# DBA 3: Data Warehousing (PDF)

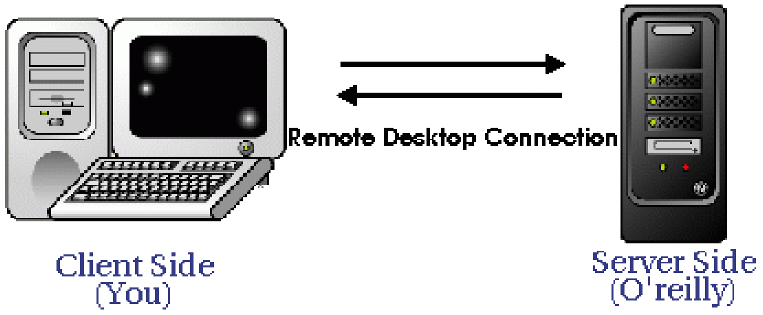
# Getting Started with Talend Open Studio DBA 3: Data Warehousing Lesson 0

## Getting Started with Talend Open Studio

**Note** These instructions will only show up here, at the beginning of this course. Although you probably won't need them instructions again, feel free to bookmark or print them out, just in case.

In this course we'll be using Talend Open Studio, an Eclipse based data warehousing tool. Our version of Talend Open Studio, or *TOS* is nearly identical to the version you can get for free from Talend's website. We've added a few features to enhance your learning experience. Our plug-in allows you to view the course, program your labs, submit your projects, and receive your grades and comments, all without leaving Talend Open Studio.

We will be using a Terminal Service session or thin client. A *thin client* allows you to connect to a remote terminal server running Talend Open Studio. The **terminal server** is a computer that **serves** a **desktop** to other computers via the network. You will be using this thin client to access Talend Open Studio on our Windows servers:



Your machine will send mouse and keyboard information to our server. Our server will send the resulting visual output back to your computer. **It will seem just like you are running Talend Open Studio on your own machine,** but in reality it is being run on our server and returned to your machine. We call our system the **Learning Sandbox**. The Learning Sandbox is a safe place where you can write and execute your own programs without worrying about breaking your own computer. It also gives you the ability to work from anywhere there's an internet connection. Since all of your work is stored on our server, there are no disks or USB drives to carry around.

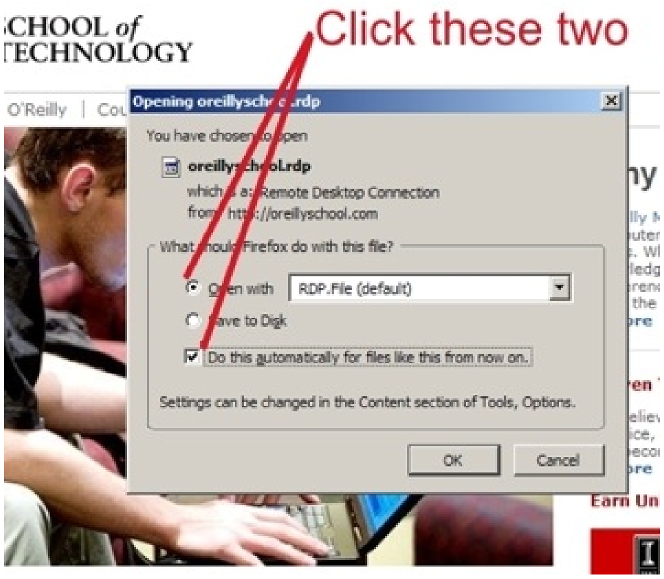
In a moment you will be asked to switch windows back to your student start page and to press the **Enter** button for your course:

After you follow the specific instructions for your machine, return to the student startup window.

### Connecting to the Sandbox - Windows

**These instructions are for Windows users only.**

Switch back to your student start page, and press the enter button for your course. You will see this:


Click these two
Opening oreillyschool.rdp
You have chosen to open
oreillyschool.rdp
which is a: Remote Desktop Connection
from: http://oreillyschool.com
What should Firefox do with this file?
Open with RDP.File (default)
Save to Disk
Do this automatically for files like this from now on.
Settings can be changed in the Content section of Tools, Options.
OK
Cancel

Make sure the **Open with** radio button is selected. If you like, you can also check the **Do this automatically for files like this from now on** checkbox. Now click **OK**.

## Connecting to the Sandbox - Macintosh

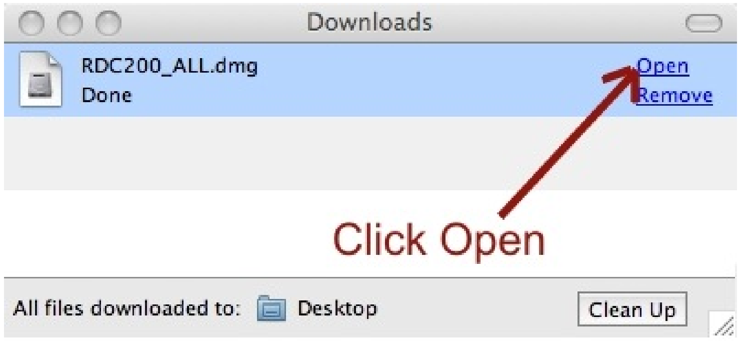
**These instructions are for Macintosh users only.**

You need to install the Remote Destkop Connection software on your computer.

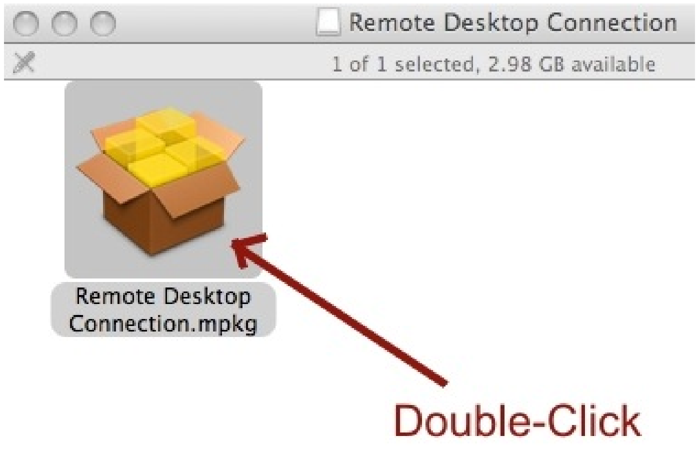
**Note** The file names and screen shots below may differ on your own computer. Microsoft occasionally updates the RDC program.

Start by downloading this file

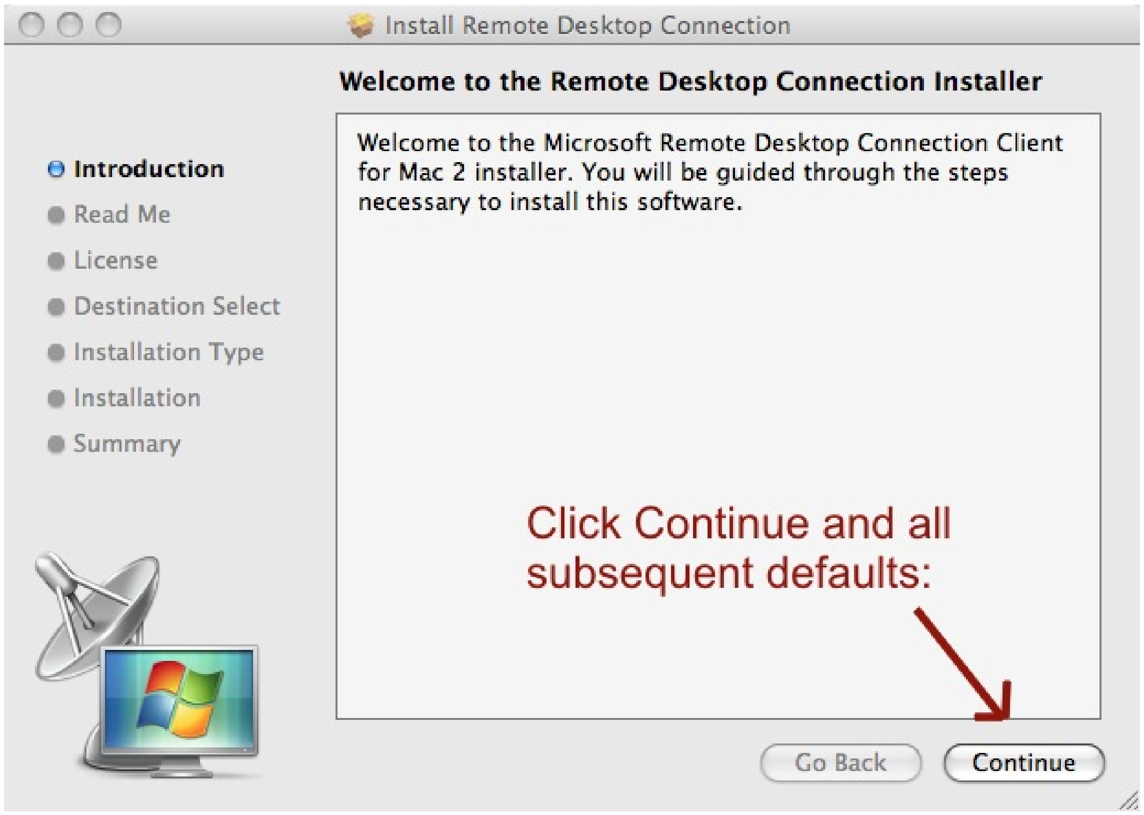
Once you've downloaded that RDC200_ALL.dmg (disk image) you need to locate it and open it.



Next you will see the following screen. Double click on the box and follow the install instructions.


Remote Desktop Connection
1 of 1 selected, 2.98 GB available
Remote Desktop Connection.mpkg
Double-Click

Click "Continue" and all the default recommendations and buttons for each screen:

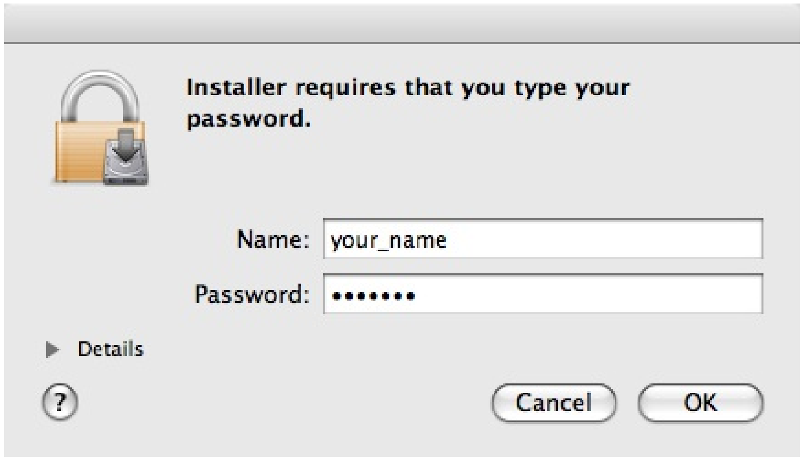


**Welcome to the Remote Desktop Connection Installer**

Welcome to the Microsoft Remote Desktop Connection Client for Mac 2 installer. You will be guided through the steps necessary to install this software.

Click Continue and all subsequent defaults:

You'll be asked to give your login and password **TO YOUR MACHINE:**

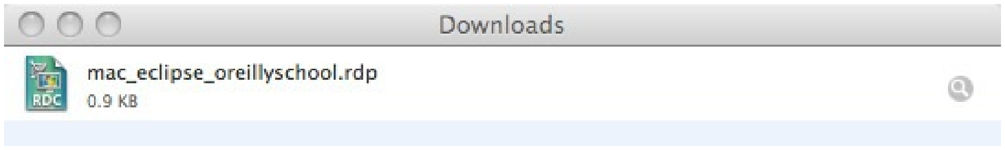


**Installer requires that you type your password.**

Name: your_name

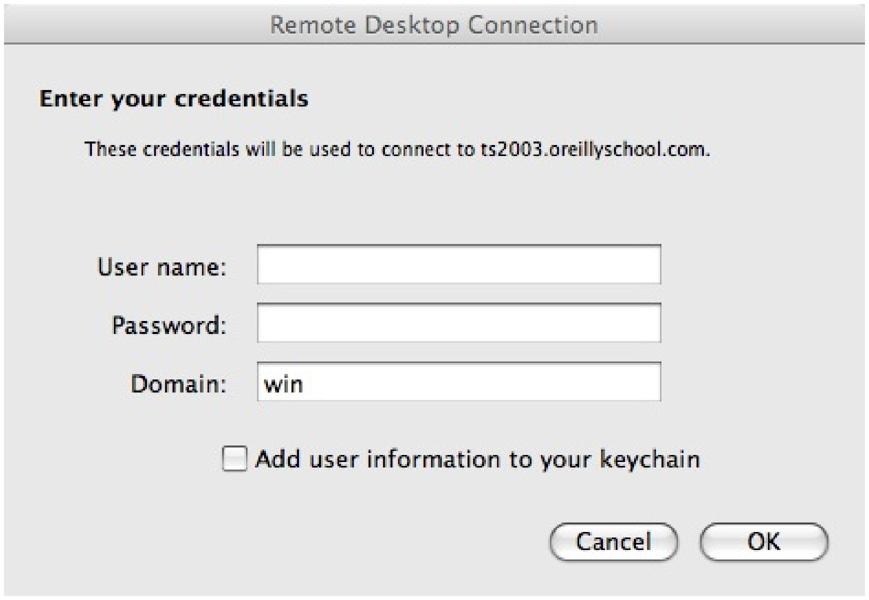
Password: •••••••

Switch back to your student start page, and press the enter button for your course. Your browser will download an RDP file. Save this file to your desktop.



mac_eclipse_oreillyschool.rdp
0.9 KB

Next, double click the RDP file to open it. You be asked for the USERNAME and PASSWORD for the O'Reilly School of Technology:



Remote Desktop Connection

**Enter your credentials**

These credentials will be used to connect to ts2003.oreillyschool.com.

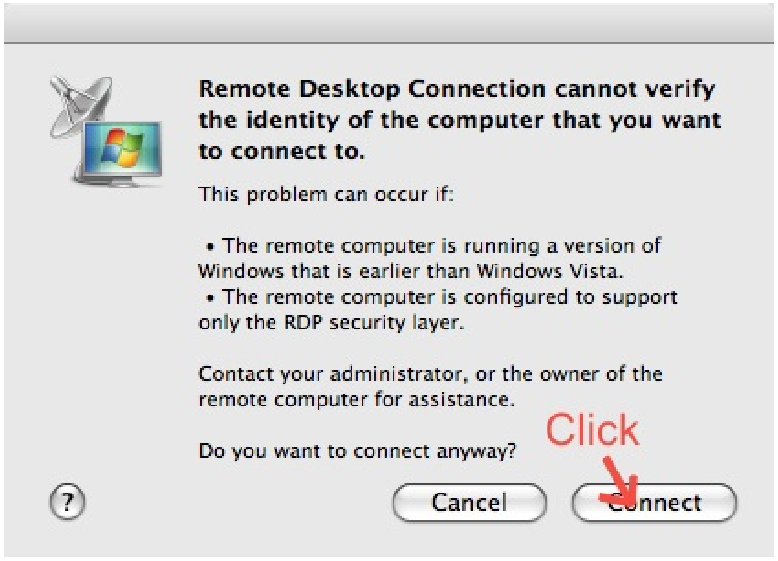
User name:

Password:

Domain: win

☐ Add user information to your keychain

If you see a warning sign, just click connect, it's just Microsoft trying to make people buy Vista.



**Remote Desktop Connection cannot verify the identity of the computer that you want to connect to.**

This problem can occur if:

• The remote computer is running a version of Windows that is earlier than Windows Vista.
• The remote computer is configured to support only the RDP security layer.

Contact your administrator, or the owner of the remote computer for assistance.

Do you want to connect anyway?

## Connecting to the Sandbox - Linux

**These instructions are for Linux users only.**

There are many distributions of Linux. Unfortunately, we do not have the resources to support them all. These instructions are geared towards the Ubuntu family of Linux distributions. If you are versed in Linux, you should be able to tailor these instructions to your specific distribution and computer setup.

In Ubuntu, you will need to make sure that Terminal Server Client is installed and functioning. Using the Synaptic or Adept package manager, simply install the Terminal Server Client. Or, from a terminal window, execute the following line

OBSERVE:

```
sudo apt-get install tsclient
```

Using the menu system for your Ubuntu or Kubuntu Linux distribution, start the Terminal Service Client application, or from a terminal window, execute the following line:

OBSERVE:

```
tsclient
```

Switch back to your student start page, and press the enter button for your course.

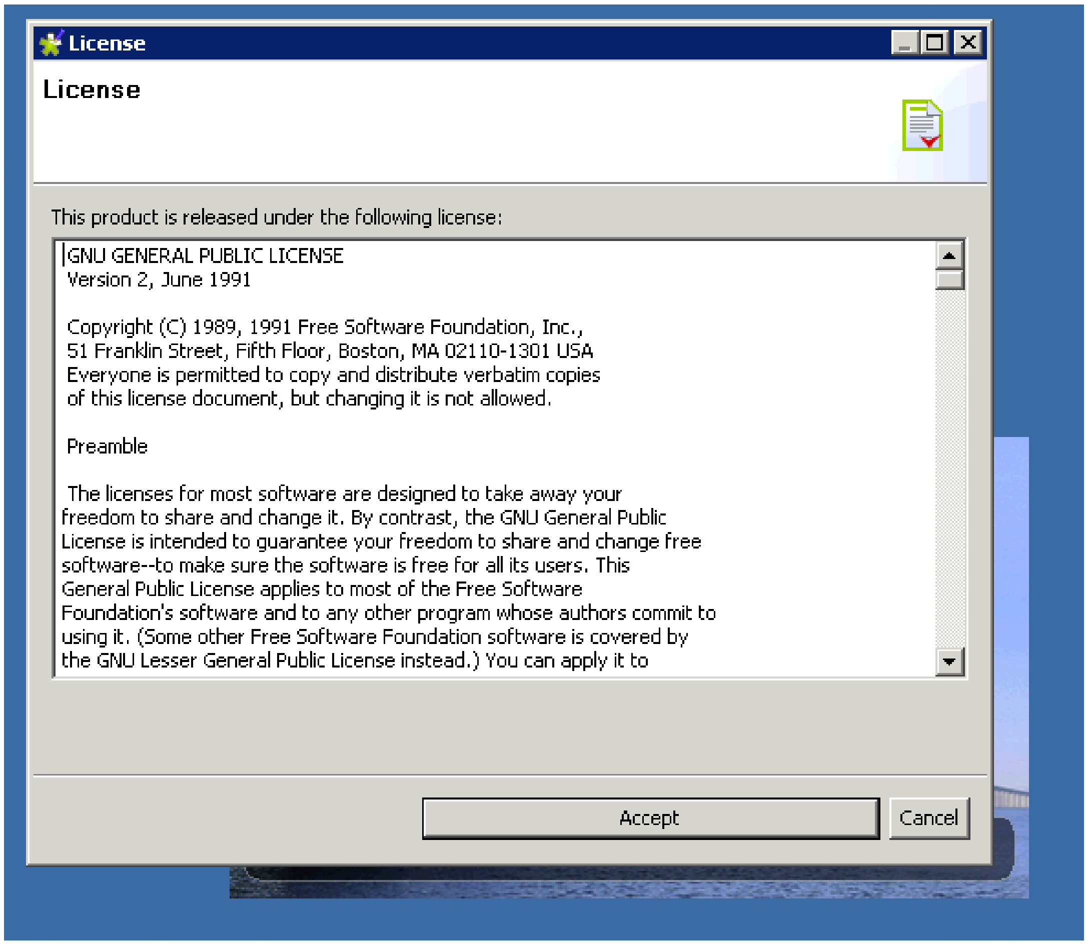
Your web browser will download an RDP file to your computer. Switch back to the Terminal Server Client program, and open the downloaded RDP file. Click **Connect**. You be asked for the USERNAME and PASSWORD for the O'Reilly School of Technology.

If you have any problems or questions, don't hesitate to contact your mentor at **learn@oreillyschool.com**.

## Initial Setup

At this point you should see the Talend splash screen:

The next screen you will see is the license acknowledgment. Click **Accept**:



License

This product is released under the following license:

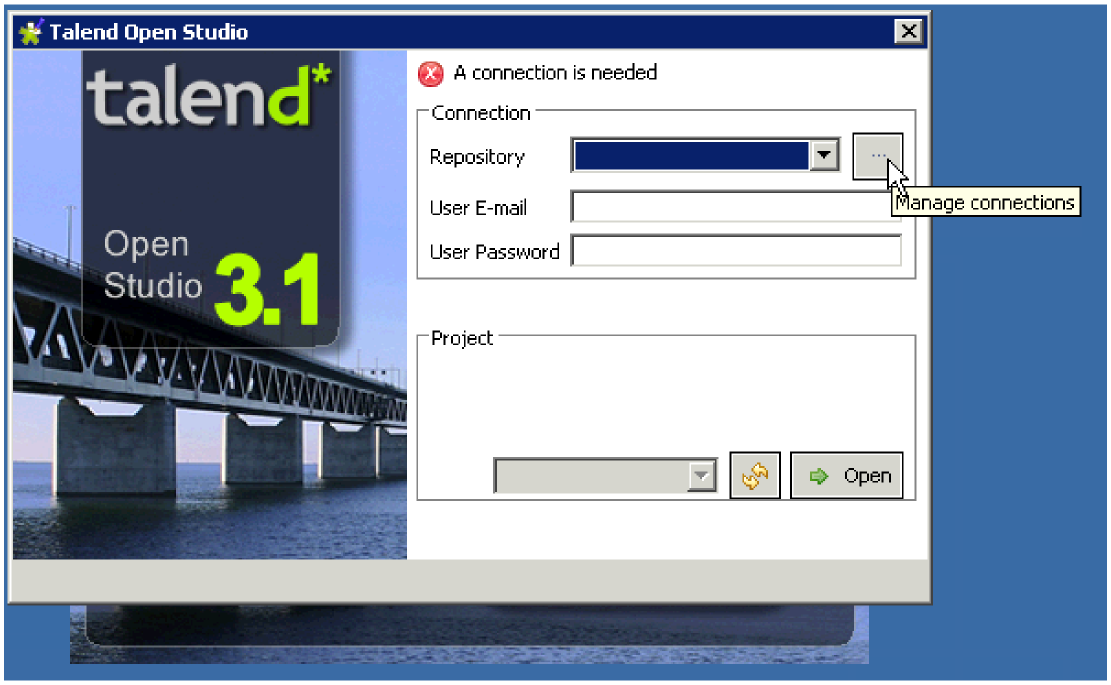
GNU GENERAL PUBLIC LICENSE
Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc.,
51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your
freedom to share and change it. By contrast, the GNU General Public
License is intended to guarantee your freedom to share and change free
software--to make sure the software is free for all its users. This
General Public License applies to most of the Free Software
Foundation's software and to any other program whose authors commit to
using it. (Some other Free Software Foundation software is covered by
the GNU Lesser General Public License instead.) You can apply it to

Before we can get started with TOS we need to setup a *repository connection* and then a product. The *repository* is where TOS will keep your objects. Click on the **...** button to manage the repository connection:


Talend Open Studio

A connection is needed

Connection

Repository

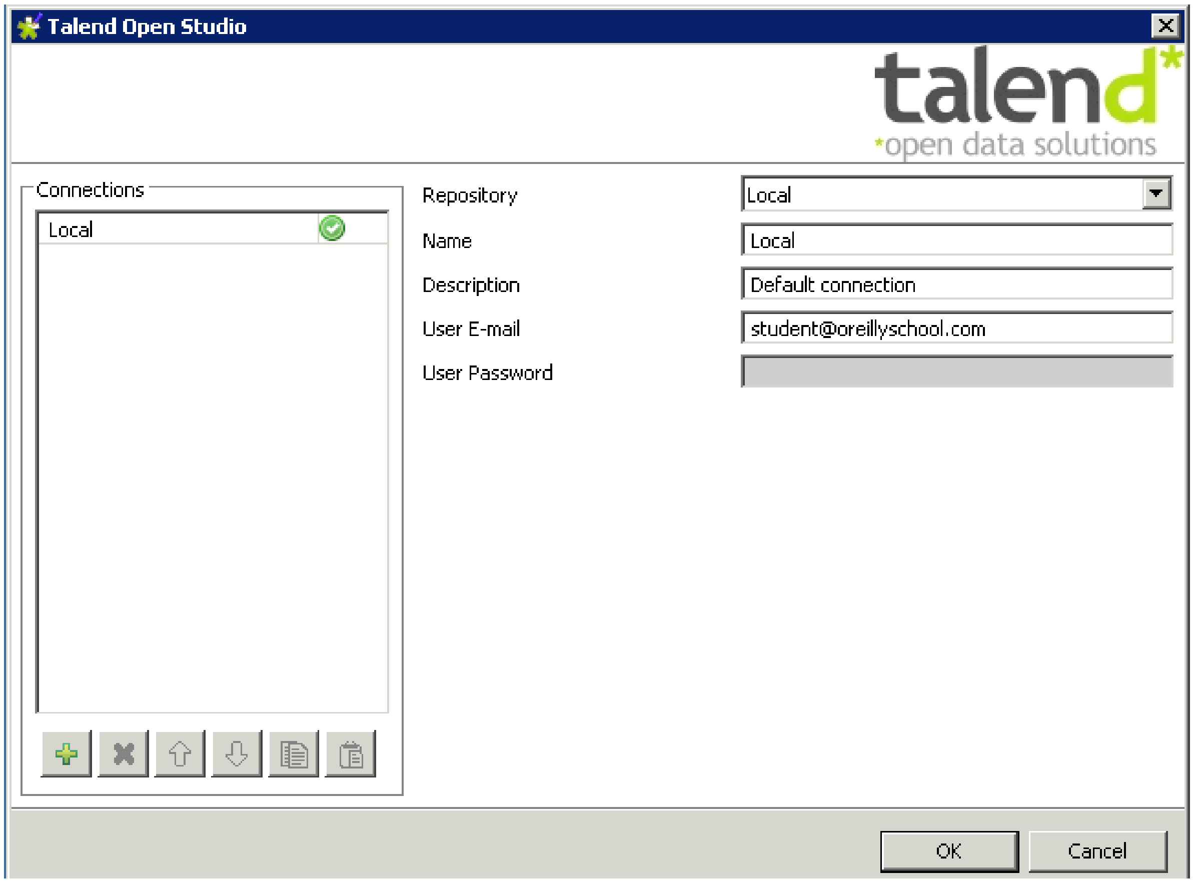
User E-mail

User Password

Manage connections

Project

Enter your email address in the repository management window to create a new repository, then click **OK**:

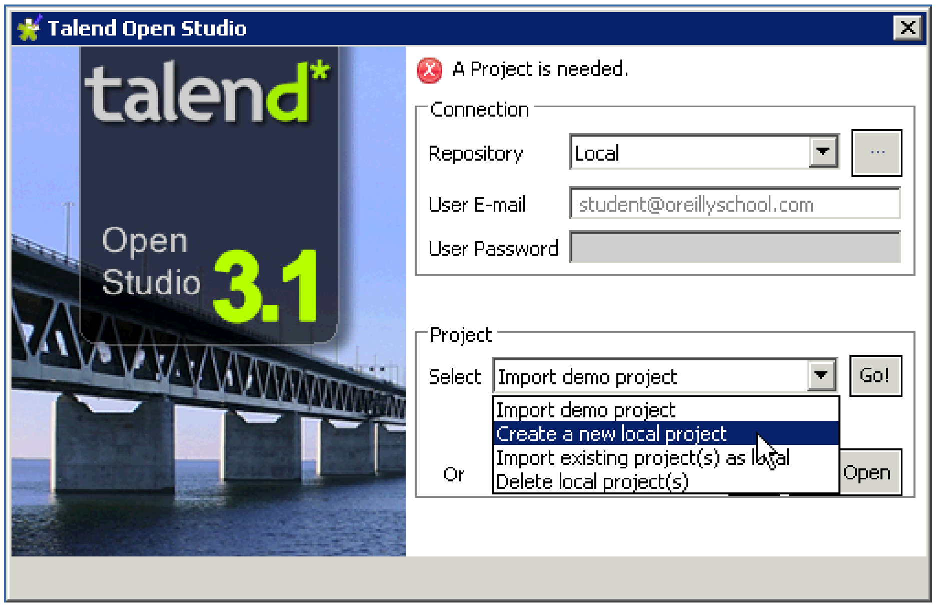

Talend Open Studio

Connections

Local

| | |
|---|---|
| Repository | Local |
| Name | Local |
| Description | Default connection |
| User E-mail | student@oreillyschool.com |
| User Password | |

Next, create your project by choosing **Create a new local project** from the drop down list and click **Go!**:


Talend Open Studio

A Project is needed.

Connection

Repository: Local

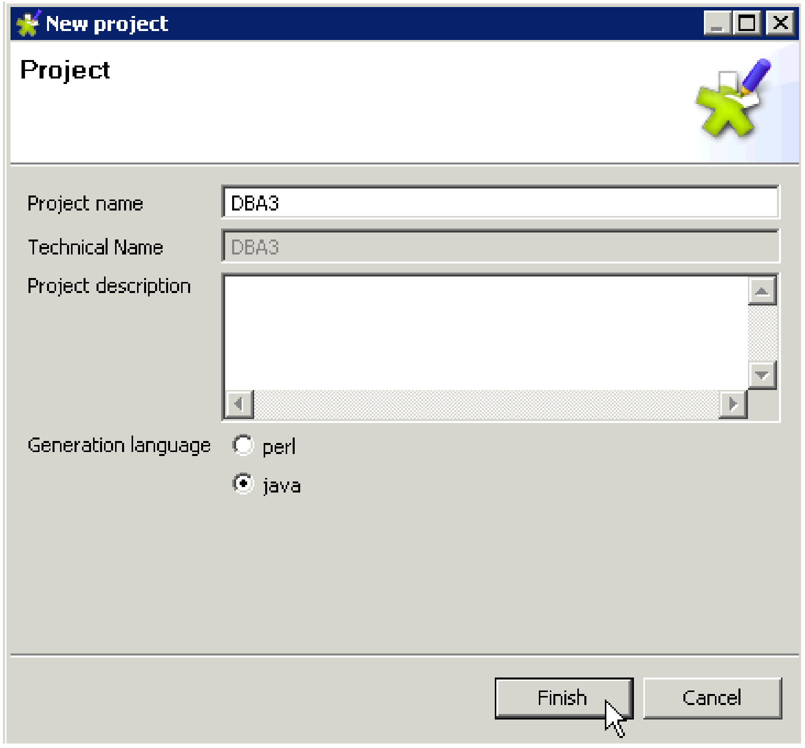
User E-mail: student@oreillyschool.com

User Password:

Project

Select: Import demo project

Name your project **DBA3**, set its language to **java**, and click **Finish**:


New project

Project

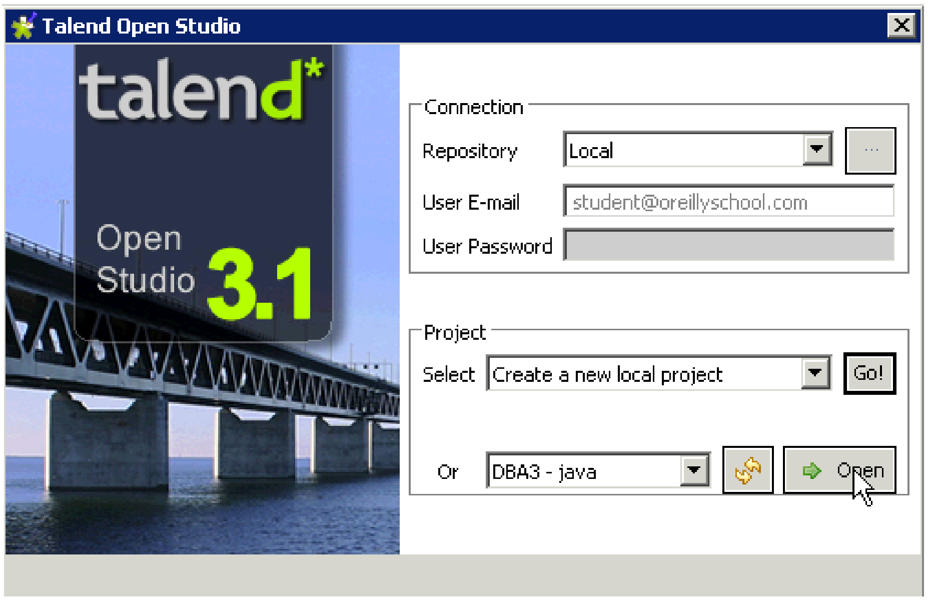
Project name: DBA3

Technical Name: DBA3

Project description:

Generation language: perl [✓ java]

Next, choose your new project from the drop-down and click **Open**:



Connection

Repository: Local

User E-mail: student@oreillyschool.com

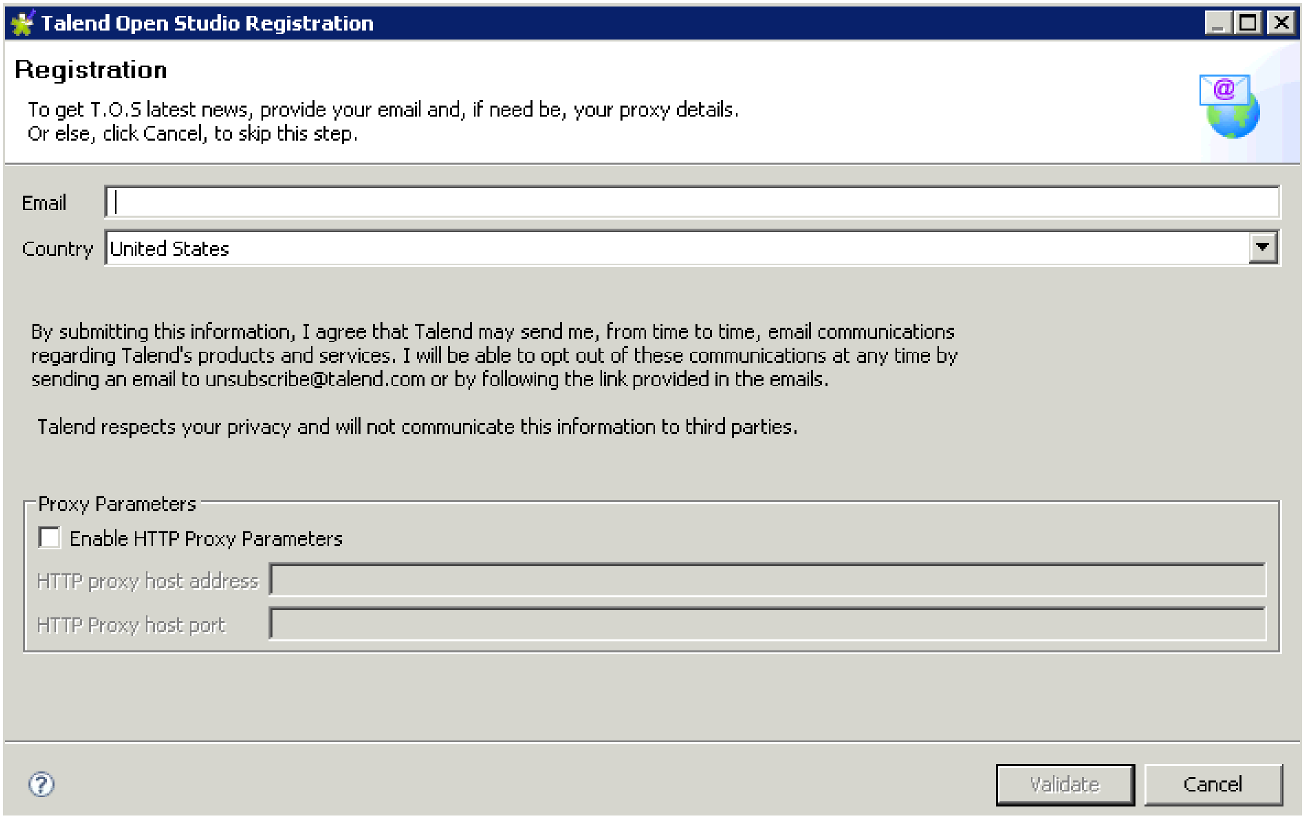
User Password:

Project

Select: Create a new local project — Go!

Or: DBA3 - java — Open

When your project loads you'll see a screen asking if you want to register with Talend. If you are interested in keeping up to date with Talend, enter your email address. Otherwise click **Cancel**:



Registration

To get T.O.S latest news, provide your email and, if need be, your proxy details.
Or else, click Cancel, to skip this step.

Email:

Country: United States

By submitting this information, I agree that Talend may send me, from time to time, email communications regarding Talend's products and services. I will be able to opt out of these communications at any time by sending an email to unsubscribe@talend.com or by following the link provided in the emails.

Talend respects your privacy and will not communicate this information to third parties.

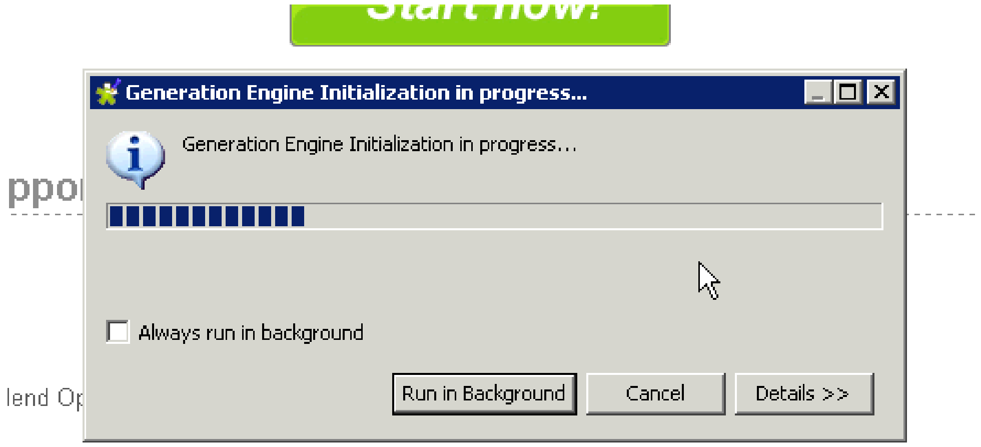
Proxy Parameters

☐ Enable HTTP Proxy Parameters

HTTP proxy host address:

HTTP Proxy host port:

The next step may take some time to complete. Under the hood, TOS generates Java code from your design instructions. This process requires some work to complete--TOS does a lot of set-up to make everything work properly. While this work is taking place, you'll see the following screen:



Start now!
Generation Engine Initialization in progress...
Generation Engine Initialization in progress...
Always run in background
Run in Background
Cancel
Details >>

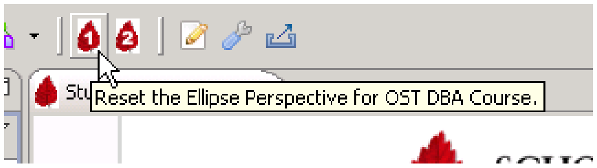
**WARNING** Do not cancel this process. If you do, it is highly likely that TOS will not work properly.

Once this process is complete, click **Start Now** to begin using TOS.

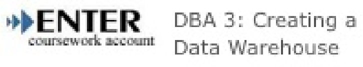
The next time you log in, you won't need to go through all of these steps. TOS will start, and you'll be able to choose the DBA 3 project you created already. You'll see the "generation" screen again, and when TOS finishes that work, you will be ready begin yours.

TOS is highly customizable - windows, palettes, and toolbars can all be moved, resized, and closed. We've also added two reset buttons to TOS (the red leaves at the top of your screen). The first red leaf resets TOS for the first part of the course, and the second red leaf resets TOS for the second part of the course. If you accidentally close some aspect of TOS, or just want to get back to the beginning, click on the **red leaf** to reset TOS.

**If you have not done so already, reset TOS by clicking on the first red leaf:**



Your student start page is on the top. Scroll down to find the DBA 3 course, and click the **Enter** button to view your syllabus:



Use your syllabus to view your lessons, projects, and quizzes.

**Note** If you have any questions or need assistance you can always contact your mentor at learn@oreillyschool.com.

## Logging Out

When you log out, you need to quit TOS instead of just closing the window. If you disconnect from your session and then reconnect using a different computer, two copies of TOS will be running, potentially overwriting each other's files.

You can prevent that problem by closing TOS from the **File->Exit** menu after you're done working.

**WARNING** Again, make sure you quit TOS when you are done working so your jobs are saved correctly!

# Introduction
# DBA 3: Data Warehousing Lesson 1

## How to Learn Using O'Reilly School of Technology Courses

Welcome to the third course in O'Reilly School of Technology's (OST) DBA series. The content of this course has been written under the assumption that you have worked through the first two courses in the series, and are familiar with MySQL. If you'd like to refresh your memory, feel free to go back over the first two courses. Then get ready to take your MySQL knowledge to the next level!

But before we dive in, let's go over the OST learning philosophy and format once more. At OST we believe that the best way to learn new technology is to play around with suggested code yourself and experiment as much as possible. As you go through the course, we encourage you to experiment with the code given in the lesson examples. At the end of each lesson you will be given a project or quiz to complete. This will be your opportunity to engage with the code, learn new skills and fulfill the most important course requirement: to have fun!

The more you experiment, the more you learn. Our learning system is designed to encourage experimentation and help you *learn how to learn*. Here are some tips for using our system effectively:

- **Learn in your own voice**

  Work through your unique ideas and trust your instincts in order to learn these new skills. We want you to facilitate your own learning, so we avoid lengthy video or audio streaming, and keep spurious animated demonstrations to a minimum.

- **Take your time**

  Learning takes time. Rushing through the work can actually slow your progress. Take time to try out new things and you'll really learn the material.

- **Create your own examples and demonstrations**

  In order to understand a complex concept, you need to understand its various parts. We'll help you do that, by offering guidance as you create a demonstration project, piece by piece.

- **Experiment with your ideas and questions**

  You're encouraged to wander from the path often to explore possibilities! We can't possibly anticipate all of your questions and ideas, so it's up to you to experiment and create on your own.

- **Accept guidance, but don't depend on it**

  Try working through any difficulties on your own first. Coming to the point of understanding on your own is the best way to learn any new skill. Our goal is for you to use the technology independent of us. Of course, you can always contact your mentor if you need help.

- **Create real projects**

  Real projects are more meaningful and rewarding to complete than simulated projects. Working on real projects will help you to understand what's involved in real world situations. After each lesson you'll be given objectives or quizzes so you can test your new knowledge.

- **Have fun!**

  Relax, keep practicing, and don't be afraid to make mistakes! There are no deadlines in this course, and your instructor will keep you at it until you've mastered each skill. We want you to experience that satisfied *I'm so cool! I did it!* feeling. And when you're finished, you'll have some really cool projects to show off.

Before we begin, let's review the structure of the lessons. We will use a lot of examples in each lesson. Whenever you see white boxes, like the one below, **type** whatever is in this box into the CodeRunner® Editor and try the example yourself:

CODE TO TYPE:

```
We'll ask you to type code that occurs in white boxes like this.

Every time you see a white box, it's your cue to type in code.
```

Similarly, when we want you to observe some code or result without altering it, we will put it in a gray box. Information presented in a gray box is for you to observe, ponder, and digest:

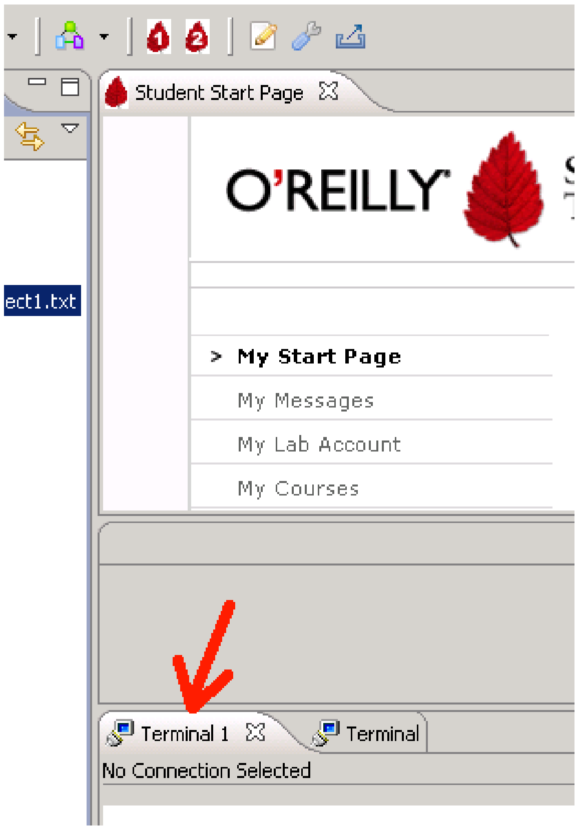
OBSERVE:

```
Gray boxes will be used for observing code.

You are not expected to type anything from these boxes.
```

| | |
|---|---|
| **Note** | If you have not read the initial course instructions Getting Started with Talend Open Studio yet, please go ahead and do that now. |

## Using the Learning Sandbox Environment

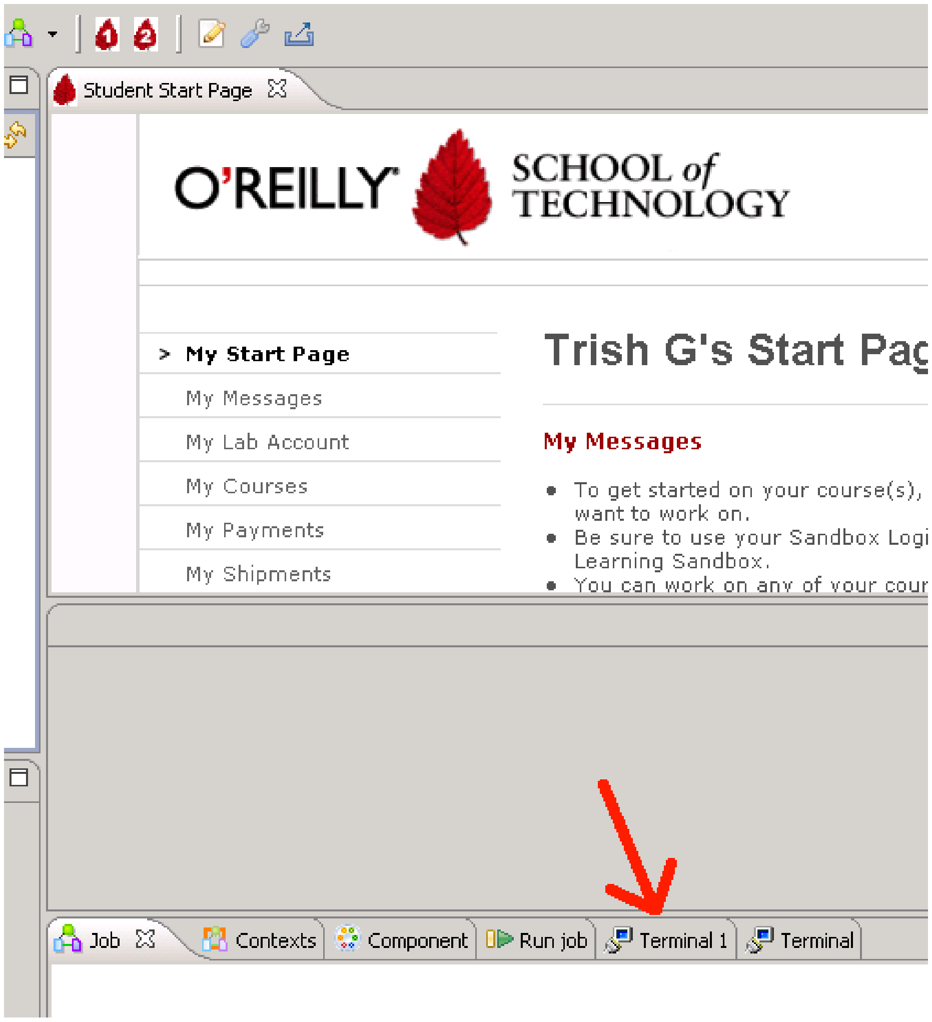
In this course you will need to connect to your Unix accounts using SSH. To allow you to do that, there are two **Terminal** views provided:



No Connection Selected

| | |
|---|---|
| **Note** | Depending on the width of your monitor, the text on the tabs may be truncated. Terminal 1 is the left terminal, and Terminal 2 is the right terminal. |

If you clicked on the second red leaf, the terminals will be located lower on the screen:

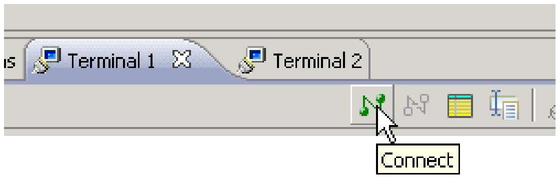


Trish G's Start Pag

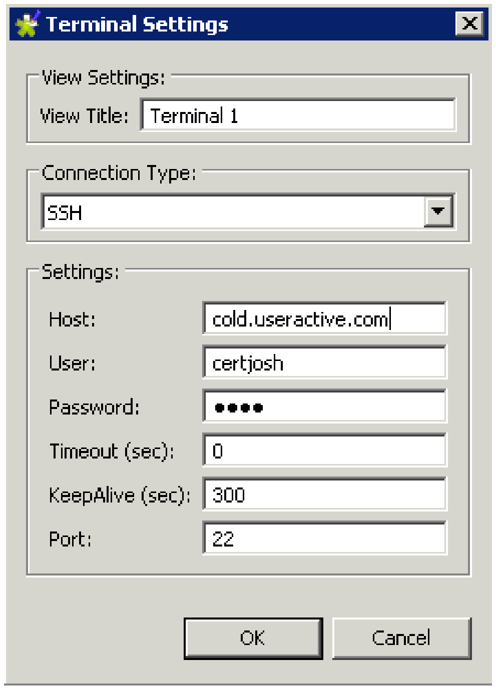
My Messages

- To get started on your course(s), want to work on.
- Be sure to use your Sandbox Logi Learning Sandbox.
- You can work on any of your cour

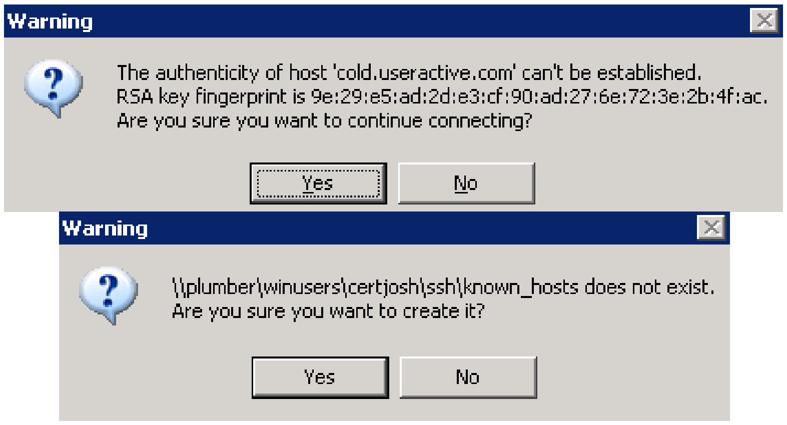
To connect, click on one of the terminal tabs and then click on **Connect**:



Change the **Connection Type** to **SSH**, set the host to **cold.useractive.com**, then enter your **username** and **password**:



The first time you connect you will see a few other warnings. Click on **Yes** for all of them:



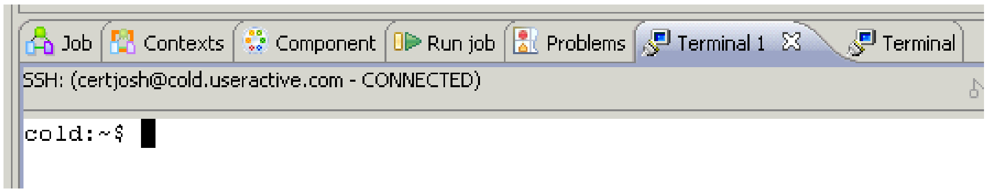
Warning

The authenticity of host 'cold.useractive.com' can't be established.
RSA key fingerprint is 9e:29:e5:ad:2d:e3:cf:90:ad:27:6e:72:3e:2b:4f:ac.
Are you sure you want to continue connecting?

Warning

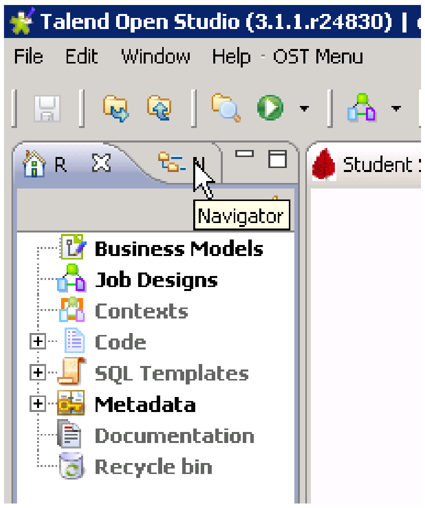
\\plumber\winusers\certjosh\ssh\known_hosts does not exist.
Are you sure you want to create it?

After you click **OK**, you will be connected to your account:

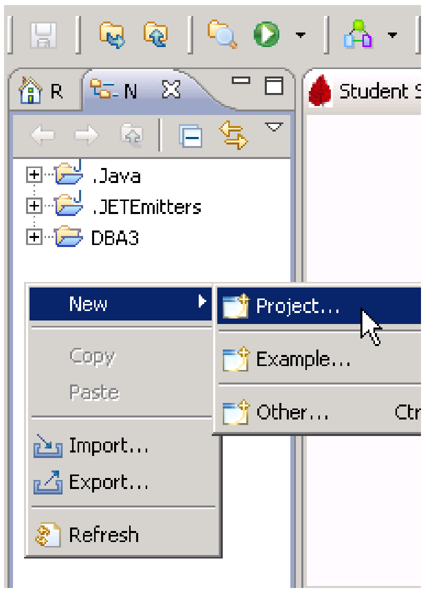


```
SSH: (certjosh@cold.useractive.com - CONNECTED)
cold:~$
```

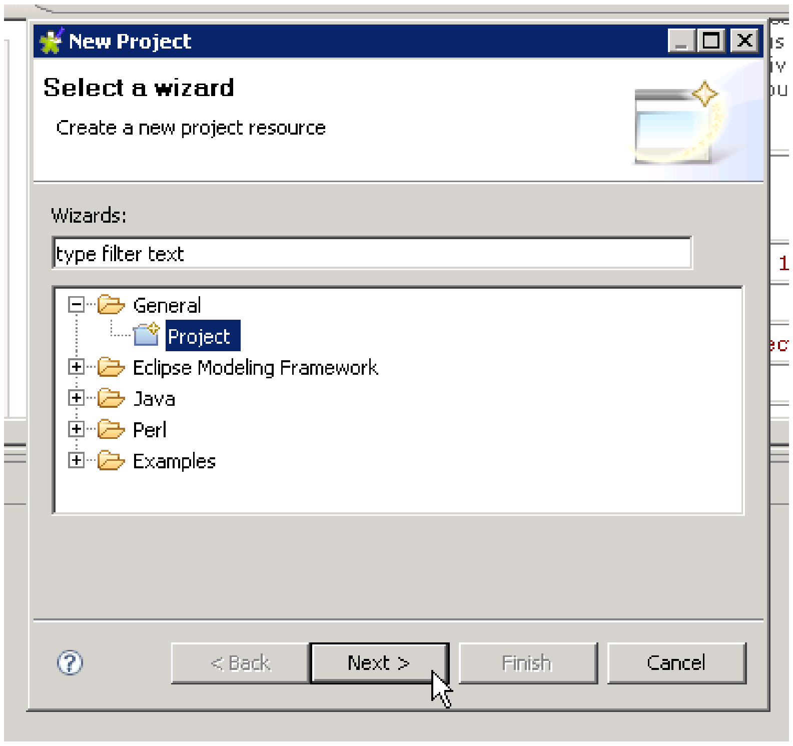
You'll also be saving some of your SQL queries and documentation in text files. We'll store these in a project accessible from TOS. To add this project, click on the **Navigation** tab:



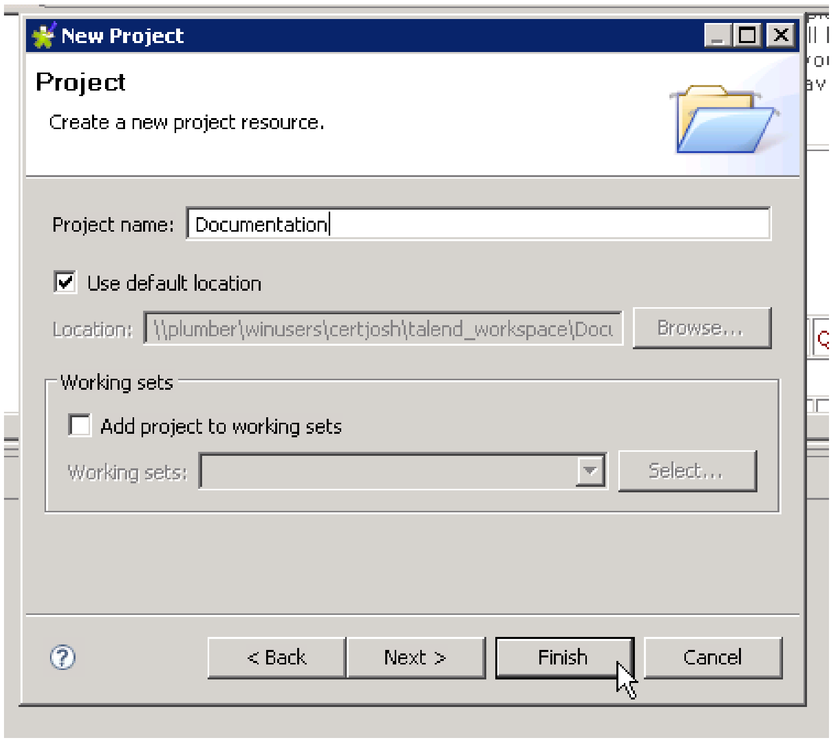
You could use this view to peek at the files that TOS stores "under the hood." We'll use it to hold our text files. Right-click in the white space under the folders, and choose **New -> Project**:



Expand the **General** folder, choose **Project**, and click **Next**:

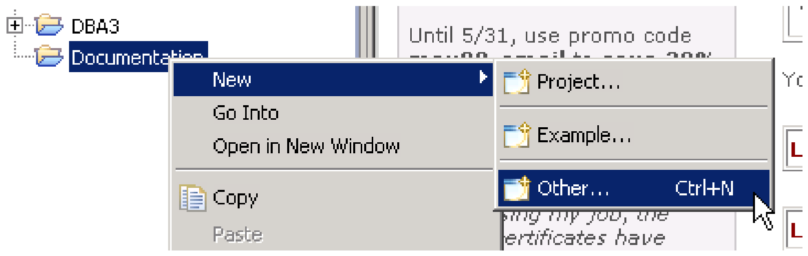


Name this project **Documentation** and click **Finish**:

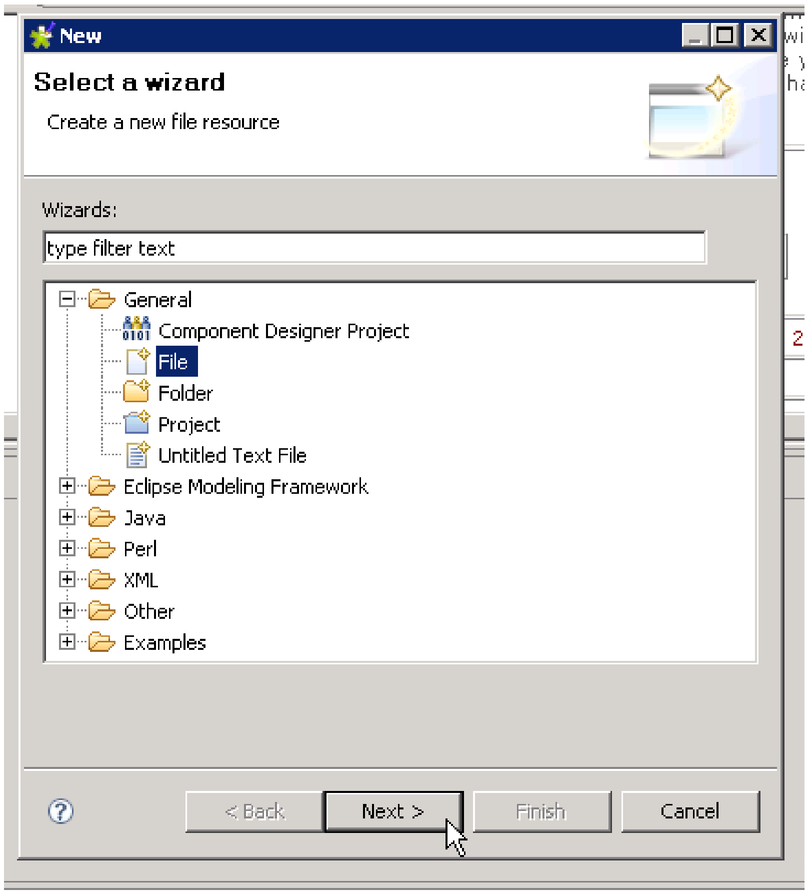

New Project
Project
Create a new project resource.
Project name: Documentation
Use default location
Location: \\plumber\winusers\certjosh\talend_workspace\Docu
Browse...
Working sets
Add project to working sets
Working sets:
Select...
< Back
Next >
Finish
Cancel

**Note** You must name your objects exactly as specified in the lesson to allow your mentor to locate your work and help you if you need it.

