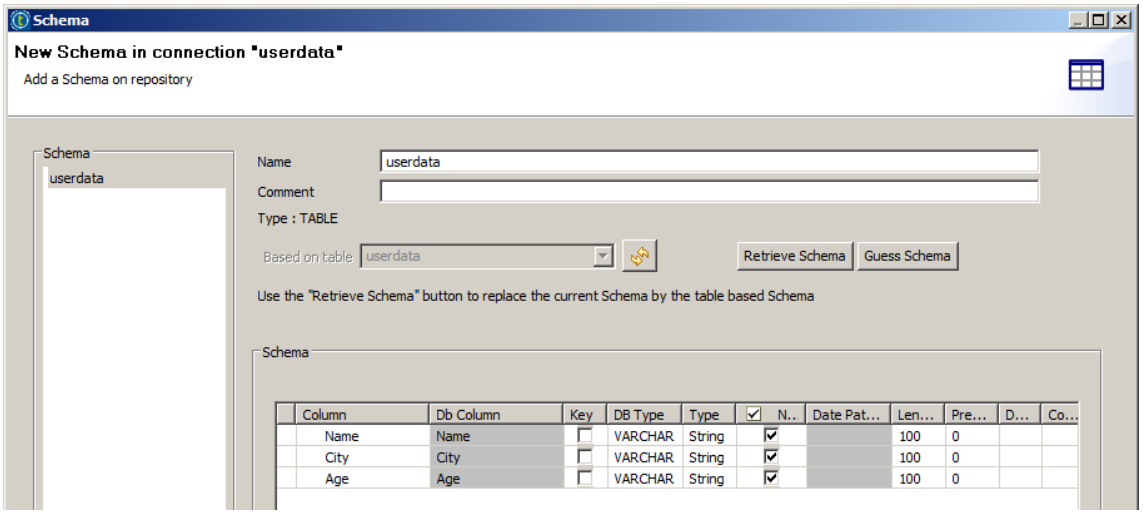
   The **Schema** window provides the opportunity to configure which aspects of the database structure to retrieve. Leave the default settings and click **Next**.



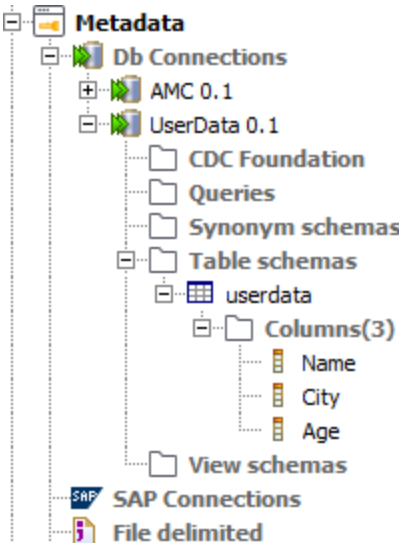
The next screen allows you to select the table on which to model the schema. Select **Dynamic > userdata** and click **Next**.



The final screen displays the resulting schema, which you can modify if desired. In this instance, there is no need, so click **Finish**.



Back in the **Repository**, notice that **Userdata > Table Schemas** is now populated.

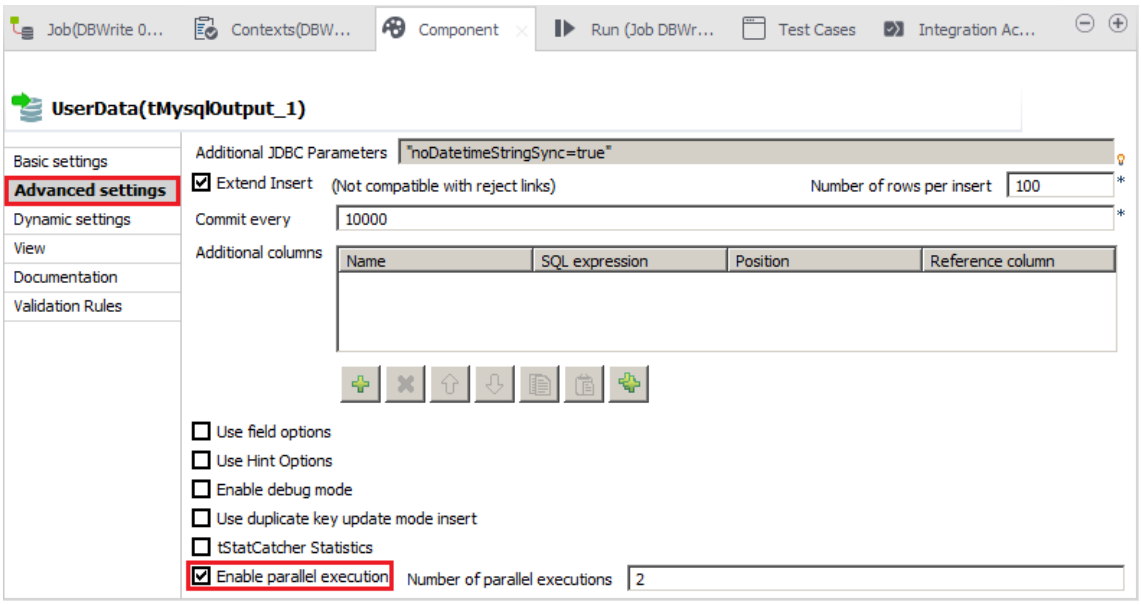


Configure Parallel Execution

1. ENABLE PARALLEL EXECUTION ON tMysqlOutput

   Double-click the **tMysqlOutput** component to open the **Component** view.

   In the **Component** view, click the **Advanced settings** tab and select **Enable parallel execution**.



   Notice that the component above is now overlaid with an icon (

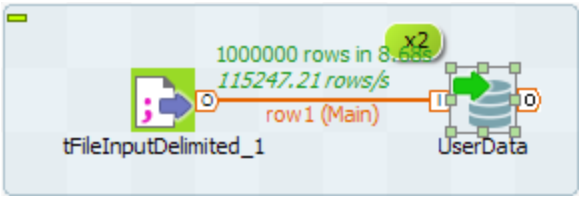
) that signifies that it is configured to execute using multiple threads.

2. RUN THE JOB

   Run the Job again, noting the new execution time.

   The amount of time drops significantly due to the parallel execution. You should see the execution time cut roughly in half by

using the parallel execution feature on the output database component.



Next you will use the automatic parallelization feature available in the Studio.

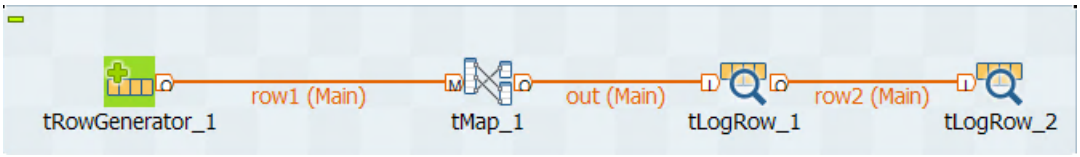
Automatic Parallelization

Overview

This exercise illustrates the automation of parallelization. Talend recommends automatic parallelization over the usage of dedicated components to achieve parallelization, which is discussed in the next section..

When you have to develop a Job to process a huge amount of data, you can enable or disable parallelization with a single click, and the Studio automates the implementation across the Job.

At the end of the Lab, your Job will look similar to this:



Generate data

You will create a Job that generated data. Then, you will enrich the data with a **tMap** component and display the rows with two **tLogRow** components.

1. CREATE A JOB

   Create a new standard Job named *AutoParallel*.

2. ADD AND CONFIGURE A tRowGenerator COMPONENT

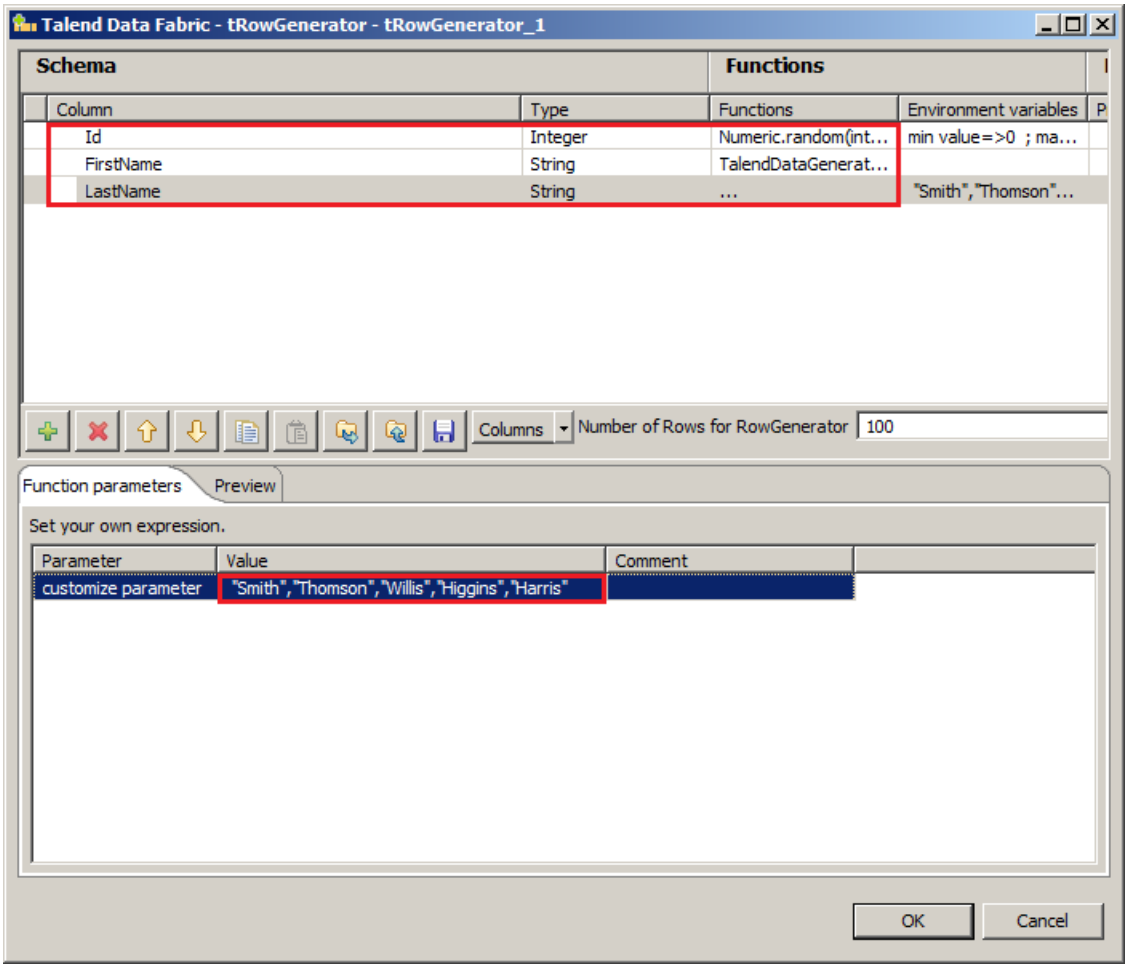
   Add a **tRowGenerator** component. Double-click the component to configure it.

   Using the **Add** button (), add three columns as follows:

   - *Id* of type *Integer*, using the function `Numeric.sequence(String,int,int)` to generate values (leave parameters at their default values)
   - *FirstName* of type *String*, using `TalendDataGenerator.getFirstName()` to generate values
   - *LastName* of type *String*. For the **Function**, specify ..., and enter the the following text into the **Value** box below: `"Smith","Thomson","Willis","Higgins","Harris"`

   The first name will be chosen randomly, while the last name value will be chosen from the list of predefined names.

When done, your configuration should looks as follows. Click **OK**.



Next, you will enrich the data with a **tMap** component.

Enrich and display data

You will now add a component that enriches the data with the thread ID, a variable which identifies a particular thread of execution. This will help you to understand how data is partitioned between the different threads.

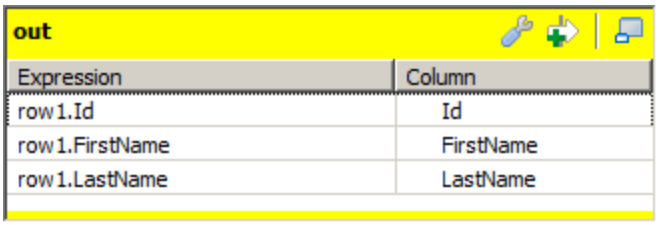
To display the data, you will use two **tLogRow** components. The first one will display the data processed by each thread and the last one will show the data after merging the threads.

1. ADD AND CONFIGURE A tMap COMPONENT

   Add a **tMap** component on the right side of the **tRowGenerator**. Connect them with a **Main** row.

   Double-click the **tMap** component to open to configure it.

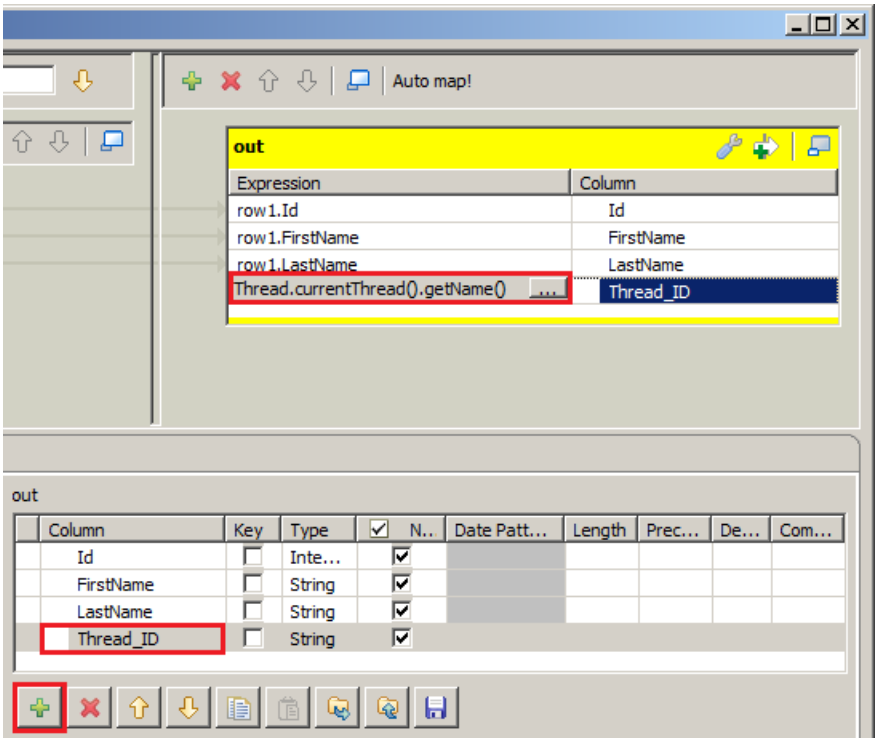
   Create a new output named *out*. Drag the **Id**, **FirstName**, and **LastName** columns to the **out** table.



2. ENRICH THE DATA

   Still in the map editor, use the **Add** button (+) below to add a new column named *Thread_ID* to the **out** table.

   Enter `Thread.currentThread().getName()` into the expression box for the new column. When done, the configuration should look as follows. Click **Ok** when done.

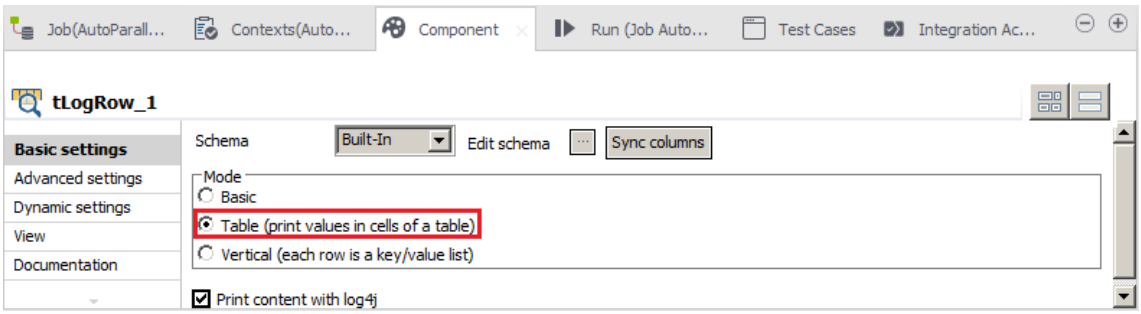


3. ADD AND CONFIGURE TWO tLogRow COMPONENTS

   Add a **tLogRow** component to the right of the **tMap** component and connect them with the **out** row.

   Add a second **tLogRow** component and connect it to the first one with a **Main** row.

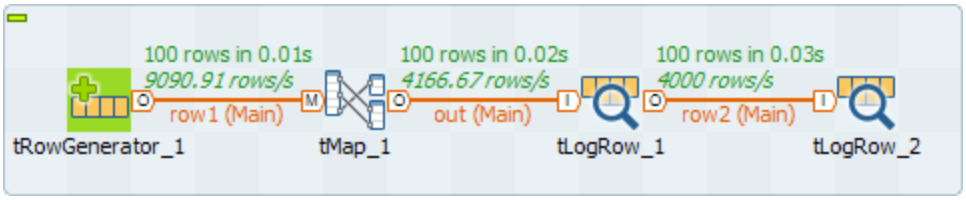
   Set both **tLogRow** components to **Table Mode**.



First Run

1. RUN THE JOB

   Run the Job and observe the statistics in the **Designer**.



   As expected, 100 rows are processed.

   Now examine the output in the console:

```
|=--+----------+--------+---------=|
|Id |FirstName |LastName|Thread_ID|
|=--+----------+--------+---------=|
|1  |Richard   |Harris  |main     |
|2  |Benjamin  |Higgins |main     |
|3  |Rutherford|Willis  |main     |
|4  |Thomas    |Smith   |main     |
|5  |John      |Willis  |main     |
|6  |George    |Higgins |main     |
|7  |Benjamin  |Willis  |main     |
|8  |Warren    |Thomson |main     |
|9  |Benjamin  |Harris  |main     |
|10 |Ulysses   |Thomson |main     |
|11 |James     |Willis  |main     |
```

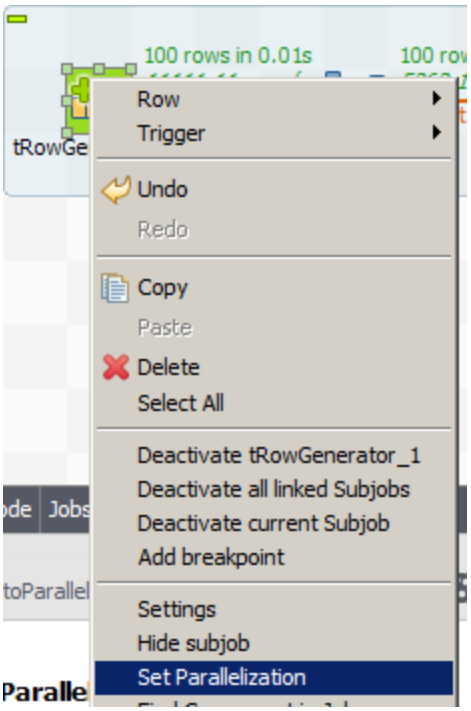
   As the parallelization is not enabled yet, the **Thread_ID** is *main* for all rows, and both **tLogRow** components display the same stream of data. Due to the random nature of the data, your specific output will vary slightly.

Enable Parallelization

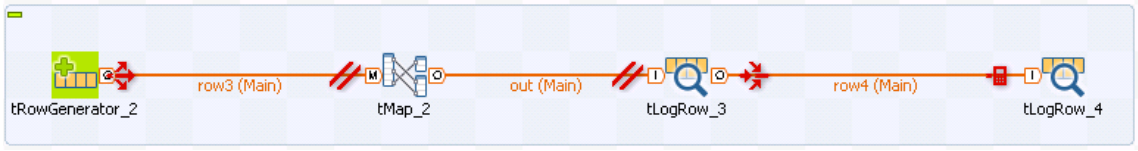
Now you will enable the parallelization and execute the Job with the default parallelization parameters.

1. ENABLE PARALLELIZATION

   Right-click the **tRowGenerator** component and select **Set Parallelization**.



This will enable parallelization in your Job.



Notice that your Job is annotated with the following icons:

- Partitioning ( ) splits the data into the specified number of threads.
- Collecting ( ) collects the split threads and sends them to a given component.
- Departitioning ( ) groups the outputs of the split threads.
- Recollecting ( ) captures the grouped results and outputs them to a given component.

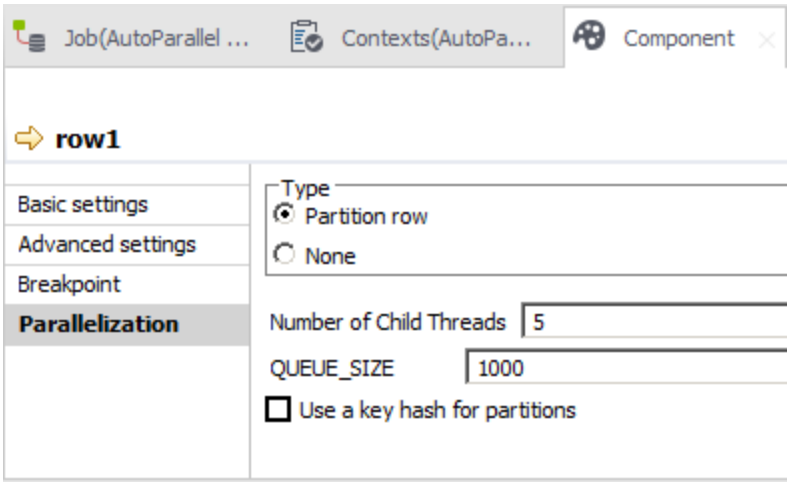
2. RUN THE JOB

Run the Job and observe the output in the console. Notice some changes in the output:

- For the first **tLogRow** component, there are now five smaller output tables
- For the second **tLogRow** component, there is still a single full table that resembles the following.

```
|Id  |FirstName  |LastName|Thread_ID       |
|=---+-----------+--------+---------------=|
|1   |Millard    |Willis  |pool-1-thread-1|
|2   |Thomas     |Higgins |pool-1-thread-2|
|3   |Calvin     |Higgins |pool-1-thread-3|
|4   |Warren     |Harris  |pool-1-thread-4|
|5   |James      |Harris  |pool-1-thread-5|
|6   |Ulysses    |Smith   |pool-1-thread-1|
|7   |Chester    |Thomson |pool-1-thread-2|
|8   |Gerald     |Willis  |pool-1-thread-3|
|9   |Gerald     |Willis  |pool-1-thread-4|
|10  |Herbert    |Willis  |pool-1-thread-5|
|11  |Bill       |Higgins |pool-1-thread-1|
```

These results are explained by the default values used for the parallelization. Examine these default values by opening the **Component** view for the output row of the **tRowGenerator** component, then clicking the **Parallelization** tab.



Notice that **Number of Child Threads** is set to five, which explains the five output tables for the first **tLogRow** component.

Also, be aware that rows are dispatched to threads in a round-robin fashion by default. This explains why rows are assigned the way they appear in the output.

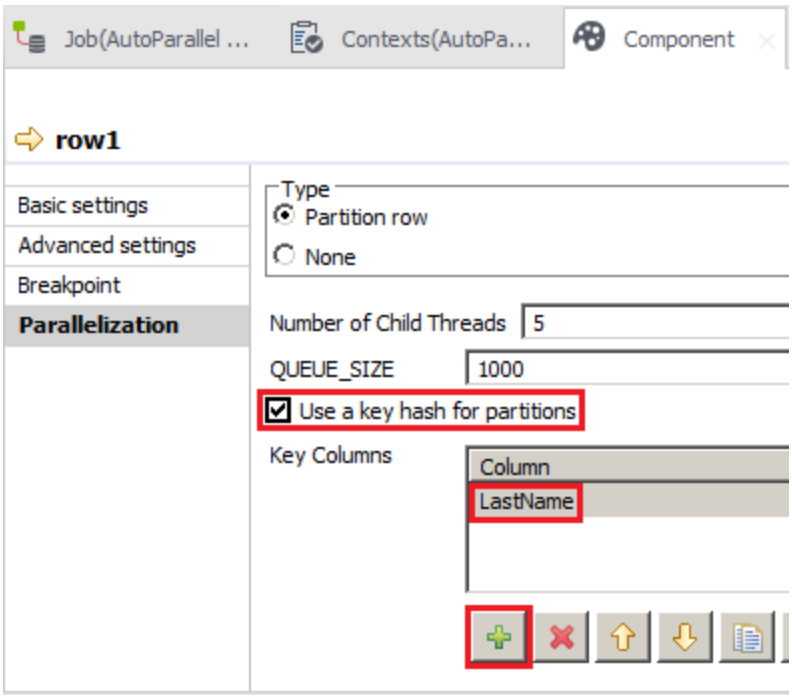
WARNING:
The value for **Number of Child Threads** should not exceed the number of cores on the system. In the next section, you will configure this value to avoid this performance-degrading condition.

Change Parallelization Settings

1. CONFIGURE THE PARTITIONING

   Still in the **Parallelization** tab, select the **Use a key hash for partitions** check box. This option configures the partitioning so that it will be based on values from specific columns. All records meeting the same defined criteria are dispatched to the same thread.

   Click the **Add** button (+) to add a key column, then select **LastName** for the column value.



Recall that the last names are chosen from a predefined list of five names. Using this criteria should have the rows dispatched to the five threads depending on the **LastName** value.

2. RUN THE JOB

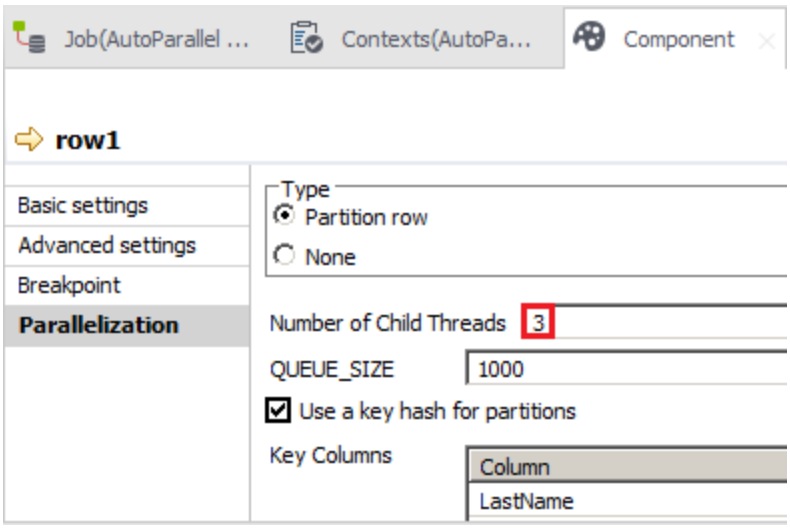
   Run the Job and observe the output in the console. Notice the change in the output:

   - For the first **tLogRow** component, there are still five output tables, but each table contains only entries with a specific value for *LastName*
   - For the second **tLogRow** component, there is still a single full table that resembles the following, but note that the thread allocations now match the *LastName* value.

```
|=--+----------+--------+--------------=|
|Id |FirstName |LastName|Thread_ID      |
|=--+----------+--------+--------------=|
|1  |Theodore  |Thomson |pool-1-thread-1|
|6  |Calvin    |Harris  |pool-1-thread-2|
|3  |Thomas    |Willis  |pool-1-thread-3|
|14 |Woodrow   |Smith   |pool-1-thread-4|
|18 |Chester   |Smith   |pool-1-thread-4|
|21 |Ulysses   |Smith   |pool-1-thread-4|
|2  |William   |Thomson |pool-1-thread-1|
|5  |Calvin    |Willis  |pool-1-thread-3|
|23 |William   |Smith   |pool-1-thread-4|
|16 |Calvin    |Higgins |pool-1-thread-5|
|4  |Rutherford|Thomson |pool-1-thread-1|
|15 |Martin    |Harris  |pool-1-thread-2|
|7  |James     |Willis  |pool-1-thread-3|
```

3. CONFIGURE THE NUMBER OF THREADS

   Back in the **Parallelization** tab of the **Component** view, set **Number of Child Threads** to *3*.



4. RUN THE JOB

   Run the Job and observe the output in the console. Notice the change in the output:

   - For the first **tLogRow** component, there are now only three output tables. The output is not cleanly split across the tables by *LastName* as it was before, because there are five possibilities and only three tables across which to spread them.
   - For the second **tLogRow** component, there is still a single full table that resembles the following, and the output represents the unclean division of names across the three child threads.

```
|=--+----------+--------+--------------=|
|Id |FirstName |LastName|Thread_ID      |
|=--+----------+--------+--------------=|
|1  |Woodrow   |Higgins |pool-1-thread-1|
|4  |Calvin    |Willis  |pool-1-thread-2|
|2  |Chester   |Thomson |pool-1-thread-3|
|3  |Warren    |Smith   |pool-1-thread-1|
|22 |Martin    |Willis  |pool-1-thread-2|
|7  |Grover    |Harris  |pool-1-thread-3|
|5  |Dwight    |Smith   |pool-1-thread-1|
|26 |Benjamin  |Willis  |pool-1-thread-2|
|9  |Benjamin  |Harris  |pool-1-thread-3|
|6  |Warren    |Smith   |pool-1-thread-1|
|29 |Lyndon    |Willis  |pool-1-thread-2|
```

Disable Parallelization

Disabling parallelization is as easy as enabling it.

1. DISABLE PARALLELIZATION

   Right-click the **tRowGenerator** component and select **Disable Parallelization**.

   This will remove the parallelization icons.

2. RUN THE JOB

   Run the Job and check the console to verify that parallelization is disabled.

```
|=--+----------+--------+--------=|
|Id |FirstName |LastName|Thread_ID|
|=--+----------+--------+--------=|
|1  |Richard   |Harris  |main     |
|2  |Benjamin  |Higgins |main     |
|3  |Rutherford|Willis  |main     |
|4  |Thomas    |Smith   |main     |
|5  |John      |Willis  |main     |
|6  |George    |Higgins |main     |
|7  |Benjamin  |Willis  |main     |
|8  |Warren    |Thomson |main     |
|9  |Benjamin  |Harris  |main     |
|10 |Ulysses   |Thomson |main     |
|11 |James     |Willis  |main     |
```

Only one output table for the first **tLogRow**component is displayed with a **Thread_ID** of *main*. The same output is logged by the second **tLogRow**component as well.

