IBM InfoSphere Streams 4.x introductory hands-on lab

for Quick Start Edition VMware image 4.0.1

July 2015
Robert Uleman

Contents

Overview

IBM InfoSphere Streams (*Streams*) enables continuous and fast analysis of possibly massive volumes of moving data to help improve the speed of business insight and decision-making. InfoSphere Streams provides an execution platform and services for user-developed applications that ingest, filter, analyze, and correlate the information in streaming data.

InfoSphere Streams includes the following major components:

Streams Runtime Engine – A collection of distributed processes that work together to facilitate the running of stream processing applications on a given set of host computers (“resources”) in a cluster. A single instantiation of these is referred to as a Streams *instance*. Resources and instances with the same security settings are managed in the context of a Streams *domain*.

Streams Processing Language (SPL) – Declarative and procedural language and framework for writing stream-processing applications. (This lab does not cover direct SPL programming.)

Streams Studio (*Studio*) – Eclipse-based Development Environment for writing, compiling, running, visualizing, and debugging Streams applications. Development tooling supports graphical (“drag & drop”) design and editing, as well as quick data visualization in a running application. The Instance Graph feature provides different views into the applications running within a Streams instance, including the display of live data flow metrics and numerous color highlighting schemes for quick understanding and diagnostics of data flows.

Streams Console – A web-based graphical user interface that is provided by the Streams Web Service (SWS), one of the runtime services. You can use the Streams Console to monitor and manage your Streams instances and applications from any computer that has HTTPS connectivity to the server that is running SWS. The Streams Console also supports data visualization in charts and tables.

Streamtool – Command-line interface to the Streams Runtime Engine. (This lab does not use the command-line interface.)

Goal and objectives

The goal of this lab is to:

> Provide an opportunity to interact with many of the components and features of Streams Studio while developing a small sample application. While the underlying SPL code is always accessible, this lab requires no direct programming and will not dwell on syntax and other features of SPL.

The objectives of this lab (along with the accompanying presentation) are to:

- Provide a basic understanding of stream computing
- Provide a brief overview of the fundamental concepts of InfoSphere Streams
- Introduce the InfoSphere Streams runtime environment
- Introduce InfoSphere Streams Studio
    - A brief introduction to navigating the IDE
    - Create and open projects

- Submit and cancel jobs
- View jobs, health, metrics, and data in the Instance Graph

- Introduce the Streams Studio graphical editing environment
  - Apply several built-in operators
  - Design your first SPL application and enhance it
- Introduce the data visualization capabilities in the Streams Console

Introduction

This hands-on lab provides a broad introduction to the components and tools of InfoSphere Streams.

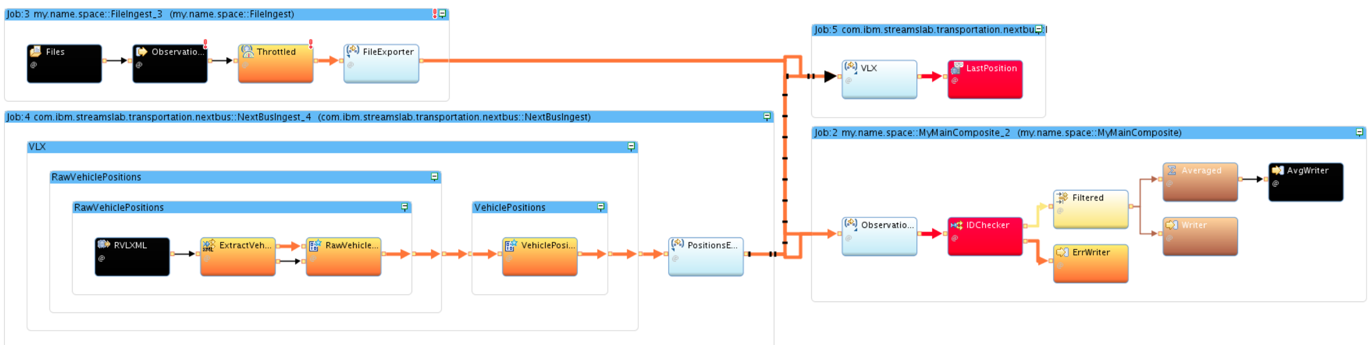


Figure 1. InfoSphere Streams lab overview

The lab is based on a trivial example from a *connected-car* automotive scenario: handling vehicle locations and speeds (and other variables). This is used to illustrate a number of techniques and principles, including:

- Graphical design of an application graph and using a Properties view to configure program details
- Runtime visualization of Streams applications and data flow metrics
- Application integration using importing and exporting of data streams based on stream properties.

Figure 1 shows a graphical overview of the completed lab. The lab environment includes five Eclipse *workspaces* (directories) numbered 1 through 5. Labs build on one another, but each workspace already contains a project with what would be the results of the previous lab. (Workspace 1 is empty; workspace 5 has the final result but there are no instructions for modifying it.) This way, you can experiment and get yourself in trouble any way you like in any lab and still go on to the next lab with everything in place, simply by switching workspaces.

The lab is broken into four parts:

Lab 1 – A simple Streams app. Start a Streams instance. Open Streams Studio; explore its views. Create an SPL application project; create an application graph with three operators; configure the operators. Run the application and verify the results.

Lab 2 – Enhance the app: add the ability to read multiple files from a given directory and slow down the flow so you can watch things happen. Learn about jobs and PEs. Use the Instance Graph to monitor the stream flows and show data.

Lab 3 – Enhance the app: add an operator to compute the average speed every five observations, separately for two cars. Add another operator to check the vehicle ID format and separate records with an unexpected ID structure onto an error stream. Use the Streams Console to visualize results.

Lab 4 – Enhance the app: Use exported application streams to create a modular application. If possible (internet access required), bring in live data. Show the live and simulated location data on a map.

Lab environment

This version of the lab instructions is based on the InfoSphere Streams 4.0.1 Quick Start Edition (QSE) VMware Image, version 1 (vmware-streamsV4.0.1-qse-v1.zip). The following is already installed on that VMware image:

- Red Hat Enterprise Linux 6.5 (64-bit)
- IBM InfoSphere Streams QSE 4.0.1.0, including Streams Studio
- V4Explorer, a sample project and instructions for exploring what's new in 4.0
- Apache Ant (1.9.2; 1.8 or later is required) and Apache Maven (3.2.3; 3.2 or later), two utilities required for building one of the components used in Lab 4.

Table 1. Streams 4.x lab, QSE edition – virtual machine information.

Parameter	Value
Host name	`Streamsqse (streamsqse.localdomain)`
User and administrator ID	`streamsadmin`
User home directory	`/home/streamsadmin`
User password	`passw0rd` (password with a zero for the O)
root password	`passw0rd`
Streams domain	`StreamsDomain` (started automatically)
Streams instance	`StreamsInstance`

The lab workspaces, data files, and additional toolkits need to be installed separately; instructions follow in the next section.

If you're not using the Quick Start Edition VMware image

You can install and run the lab in any other environment with a Streams 4.0.1 installation, but some things may look different, depending on how closely your environment matches the description above. User ID and home, domain and instance names, and operating system version may be different; and you may have to install **Ant** or **Maven** using a method appropriate for your system.

Install the lab files

1. Open your browser to the Introductory Streams Lab page on Streamsdev: **double-click** on the Hands-On Lab launcher on the desktop.
   (For non-QSE users: the URL is https://developer.ibm.com/streamsdev/docs/streams-lab-introduction/.)

2. Follow the instructions on that page.

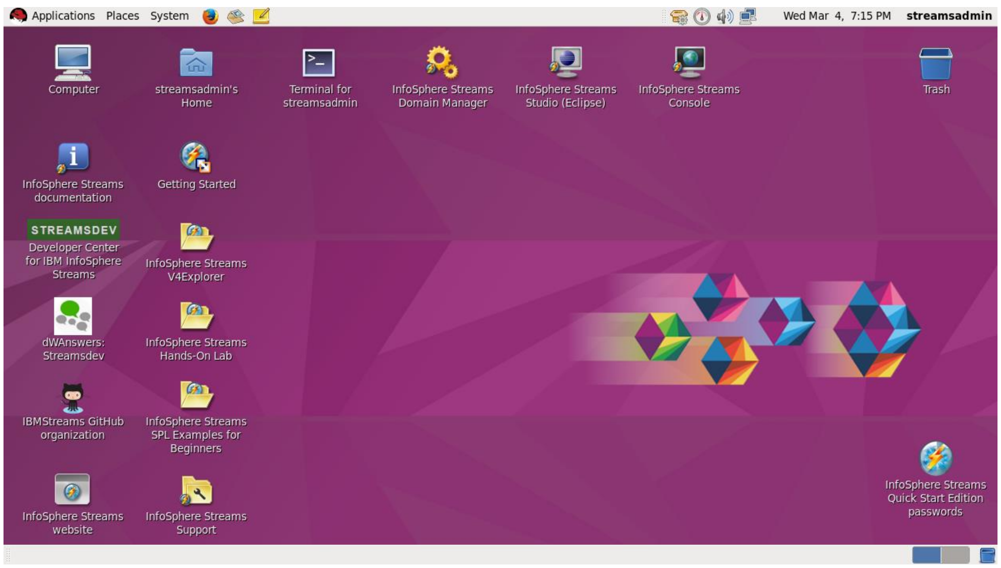
Lab 1 A simple Streams application

In this lab, you will develop an application for a simple scenario: read vehicle location, speed, and other sensor data from a file, look for observations of a specific few vehicles, and write the selected observations to another file. Even though this is not really streaming data, you will see why file I/O is important in Streams application development and, later, how it can be used to simulate streaming data.

Start a Streams Instance. Open Streams Studio; explore its views (windows). Create an SPL application project; create an application graph with three operators; configure the operators. Run the application and verify the results.

1.1 Getting started

The VMware image is set up to log you in automatically upon startup, as user **streamsadmin**. After login you are presented with a desktop containing several icons.



In Streams, a *domain* is a logical grouping of resources in a network for the purpose of common management and administration. A domain can contain one or more instances that share a security model. In the VMware image, a basic domain, **StreamsDomain**, comprising the single host that is your virtual machine, has already been created; it is started automatically when you start up the image. Starting a domain starts a small number of Linux services (*daemons*) for tasks such as authentication and authorization, auditing, supporting the Streams Console, etc. This lab will not explore domain creation and administration further.

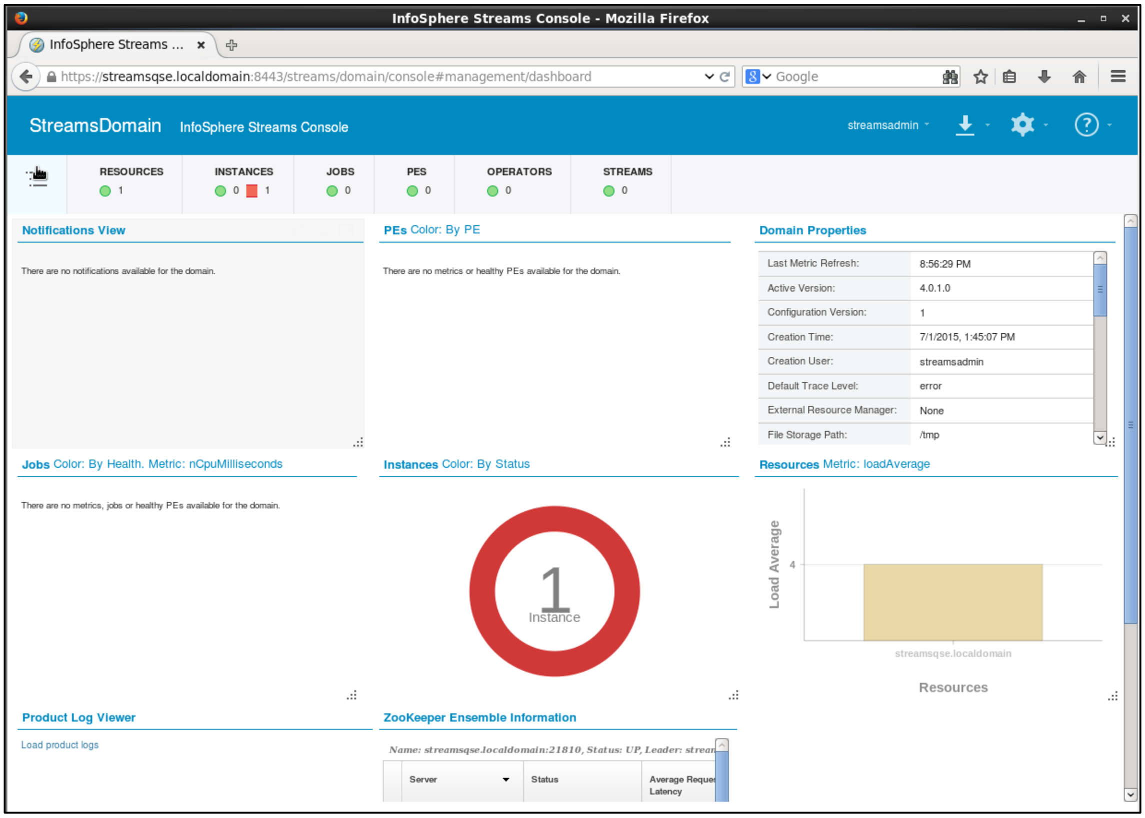
Start a previously created Streams instance. This prepares the runtime environment so you can submit applications to run. The instance is a small number of additional services running in the context of a domain; this includes a resource manager, an application manager, a scheduler, etc.

__1. Double-click the **InfoSphere Streams Console** desktop launcher.
This opens the Firefox web browser.

__a. In the **Untrusted Connection** page, which may come up once, click **I Understand the Risks**, and then **Add Exception… > Confirm Security Exception**. (This is not a bug.)

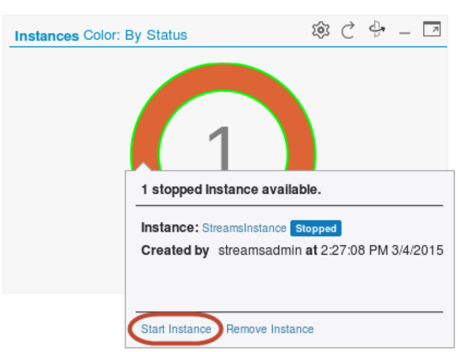
__b. Enter **User name** `streamsadmin` and **Password** `passw0rd`. Click **Log In**.

The **StreamsDomain** Console page opens in the browser, showing several views (*cards*), some with nothing to display (Notifications, PEs, Jobs, Product Logs), two with a table (Domain Properties and Zookeeper Ensemble Information), and two with a graphical display (Resources and Instances). You may have to scroll the page to see them all. The Instances card shows that there is one instance; the red ring indicates that it is not running.



All of these cards and their layout are configurable. For now, you will only use the console to start the required instance.

__2. Hover over the ring in the Instances card. In the hover popup, click **Start Instance**.



Messages pop up briefly in the bottom-right corner of the browser window, the **Notifications** card (top left) shows several entries, and the ring in the **Instances** card turns from red to yellow and then to green, indicating that the **StreamsInstance** instance is now running and healthy. The runtime environment is ready to run your applications.

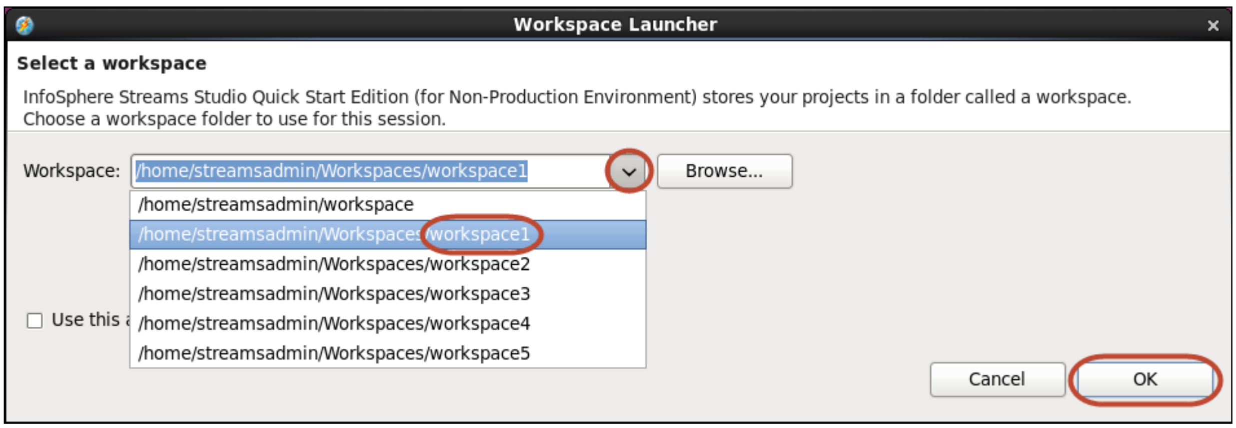
__3. Close or minimize the browser.

1.2 Starting Streams Studio

It is time to start Studio, explore some of its features, and create your first application.

__4. Double-click the **Streams Studio** desktop launcher.

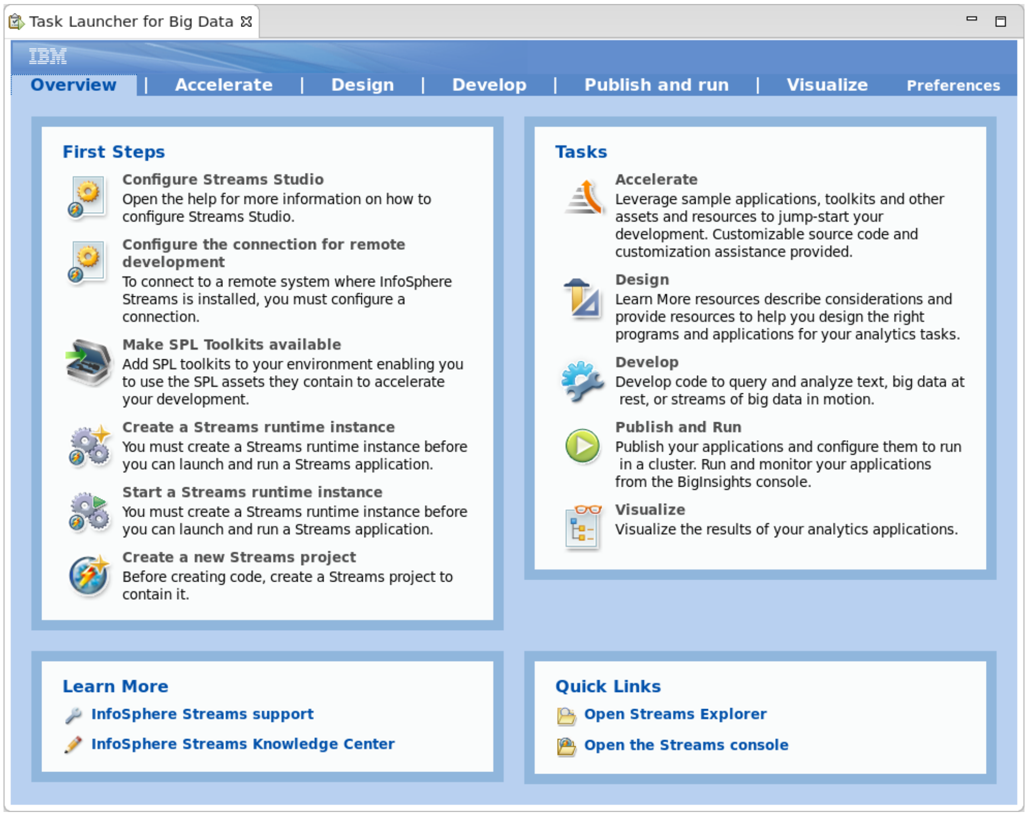
In the **Workspace Launcher**, use the dropdown to select `/home/streamsadmin/Workspaces/workspace1`, and click **OK**.