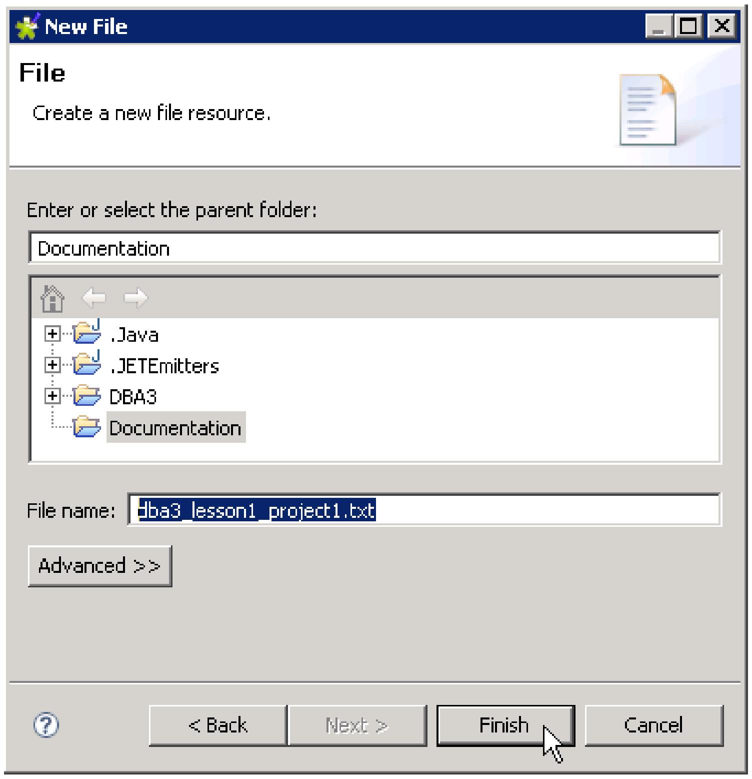
To create a new text file, right-click on **Documentation** and choose **New -> Other...**:



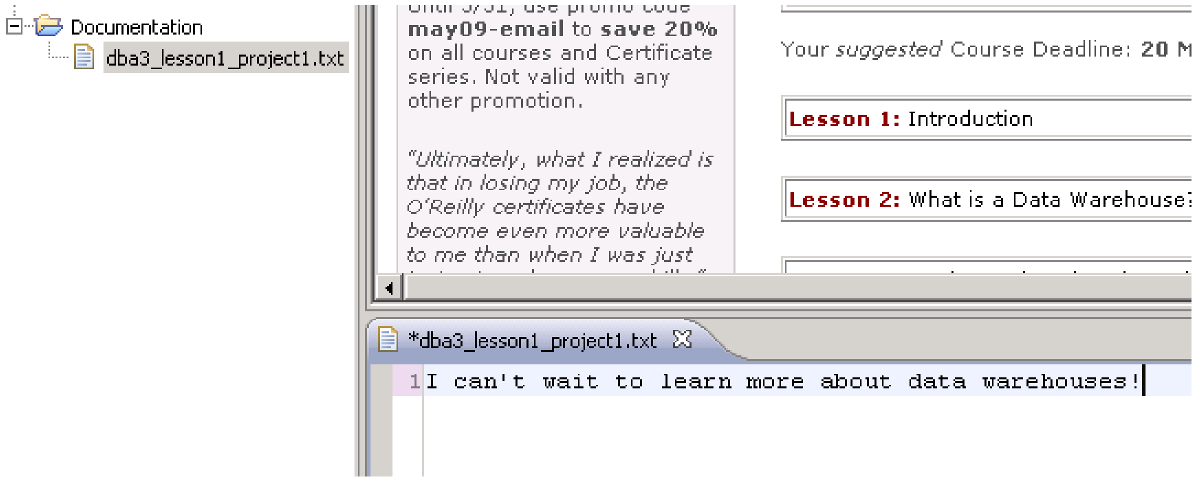
Under the **General** folder, choose **File** and click **Next**:



Make sure you select **Documentation** as the parent folder. Name your new file **dba3_lesson1_project1.txt** - you will add to this file as you complete your first project. Click on **Finish**:



An editor will open, allowing you to work on your file:



may09-email to save 20% on all courses and Certificate series. Not valid with any other promotion.

"Ultimately, what I realized is that in losing my job, the O'Reilly certificates have become even more valuable to me than when I was just

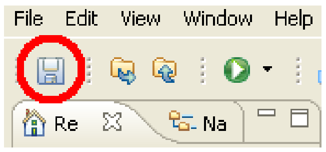
Your *suggested* Course Deadline: **20 M**

**Lesson 1:** Introduction

**Lesson 2:** What is a Data Warehouse?

```
I can't wait to learn more about data warehouses!
```

**Save** your work:



## Data Warehousing

At this point in your database education, you are familiar with SQL databases and their capabilities. By far, the most popular use for databases is the storage of operational data generated through transactions.

In the previous courses we examined the database of a DVD rental store. The database was used to keep track of customers, the DVDs in the store's inventory, and the DVDs that were currently being rented. Tables were designed and optimized for this "operational" purpose.

Keeping track of current customers and rentals is a key task of a DVD rental store database, but at some point a store manager may want to gain additional insight into the business. He may have questions such as:

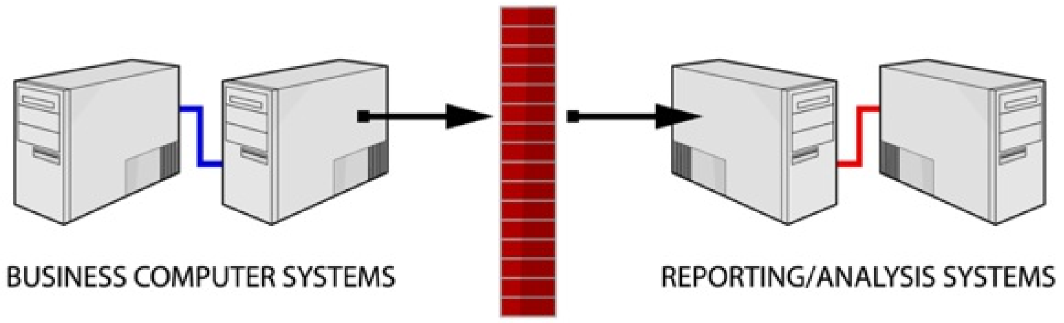
- How many new customers were added this quarter?
- What is the most popular rental?
- How much revenue did our East side store generate compared to our West side store?

- How do sales this month compare to last month or last year?
- Are movies that were popular in the theater popular rentals as well?

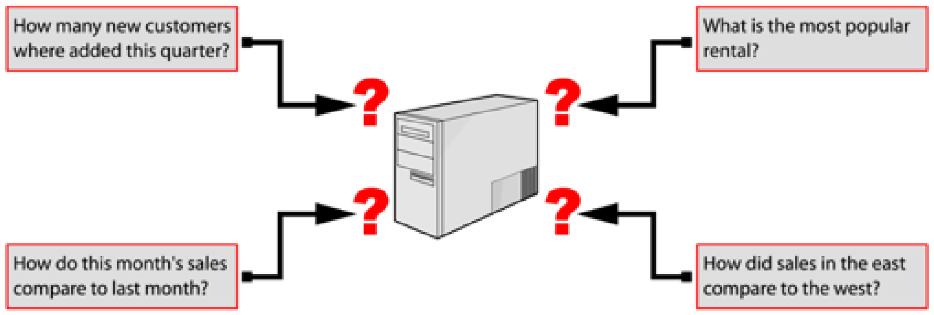
Some of our questions can be answered by writing a straightforward query against the operational database. Other more complicated questions, such as "How much revenue did our East side store generate compared to our West side store?" can be more challenging to answer. And still other questions cannot be answered at all without additional information that isn't stored in the database.

A **Data Warehouse** is designed to provide a unified platform to answer all of the questions posed above. A good data warehouse provides:

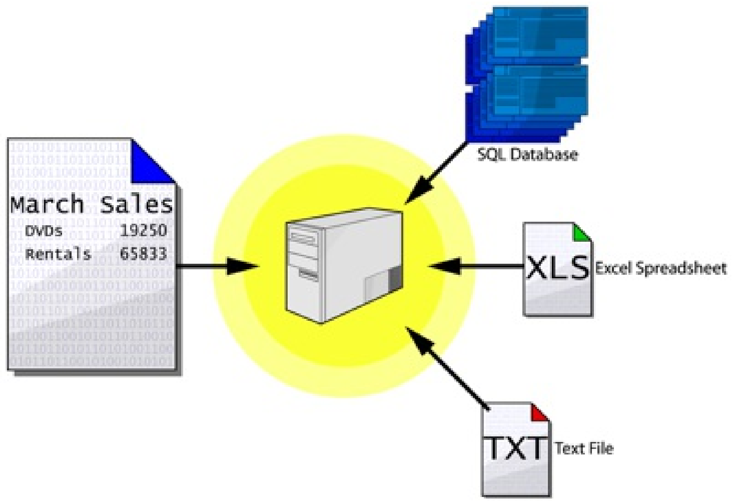
**A separate system that won't interrupt business critical operational systems:**



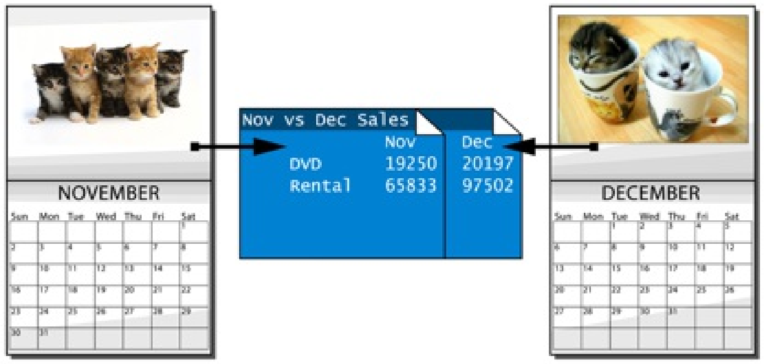
**A single point of access for all analytical queries:**


How many new customers where added this quarter?
What is the most popular rental?
How do this month's sales compare to last month?
How did sales in the east compare to the west?

**A unified and consistent view of underlying data (even data from external systems):**



**A straightforward way to analyze trends:**



In this course you'll learn everything you need to know about a data warehouse - from planning to implementation. In the next lesson we'll take a fresh look at our operational database and start planning our warehouse. See you there!

# A Data Warehouse
# DBA 3: Data Warehousing Lesson 2

## Facts and Dimensions

In the first lesson we discussed reasons to develop and use a **Data Warehouse**. The bottom line was that our video store manager wanted answers to a few good questions:

- How many new customers were added this quarter?
- What is the most popular rental?
- How much revenue did our East side store generate compared to our West side store?
- How do sales this month compare to last month or last year?
- Are movies that were popular in the theater popular rentals as well?
- Which customers rent the most DVDs each month and at which store?

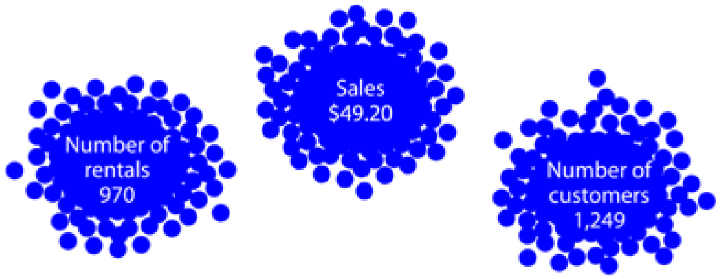
We can rewrite some of the questions so that they share a format we can use more readily in our queries:

- **How many new customers did we add by quarter**?
- **How many times were DVDs rented**, **by DVD and by month**?
- **How much sales did we do**, **by store and by month**?
- **How much sales did we do by month**?
- **How many times were DVDs rented**, **by month and by theater popularity**?
- **How many times were DVDs rented**, **by customer, month and store?**

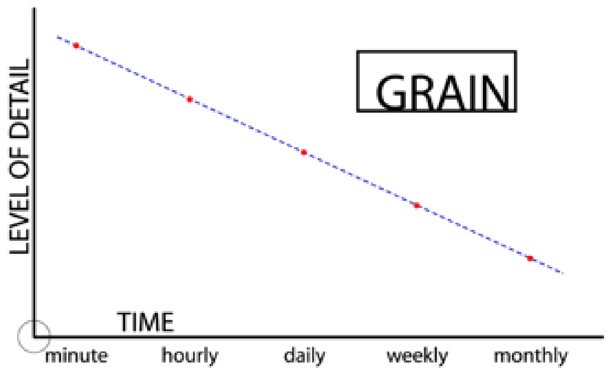
Though the questions are slightly different than they were originally, they are now structured like analytical queries, with **facts** and **dimensions**.

### Facts

**Facts** are numbers, and are sometimes referred to as *measures*. A facts relating to sales could be "Sales in US Dollars" and "Sales in Euros." Other facts could be "Hours of Work," or "Times Rented."



**Facts** have a defined **grain** - the level of detail. For example, "Sales in US Dollars" may be daily, or even hourly. If you have sales data on a daily **grain**, you cannot display sales by hour. You can, however, combine (aggregate) daily sales to larger grains such as weekly or monthly:



*Aggregates* are applied to facts in order to move to a larger grain. The most common aggregate is SUM. Other aggregates are Average (AVG), count, maximum (MAX) and minimum (MIN). Aggregates take a set of data and return a summary of that data.

Let's experiment with some aggregates in our SQL database. Switch to the SSH mode, and log into your account. In Unix mode, use the `mysql` command to connect to the sakila database as the **anonymous** user. When prompted for a password, press enter. In Unix mode, run the following command:

CODE TO TYPE:

```
cold:~$ mysql -h sql sakila -u anonymous -p
```

If you have entered everything correctly you will see this:

OBSERVE:

```
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 28527
Server version: 5.0.62-log Source distribution

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql>
```

Let's take a look at the tables, using the `show tables` command. Run this command:

CODE TO TYPE:

```
mysql> show tables;
```

OBSERVE:

```
mysql> show tables;
+----------------------------+
| Tables_in_sakila           |
+----------------------------+
| actor                      |
| actor_info                 |
| address                    |
| category                   |
| city                       |
| country                    |
| customer                   |
| customer_list              |
| film                       |
| film_actor                 |
| film_category              |
| film_list                  |
| film_text                  |
| inventory                  |
| language                   |
| nicer_but_slower_film_list |
| payment                    |
| rental                     |
| sales_by_film_category     |
| sales_by_store             |
| staff                      |
| staff_list                 |
| store                      |
+----------------------------+
23 rows in set (0.00 sec)
```

The sakila database has 23 tables and views. Let's take a closer look at the table called **payment**. Run the following command:

CODE TO TYPE:

```
mysql> describe payment;
```

OBSERVE:

```
mysql> describe payment;
+--------------+----------------------+------+-----+-------------------+----------------+
| Field        | Type                 | Null | Key | Default           | Extra          |
+--------------+----------------------+------+-----+-------------------+----------------+
| payment_id   | smallint(5) unsigned | NO   | PRI | NULL              | auto_increment |
| customer_id  | smallint(5) unsigned | NO   | MUL | NULL              |                |
| staff_id     | tinyint(3) unsigned  | NO   | MUL | NULL              |                |
| rental_id    | int(11)              | YES  | MUL | NULL              |                |
| amount       | decimal(5,2)         | NO   |     | NULL              |                |
| payment_date | datetime             | NO   |     | NULL              |                |
| last_update  | timestamp            | NO   |     | CURRENT_TIMESTAMP |                |
+--------------+----------------------+------+-----+-------------------+----------------+
7 rows in set (0.00 sec)
```

This table has has a column named **amount**, which is a **measure**.

So, "What is the total of all payments received?" Run this command to find out:

CODE TO TYPE:

```
SELECT sum(amount) from payment;
```

If you typed your query correctly you'll see the following results:

OBSERVE:

```
mysql> SELECT sum(amount) from payment;
+-------------+
| sum(amount) |
+-------------+
|    67416.51 |
+-------------+
1 row in set (0.14 sec)
```

Now let's answer this question: "What is the largest payment received?" In order to do that, run this command:

CODE TO TYPE:

```
SELECT max(amount) from payment;
```

It looks like our largest payment was $11.99:

OBSERVE:

```
mysql> SELECT max(amount) from payment;
+-------------+
| max(amount) |
+-------------+
|       11.99 |
+-------------+
1 row in set (0.02 sec)
```

So, what if you need to track an event in the data warehouse? Events don't usually have a numeric value attached to them (other than perhaps "count"). They are known as **factless facts**. We'll talk more about **factless facts** later.

By themselves, facts are not particularly useful, so next we'll add context through *dimensions*.

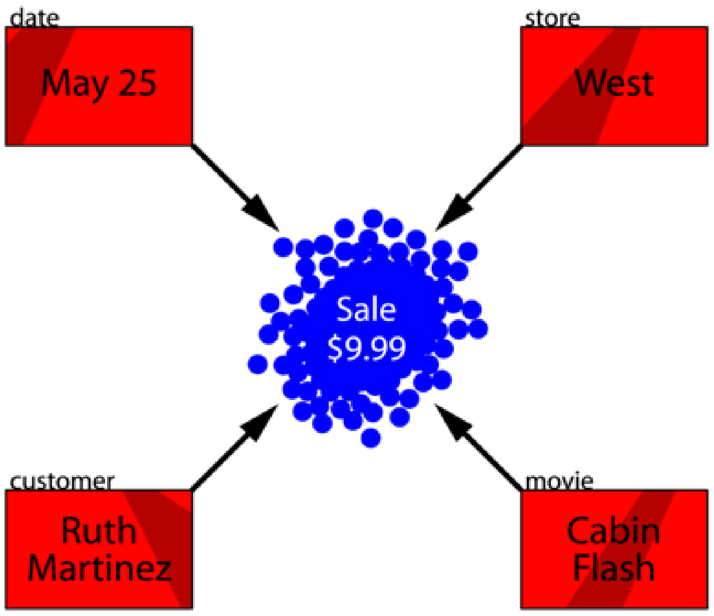
## Dimensions

**Dimensions** are used to filter, categorize, and label **facts**. A fact such as "Sales in US Dollars" might have dimensions for *Date*, *Customer*, *Store*, and *Movie*. Written in English, this might translate to something like this:

On **May 25th, Ruth Martinez rented the movie "Cabin Flash" from the West side store** for **$9.99**.

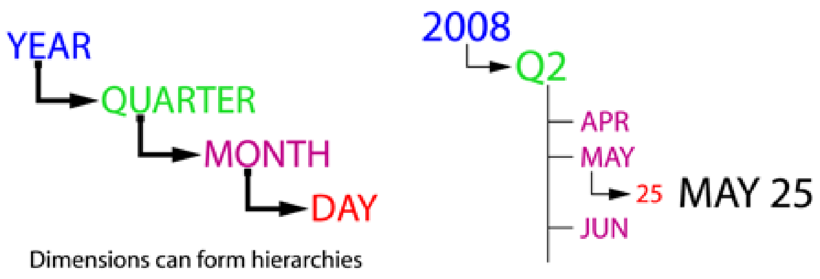
Or, broken into its components, it looks like this:

|  | Name | Value |
|---|---|---|
| **Dimension** | Date | May 25th |
| **Dimension** | Customer | Ruth Martinez |
| **Dimension** | Movie | Cabin Flash |
| **Dimension** | Store | West |
| **Fact** | Sales in US Dollars | $9.99 |

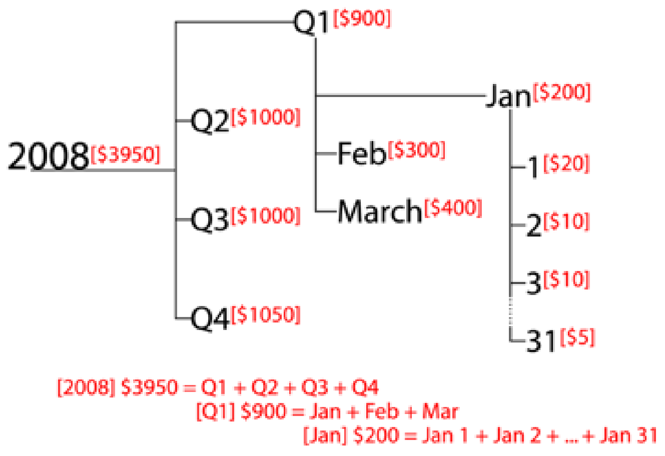


The first and most important **dimension** used in a warehouse is the *date* dimension. This dimension is often presented in a hierarchy:

Year -> Quarter -> Month -> Day



Days can "roll up" to a month. Months can "roll up" to a quarter, and quarters "roll up" to a year. Daily sales "roll up" to monthly sales, monthly sales "roll up" to quarterly sales, and quarterly sales "roll up" to yearly sales:


Q1[$900]
Q2[$1000]
2008[$3950]
Q3[$1000]
Q4[$1050]
Jan[$200]
Feb[$300]
March[$400]
1[$20]
2[$10]
3[$10]
31[$5]
[2008] $3950 = Q1 + Q2 + Q3 + Q4
[Q1] $900 = Jan + Feb + Mar
[Jan] $200 = Jan 1 + Jan 2 + ... + Jan 31

**Year**, **Quarter**, **Month**, and **Day** are not dimensions themselves. They represent *levels* in the **Date** dimension's hierarchy.

Dates often have multiple uses in a warehouse. For DVD rentals, dates are recorded at least twice: once when a movie is rented and again when the movie is returned. When the same underlying date dimension is used for both of these purposes, the dimension is known as a *role-playing* dimension.

In the SQL world we specify dimensions in the **GROUP BY** and **WHERE** clauses. Let's see these clauses in action using our examples.

First let's examine the data stored in our database that corresponds to Ruth renting "Cabin Flash" from the West store on May

25th for $9.99. For the sake of experiment, we happen to know that this data is stored with a `payment_id=491`. (Just play along for now.) Run the following command:

CODE TO TYPE:

```
select c.first_name, c.last_name, f.title, p.amount,
DATE_FORMAT(p.payment_date, '%b %D') as paymentDate, s.region
from payment p
join customer c on (p.customer_id=c.customer_id)
join rental r on (p.rental_id = r.rental_id)
join inventory i on (r.inventory_id=i.inventory_id)
join film f on (i.film_id=f.film_id)
join store s on (c.store_id=s.store_id)
where p.payment_id=491;
```

MySQL responds with our data:

OBSERVE:

```
mysql> select c.first_name, c.last_name, f.title, p.amount,
    -> DATE_FORMAT(p.payment_date, '%b %D') as paymentDate, s.region
    -> from payment p
    -> join customer c on (p.customer_id=c.customer_id)
    -> join rental r on (p.rental_id = r.rental_id)
    -> join inventory i on (r.inventory_id=i.inventory_id)
    -> join film f on (i.film_id=f.film_id)
    -> join store s on (c.store_id=s.store_id)
    -> where p.payment_id=491;
+------------+-----------+-------------+--------+-------------+--------+
| first_name | last_name | title       | amount | paymentDate | region |
+------------+-----------+-------------+--------+-------------+--------+
| RUTH       | MARTINEZ  | CABIN FLASH |   9.99 | May 25th    | West   |
+------------+-----------+-------------+--------+-------------+--------+
1 row in set (0.09 sec)
```

Looks good! Now let's answer our first question: *How much was rented on May 25th by Ruth Martinez in the West store?* Go ahead and run this command:

CODE TO TYPE:

```
select c.first_name, c.last_name, p.amount,
DATE_FORMAT(p.payment_date, '%b %D') as paymentDate, s.region
from payment p
join customer c on (p.customer_id=c.customer_id)
join store s on (c.store_id=s.store_id)
where day(p.payment_date)=25 and month(p.payment_date)=5
AND c.first_name='RUTH' and c.last_name='MARTINEZ';
```

The database does its job and returns the requested information:

OBSERVE:

```
mysql> select c.first_name, c.last_name, p.amount,
    -> DATE_FORMAT(p.payment_date, '%b %D') as paymentDate, s.region
    -> from payment p
    -> join customer c on (p.customer_id=c.customer_id)
    -> join store s on (c.store_id=s.store_id)
    -> where day(p.payment_date)=25 and month(p.payment_date)=5
    -> AND c.first_name='RUTH' and c.last_name='MARTINEZ';
+------------+-----------+--------+-------------+--------+
| first_name | last_name | amount | paymentDate | region |
+------------+-----------+--------+-------------+--------+
| RUTH       | MARTINEZ  |   0.99 | May 25th    | West   |
| RUTH       | MARTINEZ  |   9.99 | May 25th    | West   |
+------------+-----------+--------+-------------+--------+
2 rows in set (0.15 sec)
```

This is correct, but unfortunately it isn't exactly what we're after. We actually want one row of summarized data instead of two rows of detail data. We need to aggregate the **amount** fact, and make sure to *GROUP BY* our dimensions. Run this command:

CODE TO TYPE:

```
select c.first_name, c.last_name, sum(p.amount),
DATE_FORMAT(p.payment_date, '%b %D') as paymentDate, s.region
from payment p
join customer c on (p.customer_id=c.customer_id)
join store s on (c.store_id=s.store_id)
where day(p.payment_date)=25 and month(p.payment_date)=5
AND c.first_name='RUTH' and c.last_name='MARTINEZ'
GROUP BY c.first_name, c.last_name, paymentDate, s.region;
```

Excellent--now we have our desired result:

OBSERVE:

```
mysql> select c.first_name, c.last_name, sum(p.amount),
    -> DATE_FORMAT(p.payment_date, '%b %D') as paymentDate, s.region
    -> from payment p
    -> join customer c on (p.customer_id=c.customer_id)
    -> join store s on (c.store_id=s.store_id)
    -> where day(p.payment_date)=25 and month(p.payment_date)=5
    -> AND c.first_name='RUTH' and c.last_name='MARTINEZ'
    -> GROUP BY c.first_name, c.last_name, paymentDate, s.region;
+------------+-----------+---------------+-------------+--------+
| first_name | last_name | sum(p.amount) | paymentDate | region |
+------------+-----------+---------------+-------------+--------+
| RUTH       | MARTINEZ  |         10.98 | May 25th    | West   |
+------------+-----------+---------------+-------------+--------+
1 row in set (0.00 sec)
```

We were able to answer our question using the information stored in our current tables. So if that's the case, how is a **data warehouse** different than our existing **database**? Read on...

## The Dimensional Model

So why go to the trouble of creating a warehouse when our existing database has all the information we need? It seems like we've just invented a few new terms for our existing data.

In the last lesson we had several good reasons for creating a warehouse, remember? Data warehouses provide:

- a separate system that won't interrupt business critical operational systems.
- a single point of access for all analytical queries.
- a unified and consistent view of underlying data (even data from external systems).
- a straightforward way to analyze trends (such as monthly sales comparisons).

Our existing database can provide answers to some of our pertinent questions, but it doesn't provide any of the features listed above. Data warehouses do.

**Note** Generally, you will create a data warehouse on a separate physical machine from your business critical databases. For development purposes it is okay to share machines.

Our original database is fairly complex. Take a look at the database diagram.

### Selecting Facts and Dimensions

How do we choose which **facts** and **dimensions** to use in our data warehouse? *Ask the users, of course!* After all, if the system doesn't meet the needs of the users, what good is it? Ask them, "Which questions would you like to ask?" If they need an example, say something like, "I would like to see how much the total sales were for the West store last week."

Compile those questions, organize them according to effort and split them to the **facts** and **dimensions**, just like we did earlier in this lesson. Keep in mind that that facts and dimensions are generic terms, like the fact is "sales" not "total sales" and the dimensions are "region" and "date" not "east and west regions" and "month." Also, words like "total" and "top" and "longest" - they are just extra words, and are not part of the dimension or fact themselves.

Let's take the questions from earlier in the lesson, and organize them according to difficulty. We already split the **facts** and **dimensions**:

1. **How much sales did we do by month**?
2. **How much sales did we do**, **by store and by by month**?
3. **How many new customers did we add by quarter**?
4. **How many times were DVDs rented**, **by DVD and by month**?
5. **How many times were DVDs rented**, **by month and by movie popularity**?

6. **How many times were DVDs rented**, **by customer, month and store**?

The first two questions use a **sales** fact. The third question uses a **Customer Count** fact. The fourth and fifth use a **Rental Count** fact.

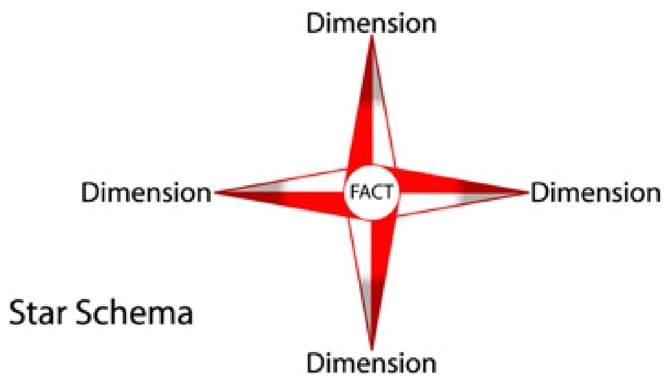
All questions use a **date** dimension. The fourth question uses a **film** dimension.

The fifth question poses a problem though, because we don't have any popularity data right now. We'll have to revisit that question later.

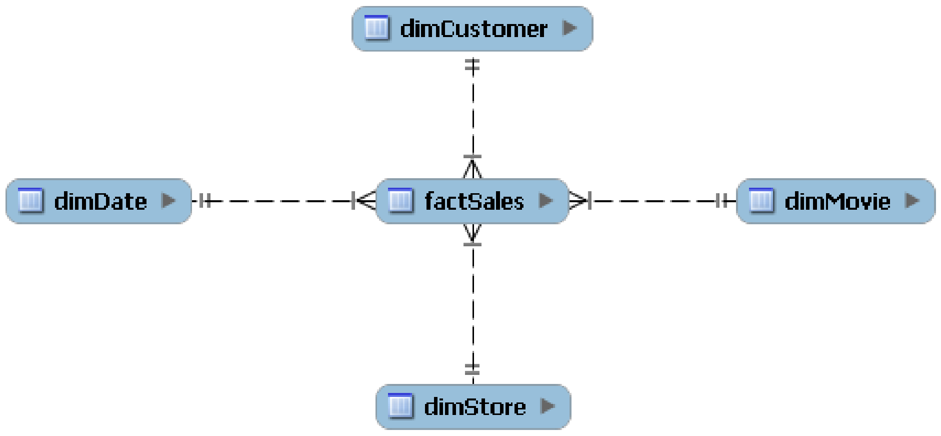
In the real world it's important to get feedback from end users so you can determine what's important to them. You can always create multiple facts or dimensions if end users don't agree on or even know what's important yet. It's perfectly fine to have two or more facts that are very similar if it helps end users get the information they need.

## Star Schema

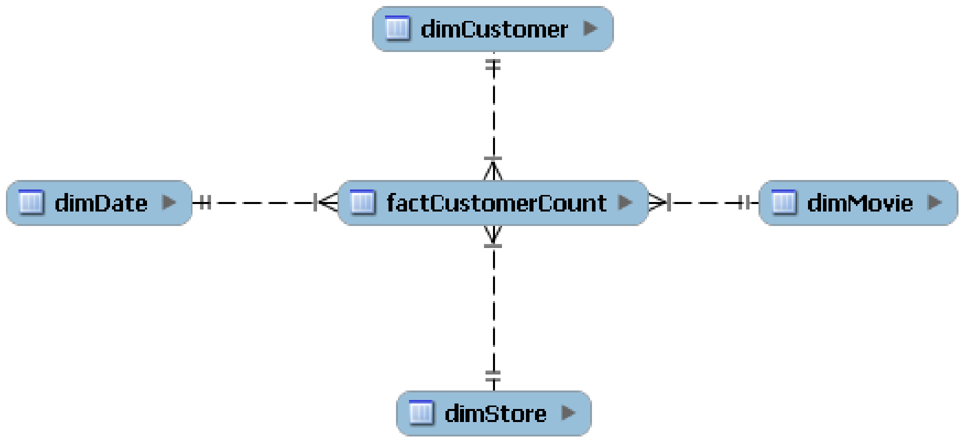
Now that we've picked our facts and dimensions, its time to organize our data. Data warehouses are typically organized using a *star schema*. Facts (measures) are stored in *fact* tables at the center of the star, and the dimensions surround the measures. Facts have foreign keys (using the *integer data type*) to each dimension table:



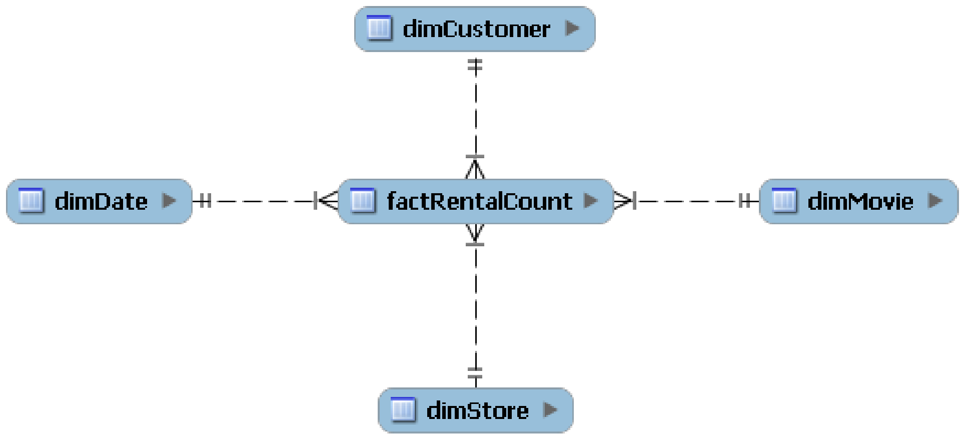
Here is our sales fact:



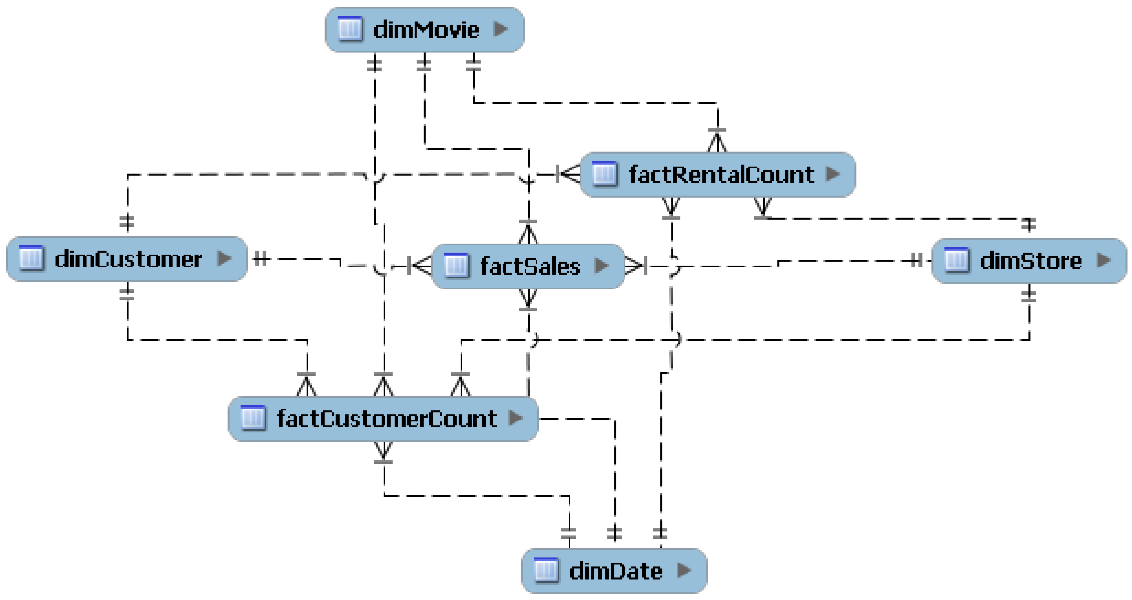
Here is our customer count fact:



Here is our rental count fact:



These separate diagrams might suggest our facts and dimensions are stored separately, but that's not the case. The dimensions are shared:



You may wonder why we're using separate tables for dimensions. Couldn't we just put the movie dimensions next to the fact in the same table? Well, we *could* do this, but we shouldn't for one good reason: performance. It is safe to assume that your fact table will become very large (millions or even billions of rows). Your dimensions may be large as well, but it is unlikely they will be nearly as large as our fact tables.

Suppose you have ten million rows of fact data and ten thousand distinct movies. Then you realize someone entered a film into your warehouse using the name "The Dude" instead of the film's actual name, "The Big Lebowski." Updating every fact row to correct that mistake could take a very long time. Even a simple query for "The Big Lebowski" could cause the database great pain; text is much more difficult to index and search than integers.

Well, we've covered a lot in this lesson.. In the next lesson we'll begin to implement our fact and dimension tables. See you there!

# Implementing the Dimensional Model, Part I
# DBA 3: Data Warehousing Lesson 3

In the last lesson we discussed facts, dimensions, and the star schema. Now it's time to implement what we learned!

## Creating the Date Dimension

The first and most important dimension in a data warehouse is the **date** dimension. Most, if not all, queries use one or more dates. In our DVD rental store there are several dates captured:

1. Customers have a `create_date`.
2. Payments have a `payment_date`.
3. Rentals have a `rental_date` and a `return_date`.

We are going to use a table to create a single date dimension to handle all of these dates. Let's call the table `dimDate`. FYI, if your dimension is reused for multiple purposes, such as recording both rental and return dates, it is a known as a *role-playing* dimension.

So, what kind of structure should we create for `dimDate`? In star schemas, facts use foreign keys of the data type *integer* to "point" to dimensions. That means we'll use a single *integer* as a primary key. At first you might envision something like this:

| Column | Type |
|---|---|
| date_key | integer, primary key, auto_increment |
| date | date |

This setup is a good start, but is it the best way to help our end users? If you recall the sample questions the users gave us, several wanted to see results *by month*. So how would you extract information about a month from a *date* type? You could use a function like `month()`, but it's probably not reasonable to expect end users to use that function.

A better solution would be to pre-calculate and pre-populate the most important date attributes as required by the users. The best way to determine what's most important is ask the users what they need. So let's suppose we did ask them, and used the information they supplied to come up with this structure:

| Column | Type |
|---|---|
| date_key | **integer, primary key** |
| date | date |
| year | smallint |
| quarter | tinyint |
| month | tinyint |
| day | tinyint |
| week | tinyint |
| **is_weekend** | boolean |
| **is_holiday** | boolean |

We kept the `date` column because it can be used to calculate attributes that didn't make it to the table. We are *not* going to use an **`auto_increment`**. Normally we would use an `auto_increment` column, but it's much more convenient to make the key a coded format such as `yyyyMMDD`. With this format, a value of `20080101` would represent January 1st, 2008.

We included two additional columns: **is_weekend** and **is_holiday**. These would be useful if we wanted to compare weekend sales or holiday sales to weekday sales. We keep the number of data types required for our columns to a minimum by consulting MySQL's documentation.

Let's go ahead and implement this table. (We'll populate it with data in a future lesson.) Switch to the terminal mode, and log into your account. Once logged in, connect to your own MySQL database. Be sure to replace **username** and **username** with your own user name. Type in the code below at the UNIX prompt:

CODE TO TYPE:

```
cold:~$ mysql -h sql -p -u username username
```

Next, create the `dimDate` table. Run this command:

CODE TO TYPE:

```
CREATE TABLE dimDate
 (
  date_key integer NOT NULL,
  date date NOT NULL,
  year smallint NOT NULL,
  quarter tinyint NOT NULL,
  month tinyint NOT NULL,
  day tinyint NOT NULL,
  week tinyint NOT NULL,
  is_weekend boolean,
  is_holiday boolean,
  PRIMARY KEY(date_key)
);
```

Execute the query. You'll see `Query OK, 0 rows affected`.

## Slowly Changing Dimensions

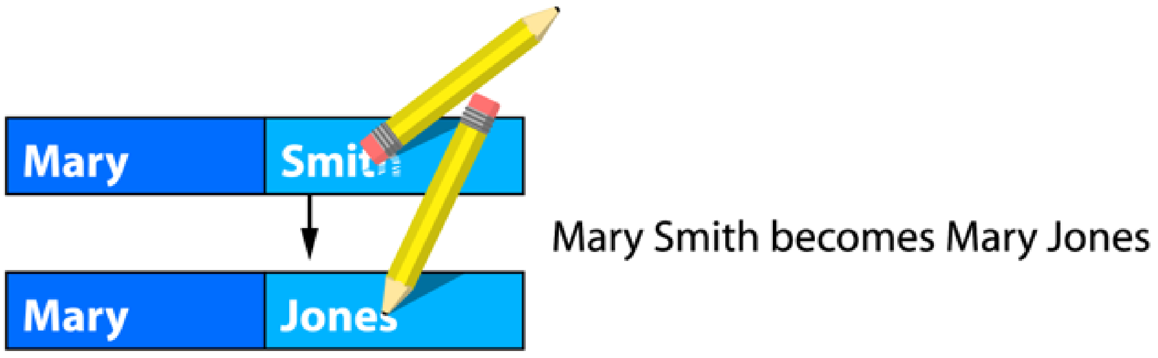
So there you are, about to create a dimension for your customers, when a business user mentions, **"I'd like to see sales according to the city in which a customer live. What happens when someone moves from one city to another? Will the sales data from last year reflect that change?"** When a dimension's values change over time, the dimension is known as a *slowly changing dimension* (or *SCD*). There are several ways we can deal with these changes.

### Type 0 SCD

The most basic SCD isn't really a change at all. If you do absolutely nothing to handle a changing dimension, that dimension is *Type 0*. In English, a type 0 translates to, "Don't do anything when this value changes."

### Type 1 SCD

A *Type 1* SCD is often the easiest way to accommodate changing dimensions. In this type, rows in the dimension tables are updated when changes occur. Suppose Mary Smith gets married in April and changes her last name to **Jones**. (She'll keep the same email address for now.)



The old dimension row would look like this:

| Customer ID | First Name | Last Name | Email | City |
|---|---|---|---|---|
| 1 | MARY | SMITH | MARY.SMITH@sakilacustomer.org | Sasebo |

The updated dimension row would look like this:

| Customer ID | First Name | Last Name | Email | City |
|---|---|---|---|---|
| 1 | MARY | **JONES** | MARY.SMITH@sakilacustomer.org | Sasebo |

Some changes are less important than others. Name changes are not always important to business users. For their purposes, it's irrelevant whether Mary Jones used to be known as Mary Smith. But suppose Mary Smith moves from one city to another in July. A Type 1 customer SCD would simply update the existing row for Mary Smith, forgetting the previous city. Now a user would be unable to see sales trends according to customer and city, because all historical data concerning Mary prior to July would now be associated with the new city.

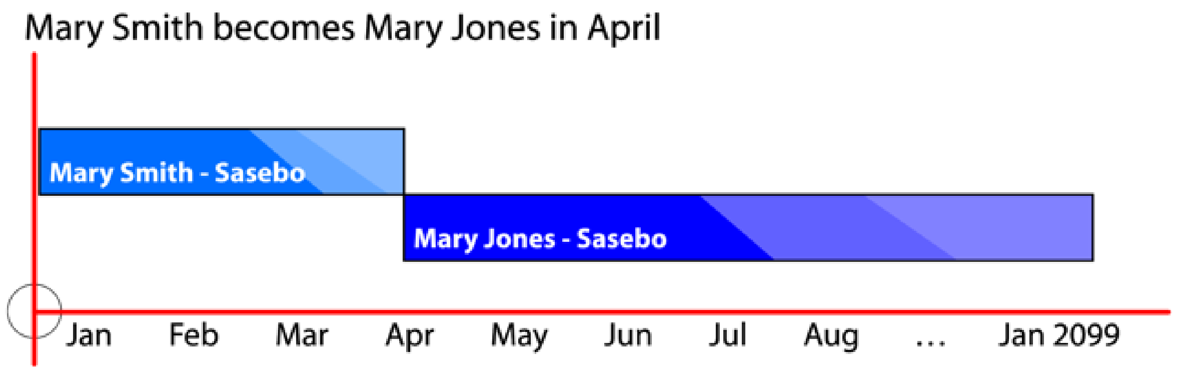
### Type 2 SCD

Type 1 isn't the best way to handle all slowly changing dimensions though. Another method to track changes in dimensions is to create a new row in the dimension table when each change occurs, and then use **begin** and **end** dates to specify the valid time period for a row.

The database row for Mary Smith would initially look like this:

| Customer Key | First Name | Last Name | Email | City | Start Date | End Date |
|---|---|---|---|---|---|---|
| 1 | MARY | SMITH | MARY.SMITH@sakilacustomer.org | Sasebo | 01-Jan-2008 | 01-JAN-2099 |

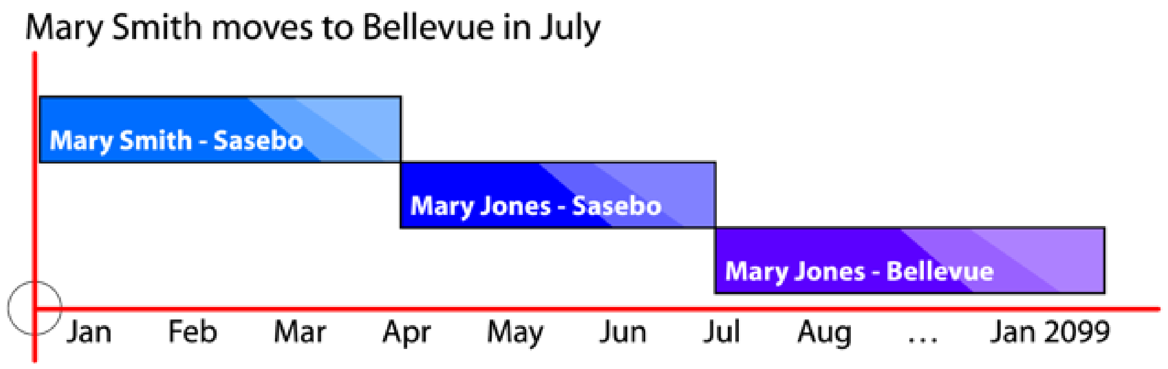
Now suppose Mary Smith gets married in April and becomes Mary Jones. Her dimension time line would look like this:



And her table would look like this:

| Customer Key | First Name | Last Name | Email | City | Start Date | End Date |
|---|---|---|---|---|---|---|
| 1 | MARY | SMITH | MARY.SMITH@sakilacustomer.org | Sasebo | 01-JAN-2008 | 01-APR-2008 |
| 2 | MARY | JONES | MARY.SMITH@sakilacustomer.org | Sasebo | 01-APR-2008 | 01-JAN-2099 |

Now let's say she moves from Sasebo to Bellevue in July, her dimension time line would look like this:



The updated dimension data would look like this:

| Customer Key | First Name | Last Name | Email | City | Start Date | End Date |
|---|---|---|---|---|---|---|
| 1 | MARY | SMITH | MARY.SMITH@sakilacustomer.org | Sasebo | 01-JAN-2008 | 01-APR-2008 |
| 2 | MARY | JONES | MARY.SMITH@sakilacustomer.org | Sasebo | 01-APR-2008 | 01-JUL-2008 |
| 3 | MARY | JONES | MARY.SMITH@sakilacustomer.org | Bellevue | 01-JUL-2008 | 01-JAN-2099 |

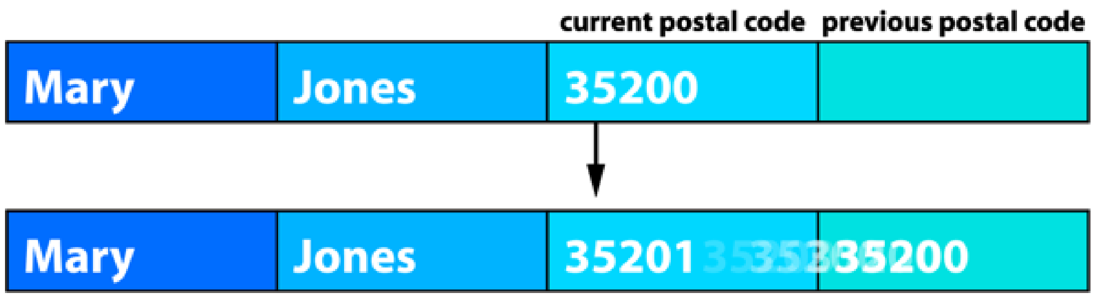
In each of the two tables that reflect Mary's new circumstances, there is one "current" row that has *01-JAN-2099* for an End Date.

**Note** Instead of using *01-JAN-2099* for an end date, some warehouses use NULL, but usually it's better to use a real date instead of NULL, because real dates can make better use of indexes.

## Type 3 SCD

Type 2 slowly changing dimensions (SCDs) allow unlimited changes, but this might be excessive for some types of changes. For example, when a postal code is changed, even though it's a fairly minor change and doesn't happen that often, it would still need to be tracked in the database. In this case, we would choose to use the Type 3 SCD method.

Suppose Mary Smith in Sasebo has her postal code changed from 35200 to 35201. The change would look like this:



The old dimension data would look like this:

| Customer Key | First Name | Last Name | Email | City | Current Postal Code | Previous Postal Code |
|---|---|---|---|---|---|---|
| 1 | MARY | SMITH | MARY.SMITH@sakilacustomer.org | Sasebo | 35200 | |

The updated dimension data would look like this:

| Customer Key | First Name | Last Name | Email | City | Current Postal Code | Previous Postal Code |
|---|---|---|---|---|---|---|
| 1 | MARY | SMITH | MARY.SMITH@sakilacustomer.org | Sasebo | 35201 | 35200 |

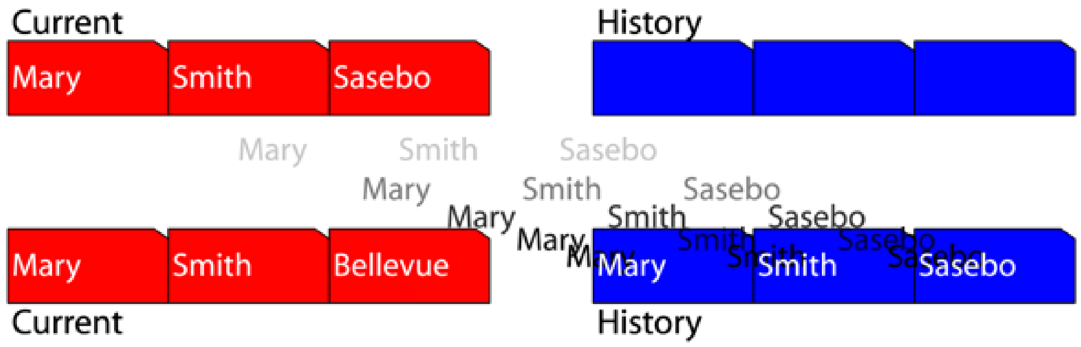
The table may or may not have an "Effective Date" column to explain when the postal code changed.

Type 3 SCDs work well for changes that happen infrequently, however this type fails to capture multiple changes.

### Type 4 SCD

A Type 4 SCD is fairly straightforward; the dimension table always contains up-to-date information. Changes are recorded in a separate *history* table. This adds complexity to dimensions, but may cause confusion because users must keep in mind that historical data is stored in a separate location.

For example, suppose Mary moves from Sasebo to Bellevue on July 15. The change would look like this:



The `dimCustomer` table would look like this:

| Customer Key | First Name | Last Name | Email | City |
|---|---|---|---|---|
| 1 | MARY | SMITH | MARY.SMITH@sakilacustomer.org | Bellevue |

The `dimCustomerHistory` table would look like this:

| Customer Key | First Name | Last Name | Email | City | Change Date |
|---|---|---|---|---|---|
| 1 | MARY | SMITH | MARY.SMITH@sakilacustomer.org | Sasebo | 15-Jul-2008 |

In practice, Type 1 and Type 2 are the most widely used ways to deal with slowly changing dimensions.

Rows do not have to be comprised entirely of a single SCD type. For example, for many data warehouses, the time that a customer name change takes place is not significant, and the change is posted for that record on-the-fly. In that case, the name columns would be of *Type 1*. Customer addresses are usually more important, so those columns would be of *Type 2*. It's perfectly fine to handle changes in this way.

## Creating the Customer Dimension

Okay, let's create our `Customer` dimension as *Type 2*. But before we do, we'll review the source of data for our dimension. The `sakila` database has a table called `customer` which will be the source of information for this dimension. Switch to the second SSH terminal, and log into your account. Use the command line mysql program to connect to the sakila database. When you're prompted for a password, click enter. In Unix mode, run the following command:

CODE TO TYPE:

```
cold:~$ mysql -h sql sakila -u anonymous -p
```

Once we're connected, we're able to see the structure of the customer table. Run the following command against the sakila database:

CODE TO TYPE:

```
describe customer;
```

As long as you have typed everything correctly, and are connected to the sakila database (not your personal database) you'll see this:

OBSERVE:

```
mysql> describe customer;
+-------------+----------------------+------+-----+-------------------+-----------------+
| Field       | Type                 | Null | Key | Default           | Extra           |
+-------------+----------------------+------+-----+-------------------+-----------------+
| customer_id | smallint(5) unsigned | NO   | PRI | NULL              | auto_increment |
| store_id    | tinyint(3) unsigned  | NO   | MUL | NULL              |                 |
| first_name  | varchar(45)          | NO   |     | NULL              |                 |
| last_name   | varchar(45)          | NO   | MUL | NULL              |                 |
| email       | varchar(50)          | YES  |     | NULL              |                 |
| address_id  | smallint(5) unsigned | NO   | MUL | NULL              |                 |
| active      | tinyint(1)           | NO   |     | 1                 |                 |
| create_date | datetime             | NO   |     | NULL              |                 |
| last_update | timestamp            | NO   |     | CURRENT_TIMESTAMP |                 |
+-------------+----------------------+------+-----+-------------------+-----------------+
9 rows in set (0.00 sec)
```

The table has a lot of information. Observe that it contains a column called address_id. This indicates that the address information is stored in a different table. Let's take a look at the address table. Now run the following command against the sakila database:

CODE TO TYPE:

```
describe address;
```

You'll see these results:

OBSERVE:

```
mysql> describe address;
+-------------+----------------------+------+-----+-------------------+-----------------+
| Field       | Type                 | Null | Key | Default           | Extra           |
+-------------+----------------------+------+-----+-------------------+-----------------+
| address_id  | smallint(5) unsigned | NO   | PRI | NULL              | auto_increment |
| address     | varchar(50)          | NO   |     | NULL              |                 |
| address2    | varchar(50)          | YES  |     | NULL              |                 |
| district    | varchar(20)          | NO   |     | NULL              |                 |
| city_id     | smallint(5) unsigned | NO   | MUL | NULL              |                 |
| postal_code | varchar(10)          | YES  |     | NULL              |                 |
| phone       | varchar(20)          | NO   |     | NULL              |                 |
| last_update | timestamp            | NO   |     | CURRENT_TIMESTAMP |                 |
+-------------+----------------------+------+-----+-------------------+-----------------+
8 rows in set (0.00 sec)
```

See the column **city_id**? It is a foreign key to the table city. Take a look at that table. Then run the following command against the sakila database:

CODE TO TYPE:

```
describe city;
```

You'll see the following structure:

OBSERVE:

```
mysql> describe city;
```

```
+-------------+----------------------+------+-----+-------------------+----------------+
| Field       | Type                 | Null | Key | Default           | Extra          |
+-------------+----------------------+------+-----+-------------------+----------------+
| city_id     | smallint(5) unsigned | NO   | PRI | NULL              | auto_increment |
| city        | varchar(50)          | NO   |     | NULL              |                |
| country_id  | smallint(5) unsigned | NO   | MUL | NULL              |                |
| last_update | timestamp            | NO   |     | CURRENT_TIMESTAMP |                |
+-------------+----------------------+------+-----+-------------------+----------------+
4 rows in set (0.00 sec)
```

It looks like this table references yet another table, using **country_id**. Let's take a look at that table as well. Then run the following command against the sakila database:

CODE TO TYPE:

```
describe country;
```

You'll see these results:

OBSERVE:

```
mysql> describe country;
+-------------+----------------------+------+-----+-------------------+----------------+
| Field       | Type                 | Null | Key | Default           | Extra          |
+-------------+----------------------+------+-----+-------------------+----------------+
| country_id  | smallint(5) unsigned | NO   | PRI | NULL              | auto_increment |
| country     | varchar(50)          | NO   |     | NULL              |                |
| last_update | timestamp            | NO   |     | CURRENT_TIMESTAMP |                |
+-------------+----------------------+------+-----+-------------------+----------------+
3 rows in set (0.00 sec)
```

Now we'll combine all of these tables into our single `dimCustomer` dimension. At first glance, we might come up with the following structure, reusing **customer_id** from our source `customer` table as the primary key for `dimCustomer`.

OBSERVE:

```
CREATE TABLE dimCustomer
(
 customer_id smallint(5) unsigned NOT NULL,
 first_name  varchar(45) NOT NULL,
 last_name   varchar(45) NOT NULL,
  email       varchar(50),
 address     varchar(50) NOT NULL,
  address2    varchar(50),
  district    varchar(20) NOT NULL,
  city        varchar(50) NOT NULL,
  country     varchar(50) NOT NULL,
  postal_code varchar(10),
  phone       varchar(20) NOT NULL,
  active      tinyint(1) NOT NULL,
  create_date datetime NOT NULL,
  last_update datetime NOT NULL,
 PRIMARY KEY(customer_id)
);
```

We *could* simply reuse this column as the key on our dimension table, but it's not a good practice because:

- It forces our slowly changing dimension to be **Type 1** instead of Type 2 since customer_id must be unique for all rows in the table.
- Changes in the source `customer` table could break the keys in the data warehouse.
- Combining multiple sources of data into a single `dimCustomer` dimension is impossible if we rely on keys generated in one system.

We'll avoid potential problems by keeping our original key, **customer_id**, and using our own **customer_key** surrogate key.

We'll populate the Type 2 slowly changing dimension (SCD) in a future lesson, but for now we'll just create the table. Switch back to the first SSH mode, the one that's connected to your personal database. Then run the following command against your personal database:

CODE TO TYPE:

```
CREATE TABLE dimCustomer
(
 customer_key int NOT NULL AUTO_INCREMENT,
 customer_id smallint(5) unsigned NOT NULL,
 first_name  varchar(45) NOT NULL,
 last_name   varchar(45) NOT NULL,
  email       varchar(50),
 address     varchar(50) NOT NULL,
  address2    varchar(50),
  district    varchar(20) NOT NULL,
  city        varchar(50) NOT NULL,
  country     varchar(50) NOT NULL,
  postal_code varchar(10),
  phone       varchar(20) NOT NULL,
  active      tinyint(1) NOT NULL,
  create_date datetime NOT NULL,
  start_date  date NOT NULL,
  end_date    date NOT NULL,
 PRIMARY KEY(customer_key)
);
```

Execute the query. If everything went okay you will see this: `Query OK, 0 rows affected`.

## Snowflake Schemas

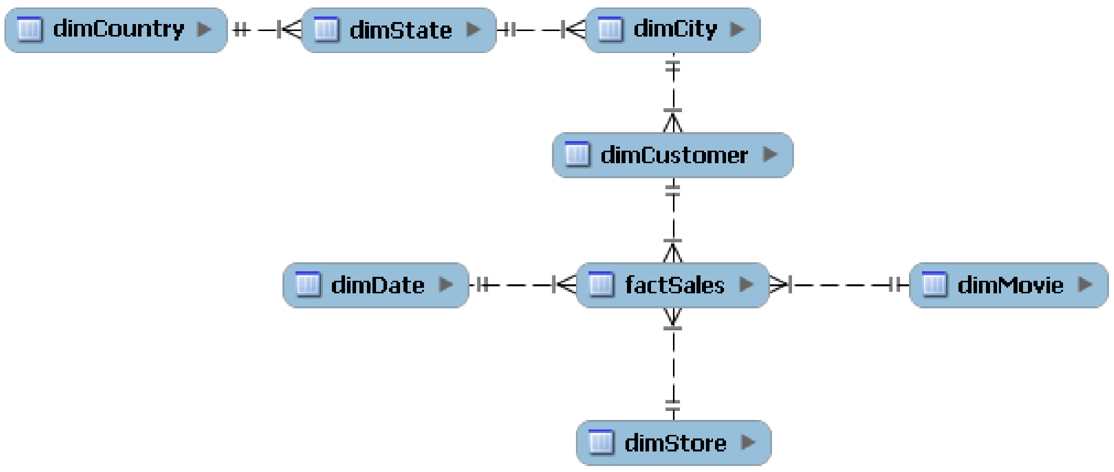
For our customer dimension, we've taken four tables and collapsed them into one table. Why did we do this?

**Simplicity**.

One of the goals of a data warehouse is to create a simple structure that users can query easily. Multiple tables means multiple joins, and added complexity. Here we traded disk space for simplicity.

We can also work in the opposite direction, using more complex schemas when our purpose calls for that. Addresses represent such a hierarchy. **Countries** have **States** (or regions), and states have **Cities**. Some business users may be interested in seeing sales data **by country**, whereas others may be interested in viewing sales data **by state** or **by city**. One way to deal with this hierarchy is with a *snowflake schema*.

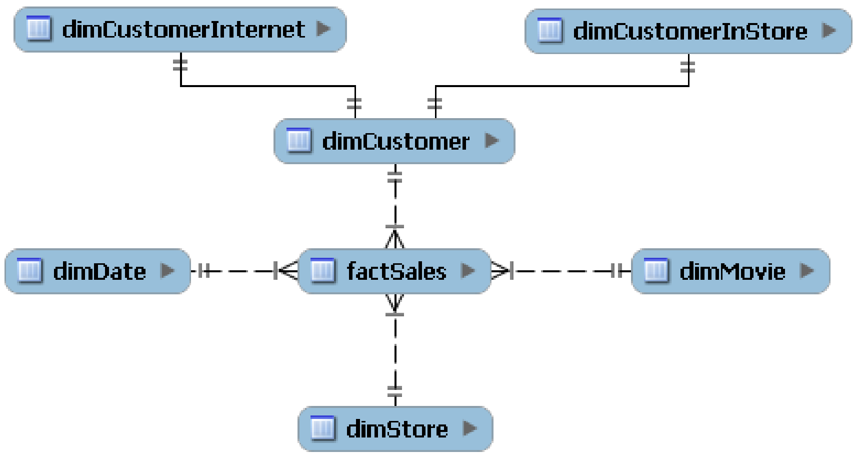
In a *snowflake schema* you split a dimension into one "primary" dimension table and one or more *snowflake* tables. It looks like this:



Snowflake schemas are also an effective way to handle a different type of problem. Suppose our DVD store starts to rent DVDs over the internet. Our store now has two types of customers - *Internet* customers and *In Store* customers. We know very little about the *In Store* customers; perhaps we only know their telephone numbers and home addresses. By comparison we know a lot about our *Internet* customers; we might have their email addresses, telephone numbers, physical addresses, movie preferences, and the number of times they have visited the web site.

Suppose we have 10,000 customers - 2,500 *In Store* and 7,500 *Internet* customers. We may actually cause confusion (and waste a lot of disk space) by shoving both types of customers into a single table. Also, we may want to see sales statistics as they vary **by customer type**.

We can use a *snowflake schema* to help with that as well. In this situation there would be one customer dimension that would hold attributes shared among Internet and In Store customers. Two additional tables - `dimCustomerInternet` and `dimCustomerInStore` would store type-specific attributes. This schema looks like this:



For right now, we'll stick to our simple star schema.

We covered a lot of material in this lesson! We'll create the rest of our dimensions in the next lesson. See you there!

# Implementing The Dimensional Model, Part II
# DBA 3: Data Warehousing Lesson 4

We began implementation of the dimensional model in the last lesson. Now it's time to finish creating our dimensions, and then create our facts.

