Query Builder

The New JMP 12 Tool For Getting Your SQL Data into JMP

By Eric Hill, Software Developer, JMP

Contents

Introduction

JMP puts a vast array of analytical tools at your fingertips, but those tools do you no good if you can't get the data you need to analyze into JMP in the way that you need it. JMP has had support for importing data from relational databases into JMP for quite some time via **File > Database > Open Table**. **Open Table** is fine as far as it goes, but it does have some notable limitations:

- Most data in relational databases is stored in a normalized way, requiring joins to get all of the data you need, but **Open Table** does not help you with joins; you have to write the SQL yourself.
- While **Open Table** provides the **Formula Editor** for building a WHERE clause, it does not, for example, show you the distinct values of a categorical variable to help you filter on it.
- It can be difficult to securely share your queries with others, because **Open Table** stores your password in the connection string of the JSL scripts that it generates.
- **Open Table** does not help you with post-processing steps that are required once data has been imported, such as data clean-up, modeling types and formatting.

Query Builder was designed from the ground up to address the limitations of **Open Table**, helping you build multi-table queries with prompted interactive filtering that can be securely shared with co-workers.

In this paper, I will use **Query Builder** with a freely available **PostgreSQL** sample database called **dellstore2**. **PostgreSQL** is an open-source database that can be freely downloaded from here:

http://www.postgresql.org/download/

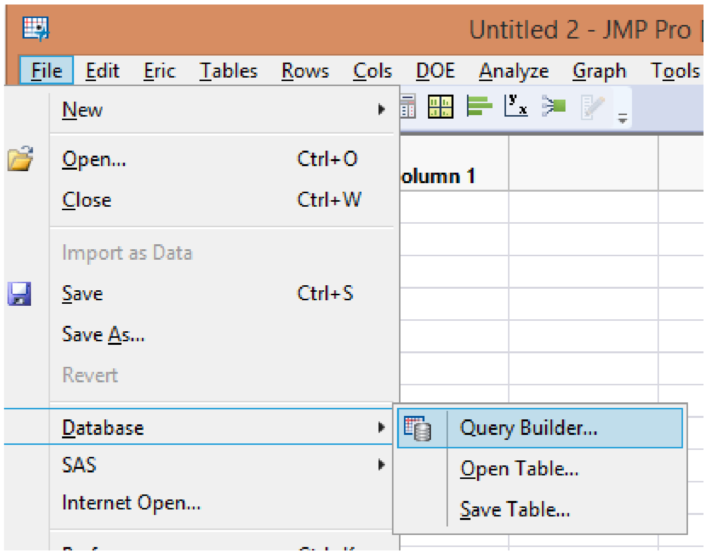
The **dellstore2** sample database can be found here:

http://wiki.postgresql.org/wiki/Sample_Databases

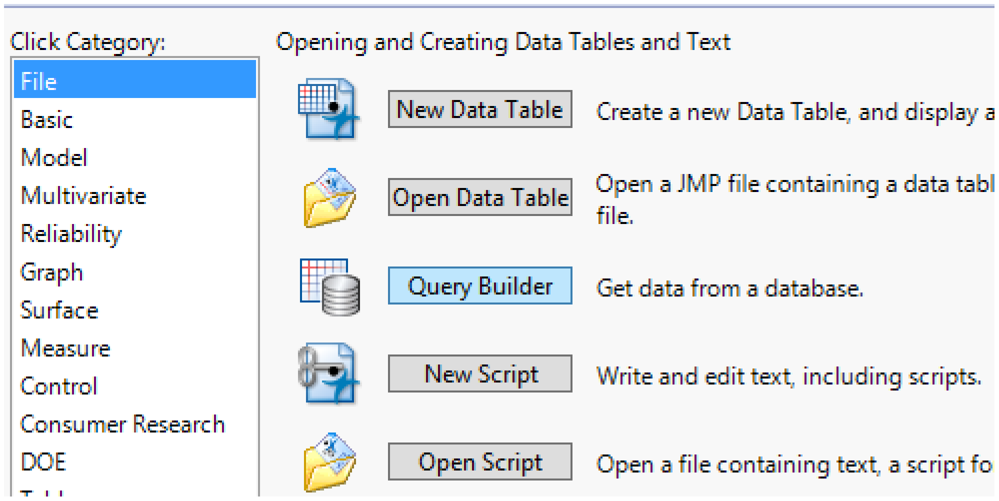
The **dellstore2** sample database is also available for other database platforms (MySQL, Oracle and SQL Server).

Section 1: Making the Connection

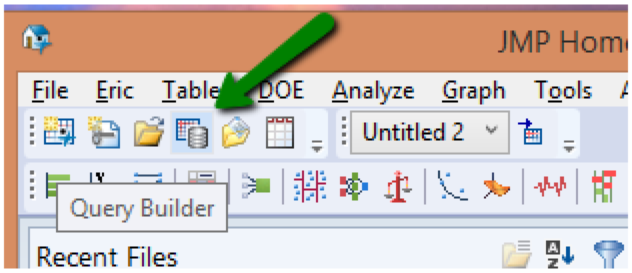
Initiate **Query Builder** using **File > Database > Query Builder** from the menu:



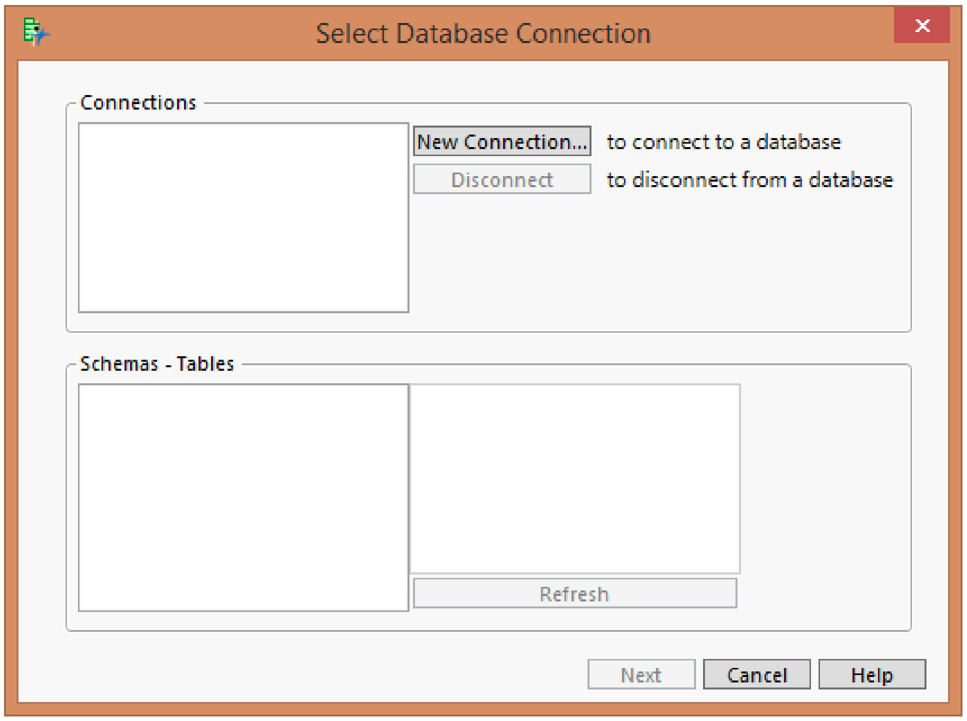
You can also launch it from the first page of **JMP Starter**:


Click Category:
File
Basic
Model
Multivariate
Reliability
Graph
Surface
Measure
Control
Consumer Research
DOE
Opening and Creating Data Tables and Text
New Data Table
Create a new Data Table, and display a
Open Data Table
Open a JMP file containing a data tabl file.
Query Builder
Get data from a database.
New Script
Write and edit text, including scripts.
Open Script
Open a file containing text, a script fo

Lastly, on Windows, you can also find a **Query Builder** tool on the **Home Window** toolbar:

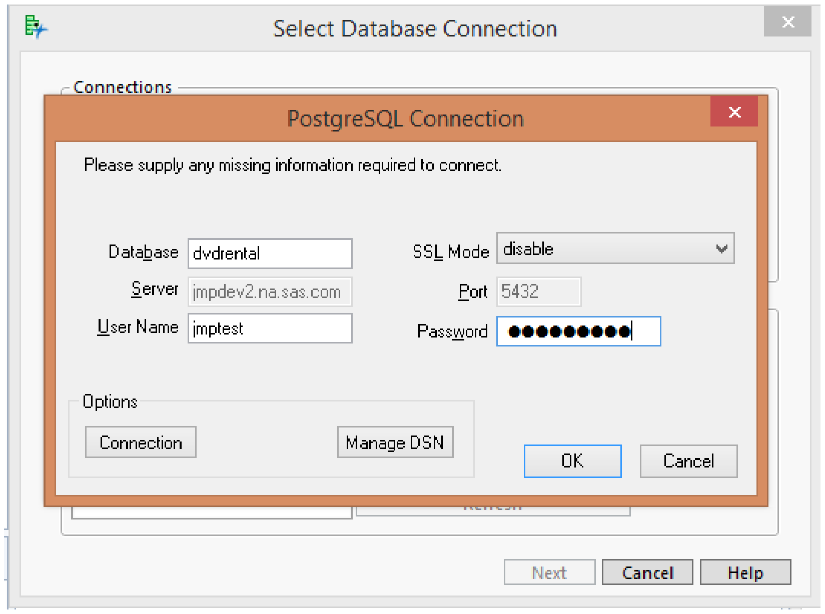


After launching **Query Builder**, you will be greeted with **Select Database Connection** dialog box:



The **Select Database Connection** dialog box is a subset of the **Open Table** window. **Query Builder** and **Open Table** use the same ODBC connections and in fact share connections, so any connections you make in either place will show up in both.

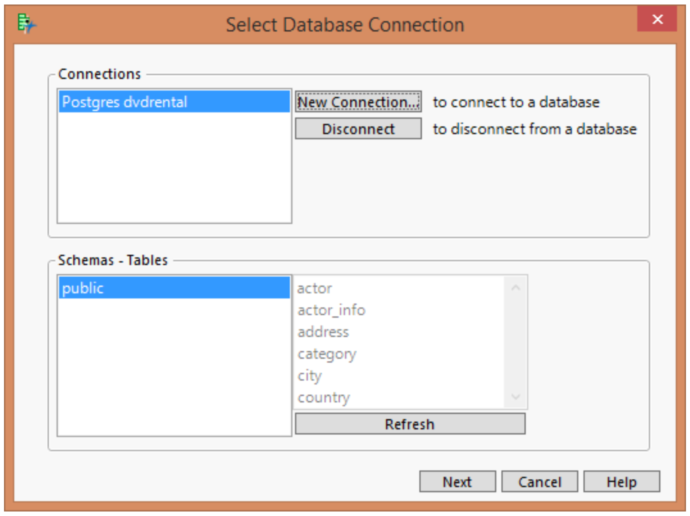
Click **New Connection...** to connect to your database using an existing DSN or to create a new DSN. You will be prompted to supply any required credentials:



Please supply any missing information required to connect.

Database	dvdrental	SSL Mode	disable
Server	jmpdev2.na.sas.com	Port	5432
User Name	jmptest	Password	●●●●●●●●●

After connecting, JMP will retrieve schemas from the connection to help you make sure this is the connection that you want:



Connections: Postgres dvdrental

New Connection... to connect to a database

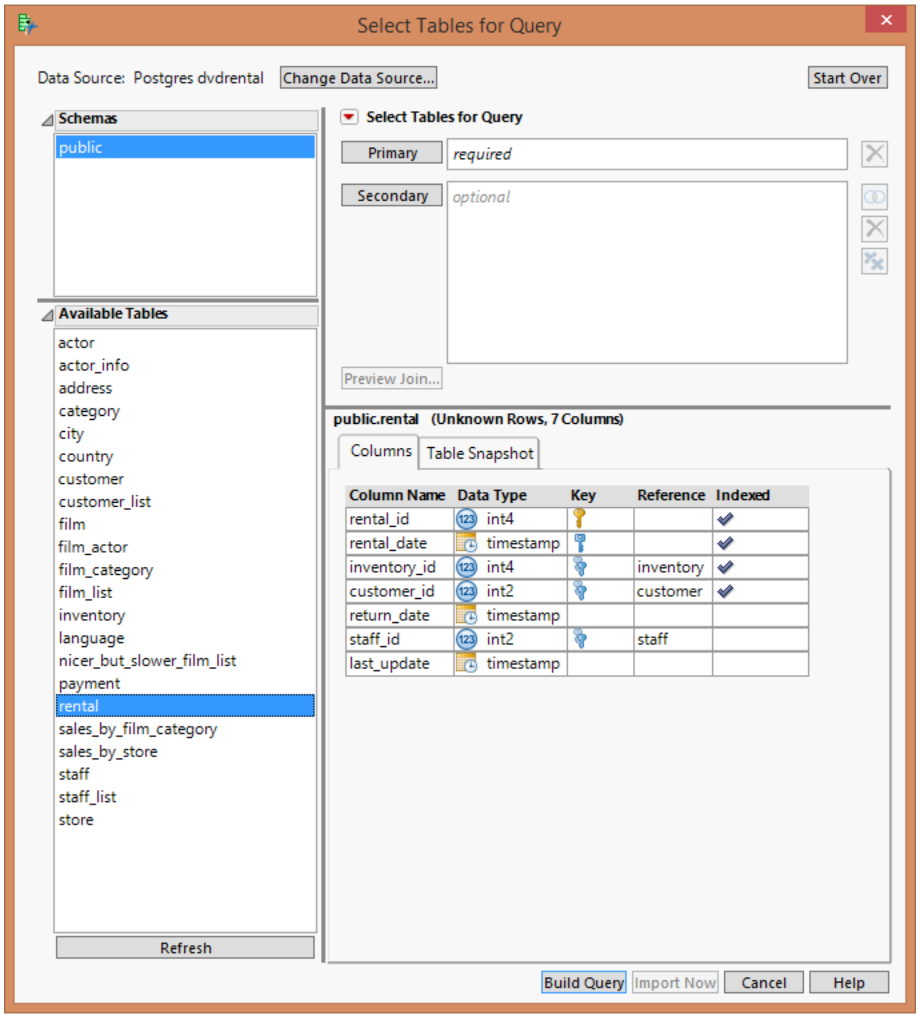
Disconnect to disconnect from a database

Schemas - Tables: public

actor
actor_info
address
category
city
country

You can also see the tables available in the selected schema. Click **Next** to continue to the next step.