Time permitting, you can perform the optional Partitioning lab or proceed to the Wrap-Up.

Partitioning

Overview

NOTE:
This section is considered optional. Go through the material if time permits.

Another way to parallelize executions is to use dedicated components such as **tPartitioner** and **tCollector**. You will build a Job that generates a configurable amount of data, sort the data, and write the sorted result to a file.

Generate Data

First, you will generate 1000000 lines of random data using the **tRowGenerator** component.

1. CREATE A JOB

   Create a new standard Job named *JobPartitioner*.

2. ADD AND CONFIGURE A tRowGenerator COMPONENT

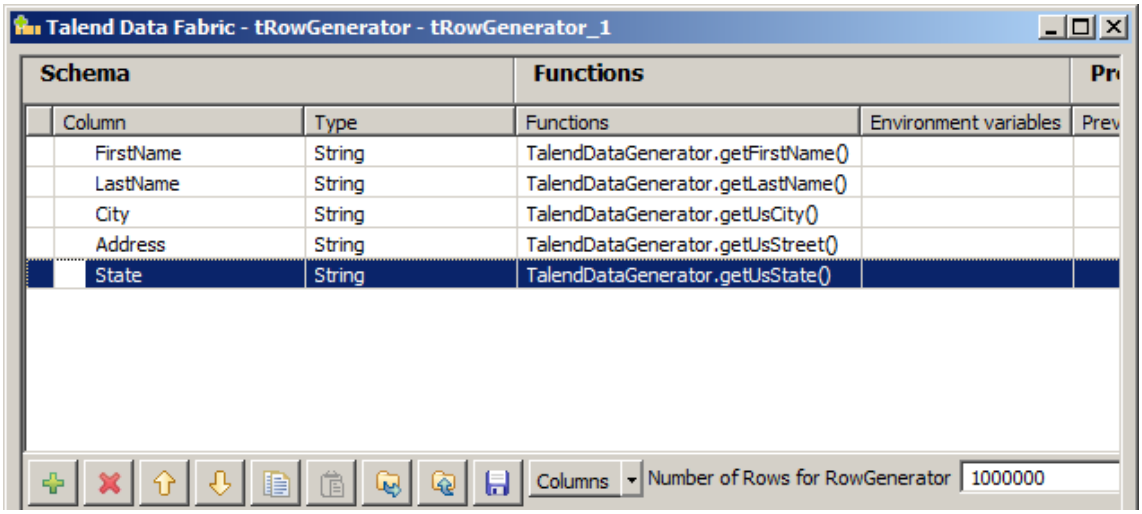
   Add a **tRowGenerator** component. Double-click the component to configure it.

   Using the **Add** button (), add five columns of type *String*. Use the following **Function** specifications to generate appropriate random data for each column:

   - *FirstName* using `TalendDataGenerator.getFirstName()`
   - *LastName* using `TalendDataGenerator.getLastName()`
   - *City* using `TalendDataGenerator.getUsCity()`
   - *Address* using `TalendDataGenerator.getUsStreet()`
   - *State* using `TalendDataGenerator.getUsState()`

   Finally, enter *1000000* in the **Number of Rows for RowGenerator** box.

   The final configuration should look like the figure below. Click **OK** when done.



Partition and collect data

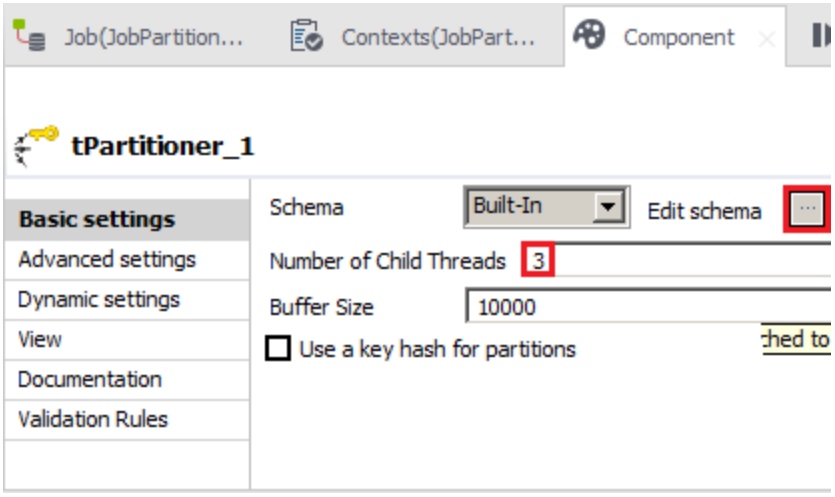
The **tPartitioner** component dispatches input records into a specific number of threads. You will use this component to start the parallel execution in your Job, in place of the built-in multithreading functionality of **tRowGenerator** you used in the previous section.

The **tCollector** component sends threads to the components that follow it for parallel execution. You will use one of these next to a **tPartitioner** to handle the threads created by **tPartitioner**.

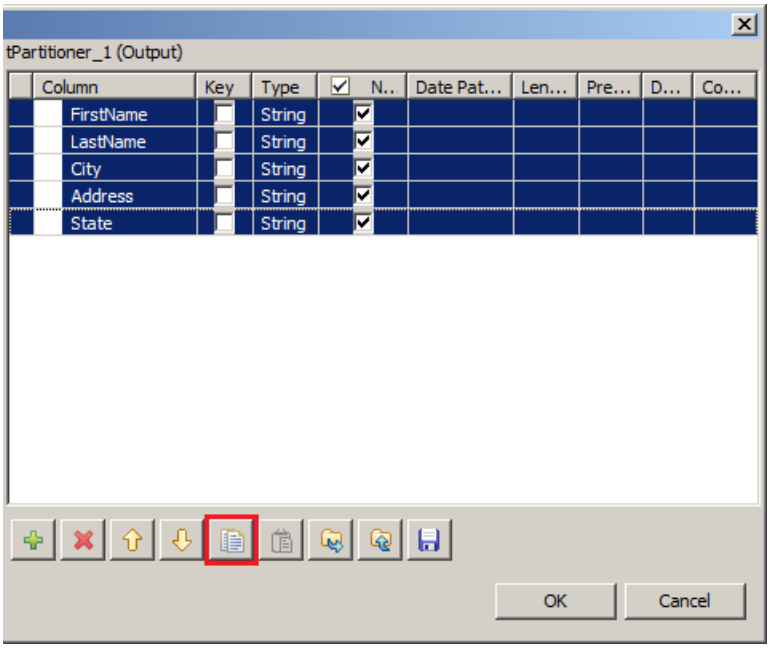
1. ADD AND CONFIGURE A tPartitioner COMPONENT

   Add a **tPartitioner** component to the right side of **tRowGenerator** and connect it with a **Main** row. Then, double-click the **tPartitioner** to open the **Component** view.

   In the Component view, enter *3* into the **Number of Child Threads** box. Then, click the ellipsis button (**...**) to edit the schema.



   In the **Schema of tPartitioner_1** window, select the five output columns and click the **Copy selected items** button () to copy the selected items to the clipboard.

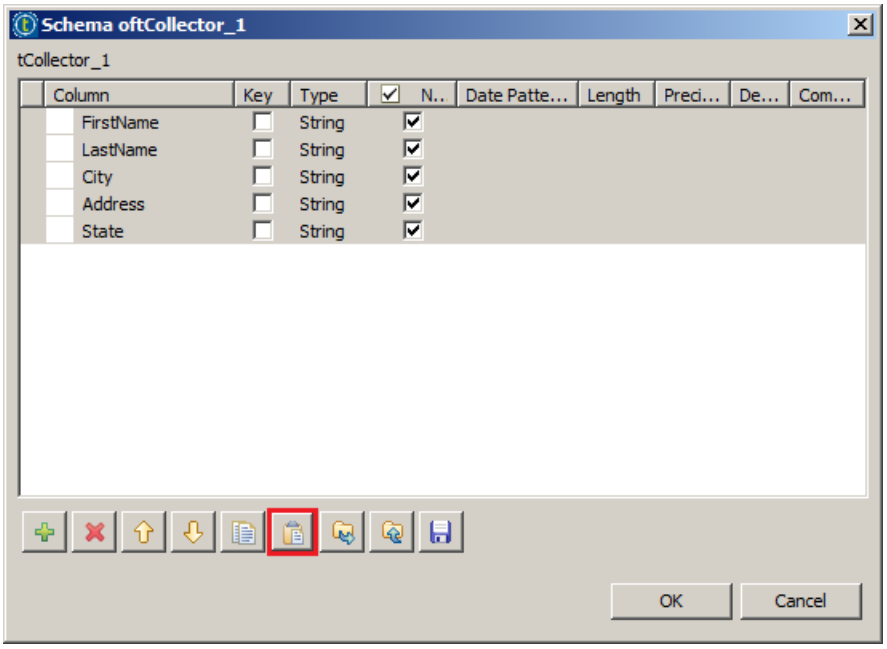


   Click **OK** to close the window. The schema just copied to the clipboard will be used shortly.

2. ADD AND CONFIGURE A tCollector COMPONENT

Add a **tCollector** component to the right side of **tPartitioner** and connect it with a **Starts** trigger. Then, double-click the **tCollector** to open the **Component** view. In the Component view, click the ellipsis button (**...**) to edit the schema.

In the Schema of tCollector_1 window, click the **Paste** button ( ) to paste the schema from the clipboard.



Click **OK** to close the schema window.

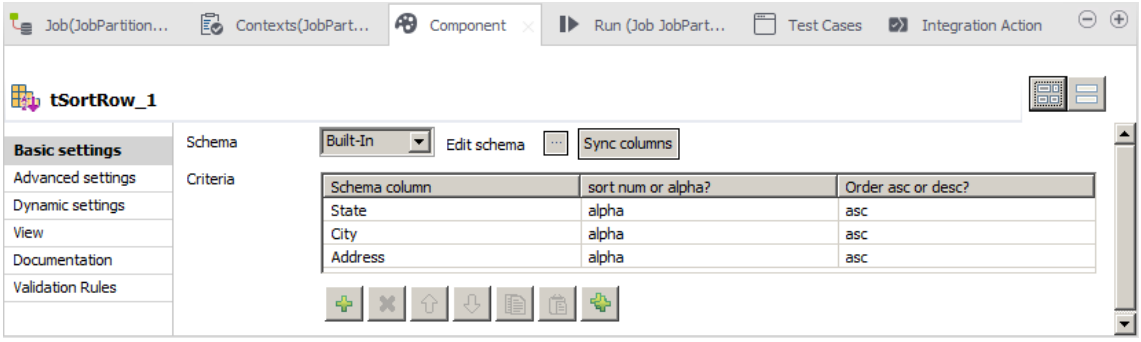
Sort Data

The data will be sorted according to the State, City, and Address columns in alphabetic order. You will use the **tSortRow** component to achieve this. As a huge amount of data will be sorted, you will use advanced settings of **tSortRow** to sort on disk and to allow temporary data to be saved.

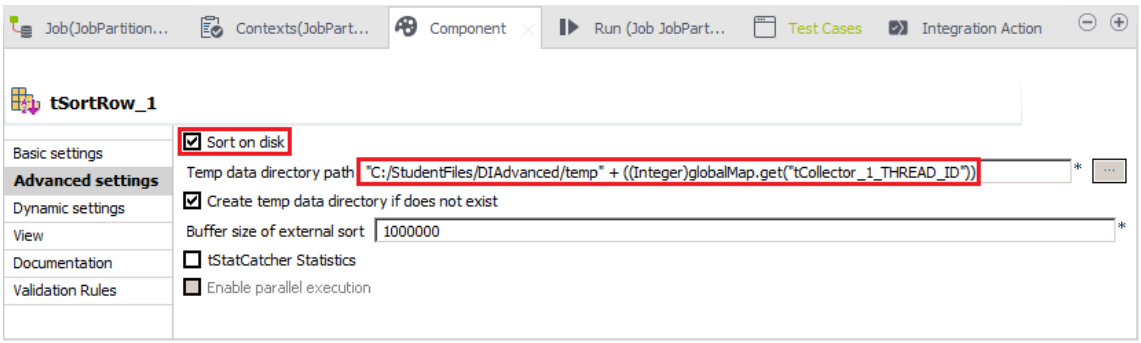
1. ADD AND CONFIGURE A tSortRow COMPONENT

   Add a **tSortRow** component to the right side of **tCollector** and connect it with a **Main** row. Then, double-click the **tSortRow** to open the **Component** view.

   Use the **Add** button to add three columns to the **Criteria** table. Configure them to sort based on *State*, *City*, and *Address* in ascending alphabetic order.



Next, click the **Advanced settings** tab. Select the **Sort on disk** check box. In the **Temp data directory path** box that appears, enter *"C:/StudentFiles/DIAdvanced/temp"* +. Press **Ctrl + Space**. From the list, double-click the entry that reads **tCollector_1_THREAD_ID**.



NOTE:
Setting the temporary data directory path here is important, because multiple threads will be accessing temporary data. A unique path for each thread avoids the possibility overwriting data of another thread.

This is done by building a path that includes a context variable containing the thread id.

Finalize Job

Although the data will be sorted, it is still split in multiple threads. You will have to collect them before writing the output to a file. You will use **tDepartitioner** and **tRecollector**components to finalize your Job.

The **tDepartitioner** component regroups the outputs of the processed parallel threads, and the **tRecollector** component captures the output of a **tDepartitioner** component and sends data to the next component.

1. ADD A tDepartitioner COMPONENT

   Add a **tDepartitioner** component to the right side of **tSortRow** and connect it with a **Main** row. Then, double-click the **tDepartitioner** to open the **Component** view. Use the ellipsis button (**...**) to verify that the schema has been updated: you should see the **FirstName**, **LastName**, **City**, **Address**, and **State** columns for both the input and output.

2. ADD AND CONFIGURE A tRecollector COMPONENT

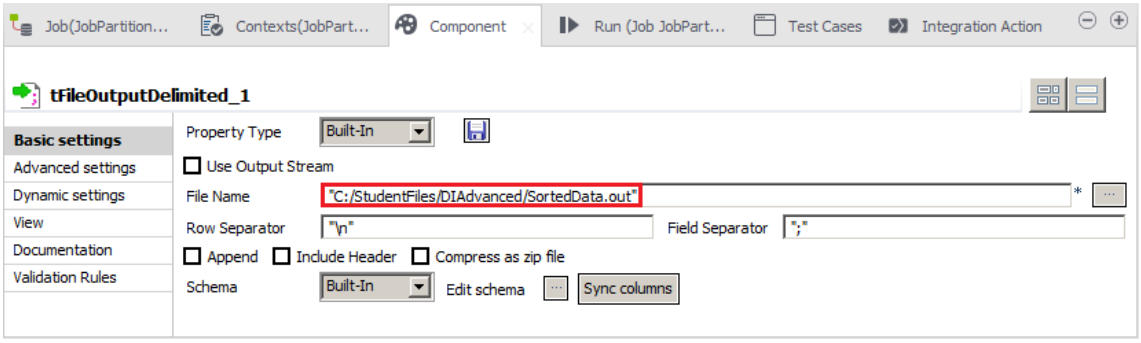
   Add a **tRecollector** component to the right side of **tDepartitioner** and connect it with a **Starts** trigger. Then, double-click the **tRecollector** to open the **Component** view.

   Use the ellipsis button (**...**) to edit the schema. Use the Paste button ( ) to paste the schema that was copied to the clipboard earlier, then click **OK**.

3. ADD AND CONFIGURE AN OUTPUT COMPONENT

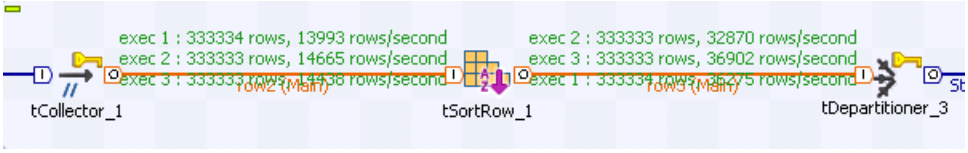
   Add a **tFileOutputDelimited** component to the right side of **tRecollector** and connect it with a **Main** row. Then, double-click the **tFileOutputDelimited** to open the **Component** view.

   In the **File Name** box, enter *"C:/StudentFiles/DIAdvanced/SortedData.out"*.



4. RUN THE JOB

Run the Job and observe the statistics in the **Designer** Workspace. As the Job was built to use three threads, you will see the number of rows processed on each thread. You should see the work distributed evenly across each thread.



WARNING:
Never use a number of threads that is higher than the number of available processors. This can lead to a degradation in performance and you will lose the advantage of multi-threading.

You have finished the last section in this lesson so it's time to Wrap-Up.

Wrap-Up

In this lesson, you set multi thread execution on an individual Job in order to run subJobs in parallel. Then you used the **tParallelize** component to run subJobs in parallel after which another subjob ran. You also enabled parallel execution for an individual database output component. Most input and output database components can be configured to take advantage of parallel execution.

You used the **Set Parallelization** feature to automate parallel processing for an existing Job.

You also may have investigated the usage of components dedicated to partitioning and collecting data to achieve multi-threaded processing. Essentially, this workflow is a manual implementation of the **Set Parallelization** feature and is, in fact, a predecessor to this functionality. It tends to be more difficult to set up and configure, but does provide backwards compatibility. For new projects, this approach is not recommended.

You can use any or all of these methods to take full advantage of the processing resources of a multi-core system to speed execution of your Jobs.

Next step

Congratulations! You have successfully completed this lesson. To save your progress, click **Check your status with this unit** below. To go to the next lesson, on the next screen, click **Completed. Let's continue >**.

LESSON 7

Joblets

This chapter discusses the following.

Introduction 118
Joblets 119
Creating a Joblet from an Existing Job 120
Creating a Joblet from Scratch 126
Triggering Joblets 129
Wrap Up 133

Introduction

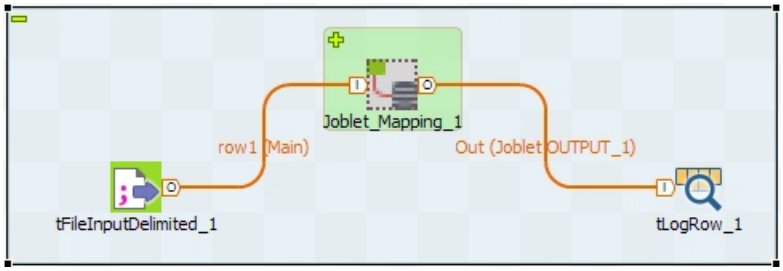
Joblets

Overview

A Joblet is a specific component that replaces Job component groups. It factorizes recurrent processing or complex transformation steps to ease the reading of a complex Job. Joblets can be reused in different Jobs or several times in the same Job.

Available Joblets appear in the Repository under the Joblet Designs section.

Unlike the **tRunJob** component, Joblet code is integrated in the Job code itself. This way, the Joblet does not impact the performance of your Job. In fact, Joblet performance is exactly the same as the original Job while using less overall resources.



As a final note, Joblet have access to the same context variables as the Job itself.

Objectives

After completing this tutorial, you will be able to:

- Create a Joblet from scratch
- Create a Joblet from an existing Job
- Create a Joblet that allows triggered executions
- Use a Joblet in a Job

The first step is to open the Job you will use for this Lab.

Creating a Joblet from an Existing Job

Overview

This lab will show you how to create a Joblet from an existing Job.

You will use a basic Job that reads data from two different files and joins them before displaying the result in the console:

- The first file contains customer information: first and last name, and detailed address information
- The second file maps state codes to a full state name

You will first run the Job to see how it works, then you will refactor a part of this Job as a Joblet.

Run the Original Job

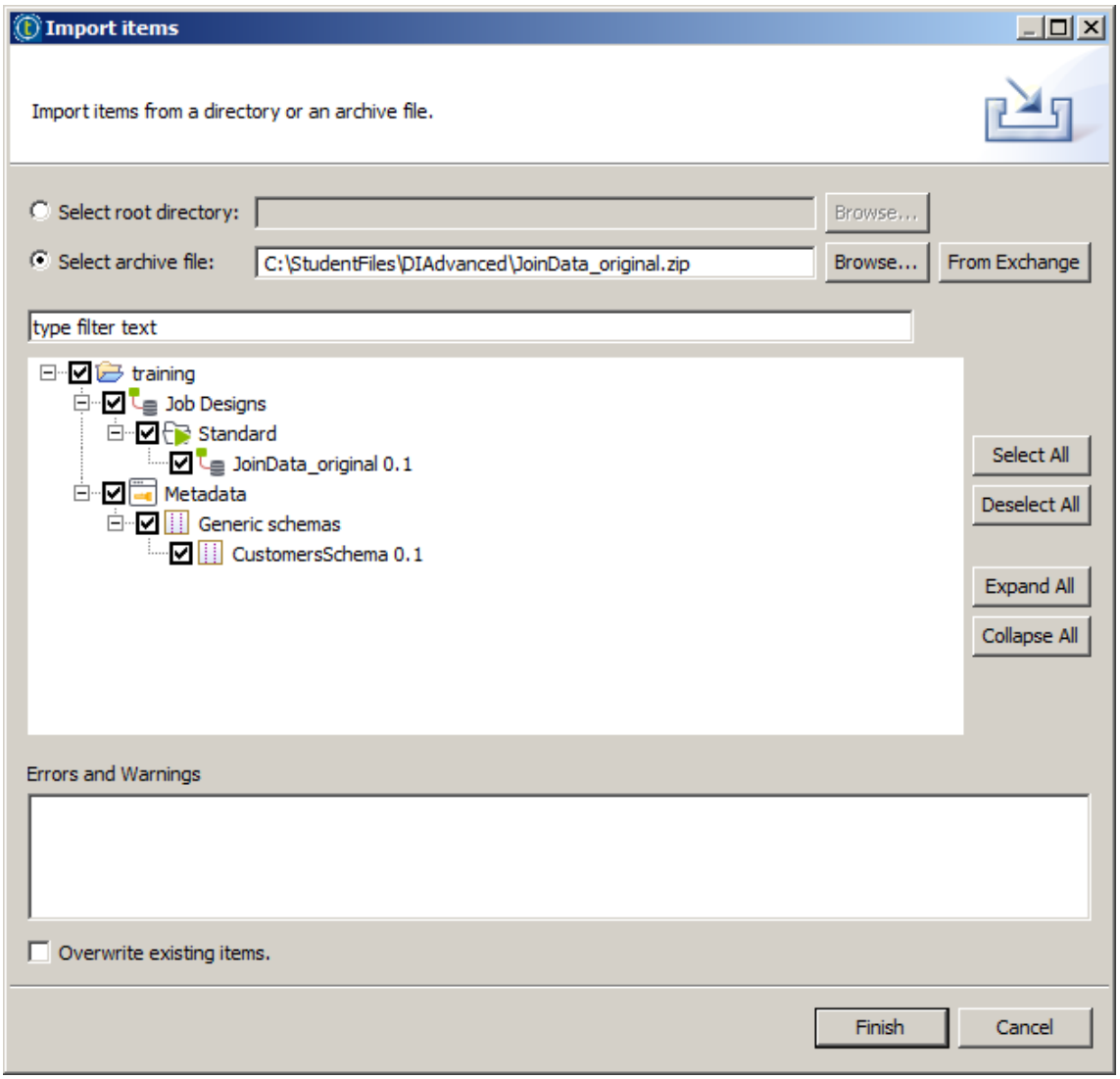
First, you will import the Job from an archive file and run it.

1. IMPORT THE JOB

   Right-click **Repository > Job Designs > Standard** and select **Import Items**.

   Choose **Select archive file**, and specify the path *C:\StudentFiles\DIAdvanced\JoinData_original.zip*.

   Click **Select All** to import the Job and its dependencies, then click **Finish**.

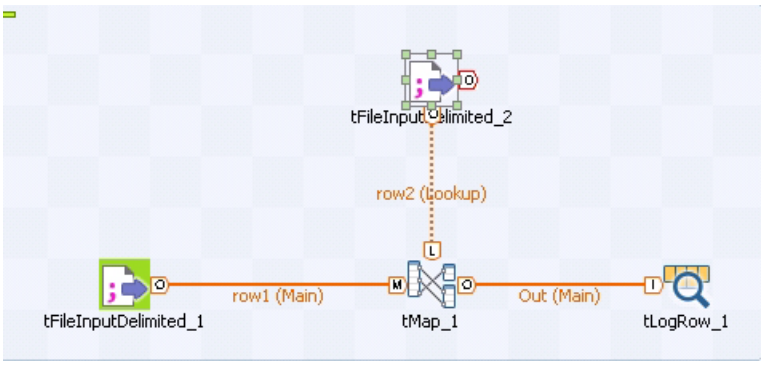


The Job **JoinData_original** appears in the **Repository**.

2. DUPLICATE THE JOB

Right-click the **JoinData_original** Job and select **Duplicate**. Name the new Job *JoinData*.

Close any other Jobs to eliminate confusion, then open the **JoinData** Job by double-clicking on it in the **Repository**.



3. RUN THE JOB

   Run the Job and examine the results in the console.

```
|-------------------+----------------------------------------+-----------|
|Name               |Address                                 |StateName  |
|===================+========================================+===========|
|Bill Coolidge      |85013 Via Real Austin                   | Illinois  |
|Thomas Coolidge    |63489 Lindbergh Blvd Springfield        | California|
|Harry Ford         |97249 Monroe Street Salt Lake City      | California|
|Warren McKinley    |82589 Westside Freeway Concord          | Alaska    |
|Andrew Taylor      |29886 Padre Boulevard Madison           | California|
|Ulysses Coolidge   |98646 Bayshore Freeway Columbus         | Minnesota |
|Theodore Clinton   |12292 San Marcos Bismarck               | New York  |
|Benjamin Jefferson |82077 Carpinteria North Sacramento      | California|
|William Van Buren  |21712 Tully Road East Albany            | Illinois  |
|Calvin Washington  |50742 Richmond Hill Charleston          | California|
|Jimmy Polk         |76143 Richmond Hill Salt Lake City      | Alaska    |
|Calvin Adams       |52386 Lake Tahoe Blvd. Montgomery       | New York  |
|Ulysses Monroe     |70511 Jones Road Trenton                | Illinois  |
|Zachary Tyler      |45040 Santa Rosa North Carson City      | Alaska    |
|Ulysses Johnson    |19989 Via Real Juneau                   | Alabama   |
|George Arthur      |89874 Calle Real Annapolis              | Alabama   |
|George Jefferson   |67703 Fontaine Road Pierre              | Illinois  |
|Herbert Grant      |90635 North Ventu Park Road Columbus    | Alaska    |
|Calvin Washington  |37446 E Fowler Avenue Pierre            | Alabama   |
|John Harrison      |86745 San Marcos Annapolis              | Alaska    |
|Benjamin Ford      |27921 Carpinteria Avenue Providence     | California|
|Andrew Fillmore    |96878 Greenwood Road Saint Paul         | Alaska    |
|Warren Arthur      |22920 Jean de la Fontaine Madison       | Minnesota |
|Calvin Pierce      |41962 Santa Rosa North Juneau           | California|
|Woodrow Clinton    |30587 San Diego Freeway Topeka          | California|
|George Jefferson   |95054 Padre Boulevard Denver            | Alaska    |
|Lyndon Eisenhower  |14379 Newbury Road Lincoln              | Alabama   |
|John Adams         |49993 El Camino Real Phoenix            | Alaska    |
```

The next step is to create a Joblet that corresponds to the mapping task in the Job.

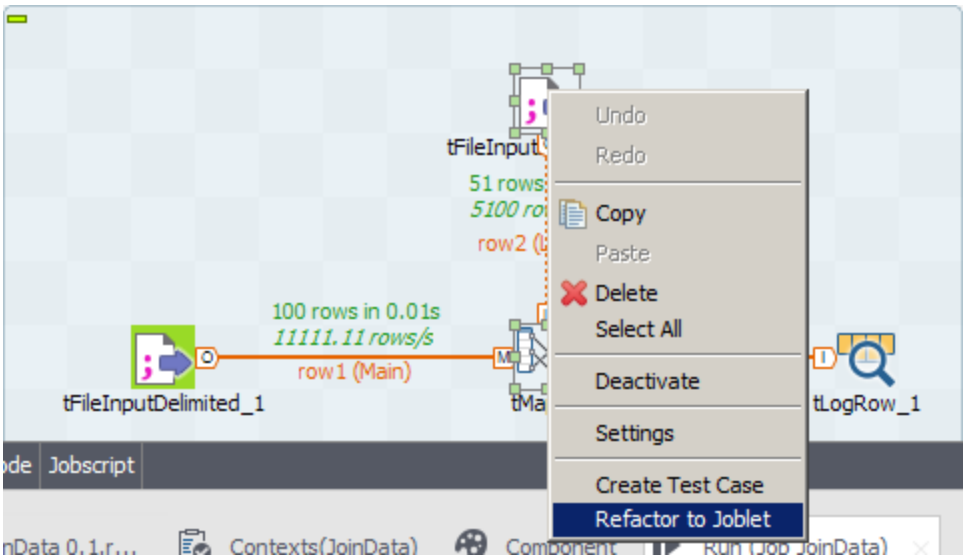
Refactor Mapping to Joblet

Creating a Joblet from an existing Job is a simple task that involves selecting the portion of the Job you want to refactor and then ask Studio for the refactoring.

The Joblet you are about to create will include two components: **tFileInputDelimited** and **tMap**.