## Creating the Movie Dimension

Before we implement `dimMovie`, let's examine the data source: the `film` table. Connect to the sakila database. Now in Unix mode, run this command:

CODE TO TYPE:

```
cold:~$ mysql -h sql sakila -u anonymous
```

Once you're connected, we'll examine the structure of the `film` table. Run the following command against the sakila database:

CODE TO TYPE:

```
describe film;
```

The table looks like this:

OBSERVE:

```
mysql> describe film;
+----------------------+---------------------------------------------------------------------+------+-
| Field                | Type                                                                | Null |
+----------------------+---------------------------------------------------------------------+------+-
| film_id              | smallint(5) unsigned                                                | NO   |
| title                | varchar(255)                                                        | NO   |
| description          | text                                                                | YES  |
| release_year         | year(4)                                                             | YES  |
| language_id          | tinyint(3) unsigned                                                 | NO   |
| original_language_id | tinyint(3) unsigned                                                 | YES  |
| rental_duration      | tinyint(3) unsigned                                                 | NO   |
| rental_rate          | decimal(4,2)                                                        | NO   |
| length               | smallint(5) unsigned                                                | YES  |
| replacement_cost     | decimal(5,2)                                                        | NO   |
| rating               | enum('G','PG','PG-13','R','NC-17')                                  | YES  |
| special_features     | set('Trailers','Commentaries','Deleted Scenes','Behind the Scenes') | YES  |
| last_update          | timestamp                                                           | NO   |
+----------------------+---------------------------------------------------------------------+------+-
13 rows in set (0.01 sec)
```

This table has two **numeric** data types: **rental_rate** and **replacement_cost**. These quantities might become **facts** stored in our data warehouse that allow us to answer questions like, "What is our profit (amount of rental income, minus film cost) for each movie?" But since that and similar questions are outside the scope for our current project, we'll omit those facts from our data warehouse and free up some space.

So it looks like we have two foreign keys: **language_id** and **original_language_id**. Both point to the `language` table. Take a look. Run the following command against the sakila database:

CODE TO TYPE:

```
describe language;
```

Execute the line to see the structure of `language`.

OBSERVE:

```
mysql> describe language;
+-------------+---------------------+------+-----+-------------------+----------------+
| Field       | Type                | Null | Key | Default           | Extra          |
+-------------+---------------------+------+-----+-------------------+----------------+
| language_id | tinyint(3) unsigned | NO   | PRI | NULL              | auto_increment |
| name        | char(20)            | NO   |     | NULL              |                |
| last_update | timestamp           | NO   |     | CURRENT_TIMESTAMP |                |
+-------------+---------------------+------+-----+-------------------+----------------+
3 rows in set (0.00 sec)
```

We'll consolidate these tables into `dimMovie`. As for changes, they are fairly infrequent in this case, so we'll implement a Type 1 slowly changing dimension. Switch to the terminal mode, and log into your account. Once you're logged in, connect to your own MySQL database. Be sure to replace **username** and **username** with your own user name. Then type the following at the UNIX prompt:

CODE TO TYPE:

```
cold:~$ mysql -h sql -p -u username username
```

Next, run the statement below, against your personal database, in order to create the `dimMovie` table:

CODE TO TYPE:

```
CREATE TABLE dimMovie
(
 movie_key          int NOT NULL AUTO_INCREMENT,
 film_id            smallint(5) unsigned NOT NULL,
  title              varchar(255) NOT NULL,
  description        text,
  release_year       year(4),
  language           varchar(20) NOT NULL,
  original_language  varchar(20),
  rental_duration    tinyint(3) unsigned NOT NULL,
  length             smallint(5) unsigned NOT NULL,
  rating             varchar(5) NOT NULL,
 special_features   varchar(60) NOT NULL,
 PRIMARY KEY (movie_key)
);
```

Once again, you'll see `Query OK, 0 rows affected`.

## Creating the Store Dimension

Now it's time to implement `dimStore`. Take a look at the data source: the `store` table. Switch terminals so that you're using the `sakila` database. Run the following command against the sakila database:

CODE TO TYPE:

```
describe store;
```

Execute the line to see the structure of the `store` table.

OBSERVE:

```
mysql> describe store;
+------------------+----------------------+------+-----+-------------------+----------------+
| Field            | Type                 | Null | Key | Default           | Extra          |
+------------------+----------------------+------+-----+-------------------+----------------+
| store_id         | tinyint(3) unsigned  | NO   | PRI | NULL              | auto_increment |
| manager_staff_id | tinyint(3) unsigned  | NO   | UNI | NULL              |                |
| address_id       | smallint(5) unsigned | NO   | MUL | NULL              |                |
| last_update      | timestamp            | NO   |     | CURRENT_TIMESTAMP |                |
| region           | varchar(10)          | YES  |     | NULL              |                |
+------------------+----------------------+------+-----+-------------------+----------------+
5 rows in set (0.00 sec)
```

Our version of the sakila database is slightly different than the version distributed by MySQL. Our version includes a `region` column. Our table also includes an **address_id** column. (Feel free to refer back to the previous lesson if you want to go over the address table structure again.)

The next interesting aspect to this table is the **manager_staff_id** column. This column is a foreign key to `staff`. Let's take a look at that table now. Run the following command against the sakila database:

CODE TO TYPE:

```
describe staff;
```

Execute the line to see the structure of `staff`.

OBSERVE:

```
mysql> describe staff;
+-------------+----------------------+------+-----+-------------------+----------------+
| Field       | Type                 | Null | Key | Default           | Extra          |
+-------------+----------------------+------+-----+-------------------+----------------+
| staff_id    | tinyint(3) unsigned  | NO   | PRI | NULL              | auto_increment |
| first_name  | varchar(45)          | NO   |     | NULL              |                |
| last_name   | varchar(45)          | NO   |     | NULL              |                |
| address_id  | smallint(5) unsigned | NO   | MUL | NULL              |                |
| picture     | blob                 | YES  |     | NULL              |                |
| email       | varchar(50)          | YES  |     | NULL              |                |
| store_id    | tinyint(3) unsigned  | NO   | MUL | NULL              |                |
| active      | tinyint(1)           | NO   |     | 1                 |                |
| username    | varchar(16)          | NO   |     | NULL              |                |
| password    | varchar(40)          | YES  |     | NULL              |                |
| last_update | timestamp            | NO   |     | CURRENT_TIMESTAMP |                |
+-------------+----------------------+------+-----+-------------------+----------------+
11 rows in set (0.00 sec)
```

We'll merge the `staff` table into a single `dimStore` dimension, and omit many of the columns from staff such as picture, email, address, username, and password. Since stores may change managers, we'll make our dimension a Type 2 SCD so we can track management changes accurately over time. That will require two additional columns: `start_date` and `end_date`. Feel free to review the Type 2 SCD section in the third lesson if you like.

Switch terminals so that you're using your personal database. Now let's create our dimension! Run the command below against your personal database:

CODE TO TYPE:

```
CREATE TABLE dimStore
(
 store_key           int NOT NULL AUTO_INCREMENT,
 store_id            smallint(5) unsigned NOT NULL,
  address             varchar(50) NOT NULL,
  address2            varchar(50),
  district            varchar(20) NOT NULL,
  city                varchar(50) NOT NULL,
  country             varchar(50) NOT NULL,
  postal_code         varchar(10),
  region              varchar(10),
  manager_first_name  varchar(45) NOT NULL,
  manager_last_name   varchar(45) NOT NULL,
  start_date          date NOT NULL,
  end_date            date NOT NULL,
 PRIMARY KEY (store_key)
);
```

So long as you see the familiar `Query OK, 0 rows affected`, you're all set.

## Creating Facts

Now that our dimensions have been created, we can implement our **facts**. Fact tables are fairly straightforward; they contain foreign keys to all dimension tables, and a single column for the fact value.

Let's get started!

### Sales

Our sales data will come from the `payment` table in the sakila database. Let's take a look. Switch back to the sakila database and run this command:

CODE TO TYPE:

```
describe payment;
```

Execute the line to see the structure of `payment`:

OBSERVE:

```
mysql> describe payment;
+--------------+----------------------+------+-----+-------------------+----------------+
| Field        | Type                 | Null | Key | Default           | Extra          |
+--------------+----------------------+------+-----+-------------------+----------------+
| payment_id   | smallint(5) unsigned | NO   | PRI | NULL              | auto_increment |
| customer_id  | smallint(5) unsigned | NO   | MUL | NULL              |                |
| staff_id     | tinyint(3) unsigned  | NO   | MUL | NULL              |                |
| rental_id    | int(11)              | YES  | MUL | NULL              |                |
| amount       | decimal(5,2)         | NO   |     | NULL              |                |
| payment_date | datetime             | NO   |     | NULL              |                |
| last_update  | timestamp            | NO   |     | CURRENT_TIMESTAMP |                |
+--------------+----------------------+------+-----+-------------------+----------------+
7 rows in set (0.00 sec)
```

We'll pay particular attention to the **amount** column. It will be the basis for our `factSales` table.

**Note** Make sure to review the sources of your facts, so you don't implement the wrong data type.

Switch back to your personal database. Let's create our fact. Run the command below against your personal database:

CODE TO TYPE:

```
CREATE TABLE factSales
(
    sales_key        INT NOT NULL AUTO_INCREMENT,
    date_key         INT NOT NULL,
    customer_key     INT NOT NULL,
    movie_key        INT NOT NULL,
    store_key        INT NOT NULL,
    sales_amount     decimal(5,2) NOT NULL,
    FOREIGN KEY fk_date (date_key) REFERENCES dimDate(date_key),
    FOREIGN KEY fk_customer (customer_key) REFERENCES dimCustomer(customer_key),
    FOREIGN KEY fk_movie (movie_key) REFERENCES dimMovie(movie_key),
    FOREIGN KEY fk_store (store_key) REFERENCES dimStore(store_key),
    PRIMARY KEY (sales_key)
);
```

Once again, if everything went according to plan, you'll see `Query OK, 0 rows affected`.

A single row in `factSales` will represent the amount of sales for a specific date, for a specific customer, for a specific movie, at a specific store.

You might think that the primary key should be a composite key across all foreign keys to the dimensions. After all, these columns should uniquely identify a fact row, right? But the problem with that type of primary key is that it tends to be very *wide*. To start, create a primary key on the surrogate key alone - **sales_key**. This will give you optimum flexibility when evaluating future indexing strategies.

## CustomerCount

Now we'll implement our `factCustomerCount`. The `factCustomerCount` is a tally of the number of customers who created accounts with our store. This table does not have a foreign key to `dimMovie` because the number of customers isn't relative to any particular movie.

We'll examine the source for this data in a future lesson. For now, let's create the fact. Make sure you are using your personal database. Review the following CREATE TABLE statement:

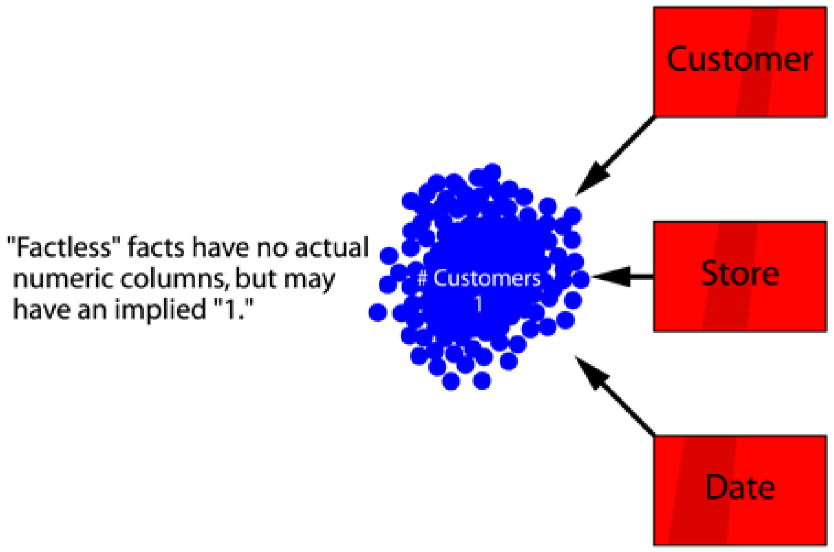
OBSERVE:

```
CREATE TABLE factCustomerCount
(
    customerCount_key INT NOT NULL AUTO_INCREMENT,
    date_key          INT NOT NULL,
    customer_key      INT NOT NULL,
    store_key         INT NOT NULL,
    customer_count    INT NOT NULL,,
    FOREIGN KEY fk_date (date_key) REFERENCES dimDate(date_key),
    FOREIGN KEY fk_customer (customer_key) REFERENCES dimCustomer(customer_key),
    FOREIGN KEY fk_store (store_key) REFERENCES dimStore(store_key),
    PRIMARY KEY (customerCount_key)
);
```

A single row in this table represents a specific customer who created an account on a specific day, at a specific store.

Before you execute the command, take a closer look at the **customer_count** measure. What values might it have?

- Since **customer_key** points to exactly one customer, **customer_count** will always have the value of **1**.
- Since **customer_count** will always be 1, we could omit the column from the table. However we will leave it in our table since it will make it easier for business users to query the table.
- Since `factCustomerCount` doesn't have any "real" facts, it is known as a **factless fact**. There will be no measure columns in this table, only foreign keys to dimensions. Factless facts are good at storing events.


Customer
Store
Date
# Customers
1
"Factless" facts have no actual numeric columns, but may have an implied "1."

Let's create the table. This time we'll specify a **default value** of 1 on **customer_count**. Run this command against your personal database:

CODE TO TYPE:

```
CREATE TABLE factCustomerCount
(
    customerCount_key INT NOT NULL AUTO_INCREMENT,
    date_key          INT NOT NULL,
    customer_key      INT NOT NULL,
    store_key         INT NOT NULL,
    customer_count    INT NOT NULL DEFAULT 1,
    FOREIGN KEY fk_date (date_key) REFERENCES dimDate(date_key),
    FOREIGN KEY fk_customer (customer_key) REFERENCES dimCustomer(customer_key),
    FOREIGN KEY fk_store (store_key) REFERENCES dimStore(store_key),
    PRIMARY KEY (customerCount_key)
);
```

Again, as long as you see `Query OK, 0 rows affected`, you're all set!

## RentalCount

Our final fact is `factRentalCount`. It's similar to `factCustomerCount` in that it is also a **factless fact**. As such, we'll also specify a default value for the **rental_count** column. (We'll populate this table in a future lesson.) Run this command against your personal database:

CODE TO TYPE:

```
CREATE TABLE factRentalCount
(
    rentalCount_key INT NOT NULL AUTO_INCREMENT,
    date_key        INT NOT NULL,
    customer_key    INT NOT NULL,
    movie_key       INT NOT NULL,
    store_key       INT NOT NULL,
    rental_count    INT NOT NULL DEFAULT 1,
    FOREIGN KEY fk_date (date_key) REFERENCES dimDate(date_key),
    FOREIGN KEY fk_customer (customer_key) REFERENCES dimCustomer(customer_key),
    FOREIGN KEY fk_movie (movie_key) REFERENCES dimMovie(movie_key),
    FOREIGN KEY fk_store (store_key) REFERENCES dimStore(store_key),
    PRIMARY KEY (rentalCount_key)
);
```

You should see: `Query OK, 0 rows affected`.

A single row in this table represents a specific customer who rented an specific movie, on a specific day, at a specific store.

We covered a lot of material in this lesson. Now that our dimensions and facts are implemented, we'll develop a strategy for transferring data from our OLTP database to our OLAP database. See you in the next lesson!

# Extract, Transform, Load (ETL)
# DBA 3: Data Warehousing Lesson 5

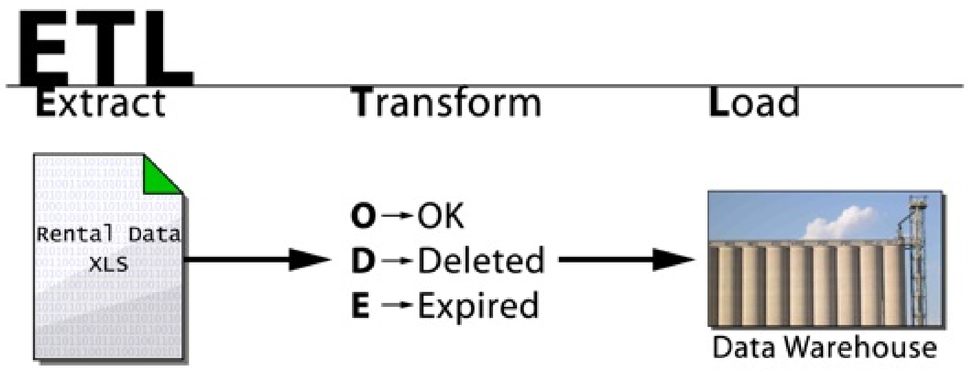
Welcome back! In the last few lessons we implemented the dimensional model. Now it's time to figure out how to populate the tables we created.

## What is ETL?

**ETL** is an acronym for **E**xtract, **T**ransform, and **L**oad. It's the process that takes data from one or more source systems, transforms and cleanses that data, and loads the result into the data warehouse.

You might be thinking *"This sounds simple! We learned about exports and imports in the last course!"*

And actually, we did learn about the **E** and **L** in the last course, but we didn't cover any **T**ransformations. An example of a Transformations would be converting codes like "D" within a source system, to a word like "Deleted" within a destination.:



And we still haven't learned how to automate data loading, or handle failures automatically and gracefully. Failure is not always an option, but when you're doing bulk exports and loads it's not too difficult to handle. If an export fails due to a disk space issue, you free up some space and try again. If an import fails, you figure out what went wrong and try again.

Failure may not be an option with **ETL**. Data warehouses hold a lot of data, and most of that data is processed on a daily (or even hourly) basis and is extremely time sensitive. If you miss a day of processing, you may lose data.

The only way to handle a large volume of complex data is to have an automated ETL process.

## Logging and Auditing

We know which data we want to pull into the data warehouse, and we know the destination tables for that data. There are several bits of information to be logged during an ETL process. First, you'll track the **data count** (the number of rows) transfered at each step in the process. This is useful information for a few good reasons:

- We can make sure no rows are "lost" in the ETL process.
- We can detect abnormal data; if we process 1000 rows one day , but the next day we only process 10, we'll know that something probably went wrong.
- In the future we can use this captured data to predict future capacity needs.

Other useful bits of information are the **start and end times** of the process:

- They can be used to alert us to problems.
- They also allow us to plan for future capacity.

Logging is not always complex. While we could use a single table to track this information, we'll split it into two tables: **etlRuns** and **etlLog**. Switch to a terminal, and log into your account, then connect to your personal database. Run this command against your personal database:

CODE TO TYPE:

```
CREATE TABLE etlRuns (
run_id integer NOT NULL AUTO_INCREMENT,
start_time datetime NOT NULL,
end_time datetime,
PRIMARY KEY(run_id)
);
```

Next we'll create the **etlLog** table, which will be used to log messages *and* statistics. Many of these columns are TOS-specific (Talend Open Studio-specific. We'll explain more about Talend in the next lesson). We will see them again when we implement logging in a later lesson. Some of this information won't be useful for every warehouse; it is up to you to decide the amount and type logging you need. Run the following command against your personal database:

CODE TO TYPE:

```
CREATE TABLE etlLog
(
 run_id integer NOT NULL,
 moment datetime NOT NULL,
 pid varchar(20),
 father_pid varchar(20),
 root_pid varchar(20),
 system_pid double,
 project varchar(50),
 job varchar(50),
 job_repository_id varchar(255),
 job_version varchar(255),
 context varchar(50),
 priority int,
 origin varchar(255),
 message_type varchar(255),
 message varchar(255),
 code int,
 duration double,
 count  int,
 reference int,
 thresholds varchar(255),
 key(run_id)
);
```

With these tables in place, we are ready to tackle **auditing**.

Why do we need auditing? Suppose we come into work one day to find that our daily sales jumped overnight from $10,000 to $1,000,000. You know that the company did not sell $1,000,000 in one day, but how do you track down the problem?

You can use the *auditing* features of the data warehouse to debug the problem. Auditing allows us to link rows in tables with specific "runs" via the `run_id` column in `etlLog`. Each row in our fact and dimension tables will have a `run_id` column, letting us know exactly when that data was added to the warehouse.

We implemented our dimensions and fact tables in the prior lesson, and those tables don't have any columns related to auditing. We did create some dimensions *type 2 SCD*, but they can't be used for auditing purposes. Instead we'll need to add a `run_id` column to all of those tables.

Let's alter our tables to add that column. Run this command against your personal database:

CODE TO TYPE:

```
ALTER TABLE dimCustomer ADD run_id int not null REFERENCES etlRuns(run_id);
ALTER TABLE dimMovie ADD run_id int not null REFERENCES etlRuns(run_id);
ALTER TABLE dimStore ADD run_id int not null REFERENCES etlRuns(run_id);
ALTER TABLE dimStaff ADD run_id int not null REFERENCES etlRuns(run_id);
ALTER TABLE factSales ADD run_id int not null REFERENCES etlRuns(run_id);
ALTER TABLE factCustomerCount ADD run_id int not null REFERENCES etlRuns(run_id);
ALTER TABLE factRentalCount ADD run_id int not null REFERENCES etlRuns(run_id);
```

**Note** We won't add auditing to `dimDate` since it will only be loaded once.

ETL processes themselves are typically broken into three parts:

1. Initial housekeeping such as create a "run", or clear temp files and tables.
2. Extract, Transform, and Load data.
3. Final housekeeping such as end a "run," send email, or clear temp files and tables.

To do the initial housekeeping we will use a stored procedure, called `etl_StartRun`. This procedure will be used to populate the `etlRuns` table and return the `run_id` to be used in all ETL processes. It will return the same `run_id` each time it is called, until the corresponding "final housekeeping" procedure `etl_EndRun` is called. Run this command against your personal database:

CODE TO TYPE:

```
DELIMITER //
CREATE PROCEDURE etl_StartRun()
BEGIN
DECLARE current_run_id INTEGER;
 SELECT max(run_id) into current_run_id
 FROM etlRuns
 WHERE end_time IS NULL;

 IF current_run_id IS NULL THEN
  BEGIN
   INSERT INTO etlRuns (start_time) VALUES (now());
   SELECT LAST_INSERT_ID() into current_run_id;
  END;
 END IF;

 SELECT 'run_id' as "key", current_run_id as value;

END;
//
DELIMITER ;
```

With that procedure out of the way, we can think about the last part of the process: a procedure to perform final housekeeping. Run this command against your personal database:

CODE TO TYPE:

```
DELIMITER //
CREATE PROCEDURE etl_EndRun
()
BEGIN
 UPDATE etlRuns SET end_time=now() where end_time IS NULL;
END;
//
DELIMITER ;
```

That looks great! Now we're ready to look at our source data.

## Getting Data into the Warehouse

Now that we have our data warehouse setup, it's time to review our source data. We defined the data we're putting into our warehouse in the previous lessons, and we have some understanding of the data source. But do we know everything about our source data? What information can we provide for our business users?

Part of ETL is **Transformation** - cleaning and transforming source data so it's easier to understand and more useful. **Transformation** can alter source data to make it better by:

- Changing customer status codes, such as O, D, C to "OK," "Deleted Customer," and "Account in Collections."
- Handling known source data errors or test data, such as all accounts with the prefix "TST_01."
- Splitting data in one column into multiple columns, for example, splitting a single field for "R:2008-05-20" into "Rental" and "2008-05-20."

Often business analysts and users are responsible for determining and documenting transformation and mapping rules. Other times, those responsibilities fall upon the programmer. But in any of those situations, it's important to have clear documentation. We want no confusion about the reasons customer codes of "D" are getting translated to "Deleted Customer" in the data warehouse.

**Note** Most companies have data scattered across many different systems, databases, and files. We'll keep things simple for this course by restricting our data sources. No matter where your data originates from, the process for getting it into the data warehouse is the same.

So, how will we document our transformation and mapping rules? **We'll use the easiest and most useful method available.** This might be a word document in some situations, or a spreadsheet in another. For this course we'll just use plain text documents to describe our transformations.

### dimDate

Our date dimension doesn't really have a source other than a calendar. So how do we come up with the data? Programmers will commonly use one of these methods:

- Create a program to populate the date table.

- Create a spreadsheet with date data in it.
- Copy the date dimension from an existing data warehouse.

Suppose one of the business users is handy with Excel, and has offered to create a spreadsheet for you. The spreadsheet will already contain all of the required information, including holidays and weekends. In this case, most of the work is done for us. We only need to load the data (which we'll do in the next lessons).

## dimCustomer

Let's take a closer look at `dimCustomer`. Back in lesson three we discovered that a customer record is stored in several tables in the sakila database: **customer**, **address**, **city** and **country**. We're not planning on doing any transformations on the data, but suppose a business user informs us that rows in the customer table where `customer_id <= 10` are actually test accounts that should be excluded from the data warehouse.

Let's write the query we need to extract the data from the customers table. Switch to the second terminal, and log into the sakila database. Run this command against the sakila database:

CODE TO TYPE:

```
SELECT
c.customer_id, c.first_name, c.last_name, c.email,
a.address, a.address2, a.district,
ci.city,
co.country,
postal_code,
a.phone,c.active, c.create_date
FROM customer c
JOIN address a on (c.address_id = a.address_id)
JOIN city ci on (a.city_id = ci.city_id)
JOIN country co on (ci.country_id = co.country_id)
WHERE customer_id > 10;
```

If you are connected to the **sakila** database and typed everything correctly, you'll see lots of results:

OBSERVE:

```
mysql> SELECT
    -> c.customer_id, c.first_name, c.last_name, c.email,
    -> a.address, a.address2, a.district,
    -> ci.city,
    -> co.country,
    -> postal_code,
    -> a.phone,c.active, c.create_date
    -> FROM customer c
    -> JOIN address a on (c.address_id = a.address_id)
    -> JOIN city ci on (a.city_id = ci.city_id)
    -> JOIN country co on (ci.country_id = co.country_id)
    -> WHERE customer_id > 10;
+-------------+------------+-------------+------------------------------------------+--------
| customer_id | first_name | last_name   | email                                    | address
+-------------+------------+-------------+------------------------------------------+--------
|         218 | VERA       | MCCOY       | VERA.MCCOY@sakilacustomer.org            | 1168 Na
|         441 | MARIO      | CHEATHAM    | MARIO.CHEATHAM@sakilacustomer.org        | 1924 Sh
|          69 | JUDY       | GRAY        | JUDY.GRAY@sakilacustomer.org             | 1031 Da
|         176 | JUNE       | CARROLL     | JUNE.CARROLL@sakilacustomer.org          | 757 Rus
|         320 | ANTHONY    | SCHWAB      | ANTHONY.SCHWAB@sakilacustomer.org        | 1892 Na
|         528 | CLAUDE     | HERZOG      | CLAUDE.HERZOG@sakilacustomer.org         | 486 Ond
...lines ommitted...
|         303 | WILLIAM    | SATTERFIELD | WILLIAM.SATTERFIELD@sakilacustomer.org   | 687 Ale
|         213 | GINA       | WILLIAMSON  | GINA.WILLIAMSON@sakilacustomer.org       | 1001 Mi
|         553 | MAX        | PITT        | MAX.PITT@sakilacustomer.org              | 1917 Ku
|         438 | BARRY      | LOVELACE    | BARRY.LOVELACE@sakilacustomer.org        | 1836 Ko
+-------------+------------+-------------+------------------------------------------+--------
589 rows in set (0.04 sec)
```

It looks like this is a good query to use to extract customer information. Save this query - we will use it in a future lesson.

## dimMovie

The next table we will populating is **dimMovie**. Data from this table comes from two tables: **film** and **language**. We will have to join on **language** twice however, since the **film** table joins to **language** on `language_id` and `original_language_id`.

Let's write the query needed to extract the data from the customers table. Run this command against the sakila database:

CODE TO TYPE:

```
SELECT f.film_id, f.title, f.description, f.release_year,
l.name as language, orig_lang.name as original_language,
f.rental_duration, f.length, f.rating, f.special_features
FROM film f
JOIN language l on (f.language_id=l.language_id)
JOIN language orig_lang on  (f.original_language_id = orig_lang.language_id);
```

Try executing the query. If you typed everything correctly, you will see the following:

OBSERVE:

```
mysql> SELECT f.film_id, f.title, f.description, f.release_year,
    -> l.name as language, orig_lang.name as original_language,
    -> f.rental_duration, f.length, f.rating, f.special_features
    -> FROM film f
    -> JOIN language l on (f.language_id=l.language_id)
    -> JOIN language orig_lang on  (f.original_language_id = orig_lang.language_id);
Empty set (0.01 sec)
```

What happened to the data? We don't have a `WHERE` clause, so that can't be the problem. But we *do* have two joins. Let's write another query to find out which join is failing us. Run this command against the sakila database:

CODE TO TYPE:

```
SELECT count(distinct language_id), count(distinct original_language_id)
FROM film f;
```

Run the query, and observe the results:

OBSERVE:

```
mysql> SELECT count(distinct language_id), count(distinct original_language_id)
    -> FROM film f;
+-----------------------------+--------------------------------------+
| count(distinct language_id) | count(distinct original_language_id) |
+-----------------------------+--------------------------------------+
|                           1 |                                    0 |
+-----------------------------+--------------------------------------+
1 row in set (0.00 sec)
```

It looks like we don't have any films that have been translated. Perhaps this is a feature in progress, or an old feature that has since been abandoned. Whatever the reason, we will need to alter our SELECT query to use a **LEFT JOIN** instead of a normal join. Run this command against the sakila database:

CODE TO TYPE:

```
SELECT f.film_id, f.title, f.description, f.release_year,
l.name as language, orig_lang.name as original_language,
f.rental_duration, f.length, f.rating, f.special_features
FROM film f
JOIN language l on (f.language_id=l.language_id)
LEFT JOIN language orig_lang on  (f.original_language_id = orig_lang.language_id);
```

As long as you typed everything correctly, you will see lots of results:

OBSERVE:

```
mysql> SELECT f.film_id, f.title, f.description, f.release_year,
    -> l.name as language, orig_lang.name as original_language,
    -> f.rental_duration, f.length, f.rating, f.special_features
    -> FROM film f
    -> JOIN language l on (f.language_id=l.language_id)
    -> LEFT JOIN language orig_lang on  (f.original_language_id = orig_lang.language_id);
+---------+-----------------------------+-------------------------------------------------------
| film_id | title                       | description
+---------+-----------------------------+-------------------------------------------------------
|       1 | ACADEMY DINOSAUR            | A Epic Drama of a Feminist And a Mad Scientist who mu
|       2 | ACE GOLDFINGER              | A Astounding Epistle of a Database Administrator And
|        3 | ADAPTATION HOLES             | A Astounding Reflection of a Lumberjack And a Car who
...lines omitted...
|      998 | ZHIVAGO CORE                 | A Fateful Yarn of a Composer And a Man who must Face
|      999 | ZOOLANDER FICTION            | A Fateful Reflection of a Waitress And a Boat who mus
|     1000 | ZORRO ARK                    | A Intrepid Panorama of a Mad Scientist And a Boy who
+----------+------------------------------+------------------------------------------------------
1000 rows in set (0.02 sec)
```

This looks great!

## dimStore

The last table we'll work on populating is **dimStore**. Data from this table comes from many tables: **store**, **staff**, **address**, **city**, and **country**. Run this command against the sakila database:

CODE TO TYPE:

```
SELECT s.store_id, a.address, a.address2, a.district,
c.city, co.country, a.postal_code, s.region,
st.first_name as manager_first_name,
st.last_name as manager_last_name
FROM
store s
JOIN staff st on (s.manager_staff_id = st.staff_id)
JOIN address a on (s.address_id = a.address_id)
JOIN city c on (a.city_id = c.city_id)
JOIN country co on (c.country_id = co.country_id);
```

Run the query, and observe the results:

OBSERVE:

```
mysql> SELECT s.store_id, a.address, a.address2, a.district,
    -> c.city, co.country, a.postal_code, s.region,
    -> st.first_name as manager_first_name,
    -> st.last_name as manager_last_name
    -> FROM
    -> store s
    -> JOIN staff st on (s.manager_staff_id = st.staff_id)
    -> JOIN address a on (s.address_id = a.address_id)
    -> JOIN city c on (a.city_id = c.city_id)
    -> JOIN country co on (c.country_id = co.country_id)
    -> ;
+----------+--------------------+----------+----------+------------+-----------+-------------+-
| store_id | address            | address2 | district | city       | country   | postal_code |
+----------+--------------------+----------+----------+------------+-----------+-------------+-
|        1 | 47 MySakila Drive  | NULL     | Alberta  | Lethbridge | Canada    |             |
|        2 | 28 MySQL Boulevard | NULL     | QLD      | Woodridge  | Australia |             |
+----------+--------------------+----------+----------+------------+-----------+-------------+-
2 rows in set (0.00 sec)
```

This looks great too!

Great job so far. In the next lesson w'll practice writing an ETL job. See you then!

# Tools for ETL
# DBA 3: Data Warehousing Lesson 6

Now that we have the mapping for our data sources done, let's consider how we are going to extract the data, transform it, and get it into the data warehouse.

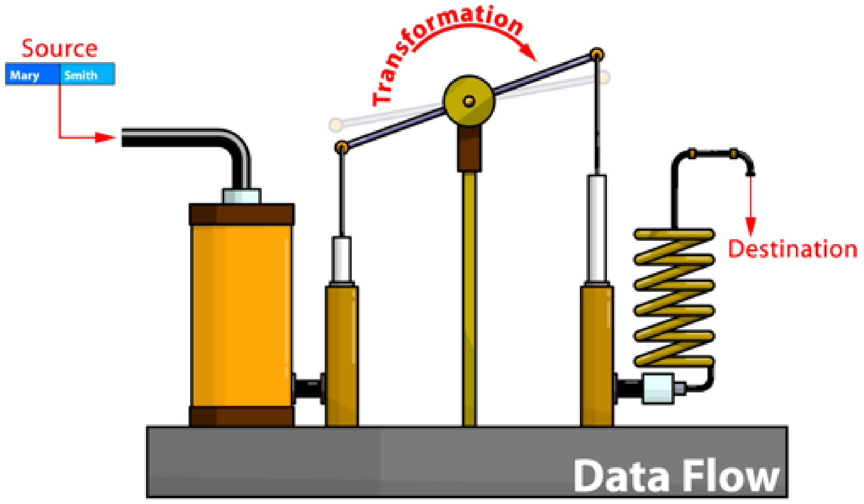
## ETL--Past, Present, and Future

There is no one right way to perform ETL. In the past people have written custom programs in any number of languages such as C, C++, Perl, Python, or used the stored procedures provided by an SQL database. Programs were glued together using BAT files on Windows machines, shell scripts on Unix machines, and everything in between.

But those ways of doing ETL are history. Since then, the tools available now have improved dramatically. Now we don't have figure out how to get Python to read data from an Excel spreadsheet for transformation and placement in an Oracle database.

In this course we'll use **Talend Open Studio** (TOS), an Eclipse-based tool that simplifies development and maintenance of ETL procedures, yet allows enough access to the underlying Java language for power-users to extend and expand its capabilities.

TOS (and other ETL tools) operate with the *data flow* model. You specify sources of data, transformations on that data, and then a destination. This corresponds fairly closely with the standard ETL concepts of extracting data, transforming data, and loading data. One or more data flows are grouped together in a *job*.



ETL tools keep track of *schemas*, a definition of the columns in a data flow that includes data type and sizes. The schema tracks which columns are allowed to be null, and which columns are part of the key.

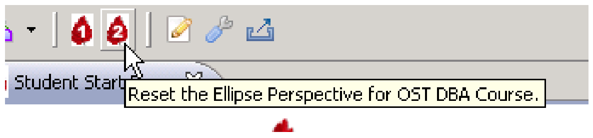
*Schemas* are an important abstraction in the ETL world. They allow us to specify the makeup of our data once, and use that specification in many different components. (We'll learn more about schemas soon.)

What does the future hold? Nobody can say for sure, but it is looking like we will see faster, easier to use, and more reliable ETL solutions that let us focus on interesting problems instead of mundane connections and transformations.

Let's create a sample job to see what we can do.

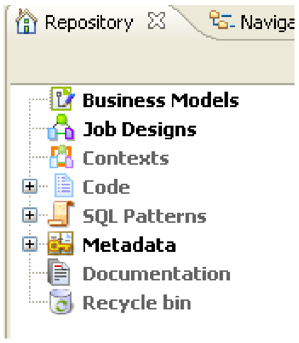
## Getting Started with Talend Open Studio

So far we haven't really used any features of Talend Open Studio. That's about to change. Before we get started, switch to the **second OST perspective** by clicking on the red leaf with a **2** inside it:



**Note** Feel free to resize any widget on your screen. You can always get back to the default perspective by clicking on the red leaf.

On the left side of the screen you will see a tab called "Repository." The text may be truncated to "Rep," depending on the width of your screen.



The repository is where TOS stores all objects related to your project. They are:

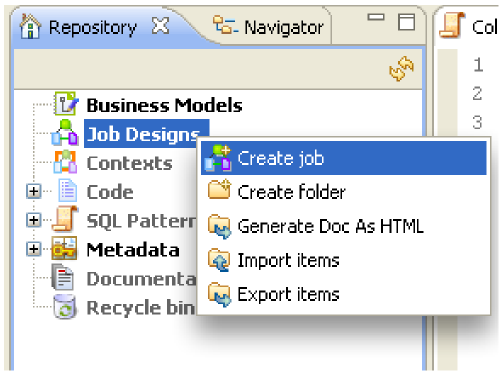
- Business Models - diagrams to document processes or flows.
- Job Designs - implemented processes or flows.
- Contexts - sets of variables or values that are shared across several jobs.
- Code - bits of Java code shared across several jobs.
- SQL Patterns - templates of SQL code that can be used as a basis for queries in jobs.
- Metadata - data about your data - database connections, file layouts, and descriptions of database tables and query results.
- Documentation - storage for word documents, spreadsheets and other items created outside of TOS.
- Recycle bin - last stop for trash, just like the recycle bin in Windows.

To simplify things, for this course we won't use Business Models, Code, SQL Patterns, or Documentation.

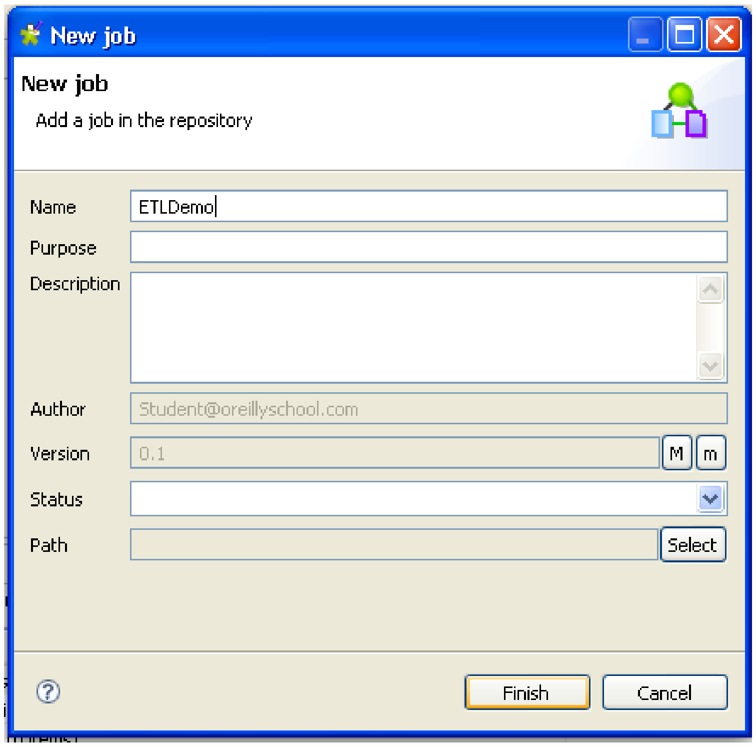
## Your First TOS Job

Now that you've read a little about TOS, it's time to create a simple job.

Right click on **Job Designs** and choose **Create Job**:

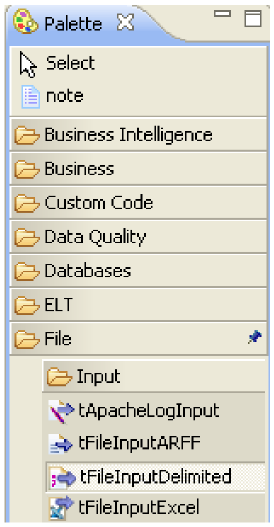


Name the job **ETLDemo** and click finish:



Name: ETLDemo
Purpose:
Description:
Author: Student@oreillyschool.com
Version: 0.1
Status:
Path:

If everything went okay, you will see a blank "canvas" for `Job ETL Demo 0.1` and a new Palette on the lower left:

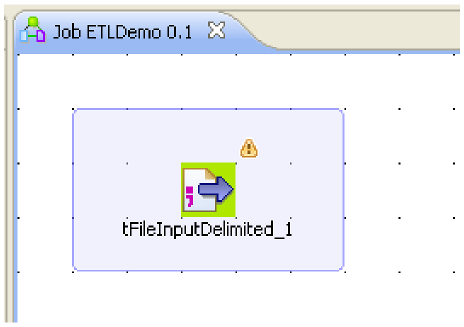


Right now your ETL job is blank, so it doesn't do anything. We need to add a data source. On the Palette, click **File** to expand that category, then click **Input**.

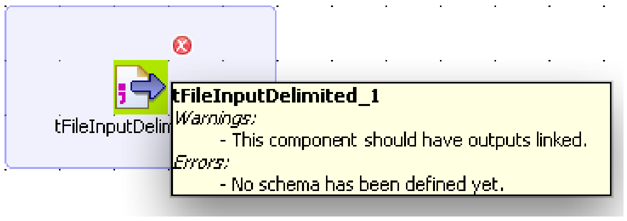
**Note** If you don't see **Input**, click on the up and down arrows.

Click **tFileInputDelimited** once to select, then move your mouse over the canvas. Click the canvas to drop the **tFileInputDelimited** widget.

Your canvas should now look like this:

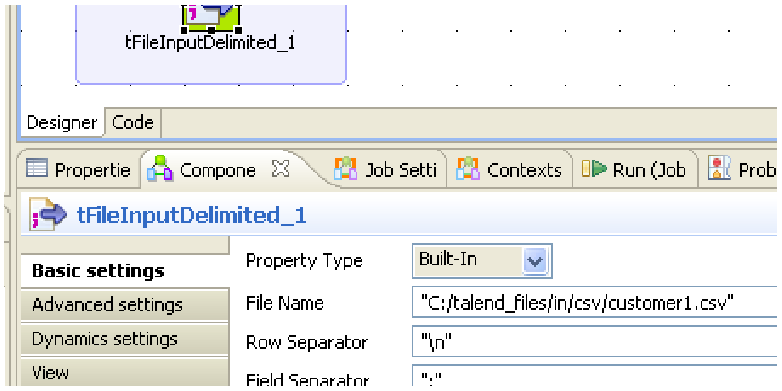


So, what's with that red circle with the X through it? Drag your mouse over that circle, and you'll see this:



tFileInputDelimited_1
*Warnings:*
- This component should have outputs linked.
*Errors:*
- No schema has been defined yet.

The warning and error occur because we haven't set any properties on the **tFileInputDelimited** widget. Let's do that now. Click once in the middle of the **tFileInputDelimited** widget, then switch to the **Component** tab at the bottom of the screen:



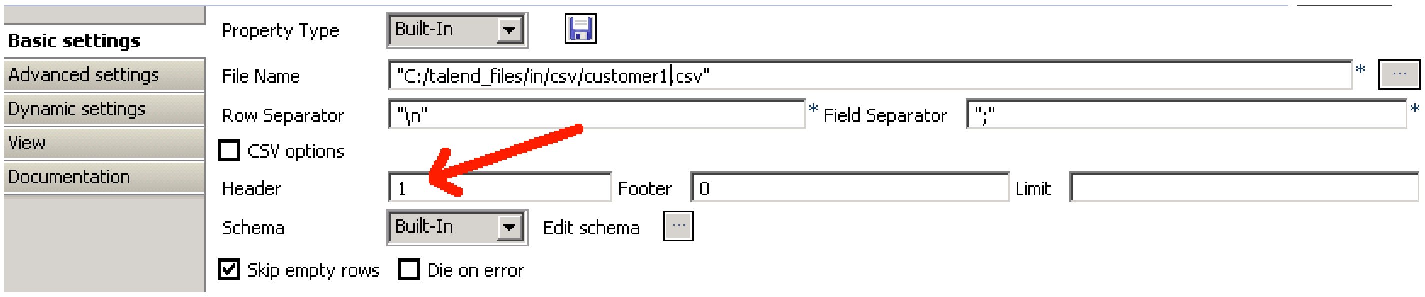
Property Type: Built-In
File Name: "C:/talend_files/in/csv/customer1.csv"
Row Separator: "\n"
Field Separator: ";"

Now you'll see the basic aspects of **tFileInputDelimited** that you can modify. We'll need to change the file to point to a sample CSV input. Change the File Name so it looks like this:

| CODE TO TYPE: |
|---|
| `"C:/talend_files/in/csv/customer1.csv"` |

**WARNING** Make sure you type forward slashes ( // ) instead of the usual back slashes ( \\ ). Under the hood, TOS is using Java to run your transformation; back slashes are used to delimit special characters in Java.

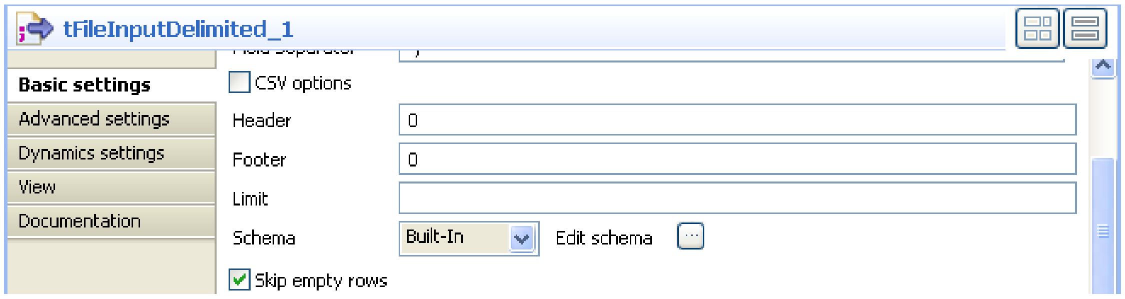
Next, we want TOS to skip over the header row in the file. To do this, change the **0** next to **Header** to a **1**. **1** tells TOS to skip one row at the beginning of the file.



Property Type: Built-In
File Name: "C:/talend_files/in/csv/customer1.csv"
Row Separator: "\n"
Field Separator: ";"
CSV options: ☐
Header: 1
Footer: 0
Limit:
Schema: Built-In
Edit schema: ...
Skip empty rows: ☑
Die on error: ☐

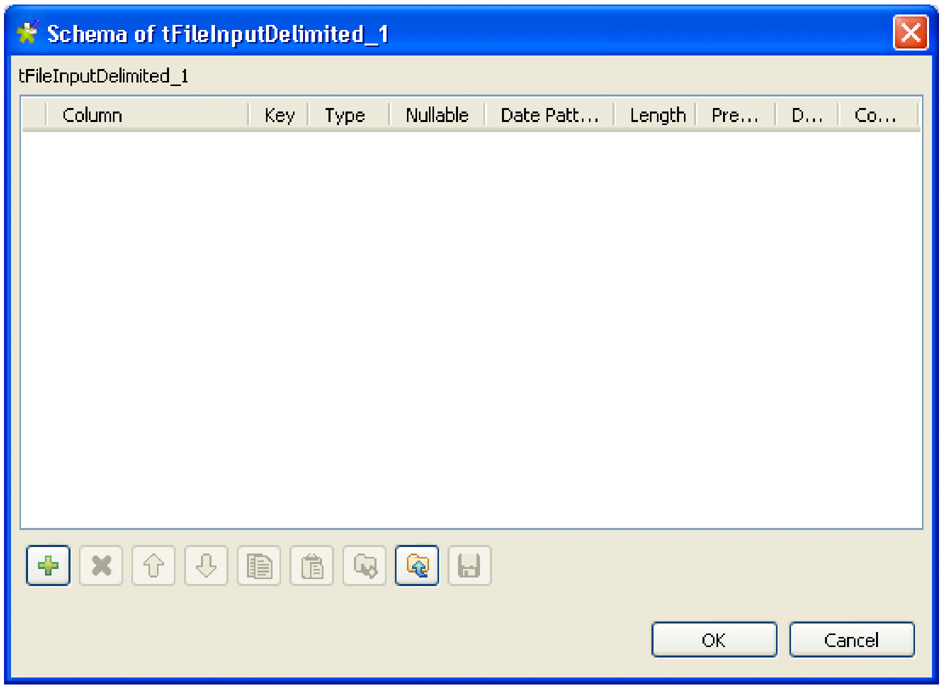
Now that we've specified the input file, we need to specify the schema (structure) of the input file. We do this by clicking the button named "..." next to **Edit Schema**. You may have to scroll the component panel to see the **Schema**.

**Note** Read through the next set of instructions before trying them. TOS uses many *modal* windows (windows that are always on top of other windows), so you won't be able to scroll in this lesson unless you close the modal window.

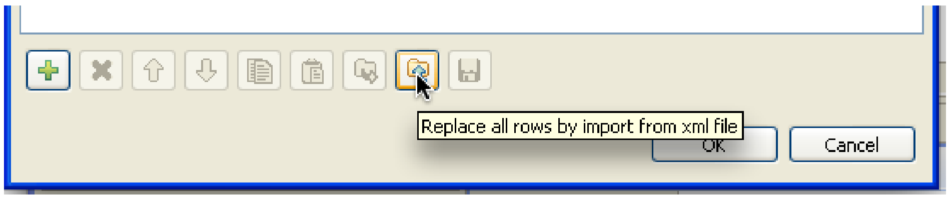


CSV options: ☐
Header: 0
Footer: 0
Limit:
Schema: Built-In
Edit schema: ...
Skip empty rows: ☑

After you click the button, you'll see an empty window:

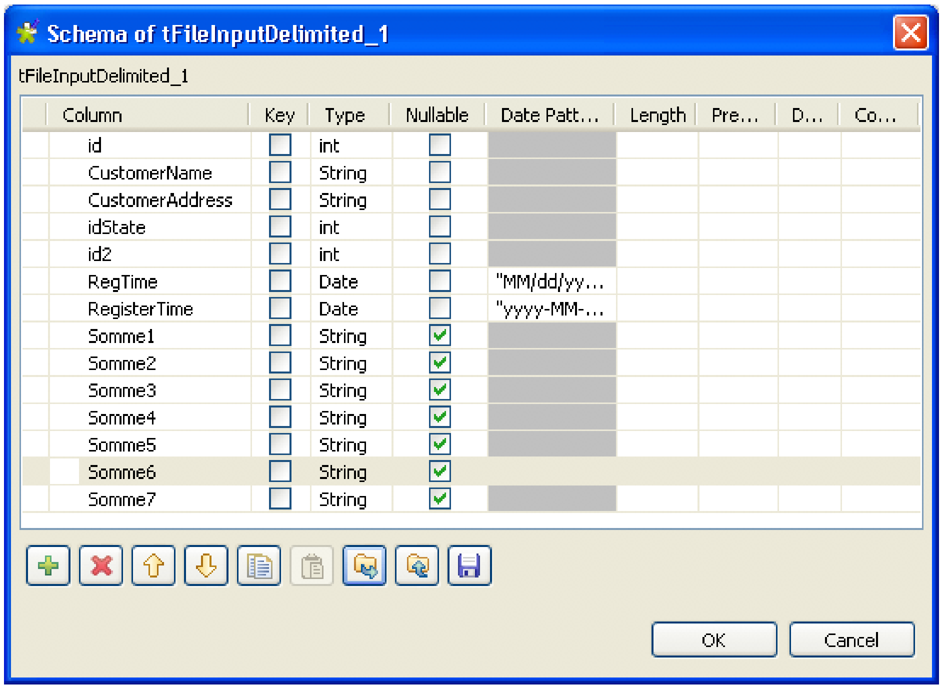


Now we could click the button to add columns to the schema. And we could enter the schema by hand, but instead we'll import it from an XML file. Click on the icon to the left of the floppy disk - it looks like this:



Pick `C:\talend_files\in\csv\customer1_Schema.xml` as the file and click **OK**.

With the schema imported, you'll see the definitions of all of the columns. It will look something like this:



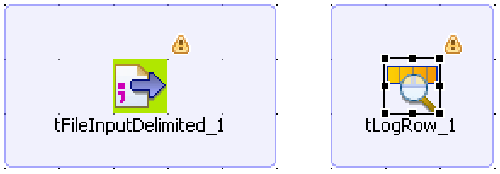
| Column | Key | Type | Nullable | Date Patt... | Length | Pre... | D... | Co... |
|---|---|---|---|---|---|---|---|---|
| id | ☐ | int | ☐ | | | | | |
| CustomerName | ☐ | String | ☐ | | | | | |
| CustomerAddress | ☐ | String | ☐ | | | | | |
| idState | ☐ | int | ☐ | | | | | |
| id2 | ☐ | int | ☐ | | | | | |
| RegTime | ☐ | Date | ☐ | "MM/dd/yy... | | | | |
| RegisterTime | ☐ | Date | ☐ | "yyyy-MM-... | | | | |
| Somme1 | ☐ | String | ✓ | | | | | |
| Somme2 | ☐ | String | ✓ | | | | | |
| Somme3 | ☐ | String | ✓ | | | | | |
| Somme4 | ☐ | String | ✓ | | | | | |
| Somme5 | ☐ | String | ✓ | | | | | |
| Somme6 | ☐ | String | ✓ | | | | | |
| Somme7 | ☐ | String | ✓ | | | | | |

Click **OK** to save your changes. The red circle with the X is gone now, replaced by a warning sign. The warning still exists because we don't have a destination for our data.

**Note** Transformations are not always necessary. Sometimes there isn't anything to do other than read data from one place and place it somewhere else.

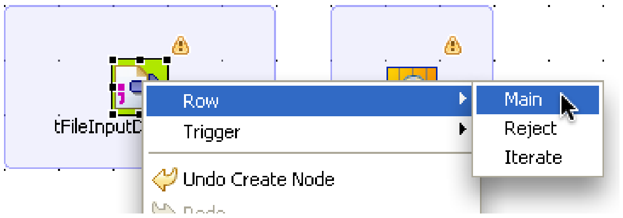
We don't really care where the data ends up, since we are just doing a little test. Instead of putting the data in a database somewhere and then querying the database or putting the data in a different text file, let's use the **tLogRow** widget to display rows on the console.

Click the **File** group to collapse it, then click the **Logs & Errors** group. Click once on **tLogRow** and drag it to the canvas:

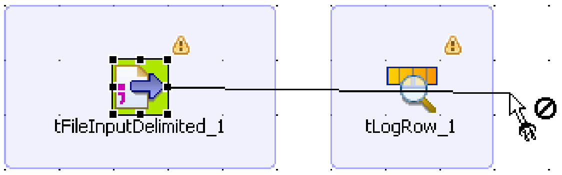


Now both components have warnings. What's the problem? Well, we haven't made any connections between the source of data and the data destination.

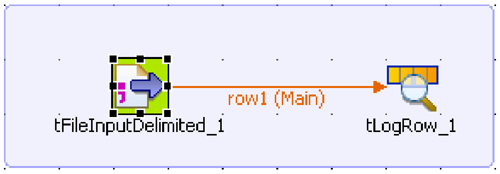
To make a connection, right click on the data source and choose **Row -> Main**:



Drop this connection on **tLogRow**:



The canvas should now look like this:

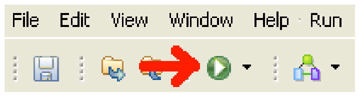


**Note** If you make a mistake, you can always select a component and hit the delete key to remove it. You can also right click on a component and choose **Delete**.

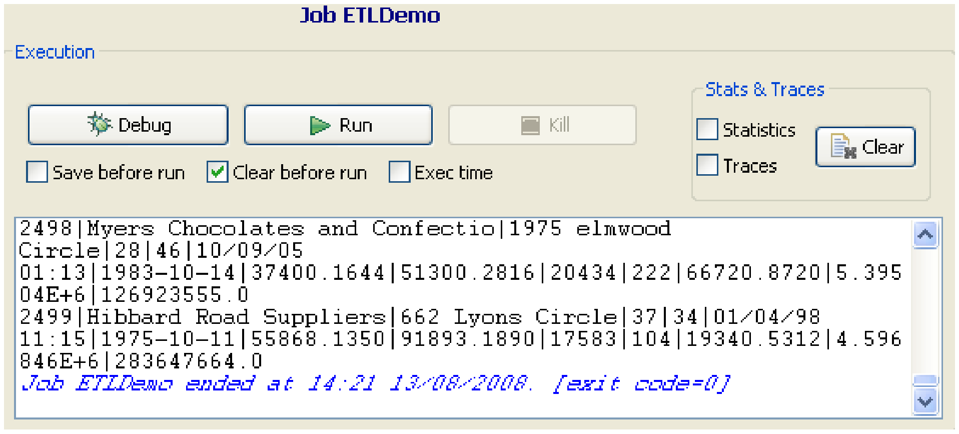
As long as your canvas looks similar to the screen shot, we're ready to run the job!

**Note** Your canvas doesn't have to look exactly the same as our image here. The layout is strictly informational. But the connections *are* important because they define how data flows through the job.

To run the job, click once on the canvas to make sure it is selected, then click the little green "Play" button at the top of the screen:



You'll see some messages and activity, then a whole bunch of data scrolling on the console:



Save before run [✓ Clear before run] Exec time

```
2498|Myers Chocolates and Confectio|1975 elmwood
Circle|28|46|10/09/05
01:13|1983-10-14|37400.1644|51300.2816|20434|222|66720.8720|5.395
04E+6|126923555.0
2499|Hibbard Road Suppliers|662 Lyons Circle|37|34|01/04/98
11:15|1975-10-11|55868.1350|91893.1890|17583|104|19340.5312|4.596
846E+6|283647664.0
Job ETLDemo ended at 14:21 13/08/2008. [exit code=0]
```

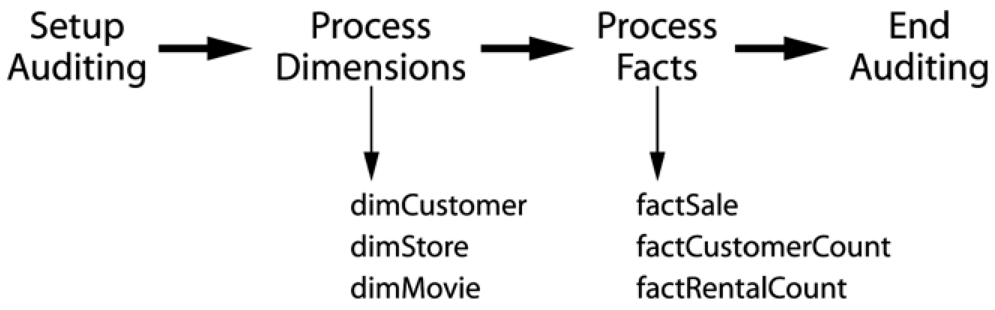
Congratulations! You've completed your first ETL job! In the next lesson we'll write our first *real* ETL job - it will import an Excel spreadsheet for our date dimension. See you there!

# ETL: The Date Dimension
# DBA 3: Data Warehousing Lesson 7

Welcome back! In the last lesson we created our first ETL job. In this lesson we'll create our first "real" job, one that will load our date dimension from an existing Excel spreadsheet.

## Job Structure

Jobs in TOS can be as large or small as you want them to be. Your initial jobs can then execute subsequent jobs. We will use this basic structure for our jobs:



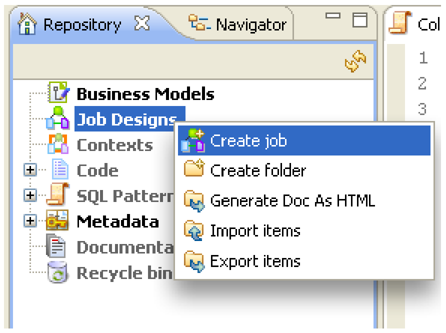
Since dimensions give context to facts, we must process dimensions before we process facts.

This structure makes it possible for us to process the various components of entire data warehouse separately: we might only process dimensions, only process facts, or only process a single dimension or fact. Flexibility like this is great when you're developing a data warehouse. Why bother to process the entire warehouse when you are tracking down an issue with a single dimension, right?

Some data warehouse tasks (like loading our date dimension) occur only once. We'll create a job for each of those types of tasks, but they will not execute as part of our normal warehouse processing.

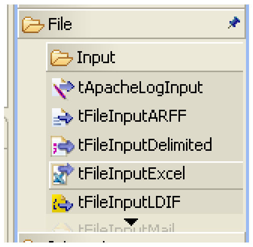
## Loading Data from Excel

Before we can load our spreadsheet to our database, we need to create a new job. Right click on **Job Designs** and choose **Create job**:

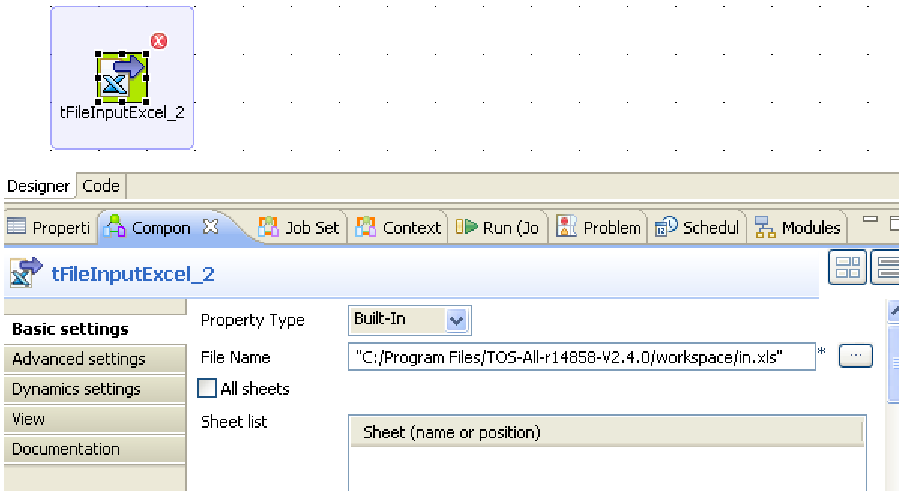


Name the new job **dimDate**.