Section 2: Selecting Tables for the Query



Data Source: Postgres dvdrental

Schemas: public

Primary: required

Secondary: optional

Available Tables:

actor
actor_info
address
category
city
country
customer
customer_list
film
film_actor
film_category
film_list
inventory
language
nicer_but_slower_film_list
payment
rental
sales_by_film_category
sales_by_store
staff
staff_list
store

public.rental (Unknown Rows, 7 Columns)

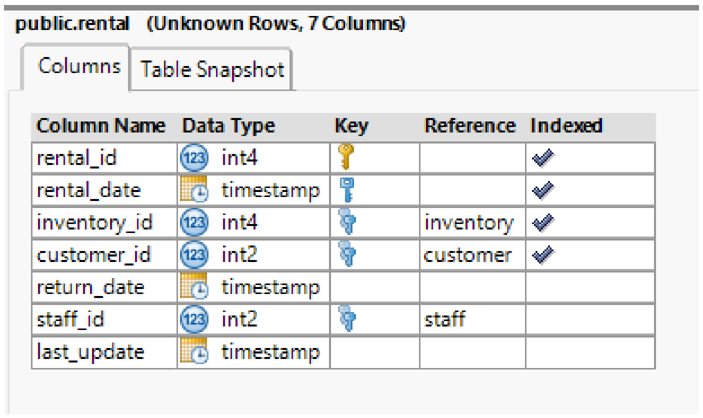
Column Name	Data Type	Key	Reference	Indexed
rental_id	int4			✓
rental_date	timestamp			✓
inventory_id	int4		inventory	✓
customer_id	int2		customer	✓
return_date	timestamp			
staff_id	int2		staff	
last_update	timestamp			

Figure 1: Selecting Tables for the Query

When you click **Next**, the **Select Tables** dialog box will appear. In the **Select Tables** dialog, you will select which tables you want to include in your query (see *Figure 1* above). Select tables in the **Available Tables** list that you want to include in the query and add them to the **Primary** and **Secondary** table lists at the upper right.

- The **Primary** table in a query is normally the table from which all rows should be retrieved, regardless of whether they have a matching value in some other table. This is your **fact table**, in data warehousing terms. There can only be one primary table in a query.
- **Secondary** tables are normally look-up tables. By default, rows from secondary tables are only included in the query when the value of a column in the **primary** table matches the value of a specified column in the **secondary** table. These are your **dimension tables**, in data warehousing terms. You can add as many **secondary** tables to the query as you want.

The panel in the lower right shows information about the selected table. The **Columns** tab displays information about each column in the table, including whether it is a key of some kind:

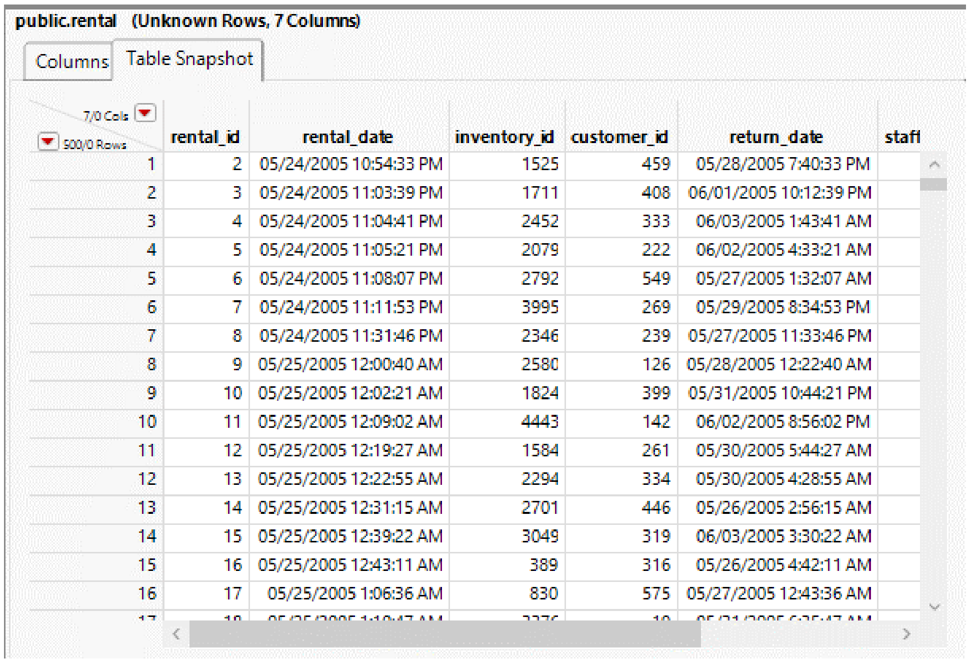


public.rental (Unknown Rows, 7 Columns)

Columns | Table Snapshot

Column Name	Data Type	Key	Reference	Indexed
rental_id	int4	🔑		✓
rental_date	timestamp	🔑		✓
inventory_id	int4	🔑	inventory	✓
customer_id	int2	🔑	customer	✓
return_date	timestamp			
staff_id	int2	🔑	staff	
last_update	timestamp			

The **Table Snapshot** panel shows a snapshot of the data so that you can make sure the table has the information you need:



public.rental (Unknown Rows, 7 Columns)

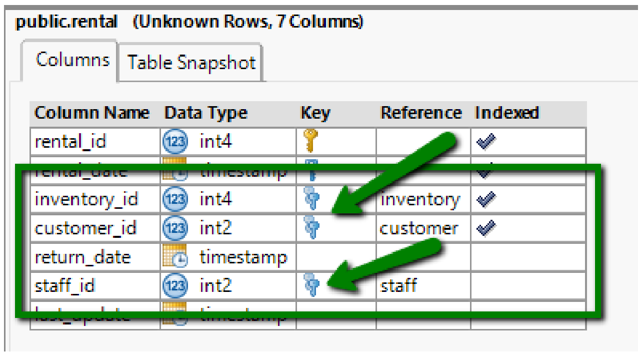
Columns | Table Snapshot

7/0 Cols 500/0 Rows	rental_id	rental_date	inventory_id	customer_id	return_date	staf
1	2	05/24/2005 10:54:33 PM	1525	459	05/28/2005 7:40:33 PM	
2	3	05/24/2005 11:03:39 PM	1711	408	06/01/2005 10:12:39 PM	
3	4	05/24/2005 11:04:41 PM	2452	333	06/03/2005 1:43:41 AM	
4	5	05/24/2005 11:05:21 PM	2079	222	06/02/2005 4:33:21 AM	
5	6	05/24/2005 11:08:07 PM	2792	549	05/27/2005 1:32:07 AM	
6	7	05/24/2005 11:11:53 PM	3995	269	05/29/2005 8:34:53 PM	
7	8	05/24/2005 11:31:46 PM	2346	239	05/27/2005 11:33:46 PM	
8	9	05/25/2005 12:00:40 AM	2580	126	05/28/2005 12:22:40 AM	
9	10	05/25/2005 12:02:21 AM	1824	399	05/31/2005 10:44:21 PM	
10	11	05/25/2005 12:09:02 AM	4443	142	06/02/2005 8:56:02 PM	
11	12	05/25/2005 12:19:27 AM	1584	261	05/30/2005 5:44:27 AM	
12	13	05/25/2005 12:22:55 AM	2294	334	05/30/2005 4:28:55 AM	
13	14	05/25/2005 12:31:15 AM	2701	446	05/26/2005 2:56:15 AM	
14	15	05/25/2005 12:39:22 AM	3049	319	06/03/2005 3:30:22 AM	
15	16	05/25/2005 12:43:11 AM	389	316	05/26/2005 4:42:11 AM	
16	17	05/25/2005 1:06:36 AM	830	575	05/27/2005 12:43:36 AM	

Normally, when building a query, you have a question or questions in mind that you are hoping to answer, and those questions drive decisions about which tables your query needs to include. The data we are exploring is from a fictitious DVD rental company. The questions I would like to answer are:

1. **Which movie genres are bringing in the most revenue?**
2. **Which actors are bringing in the most revenue?**

The **rental** table has a record for each DVD rental event over the life of this company, so that is going to be the primary table for this query. I can see from the **Columns** panel that **rental** has columns with foreign key relationships to the **inventory** and **customer** tables:



public.rental (Unknown Rows, 7 Columns)

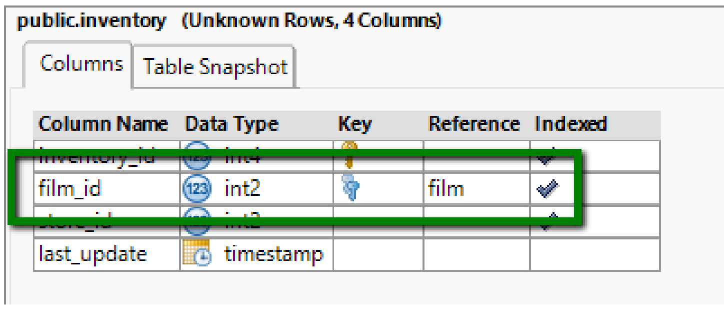
Columns | Table Snapshot

Column Name	Data Type	Key	Reference	Indexed
rental_id	int4	🔑		✓
rental_date	timestamp	🔑		✓
inventory_id	int4	🔑	inventory	✓
customer_id	int2	🔑	customer	✓
return_date	timestamp			
staff_id	int2	🔑	staff	
last_update	timestamp			

Figure 2: Foreign Keys

A **foreign key** relationship (also known as a foreign key **constraint**) means that the values in the column of one table are required to be found in the **primary key** column of *another* table. From *Figure 2*, we can see that values from the **inventory_id** column of the **rentals** table must exist in the **primary key** column of the **inventory** table, values from the **customer_id** column of the **rentals** table must exist in the **primary key** column of the **customer** table, and values from the **staff_id** column of the rentals table just exist in the **primary key** column of the **staff** table. For this query, I am looking for more information about the movie rented, rather than the customer of the staff member involved in the rental, so I will add the **inventory** table to the query as a secondary table.

Looking at the **Columns** panel for the **inventory** table, I can further see a **foreign key** relationship with the **film** table:



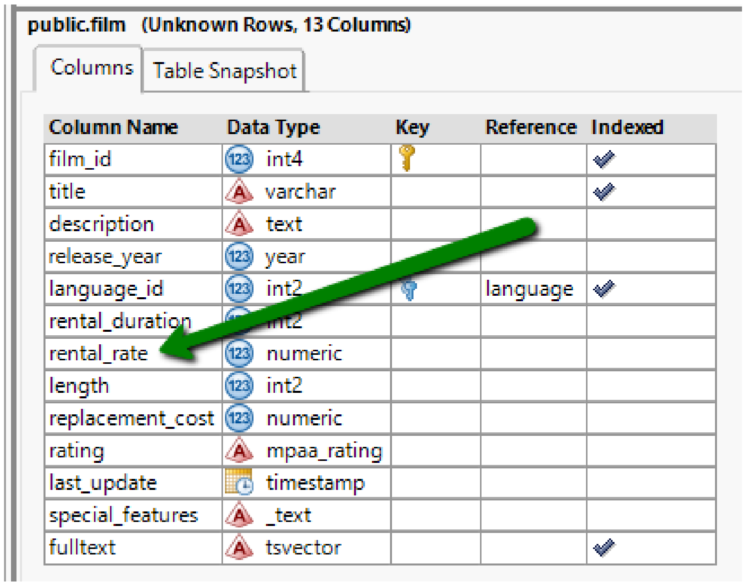
public.inventory (Unknown Rows, 4 Columns)

Columns | Table Snapshot

Column Name	Data Type	Key	Reference	Indexed
inventory_id	int4	🔑		✓
film_id	int2	🔑	film	✓
store_id	int2			✓
last_update	timestamp			

Figure 3: inventory table details

So the **film** table goes into the query also. Looking at the details of the **film** table, I see a **rental_rate** column, which should be what I need for computing revenue, but I don't see anything about the category of the film or the actors in the film:

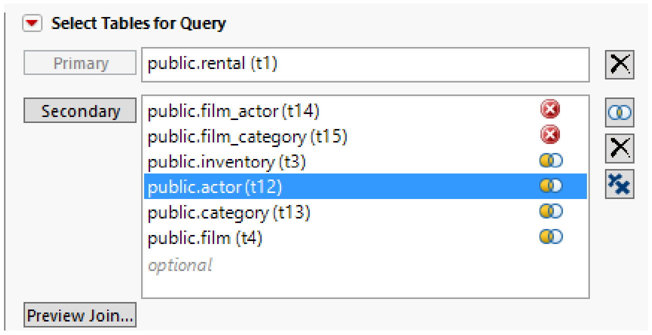

public.film (Unknown Rows, 13 Columns)

Columns | Table Snapshot

Column Name	Data Type	Key	Reference	Indexed
film_id	int4	🔑		✓
title	varchar			✓
description	text			
release_year	year			
language_id	int2	🔑	language	✓
rental_duration	int2			
rental_rate	numeric			
length	int2			
replacement_cost	numeric			
rating	mpaa_rating			
last_update	timestamp			
special_features	_text			
fulltext	tsvector			✓

Figure 4: Columns in the film table

However, looking at the **Available Tables** list, I see a **film_actor** table and a **film_category** table, which appear to map a film's ID to its actors and category, as well as **actor** and **category** tables that have more information about the actors and categories. So I am going to want these tables in the query as well. However, if I simply add those 4 tables, something appears to go wrong:


Select Tables for Query

Primary: public.rental (t1)

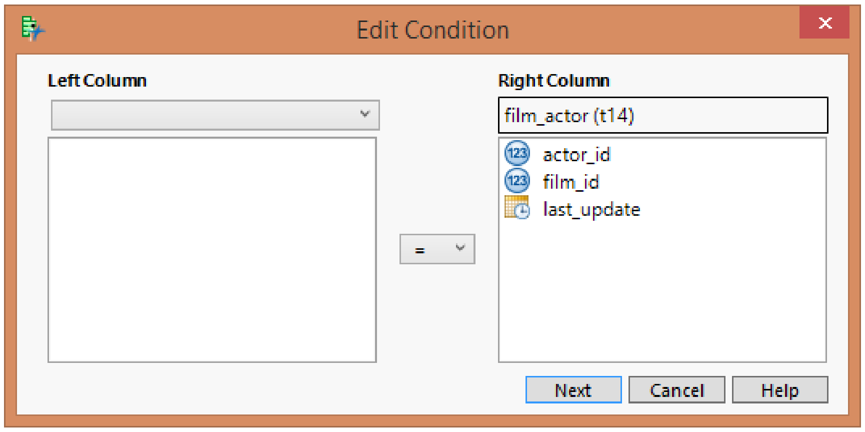
Secondary:
public.film_actor (t14)
public.film_category (t15)
public.inventory (t3)
public.actor (t12)
public.category (t13)
public.film (t4)
optional

Preview Join...

Figure 5: Automatic join unsuccessful

The "stop sign" icons with the "X" in the middle signify that JMP was unable to automatically join those two tables, so I will have to set up the joins manually. If you find that JMP's automatic join function frequently does not do what you need it to do, you can turn it off from the red triangle next to **Select Tables for Query**.