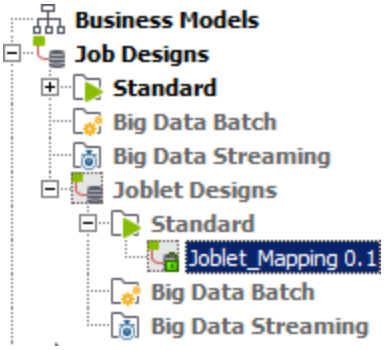
1. REFACTOR COMPONENTS

   Select both the **tMap** and **tFileInputDelimited_2** components by clicking them with the **Ctrl** key pressed. Right-click the selected components and then select **Refactor to Joblet**.

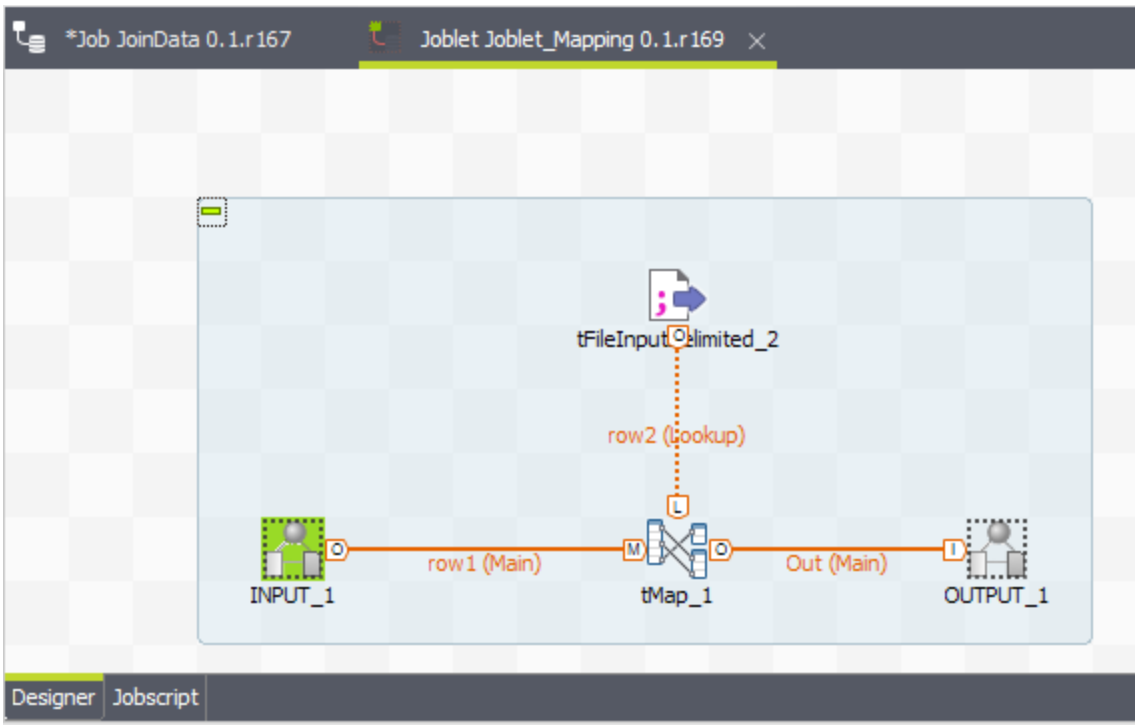


In the New Joblet window that appears, enter *Joblet_Mapping* in the **Name** field, then provide a **Purpose** and a **Description**.

Click **Finish** to save the Joblet. It appears under **Repository > Joblet Designs > Standard**.

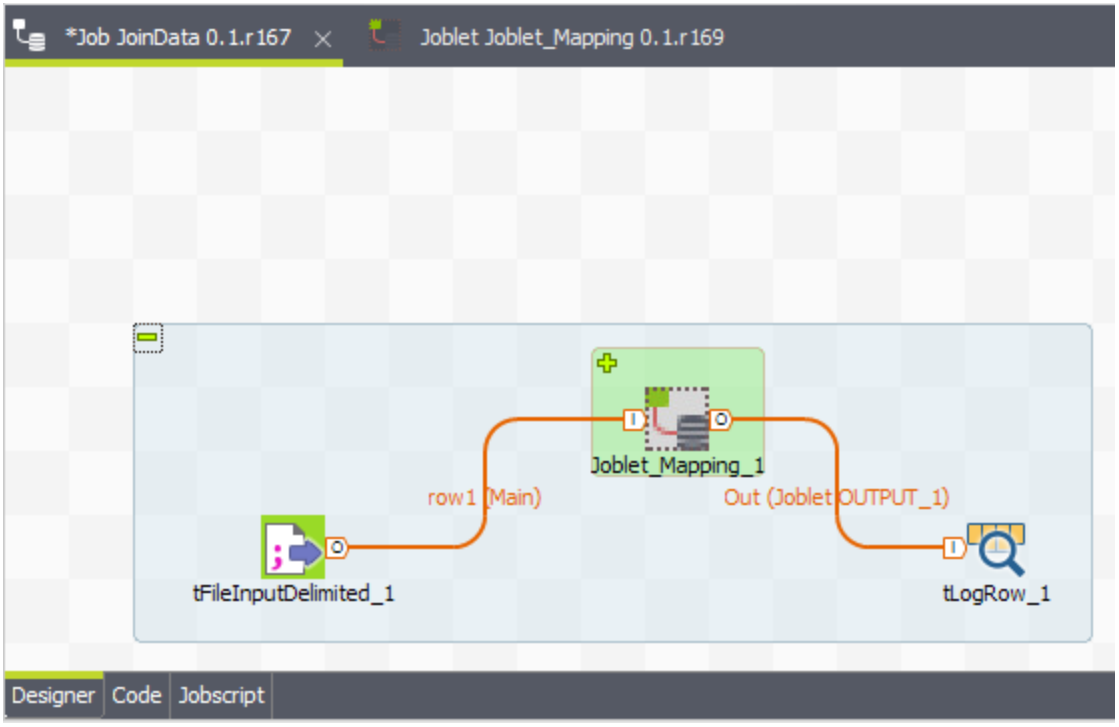


Notice that the Joblet is also opened in the **Designer**. Note the **INPUT_1** and **OUTPUT_1** components in the design. These components are needed to have input and output connections to your Joblet.

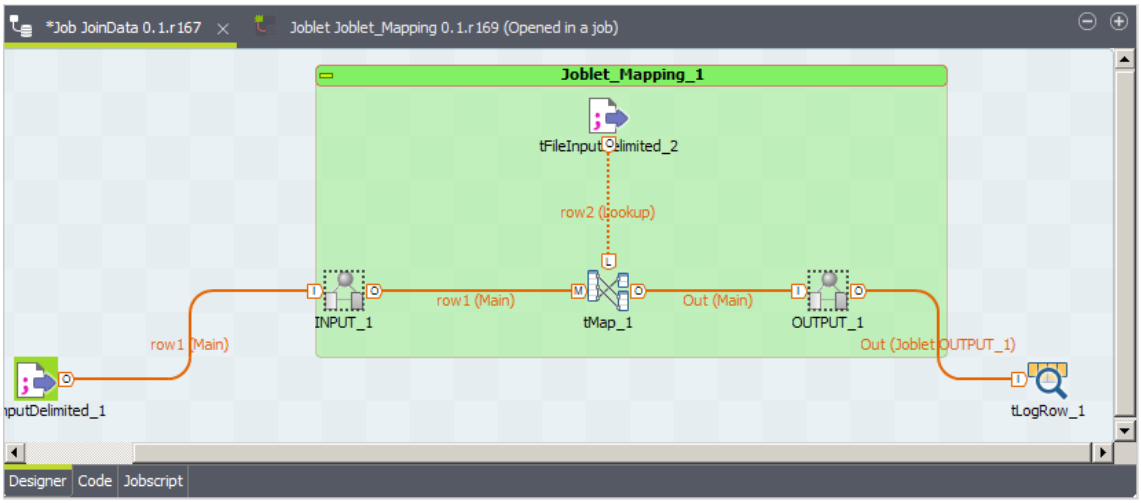


2. EXAMINE THE JOB

Turn your attention back to the **JoinData** Job. The **tMap** and **tFileInputDelimited_2** components have been replaced by the **Joblet_Mapping** component. This is the Joblet you just created.



Expand the Joblet by clicking the icon (✚) in the top-left corner of the Joblet.



3. RUN THE JOB

   Run the Job and examine the results in the console.

```
|-------------------+-----------------------------------------+-----------|
|Name               |Address                                  |StateName  |
|===================+=========================================+===========|
|Bill Coolidge      |85013 Via Real Austin                    | Illinois  |
|Thomas Coolidge    |63489 Lindbergh Blvd Springfield         | California|
|Harry Ford         |97249 Monroe Street Salt Lake City       | California|
|Warren McKinley    |82589 Westside Freeway Concord           | Alaska    |
|Andrew Taylor      |29886 Padre Boulevard Madison            | California|
|Ulysses Coolidge   |98646 Bayshore Freeway Columbus          | Minnesota |
|Theodore Clinton   |12292 San Marcos Bismarck                | New York  |
|Benjamin Jefferson |82077 Carpinteria North Sacramento       | California|
|William Van Buren  |21712 Tully Road East Albany             | Illinois  |
|Calvin Washington  |50742 Richmond Hill Charleston           | California|
|Jimmy Polk         |76143 Richmond Hill Salt Lake City       | Alaska    |
|Calvin Adams       |52386 Lake Tahoe Blvd. Montgomery        | New York  |
|Ulysses Monroe     |70511 Jones Road Trenton                 | Illinois  |
|Zachary Tyler      |45040 Santa Rosa North Carson City       | Alaska    |
|Ulysses Johnson    |19989 Via Real Juneau                    | Alabama   |
|George Arthur      |89874 Calle Real Annapolis               | Alabama   |
|George Jefferson   |67703 Fontaine Road Pierre               | Illinois  |
|Herbert Grant      |90635 North Ventu Park Road Columbus     | Alaska    |
|Calvin Washington  |37446 E Fowler Avenue Pierre             | Alabama   |
|John Harrison      |86745 San Marcos Annapolis               | Alaska    |
|Benjamin Ford      |27921 Carpinteria Avenue Providence      | California|
|Andrew Fillmore    |96878 Greenwood Road Saint Paul          | Alaska    |
|Warren Arthur      |22920 Jean de la Fontaine Madison        | Minnesota |
|Calvin Pierce      |41962 Santa Rosa North Juneau            | California|
|Woodrow Clinton    |30587 San Diego Freeway Topeka           | California|
|George Jefferson   |95054 Padre Boulevard Denver             | Alaska    |
|Lyndon Eisenhower  |14379 Newbury Road Lincoln               | Alabama   |
|John Adams         |49993 El Camino Real Phoenix             | Alaska    |
```

   You should have exactly the same result as before.

You just created a Joblet from an existing Job. In the next section you will create a Joblet from scratch and use it in the **JoinData** Job.

Creating a Joblet from Scratch

Overview

Similar to how you create new Jobs in Studio, it is also possible to create Joblets from scratch. The procedure is very similar to standard Job creation that you are already familiar with.

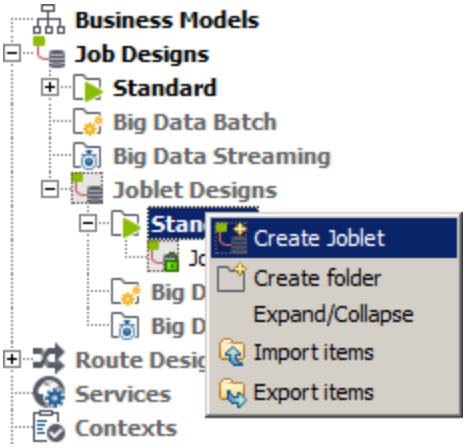
You will create a *startable* Joblet. That means that this Joblet can be used to start a Job, without any input link on the Joblet. The Joblet will read the Customers file. Later you will use this Joblet in the JoinData Job.

Creating a Joblet

In the Repository, there is a Joblet Designs folder. This is where you will create your Joblet.

1. CREATE THE JOBLET

   Right-click **Repository > Job Designs > Joblet Designs > Standard** and select **Create Joblet**.

   Name the Joblet *Joblet_InputFile*, then click **Finish**. The Joblet opens in the **Designer**.

   By default, an input and an output component are placed for you, so that your Joblet is easily connected to other components.

2. DELETE THE ORIGINAL INPUT COMPONENT

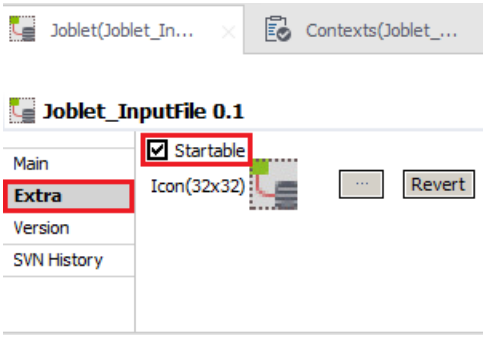
   Since you will add a new input component, the original input is not needed. Delete **INPUT_1**.

3. COPY AN INPUT COMPONENT FROM ANOTHER JOB

   Copy the **tFileInputDelimited_1** component from the **JoinData** Job and paste it to the left of **OUTPUT_1**. Connect it to the **OUTPUT_1** component with a **Main** row.

4. MAKE THE JOBLET STARTABLE

   Open the **Joblet** view and click the **Extra** tab. Ensure that the **Startable** check box is selected.



5. SAVE THE JOBLET

   Save the Joblet before continuing.

Next, you will use this Joblet in the JoinData Job.

Update the JoinData Job

You will update the Job with this new Joblet and then run it to check if it's still working correctly.

1. OPEN A JOB

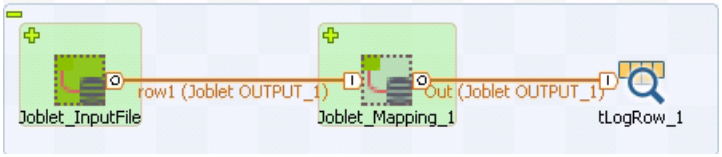
   Open the **JoinData** Job if it is not already open.

2. REPLACE THE INPUT COMPONENT WITH A JOBLET

   Delete the **tFileInputDelimited_1** component.

   Select **Repository > Job Designs > Joblet Designs > Standard > Joblet_InputFile** and drag it onto the **Designer**.

   Connect **Joblet_InputFile** to **Joblet_Mapping** with the **Joblet OUTPUT_1** row.



3. RUN THE JOB

   Run the Job and examine the results in the console.

Name	Address	StateName
Bill Coolidge	85013 Via Real Austin	Illinois
Thomas Coolidge	63489 Lindbergh Blvd Springfield	California
Harry Ford	97249 Monroe Street Salt Lake City	California
Warren McKinley	82589 Westside Freeway Concord	Alaska
Andrew Taylor	29886 Padre Boulevard Madison	California
Ulysses Coolidge	98646 Bayshore Freeway Columbus	Minnesota
Theodore Clinton	12292 San Marcos Bismarck	New York
Benjamin Jefferson	82077 Carpinteria North Sacramento	California
William Van Buren	21712 Tully Road East Albany	Illinois
Calvin Washington	50742 Richmond Hill Charleston	California
Jimmy Polk	76143 Richmond Hill Salt Lake City	Alaska
Calvin Adams	52386 Lake Tahoe Blvd. Montgomery	New York
Ulysses Monroe	70511 Jones Road Trenton	Illinois
Zachary Tyler	45040 Santa Rosa North Carson City	Alaska
Ulysses Johnson	19989 Via Real Juneau	Alabama
George Arthur	89874 Calle Real Annapolis	Alabama
George Jefferson	67703 Fontaine Road Pierre	Illinois
Herbert Grant	90635 North Ventu Park Road Columbus	Alaska
Calvin Washington	37446 E Fowler Avenue Pierre	Alabama
John Harrison	86745 San Marcos Annapolis	Alaska
Benjamin Ford	27921 Carpinteria Avenue Providence	California
Andrew Fillmore	96878 Greenwood Road Saint Paul	Alaska
Warren Arthur	22920 Jean de la Fontaine Madison	Minnesota
Calvin Pierce	41962 Santa Rosa North Juneau	California
Woodrow Clinton	30587 San Diego Freeway Topeka	California
George Jefferson	95054 Padre Boulevard Denver	Alaska
Lyndon Eisenhower	14379 Newbury Road Lincoln	Alabama
John Adams	49993 El Camino Real Phoenix	Alaska

You should have exactly the same result as before.

You have learned how to build a Joblet and how to make it startable.

The next step is to create a Joblet than can be triggered.

Triggering Joblets

Overview

It is possible to use a Joblet as a step in a Job. You can start the execution of a Joblet after the execution of a Subjob or start the execution of a Subjob after the execution of a Joblet.

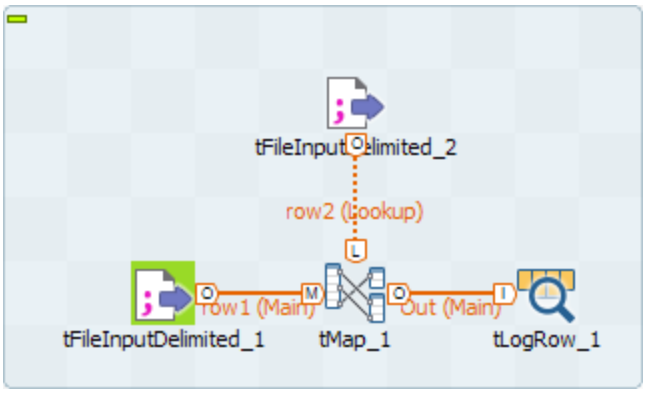
First you will create a new Joblet which allows using triggers to connect to it.

Create a Triggered Joblet

You will use components from the **JoinData_original** Job to create a Joblet and then add triggers to it.

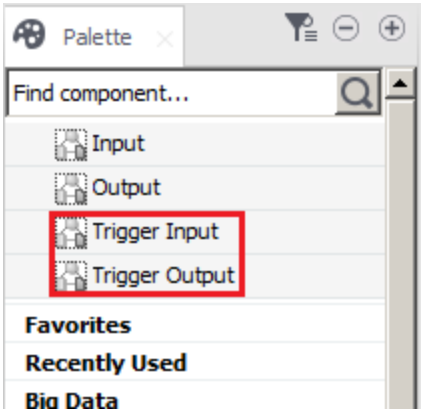
1. CREATE A JOBLET

   Create a new Joblet named *Joblet_trigger*. Copy the components from the **JoinData_original** Job and paste them into your new Joblet.



2. ADD TRIGGERS

   Turn your attention to the **Palette** view. In particular, notice the components at the top of the list, which are specific to Joblets.



   Input and **Output** are already familiar, but the **Trigger Input** and **Trigger Output** components are new. These components are used to trigger the Joblet on the input or output flow.

   Add a **Trigger Input** above **tFileInputDelimited_1**. Connect it to **tFileInputDelimited_1** with an **On Subjob Ok** trigger.

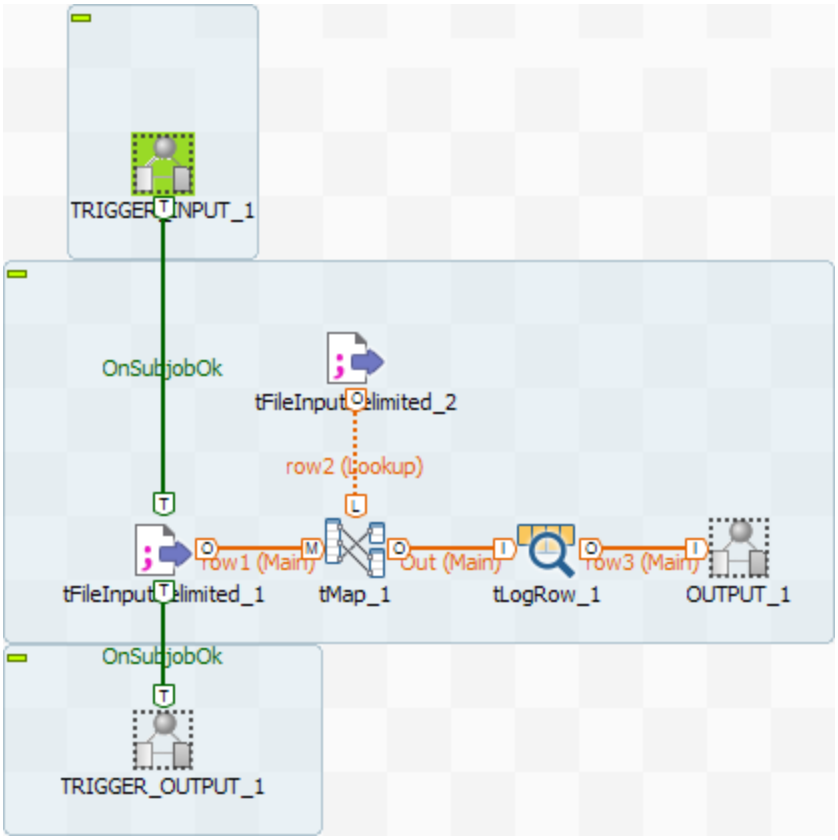
   Then, add a **Trigger Output** below **tFileInputDelimited_1**. Connect **tFileInputDelimited_1** to it using an **On Subjob Ok** trigger.

3. SET UP INPUT AND OUTPUT

Delete the default **INPUT_1** component as it is not needed.

Connect the **tLogRow** to **OUTPUT_1** with a **Main** row.

Your Joblet should resemble the following figure.



4. SAVE THE JOBLET

   Save the Joblet before continuing.

Create a Test Job

You will now test the Joblet you created by building a Job that generates an informational message, launches the Joblet, and generates another informational message.

1. CREATE A NEW JOB

   Create a new standard Job named *JoinData_trigger*.

2. ADD AND CONFIGURE A tMsgBox COMPONENT

   Add a **tMsgBox** component. Double-click it to open the **Component** view.

   In the **Message** box, enter *"Start"*.

3. ADD A JOBLET

   Drag the **Joblet_trigger** Joblet from the **Repository** and place it below the **tMsgBox** component. Connect the **tMsgBox** component to it with an **On Subjob Ok** trigger.

   Notice that there is a warning on the Joblet. Hover over the warning for and you will see that it is because no outputs are defined yet. This will be fixed shortly.

4. ADD AND CONFIGURE ANOTHER tMsgBox COMPONENT

Add a second **tMsgBox** component below the **Joblet_trigger** Joblet. Double-click it to open the **Component** view, and in the **Message** box, enter *"Processing Complete!"*.

Connect the Joblet to the **tMsgBox** component with an **On Subjob Ok** trigger.

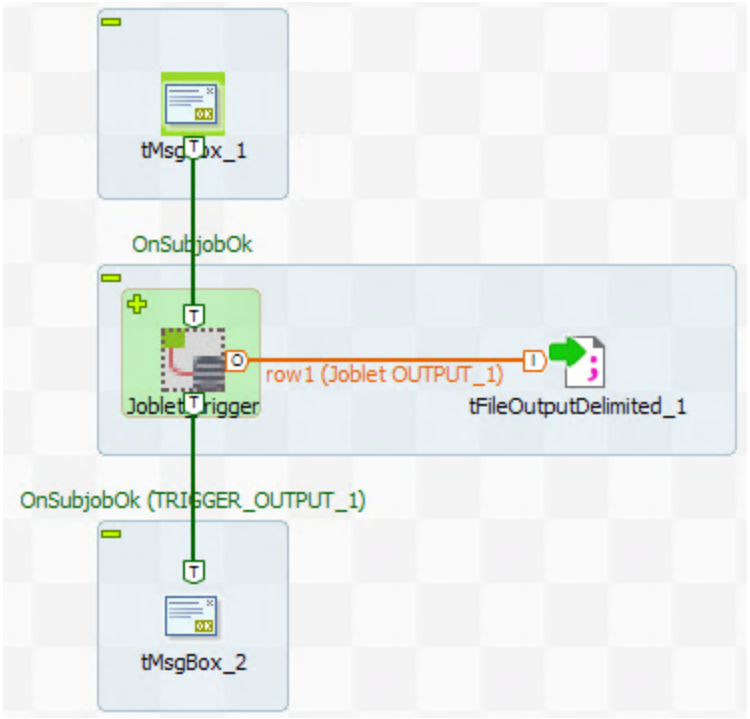
5. ADD AND CONFIGURE AN OUTPUT COMPONENT

   Add a **tFileOutputDelimited** component to the right of **Joblet_trigger**.

   Connect **Joblet_trigger** to it with the **Joblet OUTPUT_1** row.

   Open the **Component** view for the **tFileOutputDelimited** component and set the **File Name** to *"C:/StudentFiles/DIAdvanced/Customers_out.csv"*.

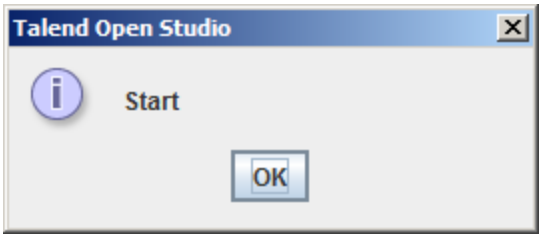
   At this point, your Job should resemble the following figure.



NOTE:
Notice that the Joblet is painted green. If a Joblet is *dirty* (that is, it has unsaved modifications), it will be painted in red within any Jobs that use it.

6. RUN THE JOB

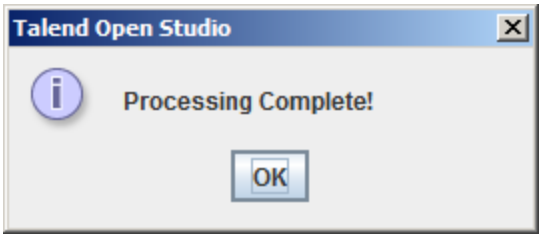
   Run the Job. The message from the first **tMsgBox** component appears. Click **OK** to allow the Joblet to proceed.



The Joblet then executes, displaying customer names and addresses in the console again, the same as before.

Name	Address	StateName
Bill Coolidge	85013 Via Real Austin	Illinois
Thomas Coolidge	63489 Lindbergh Blvd Springfield	California
Harry Ford	97249 Monroe Street Salt Lake City	California
Warren McKinley	82589 Westside Freeway Concord	Alaska
Andrew Taylor	29886 Padre Boulevard Madison	California
Ulysses Coolidge	98646 Bayshore Freeway Columbus	Minnesota
Theodore Clinton	12292 San Marcos Bismarck	New York
Benjamin Jefferson	82077 Carpinteria North Sacramento	California
William Van Buren	21712 Tully Road East Albany	Illinois
Calvin Washington	50742 Richmond Hill Charleston	California
Jimmy Polk	76143 Richmond Hill Salt Lake City	Alaska
Calvin Adams	52386 Lake Tahoe Blvd. Montgomery	New York
Ulysses Monroe	70511 Jones Road Trenton	Illinois
Zachary Tyler	45040 Santa Rosa North Carson City	Alaska
Ulysses Johnson	19989 Via Real Juneau	Alabama
George Arthur	89874 Calle Real Annapolis	Alabama
George Jefferson	67703 Fontaine Road Pierre	Illinois
Herbert Grant	90635 North Ventu Park Road Columbus	Alaska
Calvin Washington	37446 E Fowler Avenue Pierre	Alabama
John Harrison	86745 San Marcos Annapolis	Alaska
Benjamin Ford	27921 Carpinteria Avenue Providence	California
Andrew Fillmore	96878 Greenwood Road Saint Paul	Alaska
Warren Arthur	22920 Jean de la Fontaine Madison	Minnesota
Calvin Pierce	41962 Santa Rosa North Juneau	California
Woodrow Clinton	30587 San Diego Freeway Topeka	California
George Jefferson	95054 Padre Boulevard Denver	Alaska
Lyndon Eisenhower	14379 Newbury Road Lincoln	Alabama
John Adams	49993 El Camino Real Phoenix	Alaska

When the Joblet finishes, the message from the second **tMsgBox** component appears. Click **OK** and the Job will execute to completion.



To summarize:

- The first Subjob runs successfully and displays an informational message.
- The Joblet is triggered when the first subJob completes, much the same way a standard subJob would.
- Successful completion of the Joblet triggers another subJob that displays another informational message.

In this example, the subJobs in the input and output flows are very basic. They only display a message to illustrate the flow of control, but they could of course have been more complex.

Now that you have discovered how to create and use Joblets, it's time to Wrap Up.

Wrap Up

In this lesson, you discovered how Joblets allow you to factorize a portion of a Job. They can be built from scratch or from existing Jobs. Joblets can be used within Jobs like any other component, without impacting the performance of your Job.

Next step

Congratulations! You have successfully completed this lesson. To save your progress, click **Check your status with this unit** below. To go to the next lesson, on the next screen, click **Completed. Let's continue >**.

This page intentionally left blank to ensure new chapters start on right (odd number) pages.

LESSON 8

Unit Test

This chapter discusses the following.