At this point you will see the blank job canvas on the screen. In the Palette, click to expand **File --> Input**. Find the **tFileInputExcel** component and drag it to your canvas:



Your component may be named `tFileInputExcel_1`, or `tFileInputExcel_2`. Its unique name is generated automatically by TOS, and isn't really useful to us. We'll rename it so it makes more sense. Click in the middle of the **tFileInputExcel** component you just dropped on the canvas, then switch to the **Component** tab:



tFileInputExcel_2

Designer | Code

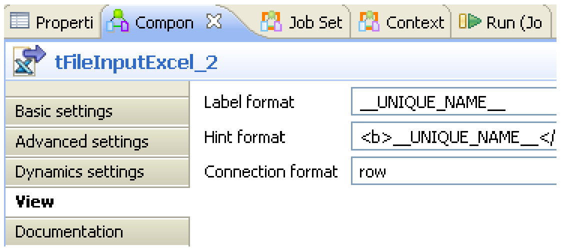
Properti | Compon | Job Set | Context | Run (Jo | Problem | Schedul | Modules

tFileInputExcel_2

Basic settings
Advanced settings
Dynamics settings
View
Documentation

Property Type: Built-In
File Name: "C:/Program Files/TOS-All-r14858-V2.4.0/workspace/in.xls"
All sheets
Sheet list: Sheet (name or position)

Now you might see the **Basic settings** sub tab. To change the name of the component, change to the **View** sub tab:



Properti | Compon | Job Set | Context | Run (Jo

tFileInputExcel_2

Basic settings
Advanced settings
Dynamics settings
**View**
Documentation

Label format: __UNIQUE_NAME__
Hint format: <b>__UNIQUE_NAME__</
Connection format: row

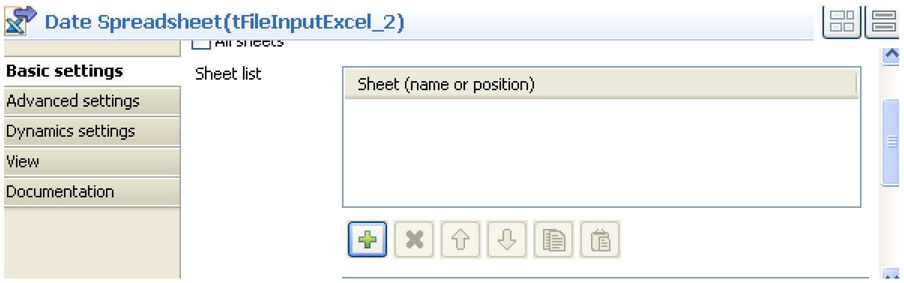
The component's current name is set to `__UNIQUE_NAME__`. Click in the label format box, and change the text to a more meaningful value: **Date Spreadsheet**.

Now we can switch back to the **Basic settings** sub tab. The current **file name** points to an invalid spreadsheet. You can click inside the text box to type in the correct location, or click the ... button to pick the file. Change the file name to:

| CODE TO TYPE: |
|---|
| `C:/talend_files/DateDimension.xls` |

**Note** If you type in the file name, be sure to use forward slashes instead of back slashes.

Excel spreadsheets are also known as *workbooks*, and workbooks contain *sheets*. Suppose your coworker tells you that the data for the date dimension is located on a sheet called **Sheet1**. Scroll down through the basic settings until you see the **Sheet list**. Find the button and click it. Change the default text to **"Sheet1"**. (Don't forget to use those double quotation marks):

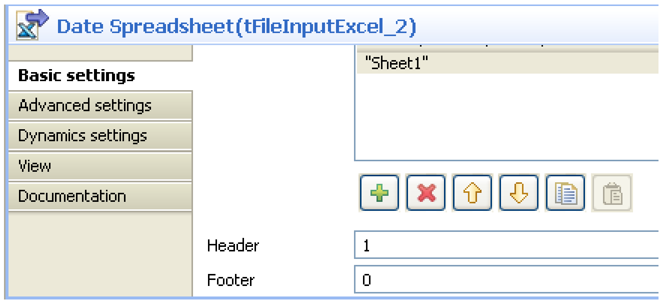


Date Spreadsheet(tFileInputExcel_2)

Basic settings
Advanced settings
Dynamics settings
View
Documentation

Sheet list: Sheet (name or position)

Your spreadsheet also contains a header row. We need to tell TOS about this header row, otherwise it would try to import it as data. Scroll down farther until you see the **Header** and **Footer** section. Change the **Header** text box to **1**:



Date Spreadsheet(tFileInputExcel_2)

Basic settings
Advanced settings
Dynamics settings
View
Documentation

"Sheet1"

Header: 1
Footer: 0

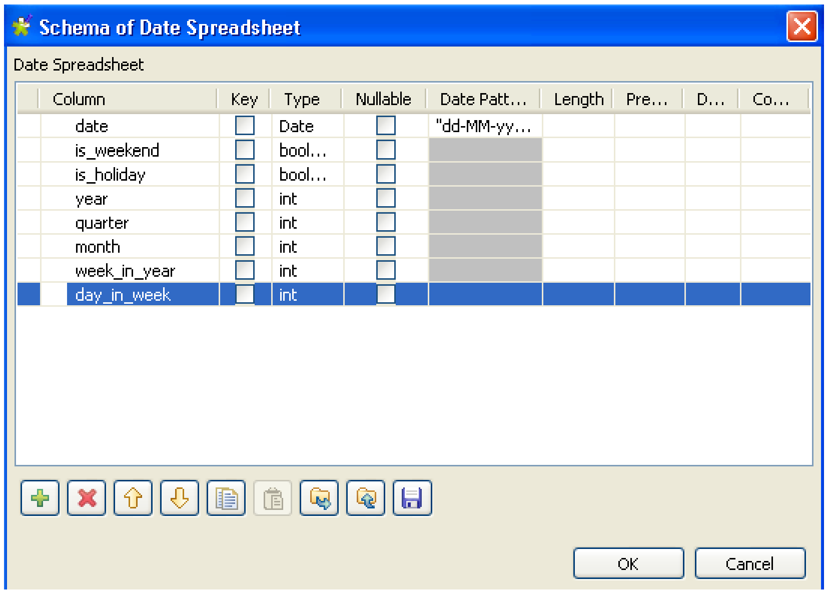
Our component is still in error because we haven't specified the *schema* of our Excel file. Typically, we would specify the schema in the *Metadata* section of the repository, but this spreadsheet is only going to be used in this single job, so in this instance, we'll keep the schema within our component. Our coworker has already provided us with the schema definition:

| Column | Type |
|---|---|
| date | Date |
| is_weekend | Boolean |
| is_holiday | Boolean |
| year | Integer |
| quarter | Character |
| month | Integer |
| week_in_year | Integer |
| day_in_week | Integer |

To edit the schema for our spreadsheet, scroll to the bottom of the component window, and click the ... button located next to **Edit Schema**. Click the button to add a new entry to the schema. Specify the column name and type for each of the columns in the previous table. All of our columns contain data, so we don't want to allow *NULL* values. Uncheck the **Nullable** box for each column.

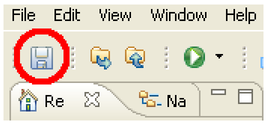
This schema defines the layout of the data in the Excel spreadsheet. It is important to match the column data types and order **precisely**. Bad things can happen if we get the order wrong. For instance, a mismatched data type could confuse TOS so that it wouldn't know whether to interpret the date "2008" as a month or a year. That kind of confusion has the potential to wreak all sorts of havoc on our work.

When you're done, your schema should look like this:



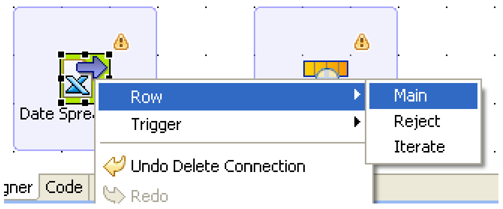
| Column | Key | Type | Nullable | Date Patt... | Length | Pre... | D... | Co... |
|---|---|---|---|---|---|---|---|---|
| date | ☐ | Date | ☐ | "dd-MM-yy... | | | | |
| is_weekend | ☐ | bool... | ☐ | | | | | |
| is_holiday | ☐ | bool... | ☐ | | | | | |
| year | ☐ | int | ☐ | | | | | |
| quarter | ☐ | int | ☐ | | | | | |
| month | ☐ | int | ☐ | | | | | |
| week_in_year | ☐ | int | ☐ | | | | | |
| day_in_week | ☐ | int | ☐ | | | | | |

Click "OK" to close the window. TOS puts an asterisk * next the filename when your job has changes that have not been saved. When you make changes to your job, get in the habit of saving them. You can save your file in one of two ways: use the **Save** command on the **File** menu or click the floppy disk icon on the toolbar:



Let's test our component to make sure it's working properly. We can test it using the **tLogRow** component. In the Palette, click to expand the **Logs & Errors** tab, then click and drag **tLogRow** to your canvas.

Link the **Date Spreadsheet** component to the **tLogRow** component by right-clicking on your **Date Spreadsheet** component, and choosing **Row -> Main**: Drop the link on **tLogRow**:



**Note** For more information on linking components, review the previous lesson.

Run your job by clicking on the button. If everything is set up correctly, you'll see output that looks like this:

OBSERVE:

```
Starting job dimDate at 16:36 14/10/2008.
```

```
01-01-2000|true|false|2000|1|1|1|7
02-01-2000|true|false|2000|1|1|2|1
03-01-2000|false|false|2000|1|1|2|2
04-01-2000|false|false|2000|1|1|2|3
... lines omitted ...
27-12-2050|false|false|2050|4|12|53|3
28-12-2050|false|false|2050|4|12|53|4
29-12-2050|false|false|2050|4|12|53|5
30-12-2050|true|false|2050|4|12|53|6
31-12-2050|true|false|2050|4|12|53|7
Job dimDate ended at 16:36 14/10/2008. [exit code=0]
```

We are off to a great start!

## Adding Columns to our Data Flow

Before we load this data into our `dimDate` table, let's review the columns in dimDate. Open a terminal and connect to your personal database. Run the following command:

CODE TO TYPE:

```
describe dimDate;
```

Run the query. Your results should look something like this:

OBSERVE:

```
mysql> describe dimDate;
+------------+-------------+------+-----+---------+-------+
| Field      | Type        | Null | Key | Default | Extra |
+------------+-------------+------+-----+---------+-------+
| date_key   | int(11)     | NO   | PRI | NULL    |       |
| date       | date        | NO   |     | NULL    |       |
| year       | smallint(6) | NO   |     | NULL    |       |
| quarter    | char(2)     | NO   |     | NULL    |       |
| month      | tinyint(4)  | NO   |     | NULL    |       |
| day        | tinyint(4)  | NO   |     | NULL    |       |
| week       | tinyint(4)  | NO   |     | NULL    |       |
| is_weekend | tinyint(1)  | YES  |     | NULL    |       |
| is_holiday | tinyint(1)  | YES  |     | NULL    |       |
| month_name | varchar(9)  | YES  |     | NULL    |       |
| day_name   | varchar(9)  | YES  |     | NULL    |       |
+------------+-------------+------+-----+---------+-------+
11 rows in set (0.00 sec)
```

Now take a look back at the schema for our data source. It contains the following columns: `date, is_weekend, is_holiday, year, quarter, month, week_in_year, day_in_week`. But it looks like our spreadsheet is missing the date_key, and day (of the month). And while the spreadsheet includes a column for day_in_week, our date dimension does not have that column.

ETL is used specifically to address these kinds of differences. Sources of data are rarely in the format we want for our data warehouse. They might be missing some information, or contain information that we don't need.

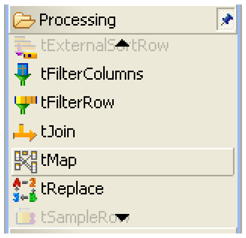
TOS has many different components, and several could be used to add columns to our data flow. For this example, we'll use a component called **tMap**. The map component is an all-purpose tool in TOS that allows you to perform joins, filters, transformations, and data splits. We will use it for transformations.

Before we add **tMap** to our canvas, we need to delete the link between **Date Spreadsheet** and **tLogRow_1**. We will leave **tLogRow_1** on our canvas for now, since we will use it to test our changes.

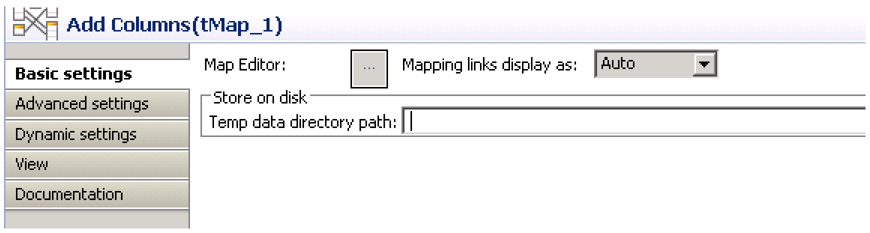
To delete the link, you can right click on the **row1 (Main)** arrow between **Date Spreadsheet** and **tLogRow_1** and choose Delete, or click once on the **row1 (Main)** arrow and press the Delete key on your keyboard.

**Note** If you accidentally delete the wrong component, just choose "Undo" from the Edit menu.

Next, expand the Palette, and click on **Processing**. Scroll down until you find the **tMap** component. Add it to your canvas, and feel free to rearrange the other components:



Rename **tMap_1** to something more meaningful - change it to **Add Columns**. Next link **Date Spreadsheet** to **Add Columns**. Before we link **Add Columns** to our logging component, let's add our new columns. Click once on **Add Columns**, then change the tab to **Component**. The window will look like this:



Add Columns(tMap_1)

Basic settings | Advanced settings | Dynamic settings | View | Documentation

Map Editor: ... Mapping links display as: Auto

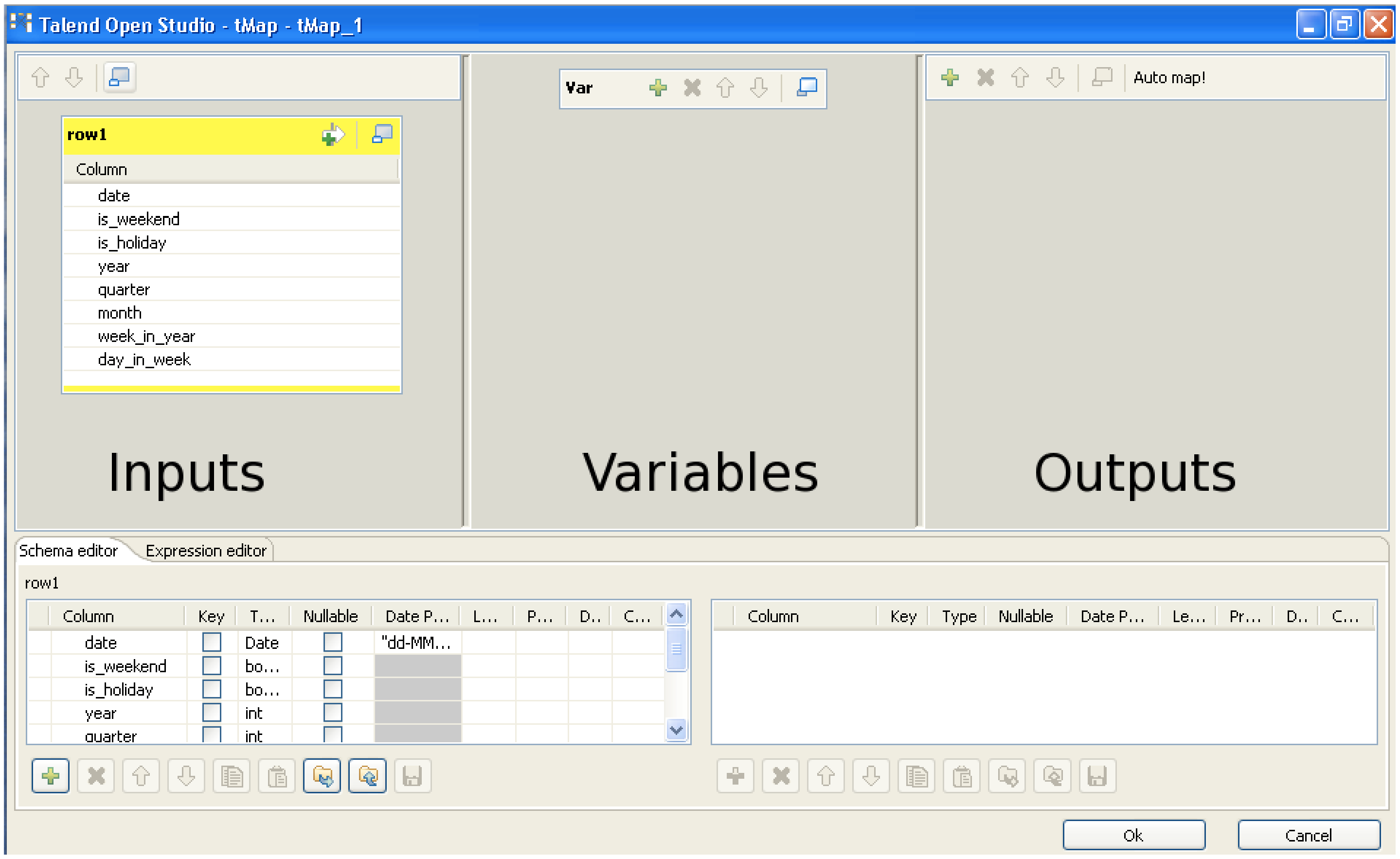
Store on disk

Temp data directory path:

**Note** The **tMap** window is another *modal* window - so you won't be able to scroll course content while you're working with **tMap**. Make sure you click **OK** to close the **tMap** window so your progress is retained, and save your job often!

To edit our transformation, click on **Basic settings** and then click on the ... button, next to **Map Editor**. You'll probably want to expand the new window to fill your entire screen.

The editor for **tMap** has three distinct areas: *Inputs*, *Variables*, and *Outputs*:



row1

| Column |
|---|
| date |
| is_weekend |
| is_holiday |
| year |
| quarter |
| month |
| week_in_year |
| day_in_week |

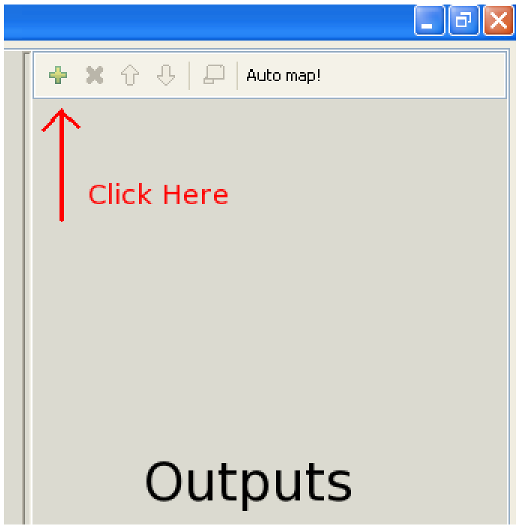
row1

| Column | Key | T... | Nullable | Date P... | L... | P... | D.. | C... |
|---|---|---|---|---|---|---|---|---|
| date | ☐ | Date | ☐ | "dd-MM... | | | | |
| is_weekend | ☐ | bo... | ☐ | | | | | |
| is_holiday | ☐ | bo... | ☐ | | | | | |
| year | ☐ | int | ☐ | | | | | |
| quarter | ☐ | int | ☐ | | | | | |

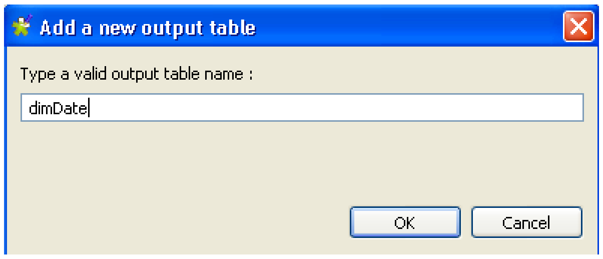
| Column | Key | Type | Nullable | Date P... | Le... | Pr... | D.. | C... |
|---|---|---|---|---|---|---|---|---|

- Each flow into **tMap** shows up in the *Input* section. You must have at least one input.
- The *Variables* section lets you set or modify variables, which is useful when you want to create counters.
- The *Output* section lets you define the way input rows are passed on to the next component. You may have multiple outputs.

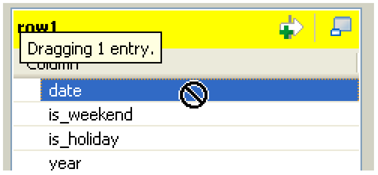
For our current dimension, we won't use variables, only inputs and outputs. Click the ✚ button in the *Outputs* section:



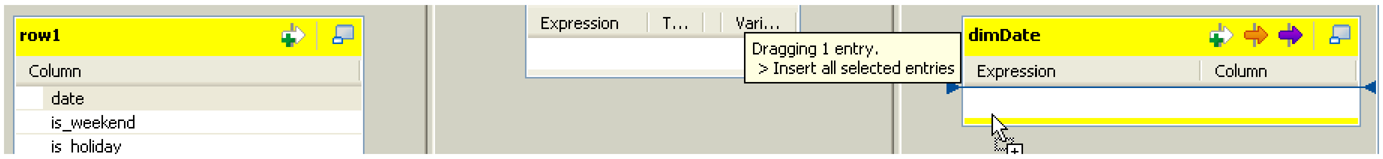
Name your new output **dimDate**:



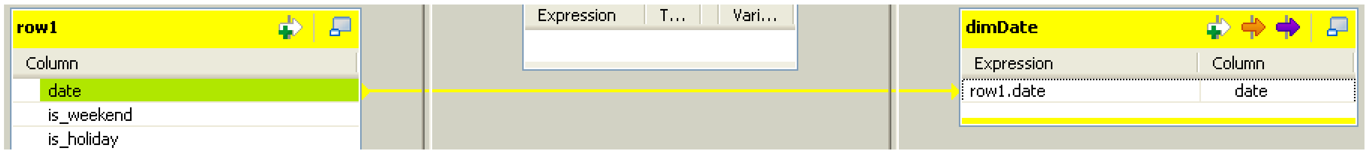
Now we need to link our input columns to output columns. Click and hold on the **date** input column:



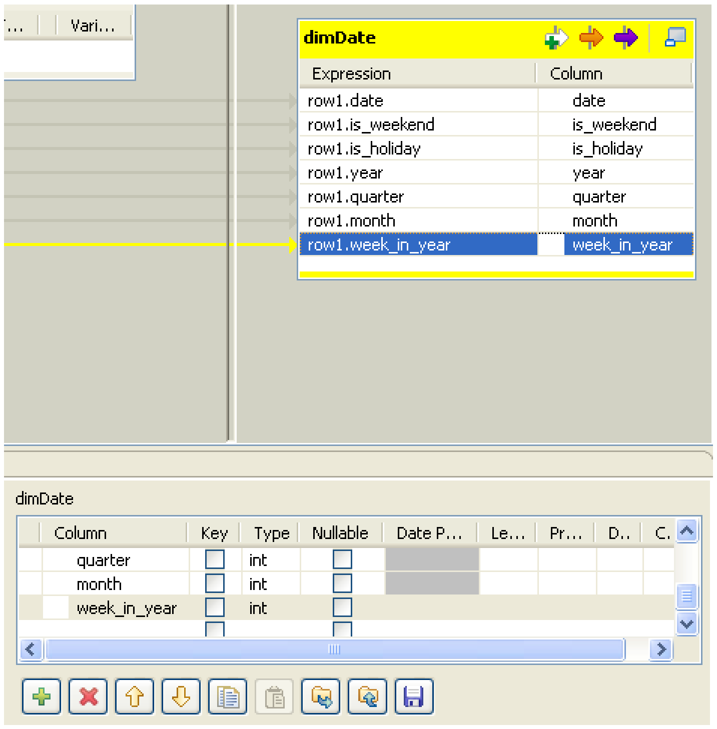
Now drag the column over to the **dimDate** output that was just created:



Once you drop the column, you'll see the link:



Repeat these steps for all of the remaining columns except `day_in_week` -- we are not using that column. When you're done, you'll see this:



dimDate

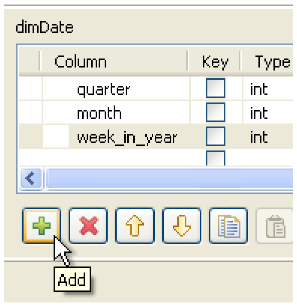
| Expression | Column |
|---|---|
| row1.date | date |
| row1.is_weekend | is_weekend |
| row1.is_holiday | is_holiday |
| row1.year | year |
| row1.quarter | quarter |
| row1.month | month |
| row1.week_in_year | week_in_year |

dimDate

| Column | Key | Type | Nullable | Date P... | Le... | Pr... | D... | C... |
|---|---|---|---|---|---|---|---|---|
| quarter | ☐ | int | ☐ | | | | | |
| month | ☐ | int | ☐ | | | | | |
| week_in_year | ☐ | int | ☐ | | | | | |

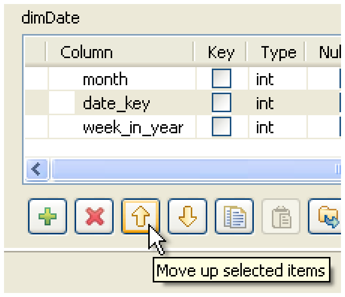
Next, let's add a new column for `date_key`. Primary keys in data warehouses are often implemented using auto-increment columns, but it's much more convenient to have `date_key` in a coded format, such as `yyyyMMdd`. Using this format, a value of `20080101` would represent January 1st, 2008.

To add a new column to the **output**, click the ➕ button in the schema section:


dimDate

| Column | Key | Type |
|---|---|---|
| quarter | ☐ | int |
| month | ☐ | int |
| week_in_year | ☐ | int |

Name this column **date_key**, change its type to **int**, and uncheck the **Nullable** checkbox. Eventually the order of columns in our output must match our actual `dimDate` table, so we might as well move the **date_key** column to the very top now. We can do this using the up and down arrows to the right of the add button:


dimDate

| Column | Key | Type | Nul |
|---|---|---|---|
| month | ☐ | int | |
| date_key | ☐ | int | |
| week_in_year | ☐ | int | |

Next, change the name of the output column from **week_in_year** to **week**.

Add another new column called **day**, change its type to **int**, and uncheck the **Nullable** checkbox. Next, add two new string columns: **month_name** and **day_name**.

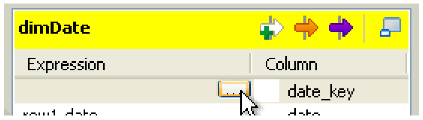
Finally, reorder the remaining columns so they match **YOUR** `dimDate` table. The order will be similar to this:

1. date_key
2. date
3. year
4. quarter
5. month
6. day
7. week

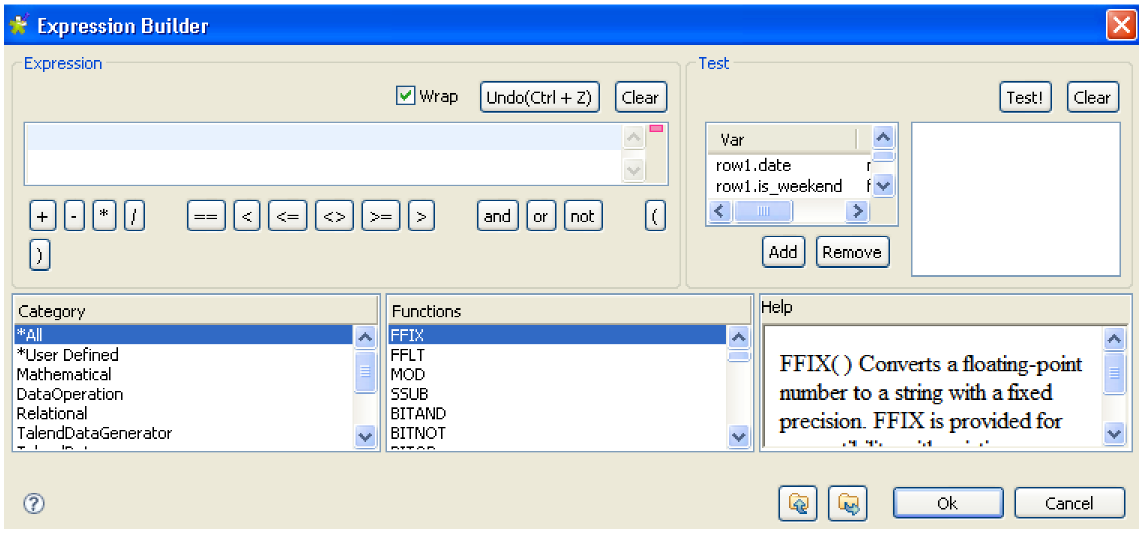
8. is_weekend
9. is_holiday
10. month_name
11. day_name

TOS inserts data into your `dimDate` table using the column order you specify. If you specify the incorrect order, TOS may insert data into the wrong column.

With **date_key** at the top, we can now use an expression to set the value of **date_key**. Click the ... button in the expression box:



The expression builder looks like this:



FFIX( ) Converts a floating-point number to a string with a fixed precision. FFIX is provided for

Because our TOS project is based on Java, expressions are also written in Java. This gives us lots of power. You can use Java string functions to create some very powerful expressions. The bottom half of the expression window is a catalog of some common expressions that you can use. We'll use a function from the `TalendDate` category to convert our `date` to "yyyyMMdd" format, then convert that string into an integer, ready for the database. (Don't worry if you're not quite an expert using Java - we'll provide you with the expressions you need for this course. If you're interested in learning more about Java, check out the Java Certificate Series.)

In the expression builder, type in this code:

CODE TO TYPE:

```
Integer.parseInt(
    TalendDate.formatDate("yyyyMMdd",row1.date)
)
```

Click "OK" to save your expression. Next, edit the expression for the **day** column. Type the code below into the expression builder:

CODE TO TYPE:

```
row1.date.getDate()
```

Next, set the expression for `month_name`. Type the code below into the expression builder:

CODE TO TYPE:

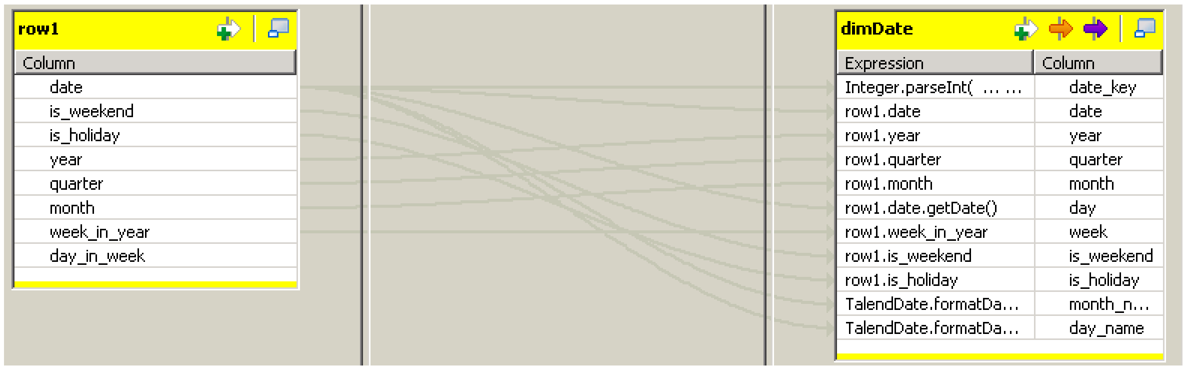
```
TalendDate.formatDate("MMMM",row1.date)
```

Finally, set the expression for `day_name`. Type the code below into the expression builder:

CODE TO TYPE:

```
TalendDate.formatDate("EEEE",row1.date)
```

We've used the `date` input column to come up with several output columns. Graphically, TOS displays this with multiple arrows from `date` going to different rows in the output:



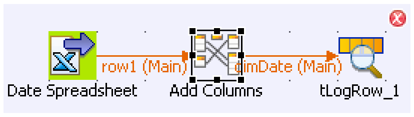
| row1 Column |
|---|
| date |
| is_weekend |
| is_holiday |
| year |
| quarter |
| month |
| week_in_year |
| day_in_week |

| dimDate Expression | Column |
|---|---|
| Integer.parseInt( ... ... | date_key |
| row1.date | date |
| row1.year | year |
| row1.quarter | quarter |
| row1.month | month |
| row1.date.getDate() | day |
| row1.week_in_year | week |
| row1.is_weekend | is_weekend |
| row1.is_holiday | is_holiday |
| TalendDate.formatDa... | month_n... |
| TalendDate.formatDa... | day_name |

With our columns complete, we are free to close the map editor. Click "OK" and then save your job.

We are nearly there! Link **Add Columns** to **tLogRow_1**.

Hey, it looks like we have a problem. There's that little red circle with an X in it:

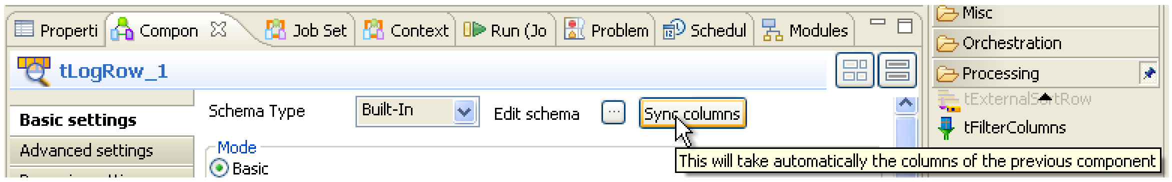


If you hover over the component, you'll see this error:

OBSERVE:

```
tLogRow_1
Errors:
    - The schema from the input link "dimDate" is different from the schema defined in the component.
```

The problem is that **tLogRow_1** still has the schema from its earlier connection. To fix this problem, click **tLogRow_1** once to select it, then change to the `Component` tab. You'll see a button called *Sync Columns* - click it.



This will take automatically the columns of the previous component

With that problem fixed, we are free to run the job.

After you run the job, you will see new output:

OBSERVE:

```
Starting job dimDate at 16:34 15/06/2009.
20000101|01-01-2000|2000|1|1|1|1|true|false
20000102|02-01-2000|2000|1|1|2|2|true|false
20000103|03-01-2000|2000|1|1|3|2|false|false
... lines omitted ...
20501229|29-12-2050|2050|4|12|29|53|false|false
20501230|30-12-2050|2050|4|12|30|53|true|false
20501231|31-12-2050|2050|4|12|31|53|true|false
Job dimDate ended at 16:34 15/06/2009. [exit code=0]
```

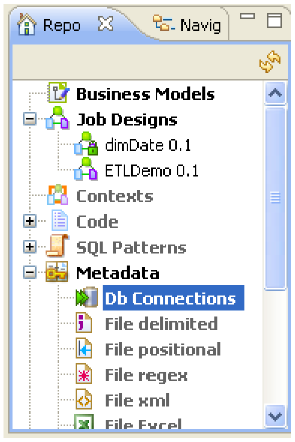
It's looking really good now.

## Adding Data to dimDate

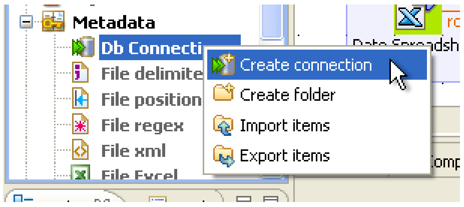
So far, we've read data from our Excel spreadsheet and added two new columns to the data flow. The next step is to deposit our data into the `dimDate` table.

We'll be connecting to MySQL often, so it would be good to keep MySQL connection information in one place. We can do this using the **Metadata** section of our repository.

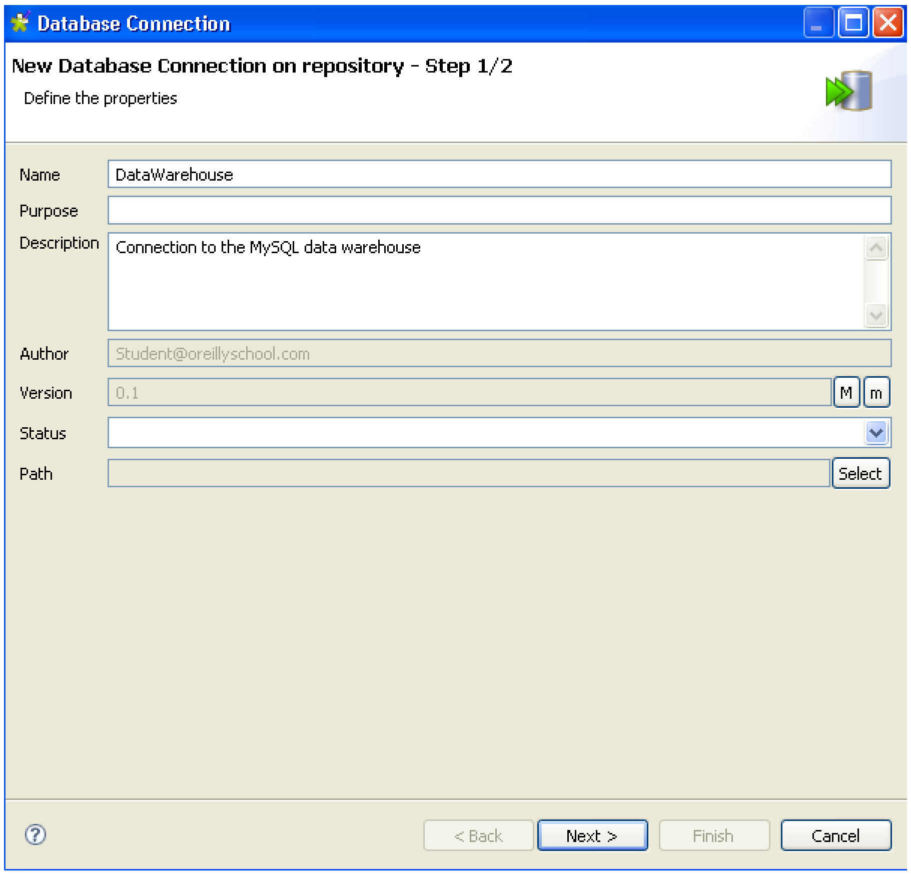
To create a connection, click to expand the **Metadata** section of the repository:



Next, right-click on **Db Connections** and choose **Create Connection**:



Spaces are not allowed in connection names, so give your connection this name: **DataWarehouse**. If you want to, you can leave the *purpose* and *description* fields blank, keep *version* set at 0.1, and leave status unselected. Those columns are just additional metadata for your connection:


Database Connection

New Database Connection on repository - Step 1/2

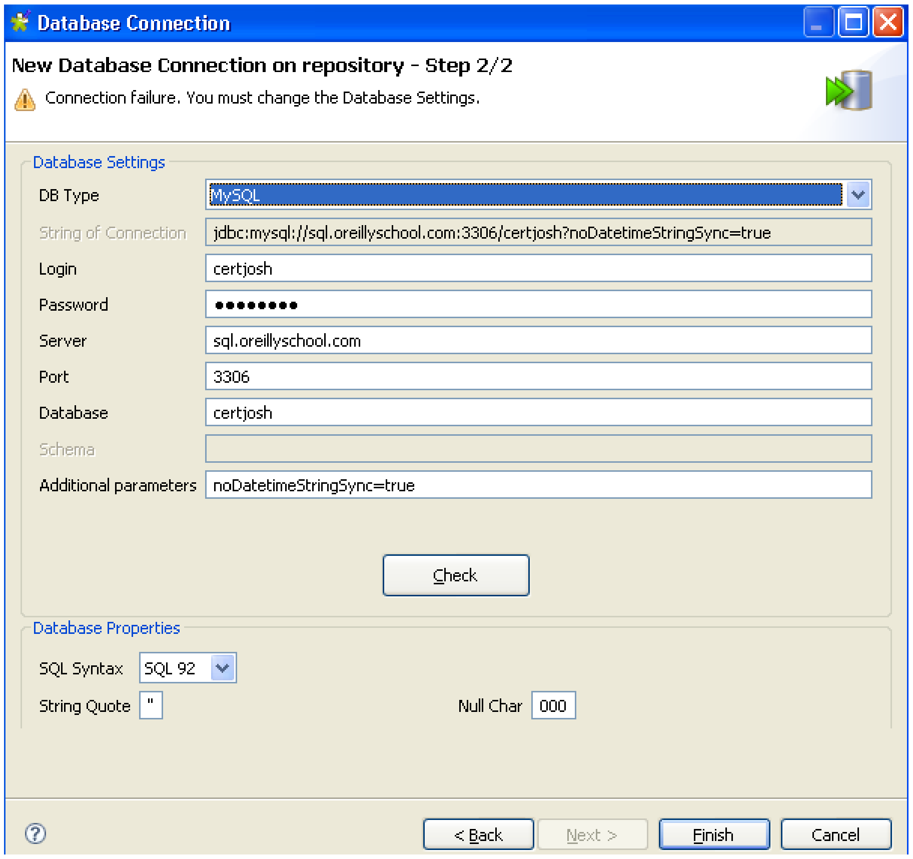
Define the properties

| | |
|---|---|
| Name | DataWarehouse |
| Purpose | |
| Description | Connection to the MySQL data warehouse |
| Author | Student@oreillyschool.com |
| Version | 0.1 |
| Status | |
| Path | |

When you're done, click **Next >**.

In the final screen, choose **MySQL** as the DB Type, and enter this information:

- Login: *your username*
- Password: *your password*
- Server: sql.oreillyschool.com
- Port: 3306
- Database: *your username*



Connection failure. You must change the Database Settings.

Database Settings

| | |
|---|---|
| DB Type | MySQL |
| String of Connection | jdbc:mysql://sql.oreillyschool.com:3306/certjosh?noDatetimeStringSync=true |
| Login | certjosh |
| Password | •••••••• |
| Server | sql.oreillyschool.com |
| Port | 3306 |
| Database | certjosh |
| Schema | |
| Additional parameters | noDatetimeStringSync=true |

Database Properties

SQL Syntax: SQL 92

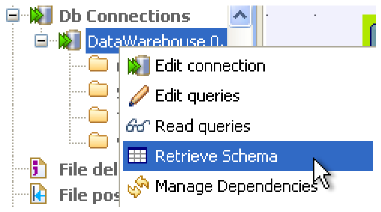
String Quote: "

Null Char: 000

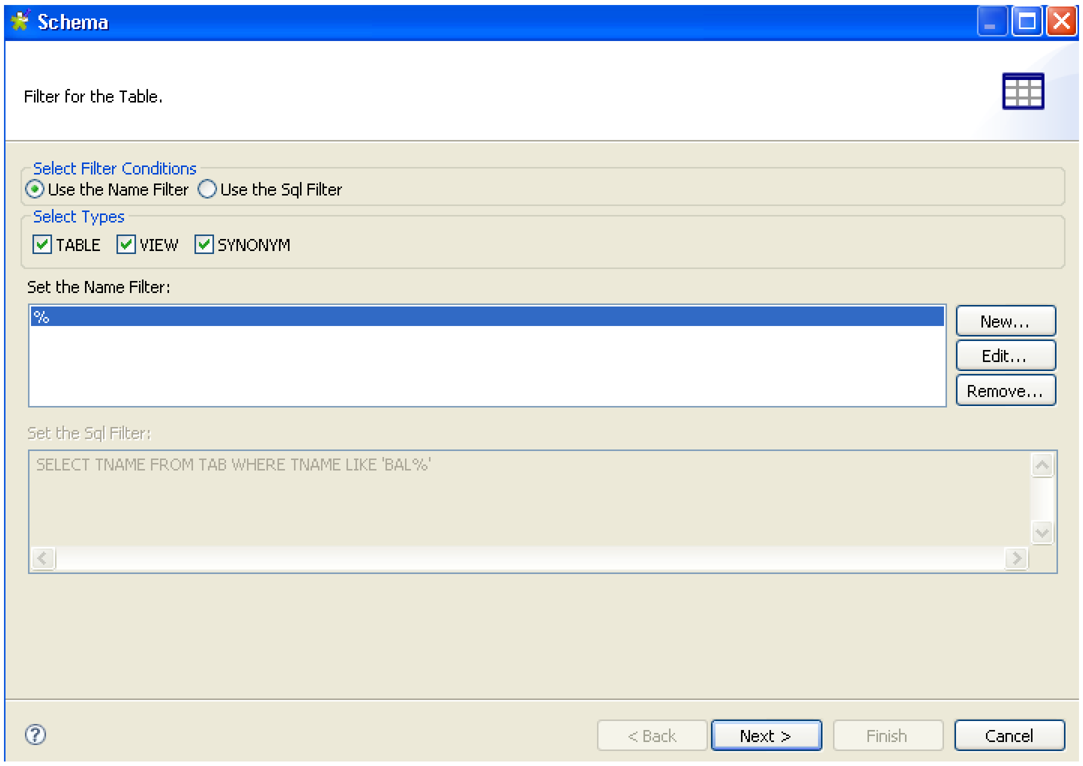
Click the **Check** button to try to connect to your database. If your connection is good, you'll see the message "`DataWarehouse`" `connection successful`. Click **Finish** to save your connection.

**Note** If you edit your database connection, TOS will ask you if you want to propagate the modifications to all jobs. Choose yes - you want your changes to apply everywhere.

Since we are storing our database connection information in the repository, we should also store our table schema in the repository. Right-click on your database connection, and choose **Retrieve Schema**:



We don't need to filter our schema, so click **Next >** at the bottom of the window:


Schema

Filter for the Table.

Select Filter Conditions
[✓ Use the Name Filter] Use the Sql Filter

Select Types
[✓ TABLE] [✓ VIEW] [✓ SYNONYM]

Set the Name Filter:

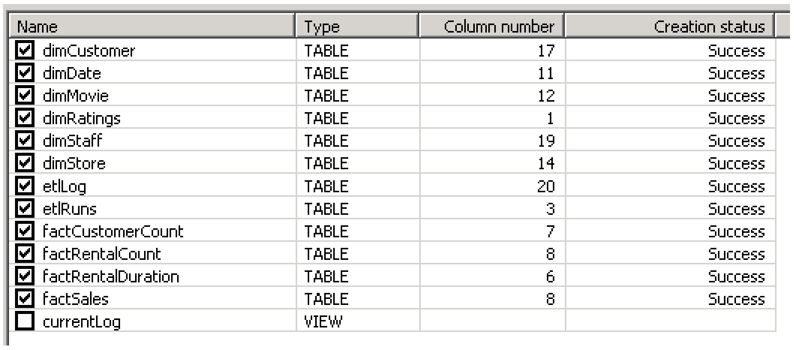
%

Set the Sql Filter:

SELECT TNAME FROM TAB WHERE TNAME LIKE 'BAL%'

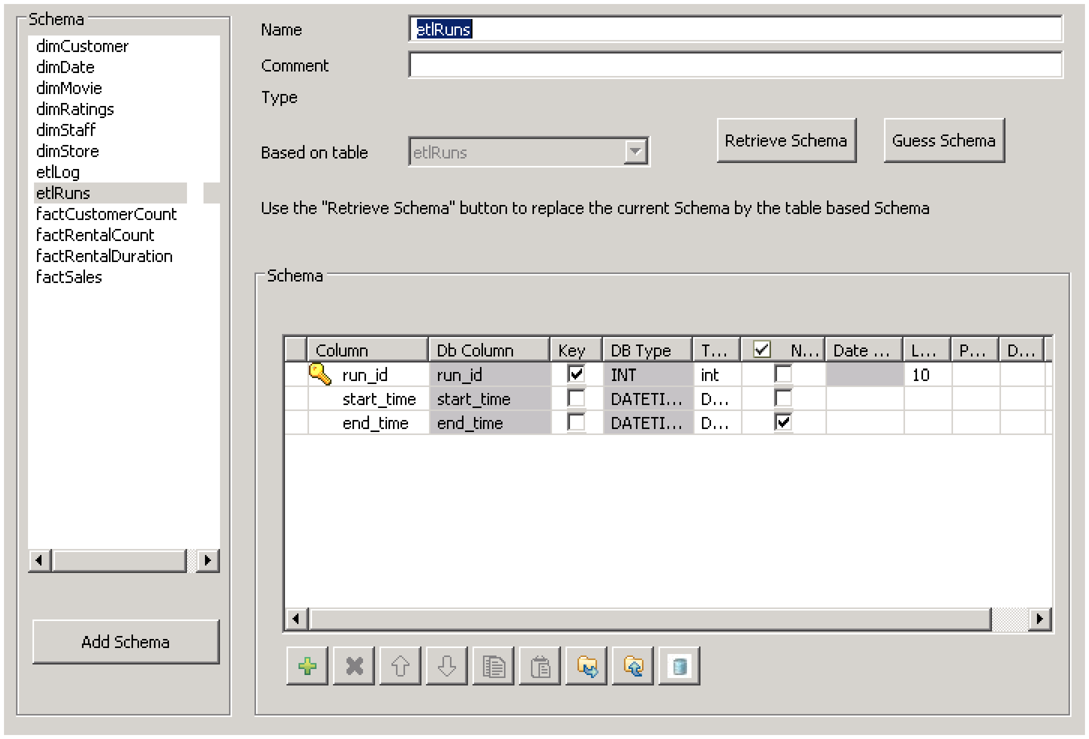
In the next window, scroll through your database until you come across the objects for this course:

- dimCustomer
- dimDate
- dimMovie
- dimStore
- etlLog
- etlRuns
- factCustomerCount
- factRentalCount
- factSales



| Name | Type | Column number | Creation status |
|---|---|---|---|
| ✓ dimCustomer | TABLE | 17 | Success |
| ✓ dimDate | TABLE | 11 | Success |
| ✓ dimMovie | TABLE | 12 | Success |
| ✓ dimRatings | TABLE | 1 | Success |
| ✓ dimStaff | TABLE | 19 | Success |
| ✓ dimStore | TABLE | 14 | Success |
| ✓ etlLog | TABLE | 20 | Success |
| ✓ etlRuns | TABLE | 3 | Success |
| ✓ factCustomerCount | TABLE | 7 | Success |
| ✓ factRentalCount | TABLE | 8 | Success |
| ✓ factRentalDuration | TABLE | 6 | Success |
| ✓ factSales | TABLE | 8 | Success |
| currentLog | VIEW | | |

Click **Next >**. In this final screen you can review the schema and make any necessary changes. We don't need to change anything, so click **Finish**:


Schema

dimCustomer
dimDate
dimMovie
dimRatings
dimStaff
dimStore
etlLog
etlRuns
factCustomerCount
factRentalCount
factRentalDuration
factSales

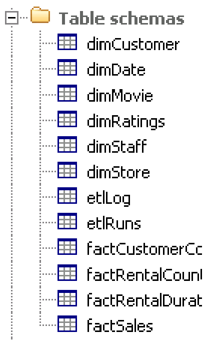
Name: etlRuns
Comment:
Type
Based on table: etlRuns

Use the "Retrieve Schema" button to replace the current Schema by the table based Schema

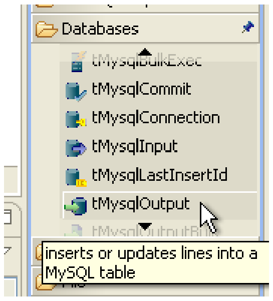
Schema

| Column | Db Column | Key | DB Type | T... | N... | Date ... | L... | P... | D... |
|---|---|---|---|---|---|---|---|---|---|
| run_id | run_id | ✓ | INT | int |  |  | 10 |  |  |
| start_time | start_time |  | DATETI... | D... |  |  |  |  |  |
| end_time | end_time |  | DATETI... | D... | ✓ |  |  |  |  |

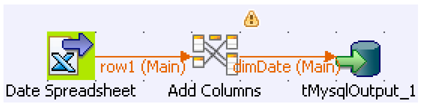
Now the repository will show the tables associated with our connection:



With our connection setup, we can swap out **tLogRow_1** with a MySQL output. First, delete **tLogRow_1** from your canvas. Next, expand the **Databases** section of the Palette, then expand the **MySQL** sub-section:



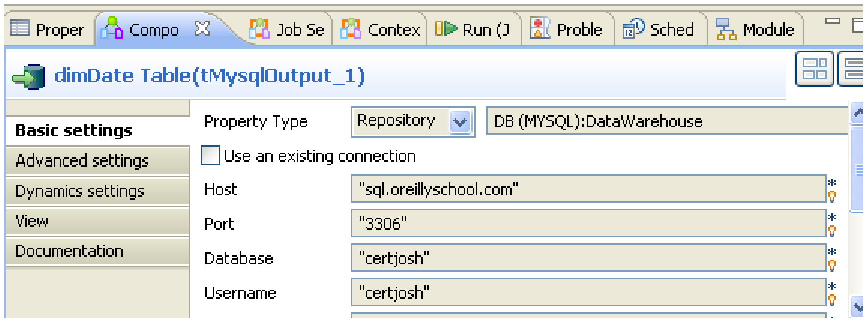
Drag **tMysqlOutput** to your canvas, and link it from the **Add Columns** component. Your canvas will now look something like this:



Next, select **tMySQLOutput_1**, and switch to the **Component** tab. Change the name of **tMySQLOutput_1** to **dimDate Table**.

Now switch back to the **Basic settings** tab. By default, TOS assumes you are going to save the database connection information inside of the component. This is known as a *Built-In* property type.

Our connection information is stored in the repository, so change the **Property Type** from *Built-In* to *Repository*. We only have one database connection in the repository, and TOS selects it for us automatically:



Property Type: Repository — DB (MYSQL):DataWarehouse
Use an existing connection
Host: "sql.oreillyschool.com"
Port: "3306"
Database: "certjosh"
Username: "certjosh"

We're almost done. Next, we'll tell our output connection where and how to place the data. Scroll down the **Basic Settings** tab to see the remaining properties.

Set the **Table** to *dimDate*. We don't want duplicate rows in our table, and we want to reload our **Table** completely each time we execute this job, so set **Action on table** to *Clear Table*. This essentially executes a `DELETE FROM dimDate;` before inserting data into `dimDate`. We want to insert data (without deleting or updating anything), so set **Action on data** to *Insert*. Finally, if there is any problem, we want to stop the job immediately, so check the box next to **Die on error**:

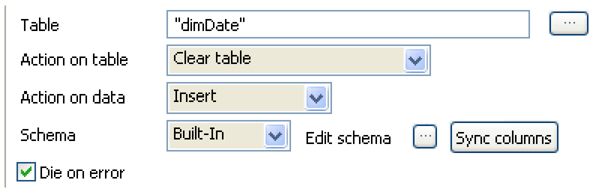


Table: "dimDate"
Action on table: Clear table
Action on data: Insert
Schema: Built-In
[✓] Die on error

Save your job. Now you're ready to run it! It might take a minute or two to read from the Excel spreadsheet and transfer everything to your database. When the job is complete, you'll see this output:

OBSERVE:

```
Starting job dimDate at 13:29 16/10/2008.
Job dimDate ended at 13:31 16/10/2008. [exit code=0]
```

We can double-check by running a quick query. Switch back to the terminal, run this command:

CODE TO TYPE:

```
SELECT * from dimDate
LIMIT 0, 10;
```

If your job ran successfully, and your query is correct, you will see this:

OBSERVE:

```
mysql> SELECT * from dimDate
    -> LIMIT 0, 10;
+----------+------------+------+---------+-------+-----+------+------------+------------+------------+
| date_key | date       | year | quarter | month | day | week | is_weekend | is_holiday | month_name |
+----------+------------+------+---------+-------+-----+------+------------+------------+------------+
| 20000101 | 2000-01-01 | 2000 |       1 |     1 |   1 |    1 |          1 |          0 | January    |
| 20000102 | 2000-01-02 | 2000 |       1 |     1 |   2 |    2 |          1 |          0 | January    |
| 20000103 | 2000-01-03 | 2000 |       1 |     1 |   3 |    2 |          0 |          0 | January    |
| 20000104 | 2000-01-04 | 2000 |       1 |     1 |   4 |    2 |          0 |          0 | January    |
| 20000105 | 2000-01-05 | 2000 |       1 |     1 |   5 |    2 |          0 |          0 | January    |
| 20000106 | 2000-01-06 | 2000 |       1 |     1 |   6 |    2 |          0 |          0 | January    |
| 20000107 | 2000-01-07 | 2000 |       1 |     1 |   7 |    2 |          1 |          0 | January    |
| 20000108 | 2000-01-08 | 2000 |       1 |     1 |   8 |    2 |          1 |          0 | January    |
| 20000109 | 2000-01-09 | 2000 |       1 |     1 |   9 |    3 |          1 |          0 | January    |
| 20000110 | 2000-01-10 | 2000 |       1 |     1 |  10 |    3 |          0 |          0 | January    |
+----------+------------+------+---------+-------+-----+------+------------+------------+------------+
10 rows in set (0.00 sec)
```

It looks great!

**dimDate** is almost complete. Because most facts reference `dimDate` and need to lookup a `date_key` based on a real date, we need to add an index on the `date` column. Run the following command against your personal database:

CODE TO TYPE:

```
ALTER TABLE dimDate ADD INDEX(date);
```

If you typed everything correctly, you'll see this:

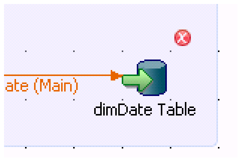
OBSERVE:

```
Query OK, 18628 rows affected (0.46 sec)
Records: 18628  Duplicates: 0  Warnings: 0
```

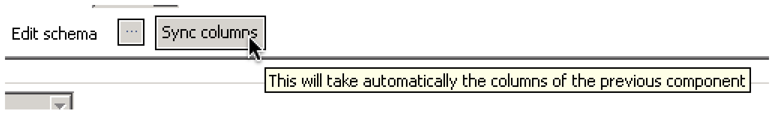
## If you run into problems...

You have a couple of debugging options if TOS gives you an error when you try to run your job or your `SELECT * from dimDate` query doesn't return any results.

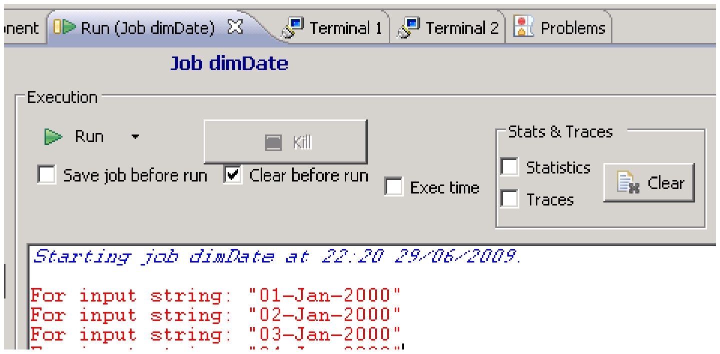
First, take a look at the job itself for any red X's:



If you see a red X, one of your components contains an error. Hover over the component and TOS should let you know what the problem is. If your schemas differ between your components, click on the **Sync Columns** button of your last component:


Edit schema
Sync columns
This will take automatically the columns of the previous component

Next, check the job output for red text that looks like this:



```
Starting job dimDate at 22:20 29/06/2009.

For input string: "01-Jan-2000"
For input string: "02-Jan-2000"
For input string: "03-Jan-2000"
```

TOS is telling us that it was unable to interpret the value for `01-Jan-2000`. Chances are, the input schema is incorrect - either the columns are out of order or a data type is incorrect. Check your schema again.

If you don't see any results from the query, other than `0 rows in set (0.00 sec)` - double-check your output schema in **tMap**. Your columns may be out of order or you might have an incorrect data type.

If you are unable to find the source of a problem, you can always contact your mentor at **learn@oreillyschool.com**.

You've accomplished a lot in this lesson - you extracted an Excel spreadsheet, transformed the columns in the spreadsheet, and loaded the data into the warehouse: **ETL**! In the next lesson we'll press on with our ETL and the remaining dimensions. See you then!

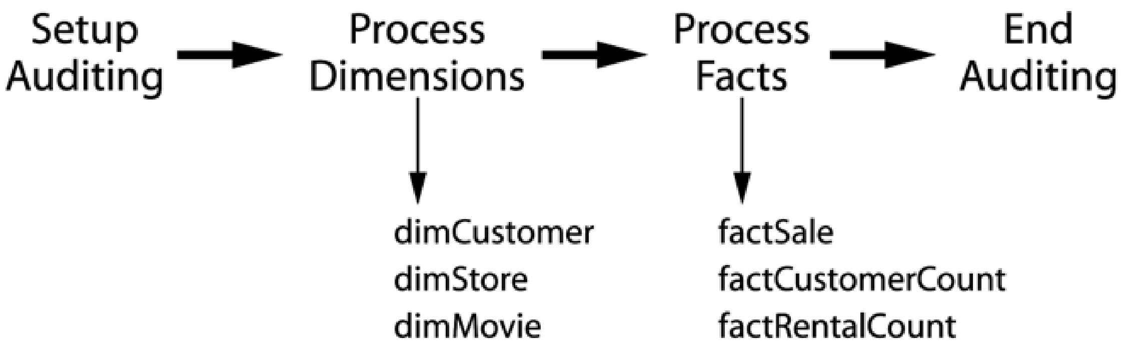
# Basic Dimension Processing
# DBA 3: Data Warehousing Lesson 8

Welcome back. In the last lesson we created our first dimension: `dimDate`. In this lesson we'll create another basic dimension: `dimMovie`.

## Loading dimMovie

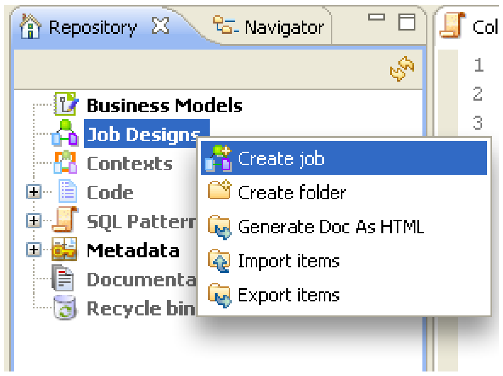
### Job Structure

Before we dive into our movie dimension, let's review our job structure:



Our job for `dimMovie` is a *sub job*, executed by `Process Dimensions`, which is a sub-job executed by `Daily Warehouse Update`. This structure allows us a great deal of flexibility; we can reload the entire data warehouse by running a single job, or we can reload the facts alone by running a single job, or reload only a single fact.

Let's set up this structure. Create a new job:



Name this job **ProcessDataWarehouse**. (Remember spaces are not allowed in job names.)

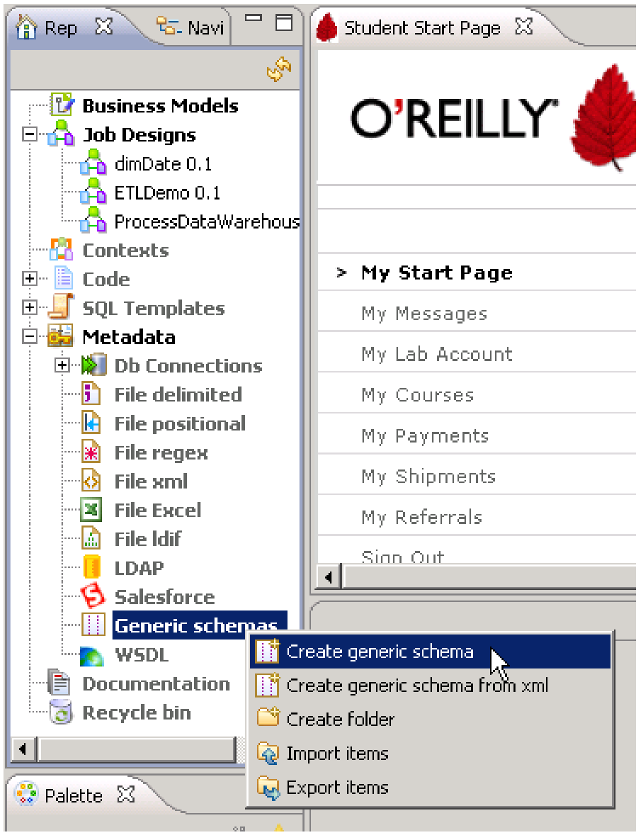
### Pre and Post Job

TOS has two special components: **tPreJob** and **tPostJob**. These components let you specify what happens before a job is run and after the job is completed. We'll use a **tPreJob** component to run our stored procedure `etl_StartRun` which we created back in lesson 5. We'll use a **tPostJob** component to run the stored procedure `etl_EndRun`.