If I select **public.film_actor (t14)** and click the **Edit Join** button (), the **Edit Condition** dialog box comes up, allowing me to configure the join. Initially, the dialog box looks like this:


Edit Condition

Left Column

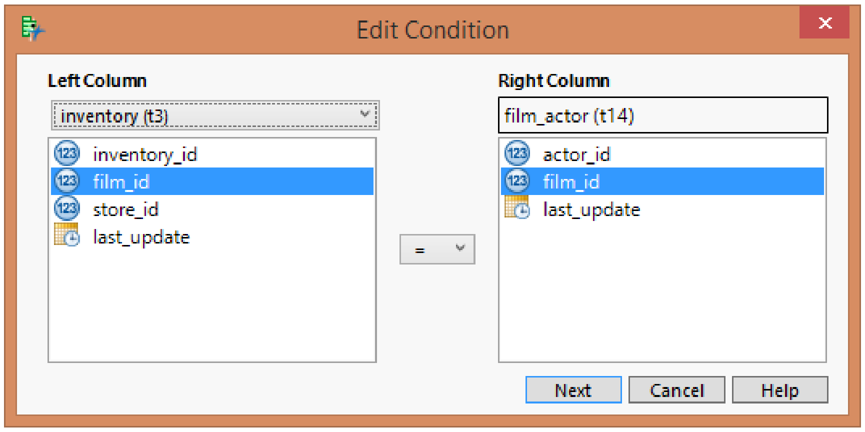
Right Column

film_actor (t14)

actor_id
film_id
last_update

Figure 6: Editing the join condition

since no join condition exists yet. I want to join **film_actor.film_id** to **inventory.film_id**. To accomplish this, I simply select the **inventory** table from the **Left Column** list and select **film_id** in both lists:


Edit Condition

Left Column

inventory (t3)

inventory_id
film_id
store_id
last_update

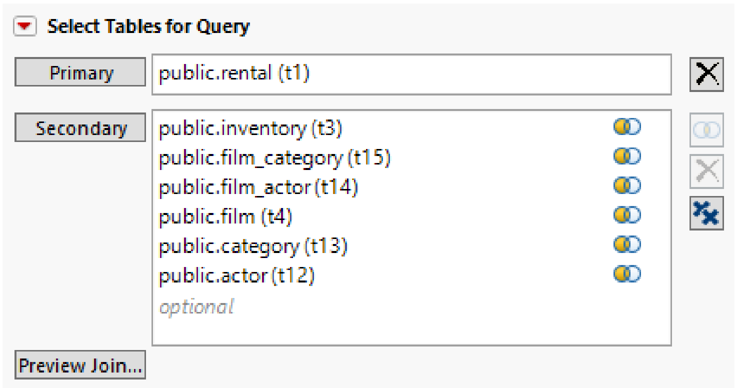
Right Column

film_actor (t14)

actor_id
film_id
last_update

Figure 7: Editing the join condition

Similarly, I can set the join condition for **public.film_category** to be **inventory.film_id = film_category.film_id**. After doing so, things look better:


Select Tables for Query

Primary: public.rental (t1)

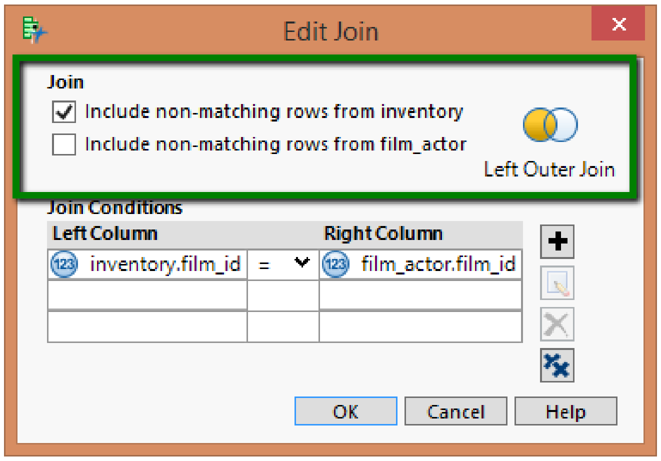
Secondary:
public.inventory (t3)
public.film_category (t15)
public.film_actor (t14)
public.film (t4)
public.category (t13)
public.actor (t12)
optional

Figure 8: Joins between tables are valid

All of the secondary tables are showing the **left outer join** Venn Diagram icon. Using the join we just made between **film_actor** and **inventory** as an example, a left outer join means that **all** rows from the **inventory** table, which is the left table in this case, will be included in the query whether there is a match in the **film_actor** table or not, but only rows from the **film_actor** table that match a row in the **inventory** table will be included.

By the way, the **t1**, **t2**, **t3**, etc., in parentheses after the table names are the table aliases that JMP assigns to the tables as you add them to the query. Table aliases make it easier to generate correct SQL and make the generated SQL more readable. You can change the table alias for a table by right-clicking on the table in the **Primary** or **Secondary** table list and selecting **Change Alias**. Table aliases cannot contain spaces or special characters.

If you need something other than left outer join, you can change the join type using the check boxes on the **Edit Join** dialog. The join type and Venn Diagram icon will change to reflect the state of the checkboxes.


Edit Join

Join
- [✓] Include non-matching rows from inventory
- [ ] Include non-matching rows from film_actor

Left Outer Join

Join Conditions

Left Column		Right Column
inventory.film_id	=	film_actor.film_id

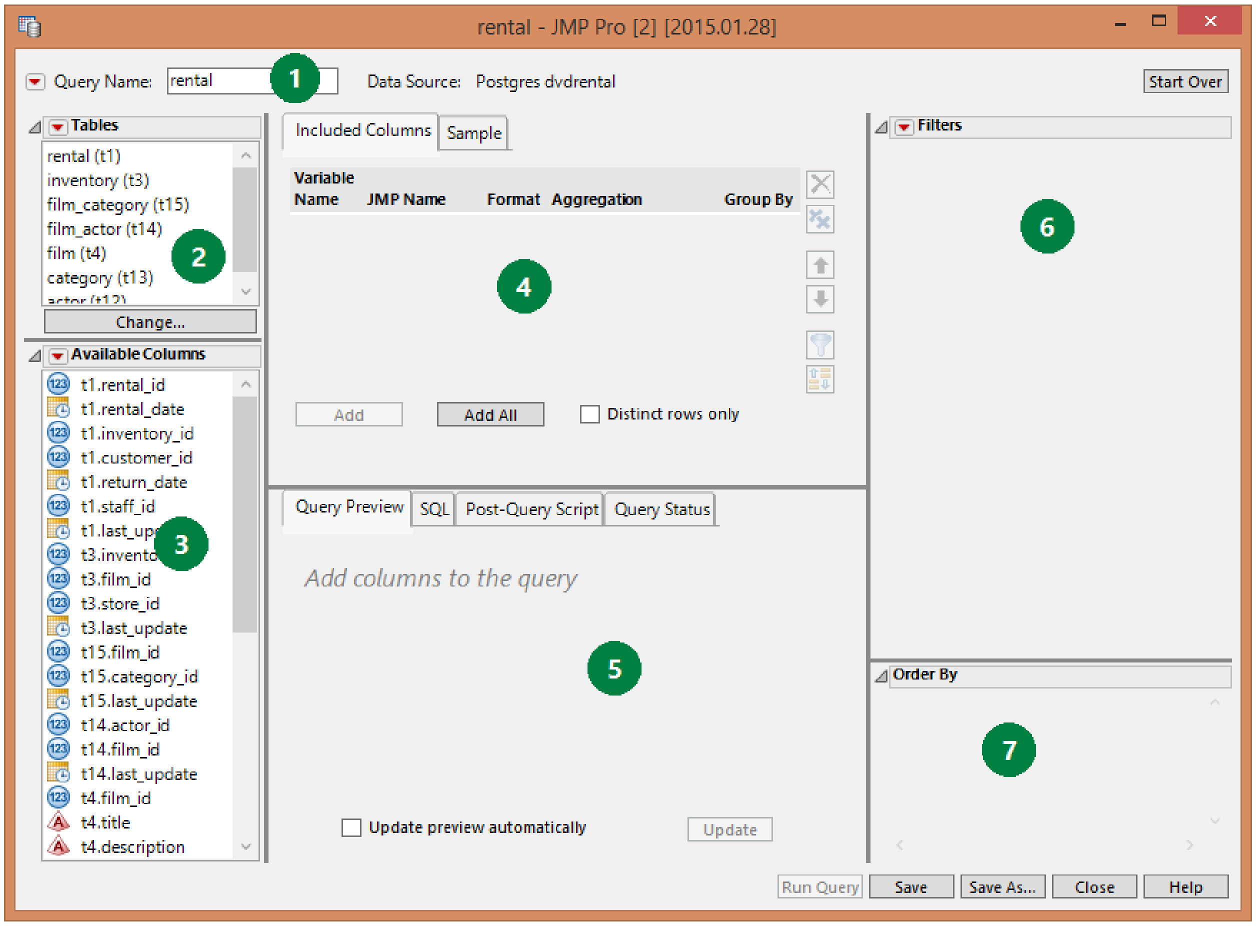
Figure 9: Changing the join type

To check whether you have the joins set up correctly, you can click the **Preview Join** button. That will run the query and bring up a snapshot of the first few hundred rows of data so that you can check that things look right.

Now that we have all the tables in the query that we need, we can click **Build Query** to bring up **Query Builder** itself, where we can specify which columns to include in the query, filter criteria to use, how the resulting data should be ordered, and much more.

Section 3: Building the Query

When the **Query Builder** window comes up initially for this query, it looks like this:



(1) The **Query Name** box holds the name of the query, which defaults to the name of the primary table. The **Query Name** is the name assigned to the JMP data table that results from running the query, and is also the default file name when saving the query to a file. The **Data Source** shows you the name of the data source that the query is using.

(2) The **Tables** box shows you the tables that are currently in the query. To add or remove tables or change joins, click the **Change…** button.

(3) The **Available Columns** list shows all of the columns from all of the tables in the query. Select columns from this list and drag them to the **Included Columns**, **Filters**, or **Order By** panels. Or, if you prefer, there are right-click options to accomplish the same thing.

(4) The **Included Columns** panel shows a grid of all columns that your query includes. Use the **Included Columns** grid to set things like the format and modeling type that data for the column should have after being imported into JMP. You can also specify aggregations along with **Group By** for columns if you want your data summarized before it comes into JMP.
There is also a **Sample** tab in this panel for generating random samples of data on the server, for those databases that support sampling.

(5) The **Query Preview** tab in this area automatically updates to show a snapshot of the data your query will return when you run it. The **SQL** tab shows the SQL that **Query Builder** is generating based on current

selections. The **Post-Query Script** tab allows you to specify a JSL script that will be run after your query completes. The **Query Status** tab shows the status of queries that are running in the background and allows you to cancel them.

(6) The **Filters** panel lets you specify criteria for filtering the data (creating the WHERE clause) and turn the filter into a prompt that runs each time you run the query.

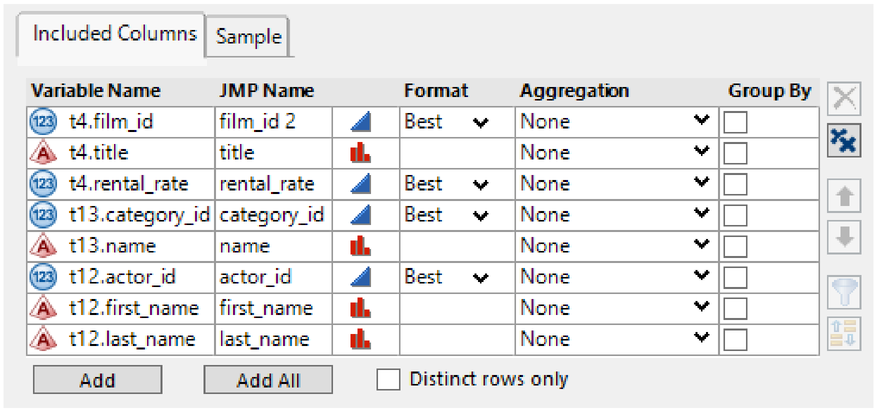
(7) The **Order By** panel lets you specify the column or columns to use for ordering the returned rows (ORDER BY clause).

Adding Columns

Every query needs some columns. Add columns to the query using any of these methods:

1. Click the **Add All** button to add all available columns from all tables in the query
2. Select specific columns from the **Available Columns** list and click **Add** or drag them over.
3. Double-click on a column in the **Available Columns** list.

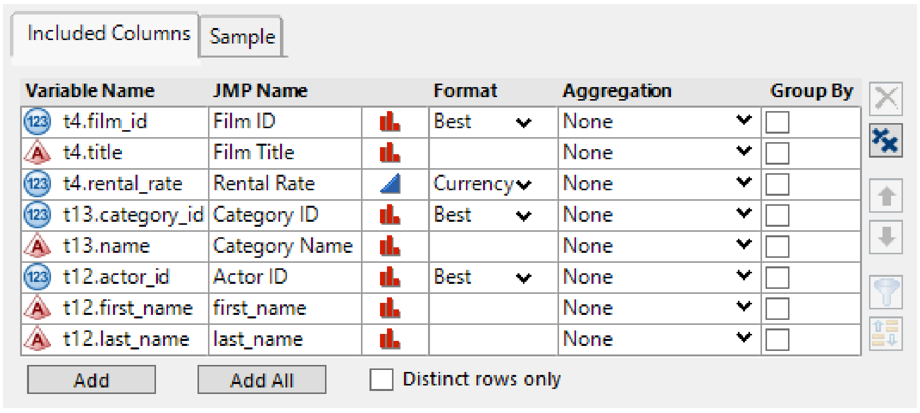
For this query, I will add **film_id**, **title**, and **rental_rate** from the **film** table, **category_id** and **name** from the **category** table, and **actor_id**, **first_name**, and **last_name** from the **actor** table.



Variable Name	JMP Name		Format	Aggregation	Group By
t4.film_id	film_id 2		Best	None	☐
t4.title	title			None	☐
t4.rental_rate	rental_rate		Best	None	☐
t13.category_id	category_id		Best	None	☐
t13.name	name			None	☐
t12.actor_id	actor_id		Best	None	☐
t12.first_name	first_name			None	☐
t12.last_name	last_name			None	☐

Figure 10: After initially adding columns

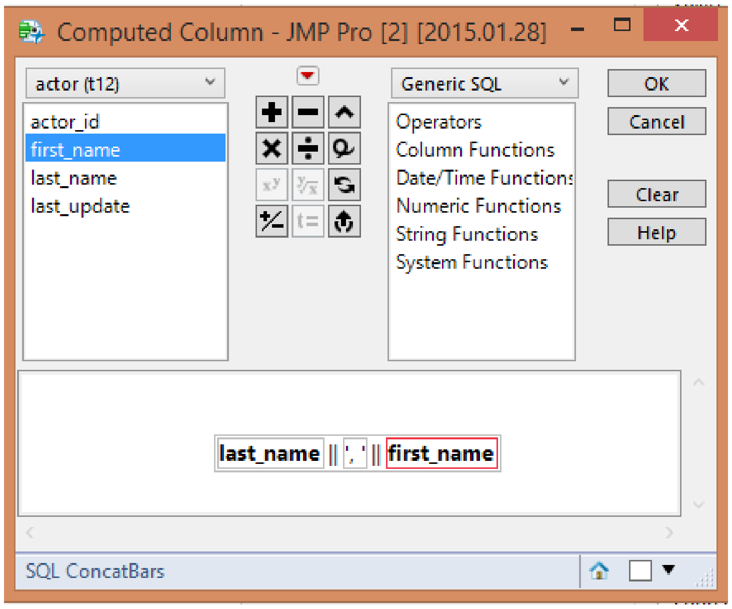
Using the **JMP Name,** modeling type and **Format** columns of the **Included Columns**, I can control what the name, modeling type and format of each column will be after the data is imported into JMP.