Introduction 136
Unit Test 137
Creating a Unit Test 139
Wrap-Up 147

Introduction

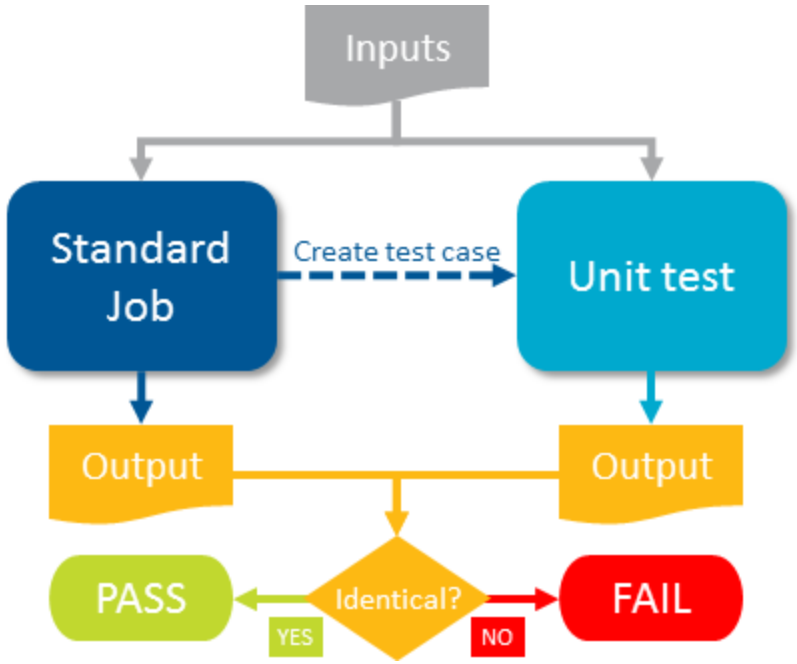
Unit Test

Overview

Testing is always a critical piece in the software development life cycle and often is the responsibility of the developer. "Unit" in "Unit Test" comes from what is typically the smallest part of an application that can be tested. Talend Studio helps the developer by automating the Unit Test process. At a high level, the process is as follows:

- Complete standard Job development
- Run the Job and save the output (used as reference data later)
- Create a test case from the standard Job
- Configure the test case (primarily input/output files and Job components)
- Run the unit test
- Observe results (debug if needed)

From a functional and process flow perspective, in this lab you will build the following:



Objectives

After completing this lesson, you will be able to:

- Create a Unit Test from a working Standard Job
- Explain the highlights of what Talend does when Unit Test creation is automated
- Configure the Unit Test avoiding several of the most common pitfalls
- Run the Unit Test

NOTE:
Unit test is a common industry term, and the name of this lesson. It is referred to often throughout this lesson. The Talend Studio supports automation of unit test creation in the Integration perspective, Mediation perspective (Enterprise Service Bus) and Big Data. If you search the Talend documentation for "Unit Test" the predominant number of results refer to Enterprise Service Bus (ESB). The feature in the Integration perspective that automates the unit test creation is **Create Test Case**. The **Create Test Case** feature is used in this lab exercise.

The first step is to modify an existing Job so it is ready for Unit Test creation.

Creating a Unit Test

Build Standard Job

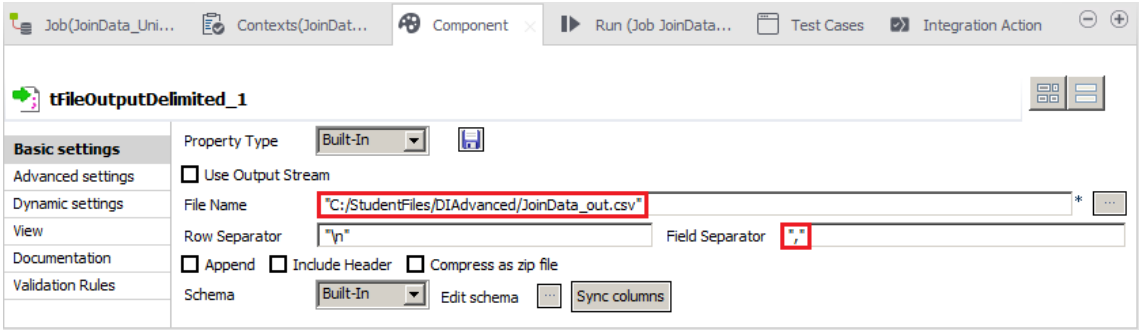
First you will create a standard Job to serve as the basis for your unit test. The Job consists of a **tMap** with two inputs, and writes output to a delimited file. Although the Job is simple, it still serves to illustrate the power and flexibility of this feature.

1. DUPLICATE A JOB

   Duplicate the **JoinData_original** Job. Name the new Job *JoinData_UnitTest* and open it.

2. ADD AND CONFIGURE AN OUTPUT COMPONENT

   Add a **tFileOutputDelimited** component and connect the **tLogRow** to it with a **Main** row. Configure the **tFileOutputDelimited** by setting **File Name** to *"C:/StudentFiles/DIAdvanced/JoinData_out.csv"* and **Field Separator** to *","*.

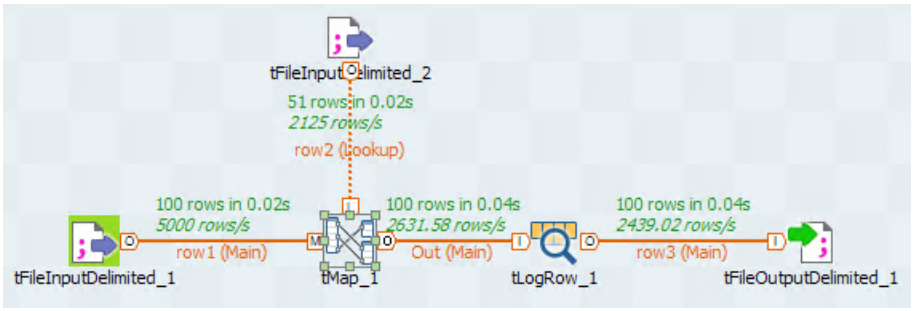


Capture Baseline Output

The unit test requires a reference file to use. Running the standard Job and saving its output will provide that. Subsequent unit tests can use this same reference file to make sure newer versions have not introduced a regression.

1. RUN THE JOB

   Run the Job.



   100 records are processed and the file *JoinData_out.csv* is created, containing 100 lines formatted as follows:

```
Bill Coolidge,85013 Via Real Austin, Illinois

Thomas Coolidge,63489 Lindbergh Blvd Springfield, California

Harry Ford,97249 Monroe Street Salt Lake City, California
```

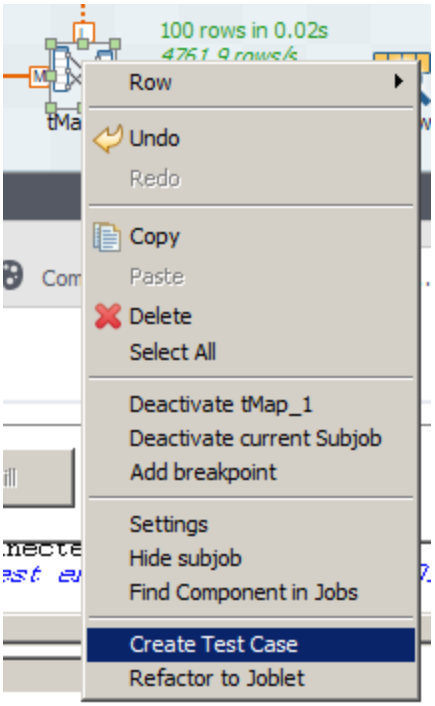
   The **tMap** has transformed raw data into human-readable names and addresses, separated by a comma.

Create the Unit Test

Now you are ready to leverage the Studio's automation capabilities for unit tests. This unit test will center around **tMap**. Unit tests around key components such as **tMap** are very powerful and even considered a best practice for many enterprises. However, realize that a unit test could be created around other components in a Job, or even several components if desired.

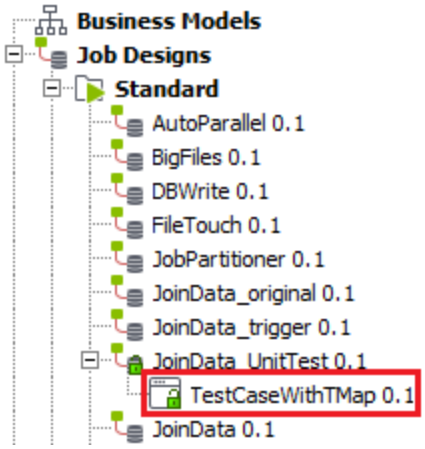
1. CREATE A TEST CASE

   Right-click the **tMap** component in the **Designer** and select **Create Test Case**.



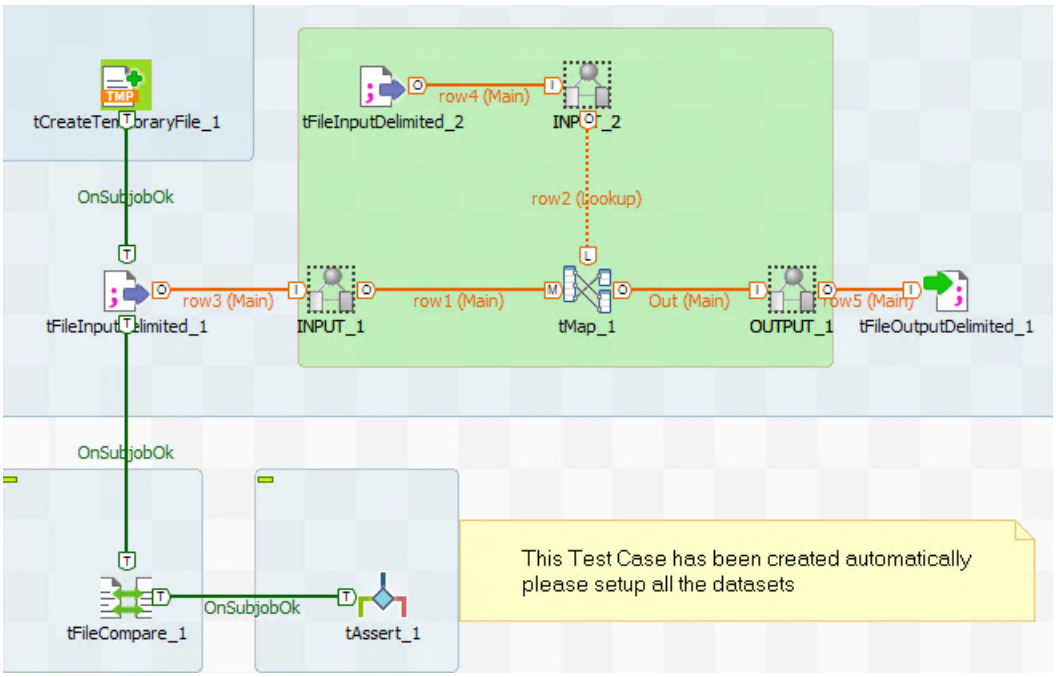
   Name the test case *TestCaseWithTMap* and click **Finish**.

   The unit test is created and placed beneath the standard Job in the **Repository**.



2. EXPLORE THE TEST CASE

   Examine the new Job. If desired, move the subJobs and components around to make the unit test more readable.



The functionality of the standard Job still exists. The **tMap** component has simply been wrapped with several components that provide input/output hooks for testing capabilities:

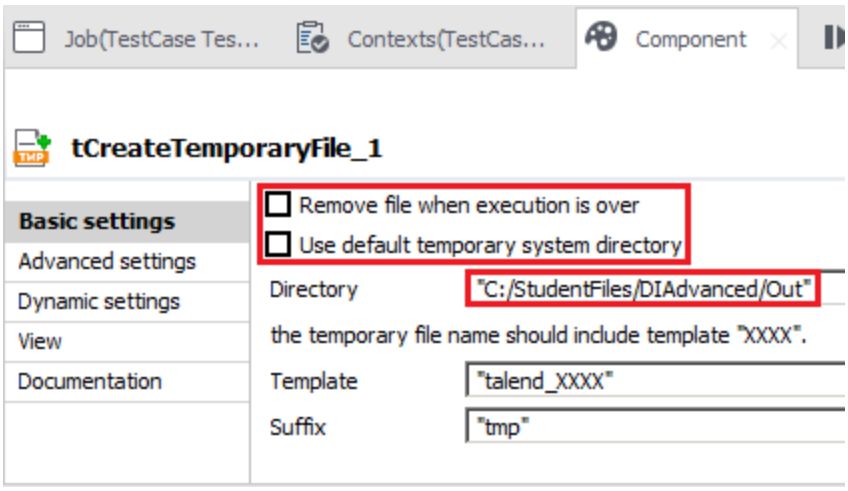
- **tFileInputDelimited** components are added to reference the original input files. The hard-coded input file names in the standard Job are replaced by context variables.
- **tCreateTemporaryFile** is used in the first executed subJob to create a temporary file to store the output of the test run.
- **tFileOutputDelimited** records processed data into the temporary file.
- **tFileCompare** compares the reference file to the temporary output file generated by the unit test when it is run and outputs the comparison results o the console.
- **tAssert** reports whether or not the unit test succeeds.

Configure the Unit Test

Before you can successfully run the test you must configure it. Pay particular attention to the input and output files, field delimiters, and whether or not to account for header rows in your input and output data.

1. CONFIGURE THE tCreateTemporary FILE COMPONENT

   Open the **Component** view for the **tCreateTemporaryFile**. Clear the first two check boxes and specify a *"C:/StudentFiles/DIAdvanced/Out"* for the **Directory**.

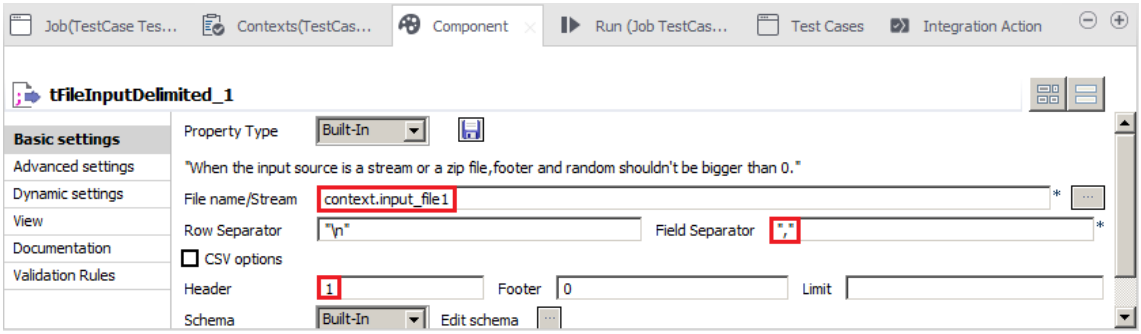


Although not necessary, disabling these options is helpful if any debugging is needed. This configuration makes it simple to track down the output of the test.

2. CONFIGURE THE INPUT COMPONENTS

   Configure the first **tFileInputDelimited** component. Based on the *Customers.csv* input file, you must change the **Field Separator** to a *","* and **Header** to *1*.

   NOTE:
   Notice that **File name/Stream** value was automatically converted to a context variable inside the unit test.



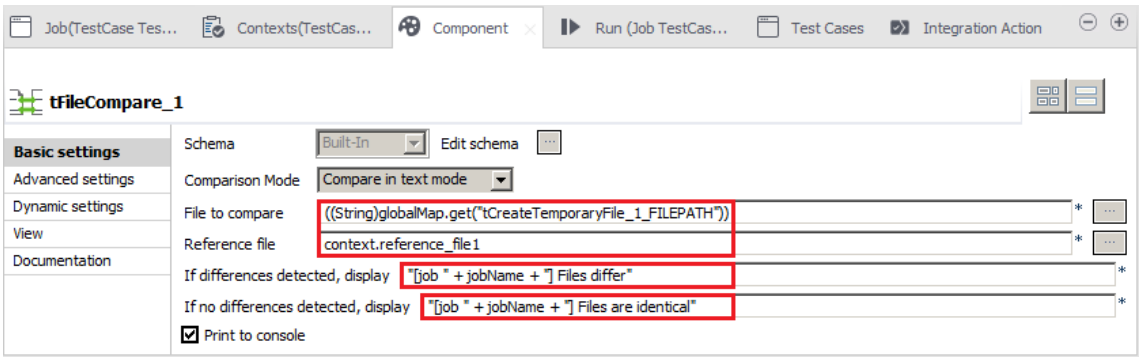
   Configure the second **tFileInputDelimited** component. Based on the *States.txt* input file, set the **Field Separator** to *","*. The **Header** should remain as *0*. Again, notice the **File Name/Stream** was converted to a context variable for you.

3. CONFIGURE THE OUTPUT COMPONENT

   Configure the **tFileOutputDelimited** component. Set the **Field Separator** to *","*. Notice the **File Name** field uses the folder and name specifications from the **tCreateTemporaryFile** component configured earlier. The following code snippet is employed to fetch the file path from the **tCreateTemporaryFile** component: `((String)globalMap.get("tCreateTemporaryFile_1_FILEPATH"))`

4. EXAMINE THE tFileCompare COMPONENT

   Look at the **tFileCompare**component. The configuration should look similar to the following figure.



Although no changes are necessary, it's worth noticing that there are variables set for the following:

- The **File to compare** and the **Reference file** it's compared to.
- The console output messages depending on whether or not differences are detected.

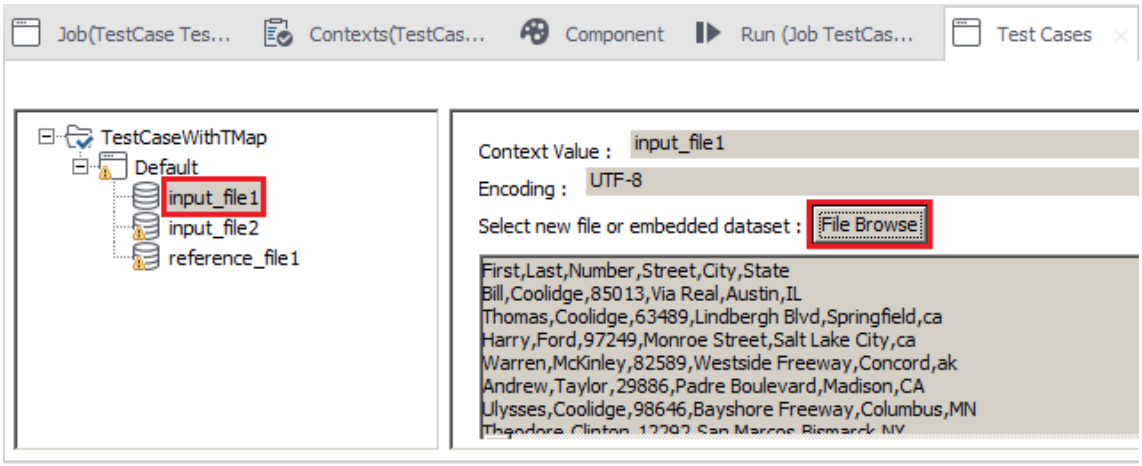
Run the Test Case

Although the unit test is configured, it cannot be run yet. The context variables created for you in the unit test do not reference actual files yet.

1. SPECIFY THE TEST INPUTS

   Switch to the **Test Cases** view. Expand **Default** in the navigation pane on the left until you see the two input files and the reference file.

   Select **input_file1** and use the **File Browse** button to specify the location of that file, *C:\StudentFiles\DIAdvanced\Customers.csv*.



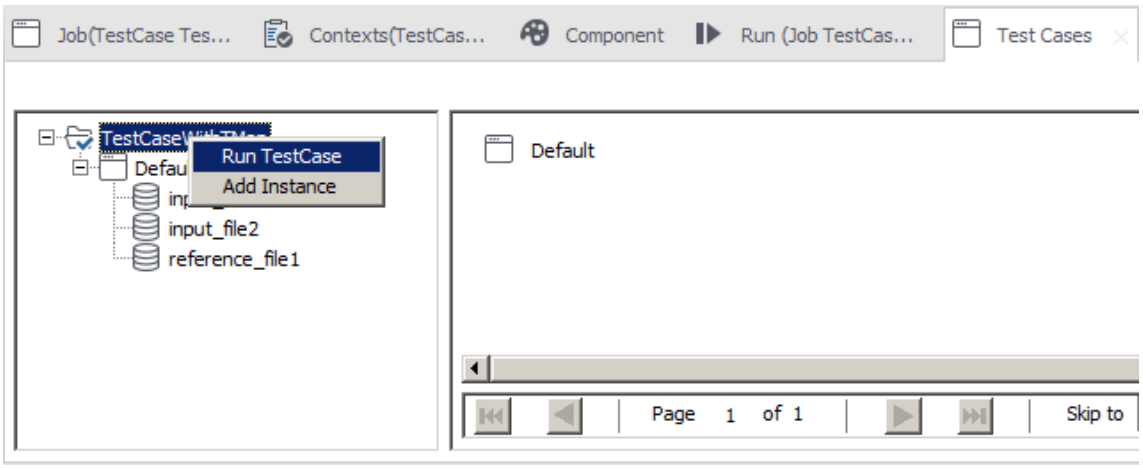
   Perform the same procedure to set **input_file2** to *C:\StudentFiles\DIAdvanced\States.txt*.

   Repeat one last time to set **reference_file1** to *C:\StudentFiles\DIAdvanced\JoinData_out.csv*. Recall that this is the file originally output from the **JoinData_UnitTest** Job, containing the basic name, address and state information in a comma delimited file without any header rows.

   You are now ready to run the unit test.

2. RUN THE TEST

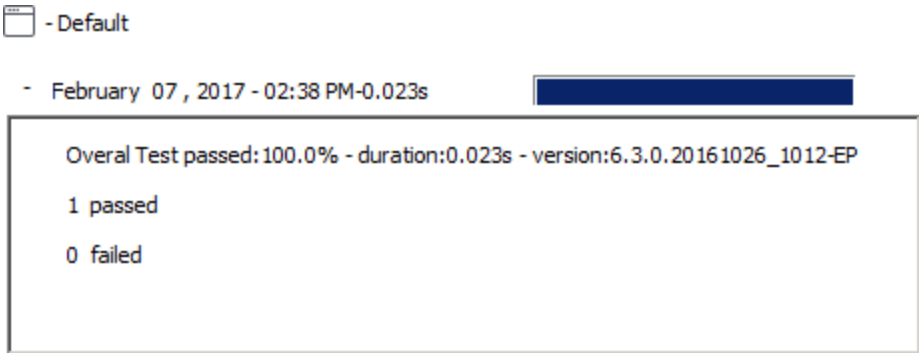
   While still in the **Test Cases** view, right-click the test case on the left pane and select **Run TestCase**.



The unit test runs.

3. EXAMINE THE RESULTS

Expand **Default** in the results panel. The results are time stamped each time you run the test. Expand the time stamp to see the test results.

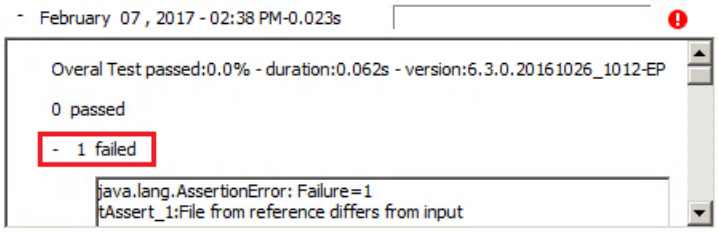


A history is built over time, with the most recent pushed onto the top of the results window.

NOTE:
If a test fails, it is flagged with a red icon ( ).

There are a few places to consider looking to debug issues in the event of a failed Unit Test:

- Expand **+ 1 failed** in the results. This item is expandable and reveals more information. In the example below, the **tAssert** component indicates that the compared files differ.



- Input components (look at the row, field separator and number of lines for header)
- Output components

» The temporary file. Based on earlier **tCreateTemporaryFile** configuration, you should see non-zero size temporary files in your output folder. Each temporary file represents a different run of the test case. For example: *talend_DDWA.tmp* or *talend_VVMW.tmp*.

You can then examine the contents of these temporary file to see if it is formatted as expected.

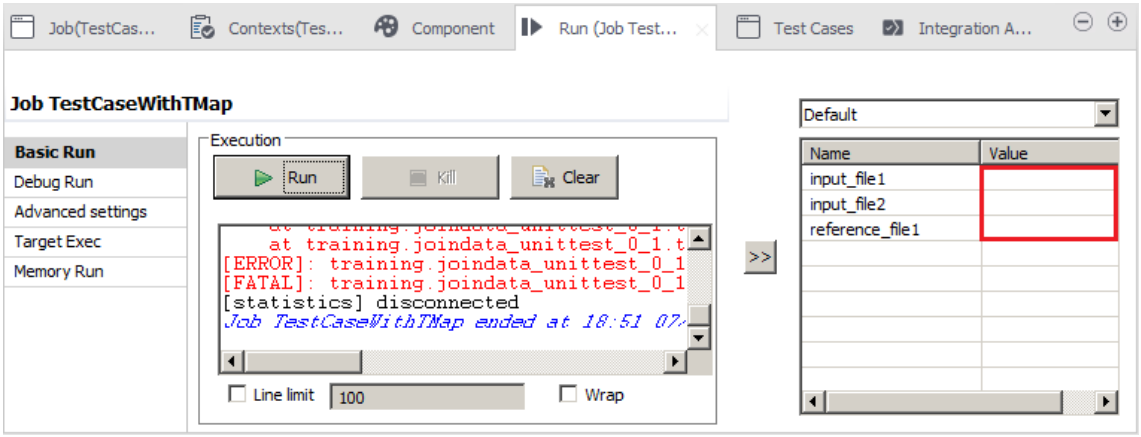
Basic Run of the Unit Test

Although your unit test ran successfully, some may want to see more information than the simple pass/fail results in the **Test Cases** view.

1. RUN THE TEST

   Switch from the **Test Cases** view to the **Run** view. Run the Job, and notice from the console output that it fails miserably!

   Turn your attention to the values for the context variables on the right-hand side of the view.

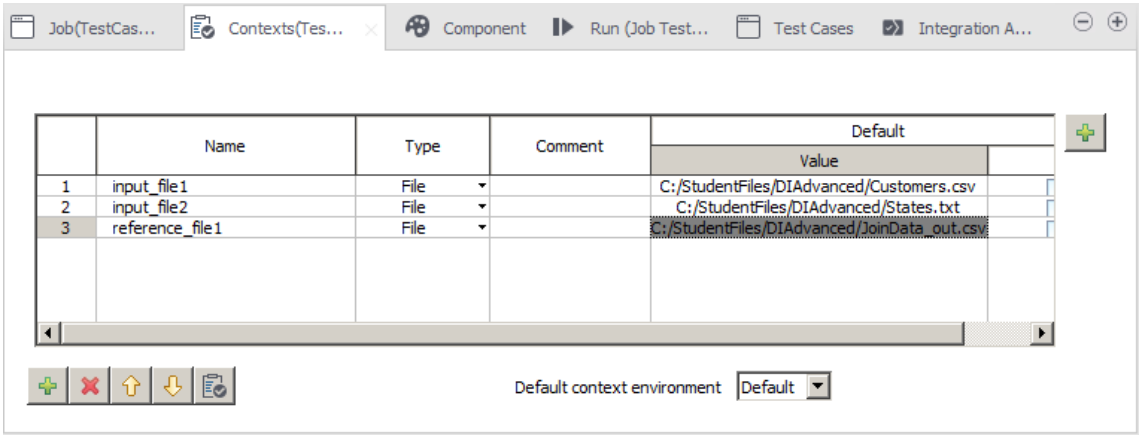


   The values for the context variables are null. It is important to understand that setting the context variables in the **Test Cases** view does not set them to the same values in other views.

   Before you can run the Job this way, you will need to set the context variables appropriately.

2. SET THE CONTEXT VARIABLES

   Switch to the **Contexts** view. Set the value for each context variable as shown in the figure below.



3. RUN THE TEST AGAIN

   Switch back to the **Run** view, and run the Job again.

```
Starting job TestCaseWithTMap at 19:26 07/02/2017.

[statistics] connecting to socket on port 3664
[statistics] connected
[job TestCaseWithTMap] Files are identical
[statistics] disconnected
Job TestCaseWithTMap ended at 19:26 07/02/2017. [exit code=0]
```

The Job succeeds, as it did before.

Realize that locking down your test files is an important part of the process. In fact, there are several decisions enterprises need to make as part of their implementation plan. There is no universal solution for all enterprises and Jobs. Here are a few example considerations with respect to processes and procedures:

- What to create unit tests for? Every component that alters content in the data flow? Or perhaps grouping several components together is sufficient for some Jobs?
- Will unit tests be run from the **Test Cases** view only, or must basic run be supported?
- Is the **tFileCompare** of the Unit Test sufficient? Does your organization require a deeper comparison? For example, a custom subJob that supplements the **tFileCompare** and incorporates a file check sum or other customized functionality.

You have completed the Unit Test lesson so it's time to Wrap-Up.

Wrap-Up

In this lesson, you learned that the **Create Test Case** feature in the Studio automates the build of a unit test. You can build the unit test around one or more components. Most of the work revolves around configuring the input and output components, then setting context variables for them prior to running the test. Finally, you saw how you can run the test from either the **Test Cases** view or standard **Run** view.

In any case, the unit test ultimately compares a reference file, which is sample output from the original Job the test was built from, against output in the data flow.

Next step

Congratulations! You have successfully completed this lesson. To save your progress, click **Check your status with this unit** below. To go to the next lesson, on the next screen, click **Completed. Let's continue >**.

This page intentionally left blank to ensure new chapters start on right (odd number) pages.

LESSON 9

Change Data Capture

This chapter discusses the following.

Introduction 150
Change Data Capture 151
Examining Databases 152
Configure the CDC Database 161
Monitoring Changes 169
Updating a Warehouse 180
Challenge 186
Solutions 187
Wrap-Up 190
Resetting the Databases 191

Introduction

Change Data Capture

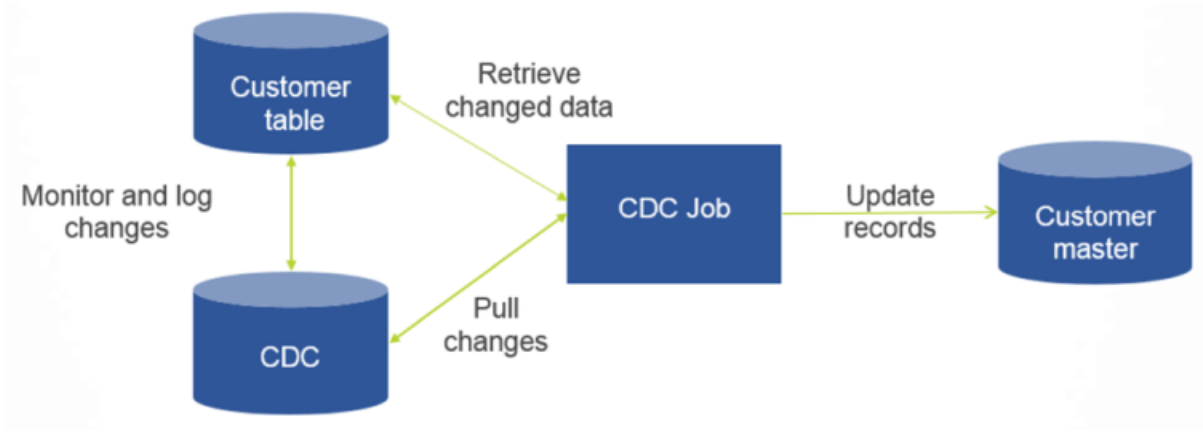
Overview

NOTE:
This is an optional lab that takes approximately two hours to complete.