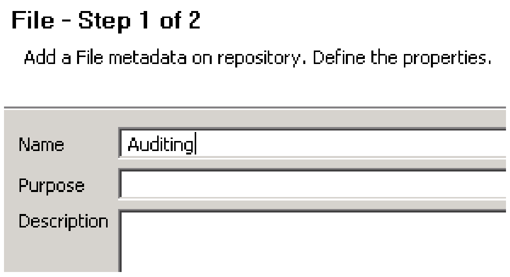
Our stored procedure returns two columns. TOS interprets these key/value pairs as part of the *context* of job execution. In TOS, the *context* is a grouping of global variables that can be referenced within the job. Essentially, these key/value pairs are turned into variables in each job and then enable us to use the `run_id` in other parts of our job.

In the last lesson we added a database connection, and used it to store the schemas for several tables. TOS also lets us store *generic* schemas, which are schemas not associated with specific connections. Let's create a generic schema that defines the columns returned by the `etl_StartRun` stored procedure.

First, expand the **Metadata** section of the repository, then right-click on **Generic schemas**. Choose **Create generic schema**:



Name it **Auditing** and click next:

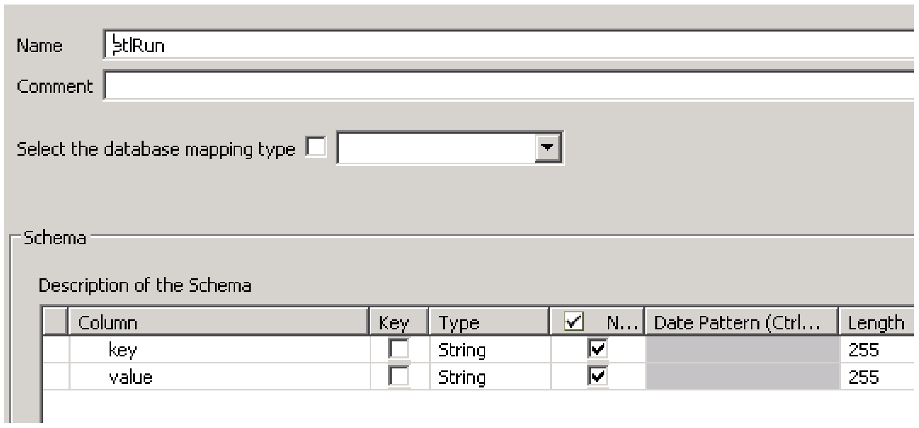


File - Step 1 of 2

Add a File metadata on repository. Define the properties.

| | |
|---|---|
| Name | Auditing |
| Purpose | |
| Description | |

Finally, name the schema **etlRun** instead of `metadata`, and add two columns to the schema. The columns will be named **key** and **value**; they are strings of length 255 and they allow nulls. When you're done, click **Finish**:



Name: etlRun

Comment:

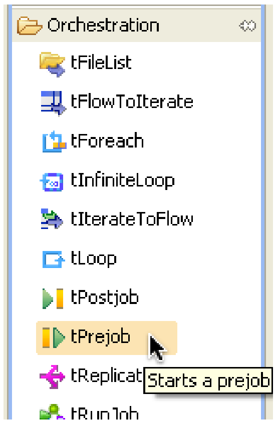
Select the database mapping type

Schema

Description of the Schema

| Column | Key | Type | ✓ N... | Date Pattern (Ctrl... | Length |
|---|---|---|---|---|---|
| key | ☐ | String | ✓ | | 255 |
| value | ☐ | String | ✓ | | 255 |

Now we're ready to start working on the actual job. Drag a **tPrejob** component from the palette to your canvas. It's located under the **Orchestration** section:



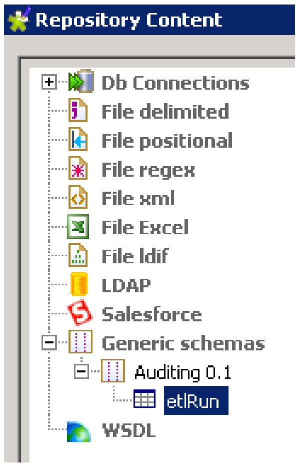
Next, drag a **tMysqlInput** component to the canvas. It's located under **Databases --> MySQL.** Position it to the right of **tPrejob**. Right-click on **tPrejob** and link **Trigger --> On Component OK** from **tPrejob** to **tMysqlInput**.

Change the name **tMysqlInput** to **etl_StartRun** using the **View** menu in the **Component** tab.

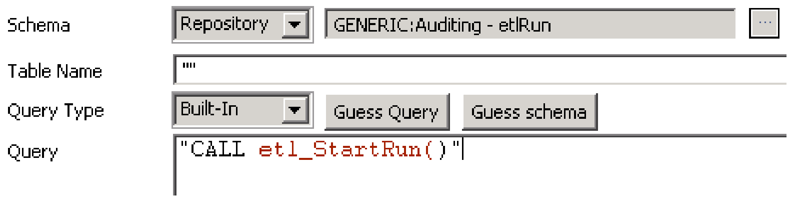
Click on **etl_StartRun**, then choose the **Component** tab below your canvas. Set these properties:

- Property Type: `Repository - DB (MYSQL):DataWarehouse` *(select your database connection from the repository)*
- Schema - `Repository - GENERIC:Auditing - etlRun` *(select the metadata you just created from the repository)*
- Query - `"CALL etl_StartRun()"` - (don't forget the quotation marks.)

Choose the generic schema you just created by clicking on the ... to the **left** of Edit Schema. Navigate to the schema that was just added and select **etlRun**:



Your screen should look something like this:



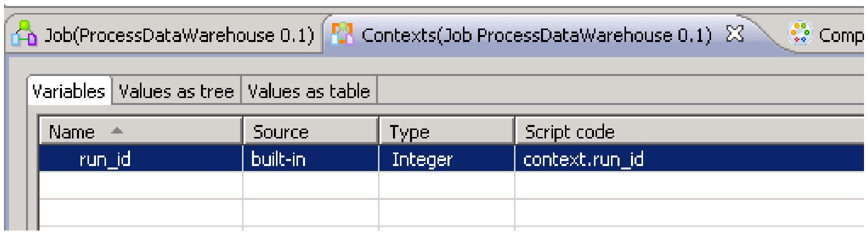
| | | |
|---|---|---|
| Schema | Repository | GENERIC:Auditing - etlRun |
| Table Name | "" | |
| Query Type | Built-In | Guess Query / Guess schema |
| Query | "CALL etl_StartRun()" | |

So far, so good. Next, drop a **tContextLoad** component on the canvas. This component is located under the **Misc** tab in the palette. This component accepts data and loads it to the current *context*.

Link the main row output of **etl_StartRun** to **tContextLoad**.

To extract the `run_id` from the context, we'll need to give TOS a little more information. At the bottom of the window, click on the **Contexts** tab. Make sure you are on the **Variables** sub-tab, then click the + to add a new entry.

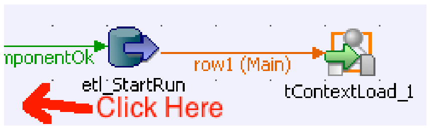
The Name is **run_id**, its source is **built-in**, its type is **Integer**, and the script code is `context.run_id`. When you are done, the tab should look like this:


Job(ProcessDataWarehouse 0.1) | Contexts(Job ProcessDataWarehouse 0.1) | Comp

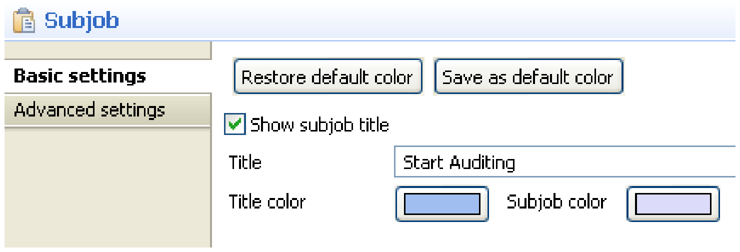
Variables | Values as tree | Values as table

| Name | Source | Type | Script code |
|---|---|---|---|
| run_id | built-in | Integer | context.run_id |

Finally, left-click in the blue area surrounding **etl_StartRun** to select the sub job.



Make sure you're on the **Component** tab, then click to select the box **Show subjob title**. Type in an appropriate title:


Subjob

Basic settings | Advanced settings

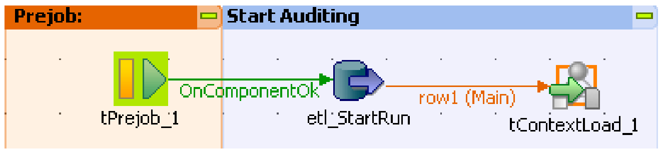
Restore default color | Save as default color

✓ Show subjob title

Title: Start Auditing

Title color | Subjob color

Your canvas should now look something like this:

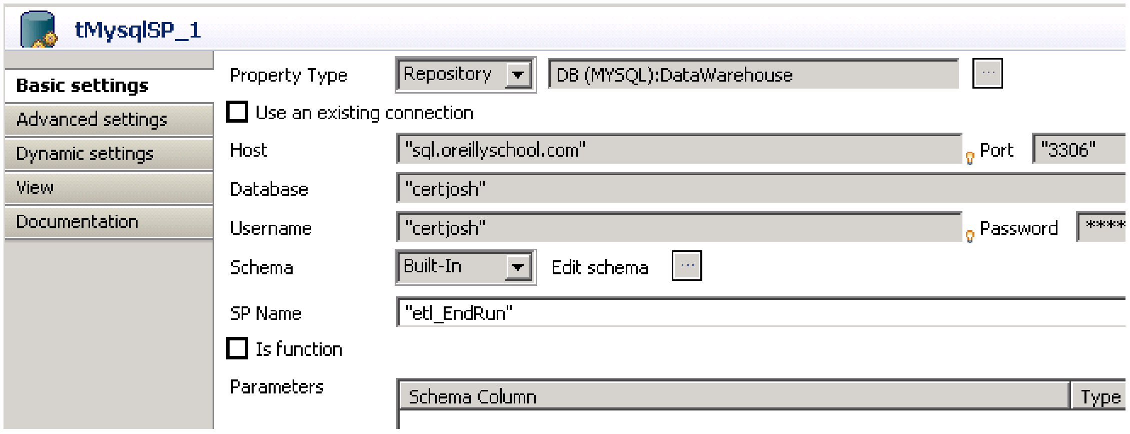


If you haven't saved the job, now is a good time to do that.

Now that we have our start in place, we need to put our end in place. Since our stored procedure doesn't output any rows, we'll use the **tMysqlSP** component instead of **tMysqlInput**. **tMysqlSP** is used specifically to call stored procedures. Okay, now execute these steps:

1. Drag a **tPostjob** component from the palette (under **orchestration**) to your canvas.
2. Drag a **tMysqlSP** component to the right of **tPostjob** (under **Databases --> MySQL**).
3. Link the **OnComponentOK** from **tPostjob** to **tMysqlSP**.
4. Set the connection properties on **tMysqlSP**.
5. Set the **SP Name** to **etl_EndRun**.

Your **tMysqlSP** component's settings should look like this:


tMysqlSP_1

Basic settings | Advanced settings | Dynamic settings | View | Documentation

Property Type: Repository | DB (MYSQL):DataWarehouse

☐ Use an existing connection

Host: "sql.oreillyschool.com" Port: "3306"

Database: "certjosh"

Username: "certjosh" Password: ****

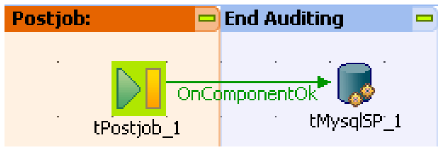
Schema: Built-In | Edit schema

SP Name: "etl_EndRun"

☐ Is function

Parameters: Schema Column | Type

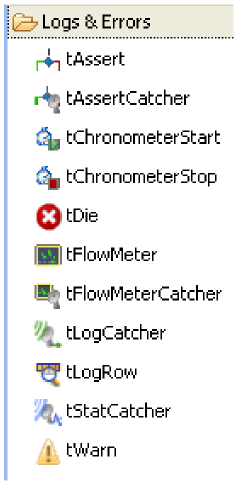
When you're done, your post job should look like this:



## Logging

TOS has two "catcher" components: **tLogCatcher** and **tStatCatcher**. These components listen for log messages and statistics and then create a flow using that information. This allows us the flexibility to write to a log file, to a database anywhere!

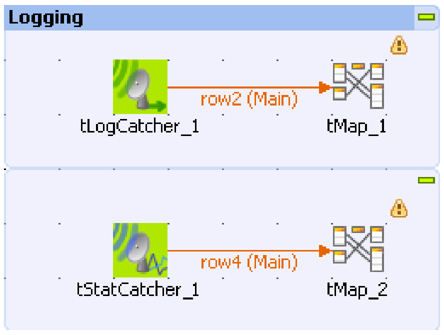
In lesson 5 we created a table called `etlLog`. We'll use the catcher components and some transformation logic from `etlLog` to save logs and stats in our table. Let's get started. Drag one **tLogCatcher** component and one **tStatCatcher** component to your canvas; they are located under the **Logs & Errors** tab:



Take a look at the schema for the components you just dropped on the canvas. Select one of them and switch to the **Component** tab. Then click the ... button next to **Edit Schema**. (This is a little misleading since you can't actually edit the schema for these components.)

You can see that these components are similar, but not exactly the same. We could have sent each component to a different database table, but then we would have to do extra work to see all our log information. With a little work we can transform, then unite the two components to log into our single `etlLog` table.

We'll use two **tMap** components to transform the log and stat output. Drag two **tMap** components from the **Processing** menu of the palette to your canvas, then link the **Main** outputs to the **tMap** components. When done that part of your canvas should look like this:



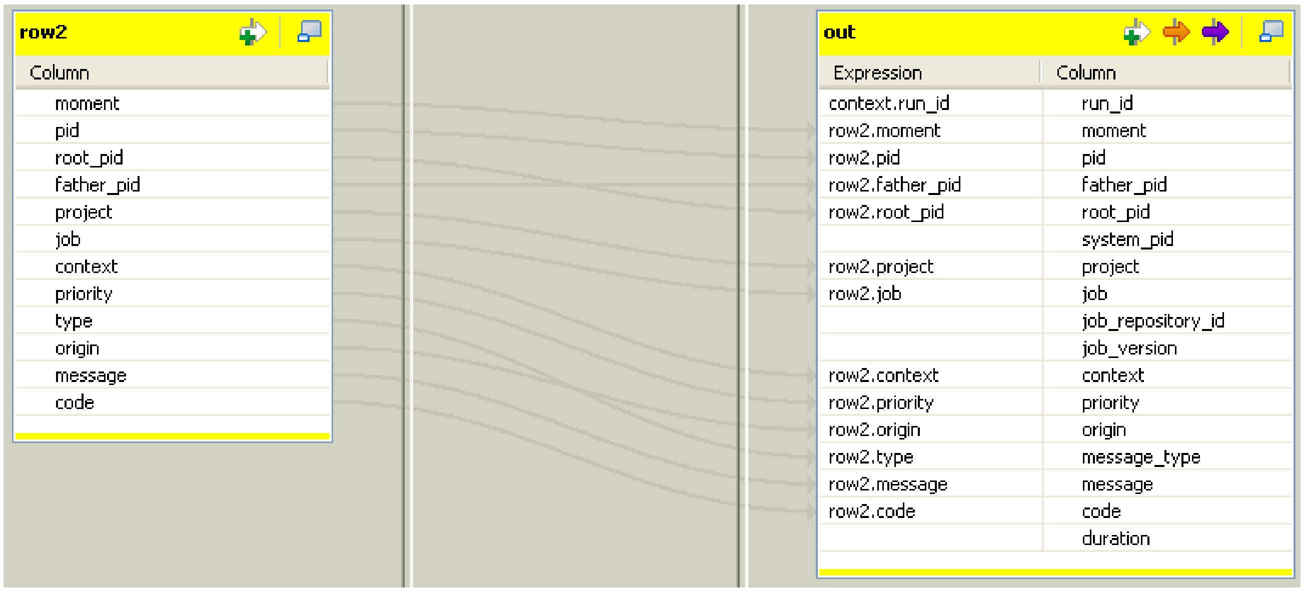
Next, double click the **tMap** component linked to **tLogCatcher**. Your input to **tMap** might not be called **row2** - that's fine. A modal window will open, you can add output information there. Some columns should have blank expressions. Those columns will be used by the stat catcher only. Some columns, like *run_id*, do not exist in the input. You'll have to add them manually. Also, be sure your columns are in the correct order. Add one output, then add or link the columns below:

| Expression | Column | Type |
|---|---|---|
| context.run_id | run_id | integer |
| row2.moment | moment | |
| row2.pid | pid | |
| row2.father_pid | father_pid | |
| row2.root_pid | root_pid | |
| *blank* | system_pid | Long, length 8 |
| row2.project | project | |
| row2.job | job | |
| *blank* | job_repository_id | string, length 255 |
| *blank* | job_version | string, length 255 |
| row2.context | context | |

| | | |
|---|---|---|
| row2.priority | priority | |
| row2.origin | origin | |
| row2.type | **message_type** | |
| row2.message | message | |
| row2.code | code | |
| *blank* | duration | Long, length 8 |

**Note** Make sure you rename **type** column **message_type**.

When you're done, your mappings should look like this:



row2

| Column |
|---|
| moment |
| pid |
| root_pid |
| father_pid |
| project |
| job |
| context |
| priority |
| type |
| origin |
| message |
| code |

out

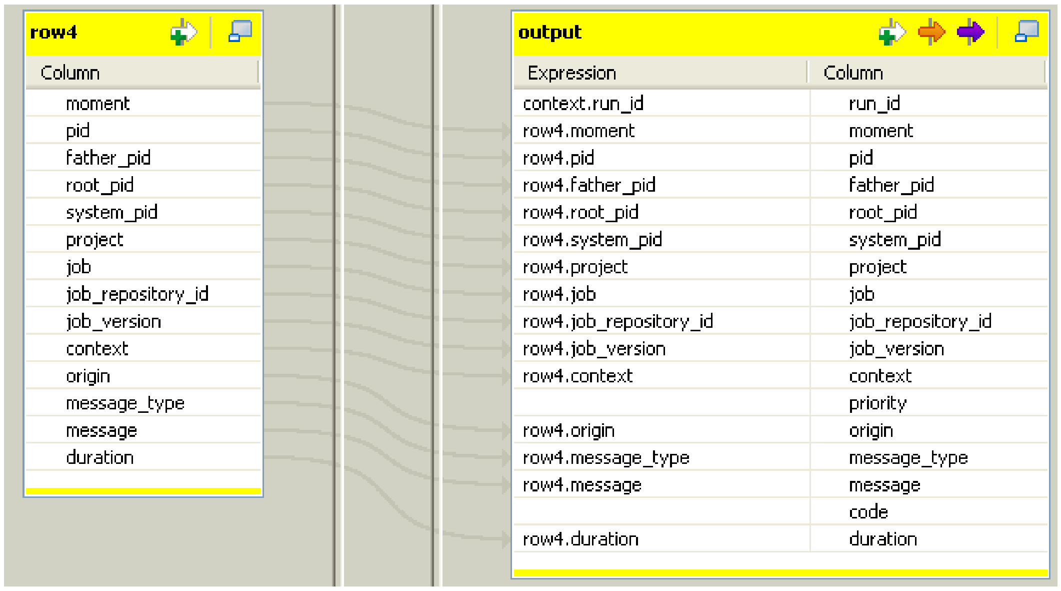
| Expression | Column |
|---|---|
| context.run_id | run_id |
| row2.moment | moment |
| row2.pid | pid |
| row2.father_pid | father_pid |
| row2.root_pid | root_pid |
| | system_pid |
| row2.project | project |
| row2.job | job |
| | job_repository_id |
| | job_version |
| row2.context | context |
| row2.priority | priority |
| row2.origin | origin |
| row2.type | message_type |
| row2.message | message |
| row2.code | code |
| | duration |

**Now is a good time to save your work.**

Next, double click the **tMap** component linked to **tLogStatCatcher**. Your input to **tMap** may not be called **row4** - that's okay. Some columns should have blank expressions. Those columns will be used by the log catcher only. Add one output, then add or link the following columns:

| Expression | Column | Type |
|---|---|---|
| context.run_id | run_id | integer |
| row4.moment | moment | |
| row4.pid | pid | |
| row4.father_pid | father_pid | |
| row4.root_pid | root_pid | |
| row4.system_pid | system_pid | |
| row4.project | project | |
| row4.job | job | |
| row4.job_repository_id | job_repository_id | |
| row4.job_version | job_version | |
| row4.context | context | |
| *blank* | priority | integer, length 3 |
| row4.origin | origin | |
| row4.message_type | message_type | |
| row4.message | message | |
| *blank* | code | integer, length 3 |
| row4.duration | duration | |

When you're done, your mappings should look like this:

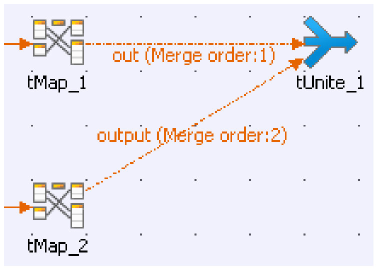


| row4 |
|---|
| Column |
| moment |
| pid |
| father_pid |
| root_pid |
| system_pid |
| project |
| job |
| job_repository_id |
| job_version |
| context |
| origin |
| message_type |
| message |
| duration |

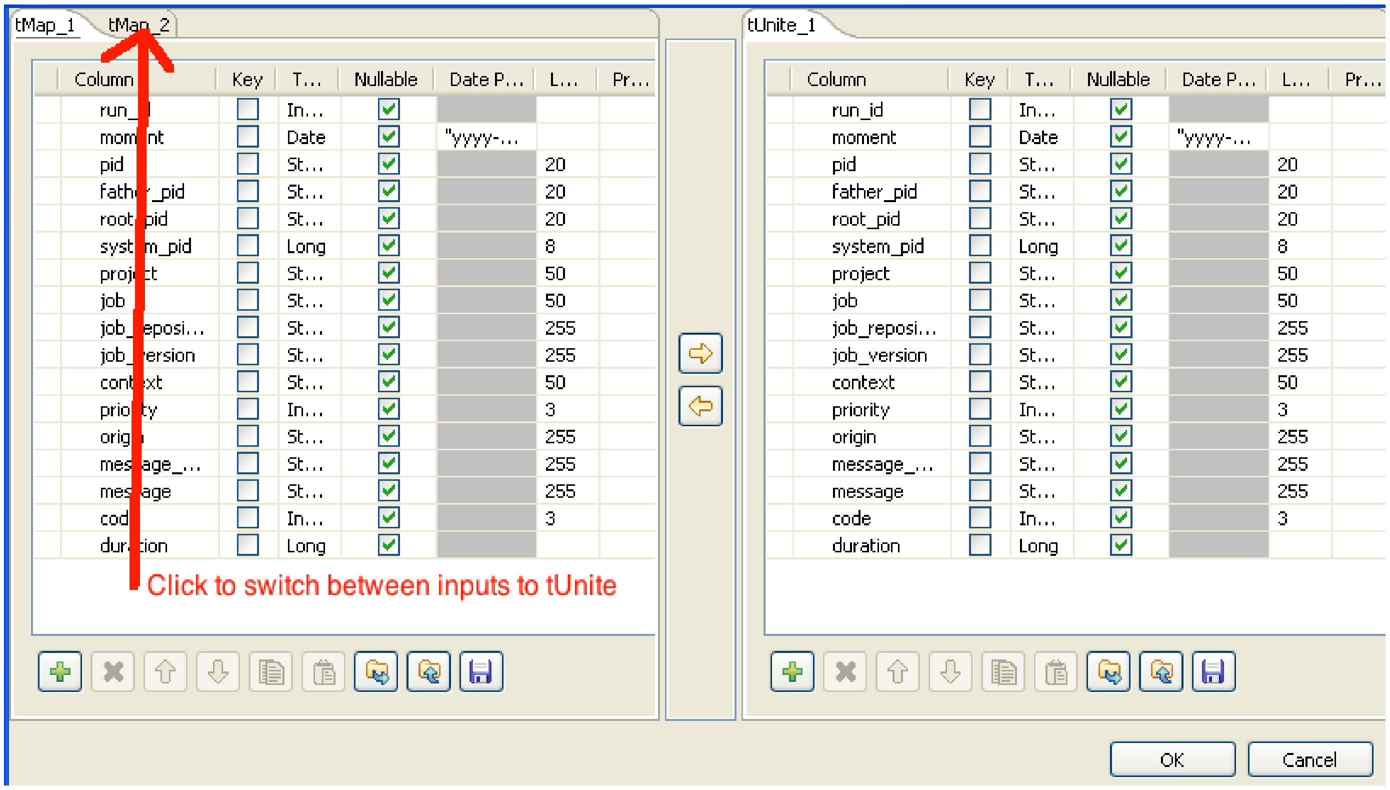
| output | |
|---|---|
| Expression | Column |
| context.run_id | run_id |
| row4.moment | moment |
| row4.pid | pid |
| row4.father_pid | father_pid |
| row4.root_pid | root_pid |
| row4.system_pid | system_pid |
| row4.project | project |
| row4.job | job |
| row4.job_repository_id | job_repository_id |
| row4.job_version | job_version |
| row4.context | context |
| | priority |
| row4.origin | origin |
| row4.message_type | message_type |
| row4.message | message |
| | code |
| row4.duration | duration |

That looks great! Now we need to take our two flows of log data and unite them into a single flow. This is done using the **tUnite** component, which is under the **Orchestration** section in the palette. Drop one onto your canvas.

Next, link the output from each of your map components to **tUnite**. When you're finished, your canvas should look similar to this (don't worry about names, it's okay if they're different):



You might see a warning on your **tUnite** component. This happens when the schema for one of the input flows is different from the others. You can check out the schemas by selecting **tUnite**, clicking on the **Component** tab, then clicking on the ... button next to **Edit Schema**.



| Column | Key | T... | Nullable | Date P... | L... | Pr... |
|---|---|---|---|---|---|---|
| run_id | | In... | ✓ | | | |
| moment | | Date | ✓ | "yyyy-... | | |
| pid | | St... | ✓ | | 20 | |
| father_pid | | St... | ✓ | | 20 | |
| root_pid | | St... | ✓ | | 20 | |
| system_pid | | Long | ✓ | | 8 | |
| project | | St... | ✓ | | 50 | |
| job | | St... | ✓ | | 50 | |
| job_reposi... | | St... | ✓ | | 255 | |
| job_version | | St... | ✓ | | 255 | |
| context | | St... | ✓ | | 50 | |
| priority | | In... | ✓ | | 3 | |
| origin | | St... | ✓ | | 255 | |
| message_... | | St... | ✓ | | 255 | |
| message | | St... | ✓ | | 255 | |
| code | | In... | ✓ | | 3 | |
| duration | | Long | ✓ | | | |

| Column | Key | T... | Nullable | Date P... | L... | Pr... |
|---|---|---|---|---|---|---|
| run_id | | In... | ✓ | | | |
| moment | | Date | ✓ | "yyyy-... | | |
| pid | | St... | ✓ | | 20 | |
| father_pid | | St... | ✓ | | 20 | |
| root_pid | | St... | ✓ | | 20 | |
| system_pid | | Long | ✓ | | 8 | |
| project | | St... | ✓ | | 50 | |
| job | | St... | ✓ | | 50 | |
| job_reposi... | | St... | ✓ | | 255 | |
| job_version | | St... | ✓ | | 255 | |
| context | | St... | ✓ | | 50 | |
| priority | | In... | ✓ | | 3 | |
| origin | | St... | ✓ | | 255 | |
| message_... | | St... | ✓ | | 255 | |
| message | | St... | ✓ | | 255 | |
| code | | In... | ✓ | | 3 | |
| duration | | Long | ✓ | | | |

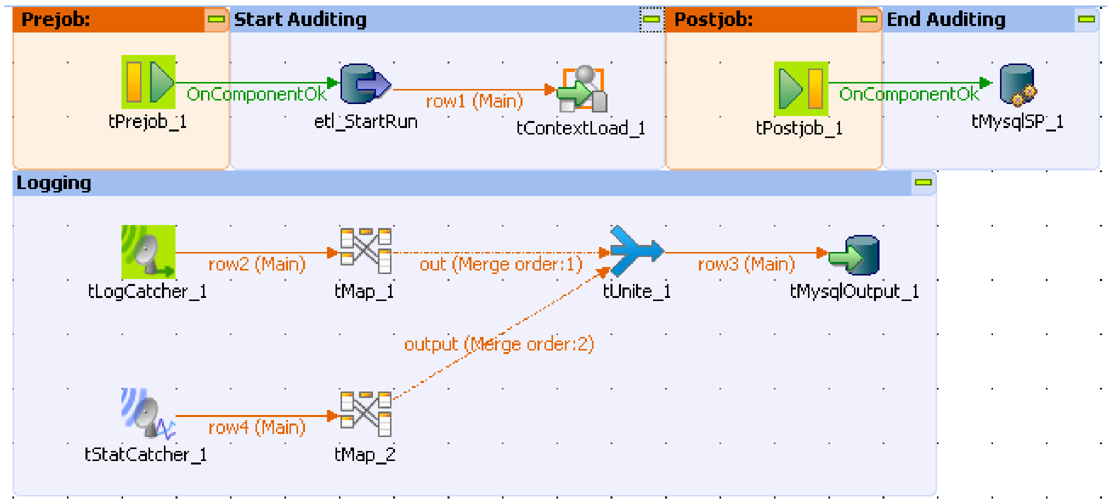
You'll have to do a visual inspection to see which column differs, then go back to the correct **tMap** component to fix the issue.

If TOS asks you whether you want to *Propagate Changes*, choose **Yes**.

Finally, drop a **tMysqlOutput** component from the **Databases** menu of the palette onto your canvas. Link the **Main** output of **tUnite** to **tMysqlOutput**.

Set the database connection. Click on the **tMysqlOutput Component** tab, and go to **Basic settings**. Set the **Property Type** to **Repository**. TOS will ask if you want to take the schema from the input component. You do, so choose **Yes**. Make sure you specify that the **Table** is **"etlLog"**, and that you want to **Insert** data. The **Action on table** should remain **None**.

Your canvas should now look similar to this:



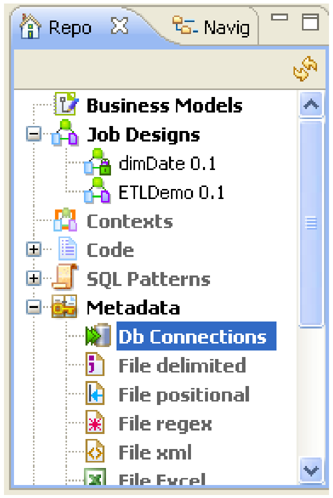
Save your work. With this code done, we are ready to start working on our dimension!

## dimMovie

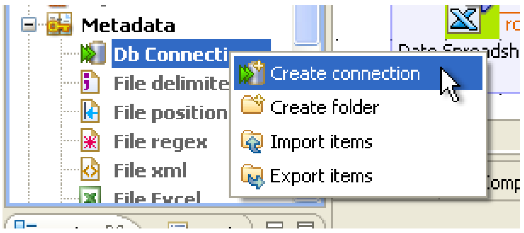
Our basic job structure is in place, so now we're free to concentrate on the actual dimension processing. Let's start with **dimMovie**. You might recall from lesson 3 that **dimMovie** is a Type-1 slowly changing dimension. This means that we do not track any changes on the dimension - instead we update rows in our data warehouse as they change in the source system. (We'll learn more about this a little later.)

This is the first time we need to connect to the `sakila` database, so we need to add a new shared database connection to the repository.

To create a connection, click to expand the **Metadata** section of the repository:



Next, right-click on **Db Connections** and choose **Create Connection**:



Name the connection **sakila** then click **Next >**. The database type is once again **MySQL**. Leave the **Login** and **Password** options blank, but set the server to **sql.oreillyschool.com**. The port is **3306**, and the Database is **sakila**. Click **Check** to make sure you can connect to `sakila`, then click **Finish** to save your new connection.

Drop a **tMysqlInput** column on your canvas. Go to the **Component** tab, and set the connection to **sakila** from the **Repository** using the [...] button on the far right of the **Property Type** row.

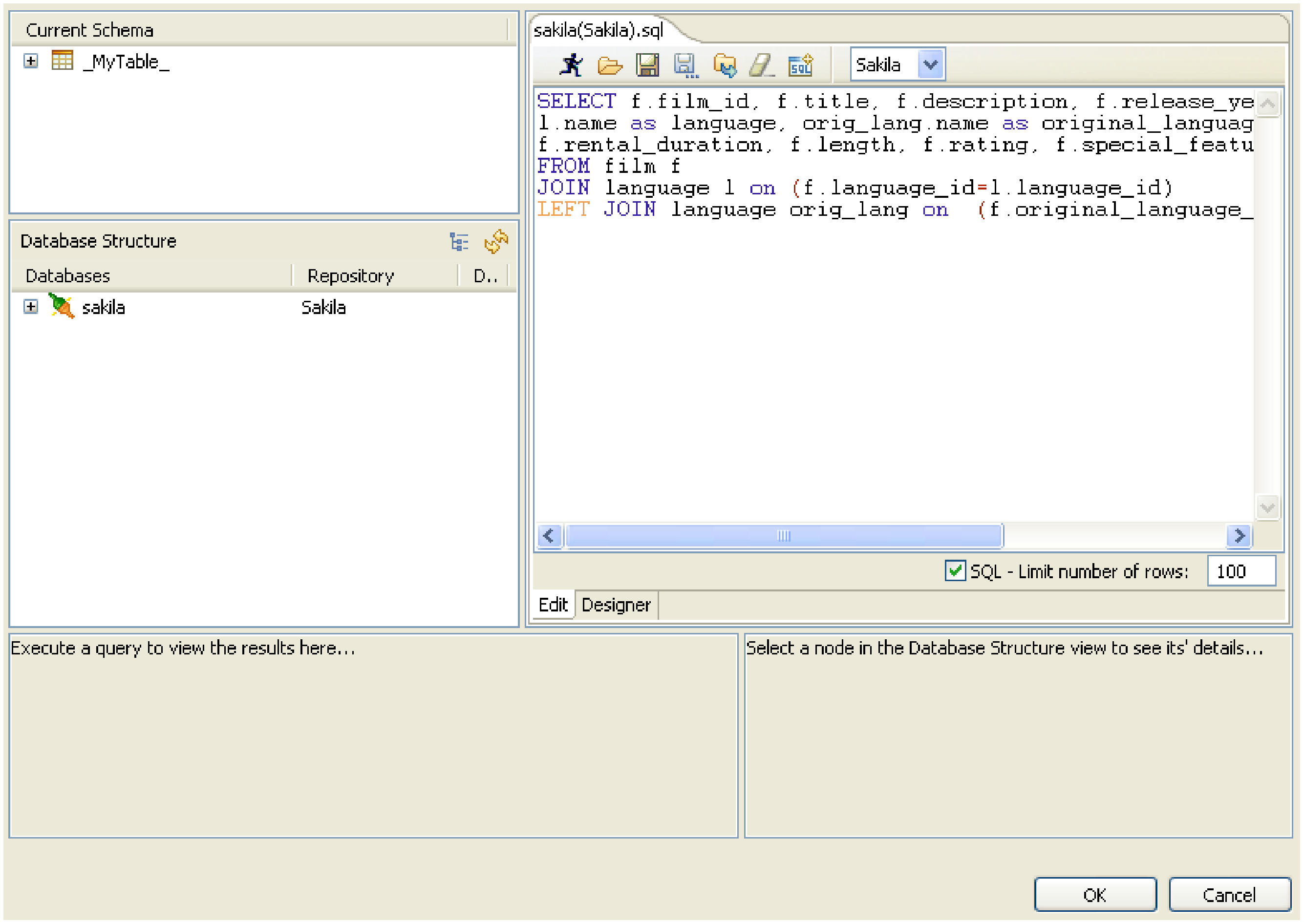
At this point you could choose to save your query in the repository next to the database connection. But since this query will only be used in this specific job, we'll leave the query in this **tMysqlInput** component. Click on the [...] to the right of the query text box.

Back in lesson 5, we wrote a query to get data out of the movie tables. We'll use that same query now. Just this once, copy and paste this query into the SQL builder window:

CODE TO USE:

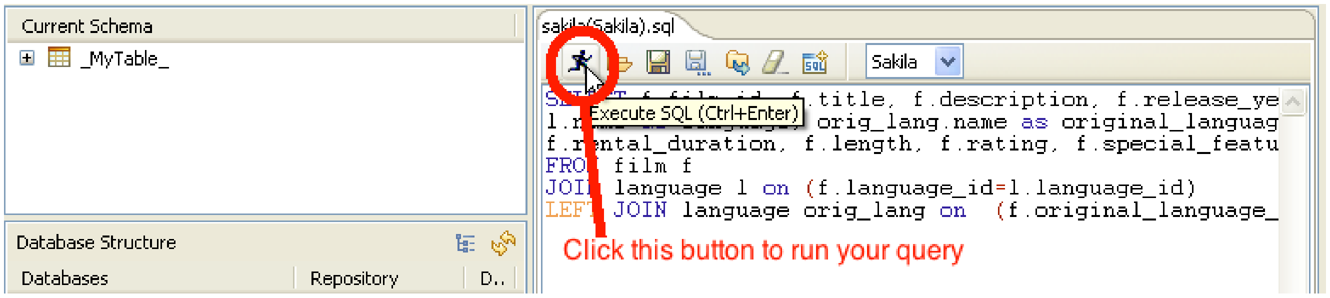
```
SELECT f.film_id, f.title, f.description, f.release_year,
l.name as language, orig_lang.name as original_language,
f.rental_duration, f.length, f.rating, f.special_features
FROM film f
JOIN language l on (f.language_id=l.language_id)
LEFT JOIN language orig_lang on  (f.original_language_id = orig_lang.language_id)
```

Your window should look something like this:


Current Schema
_MyTable_
Database Structure
Databases
Repository
D..
sakila
Sakila
sakila(Sakila).sql
Sakila
SQL - Limit number of rows:
100
Edit
Designer
Execute a query to view the results here...
Select a node in the Database Structure view to see its' details...
OK
Cancel

```
SELECT f.film_id, f.title, f.description, f.release_ye
l.name as language, orig_lang.name as original_languag
f.rental_duration, f.length, f.rating, f.special_featu
FROM film f
JOIN language l on (f.language_id=l.language_id)
LEFT JOIN language orig_lang on  (f.original_language_
```

To test your query, click on the runner, or press `ctrl-enter`. You'll see, at most, 100 rows of results:



When you're satisfied with your query, click **OK**.

Next, we need to specify our schema. TOS does have a *Guess Schema* button which can save time. Unfortunately, it has difficulty reading the schema of our query, so we'll have to enter it by hand. Click on the [...] button to the right of **Edit Schema**. Add the following columns (remember, order does matter!):

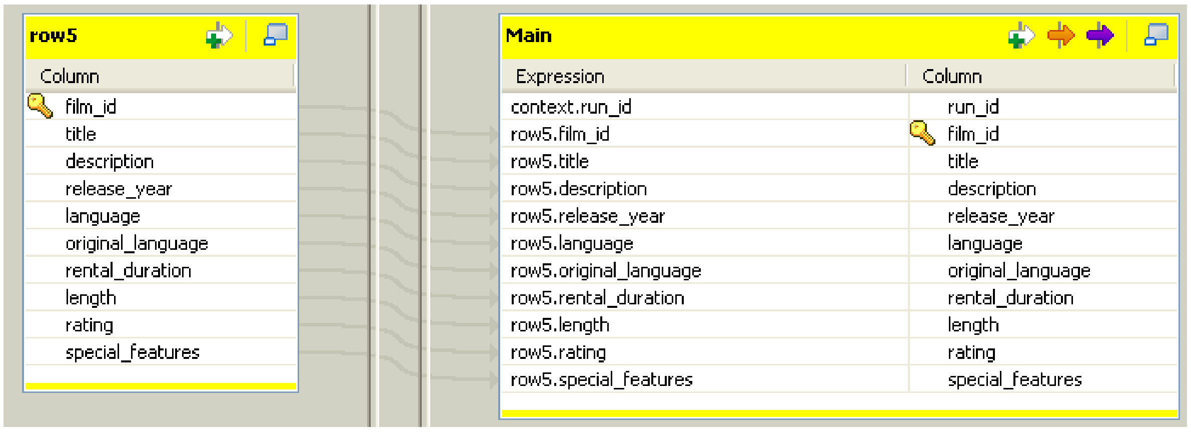
| Column | Type | Length | Notes |
|---|---|---|---|
| film_id | int | | key, not null |
| title | String | 27 | |
| description | String | 200 | |
| release_year | int | | |
| language | String | 20 | |
| original_language | String | 20 | |
| rental_duration | Integer | | |
| length | Integer | | |
| rating | String | 5 | |
| special_features | String | 60 | |

`dimMovie` doesn't really need any transformations, since the source data of the dimension is very good. But we do need to add our audit column to the flow:

**To do**

1. Drag a **tMap** component to your canvas.
2. Map the Main output of **tMysqlInput** to **tMap**.
3. Edit **tMap**, adding an output called **Main**.
4. Add a column to the output: **run_id** of type Integer that cannot be null. The expression for this column is `context.run_id`.
5. Click and drag all columns from the input to the **Main** output.

When you're done, your map should look similar to this:



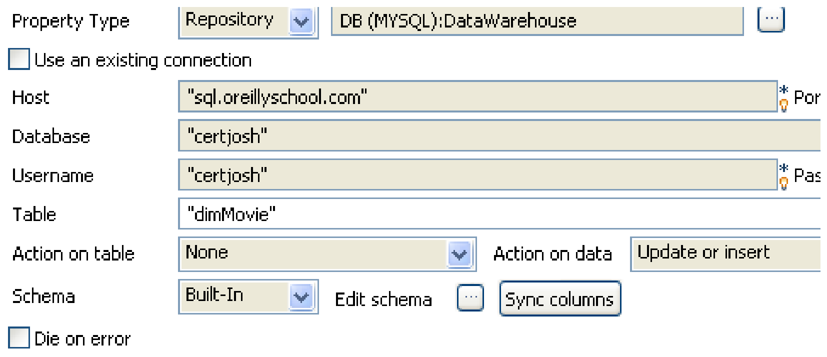
| row5 |
|---|
| Column |
| film_id |
| title |
| description |
| release_year |
| language |
| original_language |
| rental_duration |
| length |
| rating |
| special_features |

| Main | |
|---|---|
| Expression | Column |
| context.run_id | run_id |
| row5.film_id | film_id |
| row5.title | title |
| row5.description | description |
| row5.release_year | release_year |
| row5.language | language |
| row5.original_language | original_language |
| row5.rental_duration | rental_duration |
| row5.length | length |
| row5.rating | rating |
| row5.special_features | special_features |

**Now perform these steps:**

1. Drop a **tMysqlOutput** component on the canvas.
2. Set the connection to the **Repository --> DataWarehouse**.
3. Link the output of **tMap** to your new **tMysqlOutput** component.
4. Set the table to "`dimMovie`".
5. Change the **Action on data** to `Update or Insert`.

When you're done, **tMysqlOutput** should look similar to this:



| | | | |
|---|---|---|---|
| Property Type | Repository | DB (MYSQL):DataWarehouse | |
| ☐ Use an existing connection | | | |
| Host | "sql.oreillyschool.com" | Por | |
| Database | "certjosh" | | |
| Username | "certjosh" | Pas | |
| Table | "dimMovie" | | |
| Action on table | None | Action on data | Update or insert |
| Schema | Built-In | Edit schema | Sync columns |
| ☐ Die on error | | | |

What does the **Action on data** setting of `Update or Insert` mean?

Good question! `Update or Insert` means that for each row of data sent to **tMysqlOutput**, TOS will determine if the row exists in the database table `dimMovie`. If it exists and is different, the row will be updated. If it does not exist, the row is inserted.

Well then, you ask, *how does TOS know if the row exists?*. Another good question. TOS looks at the **Key** that we specified in the schema. In this case, we set the column `film_id` to be a key column, so this column is used to check to see whether the row exists. All columns specified as keys are checked when performing an insert or update.

Another question you might be asking yourself is, *why don't we just delete data from `dimMovie` and reload the whole thing?*

Yet another good question. The answer is: **foreign keys**. Our primary key for `dimMovie` is an auto increment field called `movie_key`. This column will be used to link facts to this dimension. If we wipe out `dimMovie` each time we run our load, we'll always have to reload all facts as well. This might be okay in the short term, but at some point we may not want to reload everything in our data warehouse. Using "insert or update" means that existing data does not get deleted, so a `movie_key` is preserved across runs. This is important even if the underlying database does not enforce foreign key constraints.

Now that we have our job done, it's time to run it! Click on the button at the top of the screen. If everything goes well, you'll only see two lines of output:

OBSERVE:

```
Starting job ProcessDataWarehouse at 13:29 09/06/2009.

Job ProcessDataWarehouse ended at 13:32 09/06/2009. [exit code=0]
```

You can also check the database to see what has been logged for your run. Switch to a terminal, and log into your personal database. Run the following command against your personal database:

CODE TO TYPE:

```
mysql> select * from etlRuns;
```

Your results won't be exactly the same, but they should look similar to the following:

OBSERVE:

```
mysql> select * from etlRuns;
+--------+---------------------+---------------------+
| run_id | start_time          | end_time            |
+--------+---------------------+---------------------+
|      1 | 2009-06-09 11:49:07 | 2009-06-09 11:49:12 |
+--------+---------------------+---------------------+
1 row in set (0.00 sec)
```

Next, check the `etlLog` table. Run the following command against your personal database:

CODE TO TYPE:

```
mysql> select * from etlLog;
```

You'll see some results similar to the following:

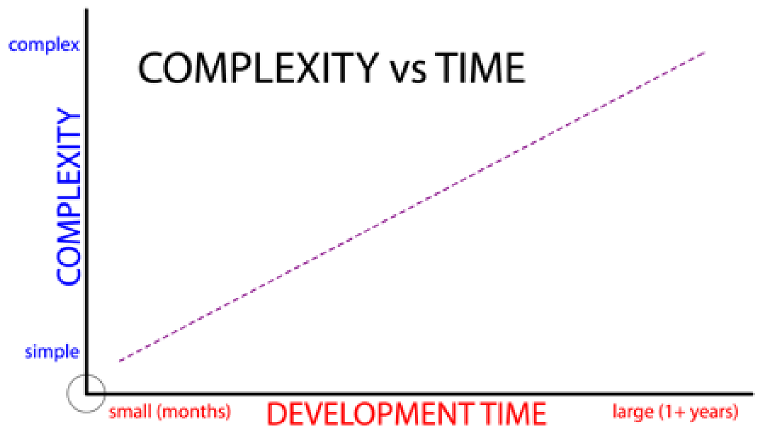
OBSERVE:

```
mysql> select * from etlLog;
+--------+---------------------+--------+------------+----------+------------+---------+------------
| run_id | moment              | pid    | father_pid | root_pid | system_pid | project | job
+--------+---------------------+--------+------------+----------+------------+---------+------------
|      1 | 2009-06-09 11:49:07 | GOrr6m | GOrr6m     | GOrr6m   |       8356 | DBA3    | ProcessDataWa
|      1 | 2009-06-09 11:49:12 | GOrr6m | GOrr6m     | GOrr6m   |       8356 | DBA3    | ProcessDataWa
+--------+---------------------+--------+------------+----------+------------+---------+------------
2 rows in set (0.00 sec)
```

These results show that our run was **successful**, and took **5594** milliseconds (~5 seconds) to run. This is duplicated by the `etlRuns` table, which shows the **start and end times** of the job. We're looking pretty good!

## Performance

When developing a software system, it is often best to start with a simple solution and move to a more complex solution as development goes on:



Our movie dimension is very small - it only has 1000 rows. Since it is so small, it is acceptable to recreate this dimension (along with other related facts) from scratch each day. This solution is simple and works well for small data warehouses.

But what if our movie dimension had 500,000 rows in it? What if our source system was an ancient computer, requiring 10 hours to extract movie data? Running a data load for 10 hours everyday would not be a great option; even if the rest of the warehouse processing only took a minute or two, there would only be 14 hours left for warehouse use. Should the movie dimension grow to be 1,000,000 rows, the warehouse load might take 20 hours to complete!

When dimensions are large, it is necessary to add complexity to the warehouse in order to reduce load times.

The first step toward optimizing performance seems straightforward: **only query the source system for new and changed records**. Our audit tables capture the date and time that the dimension was updated. That time stamp can be used to select records in the source system. But this process is often more difficult than it initially seems.

Many source systems simply don't track enough data to make this query work. The *film* table in the sakila database has a column called `last_update` which should get set when the row is created, and updated when the row changes. But what if it had a column called `date_created` instead? How would we know when a row had changed?

In many situations you will have to make changes to source systems to make data warehouse loads easier to manage. This might involve adding time stamp columns, modifying existing columns, or even creating completely new tables. Keep this in mind as we move forward.

We've covered a lot in this lesson! Stay tuned - in the next lesson we'll continue working with our dimensions and learn how to process *slowly changing dimensions*. See you then!

# SCD Processing
# DBA 3: Data Warehousing Lesson 9

Welcome back! In the last lesson we implemented our first dimension, `dimMovie`, as a Type 1 slowly changing dimension. In this lesson we'll take a look at the remaining dimensions, and implement them as Type 2 slowly changing dimensions.

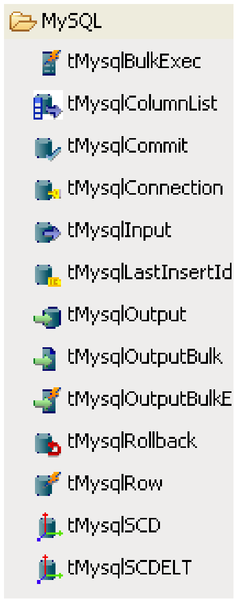
## The Algorithm: Slowly Changing Dimensions

If you think back to the third lesson, we defined several types of slowly changing dimensions. They are:

- **Type 0**-- Do nothing; dimension records are not updated.
- **Type 1**-- No history is kept; dimension records are updated in-place.
- **Type 2**-- A full history is kept; dimension records have start and end dates to determine when they are valid.
- **Type 3**-- A limited history is kept; usually the most recent history of a column or two is kept in the same row as current information.
- **Type 4**-- A full history is kept in a history table; the dimension table has current records.

We implemented a **Type 1** dimension in the last lesson. In this lesson we'll cover **Type 2**.

Slowly changing dimensions (SCD) are really common, so TOS has a specific way to deal with them: the **SCD** family of components. You'll need an open job to see the palette, so open your **ProcessDataWarehouse** job. Expand the palette under Databases, then under MySQL, and you'll see this:



MySQL has two SCD components: **tMysqlSCD** and **tMysqlSCDELT**. In TOS, components that are named **ELT** are processed on the database server itself. This can make the process much faster, because data doesn't have to travel outside of the database server to be processed. The drawback to this process is that all data must be located on a single database server - something that isn't often possible.

So how does the SCD component work for Type 2 dimensions? Like this:

OBSERVE:

```
Check each row of data to determine whether the row exists within the dimension. If it does not, inser

 1. Now that the row exists within the dimension, determine whether the specified Type 2 columns have

  a. If they have not changed, go on to the next row:
   1) If the row has changed:
    -Insert a new row into the dimension, with the start date of today and the end date of 31-DEC-2099
    -Update the prior row, setting the end date to today.
```

**Note** Remember, some warehouses use `NULL` instead of `31-DEC-2099` as the end date.

This algorithm isn't extremely difficult to implement, however it would really stink if you had to implement it for each dimension you wanted to process! Fortunately, it has been implemented, tested, and optimized for us already.

Type 3 and Type 4 SCDs are handled in almost the same way, except that the history is located elsewhere. In Type 3 it is put into the same row; in Type 4 it is put into a different table.

## Implementing the Dimensions

### dimCustomer

Let's dive right in. First, drag a **tMysqlInput** component from the palette to your canvas. Set its database connection to the `sakila` connection from the repository. For the query, we'll use this code (from lesson 5):

OBSERVE:

```
SELECT
c.customer_id, c.first_name, c.last_name, c.email,
a.address, a.address2, a.district,
ci.city,
co.country,
postal_code,
a.phone,c.active, c.create_date
FROM customer c
JOIN address a on (c.address_id = a.address_id)
JOIN city ci on (a.city_id = ci.city_id)
JOIN country co on (ci.country_id = co.country_id)
WHERE customer_id > 10;
```

Run your query to make sure you typed it in correctly. Then click on the ... button next to **Edit Schema**, and enter these columns:

| Column | Type | Nullable | Length |
|---|---|---|---|
| customer_id | int | | |
| first_name | String | | 45 |
| last_name | String | | 45 |
| email | String | Yes | 50 |
| address | String | | 50 |
| address2 | String | Yes | 50 |
| district | String | | 20 |
| city | String | | 50 |
| country | String | | 50 |
| postal_code | String | | 10 |
| phone | String | | 20 |
| active | int | | |
| create_date | Date | | |

**Note** If your dimCustomer table is slightly different, then you'll need to change your schema. For example, if your table might allow NULLs for email, you should change the schema to allow nulls in that column.

You might wonder why we aren't using the **Guess Schema** button to have TOS figure out the schema for us. It seems like that would be fast and relatively easy. But in practice, the **Guess Schema** option only works well with simple schemas and data types. That's because TOS examines the data that the database returns from the query, in order to make decisions on data types and lengths, and ignores the underlying data types set in the database tables.

That might be okay for some queries, but it doesn't work for our `dimCustomer` query. In this query, currently the `address2` column only has `NULL` values. TOS can't determine a data type when there's no data. For other columns, like `postal_code`, TOS sees only values like `90210`, so TOS guesses that the data type is `Integer`. We know that other parts of the world have different postal code formats ("SW1A 0AA" is a valid postal code in the United Kingdom), so our `dimCustomer` table uses varchar as its data type, not integer.

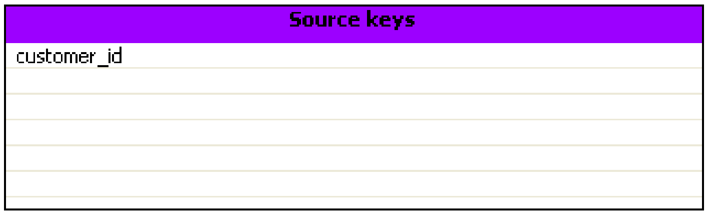
It's fine to use **Guess Schema** as a starting point, but you still need to verify manually that the columns TOS picks are of the correct data type, nullability, and length.

With the input out of the way, we are free to move on to the mapping. Drag a **tMap** component to the canvas, and link the main row of the previous **tMysqlInput** component to **tMap**. Just like before, add a new output, and add a column called **run_id** (type: integer, expression: `context.run_id`) to the output. Link every input column to the output.

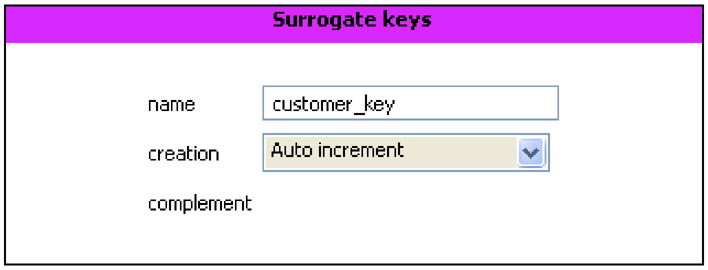
The last step for `dimCustomer` is to add a **tMysqlSCD** component to the canvas. Link the output of **tMap** to the input of **tMysqlSCD**, and allow TOS to take the schema from the input component. If TOS does not ask whether you want to use the input schema, click on **Sync Columns**.

Set the connection on **tMysqlSCD** to the data warehouse connection in the repository, and specify `dimCustomer` for the table. Once that's done, click on the ... button next to **SCD Editor**.

In this new window you'll tell **tMysqlSCD** how to handle every column in the data flow. To set up the component, drag a column from the **Unused** section to a different section. We'll start with **Source Keys**. Our source key is a single column: `customer_id`. Drag that column to the **Source Keys** section. When you're done, your screen will look like this:

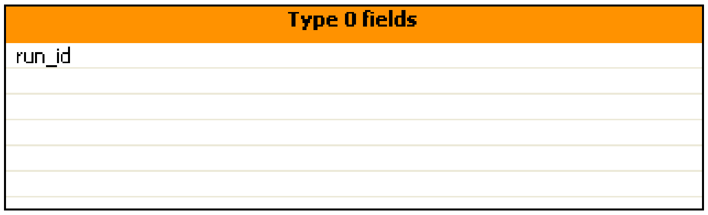


Next, we'll setup our surrogate keys. Our surrogate key does not exist in the data flow. Instead, it will be created by auto increment in MySQL. Name the surrogate key `customer_key`, and name the creation `Auto increment`. When you're finished, that section will look like this:

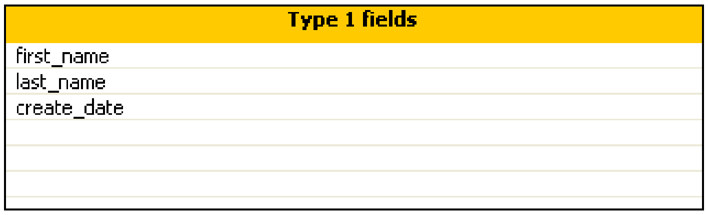


name: customer_key
creation: Auto increment
complement

Now we'll specify our *Type 0* columns. Since our dimension is only included for auditing and logging, `run_id` should never be considered part of the dimension. This value changes at each run, so we never want to track changes on it. Drag `run_id` to the *Type 0 fields* section. That section will now look like this:

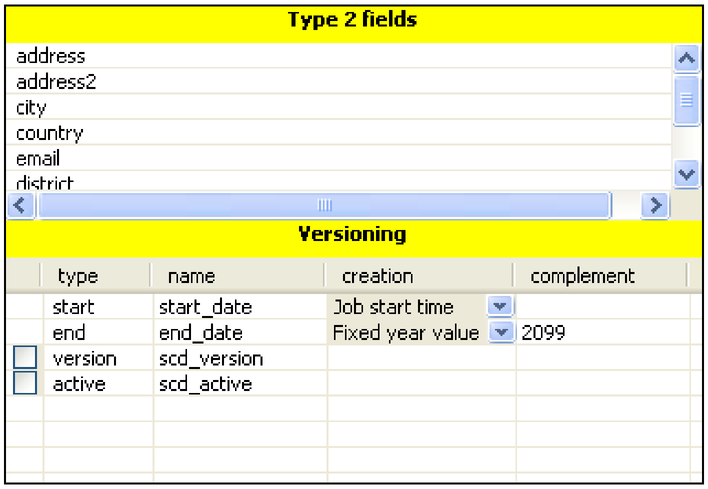


We are not particularly interested in tracking changes to our customers' names. And if a customer's *create_date* changes, it probably means the source system had an error, and the current record is being fixed, so we don't want to track changes on that either. Drag `first_name`, `last_name` and `create_date` to the *Type 1 fields* section. That section will now look like this:



Let's check out *Type 2* changes now. Type 2 changes require additional configuration because there are different ways to track history within the same table (if you'd like to review type 2 changes, refer back to lesson 3).

Drag the remaining columns from **Unused** to the *Type 2 fields* section. Then we need to tell TOS how we are keeping history. In this dimension, we are using a start and end date, but we are not using a version number column or an active flag. Rename the start column **start_date**, and set its creation to **Job start time**. Rename the end column **end_date**, change its creation to **Fixed year value**, and set its complement to **2099**. After you have made these changes, the *Type 2 fields* section will look like this:


**Type 2 fields**

address
address2
city
country
email
district

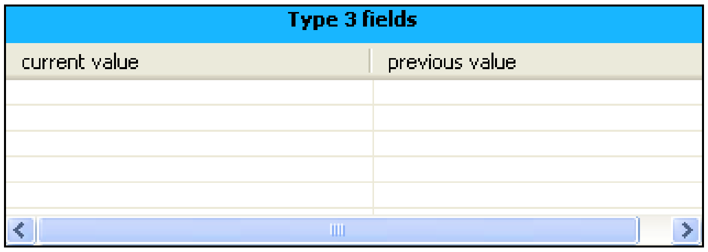
**Versioning**

| | type | name | creation | complement |
|---|---|---|---|---|
| | start | start_date | Job start time | |
| | end | end_date | Fixed year value | 2099 |
| ☐ | version | scd_version | | |
| ☐ | active | scd_active | | |

We are using two features here - *version* and *active*. *version* is an integer counter that keeps track of the number of changes; so a row with version 3 is newer than a row with version 2. *active* is a boolean indicator, which flags the most recent row as *active*.

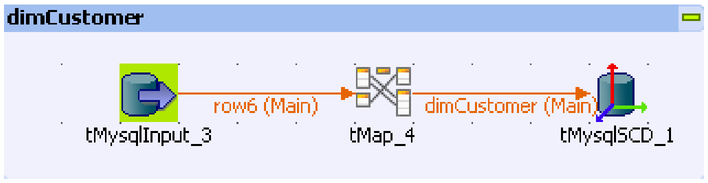
The last section is for *Type 3* changes. Here you specify the *type 3* columns and note the corresponding history columns.

This dimension doesn't have any *Type 3 fields*, so it's left blank:


**Type 3 fields**

| current value | previous value |
|---|---|
| | |

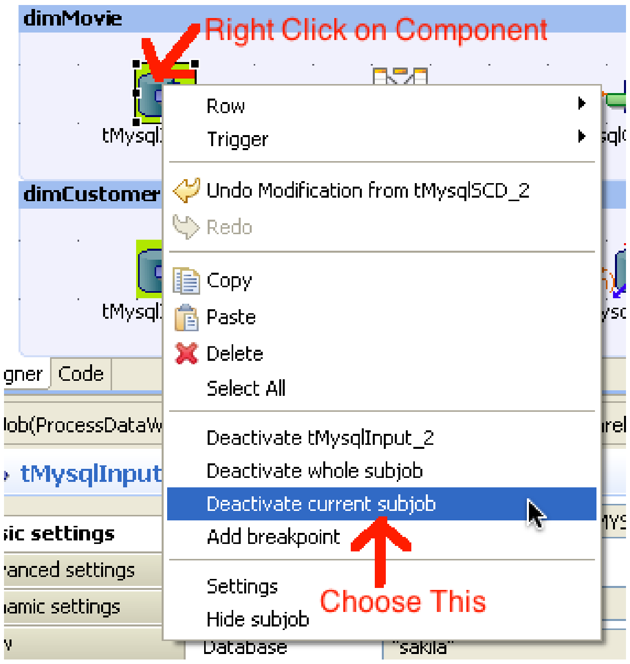
Click "OK" to close the SCD Component editor, then save your changes. When you're done, your **dimCustomer** sub job should look like this:



## Does our SCD work?

At this point, we need to test our sub job to see if `dimCustomer` is receiving data. We don't need to run our entire job though, because we're really only interested in `dimCustomer`. Fortunately TOS lets us disable sub jobs.

To do that, right click on the **tMysqlInput** component for the **dimMovie** job, then choose **Deactivate current sub job**.



The sub job for **dimMovie** is now greyed-out and disabled.