The Eclipse-based Studio application opens. The Eclipse window is divided into multiple *views* or subwindows, some of which contain multiple *tabs*. On the left are two tabs for the **Project Explorer** and the **Streams Explorer**; in the center the editors (for code and graphs) and the **Task Launcher** (see next section); on the right an **Outline** view (showing nothing until you bring up an SPL source file in an editor), and at the bottom a tabbed window with a **Console** view and two others. The entire arrangement is called a *perspective* in Eclipse; this is the **Streams Studio perspective**.

1.3 The Task Launcher for Big Data

Streams provides a set of task-oriented guides that help developers and other users get started quickly. One such guide is the **Task Launcher for Big Data**, which comes up by default when you open Studio.



__5. Take some time to explore the various tabs. In particular, the links in the **Learn More** group on each of the tabs (scroll down or maximize the view to see it) bring up a separate **Help** view in Studio, as do some of the other tasks.

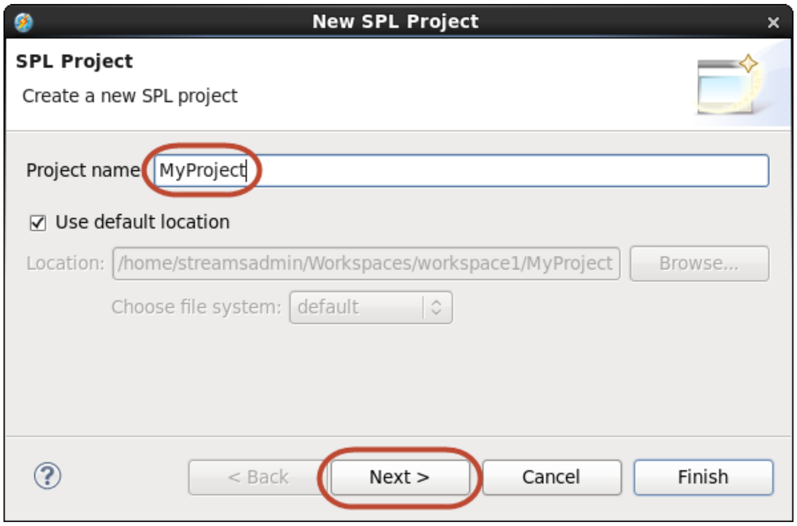
Most of the tasks under **First Steps** have already been completed; these tasks also appear at the end of the Streams installation process (outside Studio), in a separate **First Steps** utility.

> **Task Launcher preferences**
> Like almost everything in Studio, you can control the Task Launcher by setting preferences—whether it shows up at startup, whether it prompts you when you close it, etc. Advanced users will not be distracted by a feature intended to ease the learning curve for new users.

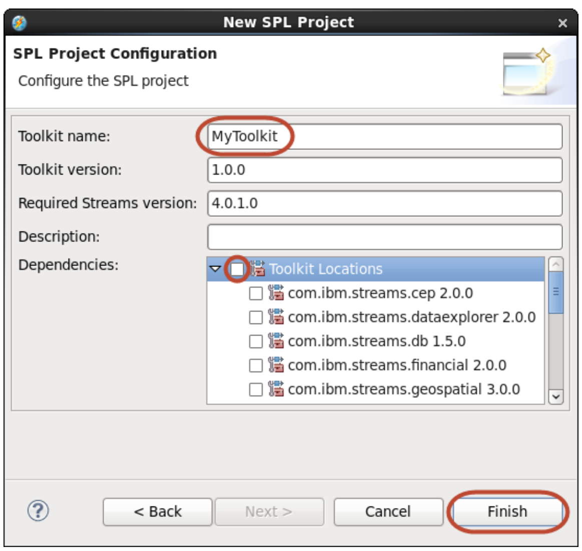
You will start two tasks from the Task Launcher: first you'll create a Streams project (a collection of files in a directory tree in the Eclipse workspace) and then an application in that project.

__6. Click **Create a new Streams project**; this task appears in the **Overview** tab, as well as in the **Develop** tab, under **Quick Links**.

__7. In the **New SPL Project** wizard, enter the **Project name** `MyProject`; keep **Use default location** checked. Click **Next >.**



__8. On the **SPL Project Configuration** panel, change the **Toolkit name** to `MyToolkit`; uncheck **Toolkit Locations** in the **Dependencies** field. Leave the rest unchanged. Click **Finish**.

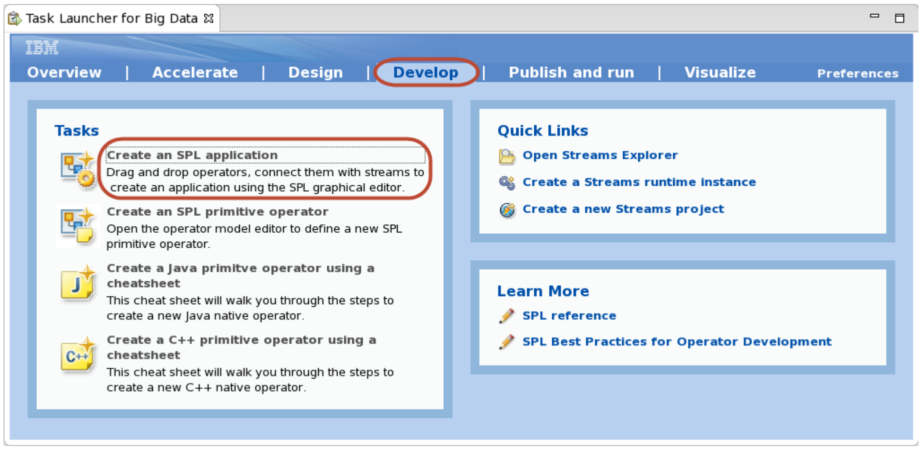


Project dependencies

In the **Dependencies** field you can signal that an application requires operators or other resources from one or more toolkits—a key aspect of the extensibility that makes Streams such a flexible and powerful platform. In this lab, you will only use building blocks from the built-in Standard Toolkit.

The new project shows up in the **Project Explorer** view, on the left.

__9. Go to the **Develop** tab of the Task Launcher for Big Data; click **Create an SPL application**.



The **New Main Composite** wizard comes up, along with a **Help** view. This wizard takes care of a number of steps in one pass: creating a namespace, an SPL source file, and a main composite.

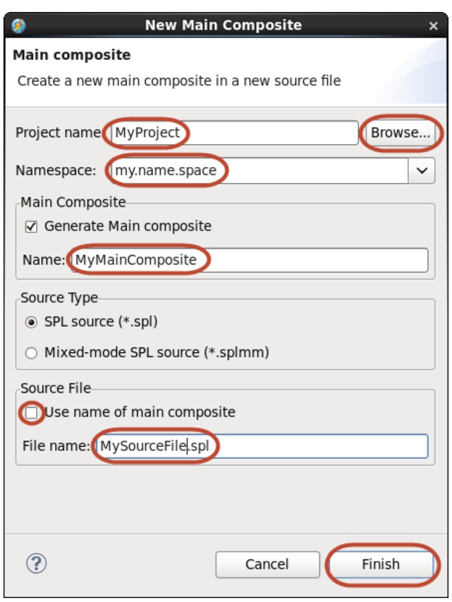
Main composite

A main composite is equivalent to an application. Usually, each main composite lives in its own source file (of the same name), but this is not required.

__10. In the **New Main Composite** wizard, enter

Project name:	`MyProject` (use the **Browse…** button)
Namespace:	`my.name.space`
Main Composite Name:	`MyMainComposite` (keep **Generate Main Composite** checked)
Source Type:	SPL source (*.spl)
Source File name:	`MySourceFile.spl` (uncheck **Use name of main composite**)

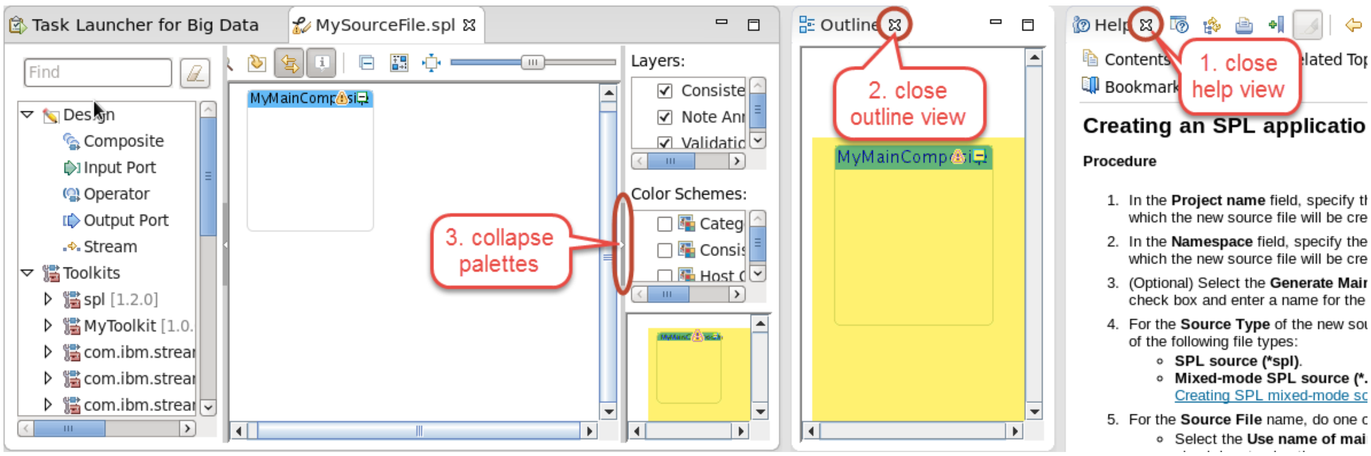
Click **Finish**.



The code module **MySourceFile.spl** opens in the graphical editor, with an empty **MyMainComposite** in the canvas.

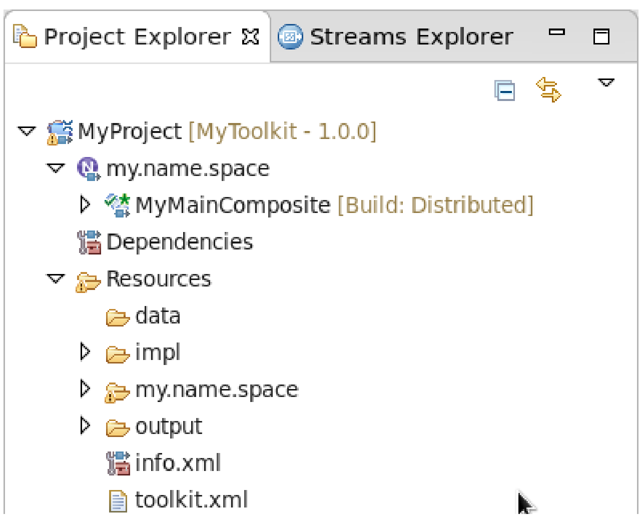
Hint

If you want to give the editor more space, close the **Help** view on the right. You can also close the **Outline** view and collapse the **Layers** and **Color Schemes** palettes (this lab does not use them).



1.4 The Project Explorer view

Let's take a quick tour of one of the supporting Studio views. The **Project Explorer** shows both an object-based and a file-based view of all the projects in the workspace.



__1. In **Project Explorer**, expand **MyProject** by clicking the *twisty* (white triangle) on the left.

The next level shows namespaces and resources (directories and files). (Ignore the elements below Resources; they are only relevant for projects that create Java operators or functions.)

__2. Under **MyProject**, expand the namespace **my.name.space**.

The next level shows main composites; other objects, such as types and functions, if you had any, would appear there as well.

__3. Under **my.name.space**, expand the main composite **MyMainComposite**.

The next level shows build configurations; here, there is only one, **Distributed**, created by default. You can create multiple builds, for debug, for standalone execution, and other variations.

__4. Expand **Resources**.

The next level shows a number of directories, as well as two XML files containing descriptions of the current application or toolkit. (In Streams, toolkit and application are the same in terms of project organization and metadata.) The only directory you will use in this lab is the data directory: by default, build configurations in Streams Studio specify this as the root for any relative path names (paths that do not begin with "/") for input and output data files.

Default data directory

Beginning with version 4.0, Streams applications do not have a default data directory unless you explicitly set one in the build specification. Here, we are simply taking advantage of a feature of Streams Studio, which will provide that specification by default. It works, because we only have a single host.

Because Streams is a distributed system that does not require a shared file system, you have to be careful when specifying file paths. A process accessing a file must run on a host that can reach it; in general this means specifying absolute paths and constraining where a particular process can run; using relative paths and a default data directory makes the application less portable.

1.5 Developing an application in the graphical editor

You are now ready to begin building the simple application. The high-level requirements are:

- Read vehicle location data from a file
- Filter vehicle location data by vehicle ID
- Write filtered vehicle location data to a file

More detailed instructions (simply read these; step-by-step instructions follow):

- Design the application graph in the graphical editor
- Use three operators (an operator is the basic building block of an application graph): one each to read a file (**FileSource**), write a file (**FileSink**), and perform the filtering (**Filter**)
- Take it step by step:
  - Define a new type for location data, to serve as the **schema**, or **stream type**, of several streams
    - A schema defines the *type*, or structure, of each packet of data, or *tuple*, flowing on the stream. The tuple type is a list of named *attributes* and their types. Think of a tuple as the Streams analogue of a row in a table or a record in a file, and of its attributes as the analogue of columns or fields.
  - Drop the required operators into the editor

- Make the required stream connections to “wire up” the graph
- Define the schema of the streams in the graph
- Specify the parameters and other details of the individual operators

Table 2. Additional specifications: stream type for vehicle location data

Name	Type	Comment
`id`	`rstring`	Vehicle ID (an *rstring* uses “raw” 8-bit characters)
`time`	`int64`	Observation timestamp (milliseconds since 00:00:00 on January 1, 1970)
`latitude`	`float64`	Latitude (degrees)
`longitude`	`float64`	Longitude (degrees)
`speed`	`float64`	Vehicle speed (km/h)
`heading`	`float64`	Direction of travel (degrees, clockwise from north)

While this lab is only intended as introductory exploration, leaving out any considerations of SPL syntax and coding best practices, one such practice is worth illustrating: rather than defining the schema (stream type) of each stream separately in the declaration of each stream, create a type ahead of time, so that each stream can simply refer to that type. This eliminates code duplication and improves consistency and maintainability.

Therefore, begin by creating a type for the vehicle location data that will be the main kind of tuple flowing through the application, using the specifications in Table 2, above. Call the new type **LocationType**.

__1. In the graphical editor for **SourceFile1.spl**, right-click anywhere on the canvas, *outside the main composite (MyMainComposite)*, and choose **Edit**. (Double-click has the same effect.)

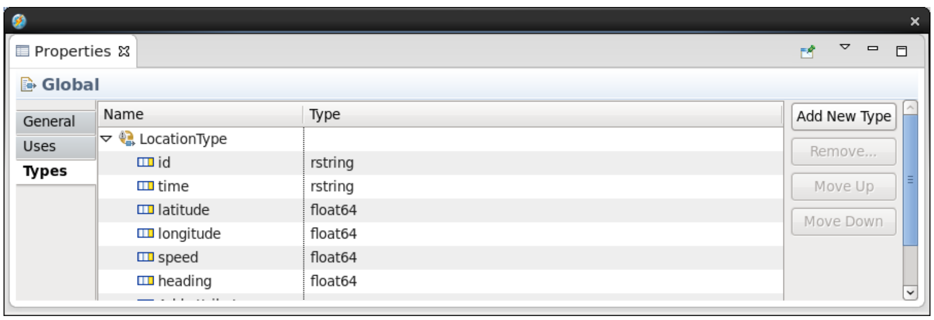
This brings up a **Properties** view, which floats above all the other views. Make sure it looks as below, with the three tabs for General, Uses, and Types; if not, dismiss it and right-click again in the graphical editor, outside the main composite.

__2. In the **Properties** view, click the **Types** tab.
Select the field under **Name**, which says `Add new type…`.
Key in `LocationType`; press **Enter**.
Select the field in the **Name** column below **LocationType**, which says `Add attribute…`.
Key in `id`; press **Tab** to go to the Type column.
Key in `rstring`; Press **Tab** to go to the next name field.

Content assist

In the **Type** column, use **Ctrl+Space** to get a list of available types. Begin typing (for example, "`r`" for `rstring`) to narrow down the list; when the type you want is selected, press **Enter** to assign it to the field. This reduces keyboard effort as well as the probability of errors.

__3. Continue typing and using **Tab** to jump to the next field to enter the attribute names and types listed in Table 2 on page 16.



The tuple type **LocationType** is now available for use as a stream type in any (main) composite within the namespace **my.name.space**. You are now ready to construct the application graph, given a few more specifications, which are listed in Table 3 on page 18. Leave the Properties view up.

Properties views

An alternative to the floating Properties view, which may obscure other views and get in your way, is the Properties tab in the view at the bottom of the perspective. It shows the same information.

Table 3. Additional specifications: other parameters

Parameter	Value
Input file	`/home/streamsadmin/data/all.cars`
Output file	`filtered.cars`
File format	`CSV` (both input and output) Do not quote strings on output
Filter condition	vehicle ID is "C101" or "C133"
Stream names	`Observations` (before filter) `Filtered` (after filter)

With this information, you can create the entire application. You will use the graphical editor. There will be no SPL coding in this lab.

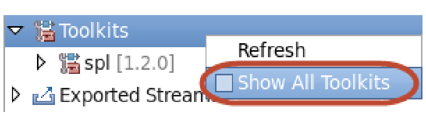
Want to see code?

If you do want to see SPL code for what you are creating, just right-click anywhere in the graphical editor and choose **Open with SPL Editor**.
Explaining what you see there is beyond the scope of this lab.