Many situations require that you keep two or more databases synchronized with each other. For example, you might have a centralized data warehouse that you need to keep current with one or more subsidiary databases. Given that today's databases are frequently massive, reloading the entire subsidiary database into the warehouse is often impractical. A more realistic solution is to monitor the subsidiary database for changes, then duplicate those changes in the master or warehouse database.

In this lesson, you will configure a Change Data Capture (CDC) database that monitors a separate database containing customer data for changes — record updates, deletions, and insertions.

The CDC database stores a list of the indexes of the records that have changed, the type of change, and a time stamp of when the change occurred, but not the actual changes themselves. You then create a Job that uses that list to update the master database with just the modified records from the subsidiary database:



Objectives

After completing this lesson, you will be able to:

- Configure a database table to be monitored for changes in a separate CDC database
- Create a Job that uses the information in a CDC database to update a master database table with just the changes from the monitored database table

The first step is to examine the existing databases.

Examining Databases

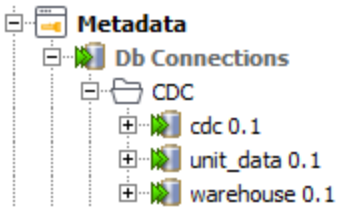
Overview

The scenario for this exercise involves a database containing customer information maintained by a single business unit and a centralized data warehouse that must be kept current with changes to the business unit database. Both of those databases have been created for you, so your first step is to examine the database connections and the data.

Retrieve Schema

1. IDENTIFY METADATA CONNECTIONS

   Expand Repository > Metadata > Db Connections > CDC to locate the three database connections that will be used in this lesson.

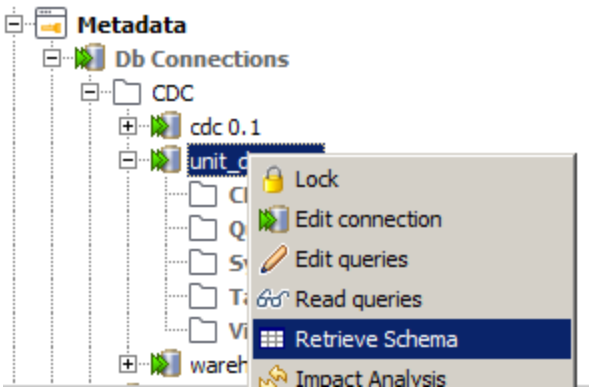
   

   There are three connections represented:

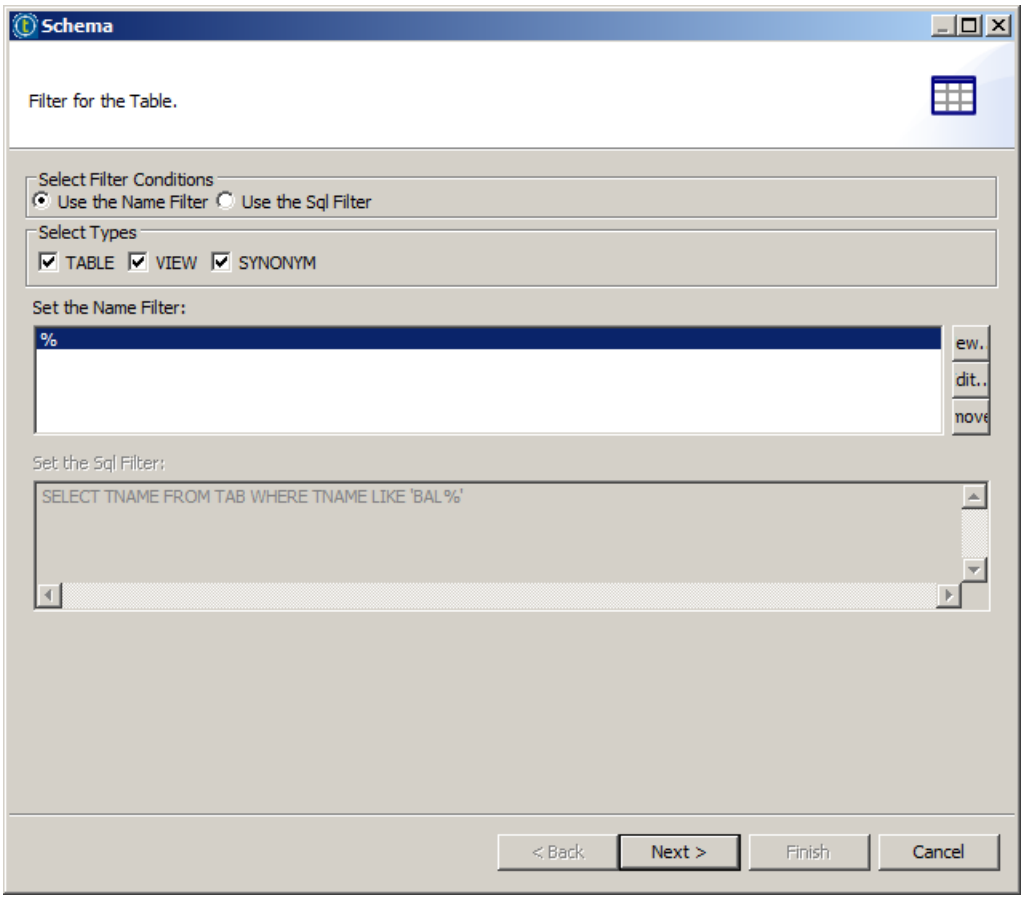
   - **unit_data** represents the connection to the subsidiary database
   - **warehouse** is the connection to the master data warehouse
   - **cdc** will be used later to connect to the database that manages the lists of changes

2. RETRIEVE THE SCHEMAS

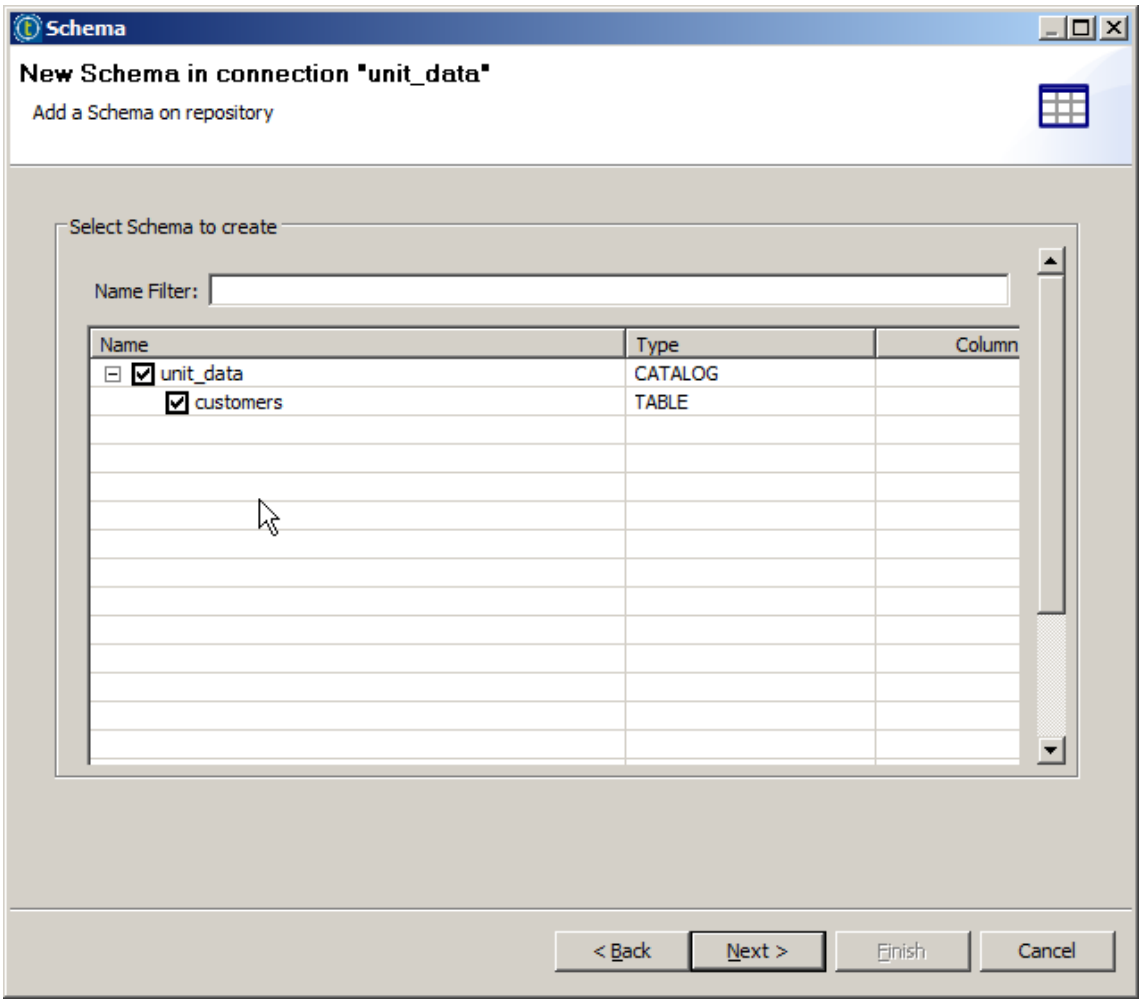
   Retrieve the schema for the **unit_data** connection by right-clicking on it and selecting **Retrieve schema**.

   In the **Schema** window, leave all settings as is and click **Next**.

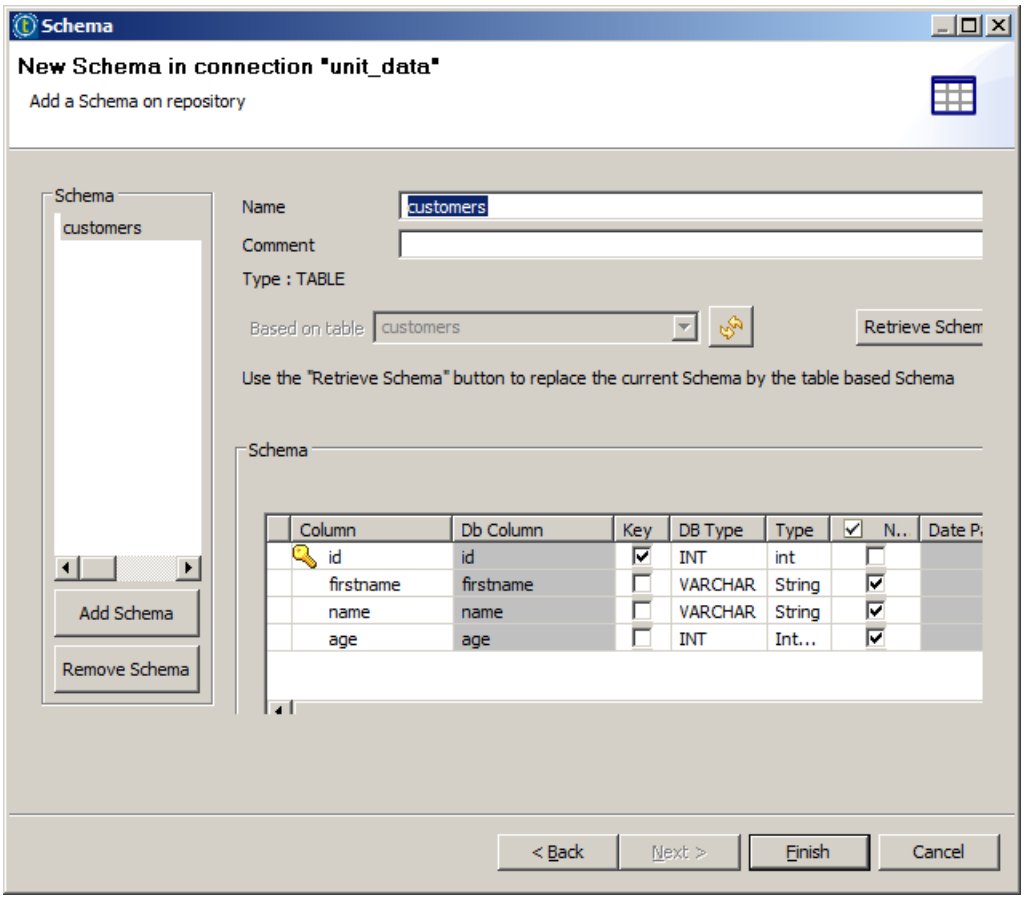


Specify the schema you want to retrieve by selecting **unit_data > customers**, then click **Next**.

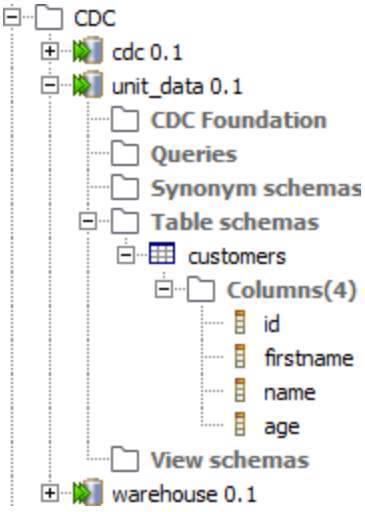


Click **OK** in the warning dialog that appears.

In the final step, examine the schema name and elements that were obtained by the database. In this case, there is no need to make any changes, so click **Finish**.



The schema appears within the **Repository**, under **unit_data > Table schemas > customers**.



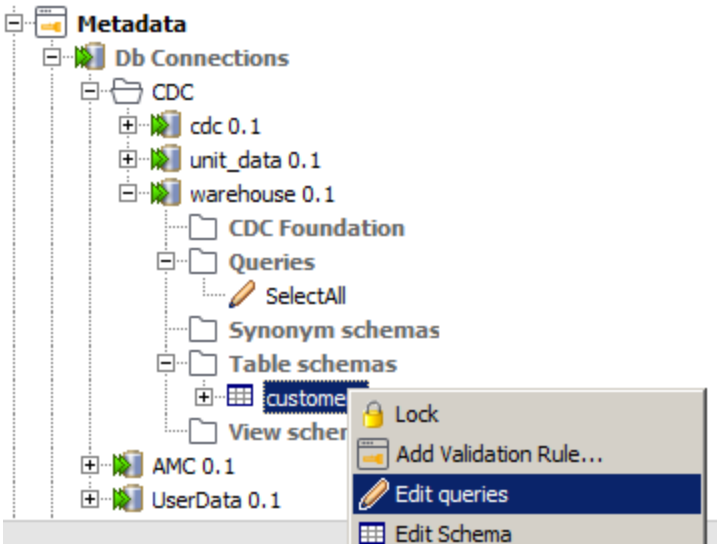
Now, repeat these steps to retrieve the schema for the **warehouse** connection.

Note that the schema is the same in both databases. That is, they both contain a single table named *customers*, that has four columns:

- *age*
- *firstname*
- *id* (this is the primary key)
- *name*

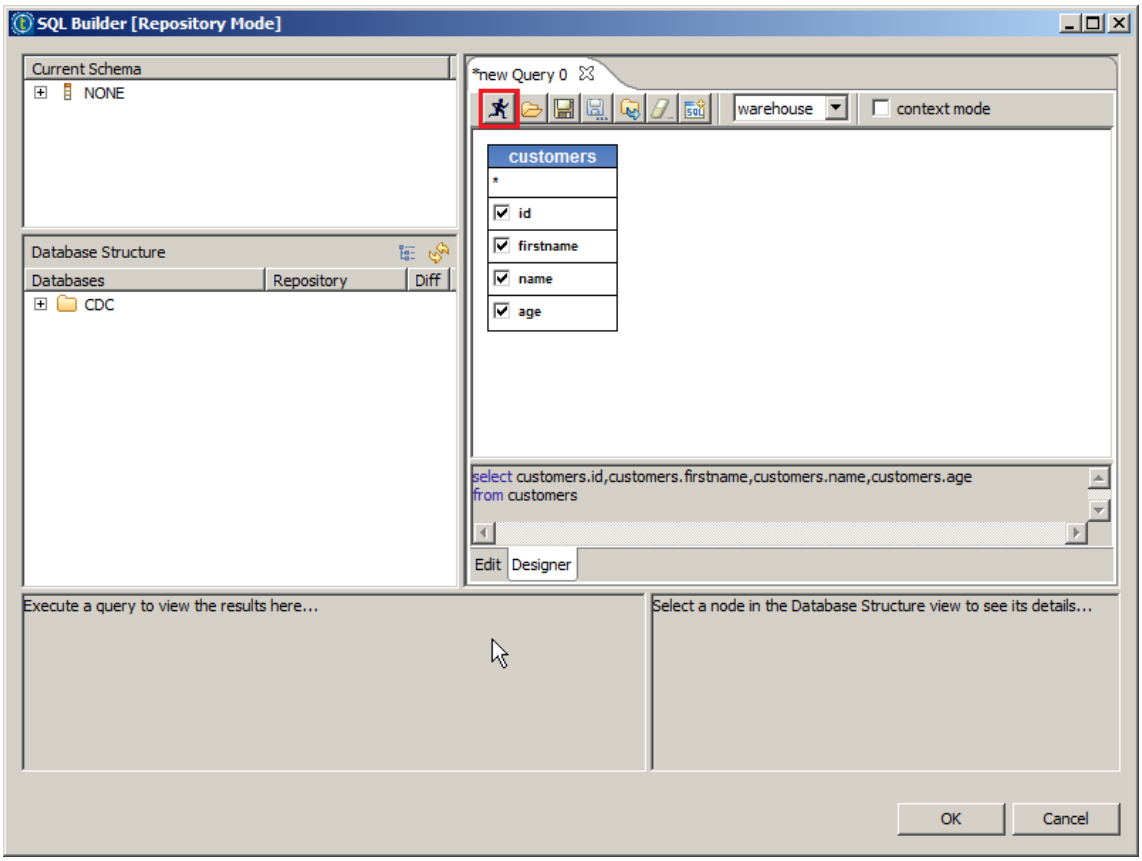
3. SET UP A QUERY

   Under the **warehouse** connection, right-click **Table schemas > customers** and select **Edit queries**.

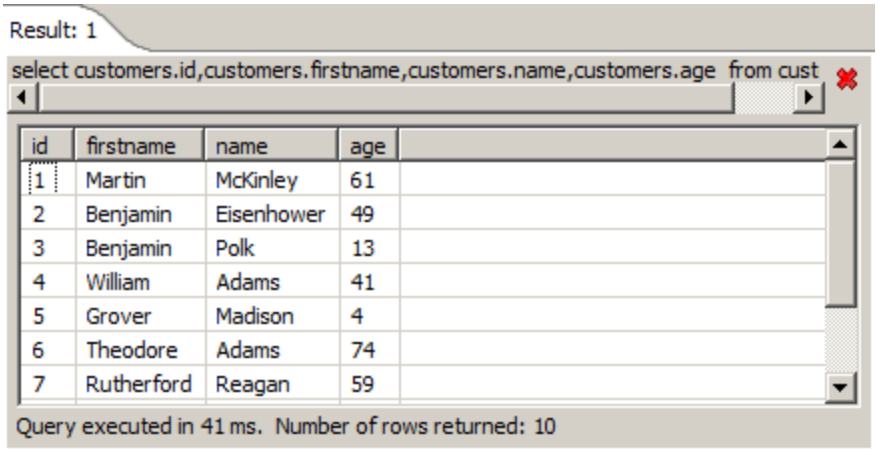


   The **SQL Builder** window appears, which allows you to create, configure, and save queries for reuse. The default query is to retrieve all values for all columns.

   Click the **Execute SQL** button () in the **new Query** tab to execute the query.

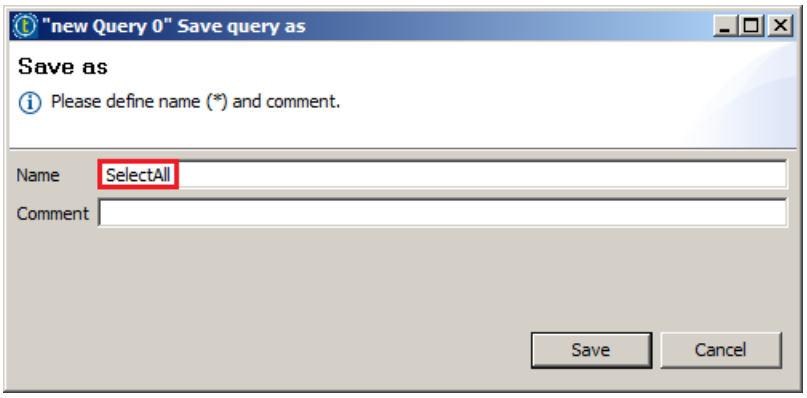


Examine the results in the result tab.



Click **OK**. When prompted to save the queries, click **Yes**.

Name the query *SelectAll*, and click **Save**.



The query appears in the **Repository** for later reuse:

Compare Data

1. CREATE A JOB

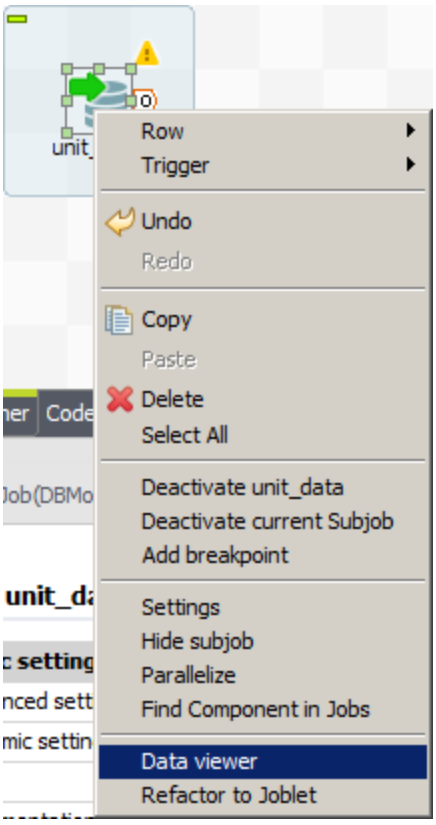
   Create a new standard Job named *DBMods*. Eventually this Job will make modifications to the **unit_data** database to be reflected in the **warehouse** database, but for now you will use it just to examine the contents of a table in the **unit_data** database.

2. POPULATE THE JOB

   Drag the **unit_data** connection from the **Repository** onto the **Designer**, choosing **tMysqlOutput** as the component type.

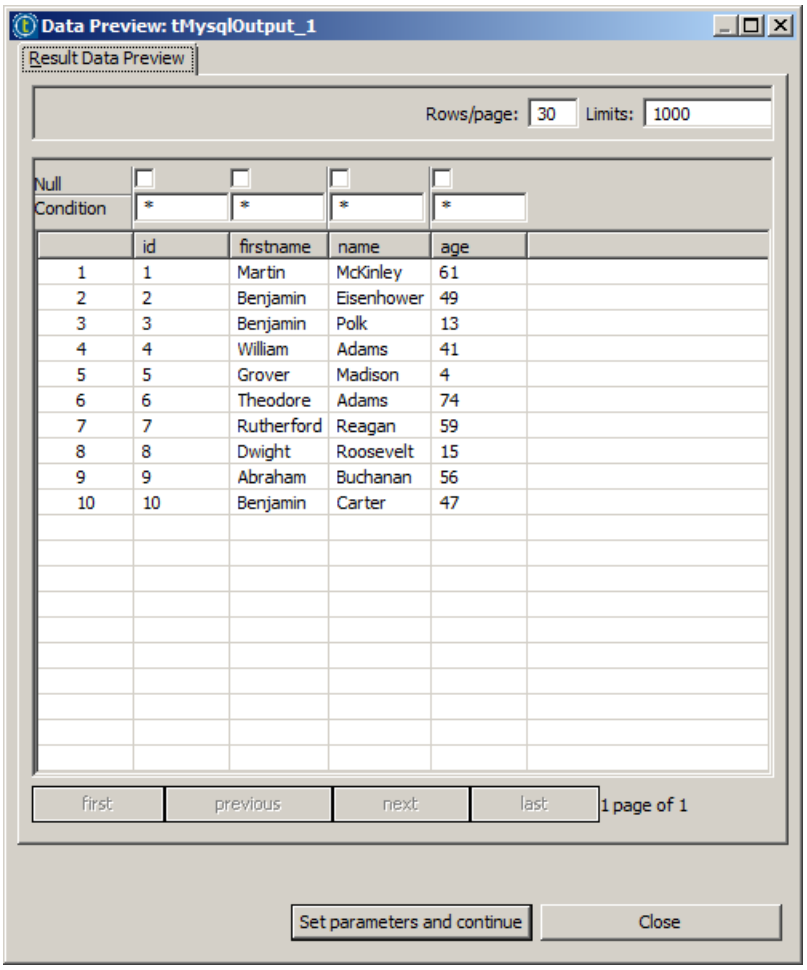
3. PREVIEW THE DATA

   Right-click the new component and select **Data viewer**.



The **Data Preview** window appears, which allows you to examine the contents of many different types of files and databases without having to leave Talend Studio.

In this case, the **customers** table contains ten rows of random name data that was generated from a Talend Job.



You can now see that the similarity between the tables in the **warehouse** and **unit_data** databases is not limited to the schema; the actual data is identical also.

4. SAVE THE JOB

   Save the Job. You will return to it later to customize it further.

Now that you've examined the existing databases, it's time to set up the CDC database. Again, note that the CDC database will not be a duplicate of the monitored database; rather, it will collect all modifications made to the monitored database.

Configure the CDC Database

Overview

The CDC database has been created for you, but is empty. Now, you will set up the CDC process so that changes to the **customers** table in the **unit_data** database get recorded in the CDC database. Later, that information will be used in order to update the **customers** table in the **warehouse** database.

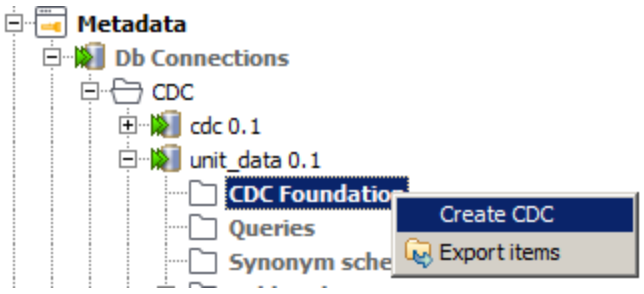
In this scenario, the **customers** table in the **unit_data** database is said to be the *source*, and the **customers** table in the **warehouse** database is the *target*.

Define Subscribers Table

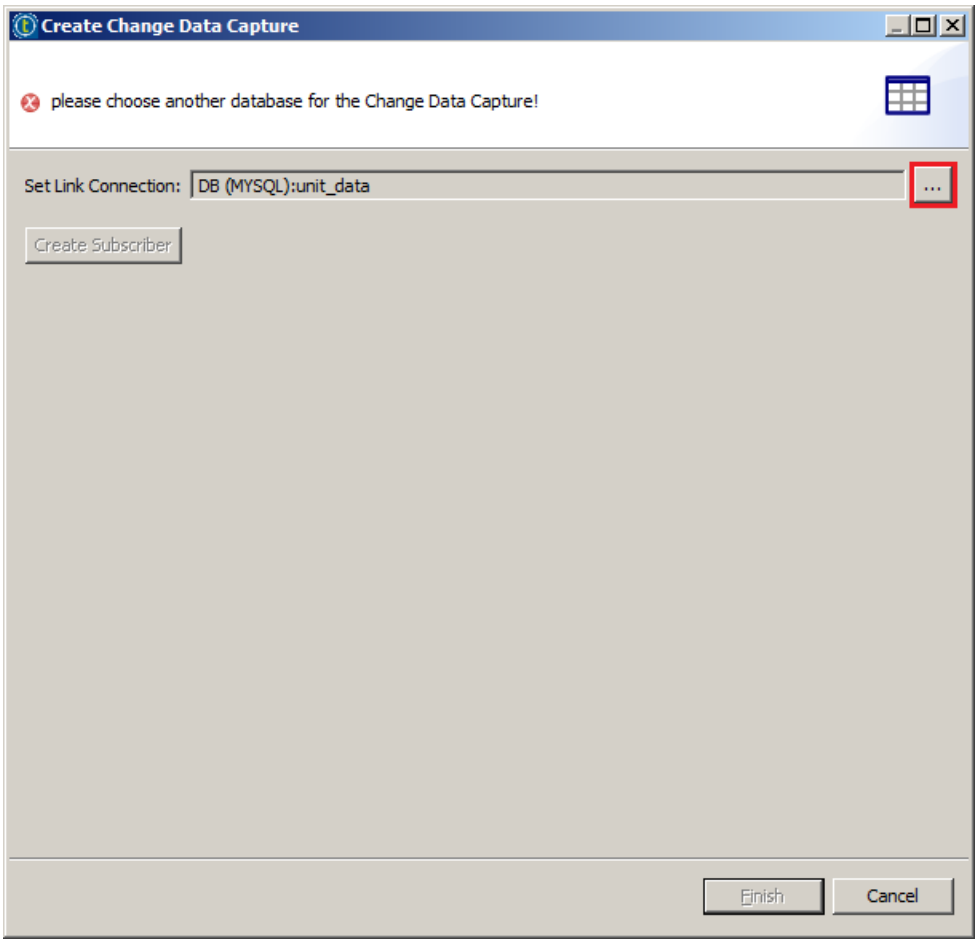
1. SET UP A SUBSCRIBER TABLE

   A *Subscriber* uses information captured in a CDC database to later update other databases. Since you are interested in monitoring the table in the **unit_data** database, you will create a subscriber table to represent this source.