Now run your job by clicking on the . As long as you've typed everything correctly, your job will run and you'll see the following output:

OBSERVE:

```
Starting job ProcessDataWarehouse at 21:08 30/06/2009.
Job ProcessDataWarehouse ended at 21:08 30/06/2009. [exit code=0]
```

The job ran, but did it populate the `dimCustomer` table? Switch to a terminal, and log into your personal database. Run the following command against your personal database:

CODE TO TYPE:

```
mysql> SELECT count(*) FROM dimCustomer;
```

If your job ran successfully, you will see this:

OBSERVE:

```
mysql> SELECT count(*) FROM dimCustomer;
+----------+
| count(*) |
+----------+
|      589 |
+----------+
1 row in set (0.00 sec)
```

To be sure everything worked, take a look at some rows. Run the following command against your personal database:

CODE TO TYPE:

```
mysql> SELECT * FROM dimCustomer
LIMIT 0, 10;
```

OBSERVE:

```
mysql> SELECT * FROM dimCustomer
    -> LIMIT 0, 10;
+--------------+-------------+------------+-----------+-----------------------------------+--
| customer_key | customer_id | first_name | last_name | email                             | a
+--------------+-------------+------------+-----------+-----------------------------------+--
|            1 |         218 | VERA       | MCCOY     | VERA.MCCOY@sakilacustomer.org     | 1
|            2 |         441 | MARIO      | CHEATHAM  | MARIO.CHEATHAM@sakilacustomer.org | 1
|            3 |          69 | JUDY       | GRAY      | JUDY.GRAY@sakilacustomer.org      | 1
|             4 |           176 | JUNE        | CARROLL    | JUNE.CARROLL@sakilacustomer.org      | 7
|             5 |           320 | ANTHONY     | SCHWAB     | ANTHONY.SCHWAB@sakilacustomer.org    | 1
|             6 |           528 | CLAUDE      | HERZOG     | CLAUDE.HERZOG@sakilacustomer.org     | 4
|             7 |           383 | MARTIN      | BALES      | MARTIN.BALES@sakilacustomer.org      | 3
|             8 |           381 | BOBBY       | BOUDREAU   | BOBBY.BOUDREAU@sakilacustomer.org    | 1
|             9 |           359 | WILLIE      | MARKHAM    | WILLIE.MARKHAM@sakilacustomer.org    | 1
|            10 |           560 | JORDAN      | ARCHULETA  | JORDAN.ARCHULETA@sakilacustomer.org  | 1
+---------------+-------------+-------------+------------+--------------------------------------+--
10 rows in set (0.01 sec)
```

If you scroll to the right, you might notice something a bit strange. Check out the `create_date` and `start_date` (copied below):

OBSERVE:

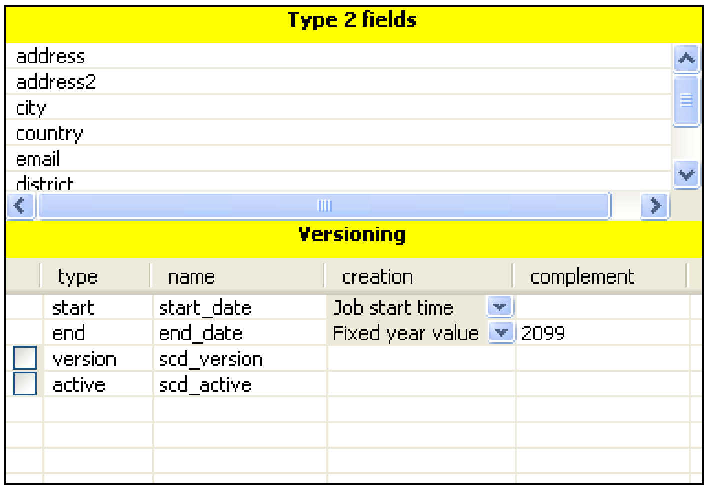
```
+--------+---------------------+------------+------------+--------+
| active | create_date         | start_date | end_date   | run_id |
+--------+---------------------+------------+------------+--------+
|      1 | 2004-03-19 00:00:00 | 2009-06-30 | 2099-01-01 |     81 |
|      1 | 2004-10-07 00:00:00 | 2009-06-30 | 2099-01-01 |     81 |
|      1 | 2004-02-25 00:00:00 | 2009-06-30 | 2099-01-01 |     81 |
|      1 | 2004-08-11 00:00:00 | 2009-06-30 | 2099-01-01 |     81 |
|      1 | 2004-07-20 00:00:00 | 2009-06-30 | 2099-01-01 |     81 |
|      1 | 2004-01-24 00:00:00 | 2009-06-30 | 2099-01-01 |     81 |
|      1 | 2004-05-31 00:00:00 | 2009-06-30 | 2099-01-01 |     81 |
|      1 | 2004-08-29 00:00:00 | 2009-06-30 | 2099-01-01 |     81 |
|      1 | 2004-08-13 00:00:00 | 2009-06-30 | 2099-01-01 |     81 |
|      1 | 2004-01-15 00:00:00 | 2009-06-30 | 2099-01-01 |     81 |
+--------+---------------------+------------+------------+--------+
```

The customer's record was created back on **March 19th, 2004**, but the row in the dimension has a `start_date` of today (**2009-06-30**).

The `start_date` and `end_date` columns on this Type 2 SCD indicate that **"this row is valid and correct between the dates of June 30th 2009 and January 1st, 2099."**

The customer with `customer_id=218` existed on **January 1st, 2007**..., but that's not what the row tells us now.

You may recall that earlier in the lesson, we set the properties for the SCD component. Specifically, we renamed the start column to **start_date**, and set its creation to **Job start time**.



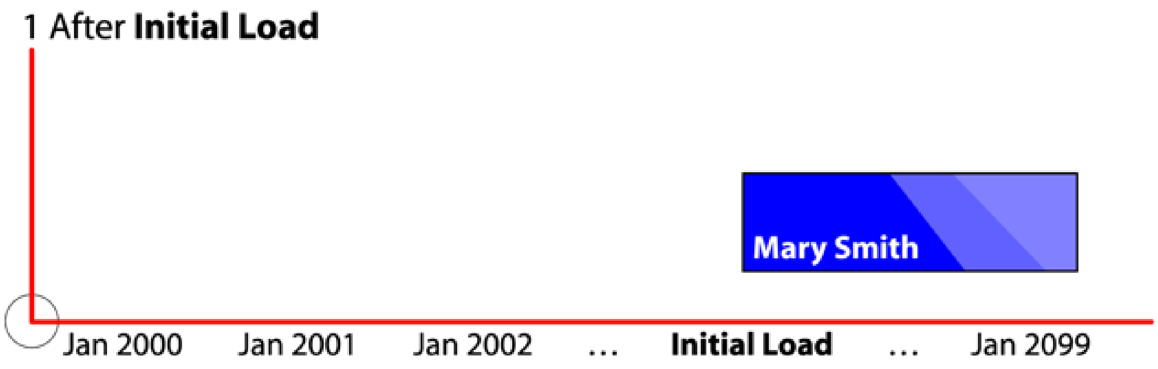
**Type 2 fields**

address
address2
city
country
email
district

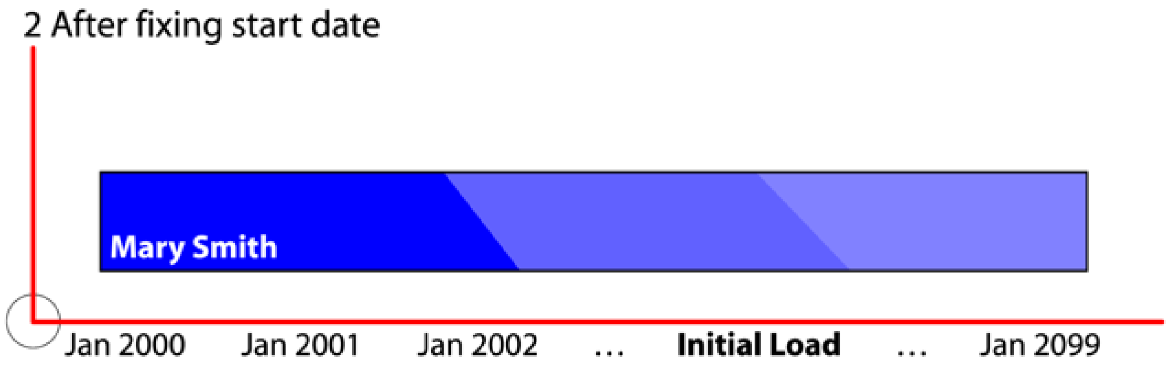
**Versioning**

|  | type | name | creation | complement |
|---|---|---|---|---|
|  | start | start_date | Job start time |  |
|  | end | end_date | Fixed year value | 2099 |
| ☐ | version | scd_version |  |  |
| ☐ | active | scd_active |  |  |

The problem here has to do with historical data. The very first time we setup our dimension, the first valid date for the row must be `start_date`, not today's date. Graphically, the time line after the initial load looks like this:



We need to fix this date, so each row has a valid start date. Graphically, the time line looks like this:



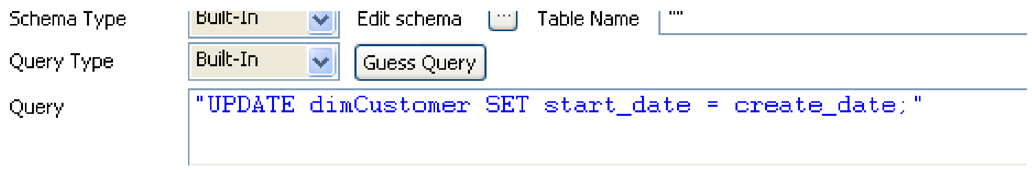
Fortunately, our source system contains the date command we need: `create_date`.

To execute this command, we could just switch to SQL mode and run an update statement. This isn't an ideal solution though, because we might forget to include this step the next time we have to reload the entire dimension.

TOS has a component that will let us execute a single query: **tMysqlRow**. Drag that component to your canvas, and name the new sub job **On Initial Load Only**. Set its connection to the data warehouse connection from the repository, and set its command to this (make sure the quotation marks around the query are correct!):

CODE TO TYPE:

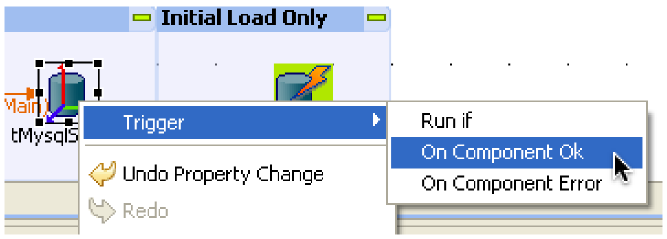
```
UPDATE dimCustomer SET start_date = create_date;
```



Query

```
"UPDATE dimCustomer SET start_date = create_date;"
```

Next, right-click on **tMysqlSCD**, and link the **On Component OK** trigger to **tMysqlRow**:



After our initial load, we'll disable this sub job.

So, what if your data source doesn't have a valid `start_date`? In that case, you'll have to figure out the earliest date where valid data is present in your dimension, perhaps **January 1st, 2000**. Generally, you'll use **Job Start time** as your `start_date`, but after the first load of your dimension, you'll have to update the `start_date` of every row in your table to **January 1st, 2000**. Here's an example of a query to update a dimension without valid start date:

OBSERVE:

```
UPDATE dimCustomer SET start_date='2000-01-01';
```

Picking the earliest valid date for your dimension is similar to picking a high `end_date` of **January 1st, 2099**.

Before we rerun our job, we should completely clear our dimension. To do this, we'll use the SQL keyword `TRUNCATE`. Run this command against your personal database:

CODE TO TYPE:

```
mysql> TRUNCATE TABLE dimCustomer;
```

When the command executes, you'll see `Query OK, 0 rows affected (0.00 sec)`, even though the table is now empty.

Rerun the job. When it completes, switch back to the terminal, and run this command against your personal database:

CODE TO TYPE:

```
mysql> SELECT count(*) FROM dimCustomer;
```

There should still be 589 rows in the table:

OBSERVE:

```
mysql> SELECT count(*) FROM dimCustomer;
+----------+
| count(*) |
+----------+
|      589 |
+----------+
1 row in set (0.00 sec)
```

Check the create and start date next. Run this command against your personal database:

CODE TO TYPE:

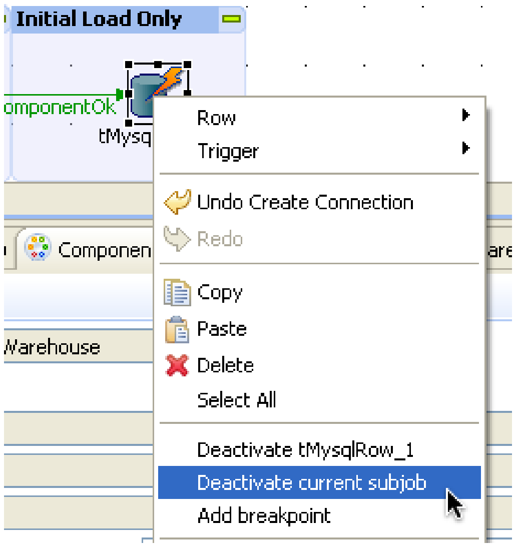
```
mysql> SELECT customer_key, first_name, last_name, address, district, city, country, create_dat
LIMIT 0, 10;
```

The results look much better:

OBSERVE:

```
mysql> SELECT customer_key, first_name, last_name, address, district, city, country, create_dat
    -> LIMIT 0, 10;
+--------------+------------+-----------+-------------------------------+--------------+-------
| customer_key | first_name | last_name | address                       | district     | city
+--------------+------------+-----------+-------------------------------+--------------+-------
|            1 | VERA       | MCCOY     | 1168 Najafabad Parkway        | KABOL        | Kabul
|            2 | MARIO      | CHEATHAM  | 1924 Shimonoseki Drive        | BATNA        | Batna
|            3 | JUDY       | GRAY      | 1031 Daugavpils Parkway       | BCHAR        | Bchar
|            4 | JUNE       | CARROLL   | 757 Rustenburg Avenue         | SKIKDA       | Skikda
|            5 | ANTHONY    | SCHWAB    | 1892 Nabereznyje Telny Lane   | TUTUILA      | Tafuna
|            6 | CLAUDE     | HERZOG    | 486 Ondo Parkway              | BENGUELA     | Bengue
|            7 | MARTIN     | BALES     | 368 Hunuco Boulevard          | NAMIBE       | Namibe
|            8 | BOBBY      | BOUDREAU  | 1368 Maracabo Boulevard       |              | South
|            9 | WILLIE     | MARKHAM   | 1623 Kingstown Drive          | BUENOS AIRES | Almira
|           10 | JORDAN     | ARCHULETA | 1229 Varanasi (Benares) Manor | BUENOS AIRES | Avella
+--------------+------------+-----------+-------------------------------+--------------+-------
10 rows in set (0.00 sec)
```

Now that the start date is looking good, we can disable the sub job that updates `start_date`. Right-click on **tMysqlRow** and choose **Deactivate current sub job**:



We have data in `dimCustomer`, but does our dimension really track history?

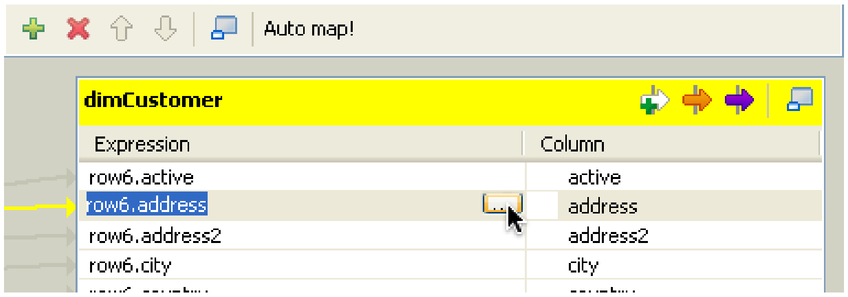
Take a look at the first row of data returned from our previous query:

OBSERVE:

```
|              1  | VERA          | MCCOY       | 1168 Najafabad Parkway          | KABOL          | Kabul
```

The FIRST NAME and LAST NAME are all in UPPERCASE letters, but the address, district, city, and country are in Mixed Case letters. Let's alter our **tMap** component so that the address, district, city, and country are in UPPERCASE letters as well. The next time we run our job, each row should change. That's how we'll be able to tell if our SCD component is working correctly or not.

Double-click on the **tMap** component for the `dimCustomer` subjob. When the map screen opens, select the `address` column on the output, and click on the ... button to open the expression editor window.

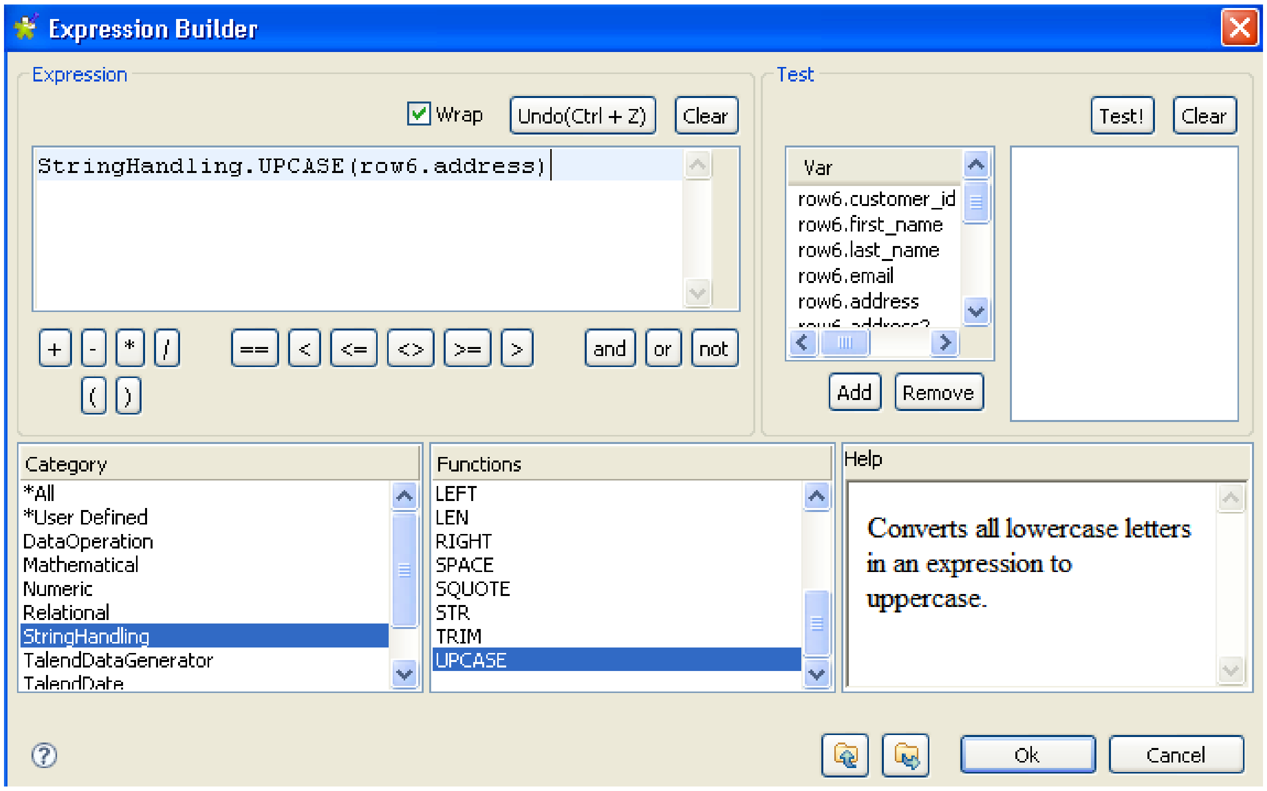


TOS has a built-in function for making a string uppercase. It's located in the *StringHandling* category, and is called `UPCASE`. Set the expression for address so it looks like this:

CODE TO TYPE:

```
StringHandling.UPCASE(row6.address)
```

Your input may not be called **row6**; make sure to use its existing name.



```
StringHandling.UPCASE(row6.address)
```

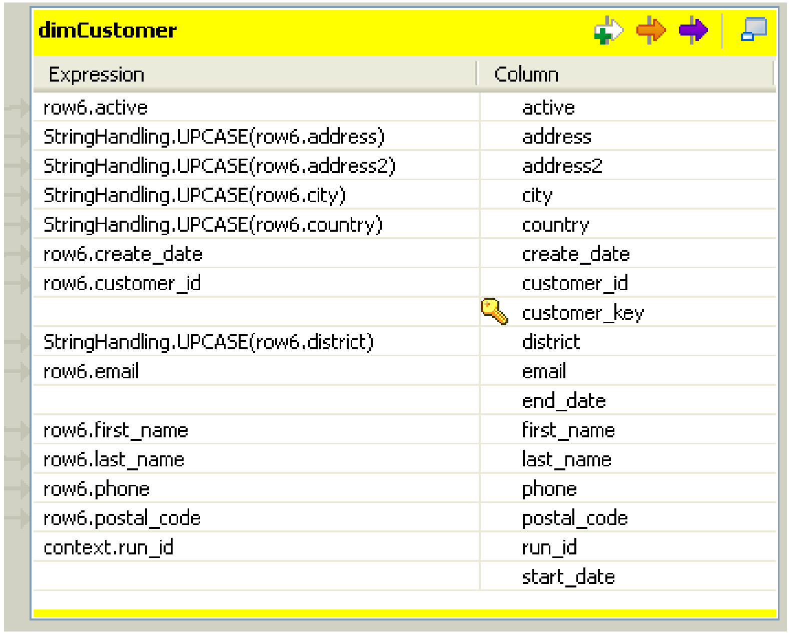
Var:
- row6.customer_id
- row6.first_name
- row6.last_name
- row6.email
- row6.address
- row6.address2

Category: *All, *User Defined, DataOperation, Mathematical, Numeric, Relational, StringHandling, TalendDataGenerator, TalendDate

Functions: LEFT, LEN, RIGHT, SPACE, SQUOTE, STR, TRIM, UPCASE

Help: Converts all lowercase letters in an expression to uppercase.

Click **OK** to close the expression builder. Make similar changes for the other columns: `address2`, `city`, `country`, and `district`. When you are done, **tMap** will look something like this:



| Expression | Column |
|---|---|
| row6.active | active |
| StringHandling.UPCASE(row6.address) | address |
| StringHandling.UPCASE(row6.address2) | address2 |
| StringHandling.UPCASE(row6.city) | city |
| StringHandling.UPCASE(row6.country) | country |
| row6.create_date | create_date |
| row6.customer_id | customer_id |
|  | customer_key |
| StringHandling.UPCASE(row6.district) | district |
| row6.email | email |
|  | end_date |
| row6.first_name | first_name |
| row6.last_name | last_name |
| row6.phone | phone |
| row6.postal_code | postal_code |
| context.run_id | run_id |
|  | start_date |

Run your job again by clicking on the ▶. As long as you've typed everything correctly, your job will run and you'll see the following output:

OBSERVE:

```
Starting job ProcessDataWarehouse at 21:58 30/06/2009.
Job ProcessDataWarehouse ended at 22:02 30/06/2009. [exit code=0]
```

Switch to SQL mode. We'll inspect a single record (`customer_id = 218`) from our last SQL query to see whether it changes. Run this command against your personal database:

CODE TO TYPE:

```
mysql> SELECT * FROM dimCustomer
WHERE customer_id=218
```

```
ORDER BY customer_key;
```

If you typed everything correctly, you'll see this:

OBSERVE:

```
mysql> SELECT * FROM dimCustomer
    -> WHERE customer_id=218
    -> ORDER BY customer_key;
+--------------+-------------+------------+-----------+-------------------------------+--------
| customer_key | customer_id | first_name | last_name | email                         | address
+--------------+-------------+------------+-----------+-------------------------------+--------
|            1 |         218 | VERA       | MCCOY     | VERA.MCCOY@sakilacustomer.org | 1168 Na
|          590 |         218 | VERA       | MCCOY     | VERA.MCCOY@sakilacustomer.org | 1168 NA
+--------------+-------------+------------+-----------+-------------------------------+--------
2 rows in set (0.01 sec)
```

Sure enough, the changes were recorded!

Since our test is over, we'll reload our dimension to get it back to a good starting point. Switch to SQL mode and run the following query:

CODE TO TYPE:

```
TRUNCATE TABLE dimCustomer;
```

When the command executes, you'll see `Query OK, 0 rows affected (0.00 sec)`. In TOS, enable the **Initial Load Only** subjob, then run the job. Once it completes, disable the **Initial Load Only** subjob, and right-click on **tMysqlInput** to disable the **dimCustomer** subjob.

We're in great shape!

## dimStore

The subjob for `dimStore` is nearly identical to `dimCustomer`. Just like before, drag a **tMysqlInput** component to your canvas. Configure it to use the **sakila** database. For the query, we'll use this code (from lesson 5):

OBSERVE:

```
SELECT s.store_id, a.address, a.address2, a.district,
c.city, co.country, a.postal_code, s.region,
st.first_name as manager_first_name,
st.last_name as manager_last_name
FROM
store s
JOIN staff st on (s.manager_staff_id = st.staff_id)
JOIN address a on (s.address_id = a.address_id)
JOIN city c on (a.city_id = c.city_id)
JOIN country co on (c.country_id = co.country_id)
```

Once again, be sure to run your query to make sure you typed it in correctly. Once that's done, click on the ... button next to **Edit Schema**, then enter these columns:

| Column | Type | Nullable | Length |
|---|---|---|---|
| store_id | int | | |
| address | String | | 50 |
| address2 | String | Yes | 50 |
| district | String | | 20 |
| city | String | | 50 |
| country | String | | 50 |
| postal_code | String | | 10 |
| region | String | | 20 |
| manager_first_name | String | | 45 |
| manager_last_name | String | | 45 |

Let's move on to the mapping. Drag a **tMap** component to the canvas, and link the main row of the previous **tMysqlInput** component to **tMap**. Add a new output and add a column called **run_id** (type: integer, expression: `context.run_id`) to the output. Link every input column to the output.

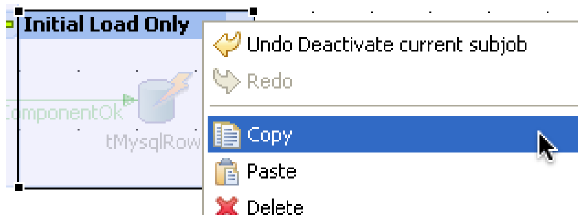
Add a **tMysqlSCD** component to the canvas. Link the output of **tMap** to the input of **tMysqlSCD**, and allow TOS to take the schema from the input component.

Set the connection on **tMysqlSCD** to the data warehouse connection in the repository, and specify `dimStore` for the table. Once that's done, click on the [...] button next to **SCD Editor**.

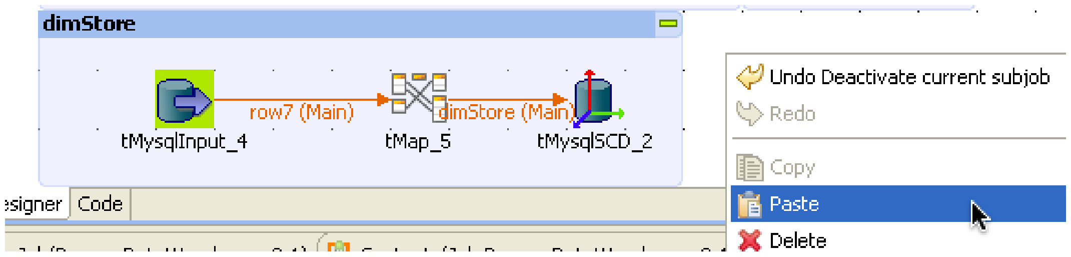
Set up these SCD columns:

| **Section** | **Columns** |
|---|---|
| Source keys | `store_id` |
| Surrogate keys | `store_key` - creation Auto Increment |
| Type 0 fields | `run_id` |
| Type 1 fields | |
| Type 2 fields | Use all remaining fields. Rename the start column **start_date**, and set its creation to **Job start time**. Rename the end column **end_date**, change its creation to **Fixed year value**, and set its compliment to **2099**. |
| Type 3 fields | |

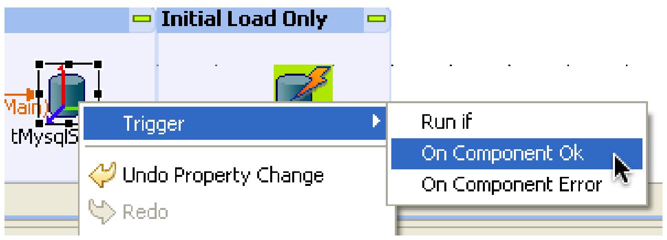
Just like we did in `dimCustomer`, we'll have to run a special update statement the first time we load our dimension here. Let's make life a little easier and reuse our work; copy the subjob we created for `dimCustomer`. Right-click on the title **Initial Load Only**, and choose **Copy**:



Once copied, right-click on your canvas and choose **Paste**. The whole subjob should be pasted on your canvas, but it is probably not next to `dimStore`. Move it so it is next to `dimStore`:



Enable the subjob you just pasted, then link the **On Component OK** trigger from **tMysqlSCD** to it:



Our source data does not give us a valid start date, so we'll have to come up with one. Let's suppose our business users informed us that we could use **January 1st, 2000** as a valid start date. Click on **tMysqlRow**, and change the query so it looks like this:

CODE TO TYPE:

```
UPDATE dimStore SET start_date = '2000-01-01';
```

Save and run your job. If everything went alright, you'll see this familiar output:

OBSERVE:

```
Starting job ProcessDataWarehouse at 13:42 16/12/2008.
Job ProcessDataWarehouse ended at 13:42 16/12/2008. [exit code=0]
```

Let's take a look at the `dimStore` table. Switch to your terminal and run the following command against your personal database:

CODE TO TYPE:

```
mysql> SELECT * from dimStore;
```

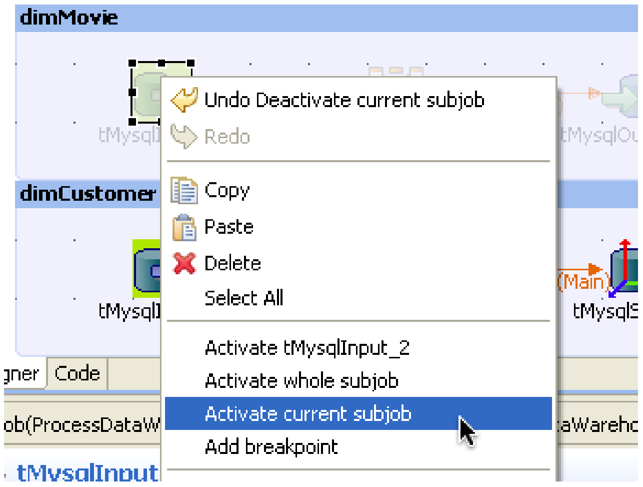
If your load ran properly, you'll see the following data:

OBSERVE:

```
mysql> SELECT * from dimStore;
+-----------+----------+--------------------+----------+----------+------------+-----------+---
| store_key | store_id | address            | address2 | district | city       | country   | po
+-----------+----------+--------------------+----------+----------+------------+-----------+---
|         1 |        1 | 47 MySakila Drive  | NULL     | Alberta  | Lethbridge | Canada    |
|         2 |        2 | 28 MySQL Boulevard | NULL     | QLD      | Woodridge  | Australia |
+-----------+----------+--------------------+----------+----------+------------+-----------+---
2 rows in set (0.00 sec)
```

This looks great! Since our inital load is complete, right-click on **tMysqlRow** in **Initial Load Only** and deactivate the current subjob.

Since our dimensions are working well now, we can reactiavte all of the subjobs we disabled earlier. Right-click on the **tMysqlInput** of the **dimMovie** job, then choose **Activate current subjob:**



Do the same for **dimCustomer**, but be sure to leave the **Inital Load Only** job disabled.

Wow. We covered a lot in this lesson! In the next lesson we'll combine our dimensions with *facts* to form our complete data warehouse. See you there!

# Processing Facts, Part I
# DBA 3: Data Warehousing Lesson 10

Good to have you back. In the last lesson we implemented our dimensions. Now we'll move on to *facts*. Let's get started!

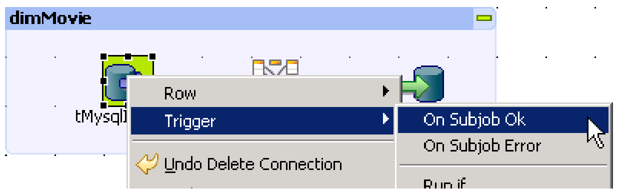
## Orchestration

Most machines have at least two processor cores to optimize performance. TOS is built to take advantage of that, so unless we specify otherwise, TOS may execute our sub jobs in parallel. That means our dimensions may be loaded at the same time. For now, we want to keep our warehouse processes simple and we don't want all of our sub jobs execute at the same time. And we definitely want our dimensions loaded before we even consider loading our **facts**.

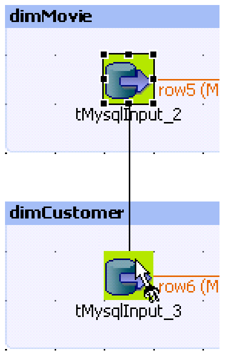
**Note** Your components' names will likely differ from the examples here. That's fine.

The first component in each sub job is the "master" component - it can be used to trigger execution of another sub job after the running sub job is complete. We will use it to link our dimension sub jobs together, and eventually to link to our **fact** sub jobs.

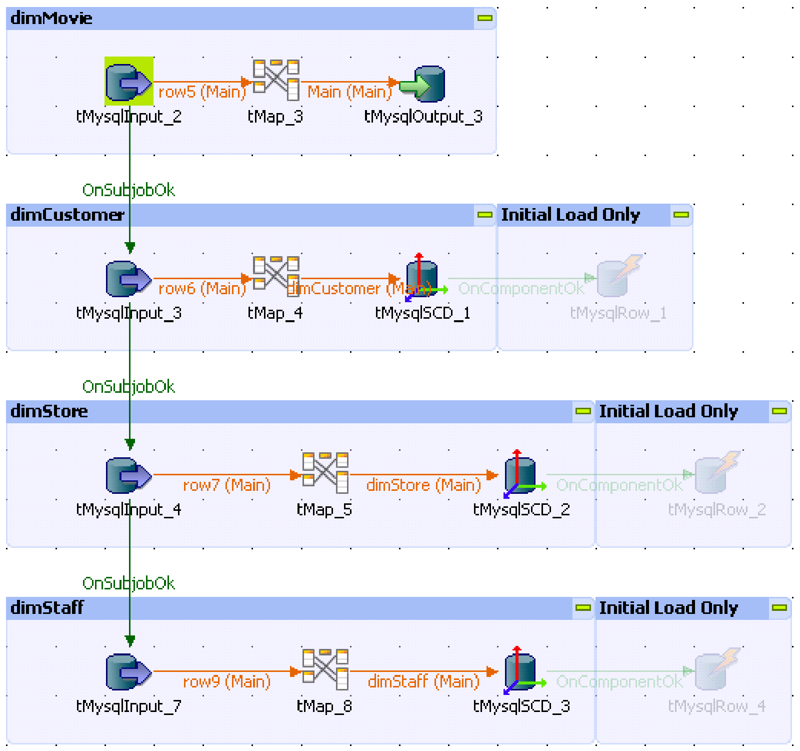
To start, right-click on the **tMysqlInput** component for **dimMovie**. Select **Trigger** and then select **On Subjob OK**:



You'll notice the cursor change. Drag the connection to the **tMysqlInput** component for **dimCustomer**, and drop the connection:



Do this same process to link **dimCustomer** to **dimStore**, and **dimStore** to **dimStaff**. Your job will look like this:



Good work so far.

## factCustomerCount

The algorithm for populating a fact is fairly short. In English, it might read like this: "**For each row, look up the keys for each dimension, respecting start and end dates for the row and the dimension. When the keys have been found, insert the row into the fact table.**"

**Note** For this lesson we will assume that we can successfully lookup each dimension `key`. In a future lesson we will study what might happen if a lookup fails.

This algorithm is fairly straightforward, but selecting the **location** and **method** used to do the lookup can have some pretty drastic performance implications. The lookup is just like a database join - you specify how data is related in order to produce a combined result.

We have two options for **location**:

1. the computer you are using to develop and run TOS jobs
2. the MySQL server

If you pick option 1, essentially you will move data from your source and your data warehouse to your computer, then send it back to your data warehouse. This process is resource intensive and as such, isn't usually the preferred option.

If you pick option 2, you need to move data from your source to your data warehouse and place it in some temporary location (called a *staging* table), then let the database server process the data. This takes some time, but is still usually much faster than option 1. And to make things just a bit more difficult, most ETL tools are not capable of processing data this way, leaving you (the developer) to write a whole mess of SQL to make this happen. Fortunately, TOS has special **ELT** components that can be used to make the process a bit easier.

We also have two options for **method**:

1. Write SQL (perhaps using stored procedures) to lookup and store foreign keys.
2. Use TOS to retrieve and join foreign keys.

To process our customer count **fact**, we will perform these steps:

1. Create a staging table called `stageCustomerCount`, with no indexes (or have TOS create the table for us).
2. Move data from the source system into the staging table.
3. Create indexes on the staging table.
4. Use the ELT family of components in TOS to populate `factCustomerCount`.

You are probably asking yourself, *"Why do I create a table without any indexes, just to add them later?"*

Great question! If you recall back in DBA 2, indexes are a great way to improve database query performance. Unfortunately, using indexes means that each `INSERT` statement takes longer to execute, because the database must do extra work to populate indexes. In other words, it usually takes more time for the database to insert to a table with indexes than it does to insert to a "blank" table first and add indexes to the table later.

In lesson 4 we implemented the table for `factCustomerCount`. Let's review its structure by reviewing the `CREATE TABLE` statement we used back then:

OBSERVE:

```
CREATE TABLE factCustomerCount
(
 customerCount_key INT NOT NULL AUTO_INCREMENT,
 date_key          INT NOT NULL REFERENCES dimDate,
 customer_key      INT NOT NULL REFERENCES dimCustomer,
 store_key         INT NOT NULL REFERENCES dimStore,
 customer_count    INT NOT NULL DEFAULT 1,
 PRIMARY KEY (customerCount_key)
);
```

A single row in this table represents a specific customer who created an account, on a specific day, at a specific store.

This is a *factless fact*, so we don't have any numeric values in our table. To make queries easier for others, we included `customer_count`, which has a default value of 1.

Let's get started by populating our staging table, `stageFactCustomerCount`. Drag three components to your canvas: **tMysqlInput**, **tMap** and **tMysqlOutput**.

Edit the properties for **tMysqlInput**, setting its connection to the **sakila** database. Then edit its query. Use this query to retrieve customer data:

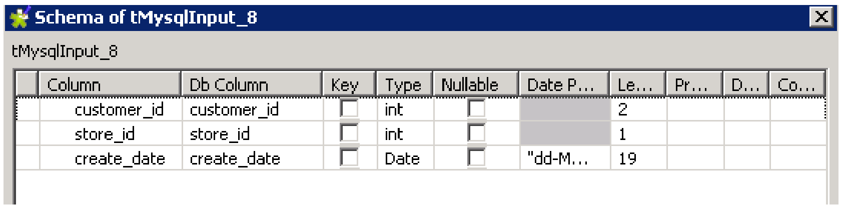
CODE TO TYPE:

```
select customer_id, store_id, create_date FROM customer;
```

Run the query to make sure you typed everything correctly.

Now let's specify the schema for **tMysqlInput**. For this query, we can use the **Guess Schema** button to do most of the work. Go ahead and click on it.

TOS identifies most of the columns without difficulty, except for one: **create_date**. **create_date** isn't a *string* value, it's a *Date*. Fix the schema, and it will look like this:

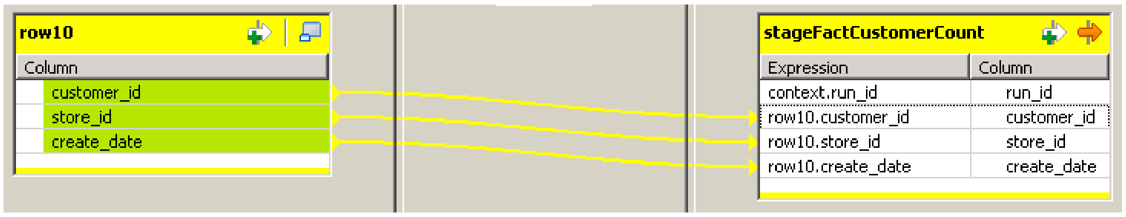

Schema of tMysqlInput_8

tMysqlInput_8

| Column | Db Column | Key | Type | Nullable | Date P... | Le... | Pr... | D... | Co... |
|---|---|---|---|---|---|---|---|---|---|
| customer_id | customer_id | ☐ | int | ☐ | | 2 | | | |
| store_id | store_id | ☐ | int | ☐ | | 1 | | | |
| create_date | create_date | ☐ | Date | ☐ | "dd-M... | 19 | | | |

With that specified, link **tMysqlInput** to **tMap**, then open **tMap** to edit its connections.

We will use this **tMap** just like we used the map components for dimension processing. We'll add our auditing column called `run_id`. Next add an output and then the `run_id` column. Finally, drag all of the input columns to the output. When you are done, your **tMap** should look like this:


row10

| Column |
|---|
| customer_id |
| store_id |
| create_date |

stageFactCustomerCount

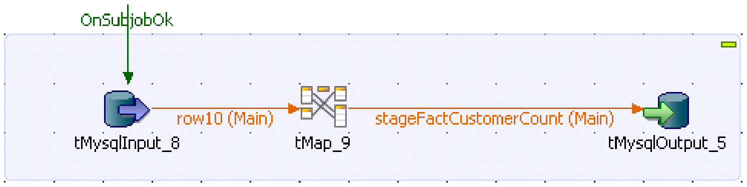
| Expression | Column |
|---|---|
| context.run_id | run_id |
| row10.customer_id | customer_id |
| row10.store_id | store_id |
| row10.create_date | create_date |

Next, connect **tMap** to **tMysqlOutput**. Set the connection properties on **tMysqlOutput** from the repository to the data warehouse connection, and set the table to **"stageFactCustomerCount"** (in quotation marks).

We could manually create the **stageFactCustomerCount** table, but then we would have to delete its data and drop its indexes before we populate it with data. Instead of doing that, we can have the **tMysqlOutput** component **Drop table if exists and create**, which accomplishes the same thing. To do this, set the *action on table* to **Drop table if exists and create**.

Now let's specify when we want to execute our sub job. Right-click on the **tMysqlInput** component of **dimStaff**, and connect the **On Subjob OK** trigger to the **tMysqlInput** of the **stageFactCustomerCount** sub job.

When you are done, your job will look something like this:



Now we can begin adding indexes to `stageFactCustomerCount` and clearing data from `factCustomerCount`.

The tMysqlRow component in TOS can only execute one statement, but we need to execute five `ALTER TABLE` statements and one `TRUNCATE TABLE` statement. We could include several tMysqlRow components, or write a single stored procedure that executes everything we need, using just one tMysqlRow component. We'll use a stored procedure to simplify things.

Switch to the terminal, and run the following command against your personal database:

CODE TO TYPE:

```
DELIMITER //
CREATE PROCEDURE etl_preFactCustomerCount ()
BEGIN
ALTER TABLE stageFactCustomerCount add index(create_date);
ALTER TABLE stageFactCustomerCount add index(customer_id);
ALTER TABLE stageFactCustomerCount add index(store_id);
TRUNCATE TABLE factCustomerCount;
END
//
```

Add a **tMysqlRow** component to your canvas. Set its database connection to the data warehouse, and set its query to the following:

CODE TO TYPE:

```
call etl_preFactCustomerCount();
```

This procedure cannot execute before we populate `stageFactCustomerCount` with data. To do that, right-click on the **tMysqlInput** component of the **stageFactCustomerCount** sub job and link its **On Subjob OK** trigger to **tMysqlRow**.

We are nearly ready to populate `factCustomerCount`, but we have one more small step to complete first. We are planning on using the **ELT** components to load our fact table, but those components need up-to-date schemas for all tables.

Back in lesson seven, we created our data warehouse database connection, and let TOS read the schema for several of our tables. We have made several changes since then, so we need to update our schema.

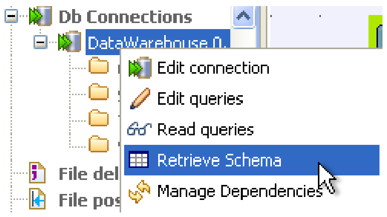
It would be nice to include `stageFactCustomerCount`, but that table hasn't been created yet. The easiest way to create the table is to execute our job. Do so by clicking on the ▶ at the top of the window.

As long as you do not have any errors in your job, you'll see output like this:

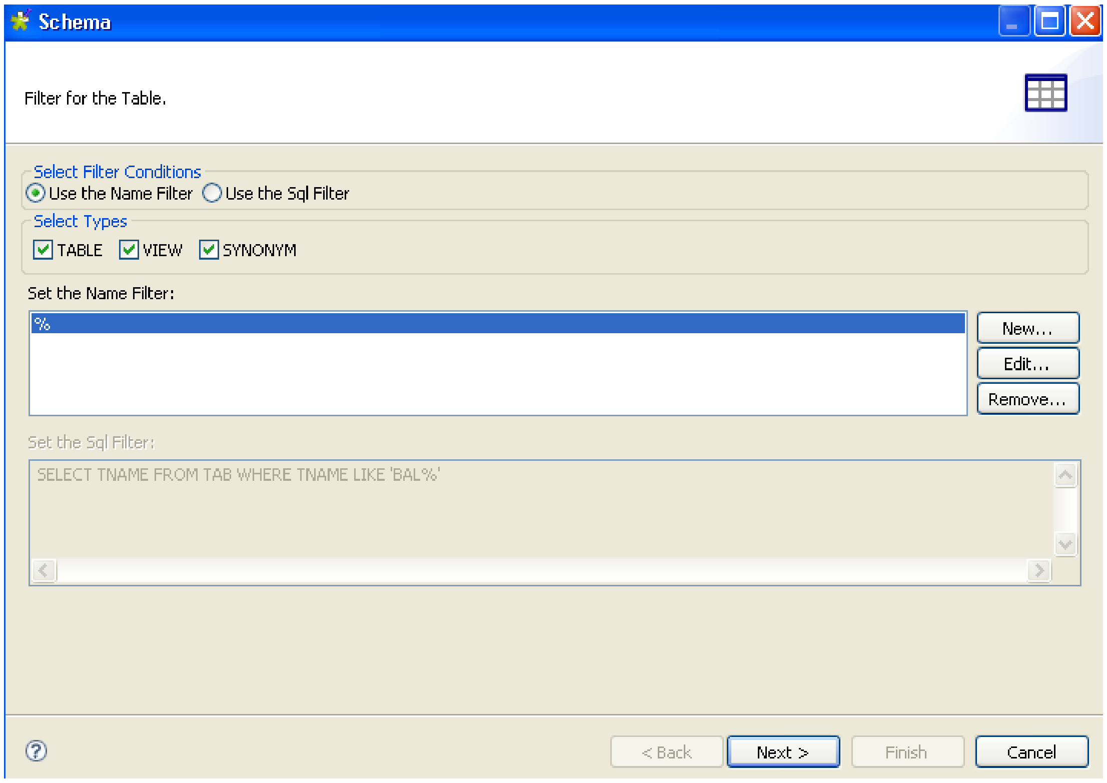
OBSERVE:

```
Starting job ProcessDataWarehouse at 13:00 21/12/2008.
Job ProcessDataWarehouse ended at 13:00 21/12/2008. [exit code=0]
```

After your job runs successfully, you can update the database schema. Right-click on the **Data Warehouse** connection in the metadata section of TOS, and choose **Retrieve Schema**:



We don't need to filter our schema, so click **Next >** at the bottom of the window:


Schema

Filter for the Table.

Select Filter Conditions
(● Use the Name Filter) ○ Use the Sql Filter

Select Types
✓ TABLE ✓ VIEW ✓ SYNONYM

Set the Name Filter:
%

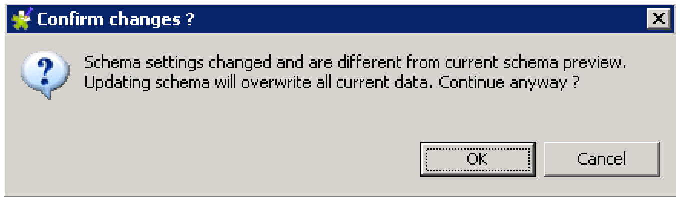
Set the Sql Filter:
SELECT TNAME FROM TAB WHERE TNAME LIKE 'BAL%'

In the next window, scroll through your database until you come across the objects for this course:

- dimCustomer
- dimDate
- dimMovie
- dimStaff
- dimStore
- etlLog
- etlRuns
- factCustomerCount
- factRentalCount
- factRentalDuration
- factSales
- stageFactCustomerCount

Make sure all of those objects are checked, then click **Next >**.

In the final screen, select each table on the left and then click on the **Retrieve Schema** button. You might see a dialog that looks like this:

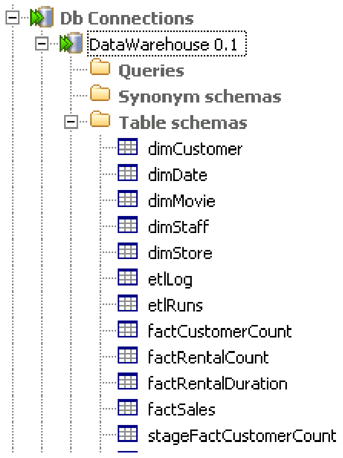

Confirm changes ?

Schema settings changed and are different from current schema preview.
Updating schema will overwrite all current data. Continue anyway ?

Click "OK." Repeat this process for each table to make sure you have the latest schema for all of them. When you are done, click

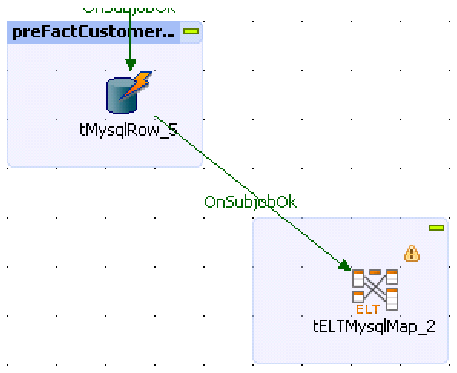
**Finish**.

The repository will still show the tables associated with our connection:



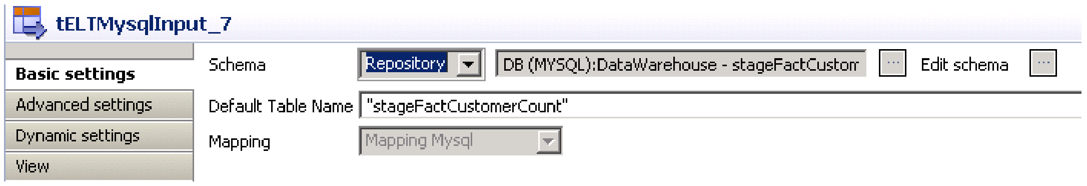
With our schemas updated, we are finally ready to populate **factCustomerCount**.

Start by putting a **tELTMysqlMap** component on your canvas. The **tELTMysqlMap** component acts as the "controller," so you'll want to position it in the middle somewhere. To ensure this component executes after the staging table is loaded with indexes, link the **On Subjob OK** trigger of the previous **tMysqlRow** to **tELTMysqlMap**:



Set the database connection for **tELTMysqlMap** from the repository, to the data warehouse.

Our next task is to add **tELTMysqlInput** components for each source table. Drop a **tELTMysqlInput** component onto the canvas. Edit its properties - set its schema to the **stageFactCustomerCount** table from the repository, under the "Db Connections" and "DataWarehouse:"


tELTMysqlInput_7

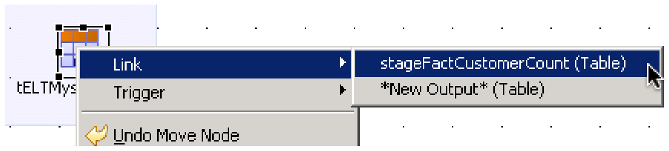
Basic settings | Advanced settings | Dynamic settings | View

Schema: Repository | DB (MYSQL):DataWarehouse - stageFactCustom | Edit schema

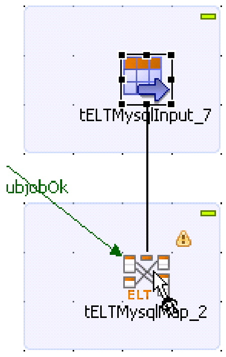
Default Table Name: "stageFactCustomerCount"

Mapping: Mapping Mysql

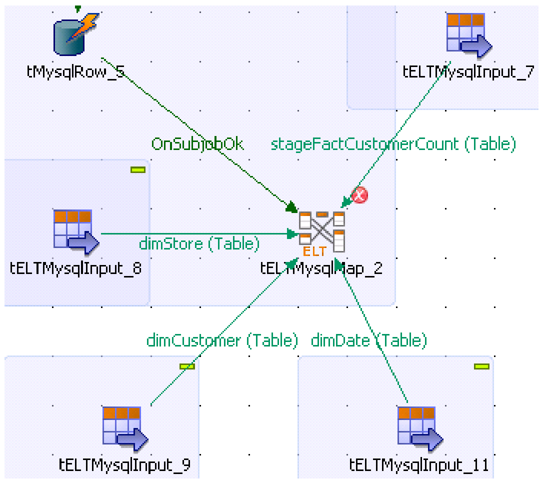
With the metadata set, we can link our **tELTMysqlInput** component to **tELTMysqlMap**. Right-click on **tELTMysqlInput** and choose **Link**, then **stageFactCustomerCount**:



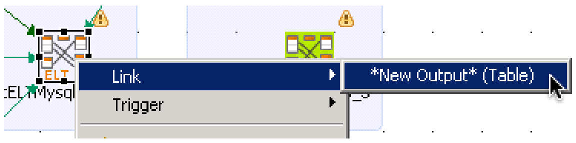
Drop the link on top of **tELTMysqlMap**:



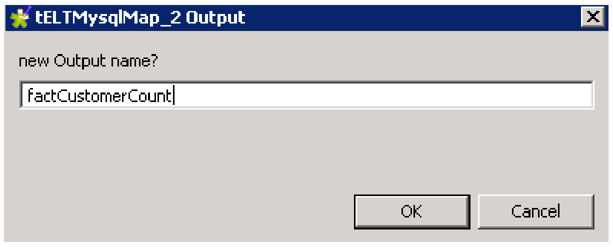
Repeat this process for the three remaining tables - **dimDate**, **dimStore** and **dimCustomer**. When you're done, your job should look something like this:



We have *inputs* to our **tELTMysqlMap** "controller" component, but what about outputs? We only need one output for this application, so drag a single **tELTMysqlOutput** component to the canvas. Right-click on **tELTMysqlMap**, choose Link, and then choose ***new output***:



Name this new output **factCustomerCount**:

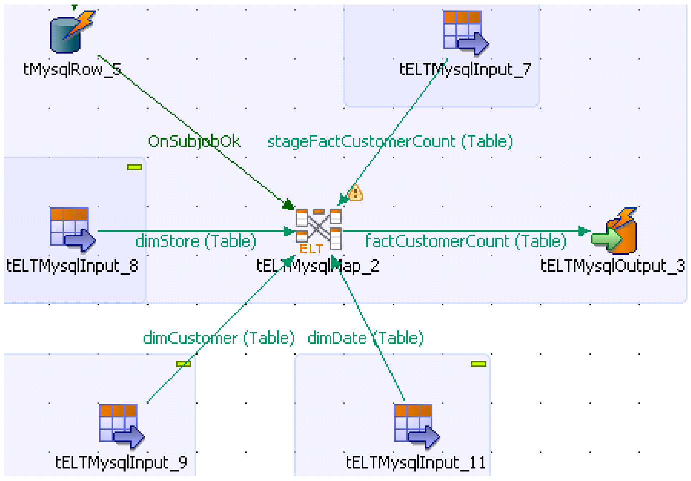

tELTMysqlMap_2 Output

new Output name?

factCustomerCount

OK Cancel

Our job now looks like this:



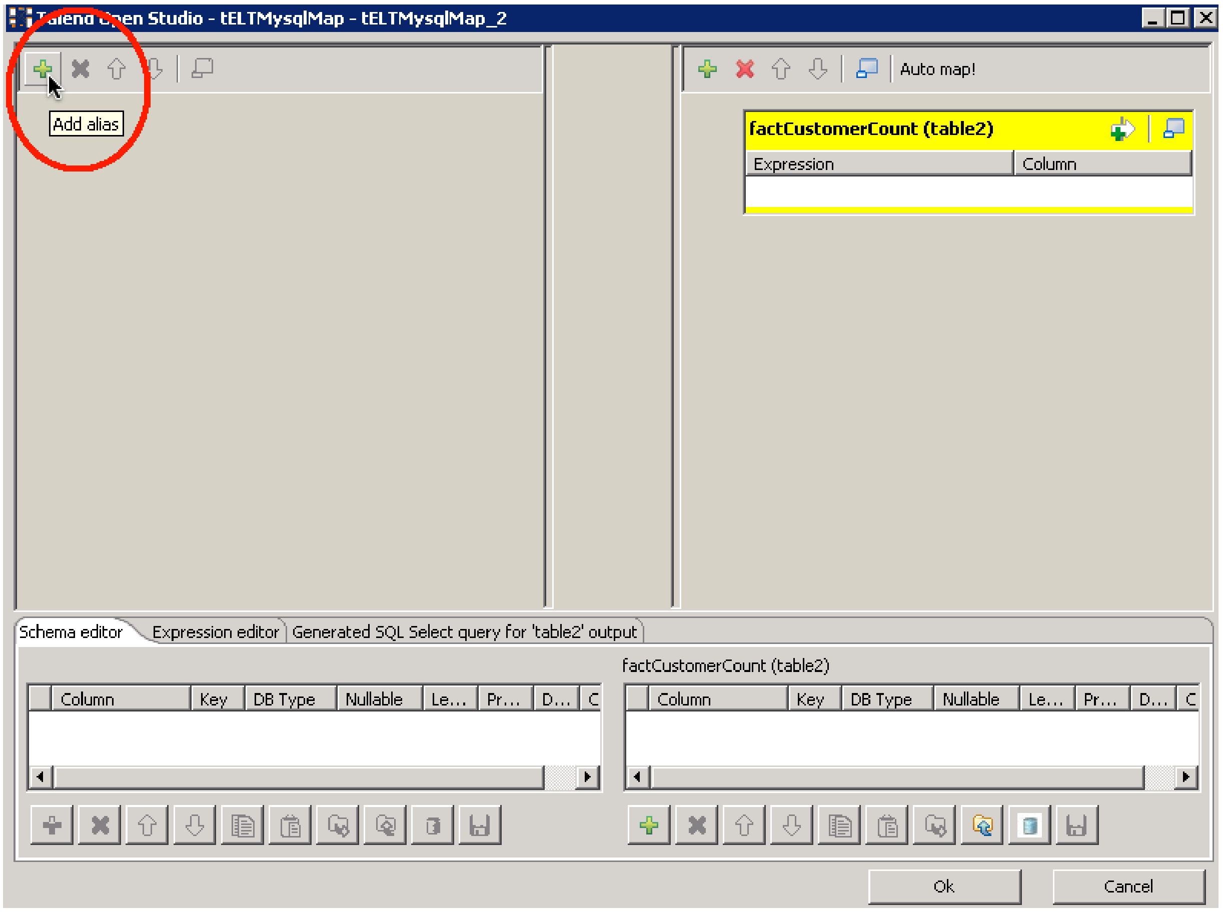
Don't worry about warnings and errors just yet - we are not done configuring our components. You do want to set the database connection for **tELTMysqlOutput** now, however, set it to your personal database.

**tELTMysqlMap** is very similar to **tMap**, with a couple of exceptions:

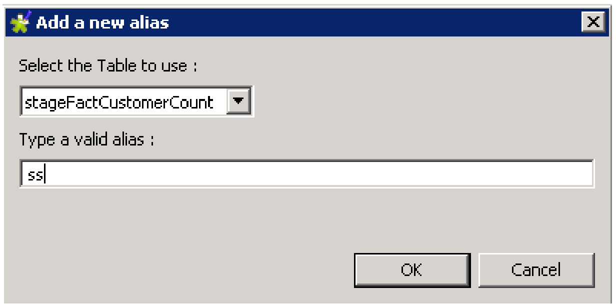
1. **tMap** has a variables section, but **tELTMysqlMap** does not. This is because **tELTMysqlMap** is executed on the Mysql server directly, which does not have any understanding of variables from TOS.
2. **tELTMysqlMap** uses SQL for its expressions, whereas **tMap** uses Java for its expressions.

Our links are now complete, but our component is in error. This is because we haven't specified how we want TOS to combine our inputs into a single output. Double-click on **tELTMysqlMap**.

Once the window loads, click on the ➕ button on the left side, to add an alias for your first input table, `stageFactCustomerCount`. We'll add this table first since it is the basis of all of our joins:


Talend Open Studio - tELTMysqlMap - tELTMysqlMap_2
Add alias
Auto map!
factCustomerCount (table2)
Expression
Column
Schema editor
Expression editor
Generated SQL Select query for 'table2' output
factCustomerCount (table2)
Column
Key
DB Type
Nullable
Le...
Pr...
D...
C
Ok
Cancel

Select the **stageFactCustomerCount** table, and specify **ss** as the *alias*:

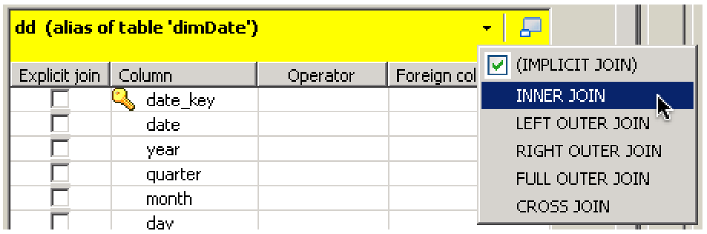


This **ss** *alias* will be translated into the SQL statement. Check it out -- at the bottom of the window, click on **Generated SQL Select query for 'table' output**:

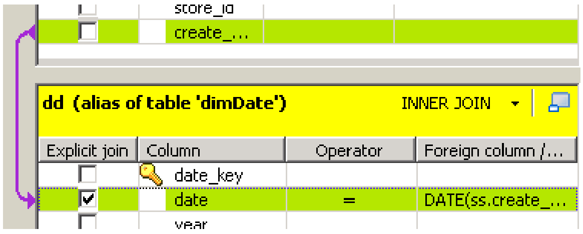
OBSERVE:
```
SELECT

FROM
  stageFactCustomerCount ss
```

Now for the next table, add an alias `dimDate`. Name the alias **dd**.