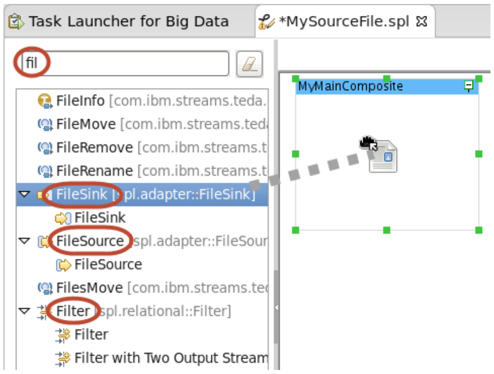
To drop the operators you want into the editor, you need to find them in the palette: this is the panel to the left of the canvas, showing various elements under the headings Design, Toolkits, Exported Streams, and Current Graph. You are looking for three specific operators: **FileSource**, **FileSink**, and **Filter**. You can filter the palette contents and quickly get the ones you want.

First, you will reduce clutter in the palette. Initially, the list of toolkits is long, because it shows all toolkits that Streams Studio knows about; in your preconfigured lab workspace, that means all toolkits installed with Streams. For now, you will not use any of those toolkits (and you have not declared any dependencies), so it is not helpful to have them in the palette.

__4. In the graphical editor for **SourceFile1.spl**, right-click on **Toolkits** in the palette; in the context menu, uncheck **Show All Toolkits**.



__5. In the graphical editor for **MySourceFile.spl**, go to the palette filter field (showing Find), and type `fil`. As it happens, this narrows the palette down to a list that includes the three operators you need. Select each of the ones with a twisty in front of it in turn and drag it into the **MyMainComposite** main composite (make sure the green handles appear before you let go). The editor names the operators: **FileSink_1**, **FileSource_2**, and **Filter_3**.



Note

Make sure that the main composite `MyMainComposite`, and not one of the previously added operators, is highlighted when you drop the next operator. If a **Confirm Overwrite** dialog comes up, simply click **No** and try again.

Note

If you drop the operator on the canvas outside the main composite, the editor creates a new composite (called **Comp_1**) and places the operator there. If that happens, simply undo **(Ctrl+Z** or **Edit > Undo Add Composite with Operator**) and try again.

Operator templates

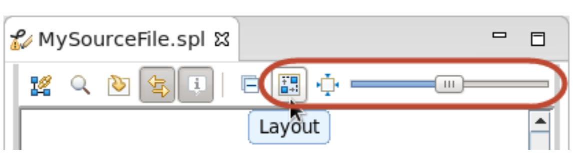
Some operators appear once in the palette; others (the ones you used) have twisties and expand into one or more subentries. These are **templates**: invocations of the operator with specific settings—for example, a **Filter** operator with a second output port to produce rejected tuples. In this lab, the generic version (with the twisty) is always correct, but in Lab 4 the templates come in handy.

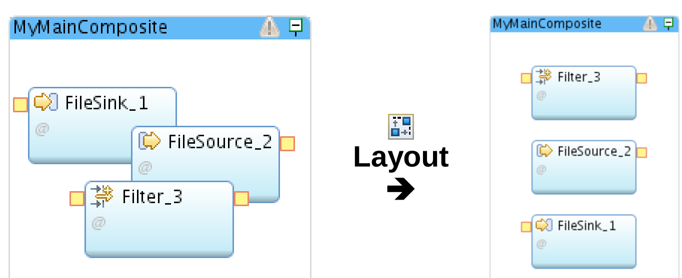
Operator names

The editor generates placeholder names for the operators you drag onto the canvas, consisting of the operator kind ("FileSink") and a sequence number ("_1"). The sequence number depends on the order in which the operators are added to the graph, and yours may not match this document. You can safely ignore that: it does not affect anything in the application, and in any case you will change the generated names later, to match the role each operator plays.

Organize layout and maximize in view

To organize the layout, click the **Layout** button in the editor's toolbar; to zoom in and use all of the space in the graphical editor canvas, click the **Fit to Content** button. The **slider** in the toolbar lets you manually control the zoom level.

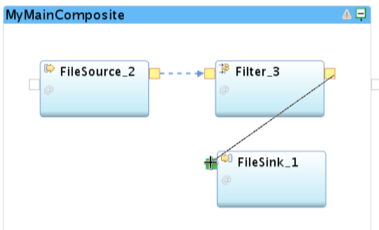




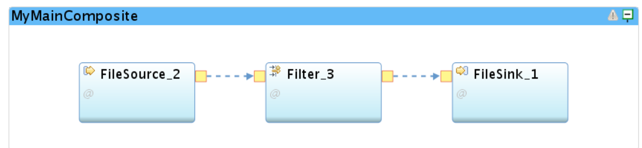
__6. Add a stream connecting **FileSource_2**'s output to **Filter_3**'s input.

Output ports are shown as little yellow boxes on the right side of an operator; input ports on the left. To create a stream, click on an output port and start dragging. The cursor changes to a **+** cross, dragging a line from the output port. Release the mouse button as you drag, and click on the input port of another operator (it turns green when you hover over it) to complete the link. The two ports are now connected by a dashed line, indicating that there is a stream but its type is not yet defined.

__7. Add another stream, from **Filter_3** to **FileSink_1**. Click **Layout** and **Fit to Content** to organize the graph.

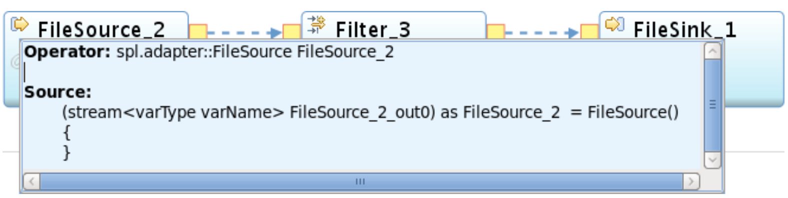


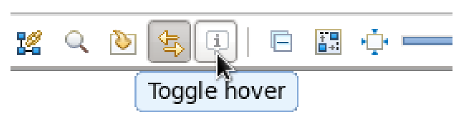
Layout ➔



Hover popups

By default, hovering over an operator in the graphical editor brings up a popup showing the SPL code behind that operator. As you build and lay out the graph, these popups may get in the way. Click the **Toggle hover** toolbar button to turn off these popups.





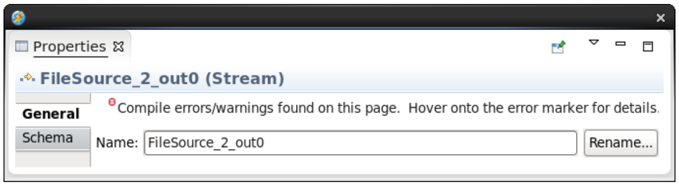
You now have the complete graph, but none of the details have been specified.

__8. Save your work: use **Ctrl+S** or the **Save** toolbar button, or **File > Save** in the menu.

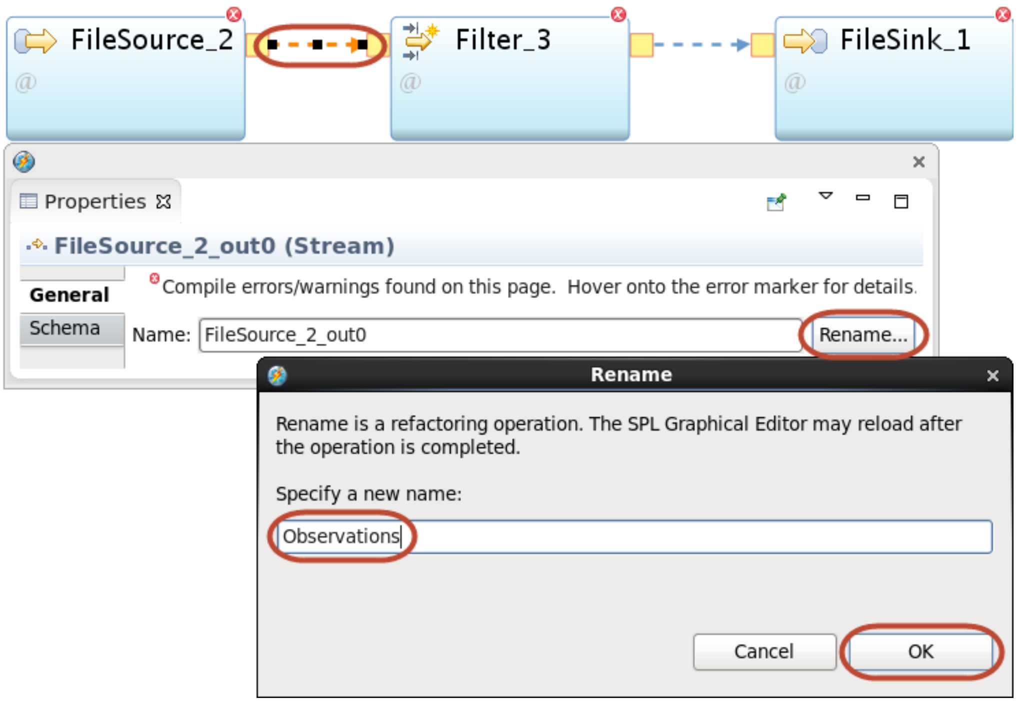
The main composite as well as the three operators it contains now have error indicators. This is expected, as the code so far contains only placeholders for the parameters and stream types, and those placeholders are not valid entries.

__9. To assign a name and type to a stream, select the stream (dashed arrow) connecting **FileSource_2** and **Filter_3**. (Sometimes you need to try a few times before the cursor catches the stream, instead of selecting the enclosing main composite.)

The **Properties** view, which you used earlier to create LocationType, now shows **Stream** properties. Reposition and resize the view if necessary so it doesn't obscure the graph you're editing. (If you closed the Properties view, double-click the stream to reopen it.)



__10. Descriptive stream names are preferable to the placeholder names generated by the editor. In the **Properties** view (**General** tab), click **Rename…**; in the **Rename** dialog, under **Specify a new name:** type `Observations`, and click **OK**.

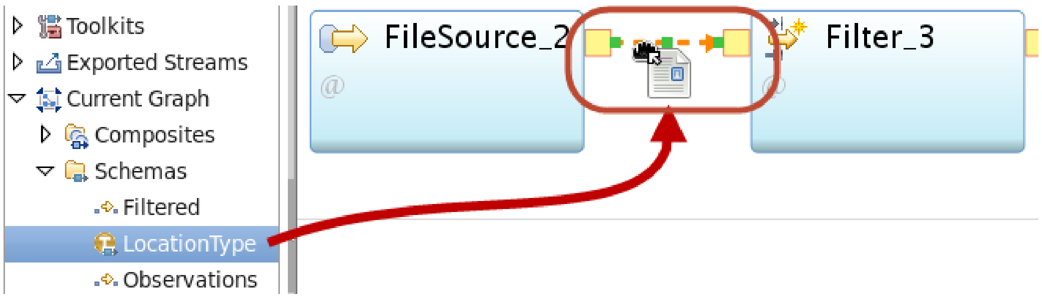


Refactoring

Why do you have to click a button and bring up a new dialog to rename a stream or an operator? Why not just type away in the field showing the name?
Because changing a name involves a *refactoring* process, meaning that all references to that name must be found and changed too. Streams Studio needs a signal to know when to kick off this process; in this case, you provide that signal by clicking **OK**.

__11. Specify the stream schema. You can do that in the Properties view, but since you have created a type for this, you can also just drag and drop it in the graphical editor, like any other object.

In the graphical editor, clear the palette filter by clicking the eraser button next to where you typed `fil` earlier; this makes all the objects visible again. Under **Current Graph**, expand **Schemas**. This shows the **LocationType** type, along with the names of the two streams in the graph. Select **LocationType** and drag it into the graph; drop it onto the **Observations** stream (the one between **FileSource_2** and **Filter_3**). Make sure the stream's selection handles turn green as you hover before you let go, as shown below.



If you still have the Properties view open to the Schema tab, it now shows a Type of **LocationType**, and **<extends>** as a placeholder under Name. This indicates that the stream type does not contain some named attribute of type LocationType, but instead inherits the entire schema with its attribute names and types.

__12. Using the same drag and drop technique, assign the **LocationType** type to the other stream, between **Filter_3** and **FileSink_1**. Select that stream so its properties show in the Properties view.

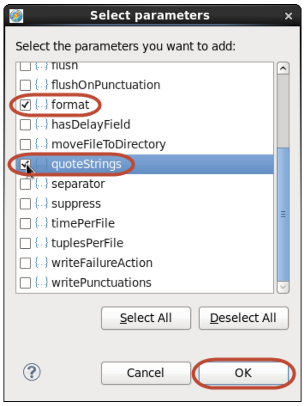
In the **Properties** view, go to the **General** tab and rename the stream to `Filtered`. (Click **Rename…**, etc.)

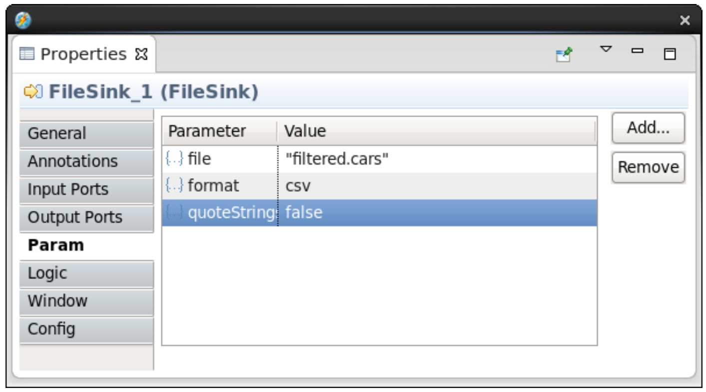
There is still an error indicator on **FileSink_1** and, because of that, on the main composite, too. This is expected, because you have not yet told the **FileSink** operator what file to write. You also need to provide details for the other operators.

__13. In the graphical editor, select **FileSink_1**. In the **Properties** view, click the **Param** tab. This shows one mandatory parameter, **file**, with a placeholder value of `parameterValue` (not a valid value, hence the error marker). Click on the field that says `parameterValue` and type `"filtered.cars"` (with the double quotes). Press **Enter**.

Note that this is a relative path (it doesn't start with "`/`"), so this file will go in the **data** subdirectory of the current project, as specified by default for this application.

__14. You need two more parameters. Click **Add…**; in the **Select parameters** dialog, check **format** and **quoteStrings** (you may have to scroll down to find it); click **OK**. For the value of **format**, enter `csv` (no quotes: this is an **enumerated** value). For the value of **quoteString**, enter `false` (no quotes: this is a **boolean** value).





__15. While you have the **FileSink** operator selected, go back to the **General** tab in the **Properties** view. Rename the operator in the same way you renamed the two streams; call it `Writer`.

__16. The **FileSource** operator needs to know what file to read. In the graphical editor, select the **FileSource_2** operator. In the **Properties** view (**Param** tab), click **Add…**; in the **Select parameters** dialog, select **file** and **format**, and click **OK**. In the value for **file**, enter `"/home/streamsadmin/data/all.cars"` (with quotes and exactly as shown here, all lowercase); for **format**, enter `csv`.

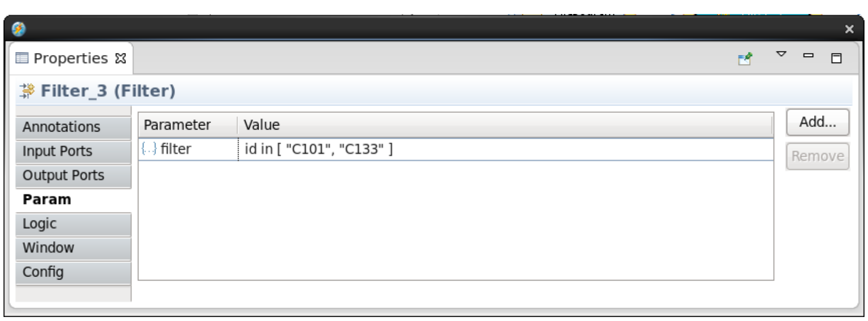
__17. Still in the **FileSource** properties, go back to the **General** tab and rename the operator. This time, simply keep the name blank. Observe how the Identifier changes to `Observations`, the name of the output stream.

Operator identifiers and aliases: confusing?

SPL automatically assigns names (Identifiers) to operators by using the name(s) of the output stream or streams. It also provides for aliases to use as identifiers instead, giving the developer more control. In case of a single output stream, using the stream name is usually fine, so we prefer to omit the alias. For multiple output streams it's usually better to provide a more descriptive alias. When there is no output stream (as for the FileSink), an alias is mandatory.

__18. You have to tell the **Filter** operator what to filter on; without a filter condition, it will simply copy every input tuple to the output. In the graphical editor, select **Filter_3**. In the **Properties** view (**Param** tab), click **Add…**; in the **Select parameters** dialog, select **filter** and click **OK**.

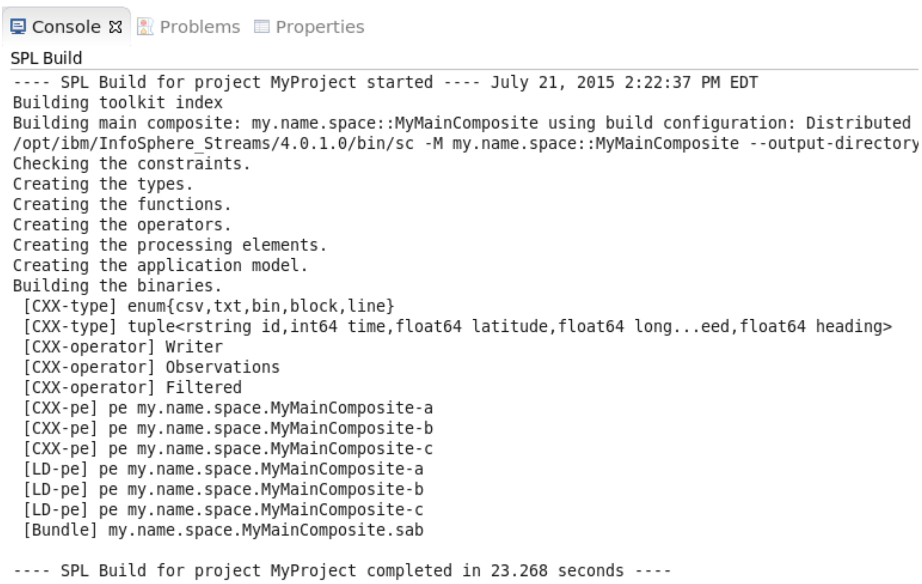
In the value field, enter the boolean expression `id in ["C101","C133"]` to indicate that only tuples for which that expression evaluates to `true` should be passed along to the output. (The expression with the key word `in` followed by a list evaluates to true only if an element of the list matches the item on the left.)



Rename the operator by blanking out the alias. In the graph it will show up as `Filtered`.

Save. The error markers disappear: you have a valid application. Dismiss the floating Properties view.