Variable Name	JMP Name		Format	Aggregation	Group By
t4.film_id	Film ID		Best	None	☐
t4.title	Film Title			None	☐
t4.rental_rate	Rental Rate		Currency	None	☐
t13.category_id	Category ID		Best	None	☐
t13.name	Category Name			None	☐
t12.actor_id	Actor ID		Best	None	☐
t12.first_name	first_name			None	☐
t12.last_name	last_name			None	☐

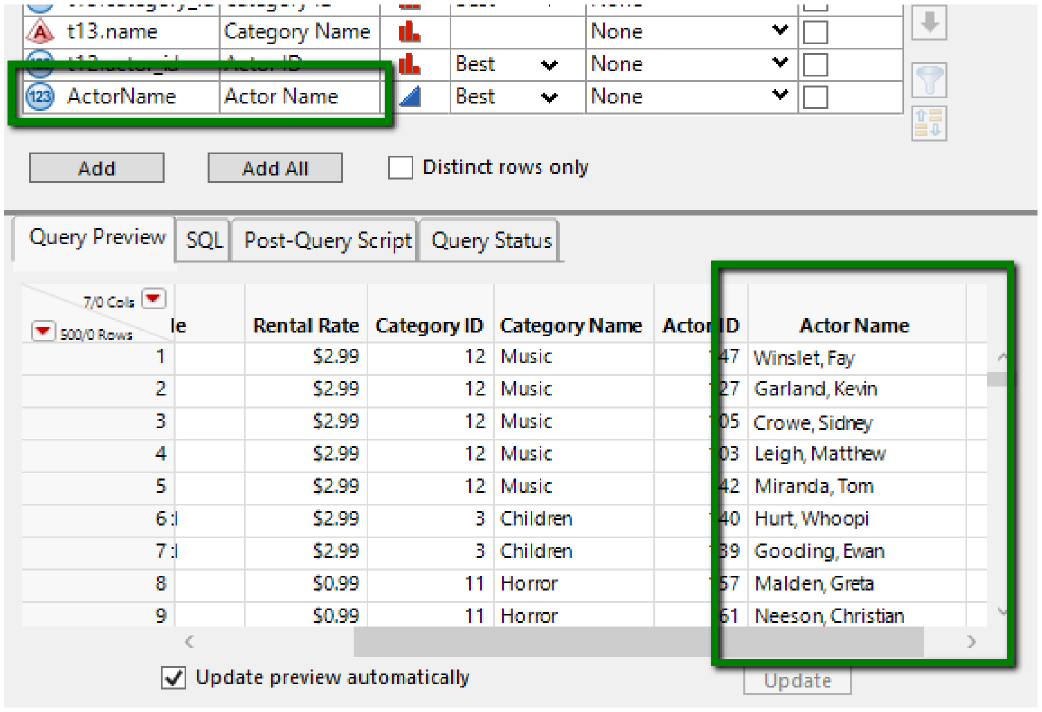
Figure 11: After adjusting column details

For the purposes of this analysis, I would also prefer that the actor names were concatenated in a single column rather than split into separate **first_name** and **last_name** columns. To achieve that, I can add a **computed column** by selecting **Add Computed Column** from the red triangle menu on the **Available Columns** panel.



```
last_name || ', ' || first_name
```

This will add a column named **Calc1** to the **Available Columns** list, but I can right-click it and use **Rename Column** to rename it to **ActorName**, add it to the query and remove **first_name** and **last_name**. Here is how the preview of the query looks after doing so:



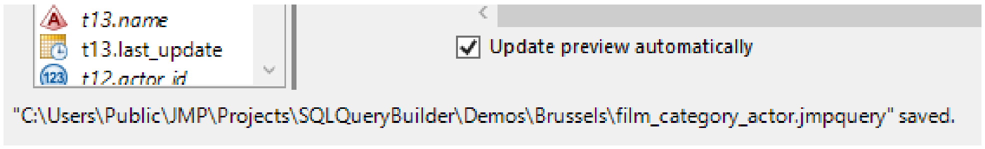
t13.name	Category Name		None	
t12.actor_id	Actor ID	Best	None	
ActorName	Actor Name	Best	None	

	le	Rental Rate	Category ID	Category Name	Actor ID	Actor Name
1		$2.99	12	Music	47	Winslet, Fay
2		$2.99	12	Music	27	Garland, Kevin
3		$2.99	12	Music	05	Crowe, Sidney
4		$2.99	12	Music	03	Leigh, Matthew
5		$2.99	12	Music	42	Miranda, Tom
6		$2.99	3	Children	40	Hurt, Whoopi
7		$2.99	3	Children	39	Gooding, Ewan
8		$0.99	11	Horror	57	Malden, Greta
9		$0.99	11	Horror	61	Neeson, Christian

Figure 12: Query Preview with computed column

Saving the Query

At this point, I have enough time invested in this query that I would like to save it so that I do not lose my work. Simply click the **Save** button at the bottom-right of the **Query Builder** window to save the query. You will be prompted to supply a file name, with the text from the **Query Name** box supplied as the default. I will name the query **film_category_actor**. A message will be displayed at the bottom-left of the **Query Builder** window confirming that the query was saved:



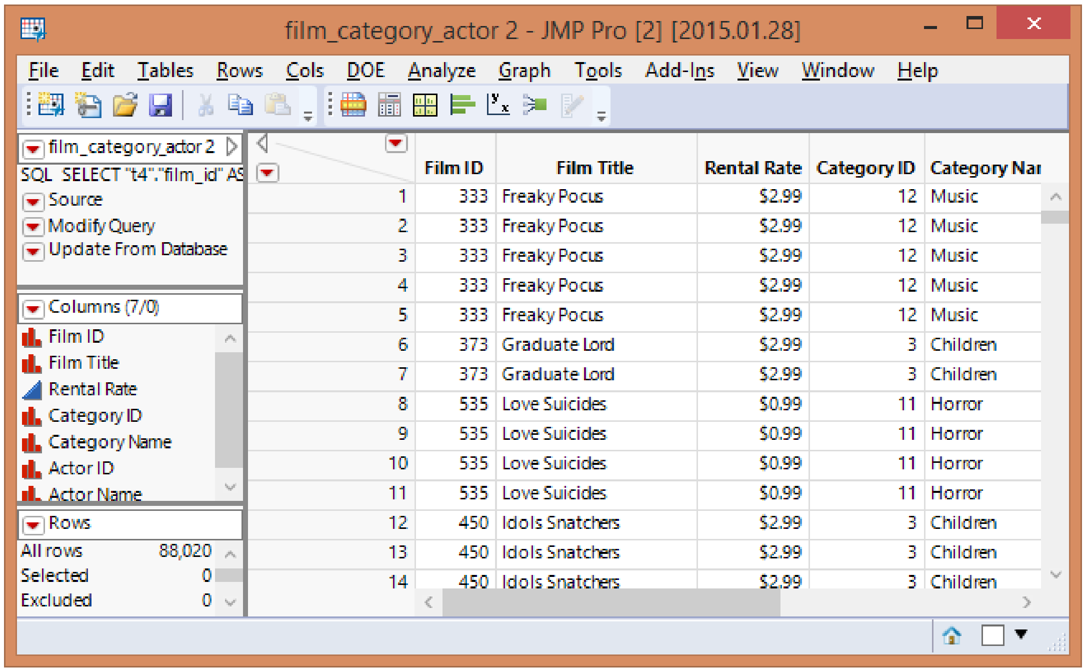
"C:\Users\Public\JMP\Projects\SQLQueryBuilder\Demos\Brussels\film_category_actor.jmpquery" saved.

Saving a query creates a file with a **.jmpquery** extension. Back on the **Query Builder** window, you'll notice that the **Query Name** has been changed to match the file name, so it is now **film_category_actor**.

If you need to create another query that is similar to the one you have open, but you want to preserve the original, you can use the **Save As…** button to save a copy with a different name.

Running the Query

I can also run the query at this point to create a JMP data table containing all the rows from the database. Click the **Run Query** button at the bottom of the **Query Builder** window to run the query. By default, JMP runs queries in the background so that you can continue working while your queries run. Use the **Query Status** tab in the bottom-center section of the window to see the status of queries that are still in process from this **Query Builder** window, or use **View > Running Queries** to see the status of all queries that are currently running in the background. This particular query takes just a second or two, returning 88,020 rows:



SQL SELECT "t4"."film_id" AS

Rows	
All rows	88,020
Selected	0
Excluded	0

	Film ID	Film Title	Rental Rate	Category ID	Category Nar
1	333	Freaky Pocus	$2.99	12	Music
2	333	Freaky Pocus	$2.99	12	Music
3	333	Freaky Pocus	$2.99	12	Music
4	333	Freaky Pocus	$2.99	12	Music
5	333	Freaky Pocus	$2.99	12	Music
6	373	Graduate Lord	$2.99	3	Children
7	373	Graduate Lord	$2.99	3	Children
8	535	Love Suicides	$0.99	11	Horror
9	535	Love Suicides	$0.99	11	Horror
10	535	Love Suicides	$0.99	11	Horror
11	535	Love Suicides	$0.99	11	Horror
12	450	Idols Snatchers	$2.99	3	Children
13	450	Idols Snatchers	$2.99	3	Children
14	450	Idols Snatchers	$2.99	3	Children

Figure 13: Results of query in JMP Data Table

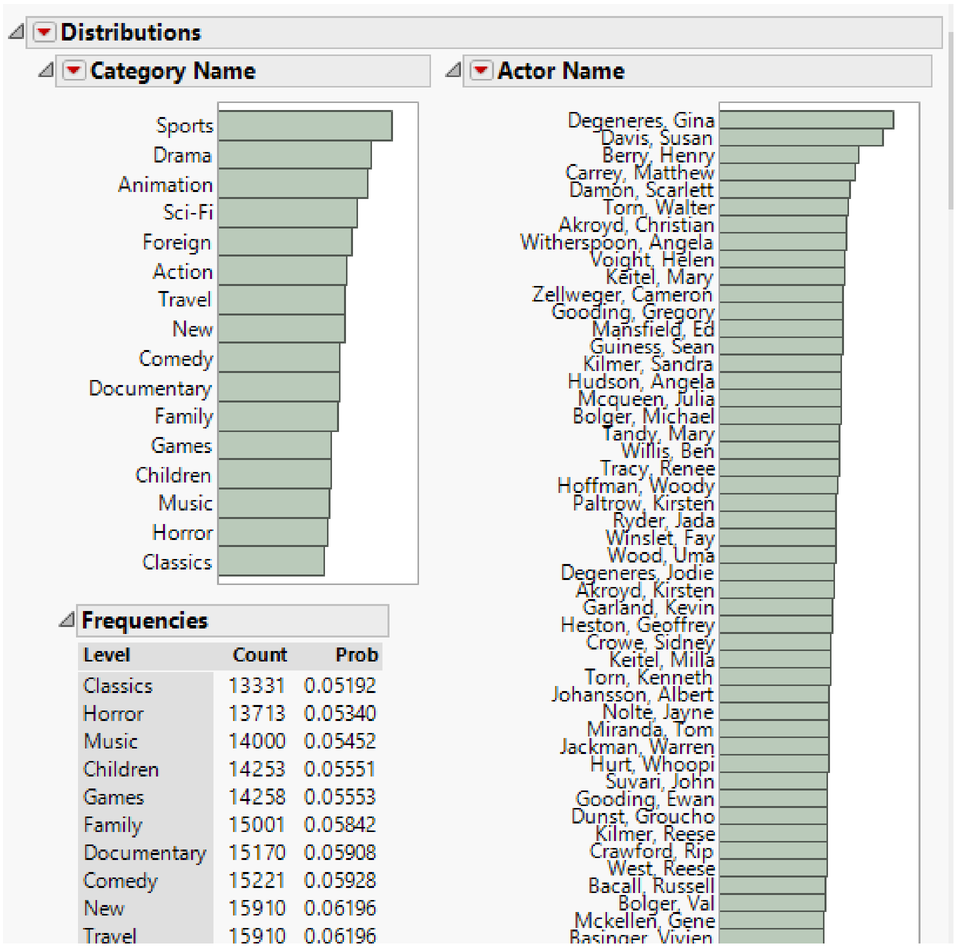
Notice that the resulting JMP data table has 3 scripts and a **SQL** table variable defined:

- The **SQL** table variable contains the SQL that was submitted to the database to perform the query.
- The **Source** script will run the query again.
- The **Modify Query** script will bring the **Query Builder** window back up with the query loaded in it.
- The **Update from Database** script will run the query again and update this JMP data table in-place with the results.

Answering the Questions

At this point, I am close to being able to answer the original questions I was seeking answers to, but I have a decision to make: Do I want to have the database summarize the results, or do I want to bring the raw data into JMP and have JMP do the work? If the raw data is not too large to fit in memory and does not take a prohibitively long time to import, then importing the raw data into JMP and doing the analysis there as many advantages – JMP is much more capable of analyzing data than a relational database.

For the task at hand, it is a simple matter of using the **Distribution** platform on **Category Name** and **Actor Name**, using **Rental Rate** as the **Freq** variable. If I order the resulting histograms by **Count Ascending**, I get a result that answers both of our questions:



Frequencies

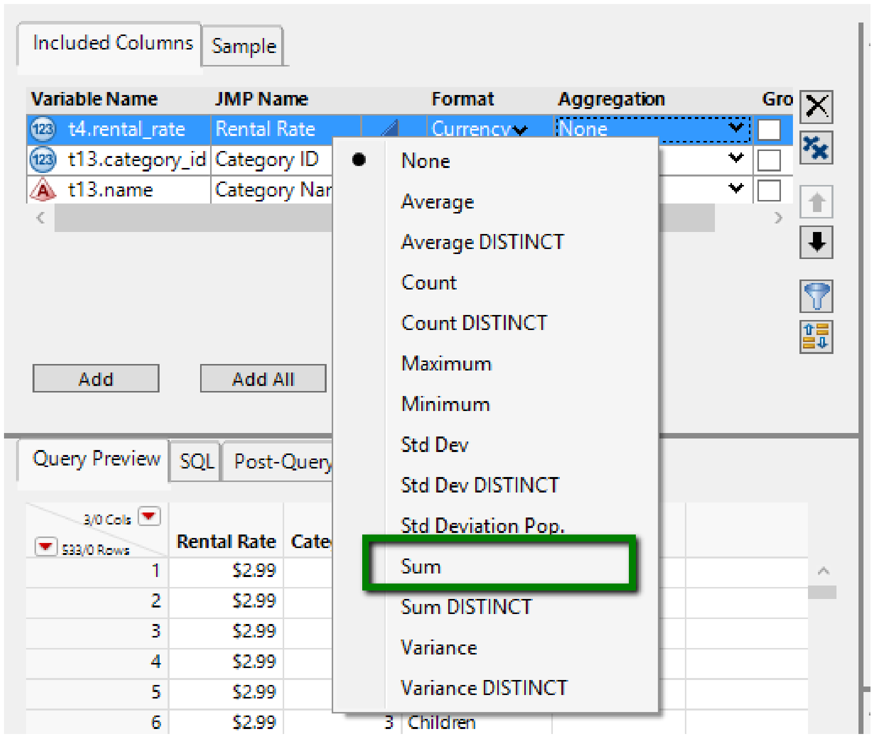
Level	Count	Prob
Classics	13331	0.05192
Horror	13713	0.05340
Music	14000	0.05452
Children	14253	0.05551
Games	14258	0.05553
Family	15001	0.05842
Documentary	15170	0.05908
Comedy	15221	0.05928
New	15910	0.06196
Travel	15910	0.06196

Figure 14: Using the Distribution Platform on the query result

If, however, bringing all the raw data in was prohibitive in size or elapsed time, I can as an alternative use the **Aggregation** column in the **Included Columns** panel to have the database summarize the data. For that, however, I will need to answer the two questions one at a time.