At this point, **dimDate** is not joined to **stageFactCustomerCount**. To specify how these tables are joined we will do two things. First, click on the triangle drop down to change the join type from `(IMPLICIT JOIN)` to `INNER JOIN`:



Next, check the **Explicit Join** box for the `date` row under **dimDate**, specify **=** as the **Operator**, and specify the **Foreign Column** as **DATE(ss.create_date)**. When you are done you'll see the join:

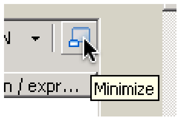


**Note** We need to use the `DATE()` function because `create_date` in `stageFactCustomerCount` is a `DATETIME` data type, and `DATE` in `dimDate` is a `DATE` data type.

We have our first join in place!

**Note** There isn't any way to save your job when you are inside of the **tELTMysqlMap** editor. Be sure to click **OK** and save your work often - you'll loose it if you accidentally hit Cancel!

To make more room on your screen you can collapse the **dimDate** table by clicking on the Minimize/Maximize button:



Next, add an alias called **dc** for the **dimCustomer** table. Collapse the other tables to make room. Then do the following:

1. Change the join type to **INNER JOIN**.
2. Check the **Explicit Join** box for the **customer_id** row.
3. Set the operator to **=**.
4. Set the **Foreign column / expression** to **ss.customer_id**.

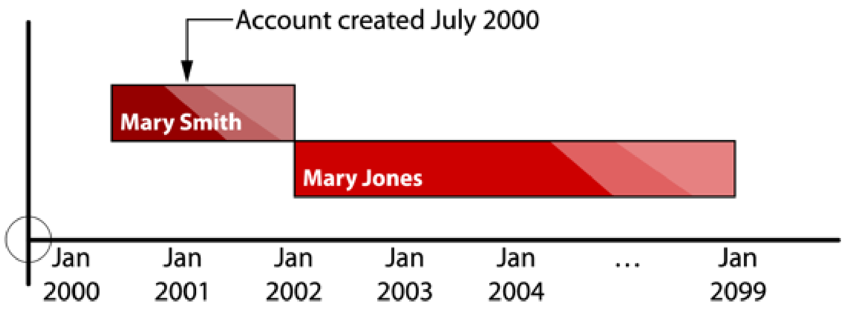
**But wait!** What about the Type-2 columns: `start_date` and `end_date`?

Great question!

It isn't enough to specify the `customer_id` for this join, because there could be multiple rows in `dimCustomer` with the same `customer_id`. We also need to join on `start_date` and `end_date`.

We'll join those columns against `dimDate` because it has a nice date column with the exact type that exists in `dimCustomer`. As for the join operator, we are looking for a row in `dimCustomer` that has a `start_date` less than or equal to `create_date` and an `end_date` greater than `create_date`.

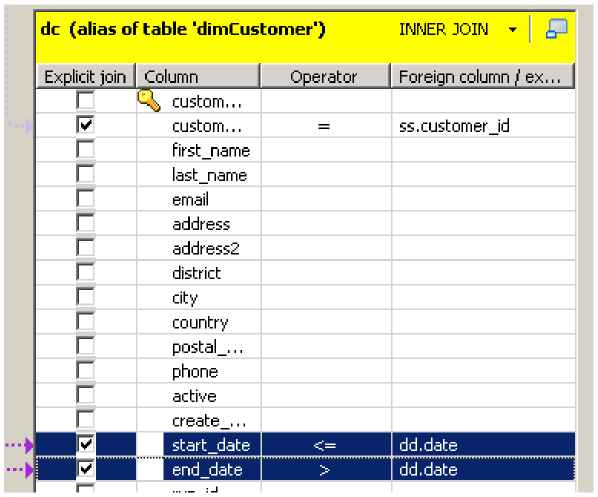
On a time line, we need to pick the row in dimDate for a specific time period:



To join on these columns as well, do the following:

1. Check the **Explicit Join** box for the **start_date** and **end_date** rows.
2. For **start_date**, set the operator to **<=** .
3. For **end_date**, set the operator to **>**.
4. For both columns, set the **Foreign column / expression** to **dd.date**.

When you're done, your join will look like this:



| Explicit join | Column | Operator | Foreign column / ex... |
|---|---|---|---|
| ☐ | custom... | | |
| ☑ | custom... | = | ss.customer_id |
| ☐ | first_name | | |
| ☐ | last_name | | |
| ☐ | email | | |
| ☐ | address | | |
| ☐ | address2 | | |
| ☐ | district | | |
| ☐ | city | | |
| ☐ | country | | |
| ☐ | postal_... | | |
| ☐ | phone | | |
| ☐ | active | | |
| ☐ | create_... | | |
| ☑ | start_date | <= | dd.date |
| ☑ | end_date | > | dd.date |

The query at the bottom of the window should look like this:

OBSERVE:

```
SELECT

FROM
 stageFactCustomerCount ss INNER JOIN  dimDate dd ON(  dd.date = DATE(ss.create_date) )
 INNER JOIN  dimCustomer dc ON(  dc.customer_id = ss.customer_id AND  dc.start_date <= dd.date AND  dc
```

Finally, add an alias called **ds** for the **dimStore** table. Once again, this dimension is a *Type-2*. Execute the following steps:

1. Change the join type to **INNER JOIN**.
2. Check the **Explicit Join** box for the **store_id** row.
3. Set the operator to **=**
4. Set the **Foreign column / expression** to **ss.store_id**.
5. Check the **Explicit Join** box for the **start_date** and **end_date** rows.
6. For **start_date**, set the operator to **<=** .
7. For **end_date**, set the operator to **>**.
8. For both columns, set the **Foreign column / expression** to **dd.date**.

Once that's done, your query will look like this:

OBSERVE:

```
SELECT

FROM
 stageFactCustomerCount ss INNER JOIN  dimDate dd ON(  dd.date = DATE(ss.create_date) )
 INNER JOIN  dimCustomer dc ON(  dc.customer_id = ss.customer_id AND  dc.start_date <= dd.date AND  dc
 INNER JOIN  dimStore ds ON(  ds.store_id = ss.store_id AND  ds.start_date <= dd.date AND  ds.end_date
```

**Note** Are you missing a table? Make sure you set the join type to `INNER JOIN`.

The only thing left to do is to specify our outputs. But before we do, let's review the structure of `factCustomerCount`. Run the following command against your personal database:

CODE TO TYPE:

```
explain factCustomerCount;
```

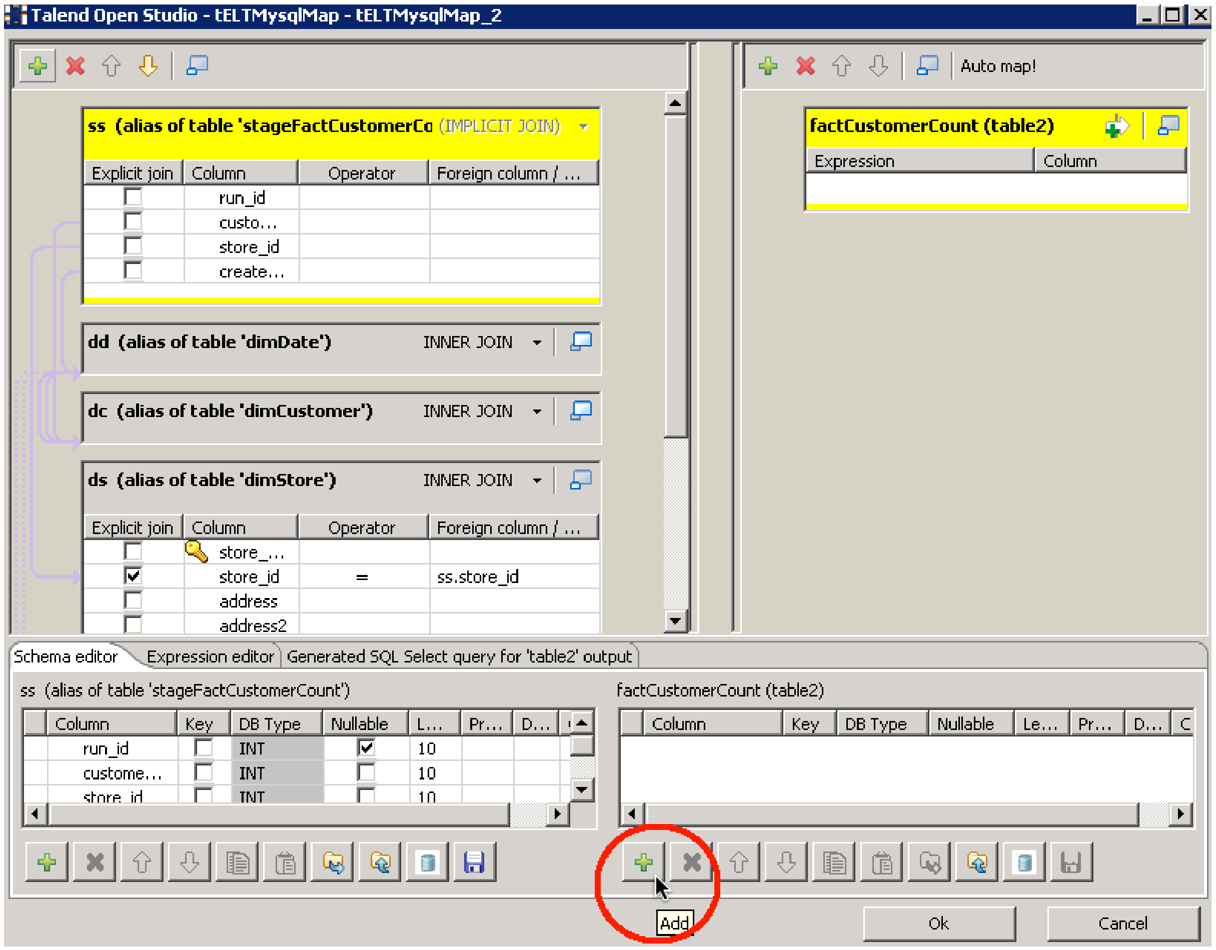
You'll see these results:

OBSERVE:

```
mysql> explain factCustomerCount;
+-------------------+---------+------+-----+---------+----------------+
| Field             | Type    | Null | Key | Default | Extra          |
+-------------------+---------+------+-----+---------+----------------+
| customerCount_key | int(11) | NO   | PRI | NULL    | auto_increment |
| date_key          | int(11) | NO   |     | NULL    |                |
| customer_key      | int(11) | NO   |     | NULL    |                |
| store_key         | int(11) | NO   |     | NULL    |                |
| customer_count    | int(11) | NO   |     | 1       |                |
| run_id            | int(11) | NO   |     | 1       |                |
+-------------------+---------+------+-----+---------+----------------+
6 rows in set (0.05 sec)
```

We need to specify all of these columns in **tELTMysqlMap**. Two columns are not present - our *surrogate key*, `customerCount_key`, and our `customer_count` column.

**ORDER IS IMPORTANT** for these columns, so we'll start by adding `customerCount_key`. To add this column, click the ➕ button:



Talend Open Studio - tELTMysqlMap - tELTMysqlMap_2

ss (alias of table 'stageFactCustomerCo (IMPLICIT JOIN)

| Explicit join | Column | Operator | Foreign column / ... |
|---|---|---|---|
| ☐ | run_id | | |
| ☐ | custo... | | |
| ☐ | store_id | | |
| ☐ | create... | | |

dd (alias of table 'dimDate') INNER JOIN

dc (alias of table 'dimCustomer') INNER JOIN

ds (alias of table 'dimStore') INNER JOIN

| Explicit join | Column | Operator | Foreign column / ... |
|---|---|---|---|
| ☐ | store_... | | |
| ✓ | store_id | = | ss.store_id |
| ☐ | address | | |
| ☐ | address2 | | |

Auto map!

factCustomerCount (table2)

| Expression | Column |
|---|---|

Schema editor | Expression editor | Generated SQL Select query for 'table2' output

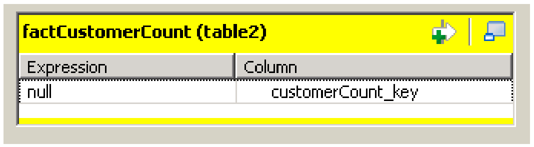
ss (alias of table 'stageFactCustomerCount')

| Column | Key | DB Type | Nullable | L... | Pr... | D... |
|---|---|---|---|---|---|---|
| run_id | ☐ | INT | ✓ | 10 | | |
| custome... | ☐ | INT | ☐ | 10 | | |
| store_id | ☐ | INT | ☐ | 10 | | |

factCustomerCount (table2)

| Column | Key | DB Type | Nullable | Le... | Pr... | D... | C |
|---|---|---|---|---|---|---|---|

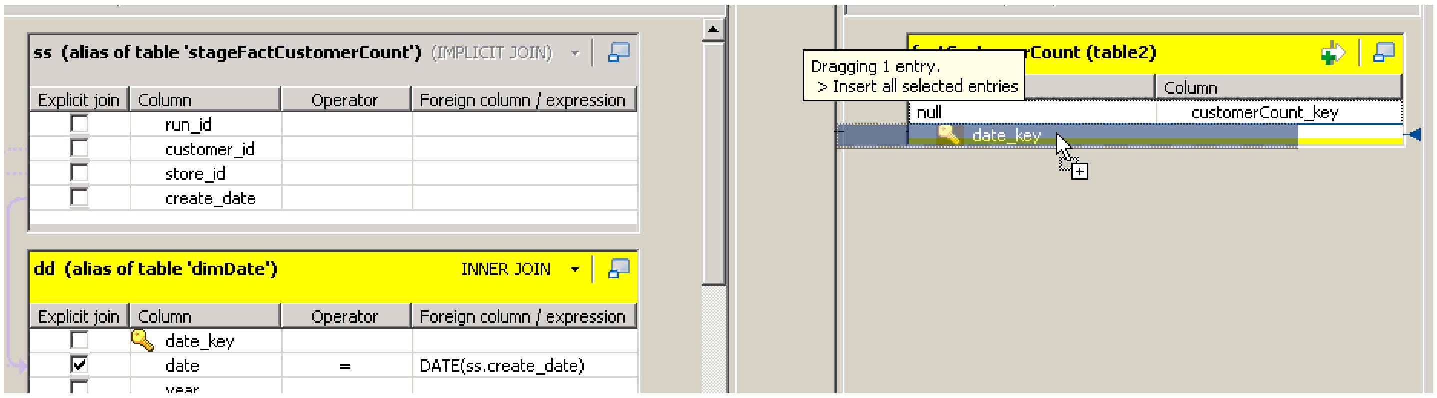
Name the column `customerCount_key`, and make sure you keep the **Nullable** box checked. Under the **Expression** column above, type in `NULL`:


factCustomerCount (table2)

| Expression | Column |
|---|---|
| null | customerCount_key |

**Note** To reiterate, **order matters** for this section, so use care when adding these columns. You can always reorder the columns using the up and down arrow buttons at the bottom of the window.

Expand **dimDate**, then drag **date_key** over to the output table, right underneath **customerCount_key**:


ss (alias of table 'stageFactCustomerCount') (IMPLICIT JOIN)

| Explicit join | Column | Operator | Foreign column / expression |
|---|---|---|---|
| ☐ | run_id | | |
| ☐ | customer_id | | |
| ☐ | store_id | | |
| ☐ | create_date | | |

dd (alias of table 'dimDate') INNER JOIN

| Explicit join | Column | Operator | Foreign column / expression |
|---|---|---|---|
| ☐ | date_key | | |
| ☑ | date | = | DATE(ss.create_date) |
| ☐ | year | | |

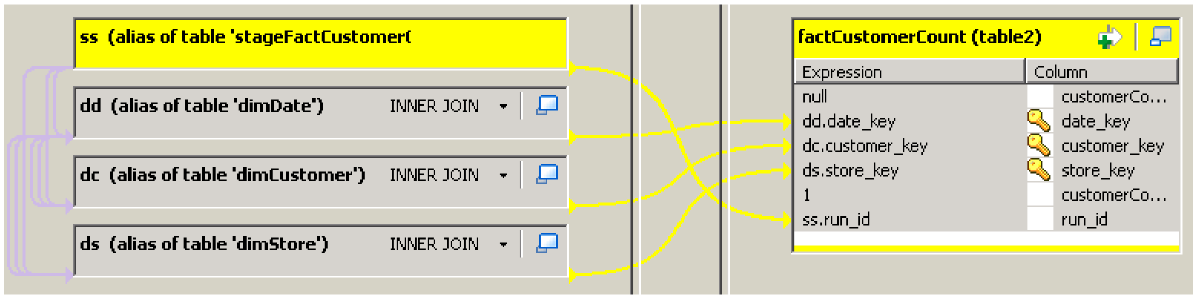
Dragging 1 entry.
> Insert all selected entries

| Expression | Column |
|---|---|
| null | customerCount_key |
| | date_key |

Repeat this process, **in order**, for **customer_key** and **store_key**.

The next column is **customerCount**. Add this by clicking the button. Name the column `customerCount`, and make sure you keep the **Nullable** box checked. Under the **Expression** column above, type in `1`.

Finally, drag the **run_id** column to the output. Your completed map will look like this:


ss (alias of table 'stageFactCustomer(
dd (alias of table 'dimDate') INNER JOIN
dc (alias of table 'dimCustomer') INNER JOIN
ds (alias of table 'dimStore') INNER JOIN

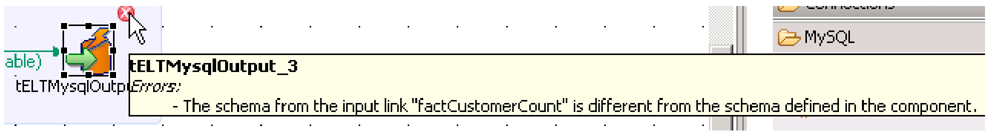
factCustomerCount (table2)

| Expression | Column |
|---|---|
| null | customerCo... |
| dd.date_key | date_key |
| dc.customer_key | customer_key |
| ds.store_key | store_key |
| 1 | customerCo... |
| ss.run_id | run_id |

Your query will look like this:

```
OBSERVE:
SELECT
null, dd.date_key , dc.customer_key , ds.store_key , null, ss.run_id
FROM
 stageFactCustomerCount ss INNER JOIN  dimDate dd ON(  dd.date = DATE(ss.create_date) )
 INNER JOIN  dimCustomer dc ON(  dc.customer_id = ss.customer_id AND  dc.start_date <= dd.date AND  dc
 INNER JOIN  dimStore ds ON(  ds.store_id = ss.store_id AND  ds.start_date <= dd.date AND  ds.end_date
```

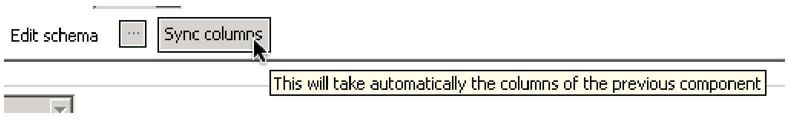
There is one last thing we need to do before we run our job. Currently there's an error on **tELTMysqlOutput**:



tELTMysqlOutput_3
*Errors:*
- The schema from the input link "factCustomerCount" is different from the schema defined in the component.

**Note** You can ignore the warning on **tELTMysqlMap** - TOS is just confused by the table aliases.

To fix this error, double click **tELTMysqlOutput**, then click on **Sync Columns**:



This will take automatically the columns of the previous component

We are now ready to run the job. Click on the at the top of the window.

As long as your components are not in error, you'll see output that looks like this:

OBSERVE:

```
Starting job ProcessDataWarehouse at 21:07 22/12/2008.
Inserting with :
INSERT INTO factCustomerCount (SELECT null, dd.date_key , dc.customer_key , ds.store_key , 1, ss.run_i

--> 589 rows inserted.

Job ProcessDataWarehouse ended at 21:07 22/12/2008. [exit code=0]
```

You did it! **factCustomerCount** now has data in it!

We covered a whole lot in this lesson! We'll finish implementing the rest of the facts in the next lesson. See you in a bit!

# Processing Facts, Part II
# DBA 3: Data Warehousing Lesson 11

In the last lesson we implemented `factCustomerCount`. In this lesson we will look at our largest fact, `factSales`.

## factSales

It was back in lesson four when we implemented the table for `factSales`. Let's review its structure by reviewing the `CREATE TABLE` statement we used:

OBSERVE:

```
CREATE TABLE factSales
(
 sales_key        INT NOT NULL AUTO_INCREMENT,
 date_key         INT NOT NULL REFERENCES dimDate,
 customer_key     INT NOT NULL REFERENCES dimCustomer,
 movie_key        INT NOT NULL REFERENCES dimMovie,
 store_key        INT NOT NULL REFERENCES dimStore,
 sales_amount     decimal(5,2) NOT NULL,
 PRIMARY KEY (sales_key)
);
```

There is only one numeric value in this **fact**: *sales_amount*. The rest are foreign keys that point to dimensions.

To process our sales **fact**, we will perform the following steps:

1. Move data from the source system into the staging table called `stageFactSales`.
2. Create indexes on the staging table.
3. Use the ELT family of components in TOS to populate `factSales`.

Let's get started by populating our staging table, `stageFactSales`. To do this, drag these three components to your canvas: **tMysqlInput**, **tMap** and **tMysqlOutput**.

Edit the properties for **tMysqlInput**, setting its connection to the **sakila** database. Next, edit its query. Use this query to retrieve sales data:

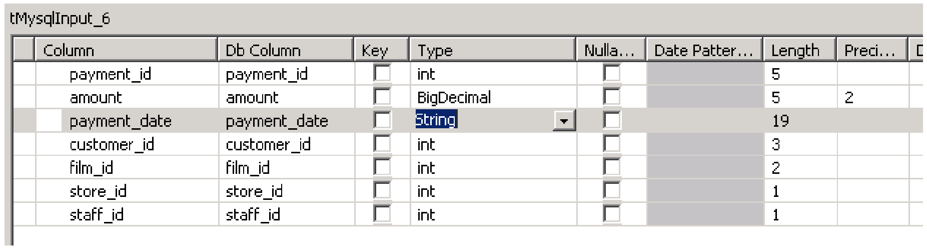
CODE TO TYPE:

```
select p.payment_id, p.amount, p.payment_date,
p.customer_id, i.film_id, i.store_id, p.staff_id
from
payment p
join rental r on ( p.rental_id = r.rental_id )
join inventory i on ( r.inventory_id = i.inventory_id )
WHERE p.customer_id > 10
```

Run the query to make sure you typed everything correctly.

Once that's done, it's time to specify the schema for **tMysqlInput**. For this query, we can use the **Guess Schema** button to do most of the work; click on it.

TOS identifies most of the columns without difficulty, except for one: **amount**. This isn't a *float* value, it's a *BigDecimal* (decimal) value of length 5 and precision 2. Fix the schema. It will ultimately look like this:



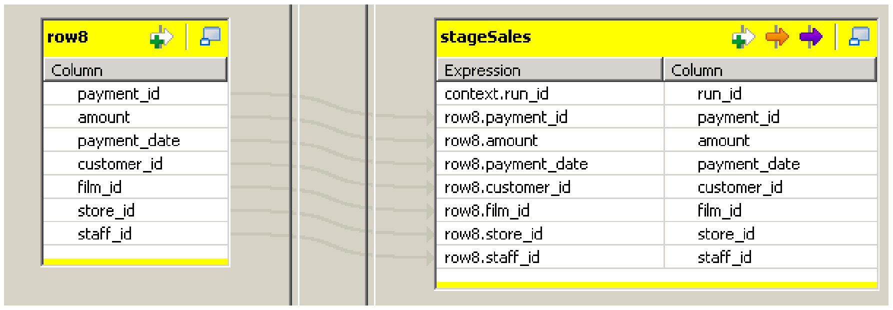
tMysqlInput_6

| Column | Db Column | Key | Type | Nulla... | Date Patter... | Length | Preci... |
|---|---|---|---|---|---|---|---|
| payment_id | payment_id | ☐ | int | ☐ | | 5 | |
| amount | amount | ☐ | BigDecimal | ☐ | | 5 | 2 |
| payment_date | payment_date | ☐ | String | ☐ | | 19 | |
| customer_id | customer_id | ☐ | int | ☐ | | 3 | |
| film_id | film_id | ☐ | int | ☐ | | 2 | |
| store_id | store_id | ☐ | int | ☐ | | 1 | |
| staff_id | staff_id | ☐ | int | ☐ | | 1 | |

**Note** You might be wondering about the differences between **float**, **double**, **decimal** and **integer** numbers. **Float** and double are *approximate* numeric types, meaning the computer may not store your value exactly like you enter it "under the hood." So, even if two float numbers look the same to you, the computer may not store them the same way, and they may not be equal to one another, at least according to the computer. **Decimal** and integer are exact numeric types, so what you see is what the computer stores.

With that specified, link **tMysqlInput** to **tMap**, then open **tMap** to edit its connections.

We will use this **tMap** just like we used the map components for dimension processing. Add our auditing column called `run_id`. Add an output, then add the `run_id` column. Finally, drag all of the input columns to the output. When you're done, your **tMap** should look somewhat like this:



**row8**

| Column |
|---|
| payment_id |
| amount |
| payment_date |
| customer_id |
| film_id |
| store_id |
| staff_id |

**stageSales**

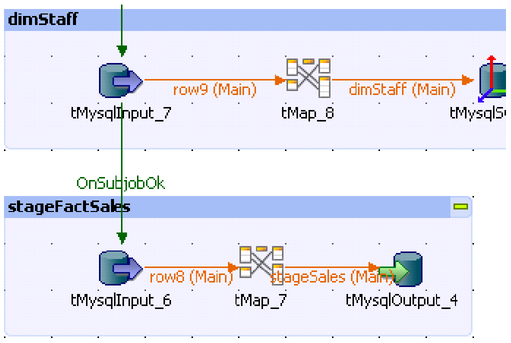
| Expression | Column |
|---|---|
| context.run_id | run_id |
| row8.payment_id | payment_id |
| row8.amount | amount |
| row8.payment_date | payment_date |
| row8.customer_id | customer_id |
| row8.film_id | film_id |
| row8.store_id | store_id |
| row8.staff_id | staff_id |

Next, connect **tMap** to **tMysqlOutput**. Set the connection properties on **tMysqlOutput** to the data warehouse connection from the repository, and set the table to **"stageFactSales"** (within quotation marks).

We could manually create the **stageFactSales** table, but then we would have to delete its data and drop its indexes before we populate it with data. Instead of doing that, we can have the **tMysqlOutput** component **Drop table if exists and create**, which accomplishes the same thing. To do this, set the *action on table* to **Drop table if exists and create**.

Right-click on the **tMysqlInput** component of **factRentalCount**, and connect the **On Subjob OK** trigger to the **tMysqlInput** of the **stageFactSales** sub job.

When you're done, your job will look something like this:



Next, we need to create our stored procedure to create our indexes and clear the `factSales` table. Switch to the terminal, and run the following query:

CODE TO TYPE:

```
DELIMITER //
CREATE PROCEDURE etl_preFactSales ()
BEGIN
ALTER TABLE stageFactSales add index(payment_id);
ALTER TABLE stageFactSales add index(payment_date);
ALTER TABLE stageFactSales add index(customer_id);
ALTER TABLE stageFactSales add index(film_id);
ALTER TABLE stageFactSales add index(store_id);
TRUNCATE TABLE factSales;
END
//
```

Now, add a **tMysqlRow** component to your canvas. Set its database connection to the data warehouse, and set its query to the following:

CODE TO TYPE:

```
call etl_preFactSales();
```

This procedure cannot execute before we populate `stageFactSales` with data. Right-click on the **tMysqlInput** component of the **stageFactSales** sub job and link its **On Subjob OK** trigger to **tMysqlRow**.

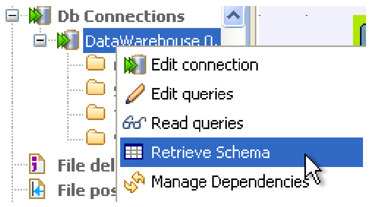
In the last lesson, we updated our table schema to include `stageFactCustomerCount`. We need to do the same for `stageFactSales`. First, run your job to create the table. Click on the button at the top of the window.

As long as you don't have any errors in your job, you'll see something like the following output:

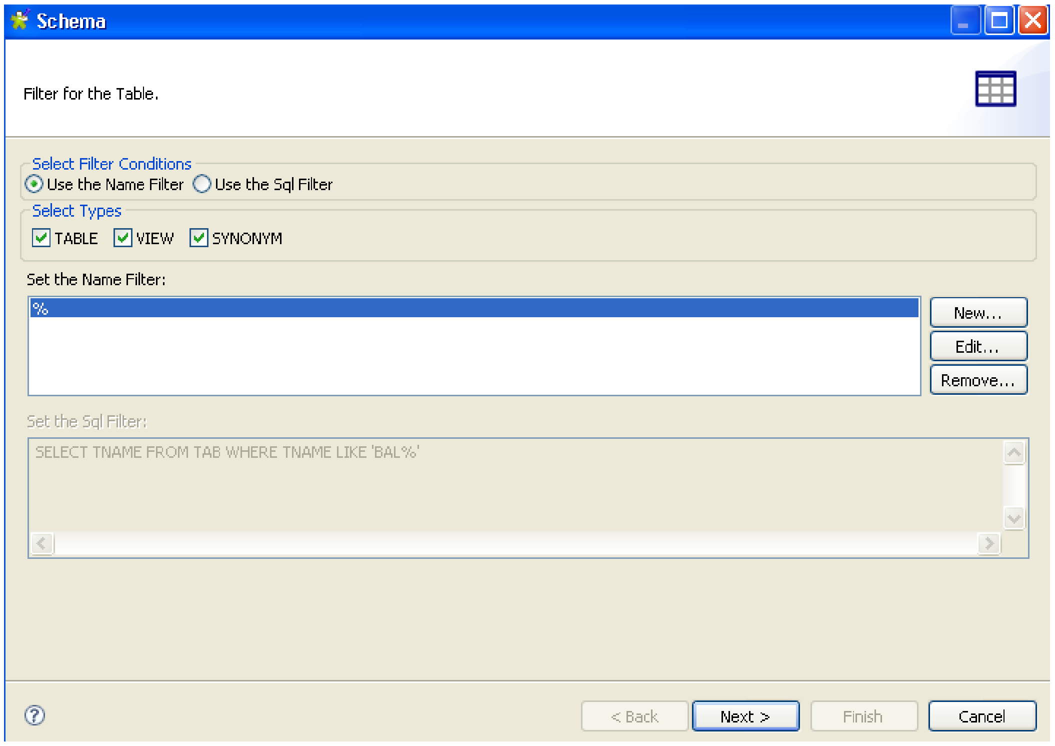
OBSERVE:

```
Starting job ProcessDataWarehouse at 13:00 21/12/2008.
Job ProcessDataWarehouse ended at 13:00 21/12/2008. [exit code=0]
```

After your job runs successfully, you can update the database schema by right-clicking on the **Data Warehouse** connection in the metadata section of TOS, and choosing **Retrieve Schema**:



We don't need to filter our schema, so click **Next >** at the bottom of the window:



Schema

Filter for the Table.

Select Filter Conditions
[✓ Use the Name Filter] Use the Sql Filter

Select Types
[✓ TABLE] [✓ VIEW] [✓ SYNONYM]

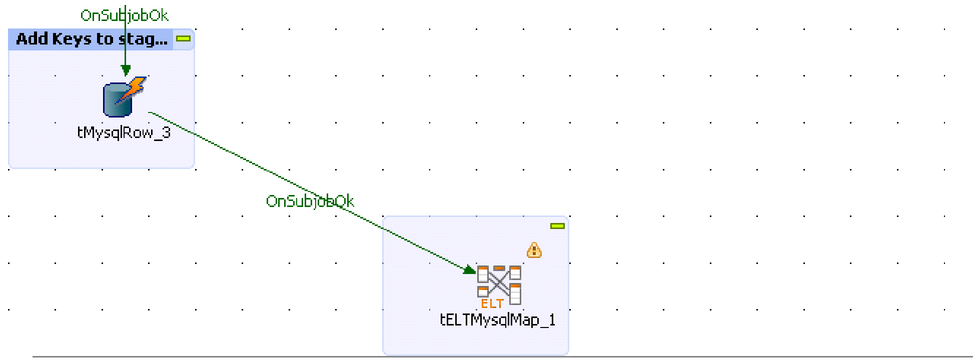
Set the Name Filter:
%

Set the Sql Filter:
SELECT TNAME FROM TAB WHERE TNAME LIKE 'BAL%'

In the next window, scroll through your database until you come across the objects for this course, then check **stageFactSales**.

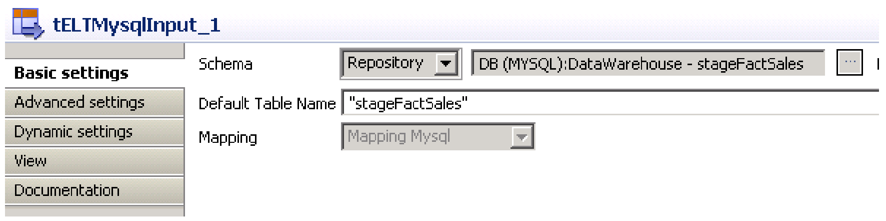
In the final screen, select **stageFactSales** on the left and then click on the **Retrieve Schema** button.

Now we're ready to create our map. Drag a **tELTMysqlMap** component to your canvas. To ensure this component executes after the staging table is loaded with indexes, link the **On Subjob OK** trigger of the previous **tMysqlRow** to **tELTMysqlMap**:



**Note** Be sure to Set the database connection for **tELTMysqlMap** to the data warehouse, from the repository.

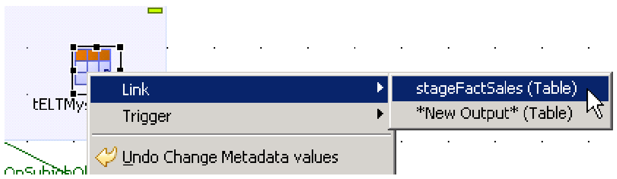
Next, we need to add **tELTMysqlInput** components for each source table. Start by adding a component for **stageFactSales**:



tELTMysqlInput_1

| | |
|---|---|
| Schema | Repository — DB (MYSQL):DataWarehouse - stageFactSales |
| Default Table Name | "stageFactSales" |
| Mapping | Mapping Mysql |

With the metadata set, we can link our **tELTMysqlInput** component to **tELTMysqlMap**. Right-click on **tELTMysqlInput** and choose **Link**, then **stageFactSales**:

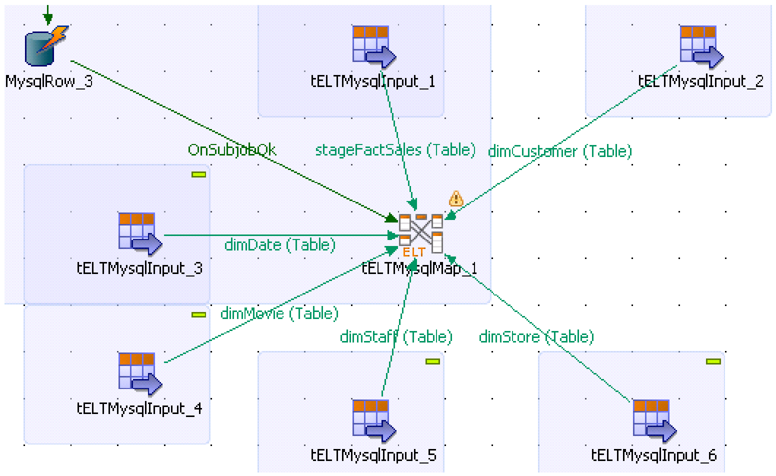


Drop the link on top of **tELTMysqlMap**.

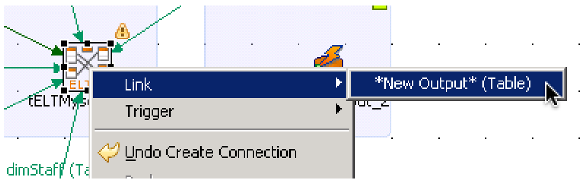
Repeat this process for the remaining required tables:

- dimCustomer
- dimDate
- dimMovie
- dimStaff
- dimStore

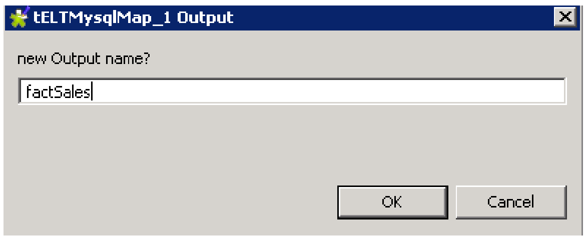
When you are done, your canvas will look something like this:



We have *inputs* to our **tELTMysqlMap** "controller" component, but what about outputs? We only need one output for this application, so drag a single **tELTMysqlOutput** component to the canvas. Right-click on **tELTMysqlMap**, choose Link, then choose ***new output***:



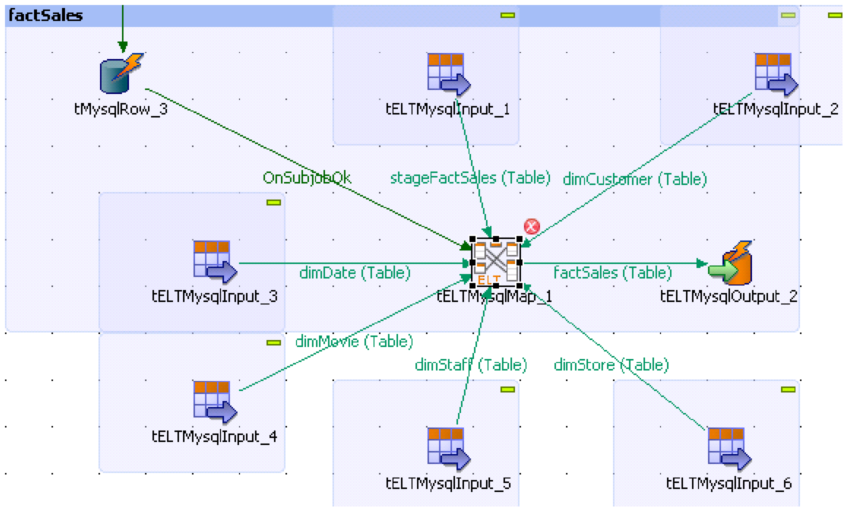
Name this new output **factSales**:



new Output name?

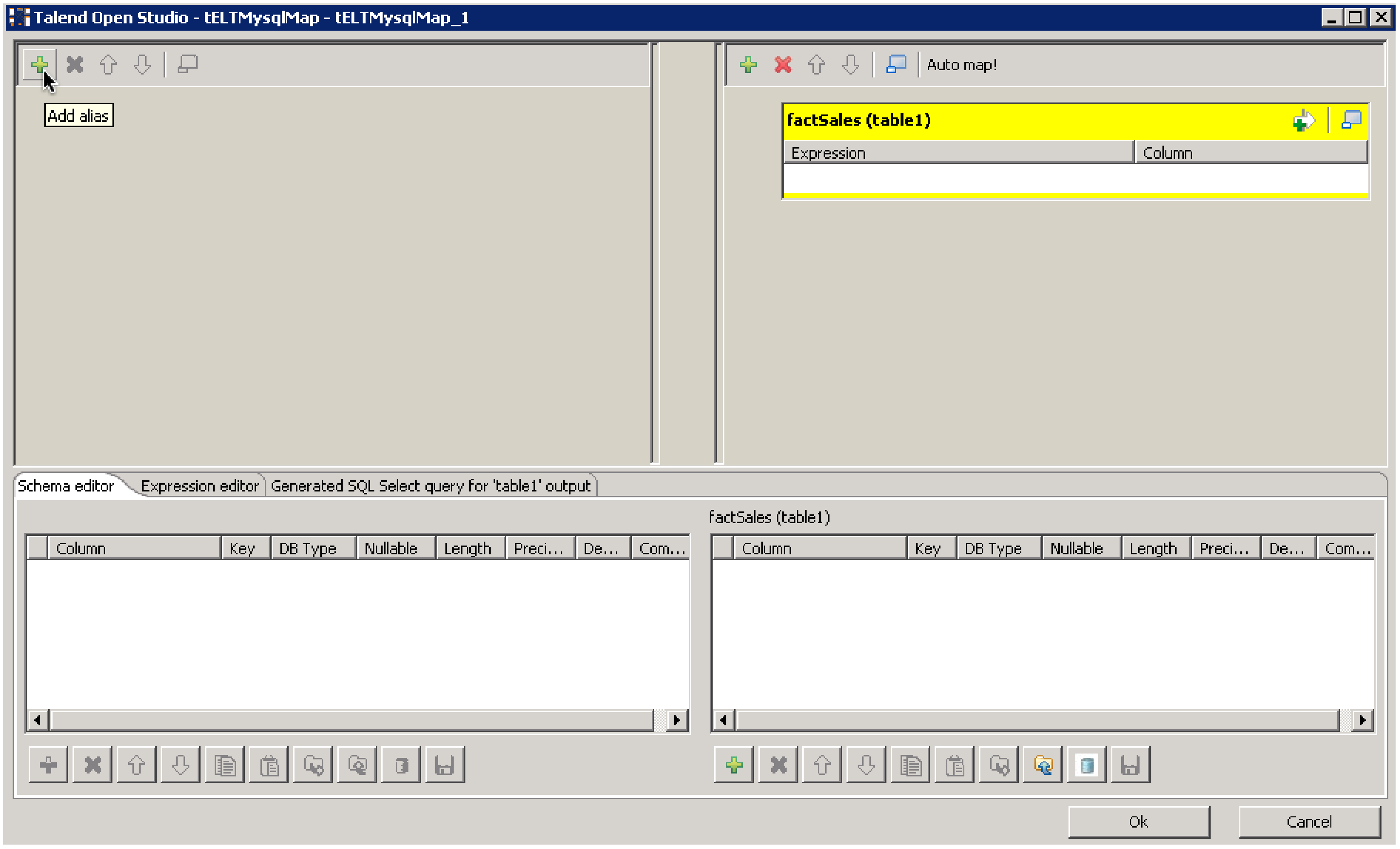
factSales

Our job now looks like this:

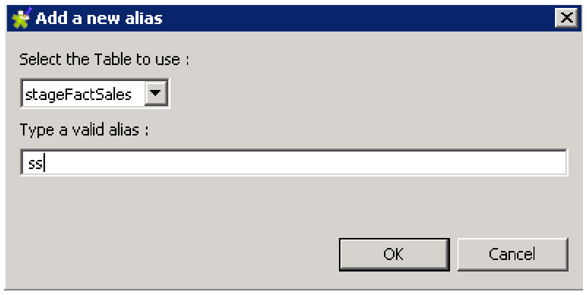


We're ready to set up our joins. Double click on **tELTMysqlMap**.

Once the window loads, click on the ✚ button on the left side to add an alias for your first input table, `stageFactSales`. We'll add this table first since it is the basis of all of our joins:



Select the **stageFactSales** table, and specify **ss** as the *alias*:



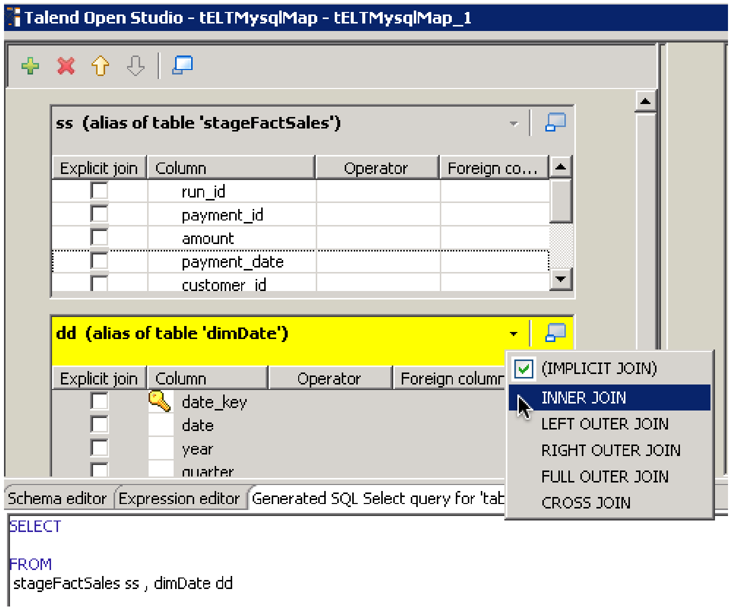
This **ss** *alias* will be translated into the SQL statement. At the bottom of the window, click on **Generated SQL Select query for 'table' output**:

OBSERVE:

```
SELECT

FROM
  stageFactSales ss
```

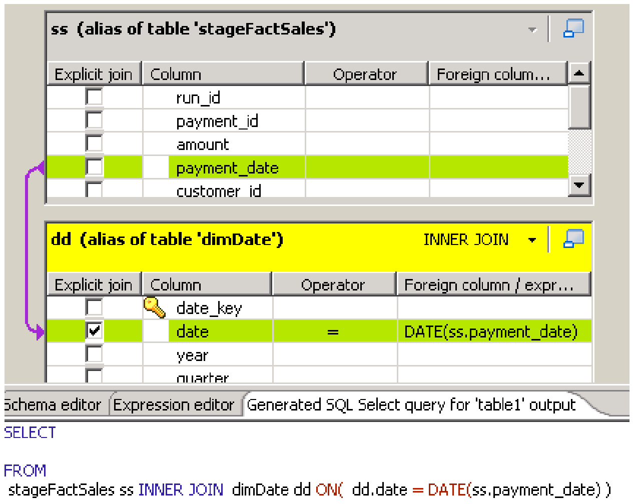
Now add an alias to the next table, `dimDate`. Name the alias **dd**.

**dimDate** is not joined to **stageFactSales** now. To specify how these tables are joined we will do two things: first, click on the triangle drop down to change the join type from `(IMPLICIT JOIN)` to `INNER JOIN`:



```
SELECT

FROM
stageFactSales ss , dimDate dd
```

Next, check the **Explicit Join** box for the `date` row under **dimDate**, specify **=** as the **Operator**, and specify the **Foreign Column** as **DATE(ss.payment_date)**. When you are done you'll see the join:



```
SELECT

FROM
 stageFactSales ss INNER JOIN  dimDate dd ON(  dd.date = DATE(ss.payment_date) )
```

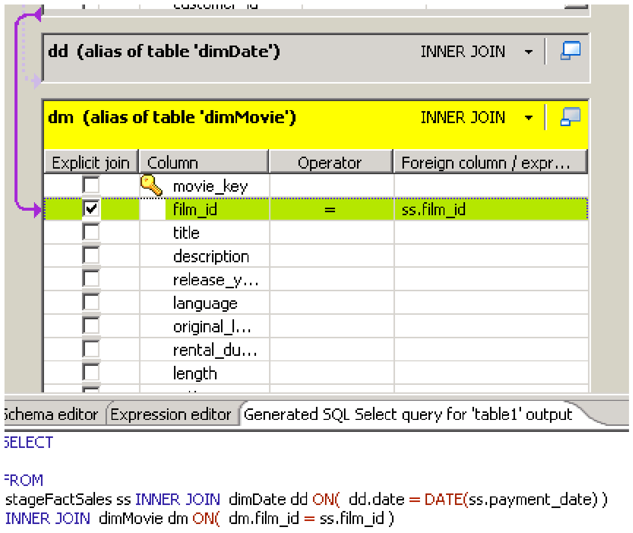
Now is a good time to save. (Actually, it's almost always a good time to save.) Click "OK" to close the map window, then save your work.

Let's move on to our next input. Add an alias for the table **dimMovie**, called **dm**.

With `dimDate` out of the way, we can specify the join to **dimMovie**. Now take these steps:

1. Change the join type to **INNER JOIN**.
2. Check the **Explicit Join** box for the **film_id** row.
3. Set the operator to **=**.
4. Set the **Foreign column / expression** to **ss.film_id**.

When you are done, you're join will show:



dd (alias of table 'dimDate') INNER JOIN

dm (alias of table 'dimMovie') INNER JOIN

| Explicit join | Column | Operator | Foreign column / expr... |
|---|---|---|---|
| ☐ | movie_key | | |
| ✓ | film_id | = | ss.film_id |
| ☐ | title | | |
| ☐ | description | | |
| ☐ | release_y... | | |
| ☐ | language | | |
| ☐ | original_l... | | |
| ☐ | rental_du... | | |
| ☐ | length | | |

Schema editor | Expression editor | Generated SQL Select query for 'table1' output

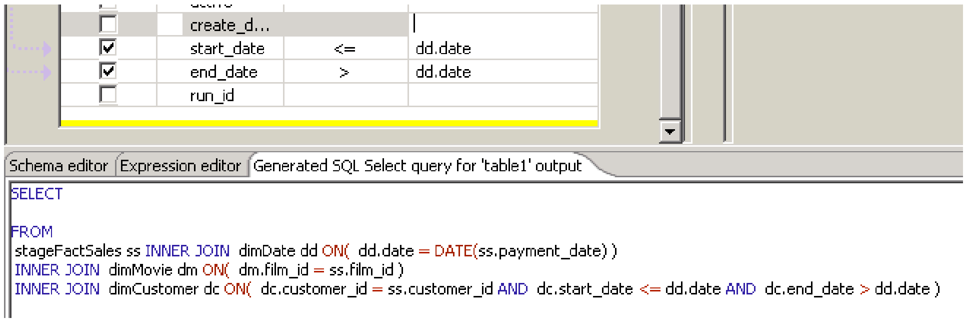
```
SELECT

FROM
stageFactSales ss INNER JOIN  dimDate dd ON(  dd.date = DATE(ss.payment_date) )
INNER JOIN  dimMovie dm ON(  dm.film_id = ss.film_id )
```

Three tables down, three more to go!

Next, add an alias called **dc** for the **dimCustomer** table, a Type-2 SCD. Collapse the other tables to make room. Then execute these steps:

1. Change the join type to **INNER JOIN**.
2. Check the **Explicit Join** box for the **customer_id** row.
3. Set the operator to **=**.
4. Set the **Foreign column / expression** to **ss.customer_id**.
5. Check the **Explicit Join** box for the **start_date** and **end_date** rows.
6. For **start_date**, set the operator to **<=** .
7. For **end_date**, set the operator to **>**.
8. For both columns, set the **Foreign column / expression** to **dd.date**.

When you're done, your join will look like this:



| | | | |
|---|---|---|---|
| ☐ | create_d... | | |
| ✓ | start_date | <= | dd.date |
| ✓ | end_date | > | dd.date |
| ☐ | run_id | | |

Schema editor | Expression editor | Generated SQL Select query for 'table1' output

```
SELECT

FROM
stageFactSales ss INNER JOIN  dimDate dd ON(  dd.date = DATE(ss.payment_date) )
INNER JOIN  dimMovie dm ON(  dm.film_id = ss.film_id )
INNER JOIN  dimCustomer dc ON(  dc.customer_id = ss.customer_id AND  dc.start_date <= dd.date AND  dc.end_date > dd.date )
```

We're almost there! Add an alias called **dst** for the **dimStaff** table. Collapse the other tables to make room. This dimension is also a *Type-2*, so we'll have to join on `start_date` and `end_date` again. Now execute these steps:

1. Change the join type to **INNER JOIN**.
2. Check the **Explicit Join** box for the **staff_id** row.
3. Set the operator to **=**.
4. Set the **Foreign column / expression** to **ss.staff_id**.
5. Check the **Explicit Join** box for the **start_date** and **end_date** rows.
6. For **start_date**, set the operator to **<=** .
7. For **end_date**, set the operator to **>**.
8. For both columns, set the **Foreign column / expression** to **dd.date**.

Finally, add an alias called **ds** for the **dimStore** table. Once again, this dimension is a *Type-2* Execute the following steps:

1. Change the join type to **INNER JOIN**.
2. Check the **Explicit Join** box for the **store_id** row.
3. Set the operator to **=**.
4. Set the **Foreign column / expression** to **ss.store_id**.
5. Check the **Explicit Join** box for the **start_date** and **end_date** rows.
6. For **start_date**, set the operator to **<=** .
7. For **end_date**, set the operator to **>**.
8. For both columns, set the **Foreign column / expression** to **dd.date**.

Once you are done with all that, your query will look like this:

OBSERVE:

```
SELECT

FROM
 stageFactSales ss INNER JOIN  dimDate dd ON(  dd.date = DATE(ss.payment_date) )
 INNER JOIN  dimMovie dm ON(  dm.film_id = ss.film_id )
 INNER JOIN  dimCustomer dc ON(  dc.customer_id = ss.customer_id AND  dc.start_date <= dd.date AND  dc
 INNER JOIN  dimStaff dst ON(  dst.staff_id = ss.staff_id AND  dst.start_date <= dd.date AND  dst.end_
 INNER JOIN  dimStore ds ON(  ds.store_id = ss.store_id AND  ds.start_date <= dd.date AND  ds.end_date
```

**Note** Are you missing a table? Make sure you set the join type to `INNER JOIN`.

The only thing left to do now is to specify our outputs. Before we do that though, let's review the structure of `factSales`. Run the following command against your personal database:

CODE TO TYPE:

```
explain factSales;
```

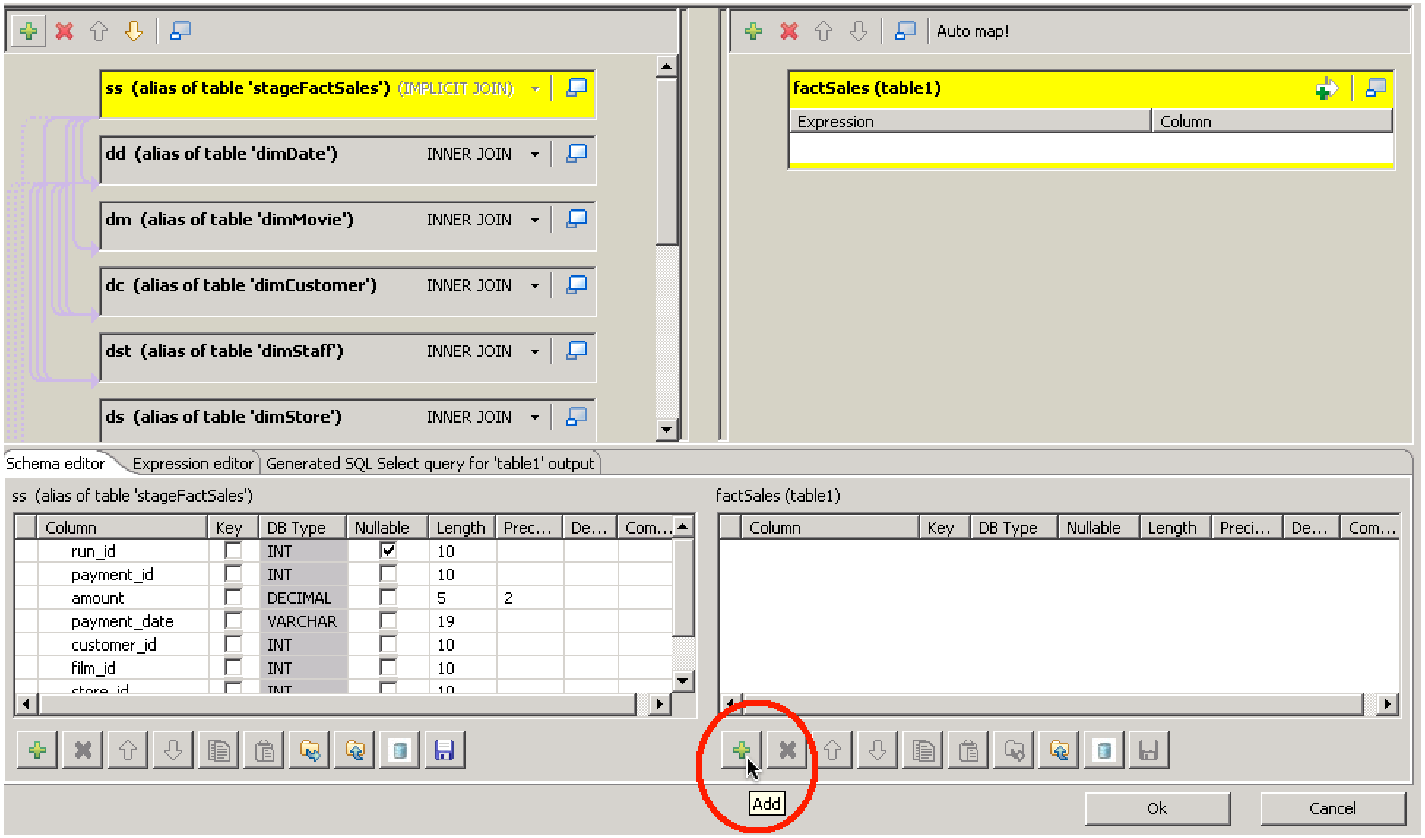
You'll see the following results:

OBSERVE:

```
mysql> explain factSales;
+--------------+--------------+------+-----+---------+----------------+
| Field        | Type         | Null | Key | Default | Extra          |
+--------------+--------------+------+-----+---------+----------------+
| sales_key    | int(11)      | NO   | PRI | NULL    | auto_increment |
| date_key     | int(11)      | NO   |     | NULL    |                |
| customer_key | int(11)      | NO   |     | NULL    |                |
| movie_key    | int(11)      | NO   |     | NULL    |                |
| store_key    | int(11)      | NO   |     | NULL    |                |
| staff_key    | int(11)      | NO   |     | NULL    |                |
| sales_amount | decimal(6,2) | YES  |     | NULL    |                |
| run_id       | int(11)      | NO   |     | NULL    |                |
+--------------+--------------+------+-----+---------+----------------+
8 rows in set (0.10 sec)
```

We need to specify all eight of these columns in **tELTMysqlMap**. The only column that isn't present is our *surrogate key*, `sales_key`.

To add this column, click the button:



ss (alias of table 'stageFactSales') (IMPLICIT JOIN)

dd (alias of table 'dimDate') INNER JOIN

dm (alias of table 'dimMovie') INNER JOIN

dc (alias of table 'dimCustomer') INNER JOIN

dst (alias of table 'dimStaff') INNER JOIN

ds (alias of table 'dimStore') INNER JOIN

factSales (table1)

| Expression | Column |
|---|---|

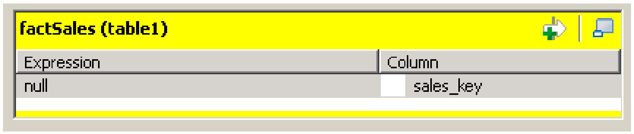
ss (alias of table 'stageFactSales')

| Column | Key | DB Type | Nullable | Length | Prec... | De... | Com... |
|---|---|---|---|---|---|---|---|
| run_id | ☐ | INT | ☑ | 10 | | | |
| payment_id | ☐ | INT | ☐ | 10 | | | |
| amount | ☐ | DECIMAL | ☐ | 5 | 2 | | |
| payment_date | ☐ | VARCHAR | ☐ | 19 | | | |
| customer_id | ☐ | INT | ☐ | 10 | | | |
| film_id | ☐ | INT | ☐ | 10 | | | |
| store_id | ☐ | INT | ☐ | 10 | | | |

factSales (table1)

| Column | Key | DB Type | Nullable | Length | Preci... | De... | Com... |
|---|---|---|---|---|---|---|---|

Name the column `sales_key`, and make sure you keep the **Nullable** box checked. Under the **Expression** column above, type in `NULL`.



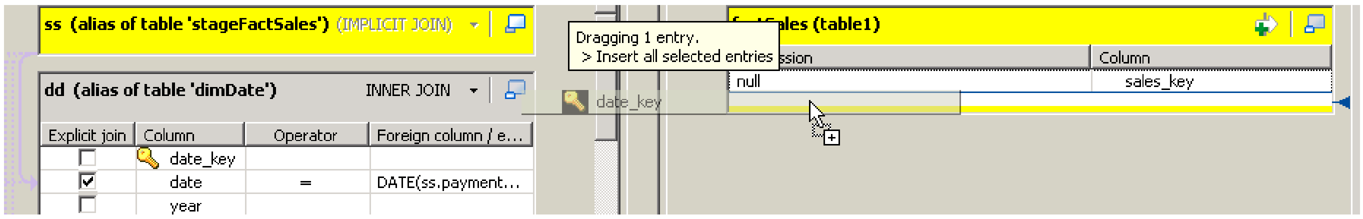
factSales (table1)

| Expression | Column |
|---|---|
| null | sales_key |

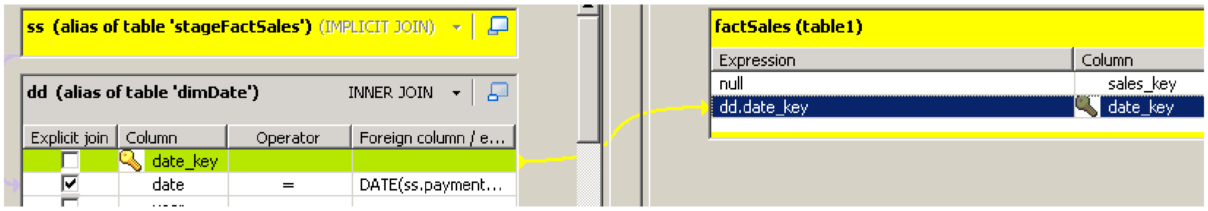
With that out of the way, we can add our other columns.

**Note** One more time: **order matters** for this section, so use care when adding these columns. You can always reorder the columns using the up and down arrow buttons at the bottom of the window.

First, expand the **dimDate**, then drag **date_key** over to the output table, right under **sales_key**.



After you drop the column, the output will look like this:



Repeat this process, **in order**, for the remaining columns:

1. customer_key
2. movie_key
3. store_key
4. staff_key
5. amount
6. run_id

Take a look at the generated query - it should look something like this:

OBSERVE:

```
SELECT
null, dd.date_key , dc.customer_key , dm.movie_key , ds.store_key , dst.staff_key , ss.amount , ss.run
FROM
 stageFactSales ss INNER JOIN  dimDate dd ON(  dd.date = DATE(ss.payment_date) )
 INNER JOIN  dimMovie dm ON(  dm.film_id = ss.film_id )
 INNER JOIN  dimCustomer dc ON(  dc.customer_id = ss.customer_id AND  dc.start_date <= dd.date AND  dc
 INNER JOIN  dimStaff dst ON(  dst.staff_id = ss.staff_id AND  dst.start_date <= dd.date AND  dst.end_
 INNER JOIN  dimStore ds ON(  ds.store_id = ss.store_id AND  ds.start_date <= dd.date AND  ds.end_date
```

We're done with **tELTMysqlMap**, so click "OK" to close the window, and save your work.