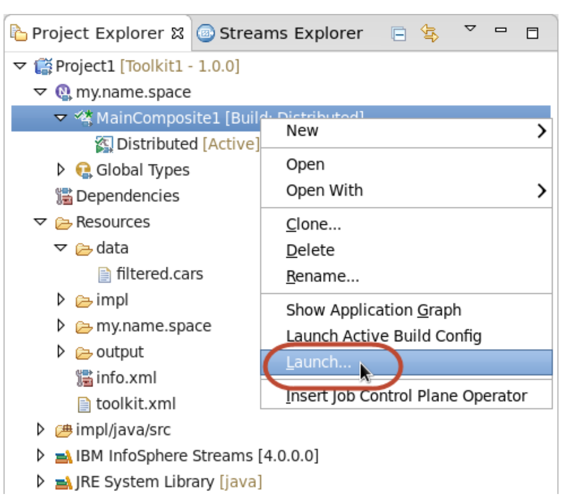
The Properties view may have been obscuring this, but each time you saved the graph, a *build* was started to compile the application; the progress messages are shown in the **Console view** (in the **SPL Build** console). If you scroll back (you may also want to enlarge or maximize the Console), you will see some builds that terminated on errors, with messages in red. The last build should have completed without any errors.



1.6 Running an Application

You are now ready to run this program or, in Streams Studio parlance, launch the build.

__1. In the **Project Explorer**, right-click **MyMainComposite** (expand **MyProject** and **my.name.space** if necessary); choose **Launch…**.



In the **Edit Configuration** dialog, click **Apply**, and then **Continue**.

The **Streams Launch** progress dialog appears briefly. To see what happened, switch consoles, from the SPL Build to the Streams Studio Console.

__2. In the Console view, click the **Display Selected Console** button (way over to the right) to switch consoles.

The **Streams Studio Console** shows that job number 0 was submitted to the instance called **StreamsInstance** in the domain **StreamsDomain**.

Launch configuration

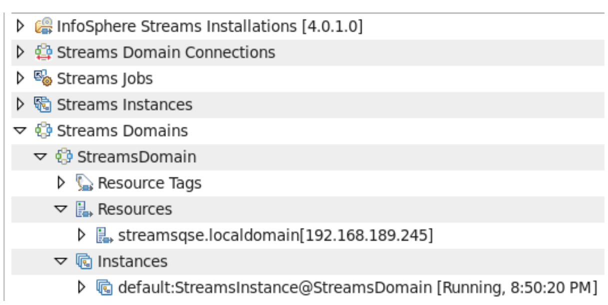
There are many options that can be set at the time you launch an application; for now, ignore all that.

Starting the instance

You started the instance in the beginning of this lab. If for some reason the instance is not running, no worries—you will be prompted to start it.

Since nothing else seems to have happened, you need to look for some effect of what you've done. First, you'll view the job running in the instance; then you'll inspect the results.

__3. Switch to the **Streams Explorer** (the second tab in the view on the left, behind the **Project Explorer**). Expand the **Streams Domains** folder, the one domain named **StreamsDomain**, and the **Resources** and **Instances** folders under that:



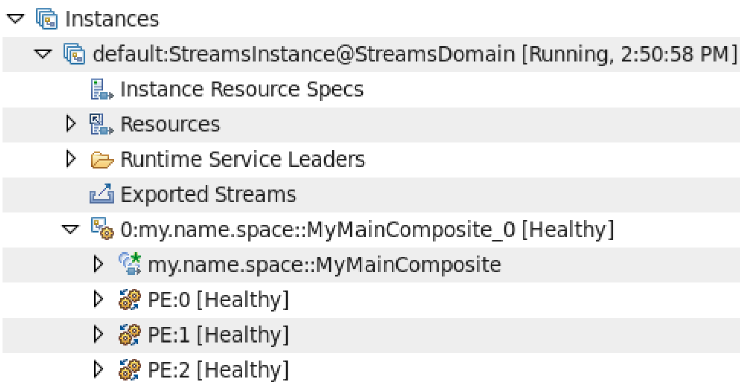
The Resources folder refers to the machines available to the domain; of course, in this virtual machine there is only one. Resource tags let you dedicate different hosts to different purposes, such as running runtime services or application processes. In this single-resource environment, this is not relevant.

Streams jobs and instances

The Streams Jobs and Streams Instances folders simply repeat, for convenience, information from the Instances folders under the listed domains, but they lack further drill-down information.

You know you have one instance, **StreamsInstance**, in the domain StreamsDomain; the entry tells you that it is the **default** instance (where launched applications will run unless otherwise specified), that it is **Running**, and its current wall clock time. (You may have to widen the view to see the status.)

__4. Expand **default:StreamsInstance@StreamsDomain**. This shows a number of elements, but the only one you're interested in for now is the last one: **0:my.name.space::MyMainComposite_0** is the job you have just submitted (a running application is called a job), and its status is **Healthy**. There is much information here about a job and its constituent operators and Processing Elements (PEs), but you are going to explore it graphically instead.



What if my job is not healthy?

If you are in an instructor-led lab, ask the instructor to help you. Otherwise, you have to diagnose the problem yourself. The most likely culprit is a mistake in the FileSource's **file** parameter, which would not be caught by the compiler but would cause a file-not-found exception at runtime. More elaborate debugging, involving trace files and other diagnostics, is beyond the scope of this lab.

__5. Right-click on **default:StreamsInstance@StreamsDomain**; choose **Show Instance Graph**.

A new view opens up in the bottom panel, showing a graph similar to that in the graphical editor; but this one shows, live, what is actually running in the instance. If you launch multiple applications (or multiple copies of the same application) you will see them all. And it lets us inspect how data is flowing, whether there are any runtime problems, etc.

If necessary, expand or maximize the **Instance Graph** view; click **Fit to Content** in the view's toolbar.

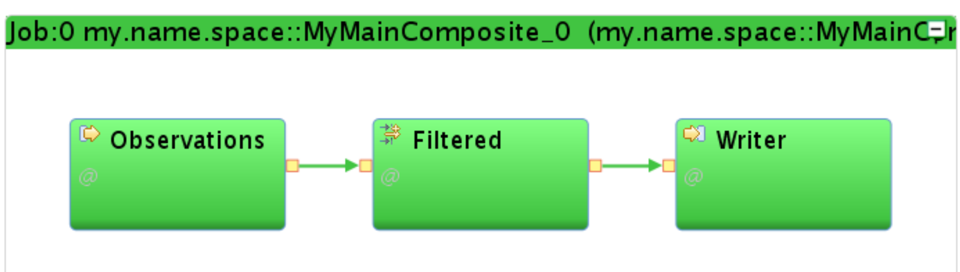


Figure 2. Finished and running application. All PEs are healthy, so the job is, too.

There is a lot more to the Instance Graph; you will explore it further in the next lab.

__6. Hover over **Filtered**; you get current information about data flows and other metrics. Among other things, it will tell you that the input received 1902 tuples, and the output sent 95 tuples. This seems reasonable, given that the output should be only a subset of the input. But it also says that the current tuple rate on both input and output is **0/sec**, so no data is currently flowing. Why? Because it has read all the data in the `all.cars` file, and there will never be any more data.

Streams jobs run forever

The input data is exhausted but the job is still running. This is always true for a Streams application running in an instance (*distributed* applications): a job can only be canceled by manual (or scripted) intervention. In principle, a stream is infinite, even though in some cases (like when reading a file) this may not be quite true.

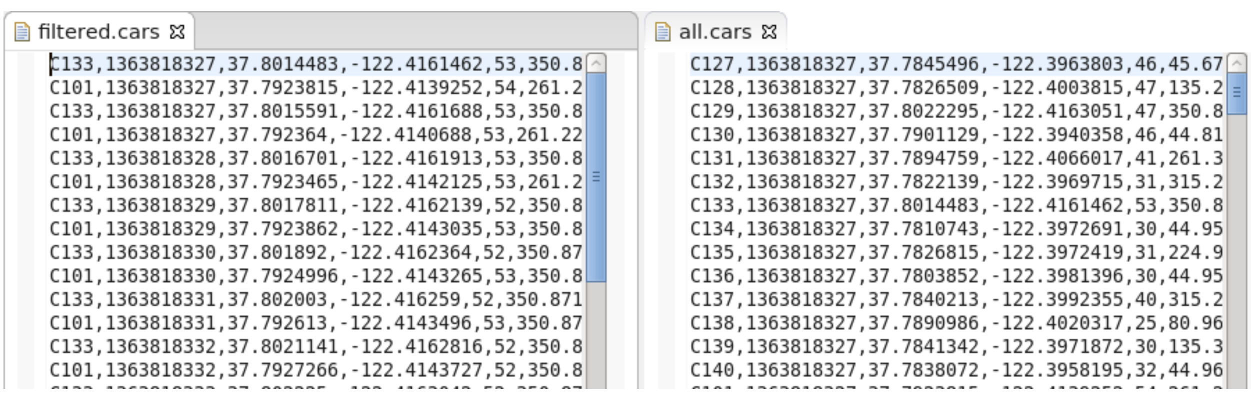
__7. To inspect the results, you must look at the input and output data.

__a. In the top menu, choose **File > Open File...**

__b. In the **Open File** dialog, browse to **streamsadmin/data/all.cars** and click **OK**.

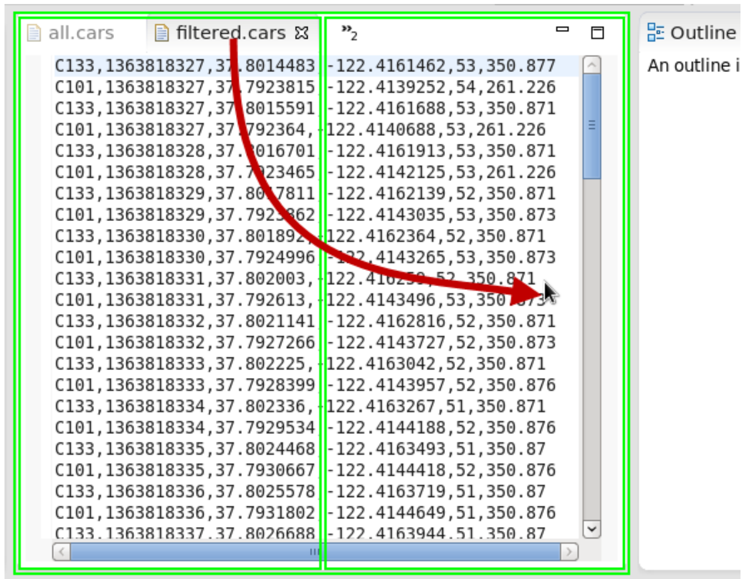
Studio opens the file in an editor view; it shows location observations for multiple vehicle IDs: C127, C128, etc.

__8. In the **Project Explorer** (the first tab in the view on the left), expand **Resources**; there should be a file under **data**, but there is no twisty in front of the directory. To update the view, right-click **data** and choose **Refresh**. The twisty appears; expand it, and double-click on **filtered.cars**. This file contains coordinates, speeds, and headings only for vehicles C101 and C133.



Show files side by side

You can show two editors side by side by dragging the tab of one of them to the right edge of the editor view. As you drag, green outlines appear, which arrange themselves side by side as you approach the right edge. To undo, drag the tab back to a position among the other tabs, or close it.



all.cars
filtered.cars
Outline

Lab 2 Understanding the flow of data

In this lab, you will further develop the vehicle data filtering application and get a more detailed understanding of the data flow and the facilities in Studio for monitoring and examining the running application. To make this easier, you will make two enhancements that let you see what is happening before the data runs out: you will slow the flow down (left to its own devices, Streams is just too fast) and you'll make it possible to read multiple files. This is a general design pattern for development and debugging.

2.1 Building on the previous results

This section is optional

If you are confident that your results from the previous lab are correct, you can continue working with them and skip this section.

To make sure everyone continues from the same point, go to the next Studio workspace that has been prepared for you.

__1. In the Studio top menu, choose
File > Switch Workspace > /home/streamsadmin/Workspaces/workspace2.

Studio shuts down and restarts with the new workspace. A project with the same application you built in Lab 1 is already available.

__2. The editor view shows the Task Launcher for Big Data; click on the tab for **MySourceFile.spl**.

In the graphical editor's palette, right-click **Toolkits** and uncheck **Show All Toolkits**. (This setting reverts to the default of showing all when you close and reopen Studio.)

Rearrange the graph as needed, using **Layout** and **Fit to Content**.

__3. Show the **Instance Graph**:

__a. In the **Streams Explorer**, expand **Streams Instances**

__b. Right-click on **default:StreamsInstance@Streamsdomain** and choose **Show Instance Graph**.

You will see the job from Lab 1 still running.

Jobs in the Instance Graph

The Instance Graph shows all jobs running in the instance. This has nothing to do with Streams Studio workspaces; regardless of which workspace Studio is currently in, the Instance Graph for a given instance will look the same, showing jobs launched by Studio from any workspace, or not launched by Studio at all (jobs may be submitted from the Linux command line, for example).

2.2 Enhancing the application

Two new operators are needed to make your application easier to monitor and debug. The **Throttle** operator copies tuples from input to output at a specified rate rather than as fast as possible. The **DirectoryScan** operator periodically scans a given directory; for each new file that satisfies optional criteria, it sends out a tuple that contains the file's full path.

Instead of using the palette's filter field to quickly pick up the operators you want, let's browse the full palette to achieve the same result.

__1. In the graphical editor's palette, expand **spl** (under **Toolkits**), and then **spl.adapter**. Drag **DirectoryScan** into the main composite. The editor names the operator **DirectoryScan_4**.

__2. Scroll down in the palette and expand **spl.utility**; scroll down further and find **Throttle**. Drag and drop it onto the stream **Observations**, exactly as you did with the **LocationType** schema in step __11. on page 21. (Make sure the stream is highlighted by green handles before you let go.)

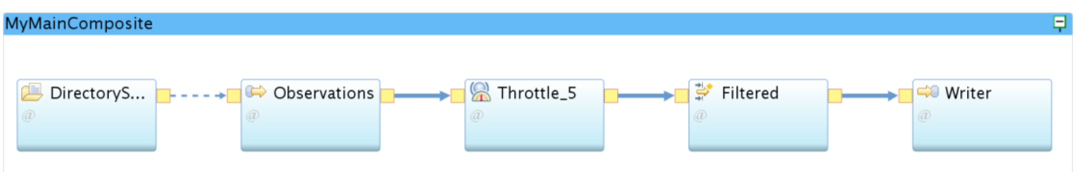
The operator will be called **Throttle_5**. The editor automatically connects the **Observations** stream to its input port and creates a new stream, with the same schema as Observations, from its output port to the input of **Filtered**. There is no need to adjust the schema of this new stream: The **Throttle** operator merely controls the rate at which tuples flow, without changing their contents. But do rename the new stream, to `Throttled`.

To straighten out the graph, click **Layout** and **Fit to Content**.

__3. Drag a stream from the output of **DirectoryScan_4** to the input of **Observations**.

> **Optional input ports**
>
> The **FileSource** operator can have an input port, but it is optional. In the original graph you did not use it, so there is no yellow box on the left. But while dragging a stream from another operator's output port, the optional input port is indicated by a lightly outlined, unfilled box, and you can connect the stream to it like any other port.

Click **Layout** and **Fit to Content**.



To finish up, you need to define the schema for the stream from the **DirectoryScan**, and tell that operator where to look for files; adjust the configuration of **Observations** (since it now gets its instructions from an input stream rather than a static parameter); and tell the **Throttle** the flow rate you want. Rename the new operators while you're at it.

__4. The **DirectoryScan** operator's output port supports only one schema: a single attribute of type `rstring`, which will hold the full path to the file; you can call that attribute anything you like.

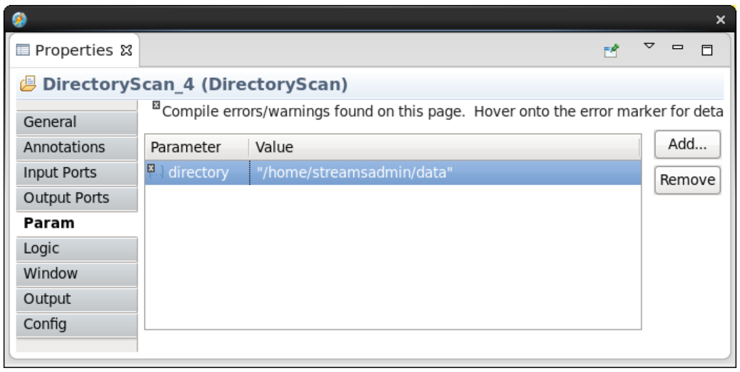
__a. Select the output stream from **DirectoryScan_4**; right-click and choose **Edit**.

__b. In the **General** tab, rename the stream to `Files`.

__c. In the **Schema** tab in the **Properties** view, click on the first **Name** field (placeholder `varName`) and type `file`; press **Tab** to move to the next field (placeholder `varType`).

__d. Enter `rstring`. Remember to use content assist (**Ctrl+Space**) to reduce typing and avoid errors. Press **Enter**.

__5. In the editor, select the **DirectoryScan_4** operator. In the **Properties** view, go to the **Param** tab and set the **directory** parameter to the value `"/home/streamsadmin/data"`. (Remember to include the double quotes.)



Rename the operator (to `Throttled`) by blanking out its alias.

A **FileSource** operator knows which file(s) to read either from a static parameter (called **file**) or from the tuples coming in on an input stream—but not both. Now that you are getting filenames from a **DirectoryScan** operator, that **file** parameter you used previously is no longer needed; in fact, it's an error to keep it.

__6. Select the **Observations** operator in the editor. In the **Properties** view (**Param** tab), click on the **file** parameter and then click **Remove**.

__7. The **Throttle** operator has a mandatory parameter for specifying the desired flow rate; it is a floating-point number with a unit of tuples per second.

In the editor, select **Throttled**. In the **Properties** view (**Param** tab), click on the Value field next to the **rate** parameter and enter `40.0`. (The decimal point is necessary to indicate a floating-point value). In the **General** tab, rename the operator by blanking out its alias.

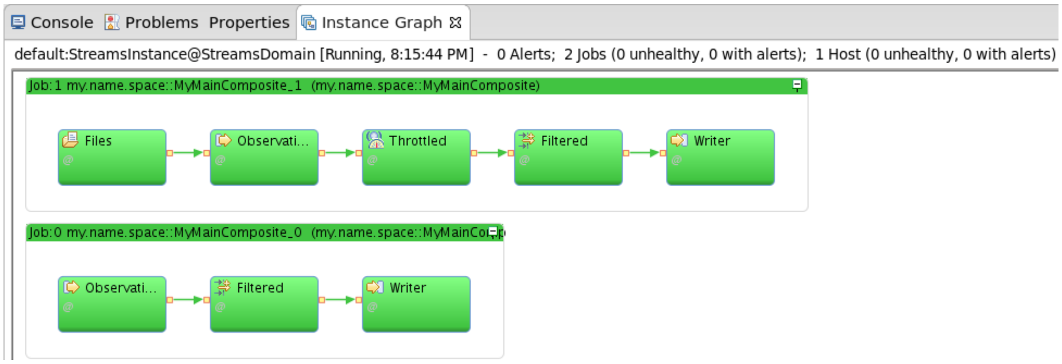
Save. There should be no errors.

2.3 Monitoring the application with the Instance Graph

The Instance Graph in Studio provides many ways to monitor what your application does and how data flows through a running job. This part of the lab is a matter of exploring those capabilities; the steps here are just hints.