To answer the question of **Which movie genres are bringing in the most revenue?**, I can remove the columns having to do with actor information from the query and ask the database to sum **Rental Rate** grouping by **Category Name**, and then I can order the result by the summed **Rental Rate**. Here are the steps:

1. Click the **Aggregation** column for **Rental Rate** and select **Sum**.

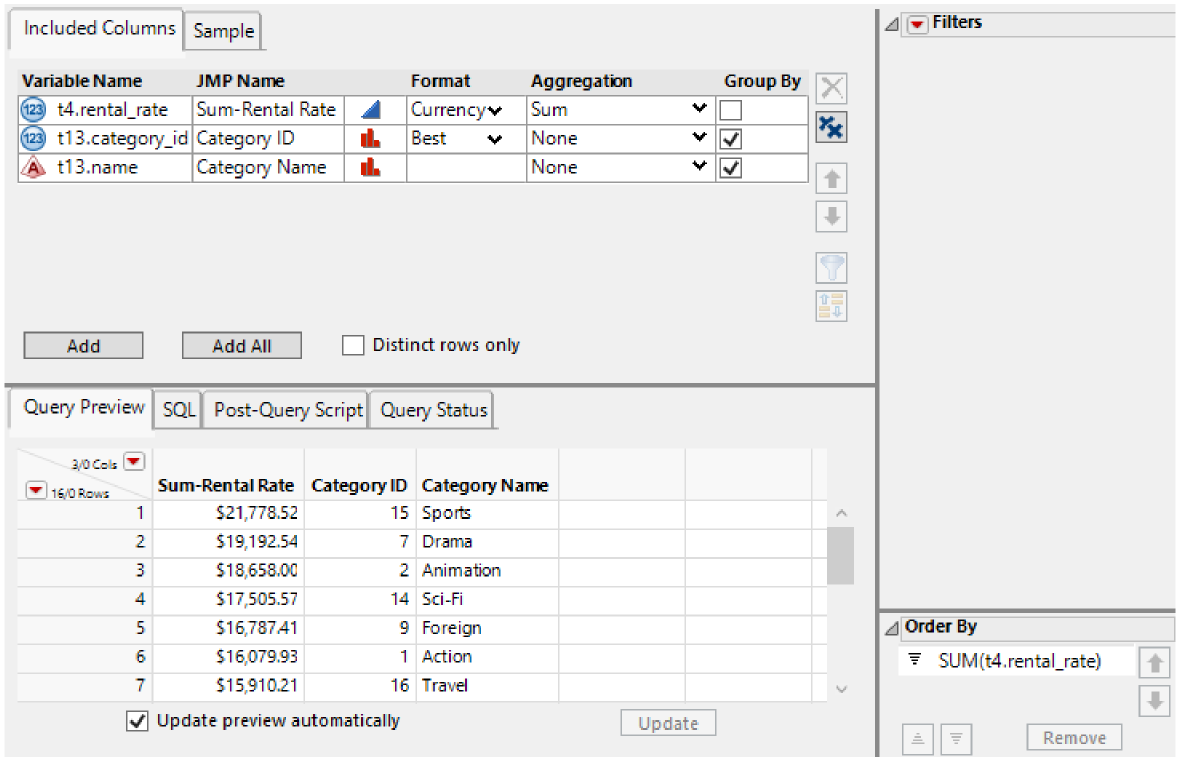


	Rental Rate
1	$2.99
2	$2.99
3	$2.99
4	$2.99
5	$2.99
6	$2.99

Notice that **Group By** becomes checked for all variables that do not have a specified aggregation. Once any column in a query specifies an aggregation, all columns in the query that do **not** specify an aggregation are required to be **Group By** columns.

2. Right-click on the **Variable Name** column for **Rental Rate** and select **Order By**.
3. Select **SUM(t4.rental_rate)** in the **Order By** panel and click the downward-pointing arrow button (**Descending**).

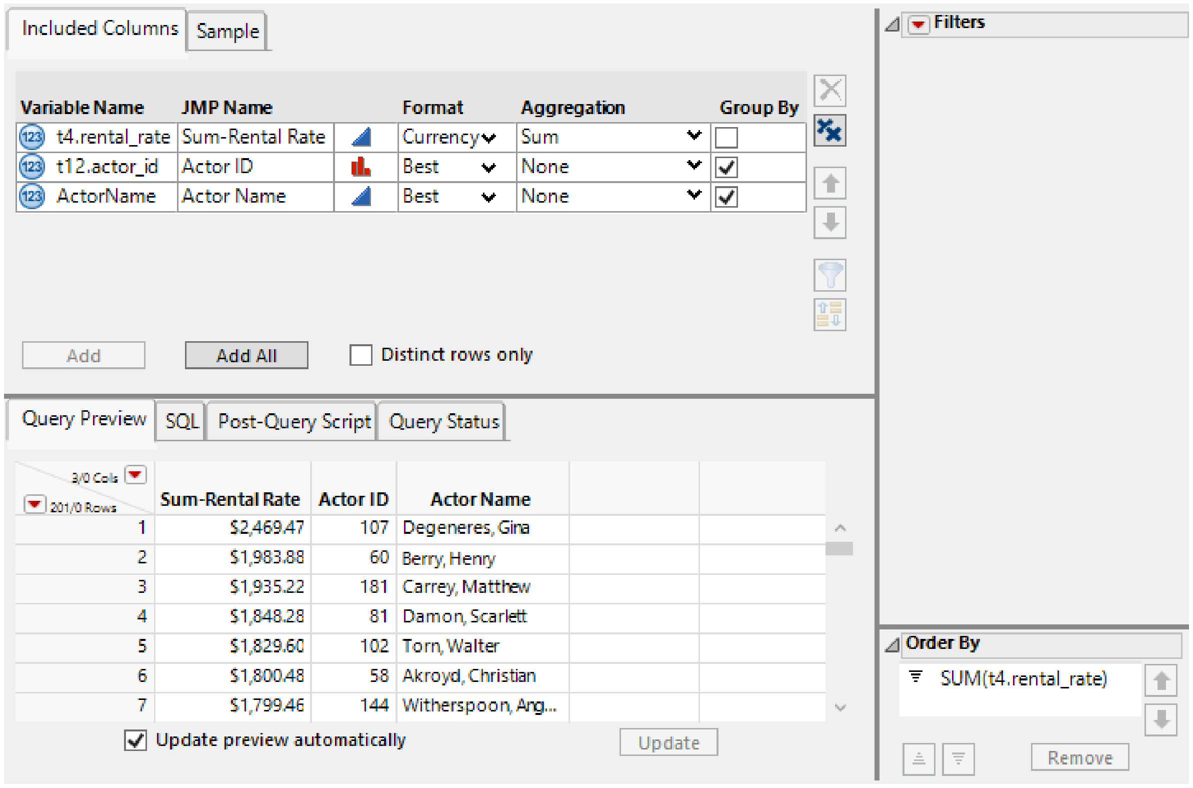
The result looks like this:



Variable Name	JMP Name	Format	Aggregation	Group By
t4.rental_rate	Sum-Rental Rate	Currency	Sum	☐
t13.category_id	Category ID	Best	None	✓
t13.name	Category Name		None	✓

	Sum-Rental Rate	Category ID	Category Name
1	$21,778.52	15	Sports
2	$19,192.54	7	Drama
3	$18,658.00	2	Animation
4	$17,505.57	14	Sci-Fi
5	$16,787.41	9	Foreign
6	$16,079.93	1	Action
7	$15,910.21	16	Travel

Similarly, taking the category columns back out and putting the actor columns back in, I can answer the question of **Which actors are bringing in the most revenue?**:

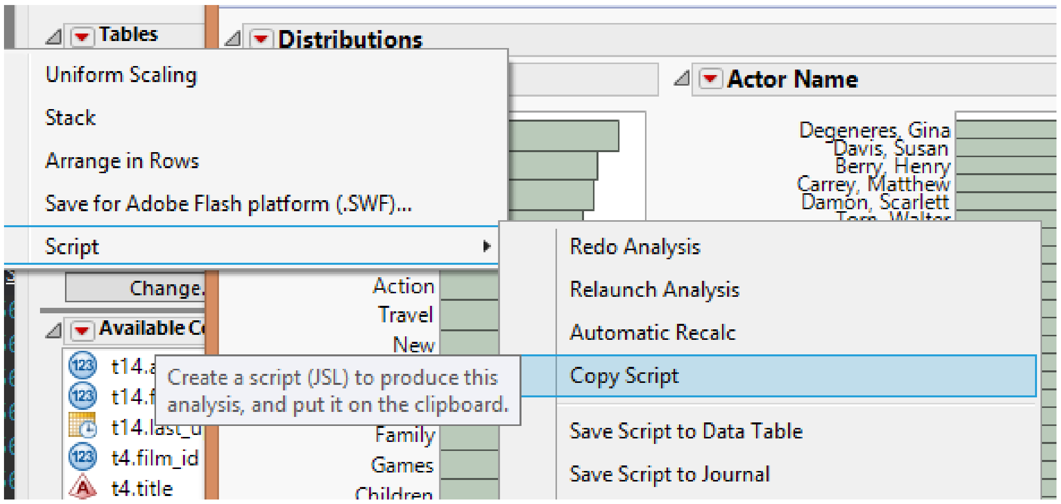


Variable Name	JMP Name	Format	Aggregation	Group By
t4.rental_rate	Sum-Rental Rate	Currency	Sum	☐
t12.actor_id	Actor ID	Best	None	✓
ActorName	Actor Name	Best	None	✓

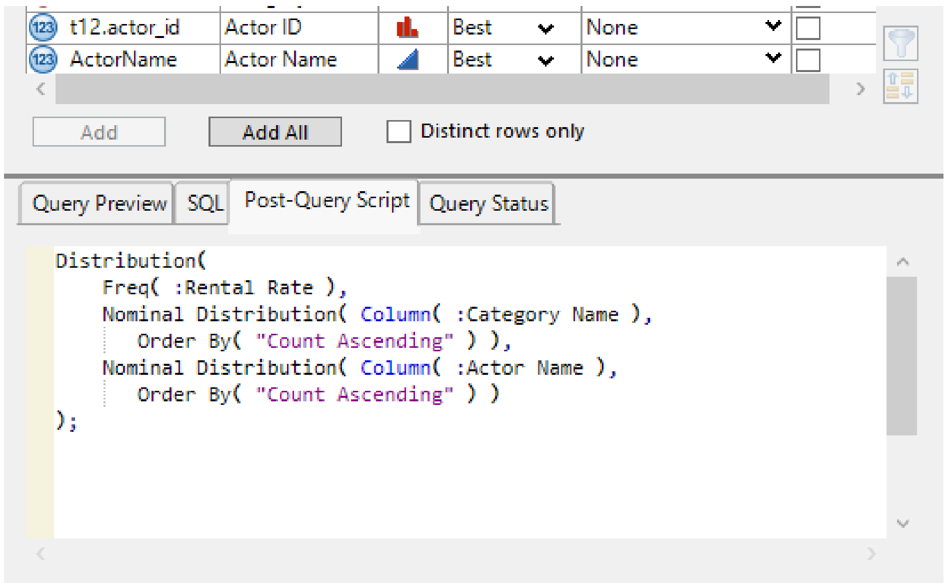
	Sum-Rental Rate	Actor ID	Actor Name
1	$2,469.47	107	Degeneres, Gina
2	$1,983.88	60	Berry, Henry
3	$1,935.22	181	Carrey, Matthew
4	$1,848.28	81	Damon, Scarlett
5	$1,829.60	102	Torn, Walter
6	$1,800.48	58	Akroyd, Christian
7	$1,799.46	144	Witherspoon, Ang...

Filtering

Let's go back a moment to the state where our query was retrieving all of the raw data and we were using the **Distribution** platform to answer our revenue questions. One option we have when doing the analysis in JMP is that we can request the script for the JMP analysis:



I can then paste that script into the **Post-Query Script** panel of **Query Builder**:



```
Distribution(
    Freq( :Rental Rate ),
    Nominal Distribution( Column( :Category Name ),
        Order By( "Count Ascending" ) ),
    Nominal Distribution( Column( :Actor Name ),
        Order By( "Count Ascending" ) )
);
```

Figure 15: Adding a post-query JSL script

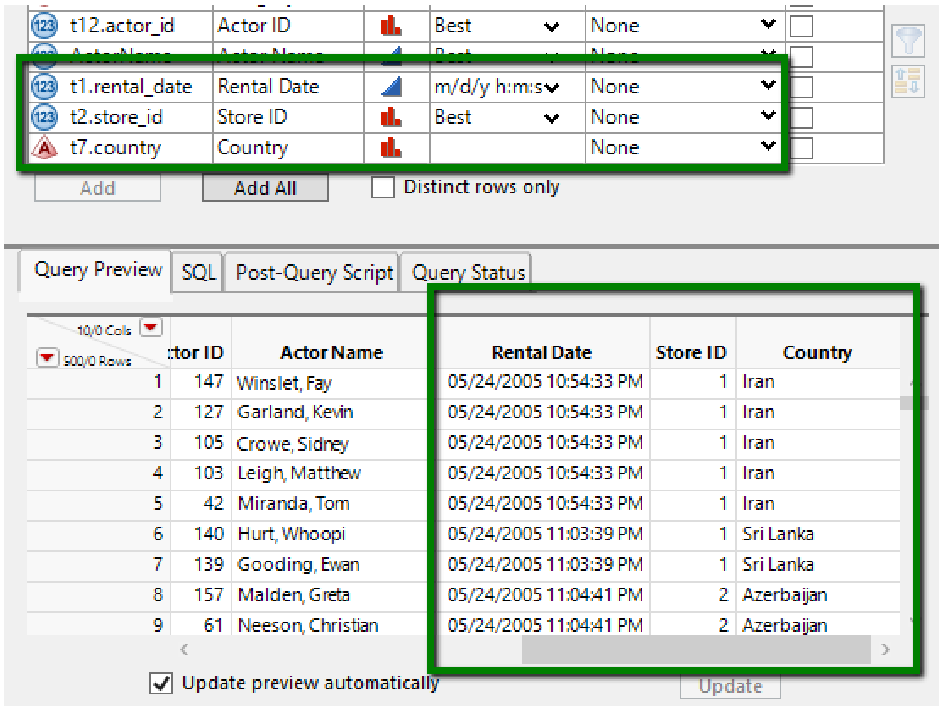
That script will now run each time I run the query.

Now, I would like to refine my questions a bit. Rather than look at category and actor revenue over for all rentals in the database, I may want to look at it based on the country where the movie was rented, the store from which it was rented, or over a certain date range.

The **rental** table has **rental_date**, and **customer_id** fields, so I'm close to having what I need. We saw earlier that **customer_id** is a foreign key reference to the **customer** table. From the **customer** table, we can get the country that the customer is from and the store from which the DVD was rented.

If I click the **Change…** button in the **Tables** panel of Query Builder, I can go back and add more tables to the query. To get country information, I will need to add the **customer**, **address**, **city**, and **country** tables to the query. Fortunately, with the key relationships that the administrator of this database has set up, all of the tables auto-join successfully.