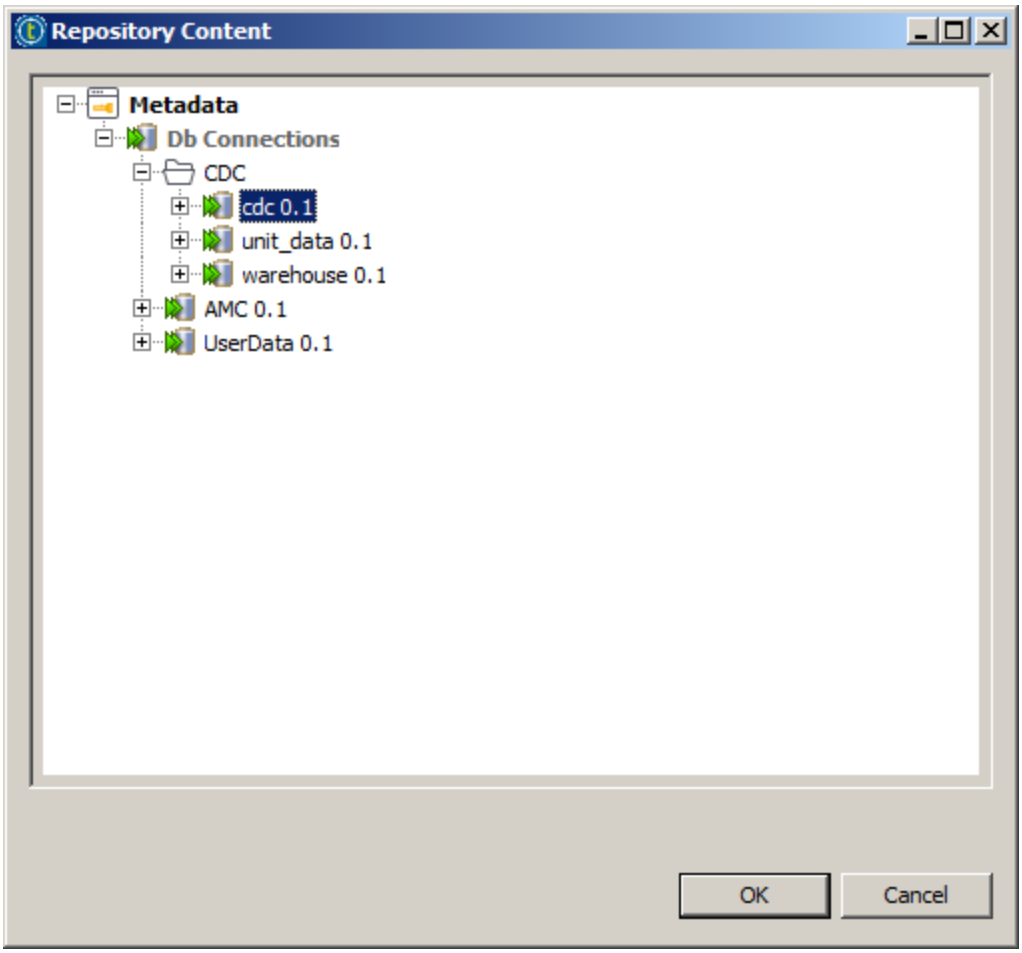
   In the **Repository**, right-click **Metadata > Db connections > CDC > unit_data > CDC Foundation** and select **Create CDC**.



   Identify the database that will monitor changes to the **unit_data** database by clicking the button marked to the right of the **Set Link Connection** box.



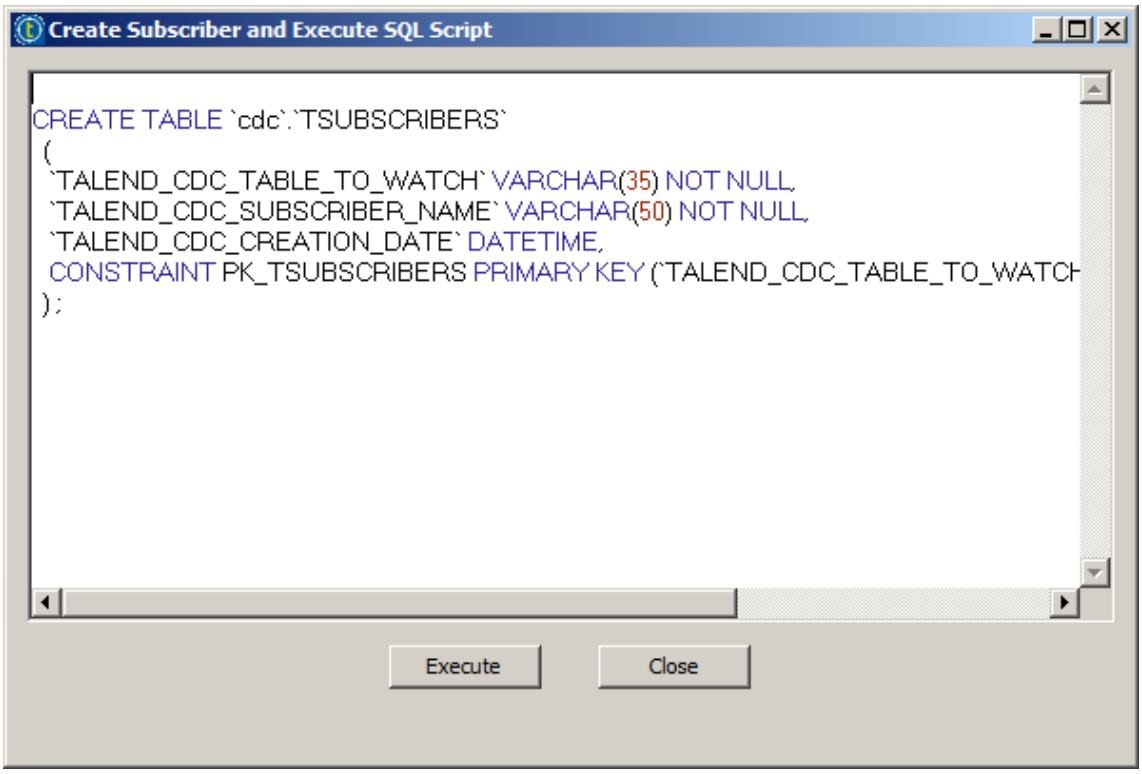
In the **Repository Content** window that appears, select **Db Connections > CDC > cdc**, then click **OK**.



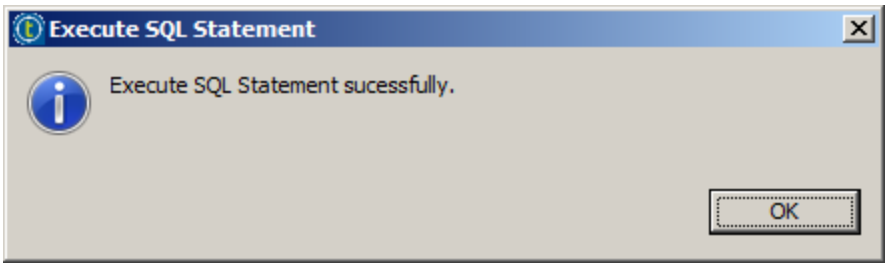
Back in the **Create Change Data Capture** window, click **Create Subscriber**.

A window appears that displays the SQL statement that will be executed to create the subscriber table. Take a moment to examine this script that creates a table named **TSUBSCRIBERS** in the **cdc** database.

When ready, click **Execute**.

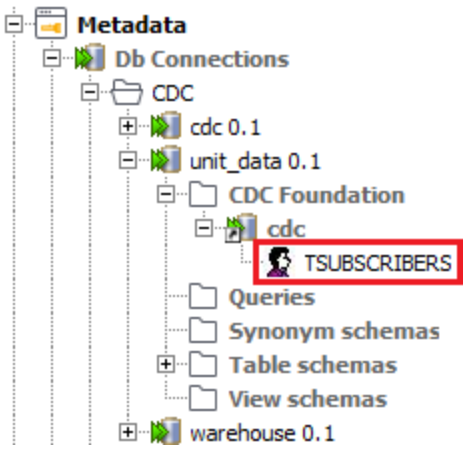


The script executes and notifies you when it completes successfully. Click **OK**.



Back in the **Create Subscriber and Execute SQL Script** window, click **Close**, then click **Finish** back in the **Create Change Data Capture** window.

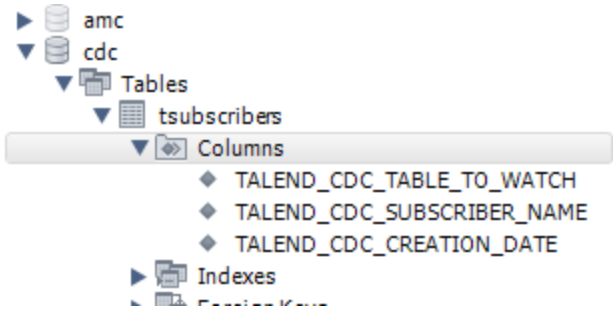
The new table appears in the **Repository**.



NOTE:
Even though the table is listed under **unit_data** it is actually in the **cdc** database. This makes it easier to see the organization when you have a single CDC database monitoring multiple tables.

2. EXAMINE THE SUBSCRIBER TABLE

In **MySQL Workbench**, expand **SCHEMAS > cdc > Tables > tsubscribers > Columns**.

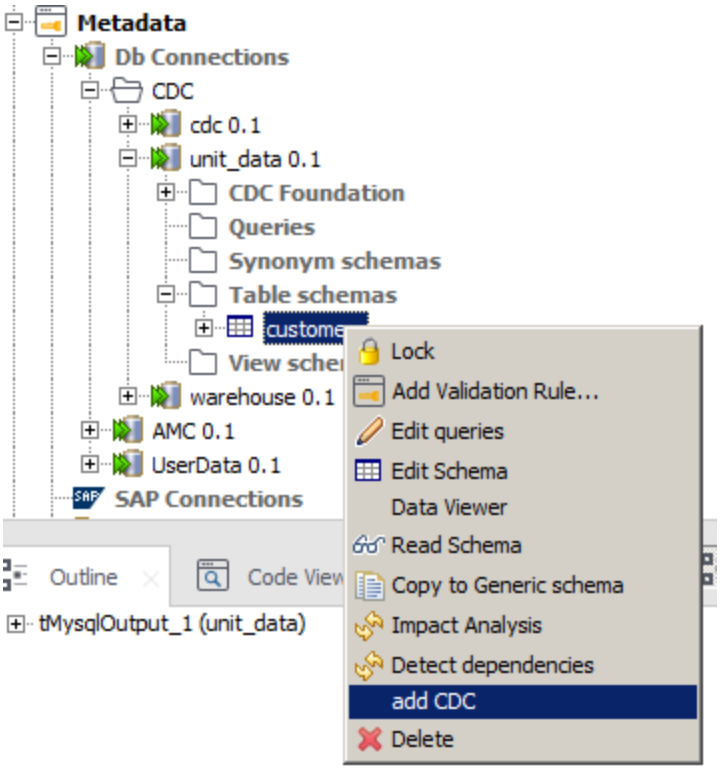


The column names should look familiar from the sql script you just ran from Talend Studio. These columns store information about what table to monitor.

NOTE:
If you do not see the table columns, then click the refresh icon ( ).

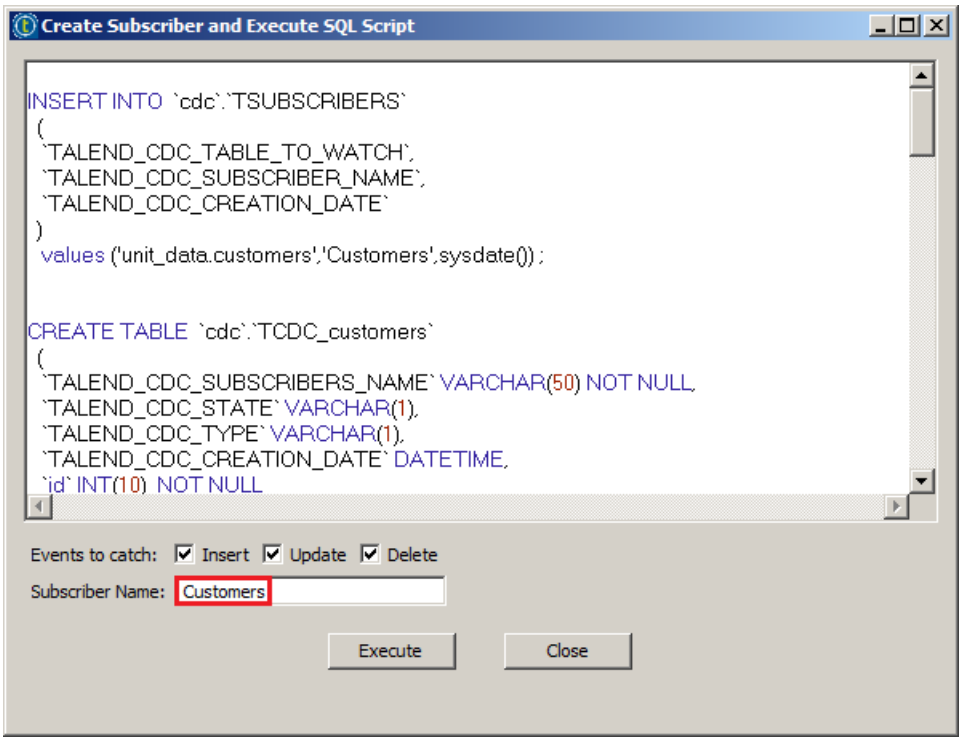
3. DEFINE THE CDC TABLE

Back in Talend Studio, within the **Repository**, right-click **Metadata > Db connections > CDC > unit_data > Table schemas > customers** and select **add CDC**.



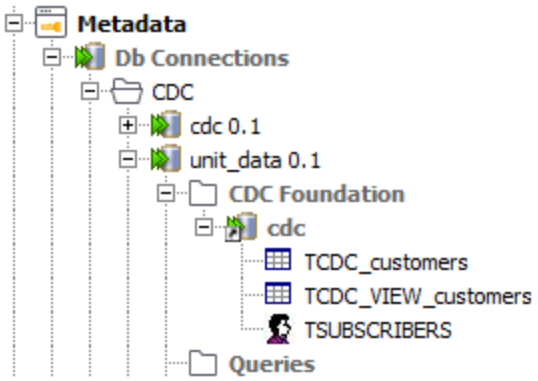
Another SQL script is displayed. This one inserts data into the existing **subscribers** table and creates a new table and view in the **cdc** database.

Enter *Customers* in the **Subscriber Name** box, then click **Execute**.



As before dismiss the windows until you return to the main Talend Studio window.

Notice the new information displayed under **unit_data**.



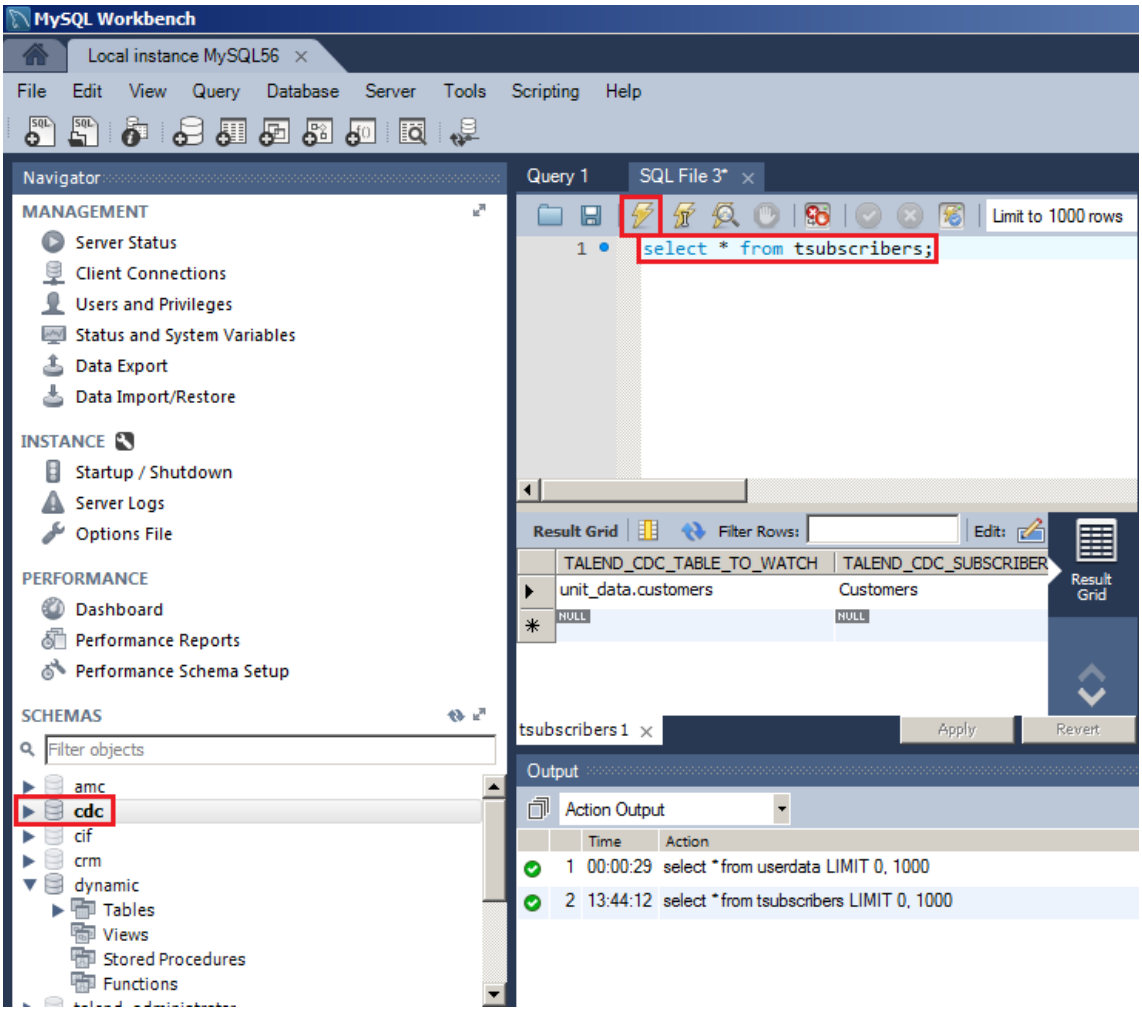
NOTE:
Again, the new information is listed under **unit_data** for the sake of convenience, but the tables and view are actually in the **cdc** database.

4. EXAMINE THE CDC TABLE

In **MySQL Workbench**, select **File > New Query Tab** from the main menu bar.

Double-click **SCHEMAS > cdc** to select the database on which to operate.

Then, enter `select * from tsubscribers;` in the new query tab and execute the query using the **Execute** button ( ).



The results show the table being monitored (**unit_data.customers**), the name of the subscriber you just created (**Customers**), and a time stamp specifying when the subscriber was created.

Right-click **SCHEMAS > cdc** and select **Refresh All**.

Expand the **SCHEMAS > cdc > Tables > tcdc_customers**. This table stores information about changes in the subscribed table (in this case, the **customers** table of **unit_data** database). The columns have the following meaning:

- TALEND_CDC_SUBSCRIBERS_NAME: the name of the subscriber as listed in the **tsubscribers** table. In this exercise, **Customers** is the subscriber name and identifies the table being monitored.
- TALEND_CDC_STATE: a flag indicating whether or not this change has been applied.
- TALEND_CDC_TYPE: one of U, D, or I specifying the type of change (update, delete, or insert, respectively).
- TALEND_CDC_CREATION_DATE: a time stamp specifying when the record changed.
- id: the key value identifying the changed record. Note that the name of this column is specific to the table being monitored. Remember that earlier when retrieving the schema you learned that the column **id** is the primary key for the **customers** table.

Expand **SCHEMAS > unit_data > Tables > customers > Triggers**. Notice that all three types of changes are being monitored: insert (I), update (U), and delete (D).

Now that the database and Studio have been configured for CDC, you are ready to make some changes to the monitored table.

Monitoring Changes

Overview

With the CDC monitoring in place, now you can make some changes to the **unit_data** database to see how CDC tracks the changes. You will extend the **DBMods** Job to apply insert, delete and update operations on a customer list. Only a few changes will be made in order to make tracking and verification simple, but of course more modifications could be applied to simulate a more realistic scenario.

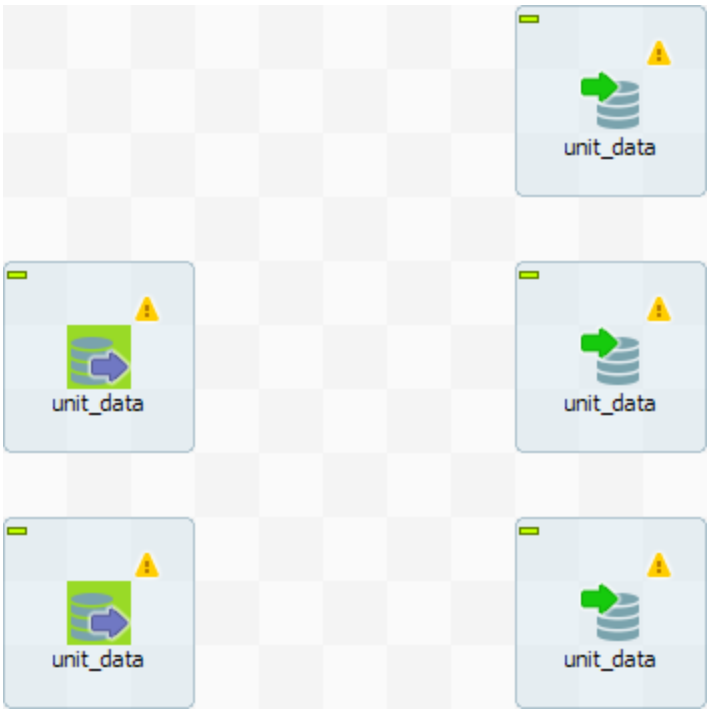
Build the Database Modification Job

1. CREATE THREE OUTPUTS

   Open the **DBMods** Job you created earlier, if it is not already open. Paste two copies of the **tMysqlOutput** component so that you have a total of three. Each one of these will be relegated to handling one of the three database operations: insert, update, and delete.

2. ADD TWO INPUTS

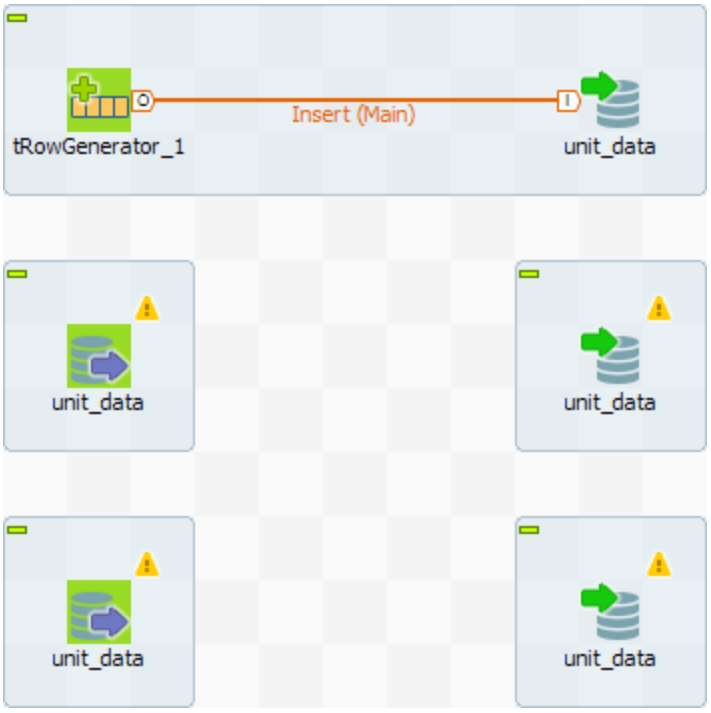
   Drag the**unit_data** database connection from the **Repository** onto the Designer, to the left of the second existing output component. This time choosing the **tMysqlInput** component. Paste a copy of this component to the left of the third output.



3. CREATE AN INSERT SUBJOB

   Add a **tRowGenerator** component to the left of the first **tMysqlOutput** component, and connect the two with a **Main** row. When prompted to get the schema of the target component, click **Yes**.

   To simplify Job maintenance, name this connection *Insert*.

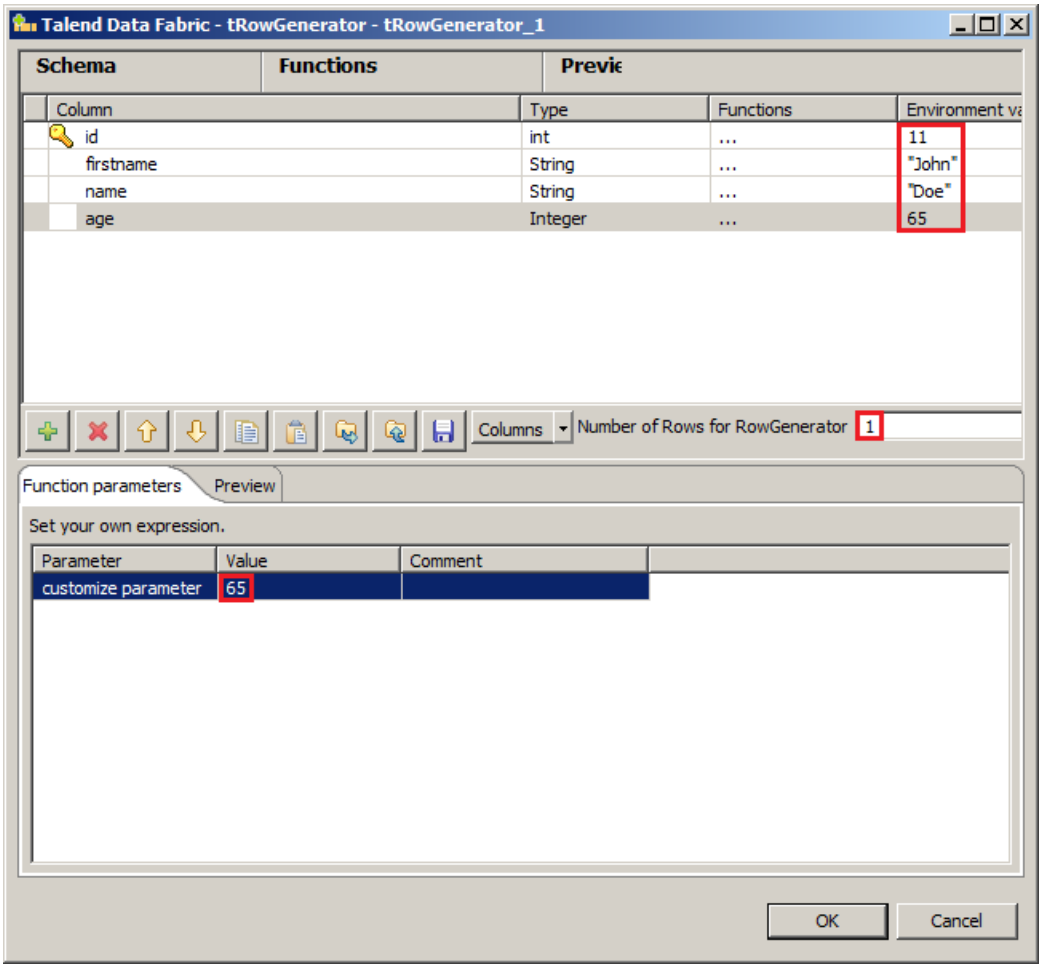


Configure the **tRowGenerator** to generate a single, hard-coded record by double-clicking it, then:

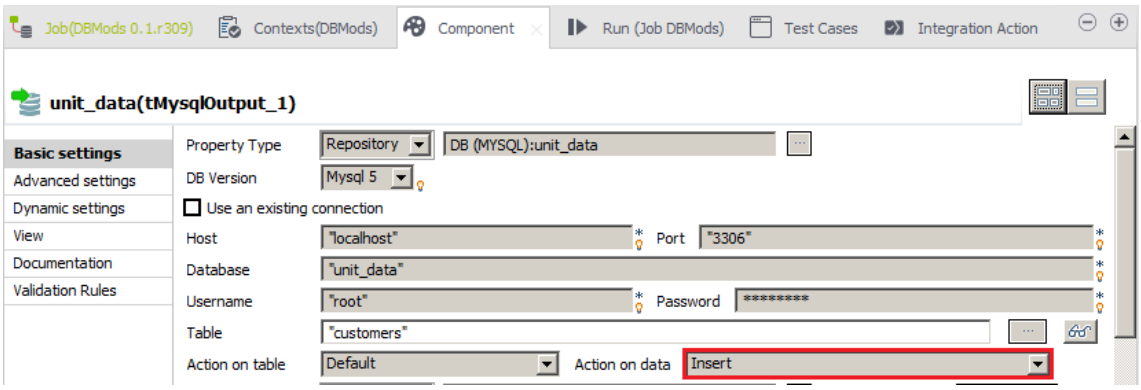
- Set the **Number of Rows** value to *1*
- Set **id** to *11*
- Set **firstname** to *"John"*
- Set **name** to *"Doe"*
- Set **age** to *65*

Set these by selecting on the appropriate row in the table above, then entering the appropriate data into the Value box below.

WARNING:
Be sure to enclose **String** values in quotation marks, but not **Integer** type entries.