__1. Launch the application: In the **Project Explorer**, right-click on the main composite (**MyMainComposite**) and choose **Launch…**. In the **Edit Configuration** dialog, click **Continue** (if you switched workspaces, you may have to click **Apply** first).

__2. Maximize the **Instance Graph** view. You now have two running jobs: the one you just launched and the one from the previous exercise. The old one is dormant (it is not getting any data), but leave it running for now.



To the right of the Instance Graph, a layout options drop-down menu and two selection panes for **Layers** and **Color Schemes** allow you to control the display. Explore the various options; here are some explanations and suggestions that will help interpret what you see.

The layout options control how the operators in the graph are grouped:

- By **Composite**: This is the default. You see two boxes representing the two main composites—that is, the two applications, and inside each composite the operators that make up the application; three for the old job and five for the new one.
- By **Category**: You can assign categories to your operators to group them any way you want; this is useful when you have a large number of operators. You did not use this facility in this lab, so this option shows all the operators without grouping—though you can still identify the two distinct flows, of course.
- By **PE:** A PE is a **Processing Element**—essentially a runtime process. Operators can be combined (*fused*) into a single PE; this couples them tightly and reduces communication latencies. That is a performance optimization beyond the scope of this lab; you've simply used the default of giving each operator its own process. This layout option shows each operator inside a box representing its PE, which in this case does not add much information.
- By **Resource**: Since the virtual machine is only a single resource (host), all operators are shown within the same box representing the resource.

__3. For the rest of this exercise, keep the layout set to **Composite**.

In the **Layers** box, only the **Alert** layer is relevant; it should be checked. The other, **Consistent Region**, is beyond the current scope (it has to do with guaranteed tuple delivery); checked or unchecked, it will have no bearing on this lab.

Move on to the **Color Schemes**. Note that they are mutually exclusive (you can only check one at a time), and you cannot interact with the checkboxes on the individual colors under each scheme. It is possible, however, to add new color schemes, and to edit existing ones. The color schemes assign colors to the operators based on their properties (the PE or job they belong to, the resource they run on, etc.) or on metrics—counters maintained by the Streams runtime to monitor diagnostic statistics, which are refreshed periodically and present a real-time view of the instance. This is extremely helpful in identifying problems quickly, especially in a large, complex graph.

__4. The default color scheme is **Health**. Expand the twisty for **Health** in the **Color Schemes** pane: green indicates that the operators (actually, their PEs) are healthy, meaning that there are no errors and they are up and running and ready to process data. You may have noticed when launching an application that the operators are red and yellow before they turn green; this reflects the initialization process, during which a PE may be waiting for another to finish loading before a stream connection can be made, for example.

__5. Check the **Flow Under 100 [nTuples/s]** color scheme. All operators (most likely) turn black, indicating that no tuples are flowing. This is because it has been more than 45 seconds or so since you launched the application; despite the throttling it finished reading the entire file.

You will need to supply more data. This is easy to do by making the same file appear multiple times in the source directory.

__6. One way to do this is by using the **File Browser**.

  __a. Open it by double-clicking the **streamsadmin's Home** icon on the desktop (you may need to minimize or move the Studio window out of the way to see it)

  __b. Double-click the **data** directory to open the directory in a new window or view.

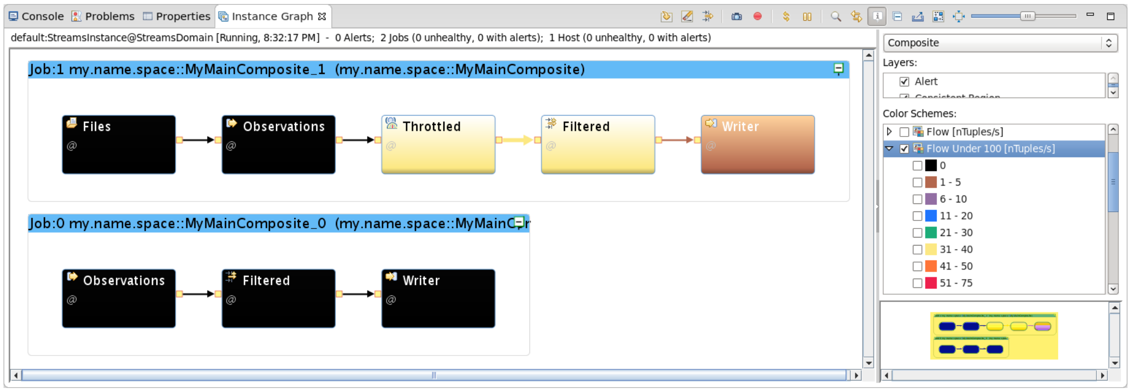
  __c. Close or minimize the **streamsadmin** File Browser window.

  __d. In the **data** window, right-click on the **all.cars** file; in the context menu, choose **Copy**.

  Now, as long as you don't copy something else, you can paste this file into the directory as many times as you want; each time, a copy of the file will be added, and the application will see it and process the data. With over 1,900 records in the file and a throttled rate of 40 tuples per second, each file takes just over 45 seconds to process.

__7. Make sure you can see the Instance Graph at the same time as the **Data** File Browser window. Right-click anywhere in the **data** window and choose **Paste**.

  After a slight delay (the **DirectoryScan** operator scans every five seconds), the colors change in the new job; the old job stays black, as it is not designed to read more than one file. Aside from some startup effects, the colors are mostly yellow (31-40) for **Throttled** and **Filtered**, and brown (1-5) for **Writer**. It makes sense for the rate after the **Filter** operator to be lower, as only a subset of tuples makes it through. After less than a minute, the operators revert to black; paste another copy of the file to start the flow again.



__8. Check the **nFinalPunctsSubmitted** color scheme. A Final Punctuation marker is a special message (not a tuple), generated by any operator to indicate that a stream will no longer carry new tuples. This marker travels downstream through the graph; any operator that has received one on every one of its input ports (every operator you've used so far has only one input port) is from then on dormant and ready to shut down, and in turn submits a Final Punctuation. Operators without output ports, like the **FileSink**, do not support this metric, so they are not colored by this scheme.

Notice that the operators in the old job are now black, indicating that they have submitted a Final Punctuation; this happened when the **FileSource** reached the end of the input file. The operators in the current job are green (no Final Punctuation) because they have not reached the end of the input data, and never will: there is no way to know when another file will appear in the source directory.

__9. Now you might as well get rid of the old job; you don't want it cluttering up your Instance Graph any further. There are at least three options.

__a. If you just want to remove some clutter, simply collapse the main composite by clicking the **minimize** button in the title bar of the composite.

__b. If you want it to disappear from view altogether but are not ready to cancel the job, just filter it out of the graph display: click the **Filter graph…** button in the **Instance Graph** view's toolbar; in the **Instance Graph Filter** dialog, check only the most recent job under **Job ID Filter**, and click **OK**.

__c. To cancel the job completely, right-click anywhere in the main composite and choose **Cancel job**; in the **Confirm** dialog, click **Yes**.

Click **Fit to Content**.

2.4 Viewing stream data

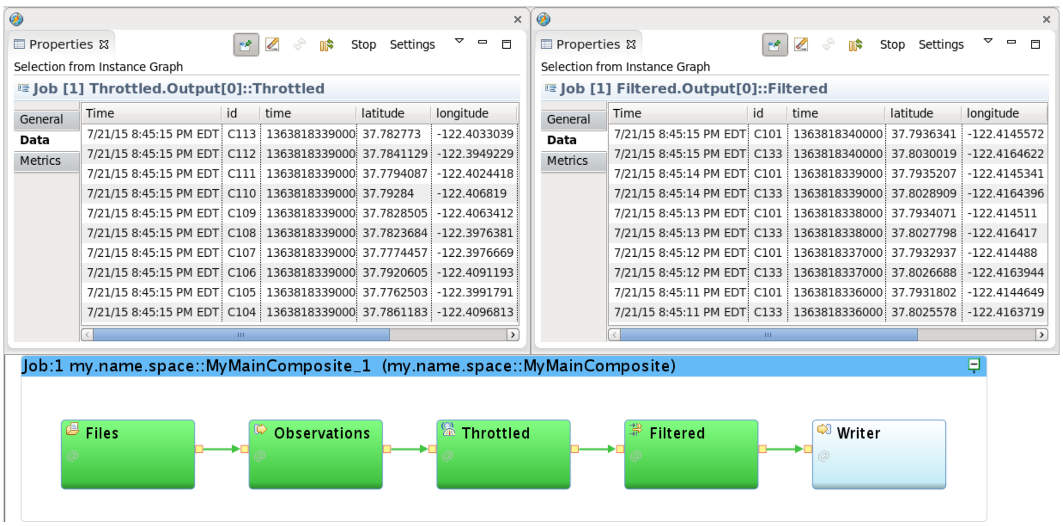
While developing, you often want to inspect not just the overall tuple flow, but the actual data. In Lab 1 you simply looked at the results file, but you can also see the data in the Instance Graph. This way, you don't have to add **FileSink**s whenever you want to capture the output of a particular operator. Let's look at the input to and output from the **Filter** operator to see if it's working as expected.

__1. In the Instance Graph, right-click on the stream **Throttled** (output of the **Throttled** operator, input to **Filtered)**; choose **Show Data**. (If you get an Untrusted Certificate Trust Manager message, select **Permanently accept the certificate** and click **OK**.)

In the **Data Visualization settings** dialog, verify that the tuple type is what you expect (attributes **id**, **time**, **latitude**, **longitude**, **speed**, and **heading**) and click **OK**. A **Properties** view appears.

__2. Repeat the previous step for the stream **Filtered** (between operators **Filtered** and **Writer)**. Move and resize both **Properties** views so you can see the both tables as well as the Instance Graph.

Now paste another copy of the input file into the **Data** directory and watch the data tables. Notice that, as expected, the Filtered stream contains only tuples with an **id** value of “C101” or “C133”, whereas the Throttle output contains a greater mix of vehicle IDs.



When you have seen enough data, dismiss the two floating Properties views.

Lab 3 Enhanced analytics

In this lab, you will enhance the app you've built by adding an operator to compute an average speed over every five observations, separately for each vehicle tracked. After that, you will use the Streams Console to visualize results.

So far, the operators you've used look at each tuple in isolation; there was no need to keep any history. For many analytical processes, however, it is necessary to remember some history to compute the desired results. In stream computing, there is no such thing as "the entire dataset", but it is possible to define buffers holding a limited sequence of consecutive tuples—for example, to compute the average over that limited subset of tuples of one or more numeric attributes. Such buffers are called **windows**. In this lab, you will use an **Aggregate** operator to compute just such an average.

3.1 Building on the previous results

This section is optional
If you are confident that your results from the previous lab are correct, you can continue working with them and skip this section.

To make sure everyone continues from the same point, go to the next Studio workspace that has been prepared for you.

__1. In the Studio top menu, choose
File > Switch Workspace > /home/streamsadmin/Workspaces/workspace3.

Studio shuts down and restarts with the new workspace. A project with the same application you built in Labs 1 and 2 is already available

__2. The editor view shows the Task Launcher for Big Data; click on the tab for **MySourceFile.spl**.

In the graphical editor's palette, right-click **Toolkits** and uncheck **Show All Toolkits**. (This setting reverts to the default of showing all when you close and reopen Studio.)

Rearrange the graph as needed, using **Layout** and **Fit to Content**.

__3. Show the **Instance Graph**:

__a. In the **Streams Explorer**, expand **Streams Instances**

__b. Right-click on **default:StreamsInstance@Streamsdomain** and choose **Show Instance Graph**.

You will see any jobs from the previous labs that are still running. Cancel or hide them as you see fit.

3.2 A window-based operator

You will compute average speeds over a window, separately for vehicles C101 and C133. Use a *tumbling* window of a fixed number of tuples: each time the window has collected the required number of tuples, the operator computes the result and submits an output tuple, discards the window contents, and is again ready to collect tuples in a now empty window. Window partitioning based on a given attribute means that the operator will allocate a separate buffer for each value of that attribute—in effect, as if you had split the stream by attribute and applied a separate operator to each substream. The specifications are summarized in Table 4, below.

Table 4. Specifications for window-based aggregation

Specification	Value
Operator type	**Aggregate**
Window specification	**Tumbling**, based on tuple **count**, 5 tuples
Window partitioning	Yes, based on vehicle ID (**id**)
Stream to be aggregated	**Filtered**

Output schema	`id - rstring`
	`time – int64`
	`avgSpeed – float64`

Aggregate computation	`Average(speed)`
Results destination	File: `average.speed`

__1. Add the two required operators.

__a. In the graphical editor's palette filter box, type `agg`; drag an **Aggregate** operator into the main composite. The editor calls it **Aggregate_6**. This is you main analytical operator.

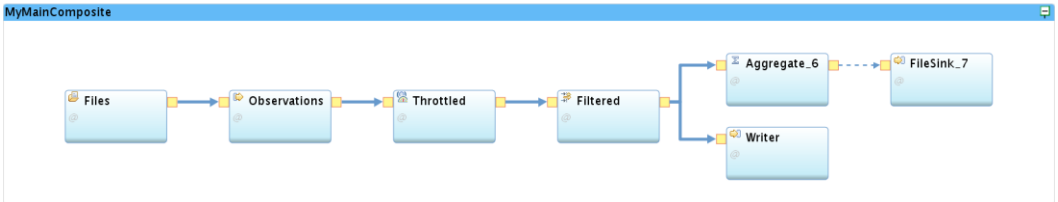
__b. In the palette filter, begin typing `filesink` until you see the **FileSink** operator; drag one into the main composite: **FileSink_7**. This will let you write the analytical results to a file.

__2. Fold the two new operators into the graph by connecting one existing stream and adding another.

__a. Drag a stream from **Filtered** to **Aggregate_6**. This means Aggregate_6 is tapping into the same stream as Writer is already consuming, so the schema is already defined. This is indicated in the editor by a solid arrow.

__b. Drag another stream from **Aggregate_6** to **FileSink_7**. This stream does not yet have a schema, so the arrow is dashed.

Click **Layout** and **Fit to Content**.



__3. Give the output of the **Aggregate** operator its own schema.

__a. Right-click on the new stream from **Aggregate_6** to **FileSink_7** and choose **Edit**.

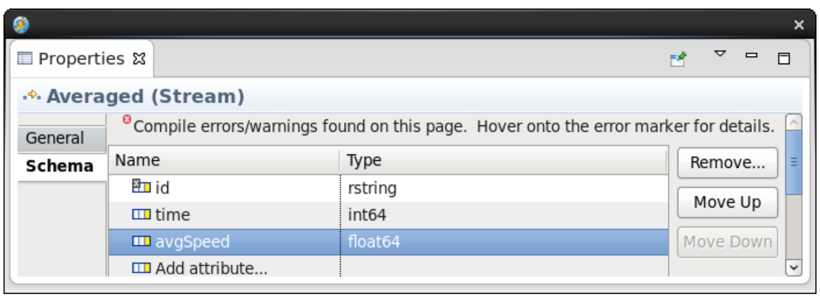
__b. In the **General** tab of the **Properties** view, rename the stream to `Averaged`.

__c. In the **Schema** tab of the **Properties** view, fill in attribute names and types.

__i. In the first field under **Name,** type `id`; press **Tab.**

__ii. Under **Type**, type `rstring`; press **Tab** to go to the next name field.

__iii. Continue typing (and using **Tab** to jump to the next field) to enter the output schema attribute names and types listed in Table 4 on page 37.



__4. Tell the Aggregate operator what to do.

__a. Select the **Aggregate_6** operator. In the **General** tab of the **Properties** view, rename the operator to `Averaged` by blanking out the alias.

Go to the **Input Ports** tab. Notice that the first and only port is called `Port0` (you cannot change that), and that it has an alias, `inPort0Alias`. Port aliases are useful because they allow you to refer to ports with identifiers that are local to the operator and independent of the names of the streams that connect to them—and therefore not vulnerable to any changes in those names. Until now, you have not used any operators where this mattered, but the Aggregate is an operator that requires you to refer to your input and output ports in some of the property tabs.

__b. The name generated by the editor is a bit unwieldy, so change it to `In`. (The capital `I` is important; `in` with lowercase `i` is an SPL key word that cannot be used as an identifier.)

__c. Go to the **Window** tab. A placeholder window specification is already filled in; you only need to edit it slightly.

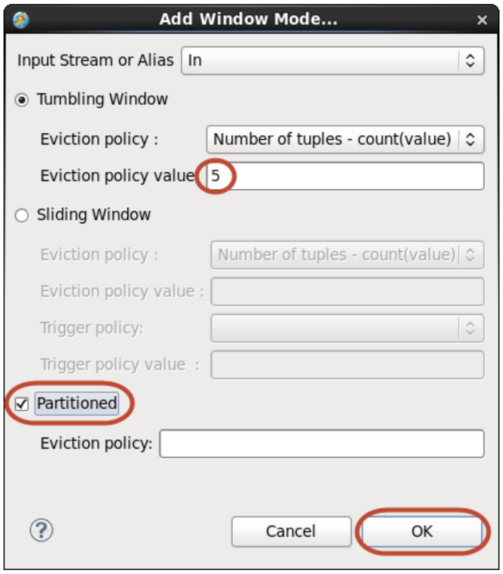
__i. Click **Edit…**;

__ii. in the **Add Window Mode…** dialog, keep **Tumbling Window** selected;

__iii. set **Eviction policy value** to `5`;

__iv. check **Partitioned** (leave **Eviction policy** blank); and

__v. click **OK**.



__d. Configure the window as partitioned on vehicle ID (the `id` attribute).

__i. In the **Param** tab, click **Add…**

__ii. in the **Select parameters** dialog, check **partitionBy** and click **OK**.

__iii. In the **partitionBy** value field, enter `id`.

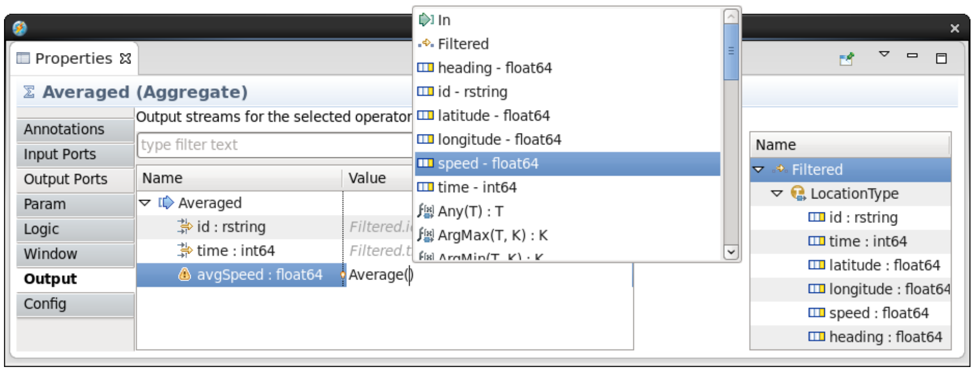
__e. Go to the **Output** tab (you may have to scroll down the list of tabs, or make the Properties view taller) to specify the output assignment. Expand the twisty in front of `Averaged` in the **Name** column; you may have to widen the columns and enlarge the view horizontally to see the full Name and Value columns. The attributes **id** and **time** will simply be copied from the most recent input tuple. This is already reflected in the Value column; by default, output attribute values are assigned from attributes of the same name, based on the last input tuple.