After adding the necessary additional tables, back in Query Builder, I can add the columns I need to the query: **rental_date**, **store_id**, and **country**. I have also set friendly JMP Names and corrected the modeling type of **store_id**:

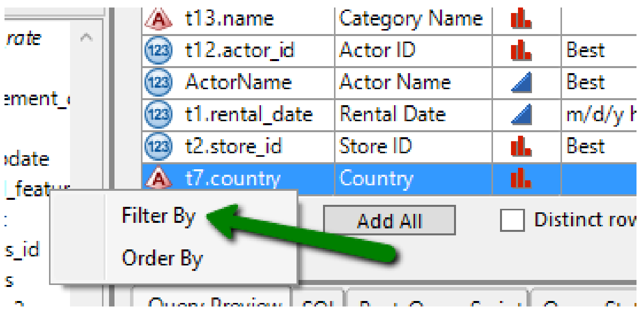


t12.actor_id	Actor ID	Best	None	
ActorName	Actor Name	Best	None	
t1.rental_date	Rental Date	m/d/y h:m:s	None	
t2.store_id	Store ID	Best	None	
t7.country	Country		None	

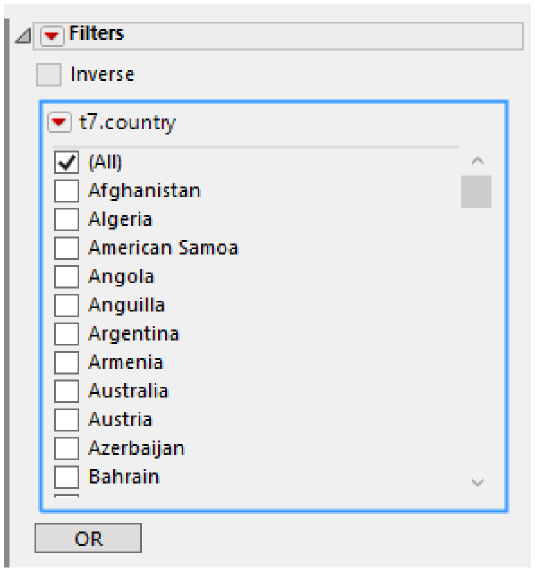
	ctor ID	Actor Name	Rental Date	Store ID	Country
1	147	Winslet, Fay	05/24/2005 10:54:33 PM	1	Iran
2	127	Garland, Kevin	05/24/2005 10:54:33 PM	1	Iran
3	105	Crowe, Sidney	05/24/2005 10:54:33 PM	1	Iran
4	103	Leigh, Matthew	05/24/2005 10:54:33 PM	1	Iran
5	42	Miranda, Tom	05/24/2005 10:54:33 PM	1	Iran
6	140	Hurt, Whoopi	05/24/2005 11:03:39 PM	1	Sri Lanka
7	139	Gooding, Ewan	05/24/2005 11:03:39 PM	1	Sri Lanka
8	157	Malden, Greta	05/24/2005 11:04:41 PM	2	Azerbaijan
9	61	Neeson, Christian	05/24/2005 11:04:41 PM	2	Azerbaijan

Figure 16: After adding Rental Date, Store ID, and Country columns

First, let's filter on the customer's country. I can right-click on the **t7.country** variable in the **Included Columns** panel and select **Filter By**. After doing so, a filter for country appears in the **Filters** panel:



t13.name	Category Name	
t12.actor_id	Actor ID	Best
ActorName	Actor Name	Best
t1.rental_date	Rental Date	m/d/y h
t2.store_id	Store ID	Best
t7.country	Country	


Filters

Inverse

t7.country

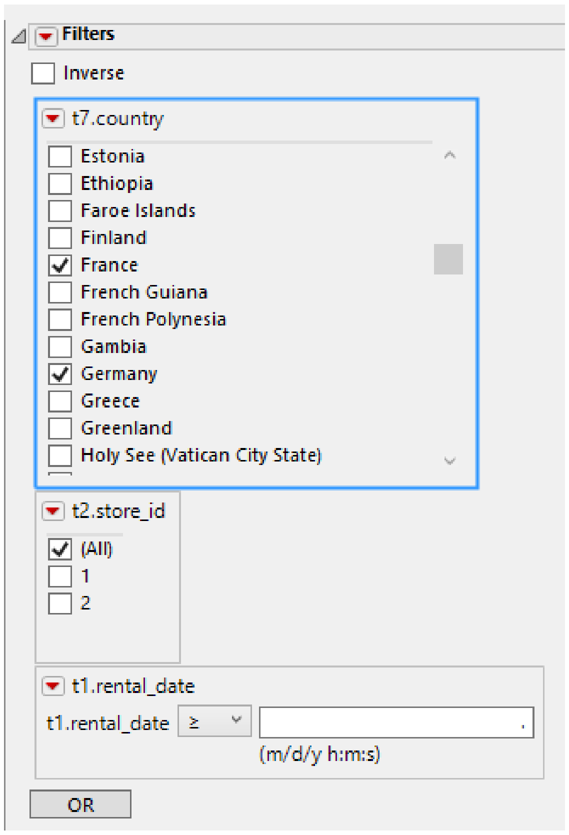
- [x] (All)
- [ ] Afghanistan
- [ ] Algeria
- [ ] American Samoa
- [ ] Angola
- [ ] Anguilla
- [ ] Argentina
- [ ] Armenia
- [ ] Australia
- [ ] Austria
- [ ] Azerbaijan
- [ ] Bahrain

OR

Figure 17: Country filter

At this point, I can simply scroll down through the list of countries and check the ones I am interested in, and then run the query, and only rows from those countries will be included.

Similarly, I can add filters for **store_id** and **rental_date**, after which the Filters panel looks like this:


Filters

Inverse

t7.country

- [ ] Estonia
- [ ] Ethiopia
- [ ] Faroe Islands
- [ ] Finland
- [x] France
- [ ] French Guiana
- [ ] French Polynesia
- [ ] Gambia
- [x] Germany
- [ ] Greece
- [ ] Greenland
- [ ] Holy See (Vatican City State)

t2.store_id

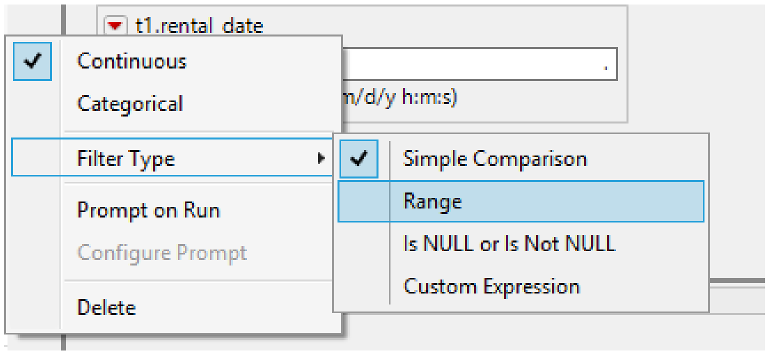
- [x] (All)
- [ ] 1
- [ ] 2

t1.rental_date

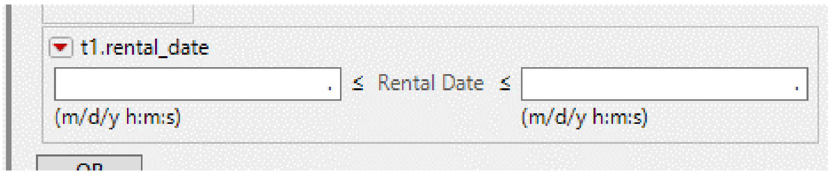
t1.rental_date ≥ (m/d/y h:m:s)

OR

When I add the **rental_date** filter, since it is a continuous variable, I get a **Simple Comparison** filter by default. What I would like instead is a **Range** filter. Using the red triangle menu next to **t1.rental_date**, I can change the filter type to **Range**:



Now the filter for **t1.rental_date** looks like:



allowing me to specify a date range. So, if I want to find all rentals from the second half of 2005, I can do:

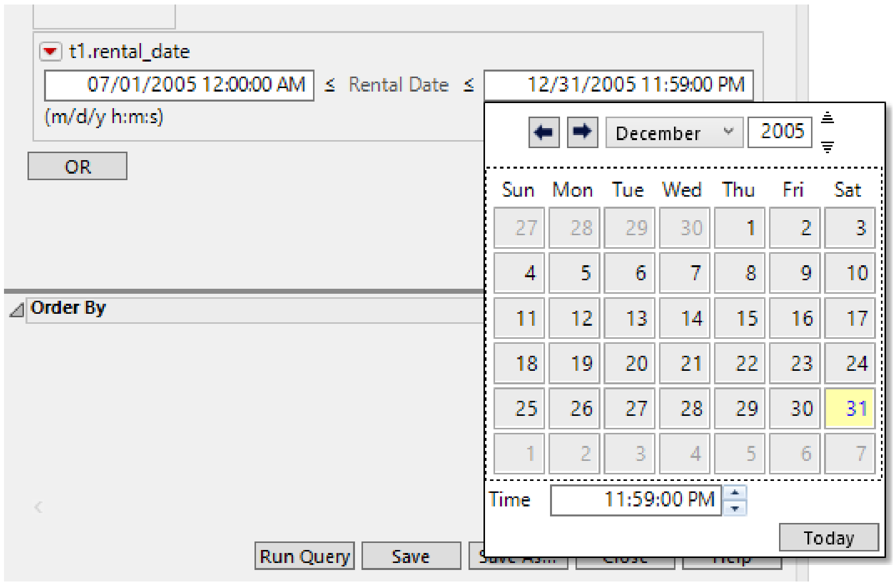
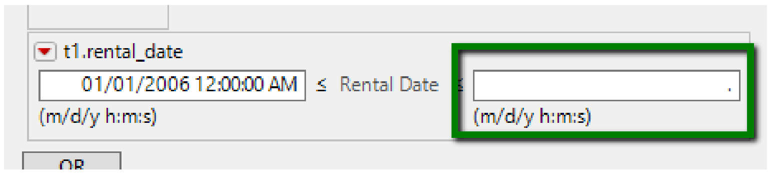


Figure 18: Date Range Filter

All of the filter types in **Query Builder** have a way for you to effectively disable them, allowing you to add several filters to a query even though not all of them may be active for a given query. For categorical filters showing a list of values, selecting the **(All)** item at the top disables the filter. For filters requesting numeric values, setting the value to **missing (.)** disables the filter (or part of the filter). So if I want all rentals from January 1, 2006 forward, I can do this:



Now the **Filters** panel for my query looks like *Figure 19* below.

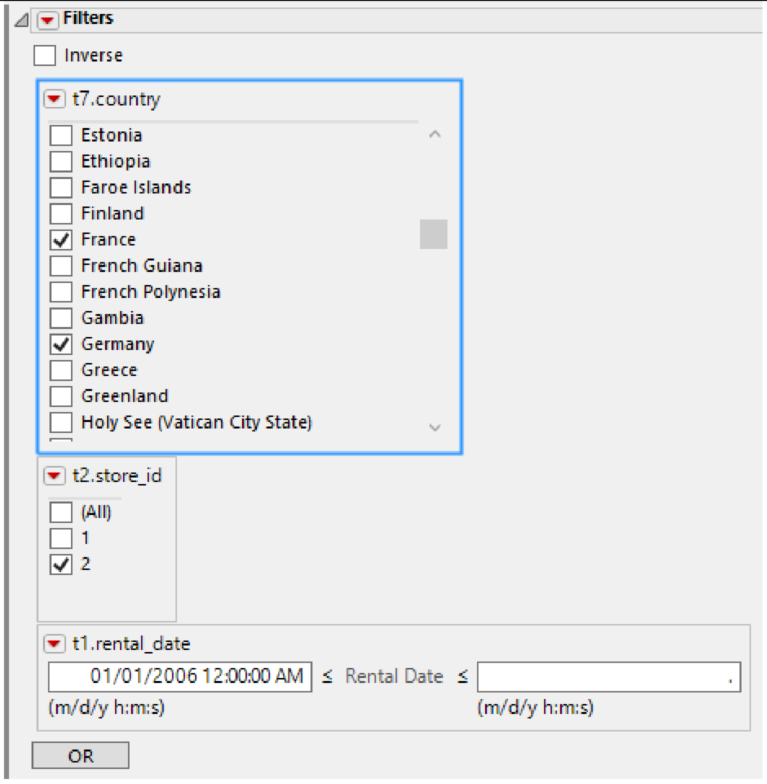


Figure 19: Using Missing to Inactivate Filter

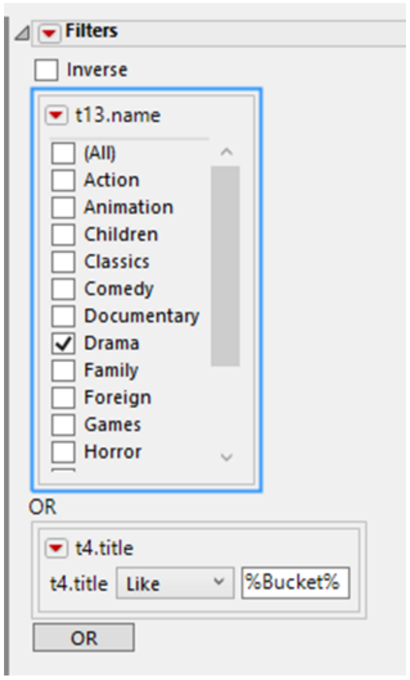
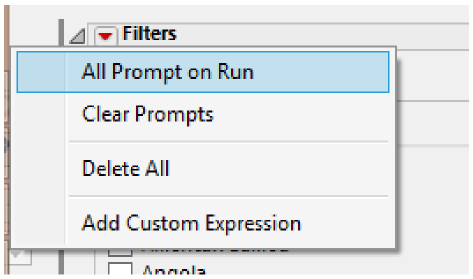


Figure 20: Filter with two OR groups

With the filter selections shown in *Figure 19*, running the query now will return rows where the customer's country is either France or Germany, and the DVD was rented at store 2 on or after January 1, 2006. By default multiple filters are AND'd together, so a row must match all filters to appear in the result. If you need an OR filter, click the **OR** button, which will create another filter group, and the two filter groups will be OR'd together. For example, the filter shown in *Figure 20* above returns rentals where the category is **Drama or** the movie title contains **"Bucket"**.