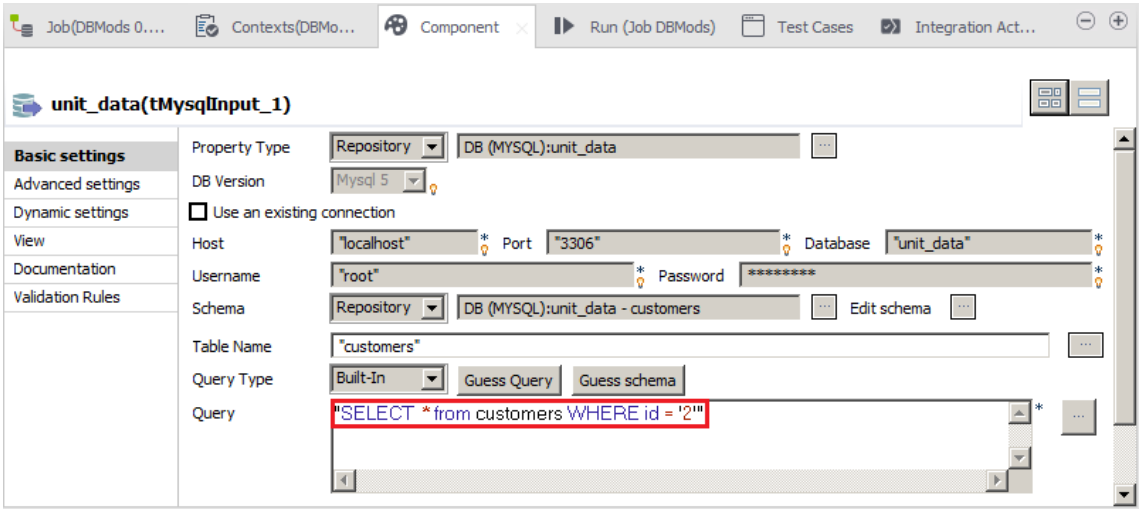


To insert this record into the database, configure the **tMySQLOutput** component by double-clicking it. In the **Component** view, ensure that **Action on data** is set to *Insert*.

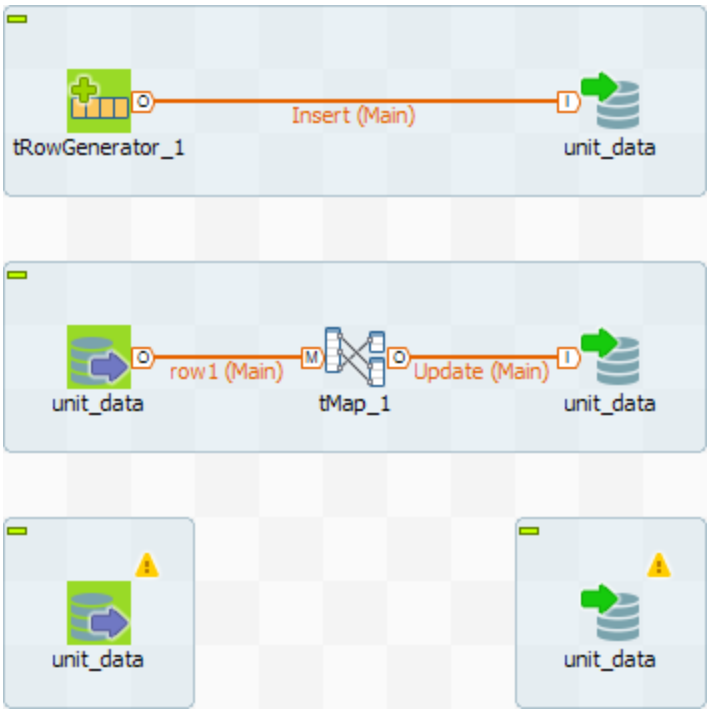


4. CREATE AN UPDATE SUBJOB

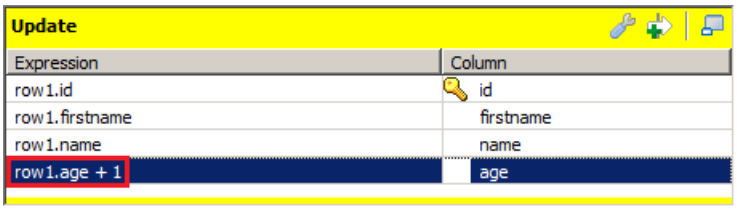
Select the second **tMysqlInput** component. In the **Component** view, replace the text in the **Query** box with *"SELECT * FROM customers WHERE id = '2'"*, which will return only a single row.



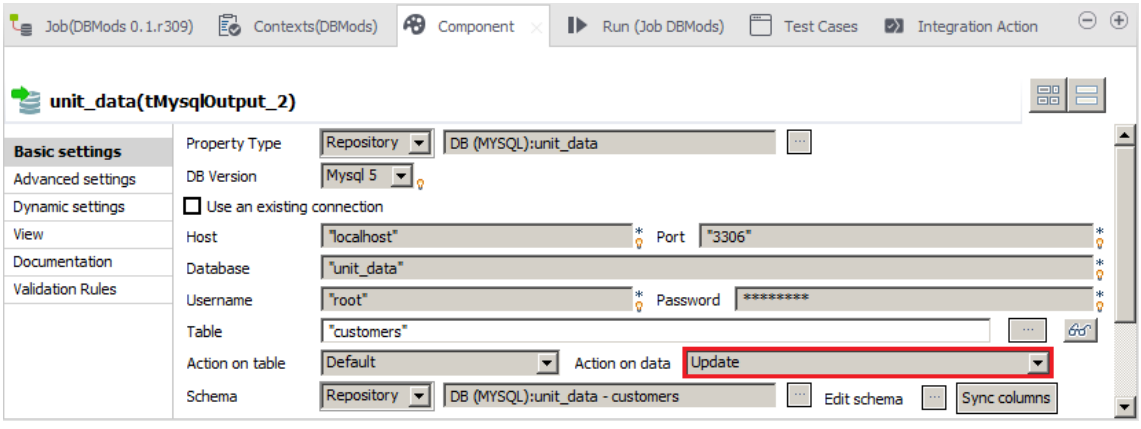
Add a **tMap** component between this input component you just modified and the corresponding output component. This will modify the input record. Connect the input component to the **tMap** using a **Main** row. Connect the **tMap** to the output using a new output named *Update*. Again, when prompted to get the schema of the target component, click **Yes**.



Double-click the **tMap** component to configure it. Drag the all the columns from the **row1** table to the **Update** table so that the output table will have the same values as the input table. Now, change the expression for the **age** column in the **Update** table to *row1.age + 1*, which will increment the age read from the database by one year.

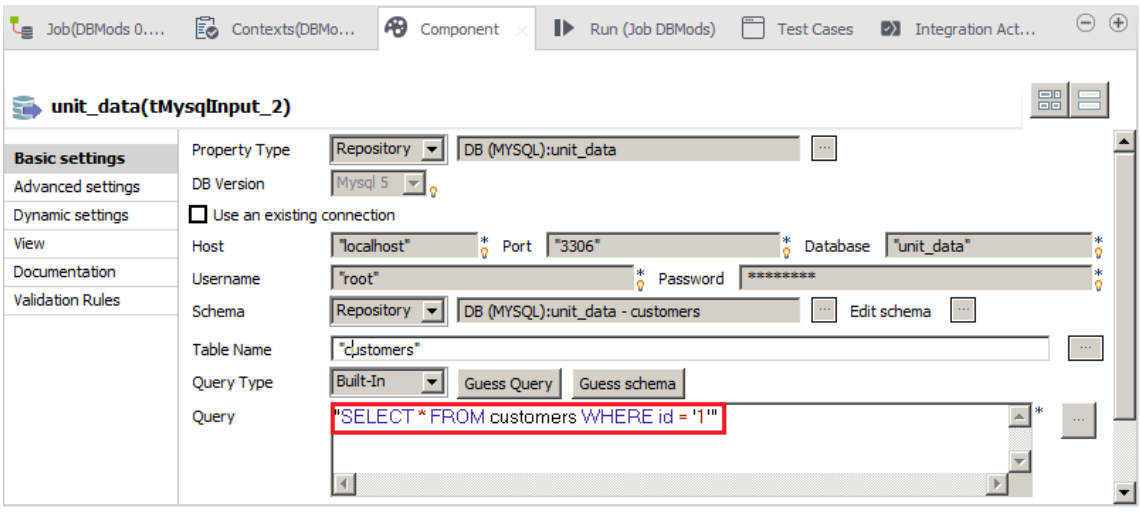


To update this record into the database, configure the **tMySQLOutput** component by double-clicking it. In the **Component** view, ensure that **Action on data** is set to *Update*.

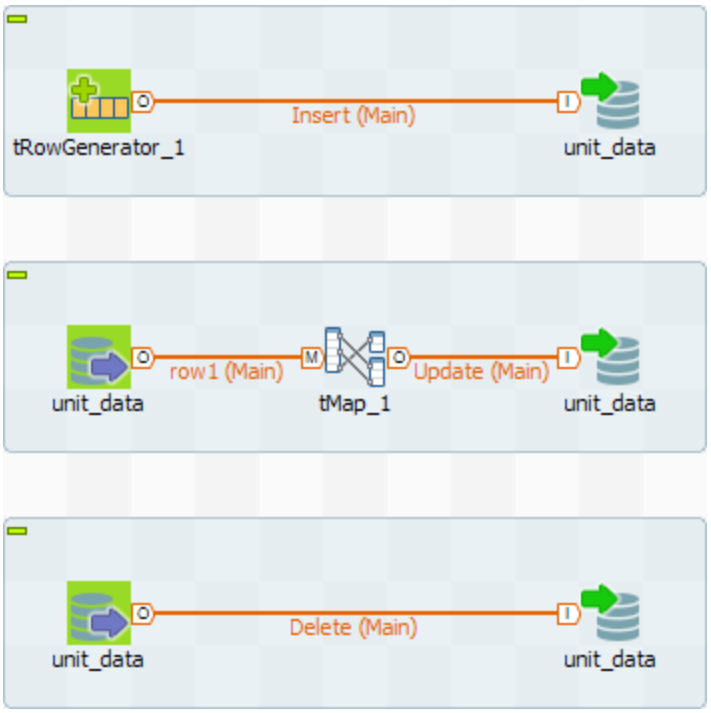


5. CREATE A DELETE SUBJOB

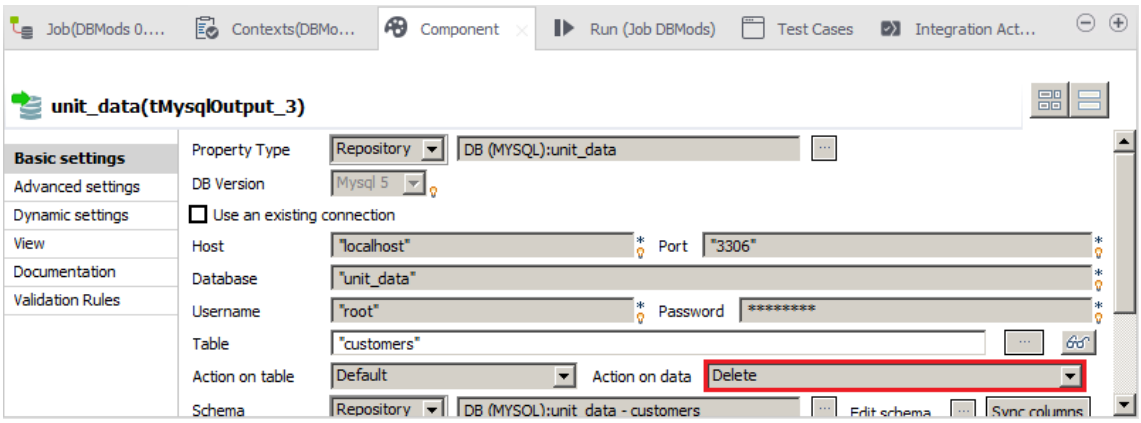
Select the last **tMysqlInput** component. In the **Component** view, replace the text in the **Query** box with *"SELECT * FROM customers WHERE id = '1'"*. As before, this will return only a single row.



Connect this component to the associated **tMysqlOutput** component with a **Main** row. Name this connection *Delete* .

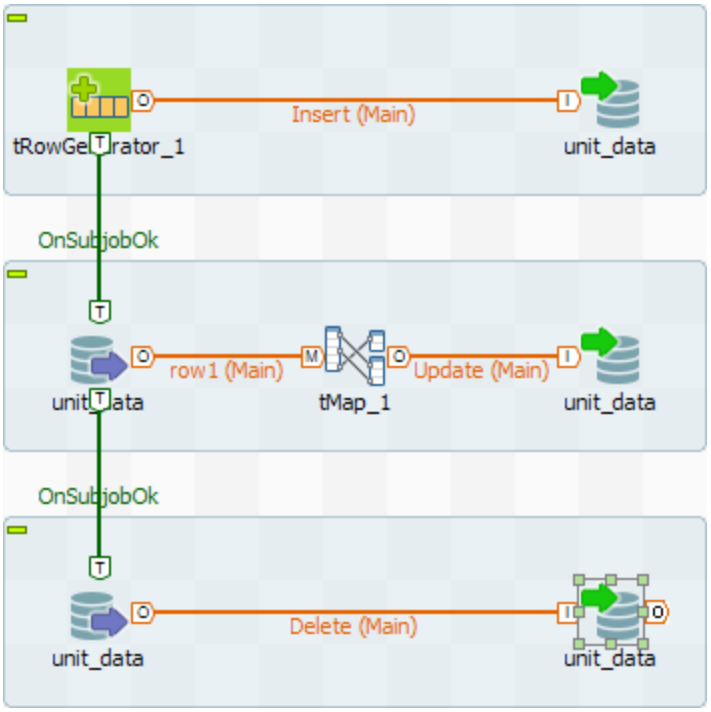


To delete this record from the database, configure the **tMySQLOutput** component by double-clicking it. In the **Component** view, ensure that **Action on data** is set to *Delete*.



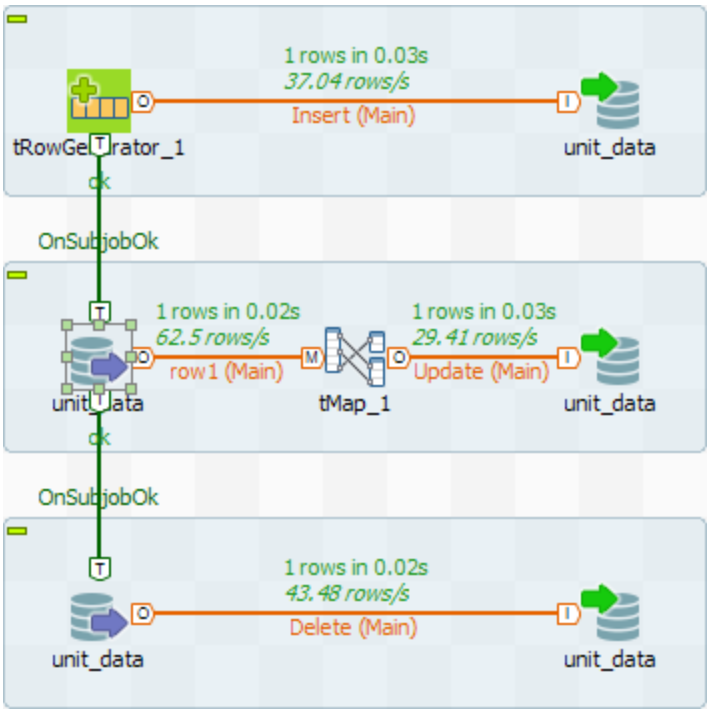
6. CONNECT THE SUBJOBS

   Connect the subJobs with **On Subjob Ok** triggers.



7. RUN THE JOB

   Run the Job. Each subJob should process a single row.



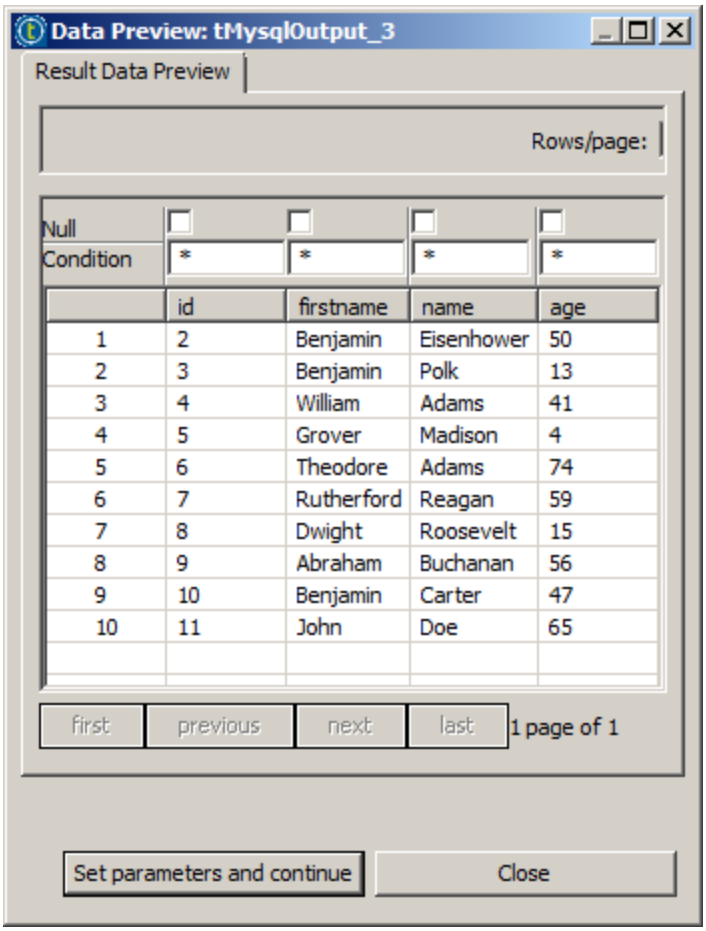
WARNING:
If you see any other result, you will need to follow the instructions in Resetting the Databases before making corrections and continuing

Examine the CDC Database

Now you will look at the CDC database to see if the modifications made to **unit_data** were detected and logged.

1. EXAMINE THE TABLE IN THE UNIT_DATA DATABASE

   Right-click the **tMysqlOutput** component in the last subJob and select **Data viewer**.



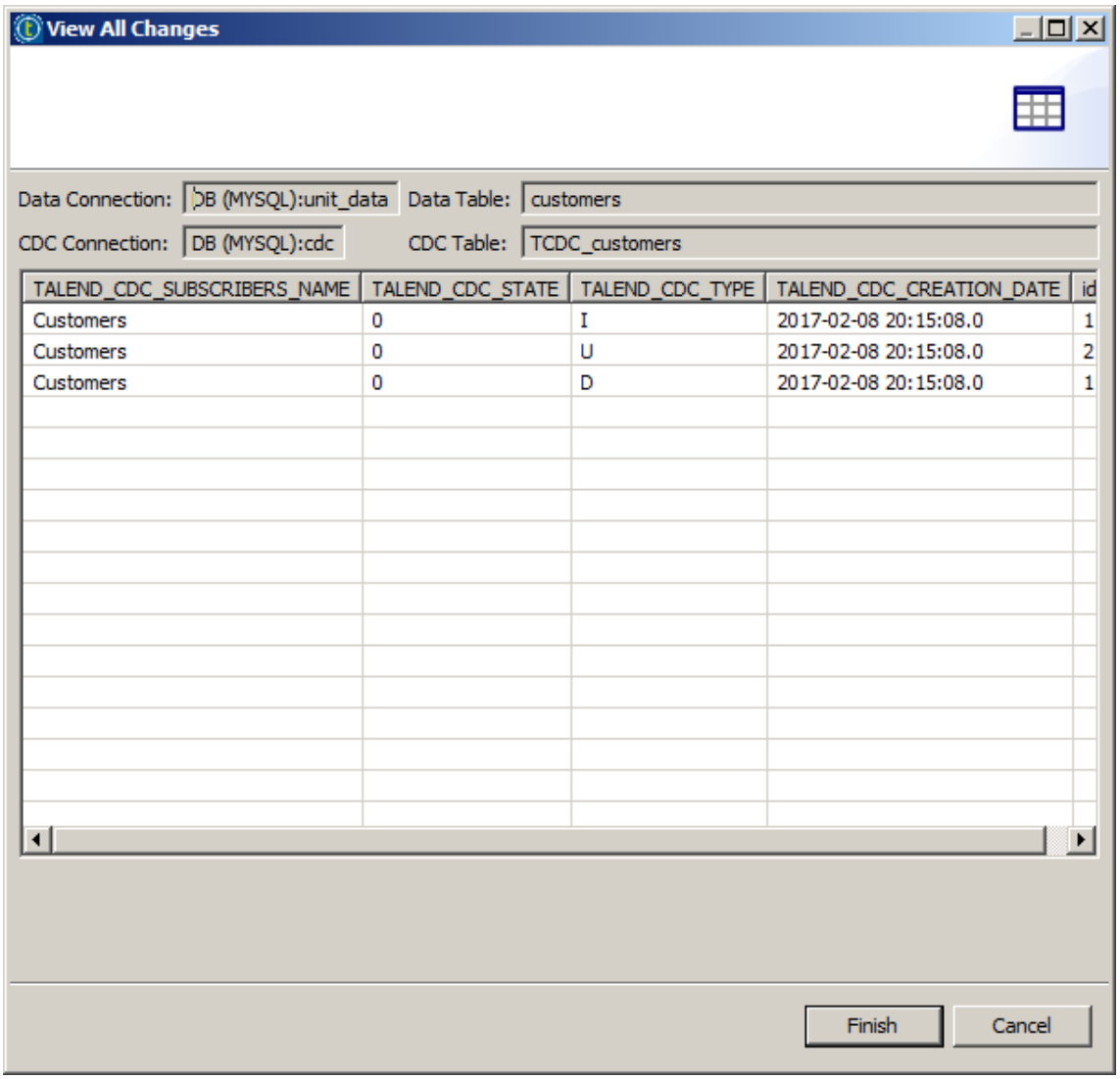
Referring back to the previous snapshot of the data, notice three changes:

- Record with **id** 1 is gone
- Record with **id** 2 has an updated age and is now one year older
- Record with **id** 11 is new

Click **Close**.

2. EXAMINE THE CDC TABLE

In the **Repository**, right-click **Metadata > Db Connections > CDC > unit_data > Table schemas > customers** and select **View All Changes**.



Notice the list of three records, each tracking one of the changes that resulted from the Job you just ran. The **id** number for each record specifies which record was changed. That is, record id 11 was inserted, id 2 was updated, and id 1 was deleted.

Again, if you see a different result, you will need to follow the instructions in Resetting the Databases before making corrections and continuing.

Click **Finish** you are through.

3. EXAMINE THE CDC TABLE USING MYSQL WORKBENCH

You can also check the CDC table directly using **MySQL Workbench**. Execute the following query *select * from tcdc_customers;* and observe the results, which match those from the previous step.

	TALEND_CDC_SUBSCRIBERS_NAME	TALEND_CDC_STATE	TALEND_CDC_TYPE	TALEND_CDC_CREATION_DATE	id
▶	Customers	0	I	2017-02-08 20:15:08	11
	Customers	0	U	2017-02-08 20:15:08	2
	Customers	0	D	2017-02-08 20:15:08	1

Now that you have made changes to the table in **unit_data** and verified that those changes were tracked in the CDC database, you can build a Job that will use the **cdc** database to update the **warehouse** database. Once completed, the **unit_data** and **warehouse** databases will contain identical data again.

Updating a Warehouse

Overview

The CDC database table now contains information about changes made to the **customers** table in the **unit_data** database. Now you will build a Job to update the **warehouse** database table using the CDC table. Once run, the **unit_data** and **warehouse** database tables will contain identical data.

Update the Warehouse

1. CREATE A JOB

   Create a new standard Job named *SyncWarehouse*.

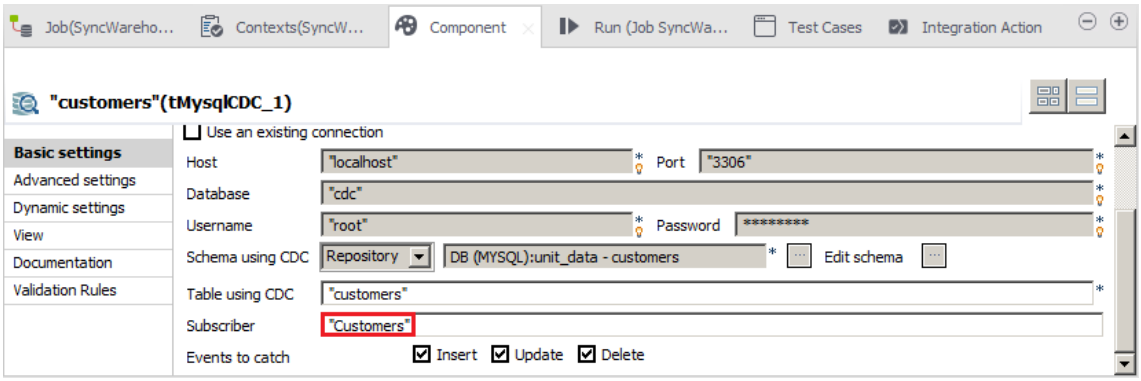
2. ADD AN INPUT COMPONENT

   From the **Repository**, click and drag **Metadata > Db connections > CDC > unit_data > Table schemas > customers** onto the **Designer**. Select **tMysqlCDC** from the **Components** list and click **OK**.

   NOTE:
   The **tMysqlCDC** component extracts changed data from the subscribed table and makes it available for processing. Click on the component and press **F1** for more information.

   Open the **Component** view and enter *"Customers"* into the **Subscriber** box.

   TIP:
   Recall that *Customers* is the name you used when creating the CDC subscriber.



3. ADD A tMap COMPONENT

   Place a **tMap** component to the right of the **tMysqlCDC** component.

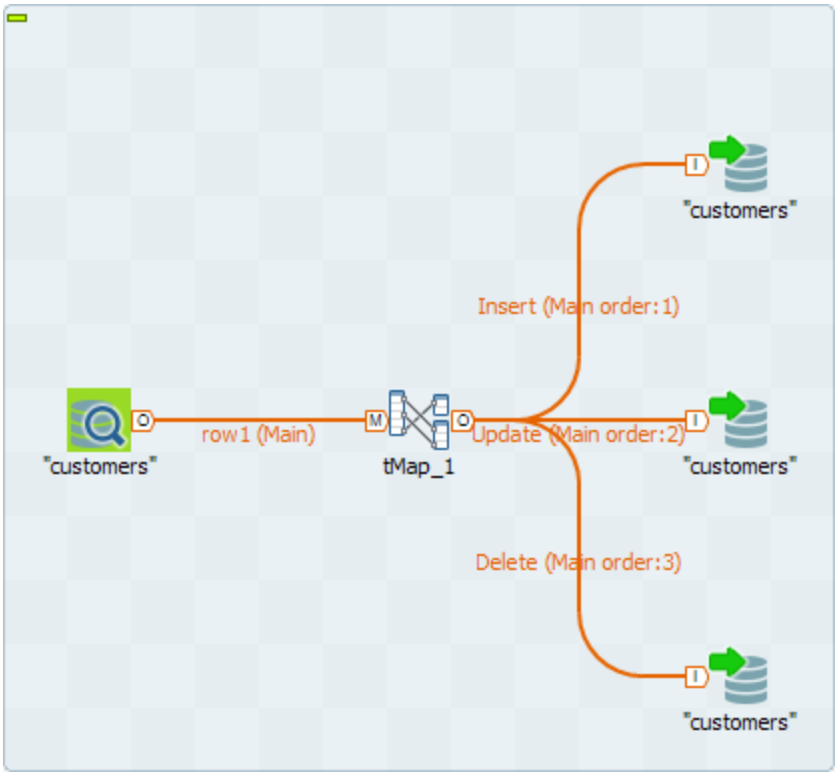
   Connect the **tMysqlCDC** to it using a **Main** row.

4. ADD AND CONFIGURE THREE OUTPUT COMPONENTS

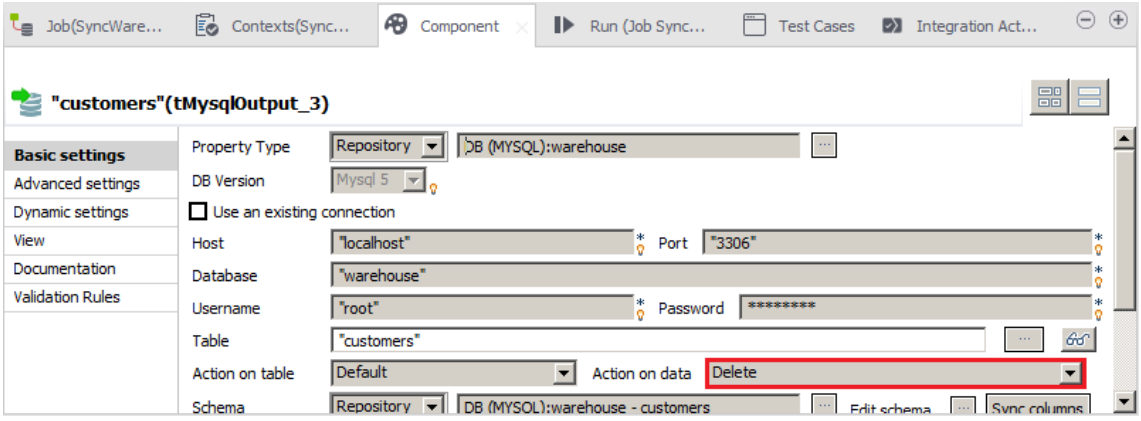
   From the **Repository**, click and drag **Metadata > Db connections > CDC > warehouse > Table schemas > customers** onto the **Designer**. Select **tMysqlOutput** from the **Components** list and click **OK**.

   WARNING:
   Be sure to use the **warehouse** schema in this step, not the **unit_data** schema you worked with previously.

Paste two additional copies of this output component, for a total of three. Each one of these components will be relegated to handling one of the three database operations: insert, update, and delete. As such, connect the **tMap** to these three output components with three new output rows named *Insert*, *Update*, and *Delete*. Each time, when prompted to get the schema of the target, select **Yes**.

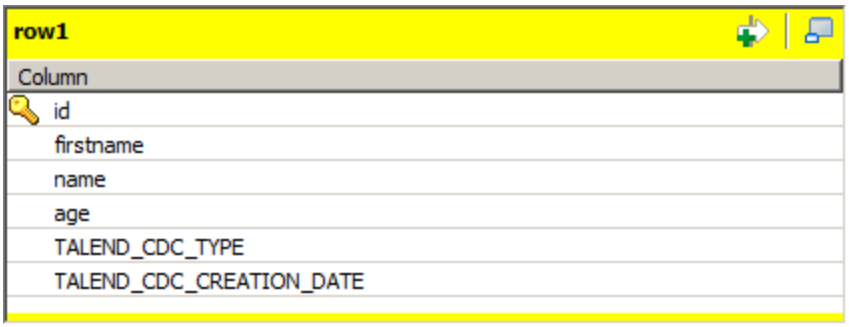


For each of these output components, refer to the **Component** view and set the **Action on data** parameter to **Insert**, **Update**, and **Delete**, respectively. Refer to the figure below for an example of setting this parameter.