Since the window is partitioned by **id**, all tuples in a window partition will have the same value for this attribute. This is not the case for **time**, but in this example it is reasonable to use the most recent value.

__i. Click **Show Inputs**; expand the **Filtered** twisty, and again **LocationType**. This shows the attributes you can use to create an output assignment expression.

__ii. Click in the value field for **avgSpeed**; type **Ctrl+Space** for content assist. In the list of possible entries, choose (double-click, or select and Enter) **Average(T) : T**. (The syntax simply means that for any input type T, the output value will also be of type T.) This inserts `Average(T)` into the field.

__iii. Again click in the value field for avgSpeed; delete the `T` inside the parentheses and keep the cursor there. Type **Ctrl+Space** to bring up content assist, and this time choose **speed - float64**.



Custom output functions

The functions shown in content assist are *custom output functions* specific to the Aggregate operator. They are not general-purpose SPL functions. Every output assignment must contain a call to one of these. The automatic assignments for the non-numeric attributes described above implicitly call the **Last(T)** aggregate function.

__5. Specify where the results go.

__a. In the editor, select the newly added operator **FileSink_7**. In the **Properties** (**General** tab), rename the operator to `AvgWriter`.

__b. In the **Param** tab, set the **file** parameter to `"average.speeds"` (with the double quotes).

__c. Click **Add…**; in the **Select parameters** dialog, check **format** and **quoteStrings**; click **OK**. Set **format** to `csv` and **quoteStrings** to `false`.

__d. Save. Close the **Properties** view. Your application is ready to launch.

__6. Launch the application. Right-click **MyMainComposite** in the **Project Explorer** and choose **Launch…**. Click **Apply** (if necessary) and **Continue** in the **Edit Configuration** dialog.

3.3 The Streams Console

The Streams Console is general-purpose, web-based administration tool. Each Streams domain has its own console environment; the console interacts with one specific domain at a time, based on its *Streams Web Service* (SWS) URL. In addition to managing and monitoring instances, resources, jobs, logging and tracing, and many other administrative things, it serves as a simple data visualization tool. It is not intended to be a production-quality dashboard, such as Cognos, but mainly a useful facility for monitoring applications and understanding data during development.

There are several ways to launch the Console, including from the **Task Launcher for Big Data**. You encountered the Console in section 1.1 on page 8, where you opened it with a desktop launcher. You can also look up the URL and open it directly in Firefox or any other browser, from any machine with https access to the server running Streams. Normal user authentication and security apply.

If you still have the Console open in the Firefox browser, skip step __1. below.

__1. In the **Streams Explorer**, expand **Streams Domains**. Right-click on **StreamsDomain** (the only domain listed) and choose **Open Streams Console**. Log in with `streamsadmin` / `passw0rd`.

The initial view monitors the domain. Each of the cards shows a specific type of object (PEs, jobs, instances, and so on), with a graphical view that lets you see at a glance what is going on. At the top is a navigation bar with buttons that show a count of objects and their state (healthy/unhealthy or stopped/starting/running) and let you get quickly to a monitoring view for that object.

Figure 3 below shows a snapshot highlighting some of the graphically depicted information. For example, the PEs card shows quickly which PEs consume very little memory and CPU (in the bottom left of the chart) and which consume (relatively) a lot. This lets a developer identify quickly which operators to focus on during performance optimization.

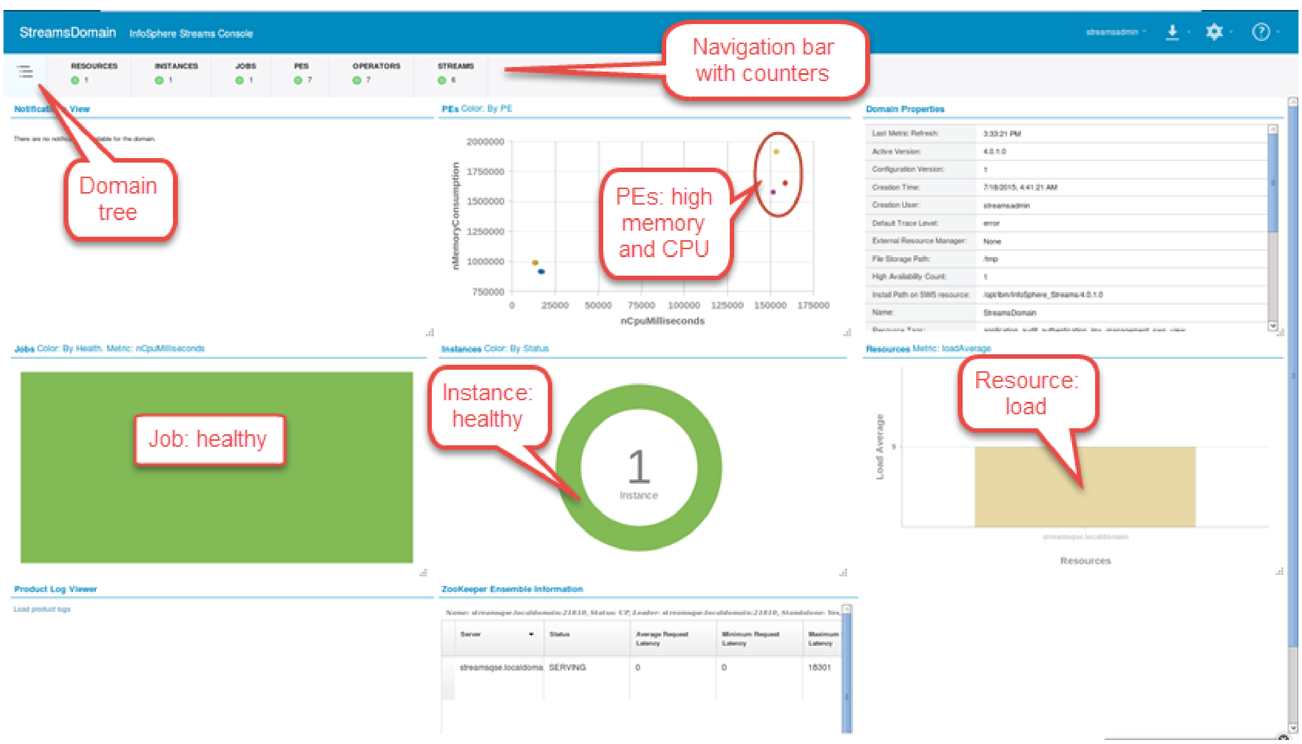
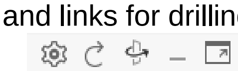


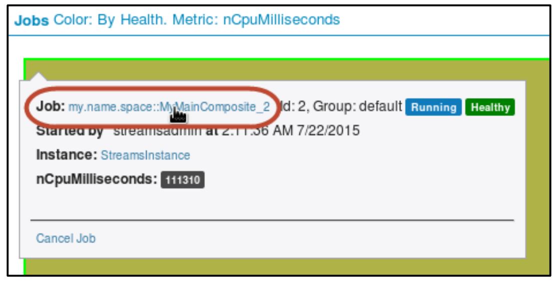
Figure 3. Monitoring the domain in the Streams Console

With only a single job running in a single instance on a single resource, many of the graphics are not very interesting, but they are extremely useful when managing a real cluster with many running jobs. Hovering over the graphic in each card pops up a panel with detailed information and links for drilling down further. Also while hovering, controls appear in the top right of the card:  **Card Settings** (color schemes, filters, and other settings appropriate for the information shown), **Refresh**, **Card Flip Action** (to show the tabular data behind a graphic), **Stack** (minimize the card), and **Max** (maximize the card). Not all cards have all controls.

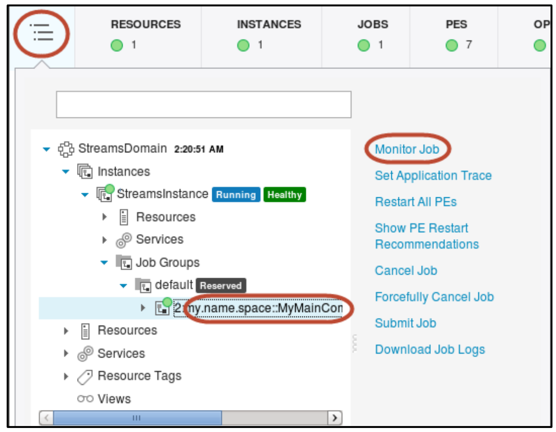
3.4 Monitoring a job

Let’s look closer at your running application. There are three ways to get from the domain to the job:

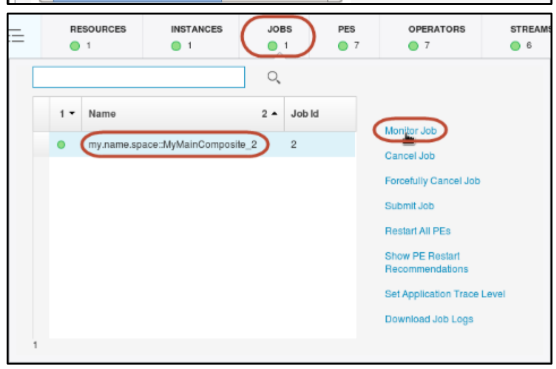
- Hover over the **Jobs** card and click the **Job** link:



- Hover over the first button in the navigation bar at the top of the page, to open the **domain tree.** This is always available, regardless of which page is open. Expand **StreamsDomain > Instances > StreamsInstance > Job Groups > default**, and select the right (or only) job. The available actions change with the selection; click **Monitor Job**.



- Hover over the **Jobs** counter in the navigation bar. In the action pane, see that the right job is selected (automatic if there is only one), and click **Monitor Job**:



Keep the data flowing

For the steps in the rest of this lab, make sure the tuples keep flowing. See step __6. on page 33 to remind you how to do this. Add enough copies of `all.cars` to the `data` folder to last the time you take, at 45 seconds per copy. For example, 20 copies will last 15 minutes. If you run out, do it again.

If you’re concerned about having too many copies, you can safely delete them once they have been read, but **make sure you always keep the original**!

The Jobs view shows the PEs card familiar from the Domain view, along with some other cards specific to jobs. Most interesting for now are the **Streams graph** and the **Tuples Transmitted/Processed Delta** cards.

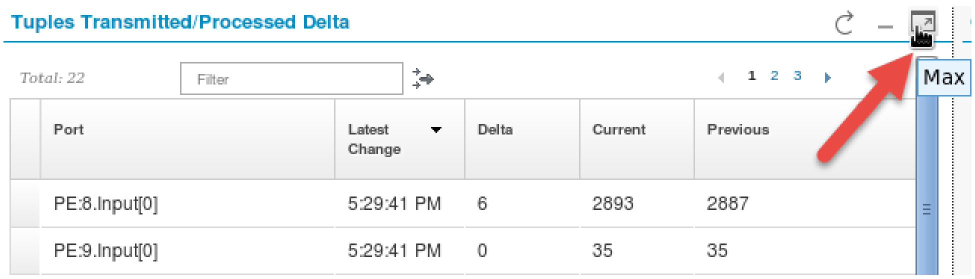
The **Tuples Transmitted/Processed Delta** card shows, the cumulative total number of tuples received and sent through all the PE ports in the job, along with the change (*Delta*) since the last refresh. By default, the refresh interval is three seconds, so the per-second tuple rate is the delta divided by three.

The table has a row for each input and output port of each PE and each operator. In this lab, each operator has its own PE, so there is a one-to-one correspondence between PE ports and operator ports, and there are two rows representing each port and showing the same metrics.

Remember PEs?

As described in step__2. on page 32, a PE or Processing Element is essentially a runtime process, encapsulating one or more operators. Where the operator is the logical unit of operation, the PE is the unit of execution at runtime.

__1. Maximize the **Tuples Transmitted/Processed Delta** card: click on the **Max** control that appears in the top right when the cursor is over the card.



__2. The job has seven PEs (and seven operators). Three of them lack either an input port (FileSource) or an output port (FileSink); the rest has exactly one input and one output port, for a total of 11 ports. Since each is represented twice, that means 22 rows. By default, the table shows only ten rows per page, requiring three pages; it is better to see all rows on one page (scrolling if necessary).

__a. Click **25** (or any higher choice) at the bottom of the table.

__b. To see the ports with the highest tuple rates together at the top, click the **Delta** column header **twice**; this sorts the Delta column in descending order. Notice that the top delta is approximately 120, which means a tuple rate of 40 per second—exactly the rate set in the Throttle operator.

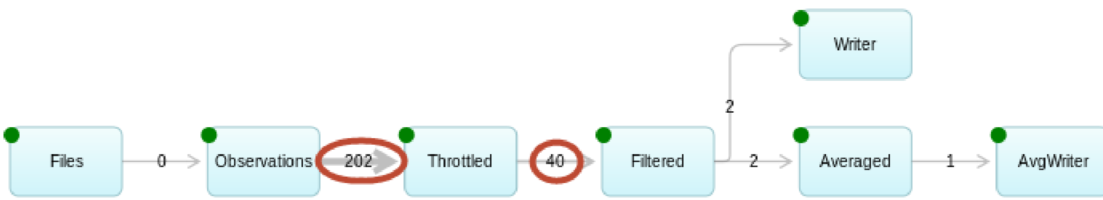
Port	Latest Change	Delta	Current	Previous
PE:10.Input[0]	6:48:17 PM	120	104698	104578
PE:12.Input[0]	6:48:17 PM	120	104699	104579
PE:12.Output[0]	6:48:17 PM	120	104698	104578
PE:10.Filtered.Input inPort0Alias	6:48:17 PM	120	104698	104578
PE:12.Throttled.Input Observations	6:48:17 PM	120	104699	104579
PE:12.Throttled.Output Throttled	6:48:17 PM	120	104698	104578
PE:8.Input[0]	6:48:17 PM	7	5230	5223

__c. Restore the card to its original size by clicking the **Max** control again.

You may have noticed that another port, lower down, occasionally has much higher deltas (over 600). This is the output port of the Observations (FileSource) operator, which connects to the input of Throttled. Can you explain this intermittent behavior? The next steps explore this further.

__3. Maximize the **Streams graph** card. You can also enlarge it, using the resize handle at the bottom right of the card, just enough to show the entire graph.

The graph is familiar from the **Instance Graph** in Studio, though it represents information in slightly different ways. It labels every stream with the tuple rate, and indicates operator health by a colored dot. As in the Instance Graph, relative tuple rate sets the thickness of the arrow. Usually the Throttled stream, at 40 tuples per second, is the thickest, but every so often the Observations stream, normally at zero, exceeds it (at over 200). This is the same behavior you may have observed in the Delta table, above.

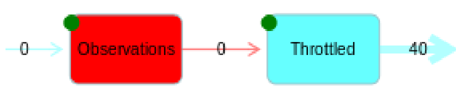


The main reason for looking at the Streams graph, other than to detect trouble (unhealthy PEs), is to identify bottlenecks that may affect throughput performance. A bottleneck is an operator that limits the flow of tuples, usually because it cannot process any more tuples per second with the CPU cycles it has. Its input port buffer fills up; eventually the operator *upstream* (to the left in the Streams graph) has to stop sending tuples; that one stops processing, causing its input buffers to fill up, and so on until this *back-pressure* reaches the front of the application. At some point, this can lead to data loss.

A good indicator of a candidate for a bottleneck is a PE that consumes lots of CPU cycles: it would be to the far right in the PEs card, as indicated in Figure 3 on page 41. Of course, as long as the PE keeps up with the tuple flow, high CPU load doesn't necessarily mean bottleneck.

So how to identify the real problem? Every operator with one or more input ports reports a metric called congestionFactor, which means how much of the available buffer space has filled up. In terms of percentages, 100 means total congestion; 0 means no congestion at all.

__4. In the **Color Scheme** dropdown of the Streams graph card, find and click **congestionFactor (PE connection)**; it's near the bottom of a long list of available color schemes.



As long as there are files left to read, the Observations operator will be red in this color scheme, indicating extreme congestion (84 to 100 percent). The rest of the operators is light blue, indicating low or no congestion; FileSinks are not colored: they have no output ports and do not support this metric.

What does this mean? The Observations stream (output of the Observations operator) is highly congested; Throttled looks fine, but it is in fact the cause of this congestion. This is exactly what you should expect, because Throttled is deliberately holding up tuples, to produce them at the prescribed rate. In other words, Throttled is the bottleneck, but by design. Observations, a FileSource, reads the file as fast as possible, all 1900-plus records at once, but when Throttled's input buffer is full, it must hold on to those records until that buffer, being emptied at 40 tuples per second, has space available again.

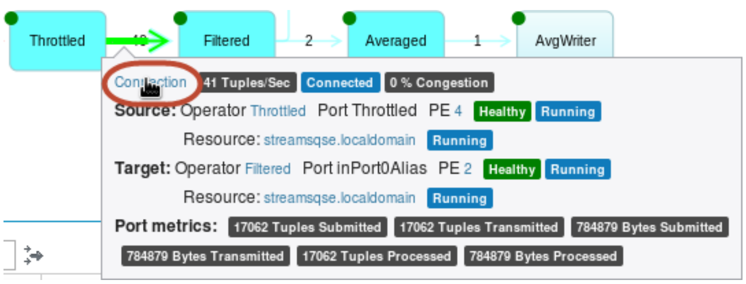
An artificial bottleneck

In most “real” applications, there is no place for a Throttle operator; you only used it here to simulate a data source with a manageable flow rate. It should be clear, however, that the Streams graph display with this color scheme is an important tool for identifying real bottlenecks, which often are a cause for concern.

3.5 Visualizing data

Data is carried in tuples; tuples flow on streams. To view data, you have to monitor a stream.

__1. In the **Streams graph**, hover over the **Throttled** stream (from Throttled to Filtered). In the panel that pops up, click **Connection**.



You will only work with three of the cards in this view: **Data Visualization Chart**, **Data Visualization View** (meaning a table), and **Tuples Per Sec Flow Rate**. As you work with each, you can resize or maximize the card as you see fit, and stack (minimize) any other cards.

__2. In the **Tuples Per Sec Flow Rate** card, adjust the scale and color scheme to fit the expected tuple rate of about 40. The default settings are for much higher rates.

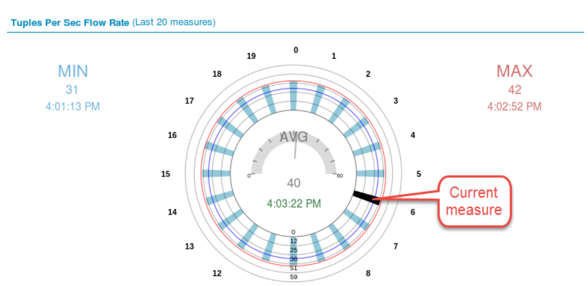
__a. Click ⚙ **Card Settings**, and then **Edit Color Scheme**.