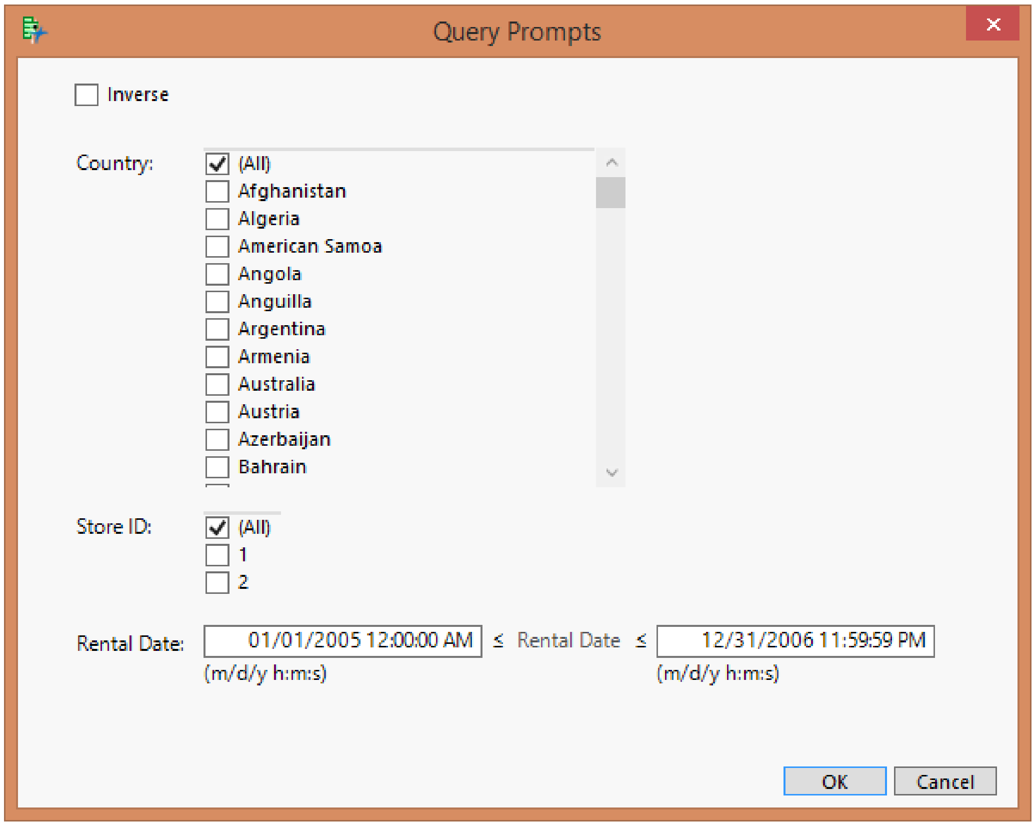
Prompting Filters

While it is nice to be able to specify filter criteria in **Query Builder**, you may also want to create a query to be used by others that don't really need to be presented with the entire **Query Builder** user interface, but you want them to be able to enter different filter criteria each time they run the query. The simplest way to accomplish this is to select **All Prompt on Run** from the red triangle menu next to **Filters**:



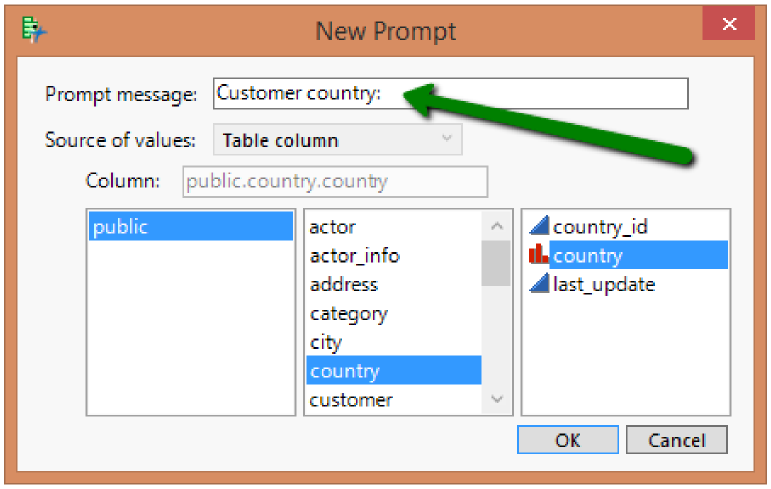
That turns all of the filters in the query into **prompting** filters. If you only want some of the filters to be prompting and not others, you can turn prompting on and off for each individual filter using its red triangle menu.

After setting **All Prompt On Run**, when I run this query, I am presented with a prompt to set the filter values the way I want:



The default settings for the prompt will be whatever was selected in **Query Builder** (or in the saved query file if the query is run directly from the file) before the query was run.

You can also customize the labels used in the prompt. For example, suppose instead of "**Country:**" I wanted the prompt to say "**Customer country:**". Back in Query Builder, using the red triangle menu for the **t7.country** filter, select **Configure Prompt**. That brings up a dialog box allowing you to change the label for the prompt and also change the table that supplies the category levels for the prompt:


New Prompt

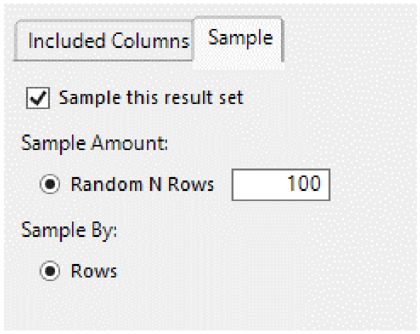
Prompt message: Customer country:

Source of values: Table column

Column: public.country.country

Sampling

If your database supports random sampling, the **Sample** tab in the top center section of **Query Builder** will show the sampling options that your database supports. I've been using PostgreSQL for the examples in this paper, which supports simple random sampling of a specified number of rows:



[✓] Sample this result set

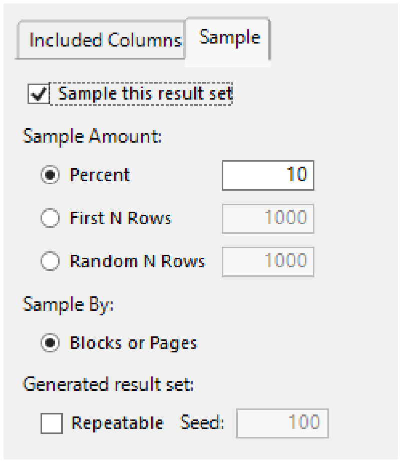
Sample Amount:

[●] Random N Rows 100

Sample By:

[●] Rows

SQL Server and Oracle support different sampling options, so the **Sample** tab will reflect the different options:



[✓] Sample this result set

Sample Amount:

[●] Percent 10

○ First N Rows 1000

○ Random N Rows 1000

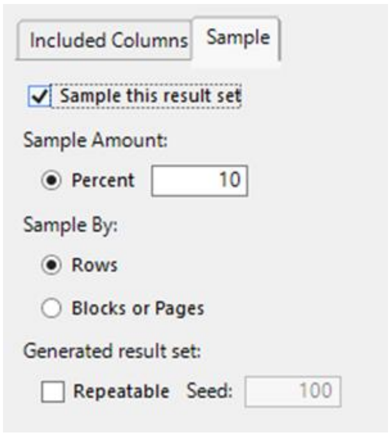
Sample By:

[●] Blocks or Pages

Generated result set:

☐ Repeatable Seed: 100

Figure 21: SQL Server sampling options



[✓] Sample this result set

Sample Amount:

[●] Percent 10

Sample By:

[●] Rows

○ Blocks or Pages

Generated result set:

☐ Repeatable Seed: 100

Figure 22: Oracle sampling options

The sampling is performed **by the database**, before the data is downloaded to JMP, so this is a good way to limit the number of rows retrieved from queries that would otherwise retrieve a prohibitively large number of rows.

Writing Your Own SQL

We recognize that, as many features as **Query Builder** has now or may have in the future, there are times when you may need a query that does things that **Query Builder** does not directly support. Also, you many have already written and painstakingly debugged some SQL that performs exactly the query that you need JMP to perform. **Query Builder** has two options for supplying your own SQL for a query:

1. **Custom Filter Expression** – If **Query Builder** is doing what you need as far as selecting tables and specifying the columns to include, but you just want to have a custom SQL expression included as part of the filter, you can select **Add Custom Expression** from the **Filters** red triangle menu:

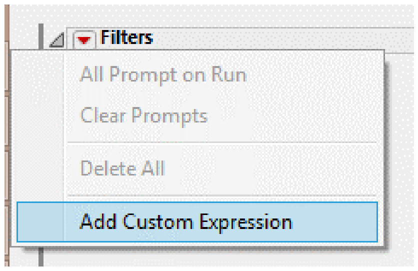
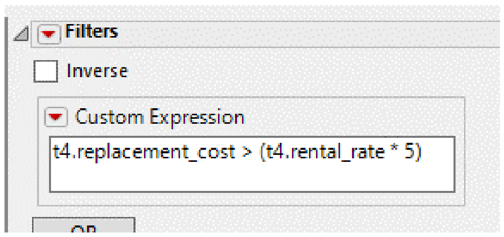


Figure 23: Adding a custom filter expression

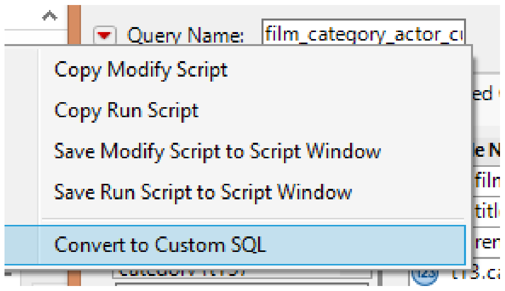


t4.replacement_cost > (t4.rental_rate * 5)

Figure 24: The resulting filter

Enter whatever SQL you want into the **Custom Expression** box, as long as it is valid SQL for the query. If a **Custom Expression** filter is set to be a prompting filter, the custom expression will appear in the filter prompt dialog and be editable.

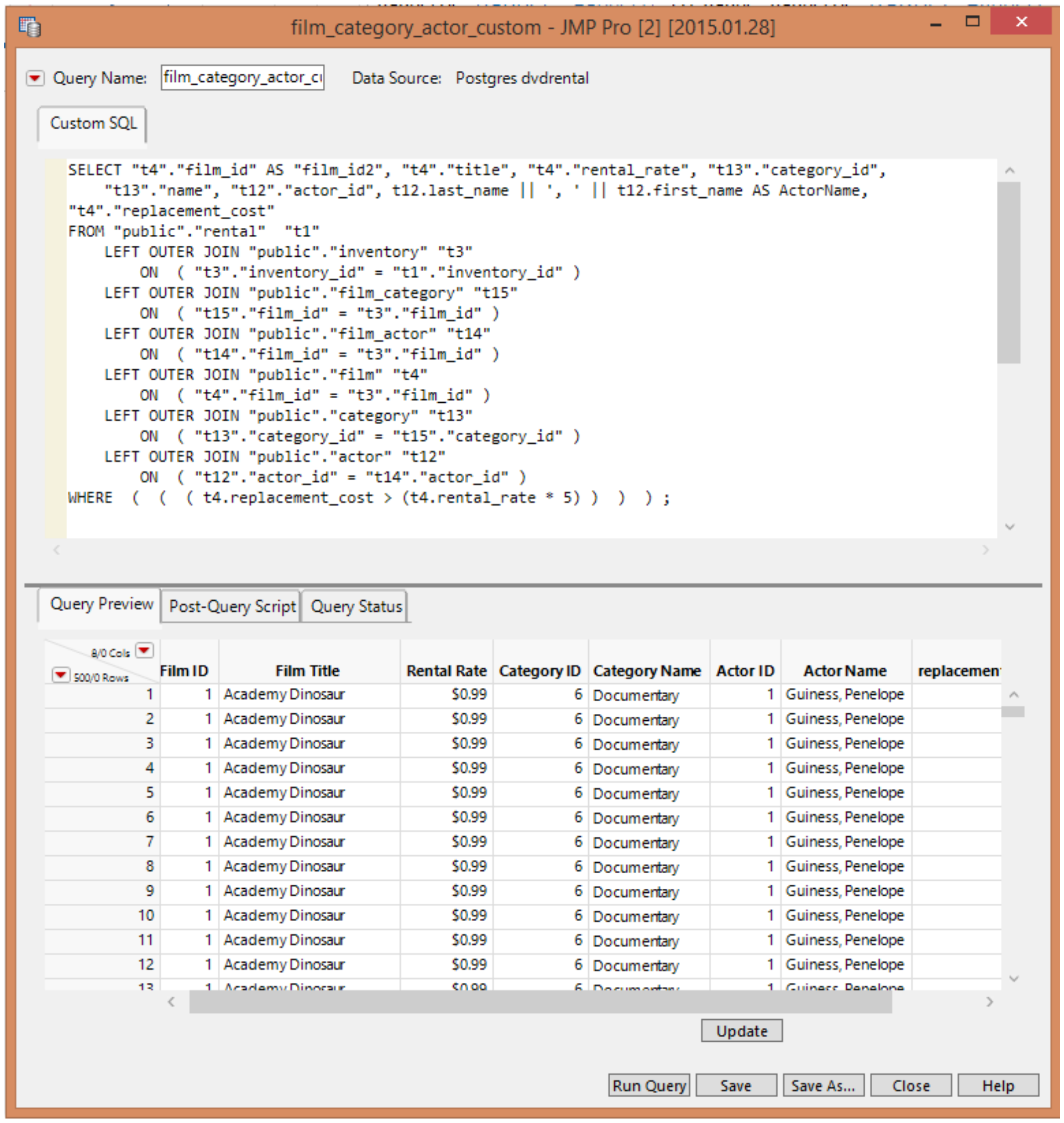
2. Convert the entire query to a **Custom SQL query** – If you just want to write your own SQL for an entire query, then, after launching **Query Builder** and specifying the data connection, you can use the red triangle menu at the top left of **Query Builder** next to the **Query Name** field and select **Convert to Custom SQL**:


Query Name:
Copy Modify Script
Copy Run Script
Save Modify Script to Script Window
Save Run Script to Script Window
Convert to Custom SQL

Query Name: film_category_actor_ci

Figure 25: Convert to Custom SQL

The **Query Builder** window will switch to custom SQL mode, giving you a large area to enter your own SQL. If you had started building the query interactively, the **Custom SQL** box will be initialized with the SQL generated based on the selections you have made so far (see *Figure 26* below).


film_category_actor_custom - JMP Pro [2] [2015.01.28]
Query Name: film_category_actor_c
Data Source: Postgres dvdrental
Custom SQL

```
SELECT "t4"."film_id" AS "film_id2", "t4"."title", "t4"."rental_rate", "t13"."category_id",
    "t13"."name", "t12"."actor_id", t12.last_name || ', ' || t12.first_name AS ActorName,
"t4"."replacement_cost"
FROM "public"."rental"  "t1"
    LEFT OUTER JOIN "public"."inventory" "t3"
        ON  ( "t3"."inventory_id" = "t1"."inventory_id" )
    LEFT OUTER JOIN "public"."film_category" "t15"
        ON  ( "t15"."film_id" = "t3"."film_id" )
    LEFT OUTER JOIN "public"."film_actor" "t14"
        ON  ( "t14"."film_id" = "t3"."film_id" )
    LEFT OUTER JOIN "public"."film" "t4"
        ON  ( "t4"."film_id" = "t3"."film_id" )
    LEFT OUTER JOIN "public"."category" "t13"
        ON  ( "t13"."category_id" = "t15"."category_id" )
    LEFT OUTER JOIN "public"."actor" "t12"
        ON  ( "t12"."actor_id" = "t14"."actor_id" )
WHERE  (  (  ( t4.replacement_cost > (t4.rental_rate * 5) )  )  ) ;
```

	Film ID	Film Title	Rental Rate	Category ID	Category Name	Actor ID	Actor Name	replacemen
1	1	Academy Dinosaur	$0.99	6	Documentary	1	Guiness, Penelope	
2	1	Academy Dinosaur	$0.99	6	Documentary	1	Guiness, Penelope	
3	1	Academy Dinosaur	$0.99	6	Documentary	1	Guiness, Penelope	
4	1	Academy Dinosaur	$0.99	6	Documentary	1	Guiness, Penelope	
5	1	Academy Dinosaur	$0.99	6	Documentary	1	Guiness, Penelope	
6	1	Academy Dinosaur	$0.99	6	Documentary	1	Guiness, Penelope	
7	1	Academy Dinosaur	$0.99	6	Documentary	1	Guiness, Penelope	
8	1	Academy Dinosaur	$0.99	6	Documentary	1	Guiness, Penelope	
9	1	Academy Dinosaur	$0.99	6	Documentary	1	Guiness, Penelope	
10	1	Academy Dinosaur	$0.99	6	Documentary	1	Guiness, Penelope	
11	1	Academy Dinosaur	$0.99	6	Documentary	1	Guiness, Penelope	
12	1	Academy Dinosaur	$0.99	6	Documentary	1	Guiness, Penelope	
13	1	Academy Dinosaur	$0.99	6	Documentary	1	Guiness, Penelope	

Figure 26: Custom SQL query

You can edit the SQL as needed, or paste in SQL from another source, and then you can click the **Update** button to update the **Query Preview**. Some caveats:

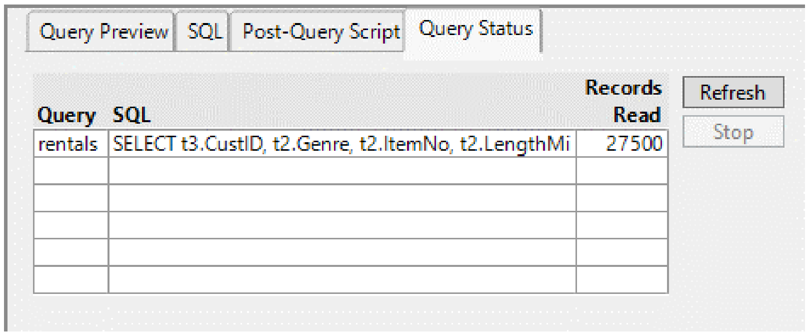
- You cannot do prompted filtering with a custom SQL query.
- Once you convert a query to custom SQL and edit the SQL, if you revert the query back to interactive, you will lose any customizations you have made to the SQL. Query Builder does not attempt to interpret the SQL.