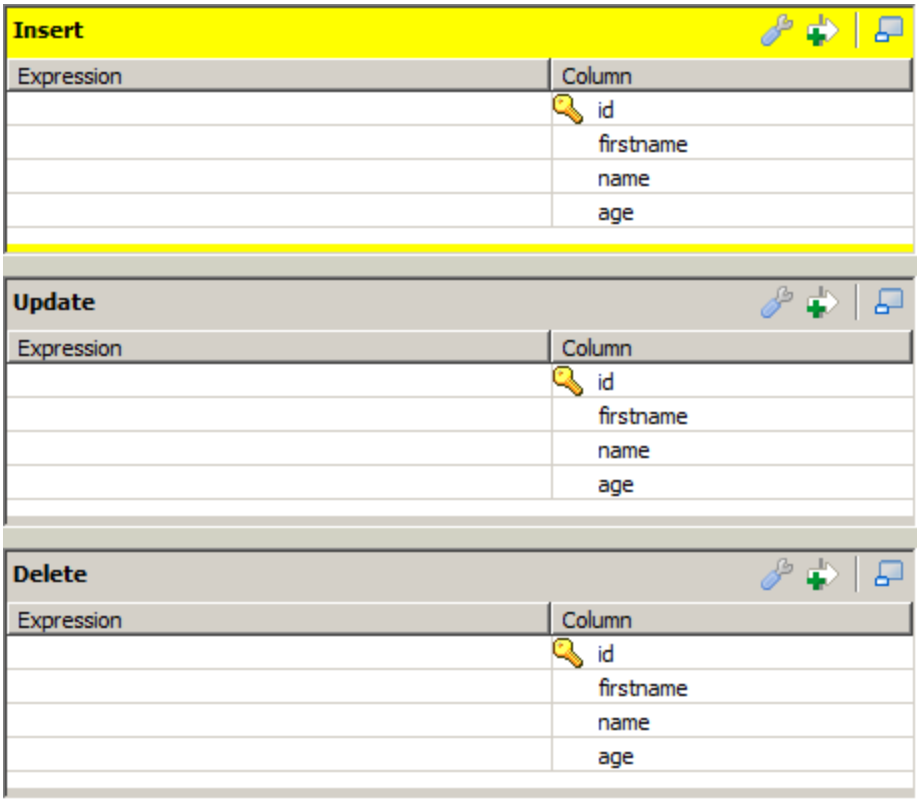


5. CONFIGURE THE tMap COMPONENT

Double-click the **tMap** component to open the mapping editor. Notice that the input schema **row1** contains the columns you examined earlier from **MySQL Workbench** for the **cdc** database. This is derived from the input component, which you created from schema metadata at the beginning of this section.



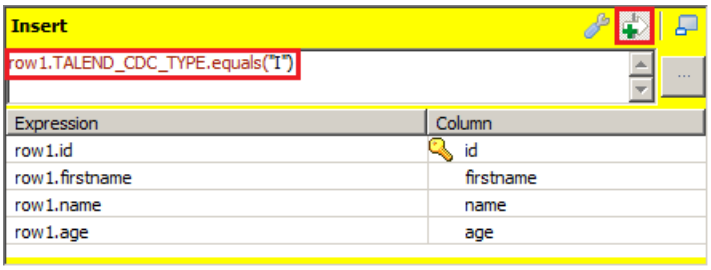
Notice also the three output tables, created as a result of the links to the three input components you created earlier.



Click the **Auto map!** button above the output schema tables. This maps all columns from the input schema table to the **Expression** field for each of the matching columns in the output schema tables.

Then, begin applying filters to each output table. Click the **Activate expression filter** icon ( ) in the upper right of the **Insert** output table. Drag **TALEND_CDC_TYPE** from the **row1** input table to the expression box. Append *.equals("I")* to the expression, so that the final expression reads:

```
row1.TALEND_CDC_TYPE.equals("I")
```



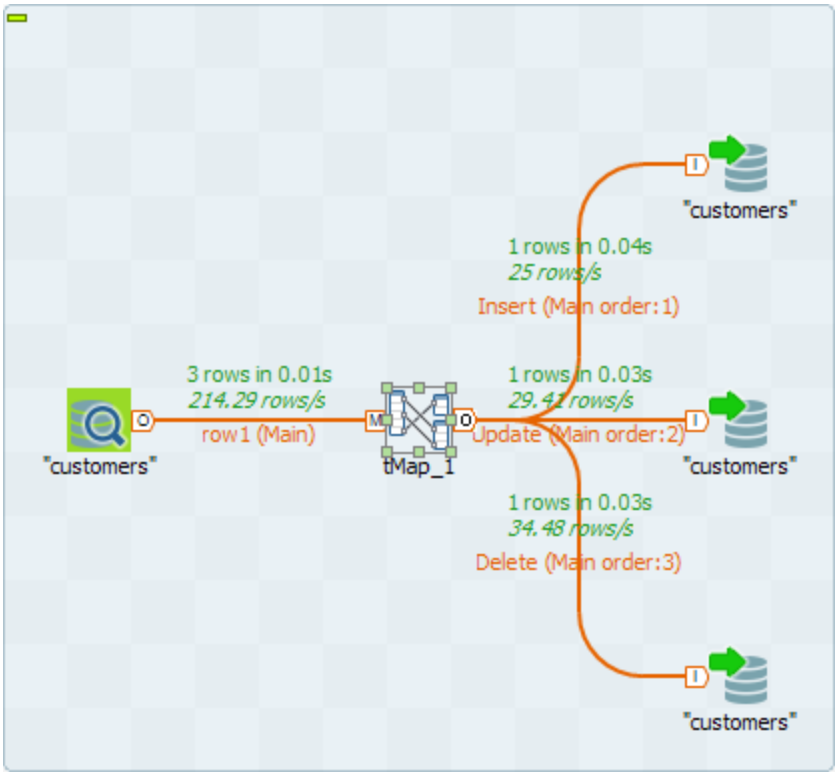
Repeat this step for the other two output tables, replacing *"I"* with *"U"* and *"D"* for the **Update** and **Delete** tables, respectively. Now, rows retrieved by the CDC component from the **unit_data** database will be sent to different output flows depending on the type of change the row represents.

Click **OK** to save the **tMap** configuration.

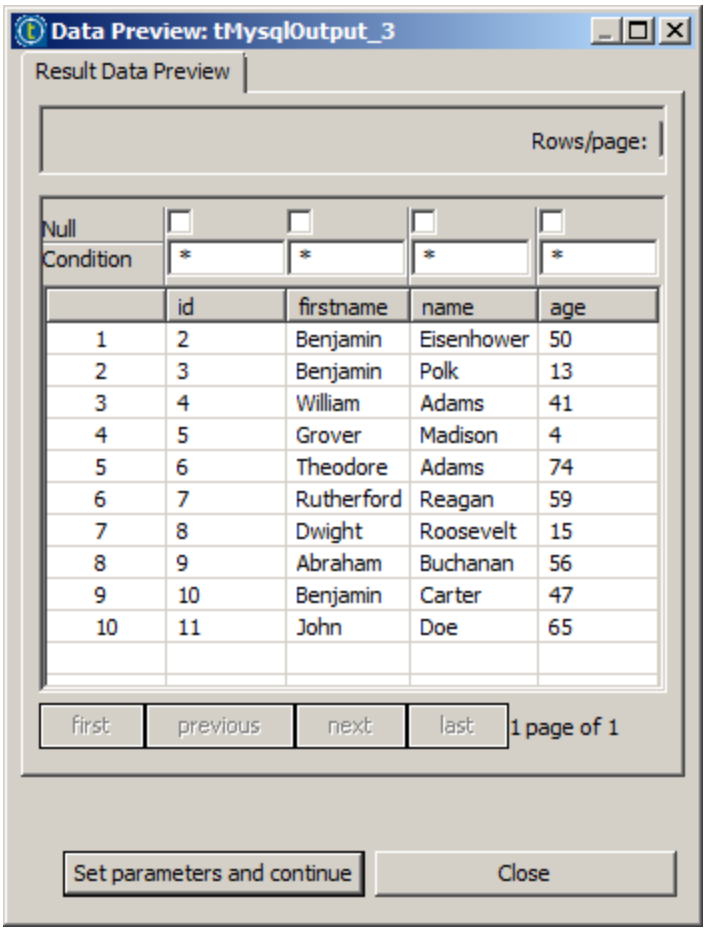
Run the Job

1. RUN THE JOB

   Run the Job. The statistics show one row processed by each output component.



Examine the **warehouse** table with the **Data viewer** by right-clicking one of the output components and selecting **Data viewer**. Verify that it now matches the table in **unit_data**.



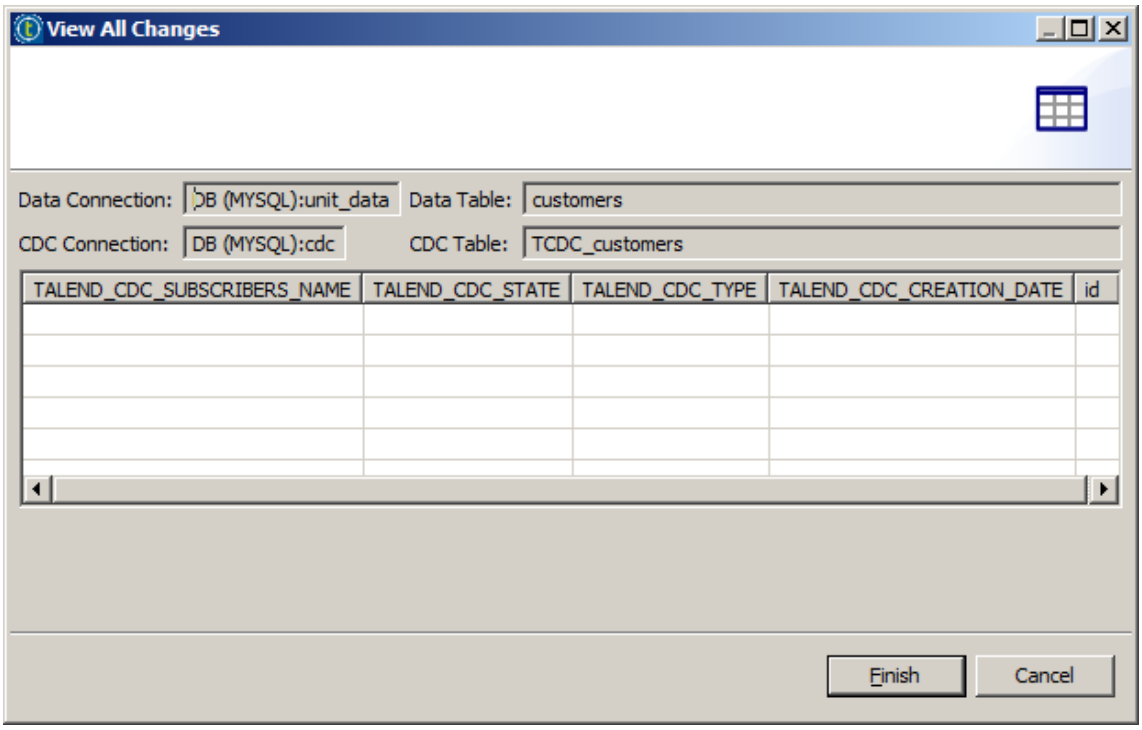
In summary, the **warehouse** table is identical to the **unit_data** table now:

- The record with id 1 was deleted
- The record with id 2 was updated
- A new record with id 11 was added

Click **Close** to close out the **Data viewer**.

2. EXAMINE THE CDC TABLE

   Right-click **Metadata > Db connections > CDC > unit_data > Table schemas > customers** and select **View All Changes**. Notice that the records are gone, now that the changes have been committed.



NOTE:
If your results differ in any way, you will need to follow the procedure in Resetting the Databases before attempting to make corrections.

You have now completed this lesson and can move on to the Challenge.

Challenge

Overview

Complete these exercises to further explore the use of Change Data Capture. See Solutions for possible solutions to these exercises.

Multiple Changes

Create and then run a new Job that modifies the record with id 8 in **unit_data** three separate times: first to change the age to 24, then to change the first name to *"Daphne"*, then to change the age to 28. Use **MySQL Workbench** to examine the change records in the **cdc** database and then run the Job **SyncWarehouse**. How many records are processed? Ensure that the end result is what you expect.

Insert and Delete

Create and then run a new Job that inserts a new record into the **unit_data** table, modifies it, and then deletes it.

> TIP:
> You may find it easier to begin by duplicating the Job **DBMods**.

Use **MySQL Workbench** to examine the change records in the **cdc** database.

Finally, duplicate the Job **SyncWarehouse.** Modify the output of the **tMap** component and the actions in each **tMysqlOutput** component to:

- insert the new record
- modify the record
- delete the record

Run the Job. How many records are processed? Ensure that the end result is what you expect.

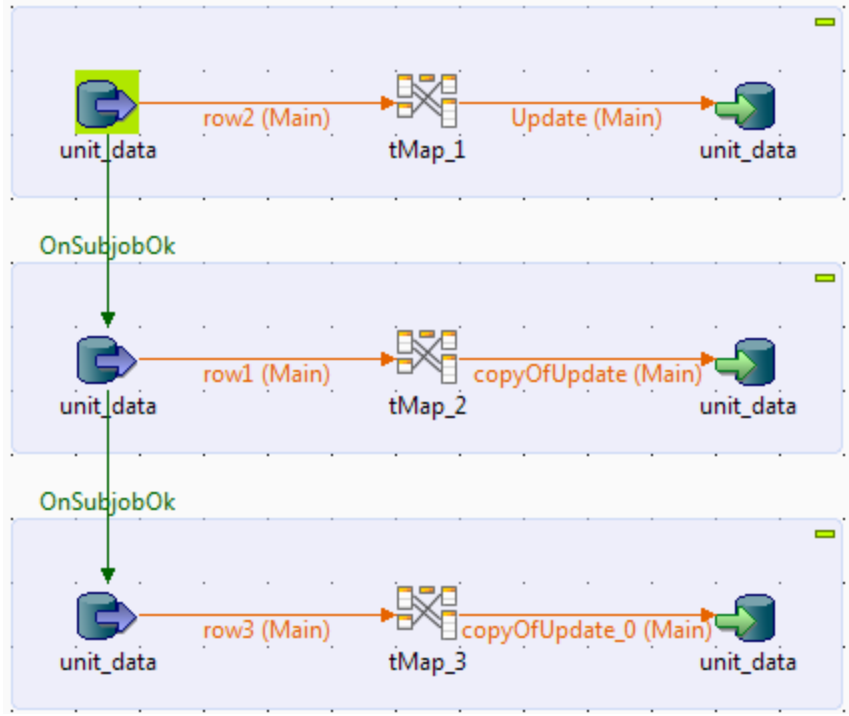
Solutions

Overview

These are possible solutions to the Challenge. Note that your solutions may differ and still be valid.

Multiple Changes

The following Job contains three Subjobs, each making one of the modifications:



Each **tMysqlInput** component is configured with the query:

```
"SELECT * FROM customers WHERE id = '8'"
```

Each Subjob is a variation on the center subjob in the Job **DBMods**, with the **tMap** component configured to make each individual modification while mapping the remaining columns as-is.

After running this Job, MySQL Workbench (or **View All Changes** from the **Repository**) shows three records in the **tcdc_customers** table :

Customers	0	U	2013-02-04 13:59:25
Customers	0	U	2013-02-04 13:59:25
Customers	0	U	2013-02-04 13:59:26

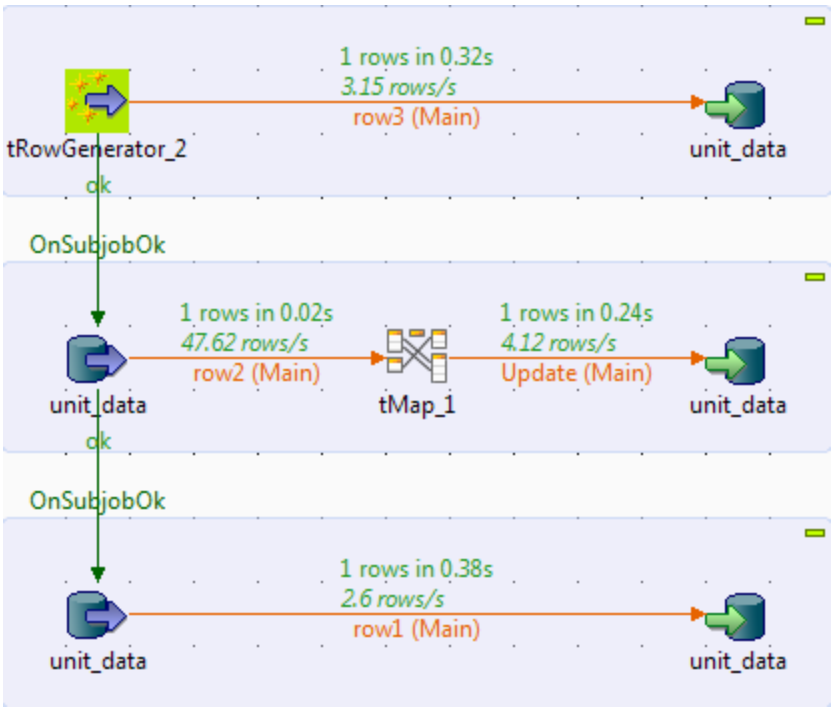
Running the Job **SyncWarehouse** processes a single update. Even though the record changed three times, the Job only needed to update the warehouse with the final, current state of the record.

The data viewer for **warehouse** shows that the affected record has the expected changes:

6	8	Daphne	Roosevelt	28

Insert then delete

The following Job inserts, modifies, and then deletes the same record:



This is basically a re-ordered variation on the Job **DBMods**, with the insert Subjob first.

The **tRowGenerator** component creates a record with id 12.

The two **tMysqlInput** components use the query:

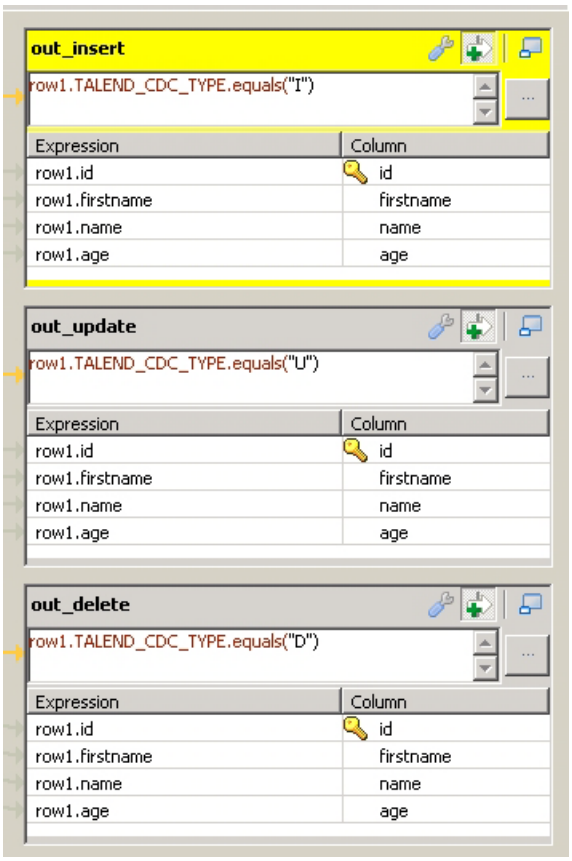
```
"SELECT * FROM customers WHERE id = '12'"
```

After running this Job, MySQL Workbench (or **View All Changes** from the **Repository**) shows three records in the **tcdc_customers** table:

TALEND_CDC_SUBSCRIBERS_NAME	TALEND_CDC_STATE	TALEND_CDC_TYPE	TALEND_CDC_CREATION_DATE	id
Customers	0	I	2013-02-04 14:10:14	13
Customers	0	U	2013-02-04 14:10:14	13
Customers	0	D	2013-02-04 14:10:14	13

Duplicate the Job **SyncWarehouse**.

Double-click **tMap** to open the editor and then modify the filters and the output names to filter "I", "U" and "D" values in the modifications list saved in the **tcdc_customers** table.



Double-click the first **tMysqlOutput** component to open the Component view. Change the **Action on Data** to **Insert**. In the second **tMysqlOutput** component, change the **Action on Data** to **Update**. Finally, set the **Action on Data** to **Delete** for the last **tMysqlOutput** component.

Run your Job, and check that the Job processes one of each type of change. The end result as displayed in the **Data viewer** is no change to the table contents:

6	8	Daphne	Roosevelt	28
7	9	Abraham	Buchanan	56
8	10	Benjamin	Carter	47
9	11	New	Person	40

Wrap-Up

In this lesson, you started with two databases containing exactly the same records in matching tables. You used the Data Viewer and stored queries to examine the contents of database tables from within Talend Studio. You then configured a third database to act as the CDC foundation for one of the databases, tracking any changes made to a customer table in the **unit_data** database. After creating a Job to make several changes to the table, you finally built a Job that used the information in the CDC database table to update the **warehouse** database with the changes from the CDC table.

Next step

Congratulations! You have successfully completed this lesson. To save your progress, click **Check your status with this unit** below. To go to the next lesson, on the next screen, click **Completed. Let's continue >**.

Resetting the Databases

Overview

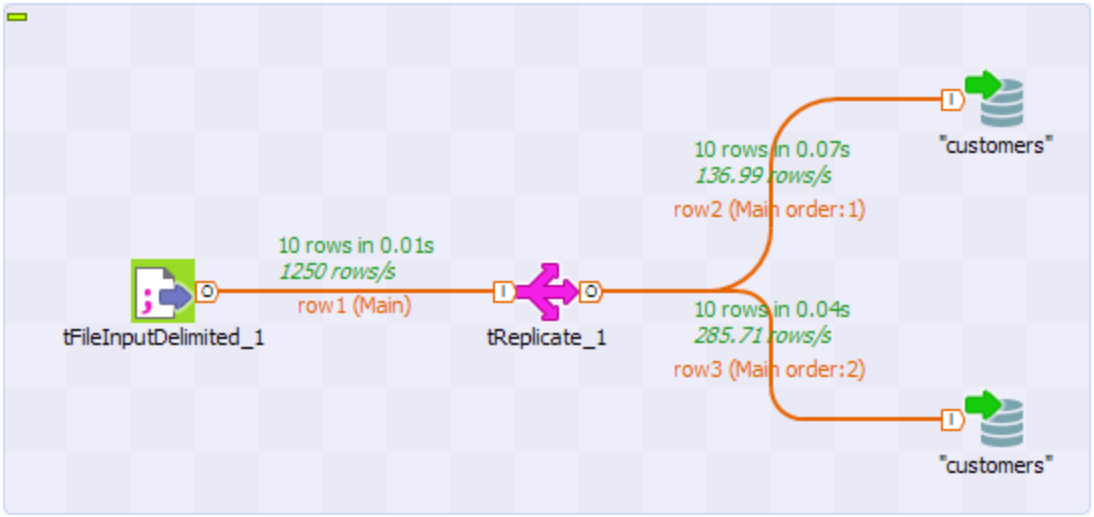
Because this lesson involves tracking changes to a database, a simple mistake can make it difficult to complete the exercises, since both the mistake and your corrections are tracked in the CDC database. The steps listed here allow you to reset the databases to their original state and clear out the information in the CDC database.

Reset Table Contents

1. RESET TABLES

   Import the **ResetDatabasesForCDC** Job from *C:/StudentFiles/DIAdvanced/ResetDatabasesForCDC.zip*, if you do not already have it imported into your **Repository**.

   Open the Job and run it.

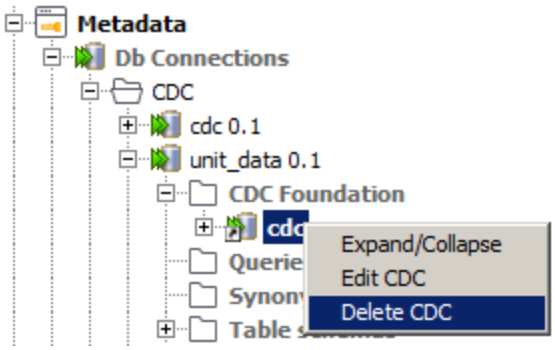


   This Job drops the current contents of the **customers** table in both the **unit_data** and **warehouse** databases, and loads them with the original data from the file *C:\StudentFiles\DIAdvanced\CustDataOrig.csv*.

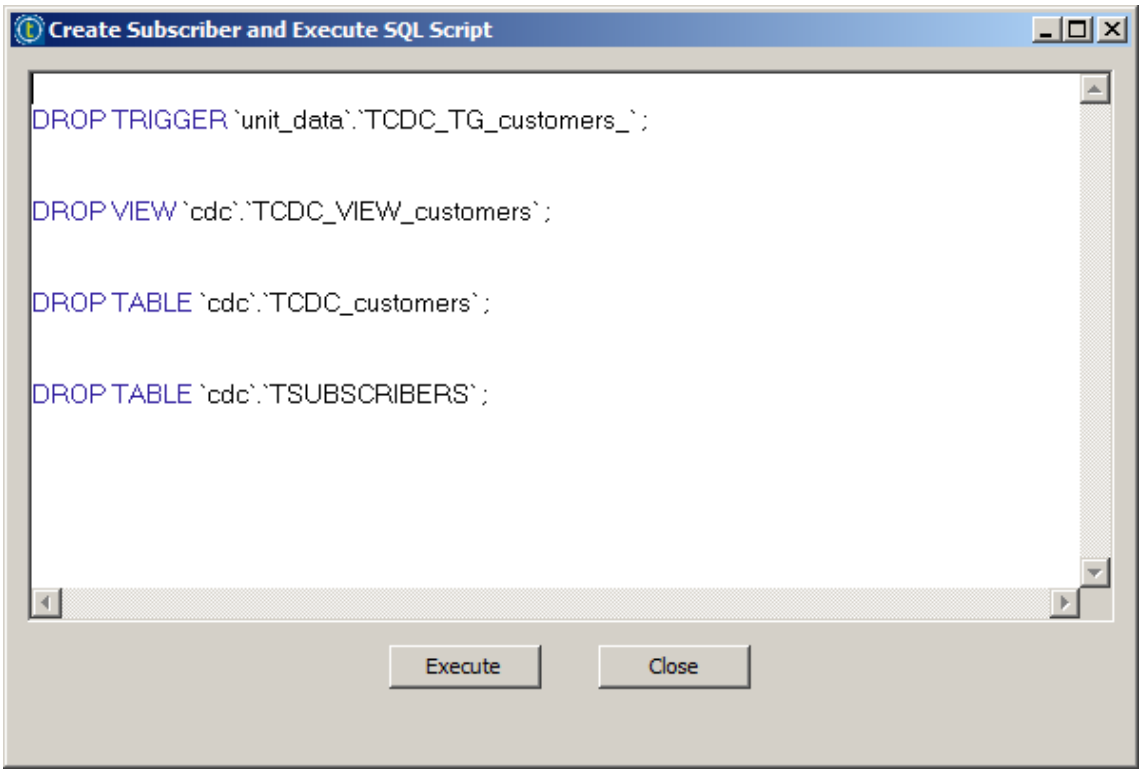
2. RESET CDC

   Since you have CDC tracking on, the changes made in the previous step are now in the **cdc** database. To reset the **cdc** table, you need to drop it.

   In the **Repository**, right-click **Metadata > Db Connections > CDC > unit_data > CDC foundataion > cdc** and select **Delete CDC**:



   A window appears that shows the SQL statements that will be run to delete the CDC. Click **Execute**.



Click **Ignore** for any warning windows that appear.

In the **Success** window, click **OK**, then click **Close**.

3. RESET CDC

   Repeat the steps in Configuring the CDC Database. Then, continue on with the next steps depending on what led up to the failure.

Continue

If your error occurred while making changes to **unit_data**, carefully go over the steps in Monitoring Changes and make any necessary corrections before running the Job again and then continuing.

If your error occurred while updating **warehouse**, run the Job **DBMods** again to make the changes, then carefully go over the steps in Updating the Warehouse and make any necessary corrections before running the Job again.

APPENDIX

Additional Information

NOTE:
If the links below yield no search results, please make sure the language filter is set to English.

- Online Documentation - Talend Help Center
- Guides that are most helpful for additional information covered in this course:
  - Talend Studio User Guide
  - Talend Component Reference Guide
  - Talend Administration Center User Guide
- Helpful courses:
  - *Talend Data Integration Basics*
  - *Talend Data Intregration Administration*
- Lesson 1 - Remote Repository
  - Talend Administration Center User Guide
- Lesson 2 - SVN in the Studio
  - SVN website - https://subversion.apache.org/
  - Talend Administration Center User Guide
- Lesson 3 - Remote Job Execution
  - Talend Data Integration User Guide - How to run a Job remotely
- Lesson 4 - Resource Usage and Basic Debugging
  - Talend Studio User Guide - Handling Job Execution
  - Eclipse website https://www.eclipse.org
- Lesson 5 - Activity Monitoring Console (AMC)
  - Talend Activity Monitoring Console User Guide
- Lesson 6 - Parallel Execution
  - Talend Job Design Patterns and Best Practice: part 2 blog
    - See *Parallelization Options* section near the bottom
    - Entire blog (both part 1 and part 2) is recommended reading for developers
- Lesson 7 - Joblets
  - Talend Data Integration Studio User Guide - Designing a Joblet
- Lesson 8 - Unit Test
  - Talend Software Development Life Cycle - Best Practices Guide
- Lesson 9 - Change Data Capture (CDC)
  - Talend Data Integration Studio User Guide - Change Data Capture

This page intentionally left blank to ensure new chapters start on right (odd number) pages.