__b. In the **Configure color scheme** dialog, change the **Upper limit** to 60; click **OK**.

__c. Change **Measures** from 60 to a smaller number, like **20**. A minute is long enough.

__d. Click ⚙ **Card Settings** again to get rid of the edit area and see the entire graphic.

The dial shows the tuple rate in the length of radial bars for each of the last 20 sampling intervals. The current one is black, hopping every three seconds to the next slot; the historical ones are colored according to the scheme you defined.



__3. The **Data Visualization** card is comparable to the **Show Data** views in Studio.

__a. Click **Card Settings**, and then **Create View**.

__b. In the **Basics** tab, change the **View Name** to something more descriptive: for example, `Selected cars`. Edit the **View Description** if you like, but this is not important.

__c. In the **Filter** tab, check **Enable Filtering** and choose **id:rstring** for the **Filter by** setting.

__d. The **Value Expression** field takes a regular expression for specifying complex patterns. Enter **`C11.`** (with the period) to select all `ids` with "C11" followed by any character. In your data, this selects 10 `ids`, from C110 through C119.

__e. In the **Buffer** tab, set the **Tuples/sec throttle** to **20** (this is the maximum). Because each car reports about once per second and you have selected 10 cars, this should allow the view to keep with all selected tuples without throttling (subsampling). In **Buffer Size Configuration**, keep the **Limit By** option of **Tuple count** and set it to **50**.

__f. Click **OK**. Click **Card Settings** again to dismiss the action menu and show the whole table.

__g. Explore the table. The whole table contains 50 rows. Click **25** at the bottom of the card to make two pages. Click the **id** column header to sort by id and see a short history of each car. Click the **time** column header (the last one, not the first column **Time**, which is not an attribute but a column generated by the view) **twice** to see the latest tuples at the top. Resize columns and scroll as needed.

Data Visualization View: Selected cars

Total: 49 1 2

Time	heading	id	latitude	longitude	speed	time
6:42:20 PM	81.943	C110	37.7927673	-122.4035154	36	1363818372000
6:42:20 PM	225.019	C111	37.780505	-122.3999971	46	1363818372000
6:42:20 PM	44.972	C112	37.7866777	-122.3921911	36	1363818372000
6:42:20 PM	224.931	C113	37.7802707	-122.4054218	47	1363818372000
6:42:20 PM	45.234	C114	37.7823825	-122.3965049	36	1363818372000
6:42:20 PM	315.292	C115	37.7844809	-122.3953885	46	1363818372000
6:42:20 PM	135.35	C116	37.7844065	-122.3975276	36	1363818372000

10 | 25 | 50 | 100 | All

__4. The **Data Visualization Chart** depicts numerical data graphically. Because the stream contains data for multiple cars (multiplexed) and it does not make sense to mix them together into a single chart, you will have to pick one car.

__a. Click **Card Settings**, and then **Create Chart**.

__b. In the **Attributes** tab, uncheck **tuple** to clear all checks; then check **speed:float64** and **heading:float64**.

Even though the two attributes have different units (km/h and degrees), their ranges are close enough to show reasonably well on the same scale, at least for some cars.

__c. In the **Filter** tab, check **Enable Filtering** and choose **id:rstring** for the **Filter by** setting. In the **Value Expression** field, enter `C117` (a good choice for showing speed and heading together).

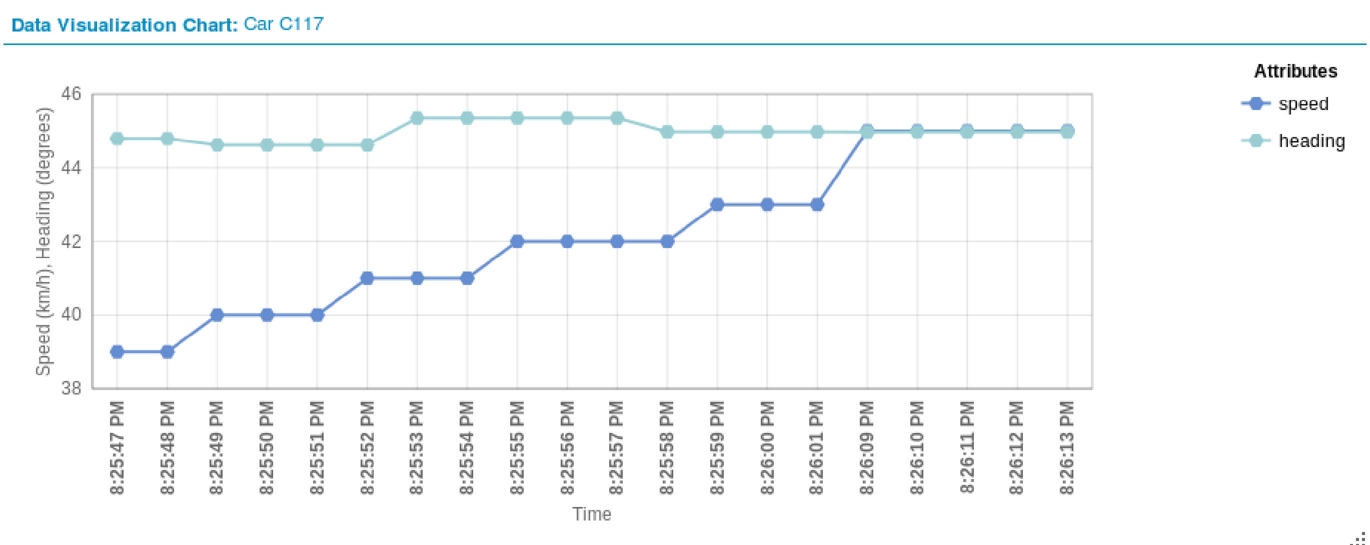
__d. In the **Buffer** tab, set the **Limit By Tuple count** value to **20**. Because you selected a single car, the default **Tuples/sec throttle** value of 1 is acceptable.

__e. In the **Chart** tab, set the Y-Axis Title to `Speed (km/h), Heading (degrees)`.

__**f.** In the **Basics** tab, set the **Chart Name** to something simple, like `Car C117`.

__g. Click **OK**. Click **Card Settings** again to dismiss the action menu and show the whole chart.

You now can gain more insight in your data by inspecting it visually. For example, it becomes clear that the heading makes big jumps between relatively constant periods; this makes sense in an American city, where most turns are 90 degrees. Also, the speed, while in principle a measured quantity best represented by a floating-point number, is only reported as whole numbers (this is also obvious from the table view), lending a stair-step quality to the plot; this may mean that some efficiency can be gained by changing the attribute type to `int32` or even `int16` or `int8`—provided that it can be verified that no source will ever have non-integer speed values.



When you are ready to move on to the next lab, close or minimize the browser.

Lab 4 Modular application design with exported streams

4.1 Building on the previous results

This section is optional
If you are confident that your results from the previous lab are correct, you can continue working with them and skip this section.

To make sure everyone continues from the same point, go to the next Studio workspace that has been prepared for you.

__1. In the Studio top menu, choose
File > Switch Workspace > /home/streamsadmin/Workspaces/workspace4.

Studio shuts down and restarts with the new workspace. A project with the same application you built in Labs 1 through 3 is already available

__2. The editor view shows the Task Launcher for Big Data; click on the tab for **MySourceFile.spl**.

In the graphical editor's palette, right-click **Toolkits** and uncheck **Show All Toolkits**. (This setting reverts to the default of showing all when you close and reopen Studio.)

Rearrange the graph as needed, using **Layout** and **Fit to Content**.

__3. Show the **Instance Graph**:

__a. In the **Streams Explorer**, expand **Streams Instances**

__b. Right-click on **default:StreamsInstance@Streamsdomain** and choose **Show Instance Graph**.

You will see any jobs from the previous labs that are still running. Cancel or hide them as you see fit.

4.2 Adding a test for unexpected data

It is generally good practice to validate the data you receive from a feed. Data formats may not be well defined, ill-formed data does occur, and transmission noise can creep in as well. You do not want your application to fail when the data does not conform to its expectations. In this lab, when you will be receiving live data; who knows what it will do to your analytical process?

As an example of this kind of validation, you will add an operator that checks one attribute, the vehicle ID (**id**). In the data file all.cars, all records have an id value of the form "C*nnn*" (presumably, with "C" for "car"). Even though it doesn't at the moment, assume that your application depends on this format; for example, it could take a different action depending on the type of vehicle indicated by that first letter (say, "B" for "bus"); and there may be a system requirement that all vehicle IDs be exactly four characters long.

Rather than silently dropping the tuple, however, it is better practice to save the “bad” data so you can audit what happened and later perhaps enhance the application.

In summary, the specifications are as follows:

Table 5. Vehicle ID (`id` attribute) specifications

Criterion	Value
First character	**“C”**
Length	**4**

In other words, if any data comes in with an unexpected value for **id**, your program will shunt it aside as invalid data. There are several operators that can take care of this; which one you use is to some degree a matter of taste. You have already used one that fits the bill, the **Filter**. So let’s use a different one here.

The **Split** operator is designed to send tuples to different output ports (or none) depending on the evaluation of an arbitrary expression; this expression can, but does not have to, involve attribute values from the incoming tuple. It can have as many output ports as you need. In this case you only need two: one for the regular flow that you’ve been dealing with (the “valid” values), and one for the rest (the “not valid” ones).

The **Split** mechanism works as follows (see the example in Figure 4, below);

- The N output ports are numbered 0, 1, …, N-1.
- A parameter called **index** contains an arbitrary expression that returns a 64-bit integer (an **int64** or a, if unsigned, a **uint64**).
- This expression is evaluated for each incoming tuple.
- The expression’s value, *n*, determines which output port *p* the tuple is submitted to:
  - If $n \geq 0$, $p = n$ modulo N
  - If $n < 0$, the tuple is dropped

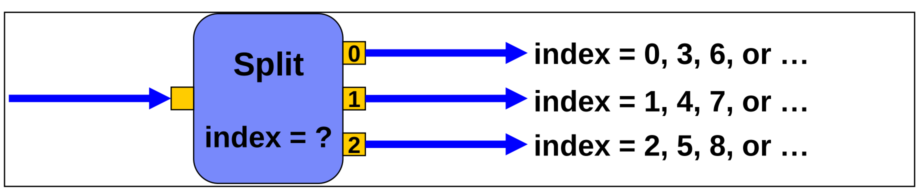


Figure 4. A Split operator with three output ports.

Remember

At any time during the steps below, use **Layout** and **Fit to Content** to keep the graph organized and visible.

__1. Add a **Split** operator to the graph.

__a. In the graphical editor, find the **Split** operator, either by using the **Find** box or browsing down to **Toolkits > spl > spl.utility > Split**.

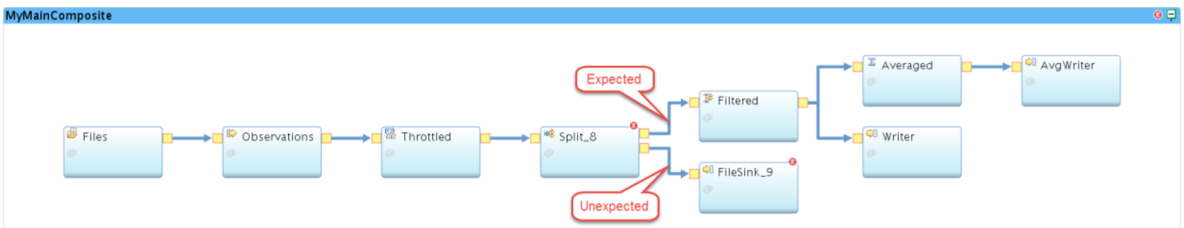
For convenience, try the template **Split with index** instead of the generic version. It comes preconfigured with the parameter and second output port that you need.

__b. Drag a Split operator into the canvas and drop it directly onto the **Throttled** stream (output of the Throttled operator).

__2. Add a **FileSink** to the graph; drag a **stream** from the second output port of the **Split** operator to the input of the new **FileSink**.

Notice that the new streams from both of the Split's output ports are solid, not dashed: they automatically inherit the schema from the original Throttled stream, so there is nothing to configure other than their names.

__3. Edit the properties of each of the new streams; in the **General** tab, rename the first stream (the one that goes to **Filtered**) to `Expected`; rename the second to `Unexpected`.



__4. Configure the **Split** operator. Because it has two output streams, it is better to set a descriptive alias than to blank it out; otherwise, it would be known by the name of the first output stream (`Expected`), which is somewhat misleading.

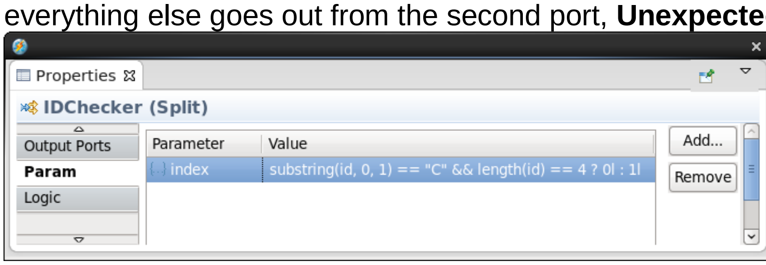
__a. Edit the operator properties. In the **General** tab, **Rename** it to `IDChecker`.

In the **Param** tab, the parameter you need is already there, with an invalid placeholder value of `indexExpression`.

__b. In the **Value** field for the **index** parameter, type the following **exactly** (NOTE: the "`l`" after `0` and `1` is a lowercase letter ell, to indicate a "long", 64-bit integer):

```
substring(id,0,1) == "C" && length(id) == 4 ? 0l : 1l
```

How to read this? "If the substring of the `id` attribute starting at offset zero with length one (in other words, the first character of `id`) is "C" and the length of the `id` attribute is four, then zero; otherwise one." So, proper IDs go out from the first port (**Expected**), and everything else goes out from the second port, **Unexpected**.



SPL expression language syntax

The syntax <boolean-expression> **?** <action-if-true> **:** <action-if-false> is known from C, Java, and other languages. The functions substring(string,start,length) and length(string) are from the Standard Toolkit. The suffix "l" (the letter ell) indicates that the numbers are 64-bit values ("long" integers). SPL does not make implicit type conversions; integer numbers with no suffix are 32-bit values, and assigning one to a 64-bit parameter would raise an error.

__5. Configure the new **FileSink** operator. You've used a FileSink in two previous labs, so refer back to those if you forgot how to do it.

__a. First **Rename** it to `ErrWriter`.

__b. Set the following parameter values:

Table 6. Parameter values for FileSink_9. Note the extra parameter `flush`.

Parameter	Value
file	`"error.observations"`
flush	`2u`
format	`Csv`
quoteStrings	`False`

Flushing buffered file writes

FileSink performs buffered file I/O, meaning that it writes to buffers maintained by system libraries rather than directly to disk. These buffers are only written out to disk (*flushed*) as they fill up, or when the requesting application terminates. When the output is a slow trickle, this can mean that you will not see anything in the file for a long time. Setting `flush` to `2u` (the `u` is for "unsigned" integer) guarantees that you will see data at least in batches of two records.

__6. Save, launch the app, and verify that the original output files, **filtered.cars** and **average.speeds**, are being written to the data directory as before, and that the new output file (**error.observations**) has at least two records in it, after a suitable amount of time: the input file contains two records with a malformed ID (and other abnormal values as well).

4.3 Splitting off the ingest module

Now it gets interesting. Within a Streams application, data flows from operator to operator on **streams**, which are fast and flexible transport links. The Streams application developer is not concerned with how these are implemented. They may work differently between operators running on different hosts, in different PEs on the same host, or within the same PE, but the logic of the graph stays the same. When an application needs to exchange data with the outside world, that requires the explicit use of **source** and **sink** operators—for file I/O, ODBC, TCP, UDP, or HTTP connections, message queues, and so on.

For Streams applications running in the same instance, however, another mode of data exchange is possible: **Export** and **Import**. An application can export a stream, making it available to other applications running in the instance; one or more applications can import such a stream, based on flexible criteria. Exported streams, once they are connected, are just like all the other streams that run between PEs within an application—fast and flexible. It's only at the time a job is submitted or canceled that the runtime services get involved to see which links need to be made or broken; once that's done, there is no difference in runtime behavior (well, almost none, but the difference is beyond the scope of this lab), and there is no performance penalty.

But there is a tremendous gain in flexibility. Application stream connections can be made based on publish-and-subscribe criteria, and this allows developers to design completely modular solutions, where one module can evolve and be replaced, removed, or replicated, without affecting the other modules. It keeps individual modules small and specialized.

In the labs so far, you have built a monolithic app, but there is a logical division. The front end of the app, from DirectoryScan to Throttle, is concerned with reading data, in this case from files, and "replaying" that data in a controlled fashion to make it look like a real-time feed. The rest of the app, from Split to FileSinks, performs analysis and writes out the results. If you split off the front end into a separate "Ingest" module, you can imagine that it would become easy to have another module, alongside it or as a replacement, that produces tuples that have the same structure and similar contents, but that come from a completely different source. And that is exactly what this lab will do: add another module that reads a live data feed and makes the data available for processing by the rest of this application.

__1. In the graphical editor, drag a **Composite** and drop it on the canvas. Not on the existing main composite, but outside of any graphical object. The editor will call it `Composite`; rename it in the **Properties** view to **FileIngest**.

Notice that the new composite appears in the Project Explorer, but it does not have a build associated with it. Create one.

__2. In the **Project Explorer**, right-click the **FileIngest** main composite. Choose **New > Distributed Build**. In the dialog that pops up, change the **Configuration name** to `Distributed` (assuming it was called `Distributed1`), accept all other defaults, and click **OK**.

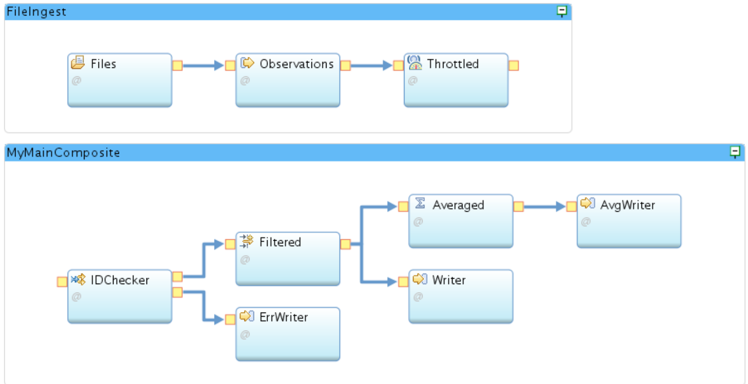
__3. Move the three front-end operators from the old main composite to the new.

__a. In the original main composite, select the three operators **Files**, **Observations**, and **Throttled**.

To do this, hold down the **Ctrl-key** while clicking each one in turn. Cut them (**Ctrl+X** or right-click, **Cut**) to the clipboard.

__b. Select the FileIngest composite; paste the three operators in (**Ctrl+V** or right-click, **Paste**).

You now have two applications (main composites) in the same code module (SPL file). This is not standard practice, but it does work. The applications, however, are not complete: you have broken the link between **Throttled** and **IDChecker**.