There is one thing we should check before we go much further - we should run `EXPLAIN` on the generated query to see how it will run. Run the following command against your personal database:

CODE TO TYPE:

```
EXPLAIN SELECT
null, dd.date_key , dc.customer_key , dm.movie_key , ds.store_key , dst.staff_key , ss.amount , ss.run
FROM
 stageFactSales ss INNER JOIN  dimDate dd ON(  dd.date = DATE(ss.payment_date) )
 INNER JOIN  dimMovie dm ON(  dm.film_id = ss.film_id )
 INNER JOIN  dimCustomer dc ON(  dc.customer_id = ss.customer_id AND  dc.start_date <= dd.date AND  dc
 INNER JOIN  dimStaff dst ON(  dst.staff_id = ss.staff_id AND  dst.start_date <= dd.date AND  dst.end_
 INNER JOIN  dimStore ds ON(  ds.store_id = ss.store_id AND  ds.start_date <= dd.date AND  ds.end_date
```

You'll see the following results:

OBSERVE:

```
mysql> explain SELECT
    -> null, dd.date_key , dc.customer_key , dm.movie_key , ds.store_key , dst.staff_key , ss.amount ,
    -> FROM
    ->  stageFactSales ss INNER JOIN  dimDate dd ON(  dd.date = DATE(ss.payment_date) )
    ->  INNER JOIN  dimMovie dm ON(  dm.film_id = ss.film_id )
    ->  INNER JOIN  dimCustomer dc ON(  dc.customer_id = ss.customer_id AND  dc.start_date <= dd.date
    ->  INNER JOIN  dimStaff dst ON(  dst.staff_id = ss.staff_id AND  dst.start_date <= dd.date AND  d
    ->  INNER JOIN  dimStore ds ON(  ds.store_id = ss.store_id AND  ds.start_date <= dd.date AND  ds.e
+----+-------------+-------+------+-------------------------------+---------+---------+---------------
| id | select_type | table | type | possible_keys                 | key     | key_len | ref
+----+-------------+-------+------+-------------------------------+---------+---------+---------------
|  1 | SIMPLE      | dst   | ALL  | NULL                          | NULL    | NULL    | NULL
|  1 | SIMPLE      | ds    | ALL  | NULL                          | NULL    | NULL    | NULL
|  1 | SIMPLE      | dm    | ALL  | NULL                          | NULL    | NULL    | NULL
|  1 | SIMPLE      | ss    | ref  | customer_id,film_id,store_id  | film_id | 4       | certjosh.dm.fil
|  1 | SIMPLE      | dc    | ALL  | NULL                          | NULL    | NULL    | NULL
|  1 | SIMPLE      | dd    | ALL  | NULL                          | NULL    | NULL    | NULL
+----+-------------+-------+------+-------------------------------+---------+---------+----------------
6 rows in set (0.08 sec)

mysql>
```

This looks pretty bad - our query isn't using any indexes. That's because we never indexed any columns (other than the *primary key*) when we created our dimensions.

Fortunately this problem has a quick fix -- we'll add indexes to most of our columns. We'll omit `start_date` and `end_date` from the indexes for now, just to keep things shorter. Run the following command against your personal database:

CODE TO TYPE:

```
alter table dimCustomer add index(customer_id);
alter table dimDate add index(date);
alter table dimMovie add index(film_id);
alter table dimStaff add index(staff_id);
alter table dimStore add index(store_id);
```

As long as you typed everything correctly you'll see the following results:

OBSERVE:

```
mysql> alter table dimCustomer add index(customer_id);
Query OK, 1178 rows affected (0.08 sec)
Records: 1178  Duplicates: 0  Warnings: 0

mysql> alter table dimDate add index(date);
Query OK, 18628 rows affected (0.15 sec)
Records: 18628  Duplicates: 0  Warnings: 0

mysql> alter table dimMovie add index(film_id);
Query OK, 1000 rows affected (0.09 sec)
Records: 1000  Duplicates: 0  Warnings: 0

mysql> alter table dimStaff add index(staff_id);
Query OK, 2 rows affected (0.09 sec)
Records: 2  Duplicates: 0  Warnings: 0

mysql> alter table dimStore add index(store_id);
Query OK, 2 rows affected (0.05 sec)
Records: 2  Duplicates: 0  Warnings: 0

mysql>
```

Let's try the `EXPLAIN` again. Run the following command against your personal database:

CODE TO TYPE:

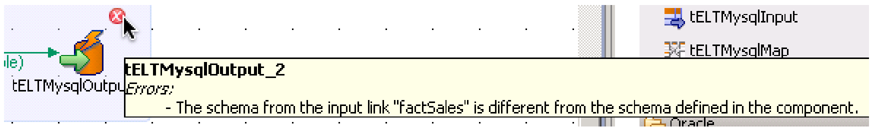
```
EXPLAIN SELECT
null, dd.date_key , dc.customer_key , dm.movie_key , ds.store_key , dst.staff_key , ss.amount , ss.run
FROM
 stageFactSales ss INNER JOIN  dimDate dd ON(  dd.date = DATE(ss.payment_date) )
 INNER JOIN  dimMovie dm ON(  dm.film_id = ss.film_id )
 INNER JOIN  dimCustomer dc ON(  dc.customer_id = ss.customer_id AND  dc.start_date <= dd.date AND  dc
 INNER JOIN  dimStaff dst ON(  dst.staff_id = ss.staff_id AND  dst.start_date <= dd.date AND  dst.end_
 INNER JOIN  dimStore ds ON(  ds.store_id = ss.store_id AND  ds.start_date <= dd.date AND  ds.end_date
```

It looks like our query is using most of the indexes we created. The `dimStaff` and `dimStore` should not be a problem, since there are only two rows in both of those tables.

OBSERVE:

```
mysql> EXPLAIN SELECT
    -> null, dd.date_key , dc.customer_key , dm.movie_key , ds.store_key , dst.staff_key , ss.amount ,
    -> FROM
    ->  stageFactSales ss INNER JOIN  dimDate dd ON(  dd.date = DATE(ss.payment_date) )
    ->  INNER JOIN  dimMovie dm ON(  dm.film_id = ss.film_id )
    ->  INNER JOIN  dimCustomer dc ON(  dc.customer_id = ss.customer_id AND  dc.start_date <= dd.date
    ->  INNER JOIN  dimStaff dst ON(  dst.staff_id = ss.staff_id AND  dst.start_date <= dd.date AND  d
    ->  INNER JOIN  dimStore ds ON(  ds.store_id = ss.store_id AND  ds.start_date <= dd.date AND  ds.e
+----+-------------+-------+------+-------------------------------+-------------+---------+-----------
| id | select_type | table | type | possible_keys                 | key         | key_len | ref
+----+-------------+-------+------+-------------------------------+-------------+---------+-----------
|  1 | SIMPLE      | ss    | ALL  | customer_id,film_id,store_id  | NULL        | NULL    | NULL
|  1 | SIMPLE      | dm    | ref  | film_id                       | film_id     | 2       | certjosh.ss
|  1 | SIMPLE      | dd    | ref  | date                          | date        | 3       | func
|  1 | SIMPLE      | dc    | ref  | customer_id                   | customer_id | 4       | certjosh.ss
|  1 | SIMPLE      | dst   | ALL  | staff_id                      | NULL        | NULL    | NULL
|  1 | SIMPLE      | ds    | ALL  | store_id                      | NULL        | NULL    | NULL
+----+-------------+-------+------+-------------------------------+-------------+---------+-----------
6 rows in set (0.99 sec)
```

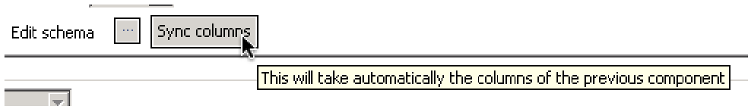
There is one last thing we need to do before we run our job. There is currently an error on **tELTMysqlOutput**:



tELTMysqlOutput_2
*Errors:*
- The schema from the input link "factSales" is different from the schema defined in the component.

**Note** You can ignore the warning on **tELTMysqlMap** -- TOS is just confused by the table aliases.

To fix this error, double-click **tELTMysqlOutput**, then click on **Sync Columns**:



This will take automatically the columns of the previous component

We are now ready to run the job. Do this by clicking on the ▶ at the top of the window.

As long as your components are not in error, you'll see output that looks like this:

OBSERVE:

```
Starting job ProcessDataWarehouse at 18:56 21/12/2008.
Inserting with :
INSERT INTO factSales (SELECT null, dd.date_key , dc.customer_key , dm.movie_key , ds.store_key , dst.

--> 15766 rows inserted.

Job ProcessDataWarehouse ended at 18:56 21/12/2008. [exit code=0]
```

Great job! We're really rolling now. Keep it up and see you shortly!

# Special Facts
# DBA 3: Data Warehousing Lesson 12

Hello! In the last lesson we implemented our **facts**. Our data warehouse is fairly straightforward. We read data from one database, do a few basic transformations, and load the data into the warehouse.

The DVD Rental store only tracks sales. It doesn't have to worry about invoicing customers, shipping products, internet orders, or tracking costs. Other businesses are much more complex. In this lesson, we'll investigate options for dealing with other types of data and situations that occur in data warehouses.

## Missing Keys

In the last lesson we loaded our fact tables, assuming that we could resolve all of the foreign keys required for the dimensions.

But what if a key cannot be found? Let's use our `stageFactCustomerCount` and `factCustomerCount` tables to help us work on this problem . First, run your job to make sure your tables contain data. Take a look at the output (some lines have been omitted):

OBSERVE:

```
Starting job ProcessDataWarehouse at 14:24 23/12/2008.
Inserting with :
INSERT INTO factCustomerCount (SELECT null, dd.date_key , dc.customer_key , ds.store_key , 1, ss.run_i

--> 589 rows inserted.
```

We know that **589** rows were put into `factCustomerCount`. How many were in `stageFactCustomerCount`? Run this command against your personal database:

CODE TO TYPE:

```
select count(*) from stageFactCustomerCount;
```

Oh my! It looks like `stageFactCustomerCount` has **599** rows!

OBSERVE:

```
mysql> select count(*) from stageFactCustomerCount;
+----------+
| count(*) |
+----------+
|      599 |
+----------+
1 row in set (0.05 sec)

mysql>
```

Our join has excluded 10 rows. How do we find the missing rows?

### Debugging tELTMysqlMap

Remember the output from TOS? Take a look:

OBSERVE: Output from TOS

```
Starting job ProcessDataWarehouse at 14:24 23/12/2008.
Inserting with :
INSERT INTO factCustomerCount (SELECT null, dd.date_key , dc.customer_key , ds.store_key , 1, s

--> 589 rows inserted.

Inserting with :
INSERT INTO factSales (SELECT null, dd.date_key , dc.customer_key , IFNULL(dm.movie_key , -1),

--> 15767 rows inserted.

Job ProcessDataWarehouse ended at 14:24 23/12/2008. [exit code=0]
```

The output includes the `INSERT ... SELECT` statement used to populate **factCustomerCount**. If we get rid of the `INSERT` part, and reformat the query slightly, it will look like this:

OBSERVE:

```
SELECT null, dd.date_key , dc.customer_key , ds.store_key , 1, ss.run_id
FROM  stageFactCustomerCount ss INNER JOIN  dimDate dd ON(  dd.date = DATE(ss.create_date) )
INNER JOIN  dimCustomer dc ON(  dc.customer_id = ss.customer_id AND  dc.start_date <= dd.date A
INNER JOIN  dimStore ds ON(  ds.store_id = ss.store_id AND  ds.start_date <= dd.date AND  ds.en
```

Because we used inner joins in our query, non-matching rows are excluded from the results. There is hope after all. =)

We can change this query so it returns only non-matching rows. We do that by converting the **INNER JOIN** to **LEFT JOIN** and adding a **WHERE** clause. We also want to add three columns to the SELECT part, to help us debug: **ss.customer_id, ss.store_id, ss.create_date**. Run this command against your personal database:

CODE TO TYPE:

```
SELECT
ss.customer_id, ss.store_id, ss.create_date,
dd.date_key , dc.customer_key , ds.store_key , ss.run_id
FROM  stageFactCustomerCount ss
INNER JOIN  dimDate dd ON(  dd.date = DATE(ss.create_date) )
LEFT JOIN  dimCustomer dc ON(  dc.customer_id = ss.customer_id AND  dc.start_date <= dd.date AN
LEFT JOIN  dimStore ds ON(  ds.store_id = ss.store_id AND  ds.start_date <= dd.date AND  ds.end
WHERE dd.date_key IS NULL OR dc.customer_key IS NULL OR ds.store_key IS NULL
```

This query will return rows from `stageFactCustomerCount` that do not have corresponding rows in either `dimDate`, `dimCustomer` or `dimStore`. Non-matching rows will have a NULL **date_key**, **customer_key**, or a NULL **store_key**.

After you run the query, you'll see the missing rows:

OBSERVE:

```
mysql> SELECT
    -> ss.customer_id, ss.store_id, ss.create_date,
    -> dd.date_key , dc.customer_key , ds.store_key , ss.run_id
    -> FROM  stageFactCustomerCount ss
    -> INNER JOIN  dimDate dd ON(  dd.date = DATE(ss.create_date) )
    -> LEFT JOIN  dimCustomer dc ON(  dc.customer_id = ss.customer_id AND  dc.start_date <= dd.
    -> LEFT JOIN  dimStore ds ON(  ds.store_id = ss.store_id AND  ds.start_date <= dd.date AND
    -> WHERE dd.date_key IS NULL OR dc.customer_key IS NULL OR ds.store_key IS NULL;
+-------------+----------+---------------------+----------+--------------+-----------+--------
| customer_id | store_id | create_date         | date_key | customer_key | store_key | run_id |
+-------------+----------+---------------------+----------+--------------+-----------+--------
|           1 |        1 | 2004-06-26 00:00:00 | 20040626 |         NULL |         1 |     76 |
|           2 |        1 | 2004-10-12 00:00:00 | 20041012 |         NULL |         1 |     76 |
|           3 |        1 | 2004-06-12 00:00:00 | 20040612 |         NULL |         1 |     76 |
|           4 |        2 | 2004-11-22 00:00:00 | 20041122 |         NULL |         2 |     76 |
|           5 |        1 | 2004-02-17 00:00:00 | 20040217 |         NULL |         1 |     76 |
|           6 |        2 | 2004-12-18 00:00:00 | 20041218 |         NULL |         2 |     76 |
|           7 |        1 | 2004-06-09 00:00:00 | 20040609 |         NULL |         1 |     76 |
|           8 |        2 | 2004-04-18 00:00:00 | 20040418 |         NULL |         2 |     76 |
|           9 |        2 | 2004-02-29 00:00:00 | 20040229 |         NULL |         2 |     76 |
|          10 |        1 | 2004-12-03 00:00:00 | 20041203 |         NULL |         1 |     76 |
+-------------+----------+---------------------+----------+--------------+-----------+--------
10 rows in set (0.10 sec)
```

It looks like we found our missing rows: **customer_id** 1 through 10. If you recall, we specifically excluded these customers from our **dimCustomer** dimension, because our business users told us they were "test" accounts.

To fix this problem, we'll alter our select statement. Edit the query on the **tMysqlInput** component for **stageCustomerCount**. Change the query so it looks like this (make sure to use the correct quotation marks):

CODE TO TYPE:

```
select customer_id, store_id, create_date
from customer WHERE customer_id > 10;
```

## Handling Missing Keys

We process dimensions before we process facts so we can be sure that our dimensions are current and then our facts can load correctly. If we load our sales facts in our data warehouse, but one row doesn't have a movie associated with it, we have several options to deal with the missing movie:

1. Ignore the row - don't load it.
2. Stop the current process, alerting someone to the error.
3. Load the row, but set the row to a special "!!!! MISSING MOVIE !!!!" entry in the `dimMovie` dimension.

Logically, your business users will have the last word on handling this situation. Most will not choose pick options **#1 or #2** for these reasons:

1. Ignoring the row can cause daily totals to be off, so the warehouse reports may not match existing reports from the source systems.
2. Stopping the current process to alert users to the error causes the whole process to be interrupted, and will cause the entire warehouse to be inaccessible.

Since our fact tables are reloaded on a daily basis, option **#3** is attractive because the row in error can be fixed within the source system and the warehouse will be "fixed" on the next run.

Let's implement this fix. We'll start my altering dimMovie so film_id is a normal integer. Currently it is `unsigned`, meaning that no negative values are allowed. Run this command against your personal database:

CODE TO TYPE:

```
ALTER TABLE dimMovie modify film_id int not null;
```

Next we will add the special missing row to `dimMovie`. Run this command against your personal database:

CODE TO TYPE:

```
insert into dimMovie (movie_key, film_id, title, run_id ) values (-1, -1, '!!! MISSING MOVIE !!
```

Let's check the row we just inserted to see what it looks like. Run this command against your personal database:

CODE TO TYPE:

```
select * from dimMovie where movie_key=-1;
```

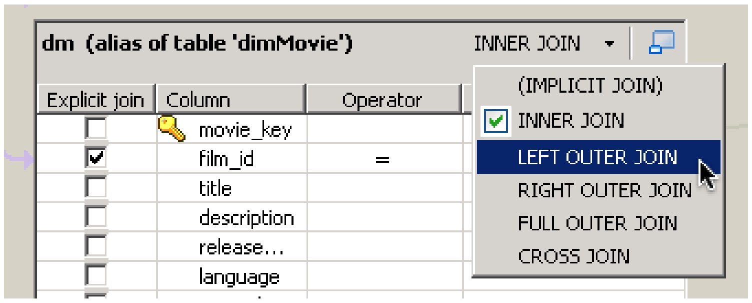
As long as you typed everything correctly you'll see this:

OBSERVE:

```
select * from dimMovie where movie_key=-1;
+-----------+---------+----------------------+-------------+--------------+----------+--------
| movie_key | film_id | title                | description | release_year | language | origina
+-----------+---------+----------------------+-------------+--------------+----------+--------
|        -1 |      -1 | !!! MISSING MOVIE !!! | NULL        |         NULL |          | NULL
+-----------+---------+----------------------+-------------+--------------+----------+--------
1 row in set (0.05 sec)

mysql>
```

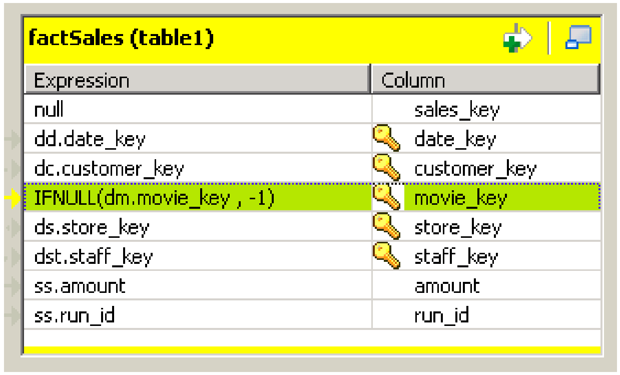
Everything looks good. Next, open your job in TOS, and double click on the **tELTMysqlMap** component for **factSales**.

Change the join type on **dm** (dimMovie) from `INNER JOIN` to `LEFT OUTER JOIN`:



We can use MySQL's IFNULL function to be sure our join was successful. If the join failed, `dm.movie_key` will be null, so the `IFNULL` function will return **-1**. If the join worked, `dm.movie_key` will have a value, which will be returned by `IFNULL`.

Next, change the expression on the **factSales** output from `dm.movie_key` to `IFNULL(dm.movie_key, -1)`:


factSales (table1)

| Expression | Column |
|---|---|
| null | sales_key |
| dd.date_key | date_key |
| dc.customer_key | customer_key |
| IFNULL(dm.movie_key , -1) | movie_key |
| ds.store_key | store_key |
| dst.staff_key | staff_key |
| ss.amount | amount |
| ss.run_id | run_id |

Click OK, then save your job.

We don't have access to our source system, so we cannot insert test data into sakila. Instead, we can temporarily modify our stored procedure to insert test data into `stageFactSales`. Run this command against your personal database:

CODE TO TYPE:

```
DROP PROCEDURE etl_preFactSales;
DELIMITER //
CREATE PROCEDURE etl_preFactSales ()
BEGIN
ALTER TABLE stageFactSales add index(payment_id);
ALTER TABLE stageFactSales add index(payment_date);
ALTER TABLE stageFactSales add index(customer_id);
ALTER TABLE stageFactSales add index(film_id);
ALTER TABLE stageFactSales add index(store_id);
TRUNCATE TABLE factSales;
INSERT INTO stageFactSales values (-1, -1, '999.99', '2005-01-01', 11, 99999, 1, 1);
END
//
```

After you make that change, run your job. You'll see the familiar output:

OBSERVE:

```
Starting job ProcessDataWarehouse at 14:24 23/12/2008.
Inserting with :
INSERT INTO factCustomerCount (SELECT null, dd.date_key , dc.customer_key , ds.store_key , 1, s

--> 589 rows inserted.

Inserting with :
INSERT INTO factSales (SELECT null, dd.date_key , dc.customer_key , IFNULL(dm.movie_key , -1),

--> 15767 rows inserted.

Job ProcessDataWarehouse ended at 14:24 23/12/2008. [exit code=0]
```

Switch back to MySql mode to see if your change worked. Run this command against your personal database:

CODE TO TYPE:

```
select * from factSales where movie_key=-1;
```

If your job ran successfully, you'll see this:

OBSERVE:

```
mysql> select * from factSales where movie_key=-1;
+-----------+----------+--------------+-----------+-----------+-----------+--------------+----
| sales_key | date_key | customer_key | movie_key | store_key | staff_key | sales_amount | run_
+-----------+----------+--------------+-----------+-----------+-----------+--------------+----
```

```
|      6858 | 20050101 |           884 |        -1 |          1 |          1 |        999.99 |
+-----------+----------+--------------+-----------+-----------+-----------+--------------+-----
1 row in set (0.08 sec)

mysql>
```

It looks like your left join saved the day!

## Aggregating

Data warehouses are built with the presumption that data needs to be aggregated in different ways. If the increment we are using in our data warehouse is one day, and we query the warehouse to see *sales in May*, we need to `SUM(Sales)` for each day in May.

This works for most types of facts, but what if the fact under consideration is an account balance? Account balances are usually stored as **point in time** values. Take a look:

| Date | Description | Payment | Deposit | Balance |
|---|---|---|---|---|
| 01/01 | Starting Balance | | | **592.20** |
| 01/02 | Grocery Store | 25.90 | | **566.30** |
| 01/03 | Computer Store | 19.50 | | **546.80** |
| 01/04 | Consulting Work | | 1500.00 | **2046.80** |

The account balance on **01/03** is **546.80**, and the account balance on **01/04** is **2046.80**. If today is January 4th, the account balance for the month of January is **2046.80**, not 592.20 + 566.30 + 546.80 + 2046.80 = 3752.10. Likewise, the account balance for **2008** is also **2046.80**, not 3752.10.

The proper aggregate for an account balance is not `SUM`, it is `LAST`.

If we take a look at MySQL's group by functions you'll notice there is `MAX`, `MIN`, and of course `SUM`, however there is no `LAST` or `FIRST`. This is because `LAST` and `FIRST` are not currently supported by MySQL.

Getting around this problem is tricky with MySQL. Our only option is to use **ORDER BY** to sort the results by date, so the oldest record will appear first, and to **LIMIT** our result to one row. A sample query to get the last value from `factSales` would look like this:

CODE TO TYPE:

```
SELECT *
FROM factSales
ORDER BY date_key DESC
LIMIT 0, 1;
```

MySQL would return the last row:

OBSERVE:

```
mysql> SELECT *
FROM factSales
ORDER BY date_key DESC
LIMIT 0, 1;
+-----------+----------+--------------+-----------+-----------+-----------+--------------+--------+
| sales_key | date_key | customer_key | movie_key | store_key | staff_key | sales_amount | run_id |
+-----------+----------+--------------+-----------+-----------+-----------+--------------+--------+
|         7 | 20060214 |          591 |       267 |         1 |         1 |         4.99 |     76 |
+-----------+----------+--------------+-----------+-----------+-----------+--------------+--------+
1 row in set (0.06 sec)

mysql>
```

Other databases have extensions that make this type of query easier.

## Deaggregating Data

We already saw that aggregating certain types of data can pose some problems. In some situations, data may already be aggregated, causing a different type of problem.

Suppose the DVD store started shipping packages. Shippers usually want to know the weight of packages, since it is used to calculate shipping cost. If someone orders four DVDs at the same time, those four DVDs are combined into a single package and sent to the customer. Their order may look something like this:

| Title | Price |
|---|---|
| DADDY PITTSBURGH | 9.99 |
| TITANIC BOONDOCK | 5.99 |
| NEWTON LABYRINTH | 5.99 |
| APOLLO TEEN | 9.99 |
| Shipping & Handling | 5.90 |
| **== Total ==** | **37.86** |

Our business user wants to know, for example, how much shipping costs were for the movie APOLLO TEEN? Looking at our data, we only know that it cost **$5.90** to ship APOLLO TEEN along with DADDY PITTSBURG, TITANIC BOONDOCK and NEWTON LABYRINTH.

Our business user uses this formula to calculate shipping costs:

**Shipping on an item = Shipping & Handling / # of Items**

With this formula in mind, we can calculate the shipping & handling on each individual item in the order:

| Title | Price | Shipping |
|---|---|---|
| DADDY PITTSBURGH | 9.99 | **5.90/4 = 1.475** |
| TITANIC BOONDOCK | 5.99 | **5.90/4 = 1.475** |
| NEWTON LABYRINTH | 5.99 | **5.90/4 = 1.475** |
| APOLLO TEEN | 9.99 | **5.90/4 = 1.475** |
| Shipping & Handling | 5.90 | 1.475 * 4 = 5.90 |
| **== Total ==** | **37.86** | |

So now you might ask yourself, "*How can shipping be 1.475? Shouldn't it be rounded?*"

The answer to that question (like the shipping calculation itself) can only be answered by your business users. Possible solutions might be to *do nothing* - meaning let the end users pick if and how they want to round the data, or to implement and document a rounding algorithm.

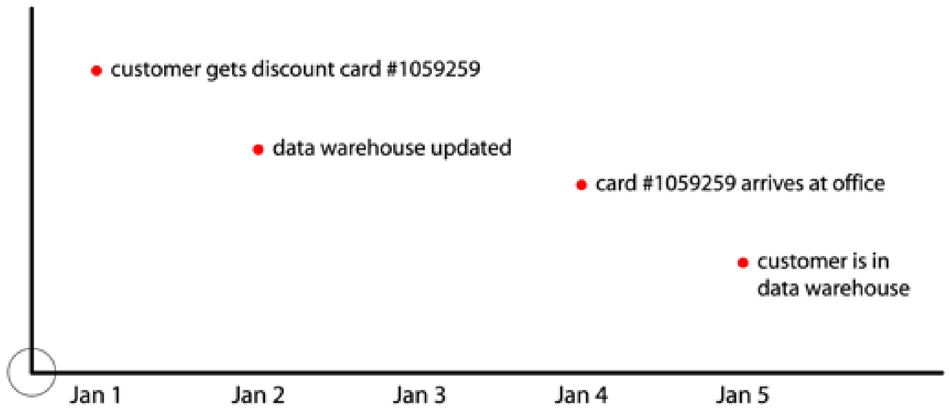
So where and how would you implement this deaggregation? You would have to add a **T**ransformation step to your fact processing to calculate the shipping & handling value.

## Early Arriving Facts

The next type of problem you may encounter in your data warehouse is **Early Arriving Facts**, This is also known as **Late Arriving Dimensions**.

Here's a sample scenario of such a dilemma: On January 1st, a new customer gets in line to check out at the grocery store. She decides to sign up for the store's discount card. She fills out the paperwork and is allowed to use the card (card #1059259) for her purchase. On January 4th, the card finally arrives at the corporate office, where the information is entered into the database. The card number is unique to a customer, so it forms the *primary key* in the database.

A time line of the events would look like this:


customer gets discount card #1059259
data warehouse updated
card #1059259 arrives at office
customer is in data warehouse
Jan 1
Jan 2
Jan 3
Jan 4
Jan 5

The customer's purchases are entered into the data warehouse on the morning of January 2nd, however no details exist in the customer dimension (under card #1059259 ) until January 5th.

This type of situation is slightly different than **Missing Keys**, because the keys are not exactly missing, they're just *late*.

How do you deal with this type of situation? First, you may have to alter the rules on `dimCustomer` to allow NULLs in most columns. The only data we know about "late" customers is the card number (because it is the primary key).

Next, change your fact process like so:

1. Go through every fact row, check to see if the customer exists in `dimCustomer` (perhaps using the `LEFT JOIN/IS NULL` techniques from earlier in this lesson).
2. For every missing dimension row, insert a new row into the dimension, possibly using default values such as "PENDING ACCOUNT" for first and last name.
3. Once we are certain the corresponding dimension exists, add the fact row to the table.

**Note** We assume that `dimCustomer` has already been processed and is up-to-date. If your dimension isn't current, then a lot of fact data is going to appear to be late!

After January 1st, dimCustomer would look something like this:

| Card # | First Name | Last Name | Address | Phone | Start Date | End Date |
|---|---|---|---|---|---|---|
| 1059259 | PENDING | ACCOUNT | *NULL* | *NULL* | 01-Jan | 31-DEC-2099 |

So, what happens on January 5th, when the dimension data finally catches up to the fact data?

**Nothing.**

Our existing dimension process will see the customer's details, and update the dimension accordingly. After January 5th, there will be two rows in the database for card #1059259:

| Card # | First Name | Last Name | Address | Phone | Start Date | End Date |
|---|---|---|---|---|---|---|
| 1059259 | PENDING | ACCOUNT | *NULL* | *NULL* | 01-Jan | 05-Jan |
| 1059259 | Sue | Shopper | *519 Wells St.* | *555-2592* | 05-Jan | 31-DEC-2099 |

This reflects history, exactly as it happened. Between January 1st and January 5th we didn't know the details for the customer with card #1059259, and after January 5th we did.

We covered a lot of information in this lesson! Good job. In the next lesson we'll examine some common queries that people run against data warehouses. See you there!

# Querying a Relational Data Warehouse
# DBA 3: Data Warehousing Lesson 13

We've spent most of our time implementing our data warehouse, but we still haven't really seen the fruits of our labor. In this lesson we'll examine basic SQL queries you can use to unlock information stored in the warehouse.

**Note** In the next course, DBA 4, you'll learn all about using **MDX** - **M**ultiple **D**imension **E**xpressions - to query data warehouses, so stay tuned!

## Viewing Data

In the first lesson, we discussed the goals of a data warehouse. We want to create:

- a separate system that won't interrupt business critical operational systems.
- a single point of access for all analytical queries.
- a unified, consistent view of underlying data (even data from external systems).
- a straightforward way to analyze trends (to see the way sales compare from month to month).

Our current structure of **dimensions** and **facts** is consistent and straightforward, but we can do better. Columns like `run_id` are not important to end users, and may even be confusing to them. Tables like `etlRuns` don't matter to anyone except database administrators, so they should be hidden from end users.

Ease of use aside, views also provide other benefits:

- Some information in the data warehouse may be very sensitive, so views can be used to provide row level security.
- Views can be used to provide consistency to the data warehouse, especially as underlying tables grow in complexity and undergo changes.
- Columns can be renamed to make things easier to understand.

For our views, we will:

- name them with a common prefix - **Fact_** for fact tables, and **Dimension_** for dimension tables.
- omit surrogate keys from fact tables (such as `sales_key`).
- omit `start_date` and `end_date` from Type-2 slowly changing dimensions.
- omit `run_id` from all tables.
- keep foreign key columns (the `_key` columns) unchanged.
- keep keys from source systems (like `customer_id`) unchanged.
- rename fact and dimension columns to more readable equivalents (for example, `customer_count` would become `Customer Count`).

Let's get started and create a view for our `factCustomerCount` table. In this view we will omit the `customerCount_key` and `run_id`. Switch to MySQL mode, and run this command against your personal database::

CODE TO TYPE:

```
CREATE VIEW Fact_CustomerCount
AS
SELECT date_key, customer_key, store_key, customer_count as `Customer Count`
FROM factCustomerCount;
```

If you typed everything correctly, you'll see these results:

OBSERVE:

```
mysql> CREATE VIEW Fact_CustomerCount
    -> AS
    -> SELECT date_key, customer_key, store_key, customer_count as "Customer Count"
    -> FROM factCustomerCount;
Query OK, 0 rows affected (0.06 sec)

mysql>
```

It all looks good so far. To test out this query, let's check out the first ten rows. Run this command against your personal database:

CODE TO TYPE:

```
SELECT * from Fact_CustomerCount
LIMIT 0, 10;
```

This is easier for end users to understand:

OBSERVE:

```
mysql> SELECT * from Fact_CustomerCount
    -> LIMIT 0, 10;
+----------+--------------+-----------+----------------+
| date_key | customer_key | store_key | Customer Count |
+----------+--------------+-----------+----------------+
| 20040319 |          590 |         1 |              1 |
| 20041007 |          591 |         1 |              1 |
| 20040811 |          593 |         1 |              1 |
| 20040124 |          595 |         1 |              1 |
| 20040531 |          596 |         1 |              1 |
| 20040115 |          599 |         1 |              1 |
| 20040205 |          600 |         1 |              1 |
| 20040921 |          602 |         1 |              1 |
| 20040427 |          604 |         1 |              1 |
| 20041114 |          605 |         1 |              1 |
+----------+--------------+-----------+----------------+
10 rows in set (0.08 sec)
```

Next, let's create a view for `factSales`. In this view, we will omit the `sales_key` and `run_id` columns. Run this command against your personal database:

CODE TO TYPE:

```
CREATE VIEW Fact_Sales
AS
SELECT date_key, customer_key, movie_key, store_key, staff_key, sales_amount as `Sales Amount`
FROM factSales;
```

If you typed everything correctly, you'll see this familiar MySQL result: `Query OK, 0 rows affected (0.13 sec).`

With a couple of **facts** out of the way, let's turn our attention to **dimensions**. Start with `dimCustomer`. Run this command against your personal database:

CODE TO TYPE:

```
CREATE VIEW Dimension_Customer
AS
SELECT customer_key, customer_id, first_name as `First Name`, last_name as `Last Name`,
Email, Address, address2 as "Address 2",
District, City, Country, postal_code as `Postal Code`,
Phone, Active, create_date as `Create Date`
FROM dimCustomer;
```

Great! Let's move on to `dimDate`. Run this command against your personal database:

CODE TO TYPE:

```
CREATE VIEW Dimension_Date
AS
SELECT date_key, Date, Year, Quarter, Month, month_name as `Month Name`, Day,
day_name as `Day of Week`, week as `Week In Year`,
is_weekend as `Is Weekend`, is_holiday as `Is Holiday`
FROM dimDate;
```

The next view we'll create is for `dimMovie`. Run this command against your personal database:

CODE TO TYPE:

```
CREATE VIEW Dimension_Movie
AS
```

```
SELECT movie_key, film_id, Title, Description, release_year as `Release Year`,
Language, original_language as `Original Language`,
rental_duration as `Rental Duration`, Length,
Rating, special_features as `Special Features`
FROM dimMovie;
```

The last view we'll create for now is for the `dimStore` table. Run this command against your personal database:

CODE TO TYPE:

```
CREATE VIEW Dimension_Store
AS
SELECT store_key, store_id, Address, address2 as `Address 2`, District,
City, Country, postal_code as `Postal Code`, Region,
manager_first_name as `Manger First Name`,
manager_last_name as `Manager Last Name`
FROM dimStore;
```

Great! We're ready to get started with our queries!

## Answering Questions

We'll use a standard template for querying the data warehouse. In the text below, **blue** signifies **facts**, and **red** signifies **dimensions**. It isn't necessary to use all parts of our template, especially if we aren't interested in limiting our query using a `WHERE` clause.

OBSERVE:

```
SELECT columns from dimension tables,
SUM( fact columns )
FROM fact view
INNER JOIN dimension view 1 on (fact column = dimension column )
INNER JOIN dimension view 2 on (fact column = dimension column )
.
.
WHERE Limits to dimensions
limits to facts
GROUP BY dimension columns
ORDER BY dimension columns, fact columns
LIMIT 0, 5 (optional "top 5" results)
```

In lesson two we discussed questions that would be posed by management. We rewrote these questions so that they were in the format of **facts** and **dimensions**. Now let's try to answer some of them!

First up: **How many new customers did we add by quarter**?

To answer this question, we'll need data from the **Fact_CustomerCount** and **Dimension_Date** tables. Let's write a query using the template we already created. Although our question does not specify a particular sorting order, we'll sort by **Quarter**. Run this command against your personal database:

CODE TO TYPE:

```
SELECT dd.Quarter,
SUM( `Customer Count` ) as `Customer Count`
FROM Fact_CustomerCount fc
INNER JOIN Dimension_Date dd on (fc.date_key = dd.date_key)
GROUP BY dd.Quarter
ORDER BY dd.Quarter;
```

MySQL will reply with your answer:

OBSERVE:

```
mysql> SELECT dd.Quarter,
    -> SUM( `Customer Count` ) as `Customer Count`
    -> FROM Fact_CustomerCount fc
    -> INNER JOIN Dimension_Date dd on (fc.date_key = dd.date_key)
    -> GROUP BY dd.Quarter
    -> ORDER BY dd.Quarter;
+---------+----------------+
| Quarter | Customer Count |
+---------+----------------+
| Q1      |            158 |
| Q2      |            140 |
| Q3      |            143 |
| Q4      |            148 |
+---------+----------------+
4 rows in set (0.06 sec)

mysql>
```

That's pretty slick! We didn't even have to figure out the specific quarter each customer registered.

**Note** If you see an error that looks like this: **ERROR 1305 (42000): FUNCTION certjosh.SUM does not exist**, make sure you do not have any spaces between `SUM` and `(`. `SUM(column)`. This works in MySQL, but `SUM (column)` will return an error.

Now suppose we want to extend this query, so it answers these questions: **How many new customers did we add by quarter** and **by month**? Let's go back to our template, and add a new column. Run this command against your personal database:

CODE TO TYPE:

```
SELECT dd.Quarter, dd.`Month Name`,
SUM( `Customer Count` ) as `Customer Count`
FROM Fact_CustomerCount fc
INNER JOIN Dimension_Date dd on (fc.date_key = dd.date_key)
GROUP BY dd.Quarter, dd.`Month Name`
ORDER BY dd.Quarter, dd.`Month Name`;
```

It looks like the database gave us exactly what we asked for, even if it isn't exactly what we wanted:

OBSERVE:

```
mysql> SELECT dd.Quarter, dd.`Month Name`,
    -> SUM( `Customer Count` ) as `Customer Count`
    -> FROM Fact_CustomerCount fc
    -> INNER JOIN Dimension_Date dd on (fc.date_key = dd.date_key)
    -> GROUP BY dd.Quarter, dd.`Month Name`
    -> ORDER BY dd.Quarter, dd.`Month Name`;
+---------+------------+----------------+
| Quarter | Month Name | Customer Count |
+---------+------------+----------------+
| Q1      | February   |             47 |
| Q1      | January    |             54 |
| Q1      | March      |             57 |
| Q2      | April      |             44 |
| Q2      | June       |             51 |
| Q2      | May        |             45 |
| Q3      | August     |             51 |
| Q3      | July       |             46 |
| Q3      | September  |             46 |
| Q4      | December   |             49 |
| Q4      | November   |             51 |
| Q4      | October    |             48 |
+---------+------------+----------------+
12 rows in set (0.06 sec)
```

Let's try ordering the months by *number* instead of by *name*, so January would occur before February in our results. Run this command against your personal database:

CODE TO TYPE:

```
SELECT dd.Quarter, dd.`Month Name`,
SUM( `Customer Count` ) as `Customer Count`
FROM Fact_CustomerCount fc
INNER JOIN Dimension_Date dd on (fc.date_key = dd.date_key)
GROUP BY dd.Quarter, dd.`Month Name`
ORDER BY dd.Quarter, dd.`Month`;
```

Now that's more like it!

OBSERVE:

```
mysql> SELECT dd.Quarter, dd.`Month Name`,
    -> SUM( `Customer Count` ) as `Customer Count`
    -> FROM Fact_CustomerCount fc
    -> INNER JOIN Dimension_Date dd on (fc.date_key = dd.date_key)
    -> GROUP BY dd.Quarter, dd.`Month Name`
    -> ORDER BY dd.Quarter, dd.`Month`;
+---------+------------+----------------+
| Quarter | Month Name | Customer Count |
+---------+------------+----------------+
| Q1      | January    |             54 |
| Q1      | February   |             47 |
| Q1      | March      |             57 |
| Q2      | April      |             44 |
| Q2      | May        |             45 |
| Q2      | June       |             51 |
| Q3      | July       |             46 |
| Q3      | August     |             51 |
| Q3      | September  |             46 |
| Q4      | October    |             48 |
| Q4      | November   |             51 |
| Q4      | December   |             49 |
+---------+------------+----------------+
```

Okay, let's move on to a new question: **What was the amount of sales revenue we earned**, **by store and by by month**? To answer these questions, we'll need to use the **Fact_Sales**, **Dimension_Store**, and **Dimension_Date** views. This time we will try an alternate **ORDER BY** syntax; we'll specify the columns by *position* instead of by name. For this query, 1 is the first column, **ds.Address**, and 2 is the second column, **ds.`Month Name`**. Run this command against your personal database:

CODE TO TYPE:

```
SELECT ds.Address, dd.`Month Name`,
SUM( `Sales Amount` ) as `Sales Amount`
FROM Fact_Sales fs
INNER JOIN Dimension_Store ds on (fs.store_key = ds.store_key)
INNER JOIN Dimension_Date dd on (fs.date_key = dd.date_key )
GROUP BY 1, 2
ORDER BY ds.Address, dd.`Month`;
```

Once again, our warehouse answers our questions right away:

OBSERVE:

```
mysql> SELECT ds.Address, dd.`Month Name`,
    -> SUM( `Sales Amount` ) as `Sales Amount`
    -> FROM Fact_Sales fs
    -> INNER JOIN Dimension_Store ds on (fs.store_key = ds.store_key)
    -> INNER JOIN Dimension_Date dd on (fs.date_key = dd.date_key )
    -> GROUP BY ds.Address, dd.`Month Name`
    -> ORDER BY 1, 2;
+--------------------+------------+--------------+
| Address            | Month Name | Sales Amount |
+--------------------+------------+--------------+
| 28 MySQL Boulevard | February   |       270.09 |
| 28 MySQL Boulevard | May        |      2328.30 |
| 28 MySQL Boulevard | June       |      4829.30 |
| 28 MySQL Boulevard | July       |     13873.70 |
| 28 MySQL Boulevard | August     |     11910.70 |
| 47 MySakila Drive  | January    |       999.99 |
| 47 MySakila Drive  | February   |       238.11 |
| 47 MySakila Drive  | May        |      2418.35 |
| 47 MySakila Drive  | June       |      4640.01 |
| 47 MySakila Drive  | July       |     14020.33 |
| 47 MySakila Drive  | August     |     11740.45 |
+--------------------+------------+--------------+
11 rows in set (0.70 sec)
```

**Note** The *sakila* database is a random set of data. That's the reason there were no sales for "47 MySakila Drive" in March.

Now suppose we want to find out the **top five sales**, by store and by month. Let's give it a try! Run this command against your personal database:

CODE TO TYPE:

```
SELECT ds.Address, dd.`Month Name`,
SUM( `Sales Amount` ) as `Sales Amount`
FROM Fact_Sales fs
INNER JOIN Dimension_Store ds on (fs.store_key = ds.store_key)
INNER JOIN Dimension_Date dd on (fs.date_key = dd.date_key )
GROUP BY ds.Address, dd.`Month Name`
ORDER BY 1, 2
LIMIT 0, 5;
```

Once again, MySQL answered our questions, but it isn't exactly the information we want:

OBSERVE:

```
mysql> SELECT ds.Address, dd.`Month Name`,
    -> SUM( `Sales Amount` ) as `Sales Amount`
    -> FROM Fact_Sales fs
    -> INNER JOIN Dimension_Store ds on (fs.store_key = ds.store_key)
    -> INNER JOIN Dimension_Date dd on (fs.date_key = dd.date_key )
    -> GROUP BY ds.Address, dd.`Month Name`
    -> ORDER BY 1, 2
    -> LIMIT 0, 5;
+--------------------+------------+--------------+
| Address            | Month Name | Sales Amount |
+--------------------+------------+--------------+
| 28 MySQL Boulevard | August     |     11910.70 |
| 28 MySQL Boulevard | February   |       270.09 |
| 28 MySQL Boulevard | July       |     13873.70 |
| 28 MySQL Boulevard | June       |      4829.30 |
| 28 MySQL Boulevard | May        |      2328.30 |
+--------------------+------------+--------------+
5 rows in set (0.39 sec)
```

We retrieved the top five results, but we didn't order by **Sales Amount**, and then in descending order from there. Run this command against your personal database:

CODE TO TYPE:

```
SELECT ds.Address, dd.`Month Name`,
SUM( `Sales Amount` ) as `Sales Amount`
FROM Fact_Sales fs
INNER JOIN Dimension_Store ds on (fs.store_key = ds.store_key)
INNER JOIN Dimension_Date dd on (fs.date_key = dd.date_key )
GROUP BY ds.Address, dd.`Month Name`
ORDER BY 3 DESC, 1, 2
LIMIT 0, 5;
```

The results look much better now:

OBSERVE:

```
mysql> SELECT ds.Address, dd.`Month Name`,
    -> SUM( `Sales Amount` ) as `Sales Amount`
    -> FROM Fact_Sales fs
    -> INNER JOIN Dimension_Store ds on (fs.store_key = ds.store_key)
    -> INNER JOIN Dimension_Date dd on (fs.date_key = dd.date_key )
    -> GROUP BY ds.Address, dd.`Month Name`
    -> ORDER BY 3 DESC, 1, 2
    -> LIMIT 0, 5;
+--------------------+------------+--------------+
| Address            | Month Name | Sales Amount |
+--------------------+------------+--------------+
| 47 MySakila Drive  | July       |     14020.33 |
| 28 MySQL Boulevard | July       |     13873.70 |
| 28 MySQL Boulevard | August     |     11910.70 |
| 47 MySakila Drive  | August     |     11740.45 |
| 28 MySQL Boulevard | June       |      4829.30 |
+--------------------+------------+--------------+
5 rows in set (0.38 sec)
```

## Problems with Queries

So far we've written several queries against our data warehouse. The biggest problem we've run into so far was that some results were not ordered correctly. So what else could go wrong?

## Bad Joins

There is another more serious problem that may take place in our data warehouse - **bad joins**.

Suppose you come into the office one day, and are asked to answer a question we've seen many times before: **How many new customers did we add by quarter**? Let's approach this question again. Run this command against your personal database:

CODE TO TYPE:

```
SELECT dd.Quarter,
SUM( `Customer Count` ) as `Customer Count`
FROM Fact_CustomerCount fc
INNER JOIN Dimension_Date dd on (fc.date_key = fc.date_key)
GROUP BY dd.Quarter
ORDER BY dd.Quarter;
```

You probably noticed right away that something was strange. The query takes a very long time to return results, and when it finally does, they look really strange:

OBSERVE:

```
mysql> SELECT dd.Quarter,
    -> SUM( `Customer Count` ) as `Customer Count`
    -> FROM Fact_CustomerCount fc
    -> INNER JOIN Dimension_Date dd on (fc.date_key = fc.date_key)
    -> GROUP BY dd.Quarter
    -> ORDER BY dd.Quarter;
+---------+----------------+
| Quarter | Customer Count |
+---------+----------------+
| Q1      |        2711167 |
| Q2      |        2733549 |
| Q3      |        2763588 |
| Q4      |        2763588 |
+---------+----------------+
4 rows in set (14.04 sec)
```

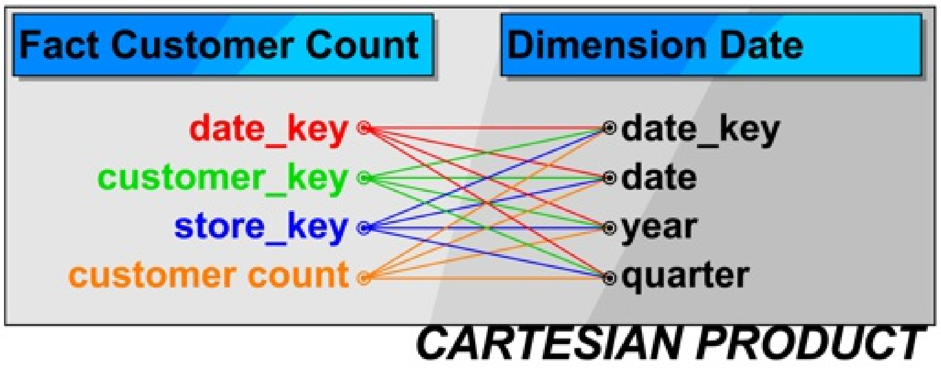
Compare these results to the results we calculated previously:

OBSERVE:

```
+---------+----------------+
| Quarter | Customer Count |
+---------+----------------+
| Q1      |            158 |
| Q2      |            140 |
| Q3      |            143 |
| Q4      |            148 |
+---------+----------------+
```

So what happened here? It was a **bad join**. Instead of writing `(fc.date_key = fc.date_key)` for our join criteria, we should have written `(fc.date_key = dd.date_key)`.

Back in our query we forgot to specify how **Fact_CustomerCount** joins to **Dimension_Date**. This caused MySQL to return the *cartesian product* of those two tables instead of the properly joined results.



The real danger behind **bad joins** is that they can often go undetected. This example is extreme - our business users would likely know there is a problem with the query, since the results for Q1 are over 17,000 times the actual value. That is pretty far off! But what if our company typically added 3,000,000 new customers in a quarter? Then 2,711,167 wouldn't seem so far off at all.

The best way to prevent **bad joins** is to have many different people review each query written against the data warehouse. No query tool can tell you if your join is bad, or if your query is otherwise written incorrectly.

## Incorrect Filtering

Now suppose your boss wants to know which movies had sales greater than $10.00. You sit down at your desk, and write a quick query to find the answer the question. Run this command against your personal database:

CODE TO TYPE:

```
SELECT dm.Title,
SUM( `Sales Amount` ) as `Sales Amount`
FROM Fact_Sales fs
INNER JOIN Dimension_Movie dm on (fs.movie_key = dm.movie_key)
WHERE fs.`Sales Amount` > 10
GROUP BY 1;
```

It looks like fifty movies have had sales greater $10.00:

OBSERVE:

```
mysql> SELECT dm.Title,
    -> SUM( `Sales Amount` ) as `Sales Amount`
    -> FROM Fact_Sales fs
    -> INNER JOIN Dimension_Movie dm on (fs.movie_key = dm.movie_key)
    -> WHERE fs.`Sales Amount` > 10
    -> GROUP BY 1;
+---------------------------+--------------+
| Title                     | Sales Amount |
+---------------------------+--------------+
| !!! MISSING MOVIE !!!     |       999.99 |
| AMERICAN CIRCUS           |        43.96 |
| AUTUMN CROW               |        10.99 |
| BACKLASH UNDEFEATED       |        10.99 |
| BEAST HUNCHBACK           |        21.98 |
| BEHAVIOR RUNAWAY          |        21.98 |
| BILKO ANONYMOUS           |        43.96 |
| BRIGHT ENCOUNTERS         |        10.99 |
| CARIBBEAN LIBERTY         |        43.96 |
| CASUALTIES ENCINO         |        10.99 |
| DAUGHTER MADIGAN          |        10.99 |
| DOORS PRESIDENT           |        10.99 |
| FLASH WARS                |        10.99 |
| FLINTSTONES HAPPINESS     |        44.96 |
| FOOL MOCKINGBIRD          |        32.97 |
| GARDEN ISLAND             |        10.99 |
| HUSTLER PARTY             |        32.97 |
| INNOCENT USUAL            |        21.98 |
| ISHTAR ROCKETEER          |        10.99 |
| KING EVOLUTION            |        21.98 |
| KISSING DOLLS             |        43.96 |
| MAIDEN HOME               |        21.98 |
| MIDSUMMER GROUNDHOG       |        22.98 |
| MINDS TRUMAN              |        32.97 |
| MINE TITANS               |        55.95 |
| NIGHTMARE CHILL           |        10.99 |
| PANIC CLUB                |        10.99 |
| PATHS CONTROL             |        10.99 |
| PINOCCHIO SIMON           |        10.99 |
| RANGE MOONWALKER          |        21.98 |
| SATISFACTION CONFIDENTIAL |        10.99 |
| SATURDAY LAMBS            |        54.95 |
| SCORPION APOLLO           |        23.98 |
| SECRETS PARADISE          |        10.99 |
| SHOW LORD                 |        22.98 |
| STING PERSONAL            |        33.97 |
| STRANGER STRANGERS        |        10.99 |
| SUIT WALLS                |        21.98 |
| SUNRISE LEAGUE            |        21.98 |
| TEEN APOLLO                |          21.98 |
| TELEGRAPH VOYAGE           |          65.94 |
| TIES HUNGER                |          11.99 |
| TITANIC BOONDOCK           |          21.98 |
| TORQUE BOUND               |          43.96 |
| TRAP GUYS                  |          33.97 |
| TYCOON GATHERING           |          43.96 |
| VIRTUAL SPOILERS           |          44.96 |
| WIFE TURN                  |          43.96 |
| WONDERLAND CHRISTMAS       |          10.99 |
| ZORRO ARK                  |          10.99 |
+----------------------------+--------------+
50 rows in set (0.10 sec)
```

At first glance this answer appears to be correct, but is this result exactly what we wanted? Not quite. Let's take a look at the query again, with an English translation for each line:

OBSERVE:

```
SELECT dm.Title,                              --Show the movie title
SUM( `Sales Amount` ) as `Sales Amount`       --And total sales per movie
FROM Fact_Sales fs                            --from Fact_Sales
INNER JOIN Dimension_Movie dm on (fs.movie_key = dm.movie_key) --and from Dimension_Movie
WHERE fs.`Sales Amount` > 10                  --Where Sales Amount in Fact_Sales is greater than 1
GROUP BY 1;
```

Instead of returning a result of movies with *total sales* greater than $10.00, we have returned a result of movies with *one-time sales* greater than $10.00. We filtered our data incorrectly.

So how do we fix this error? One way would be to use a sub-query. We'll **calculate the total sales for each movie**, then **limit those results to Sales Amount > 10**. Run this command against your personal database:

CODE TO TYPE:

```
SELECT Title, `Sales Amount`
FROM (SELECT dm.Title,
 SUM( `Sales Amount` ) as `Sales Amount`
 FROM Fact_Sales fs
 INNER JOIN Dimension_Movie dm on (fs.movie_key = dm.movie_key)
 GROUP BY 1
) as subQuery
WHERE `Sales Amount` > 10;
```

This query returns many more movies in the result - it looks like nearly every movie has generated sales greater than $10.00.

OBSERVE:

```
+------------------------------+--------------+
| Title                        | Sales Amount |
+------------------------------+--------------+
| !!! MISSING MOVIE !!!        |       999.99 |
| ACADEMY DINOSAUR             |        35.78 |
| ACE GOLDFINGER               |        52.93 |
| ADAPTATION HOLES             |        32.89 |
| AFFAIR PREJUDICE             |        91.77 |
...lines omitted...
| WRATH MILE                   |        23.86 |
| WRONG BEHAVIOR               |        62.80 |
| WYOMING STORM                |        72.87 |
| YENTL IDAHO                  |       130.78 |
| YOUTH KICK                   |        12.95 |
| ZHIVAGO CORE                 |        14.91 |
| ZOOLANDER FICTION            |        67.84 |
| ZORRO ARK                    |       214.69 |
+------------------------------+--------------+
947 rows in set (0.74 sec)
```

This type of problem is difficult to spot, especially when our first query seems to work. It's important to have peers review queries to make sure everything is written correctly.

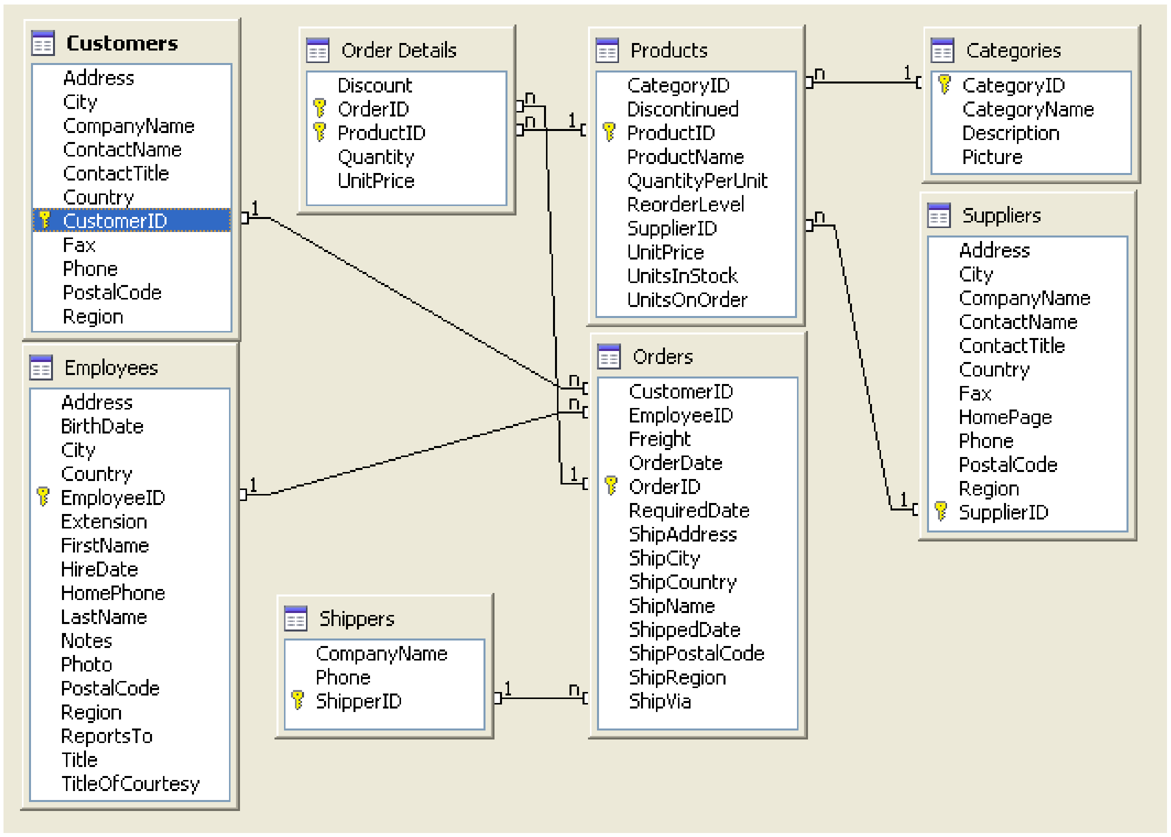
As usual, we've covered a lot in this lesson! You're doing really great, and you're nearly done with this course. The next lesson will be a description of your final project. See you then!

# Final Project
# DBA 3: Data Warehousing Lesson 14

## Northwind Traders

*Northwind Traders* is a sample database that Microsoft distributed with its Access product. It's an old database (the newest dates in it are from 1996), but is a great example of database design.

Here's a diagram of the database:



For your final project, you'll use *Northwind Traders* as a data source to design, implement, and populate a data warehouse.

You are required to implement these dimensions:

- Date
- Employees
- Customers
- Suppliers
- Products
- Orders

and these facts:

- Order Unit Price
- Supplier Unit Price

A read-only copy of the Northwind database is available on the server:

OBSERVE:
```
C:\talend_files\Nwind.mdb
```

**Note** If you type in the file name, be sure to use forward slashes instead of back slashes: `C:/talend_files/Nwind.mdb`

Instead of using MySQL components as the source for your data, you'll need to use the Access components. You can use the SQL

Builder tool in TOS to examine the tables in Northwind and see the various data types of columns. To use the SQL Builder, click on the ... to the right of the query box for tAccessInput.

Your data warehouse will be located in your existing MySQL database. To distinguish tables for your final project from other tables, use the prefix `fp_` for your table names. For example, you might name your date dimension: `fp_dimDate`.

## fp_dimDate

Your date dimension should be called **fp_dimDate**.

The dates in *Northwind Traders* range from 1994 to 1996. Make sure your date dimension has all of the required dates in it. You can use the file `c:\talend_files\NwindDates.xls` to load your date dimension if you like. This date dimension is not the same as the one used in the class - its columns are: `date`, `is_weekend`, `year`, `quarter`, `month`, and `day`.

## fp_dimEmployees

Use the following query to populate **fp_dimEmployees**:

CODE TO TYPE:

```
SELECT EmployeeID, LastName, FirstName, Title, TitleOfCourtesy,
BirthDate, HireDate, Address, City, Region, PostalCode, Country,
HomePhone, Extension
FROM Employees;
```

## fp_dimCustomers

Use the following query to populate **fp_dimCustomers**, a **Type-2 SCD**:

CODE TO TYPE:

```
SELECT CustomerID, CompanyName, ContactName, ContactTitle, Address, City,
Region, PostalCode, Country, Phone, Fax
FROM Customers;
```

## fp_dimSuppliers

Use the following query to populate **fp_dimSuppliers**, a **Type-2 SCD**:

CODE TO TYPE:

```
SELECT SupplierID, CompanyName, ContactName, ContactTitle,
Address, City, Region, PostalCode, Country, Phone, Fax, HomePage
FROM Suppliers;
```

## fp_dimProducts

Use the following query to populate **fp_dimProducts**, a **Type-2 SCD**:

CODE TO TYPE:

```
SELECT Products.ProductID, Products.ProductName, Products.Discontinued,
Categories.CategoryName, Categories.Description as CategoryDescription
FROM Products
INNER JOIN Categories on Products.CategoryID = Categories.CategoryID;
```

## fp_dimOrders

Use the following query to populate **fp_dimOrders**:

CODE TO TYPE:

```
SELECT Orders.OrderID, Customers.CompanyName as CustomerName, Customers.ContactName,
Orders.OrderDate, Orders.RequiredDate,
Orders.ShipName, Orders.ShipAddress, Orders.ShipCity,
Orders.ShipRegion, Orders.ShipPostalCode, Orders.ShipCountry
FROM Orders
INNER JOIN Customers on Orders.CustomerID = Customers.CustomerID;
```

## Order Unit Price

Use the following query to retrieve data for eventual placement in your **fp_factOrderUnitPrice** table:

CODE TO TYPE:

```
SELECT od.OrderID, od.ProductID, od.UnitPrice, o.OrderDate
FROM [Order Details] as od
INNER JOIN Orders as o on od.OrderID = o.OrderID;
```

**Note** You will need to use a staging table for your fact process.

## Supplier Unit Price

Use the following query to retrieve data for eventual placement in your **fp_factSupplierUnitPrice** table:

CODE TO TYPE:

```
SELECT ProductID, SupplierID, UnitPrice
FROM Products;
```

**Note** You will need to use a staging table for your fact process.

There is no dates for this fact. We want to join on the "latest" values for the Product and Supplier dimensions. To do so you need to add the expression `end_date='2099-01-01'` to your join.

As always, feel free to contact your mentor if you have any questions.

**Good luck!**

## Hints for Access

The SQL syntax for Access can be different from the SQL syntax for MySQL. Unfortunately, Access does not have the most clearly stated error messages. If you see a message like:

OBSERVE:

```
[Microsoft][ODBC Microsoft Access Driver] Too few parameters. Expected 1.
```

...then you have misspelled a column or table name.

You'll also want to be aware of this SQL difference between MySQL and Access. In Access, when the table name contains a space, "Order Details" needs to be escaped using brackets: `SELECT ... FROM [Order Details]`.

Again, let your mentor know if you run into a problem. Thanks for playing and have fun with this last project. It's been great working with you!