Section 4: Managing Background Queries

Query Builder runs queries in the background by default, and it is also possible from JMP Scripting Language (JSL) to run queries in the background (see **Section 6**). JMP provides two ways for you to view the list of currently running background queries and stop running queries if necessary.

Query Status Panel

In the bottom middle section of Query Builder, there is a **Query Status** tab. Any background queries that were launched **from that Query Builder window** will appear in that tab:

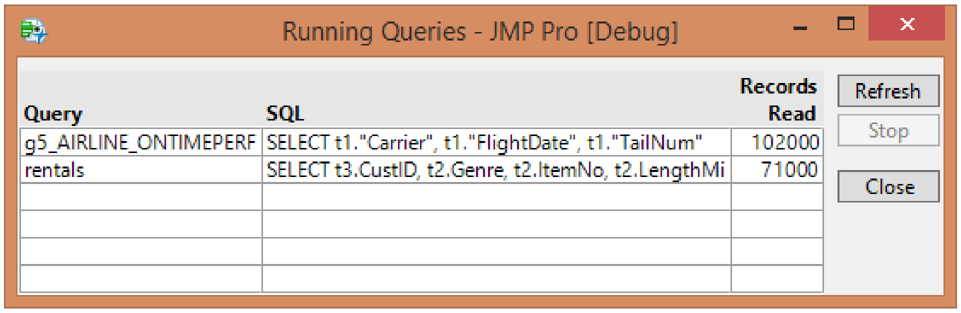


Query	SQL	Records Read
rentals	SELECT t3.CustID, t2.Genre, t2.ItemNo, t2.LengthMi	27500

If you want to stop a running query, simply select it in the list and click the **Stop** button. If some rows have been retrieved, JMP will open a data table containing whatever rows have already been retrieved. The name of the data table will reflect that the result is partial.

Runninq Queries Window

From the **View** menu on the Home Window or other JMP window that includes a menu, select **Running Queries**. This will open a window that looks the same as the **Query Status** tab of **Query Builder**, but the **Running Queries** window will contain **all** queries that are currently running in the background, regardless of how they were invoked (a Query Builder window or JMP Scripting Language):

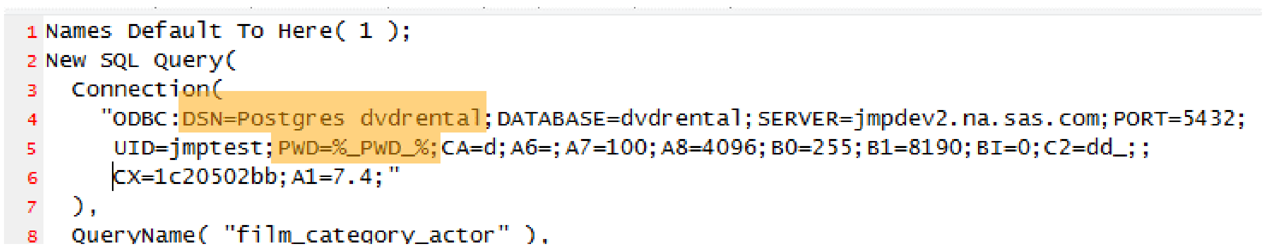


Query	SQL	Records Read
g5_AIRLINE_ONTIMEPERF	SELECT t1."Carrier", t1."FlightDate", t1."TailNum"	102000
rentals	SELECT t3.CustID, t2.Genre, t2.ItemNo, t2.LengthMi	71000

Here again, select a query and click **Stop** to stop it from running, opening any partial result.

Section 5: Sharing Queries

In the introduction, it was mentioned that **Open Table** created scripts that were difficult to share, because the database password is included in the connection string. **Query Builder** works differently. If you save a **.jmpquery** file for a query whose connection required a password, and you open that file in a text editor, you will see something like this:



```
Names Default To Here( 1 );
New SQL Query(
  Connection(
    "ODBC:DSN=Postgres dvdrental;DATABASE=dvdrental;SERVER=jmpdev2.na.sas.com;PORT=5432;
     UID=jmptest;PWD=%_PWD_%;CA=d;A6=;A7=100;A8=4096;B0=255;B1=8190;BI=0;C2=dd_;;
     CX=1c20502bb;A1=7.4;"
  ),
  QueryName( "film_category_actor" ),
```

You can see that the user name (**UID**) is present in the connection string, but then you see **PWD=%_PWD_%**. "%_PWD_%" is not the password; it is a placeholder put there by Query Builder. When Query Builder sees that placeholder, it knows that a password was required originally, so if this query is opened at a time when the user opening it does not already have a connection to the **Postgres dvdrental** DSN, Query Builder will know to prompt the user for the password.

And what about the DSN itself? What if you send this .jmpquery file to a JMP user who does not have a DSN named **Postgres dvdrental** defined on their machine? **Open Table** is unable to proceed in such a scenario, but **Query Builder** will inform the user of this situation and give them a chance to connect to a different DSN (that presumably connects to the same database) or create a new one.

Because of the potential of different users having different DSN's defined on their machines, some organizations have moved to **DSN-less connections**. **Query Builder** cannot currently convert a DSN-based connection string to a DSN-less one, but if you change a connection string in the script for a query so that instead of specifying a **DSN**, it specifies a **DRIVER** and **SERVER**, **Query Builder** will propagate this DSN-less connection in any scripts or .jmpquery files that it generates.

Section 6: Running Queries from JMP Scripting Language (JSL)

Queries built by Query Builder can also be run from **JMP Scripting Language (JSL)** scripts. The best option is to save the query as a **.jmpquery** file, open it from JSL, and run it. Suppose, for example, that you have a query saved in the **c:\users\public\temp** folder named **pg_rentals.jmpquery**. Here are some things you can do with that query file from JSL:

- Open a .jmpquery file into Query Builder:

  ```
  open( "c:\users\public\temp\pg_rentals.jmpquery" );
  ```

- Get the query as a scriptable object without opening it in Query Builder:

  ```
  query = open( "c:\users\public\temp\pg_rentals.jmpquery", Private );
  ```

- Open the query in **Query Builder** for editing:

  ```
  query << Modify;
  ```

- Run the query in the foreground or background, depending on the user's preference setting:

  ```
  query << Run();
  ```

- Run the query in the foreground. The resulting data table will open in JMP when the query finishes. dtResult will be a reference to the result data table.

```
dtResult = query << Run Foreground();
```

- Run the query in the foreground, but do **not** open the resulting data table in JMP; keep it private. dtResult will be a reference to the result data table.

```
dtResult = query << Run Foreground( Private );
```

- Run the query in the background. The resulting data table will open in JMP when the query finishes. Run Background cannot return a reference to the result data table because the query is simply launched in the background. Examples following this one show how to get a reference to the result data table when running queries in the background.

```
query << Run Background();
```

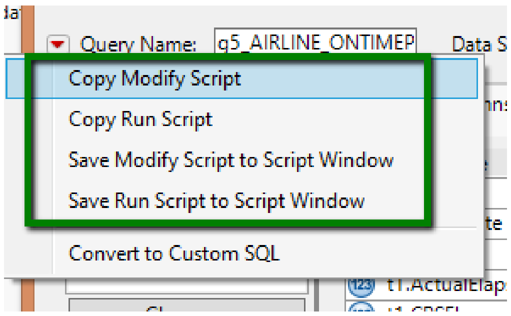
- Run the query in the background. If the query succeeds, invoke the Distribution platform on the result data table (**queryResult** is a special JSL variable within an **On Run Complete** script that refers to the result data table). If the query is canceled by the user or has an error, write a message to the JMP Log.

```
query << Run Background(
        On Run Complete(
                queryResult << Distribution( Nominal Distribution( Column( :title ) ) );
        ),
        On Run Canceled( Write( "\!NThe query was canceled!" ) ),
        On Error( Write( "\!NError running query!" ); )
);
```

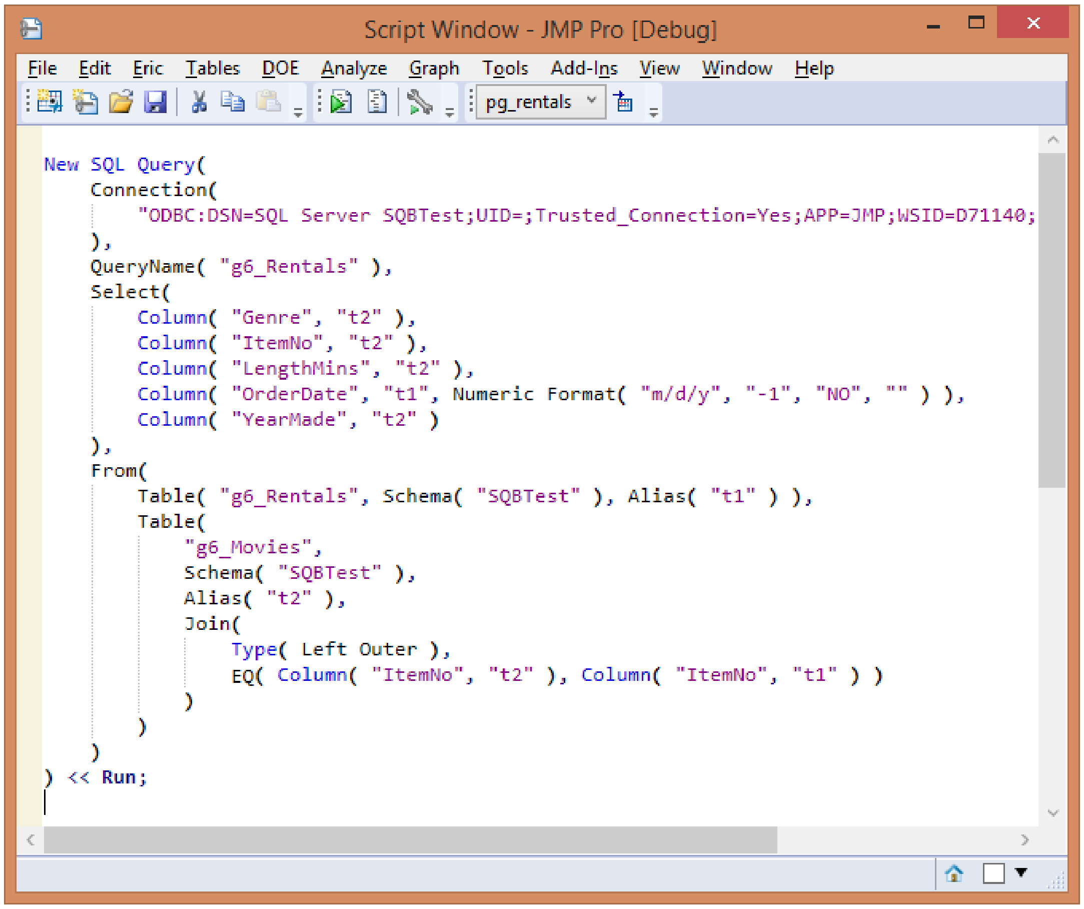
- Run the query in the background. Upon successful completion of the query, invoke the MyFunc function. The query result data table is always passed as the first parameter to the function.

```
::MyFunc = Function( {dt}, {Default Local},
   Write( "\!nNumber of Rentals: ", NRows(dt) );
   dt << Distribution( Nominal Distribution( Column( :title ) ) );
);
query << Run Background( On Run Complete( ::MyFunc ) );
```

If opening a query from a separate .jmpquery file in your JSL script is not satisfactory, you can get the raw script for creating a query scriptable object from the red triangle menu at the top left of the **Query Builder** window:


Query Name: q5_AIRLINE_ONTIMEP
Data S
Copy Modify Script
Copy Run Script
Save Modify Script to Script Window
Save Run Script to Script Window
Convert to Custom SQL
t1.ActualElap

For example, invoking **Save Run Script to Script Window** for a simple query gives me:


Script Window - JMP Pro [Debug]
File Edit Eric Tables DOE Analyze Graph Tools Add-Ins View Window Help
pg_rentals

```
New SQL Query(
    Connection(
        "ODBC:DSN=SQL Server SQBTest;UID=;Trusted_Connection=Yes;APP=JMP;WSID=D71140;
    ),
    QueryName( "g6_Rentals" ),
    Select(
        Column( "Genre", "t2" ),
        Column( "ItemNo", "t2" ),
        Column( "LengthMins", "t2" ),
        Column( "OrderDate", "t1", Numeric Format( "m/d/y", "-1", "NO", "" ) ),
        Column( "YearMade", "t2" )
    ),
    From(
        Table( "g6_Rentals", Schema( "SQBTest" ), Alias( "t1" ) ),
        Table(
            "g6_Movies",
            Schema( "SQBTest" ),
            Alias( "t2" ),
            Join(
                Type( Left Outer ),
                EQ( Column( "ItemNo", "t2" ), Column( "ItemNo", "t1" ) )
            )
        )
    )
) << Run;
```

New SQL Query returns a SQL Query scriptable object just like **Open** does when opening a .jmpquery file and can be used in all of the same ways.

Conclusion

Query Builder is a new feature in JMP 12 that helps you build highly customized multi-table SQL queries against your relational databases. The goal is to make it easier for you to get your data into JMP in analysis-ready form more quickly than was possible before and to make it easier to share your queries securely. We hope you will try it out and send us feedback (eric.hill@jmp.com) on what works well and what doesn't work for you.