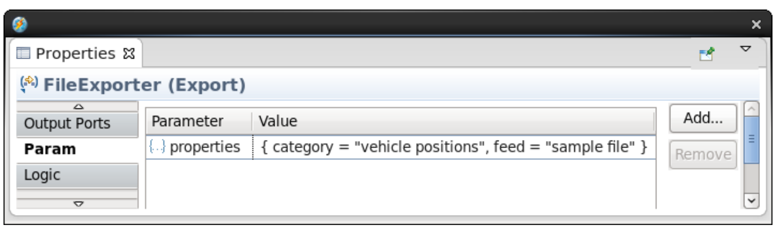
__4. Set up the new application (**FileIngest**) for stream export.

__a. In the palette, find the **Export** operator and drop one (the template **Export with Properties**, if you like) into the **FileIngest** composite.

__b. Drag a stream from **Throttled** to the **Export** operator. Note that the schema is remembered even while there was no stream, since it belongs to the output port of **Throttled**.

__c. Edit the **Export** operator's properties; **Rename** it to `FileExporter`.

__d. In the **Param** tab, edit the value for the **properties** parameter. (You have to add it first, if you did not use the Export with Properties template.) In the **Value** field for **properties**, enter the following "tuple literal":
`{ category = "vehicle positions", feed = "sample file" }`



What this does is "publish" the stream with a set of properties that are completely arbitrary pairs of names and values (similar to tags on a web page). The idea is that an importing application can look for streams that satisfy a certain *subscription*: a set of properties that need to match.

__e. Save. The **FileIngest** application builds, but the old one still has errors.

__5. Set up the original application for stream import.

__a. In the palette, find the **Import** operator and drop it into the old main composite. Just use the generic one.

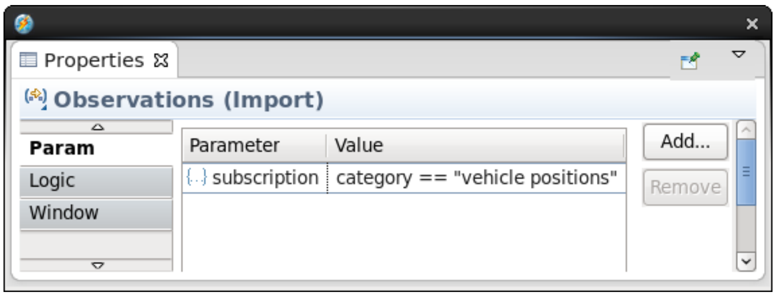
__b. Drag a stream from **Import_11** to **IDChecker**.

__c. Assign a schema to this stream, by dragging and dropping **LocationType** from the palette. **Rename** it to `Observations`. (There is already another stream called Observations, but it is now in a different main composite, so there is no name collision.)

__d. Select the **Import** operator and **Rename** it to Observations by blanking out the alias.

__e. In the **Param** tab, edit the Value for parameter **subscription**. Replace the placeholder `parameterValue`, with the following boolean expression:

```
category == "vehicle positions"
```



Notice that this is only looking for one property: the key `category` and the value `"vehicle positions"`. It is perfectly fine to ignore the other one that happens to be available; if the subscription predicate is satisfied, the connection is made (as long as the stream types match).

__f. Save.

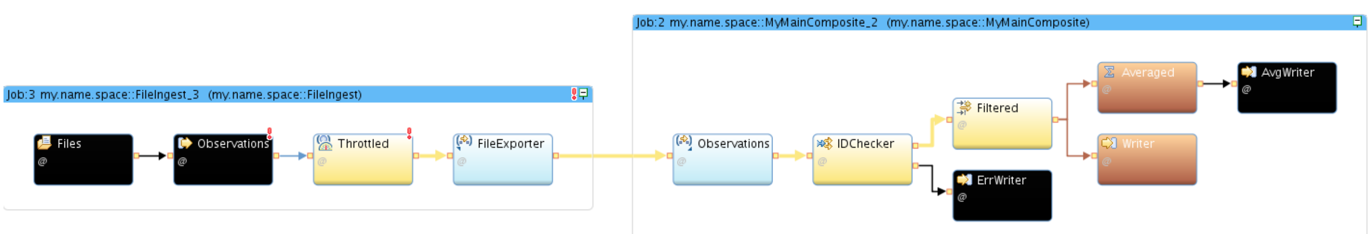
__6. Test the new arrangement of two collaborating applications.

__a. In the **Instance Graph**, cancel any remaining jobs. Set the color scheme to **Flow Under 100 [nTuples/s]**. Enlarge the view so you can comfortably see the two jobs.

__b. In the Project Explorer, launch the old application, **MyMainComposite**.

__c. Launch the new application **FileIngest**.

Notice that the tuples flow from operator to operator throughout the instance graph, even though they are divided into two main composites. Leave the two applications running; you'll be adding a third.



4.4 Adding a live feed

Note

This section assumes that you have internet connectivity. If your lab environment is not connected, you can still go through the steps of importing and launching the application and seeing the connections being made, but no live data will flow. In most cases, there will be no other symptoms.

In instructor-led events in venues where participants' machines have no connectivity, the instructor may go through this section as a demonstration rather than an exercise for everyone to complete.

Rather than building a live-data ingest application from scratch, you will i**mport a Streams project** that has already been prepared. This application uses an operator called **HTTPGetXMLContent**, from a version of the **com.ibm.streamsx.inet Toolkit** that can currently only be found on GitHub, to connect to a web services feed from **NextBus.com** and periodically (every 30 seconds) download the current locations, speeds, and headings of San Francisco Muni's buses and trams. It parses, filters, and transforms the data and makes the result look similar to the file data—though some differences remain. It exports the resulting stream with a set of properties that match the subscription of your processing app; when you launch the **NextBus** app, the connection is automatically made and data flows continuously until you cancel the job.

__1. Before you can use the NextBus project, you must tell Studio where to find the version of the com.ibm.streamsx.inet toolkit that it depends on.

__a. In the **Streams Explorer**, expand **InfoSphere Streams Installations [4.0.1.0] > InfoSphere Streams 4.0.1.0 > Toolkit Locations**.

__b. Right-click **Toolkit Locations** and choose **Add Toolkit Location…**.

__c. In the **Add toolkit location** dialog, click **Directory…** and browse to **My Home > Toolkits**. (My Home is all the way at the top; the dialog comes up in the completely separate Root tree.) Select **Toolkits** and click **OK**.

__d. Click **OK** again.

If you expand the new location, **(Local) /home/streamsadmin/Toolkits**, you see **com.ibm.streamsx.inet[2.5.0]**. This is different from the version of this toolkit that is installed with Streams (2.0.1), so the NextBus project can select the right one by version. (You'll find the 2.0.1 version under the location STREAMS_SPLPATH.)

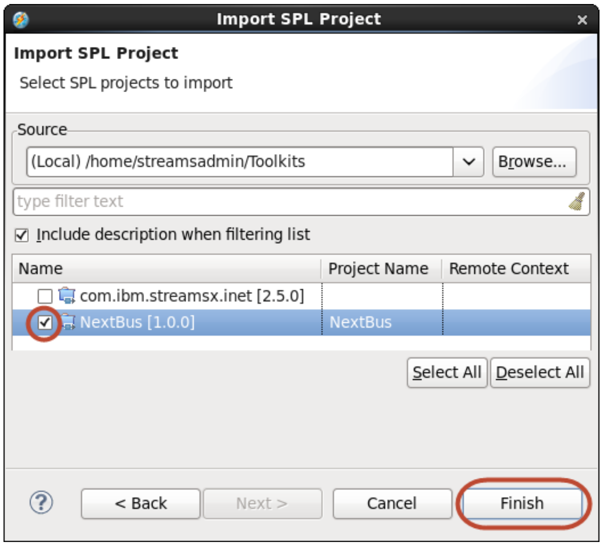
__2. Import the NextBus project.

__a. In the top Eclipse menu, choose **File > Import…**.

__b. In the Import dialog, select **InfoSphere Streams Studio > SPL Project**; click **Next >**.

__c. Click **Browse…**; in the file browser, expand **My Home** and select **Toolkits**. Click **OK**.

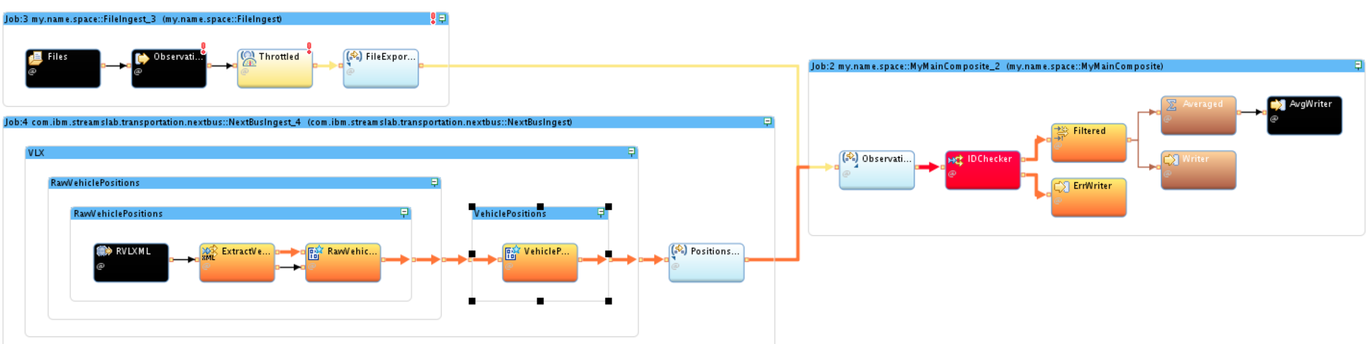
__d. Select **NextBus** and click **Finish**.

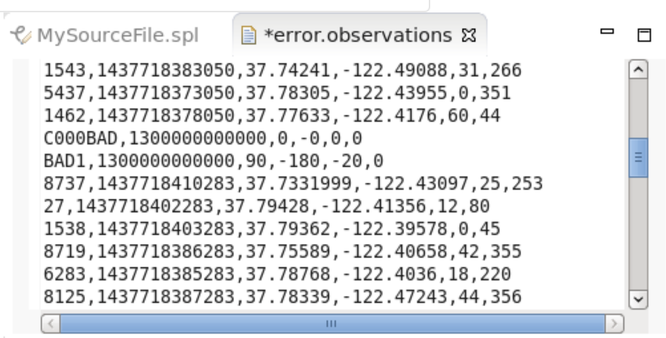


__3. Expand project **NextBus** and namespace **com.ibm.streamslab.transportation.nextbus**. Launch application **NextBusIngest** (you may have to wait till the project build finishes).

__4. Maximize and organize the Instance Graph. If you want, you can expand the nested composites in the NextBusIngest job.

You should see the three applications all connected; tuples flow from the **FileIngest** job whenever you copy another file into the **Data** directory. Tuples flow from the **NextBus** job in 30-second bursts. The **error.observations** file gradually fills with records from **NextBus**: their vehicle IDs do not conform to the “C*nnn*” format. (Refresh the file periodically: click in the editor showing the file and click **Yes** in the **File Changed** dialog, which comes up when Studio detects that the underlying contents have changed.)



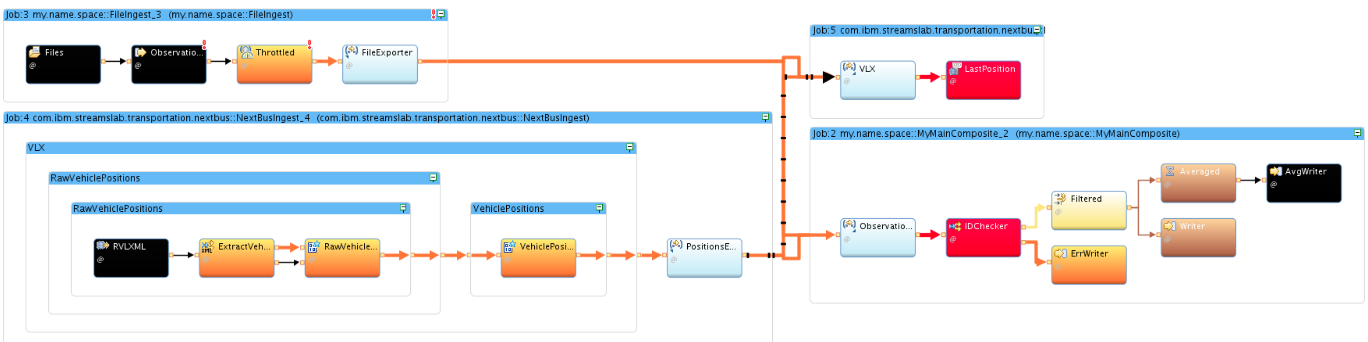


4.5 Showing location data on the map

The NextBus toolkit comes with another application that lets you view data in a way that is more natural for data that involves locations: on a map. Just like MyApplication, it is designed to connect to the kind of stream exported by NextBusIngest and FileIngest. Without any further configuration, it can take the latitude and longitude values in the tuples, along with an ID attribute, and generate an appropriate map.[1] It is not the prettiest or most dynamic of maps, but real cartography is a lot of work, and this is only intended as a quick method for learning about your data—similar to the views and charts in the Streams Console.

__1. In the NextBus toolkit, launch **NextBusVisualize**. In the **Edit Configuration** dialog, scroll down to see the **Submission Time Values**; note the value of the **port** variable: **8080** (widen the Name column to see the full name).

In the Instance Graph, each of the two exported streams is connected to each of the downstream jobs. The arrows look a bit confusing, but if you select each of the branches in turn, you can untangle them.



__2. Open the Firefox **browser**; go to http://streamsqse.localdomain:8080/streamsx.inet.resources/.

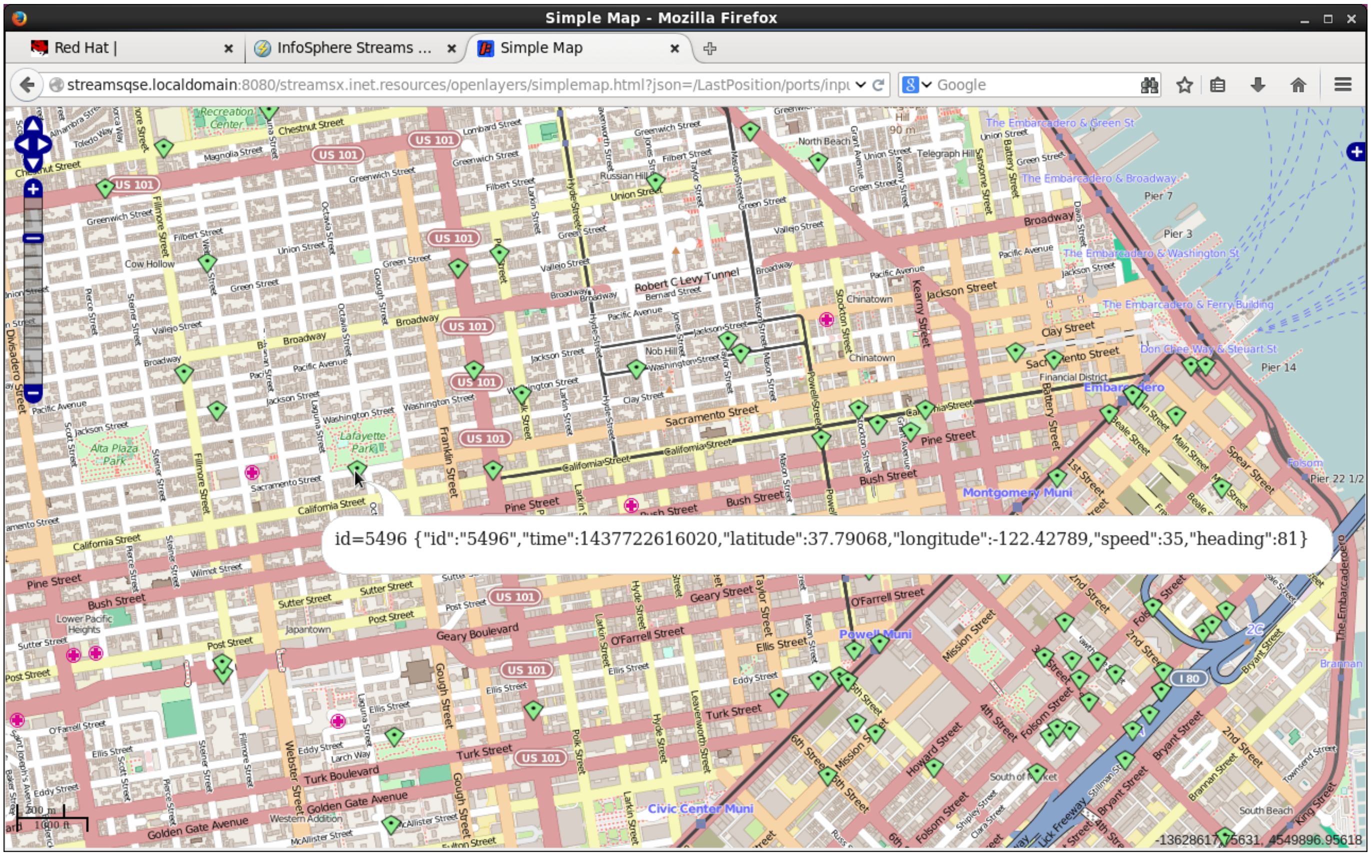
__3. This loads a simple information page; click **/streamsx.inet.resources/dojo/viewall.html** - Live table views for all exposed output ports

__4. This shows a table with available links; click **Live Map**.

__5. You will see a map of the San Francisco Bay Area, with a large number of green markers crowding the city of San Francisco. Use the map controls or mouse wheel to **zoom in** and (hold down the left mouse button to drag and center the map), so you can see the individual vehicles. They jump around as their locations are updated: the map is live!

The map refreshes every 5 seconds, but remember that NextBus data is only updated every 30 seconds. Hover over or click any one of the markers to get the full list of attributes for that vehicle.

[1] Maps from OpenStreetMap (http://www.openstreetmap.org), via OpenLayers (http://openlayers.org).



4.6 Putting it all together

This lab has barely scratched the surface of what Streams is capable of. Apart from a very small number of SPL expressions, no coding was involved in building a progressively more interesting application. Ultimately, with the help of a prepared but still simple additional module, this set of applications is able to handle a continuous flow of live vehicle location observations, and distinguish between different kinds of content. Simply by using a few building blocks, sketching a graph that governs how the tuples flow, and setting some parameters in a property view, the beginnings of a reasonably powerful solution have taken shape.

Your next steps are to decide whether this application development platform for streaming data can help you build solutions that enhance your business; whether what you have seen gives you a sense that it can help solve problems you already have in mind; and whether your organization has access to the required development skills or should invest in building them. IBM provides formal training courses for this product, and our technical sales specialists can help identify opportunities for streaming data-based solutions, organize workshops, and move to pilot projects and beyond.

HAPPY STREAMING