Microsoft

Creating and Customizing TFS Reports

John Socha-Leialoha (Murphy and Associates)

patterns & practices

Copyright

Contents

Introduction

If you've used Microsoft® Visual Studio® Team Foundation Server (TFS), you may have seen reports that you would like to customize. I recently volunteered to revise some of the standard reports for the Microsoft Solutions Framework (MSF) team at Microsoft, and I have to admit that I didn't realize what I had signed up for. It quickly became apparent that I'd volunteered for more than I was expecting. Working with reports can be very intimidating because it uses different technologies that you may not be familiar with. This article provides an introduction to the important concepts you'll need to learn a "minimal path" through the technologies.

Reporting in TFS is built on top of Microsoft SQL Server Reporting Services and Microsoft SQL Server Analysis Services. You'll also need some additional tools on top of Visual Studio, such as Business Intelligence Development Studio.

TFS Databases

Let's start by looking at how Team Foundation Server (TFS) stores information that you might want to use in your reports. The TFS database can be broken down into three stores, as shown in Figure 1, with data constantly flowing from left to right. Each of these stores is optimized for a specific type of usage, as explained later in this article.

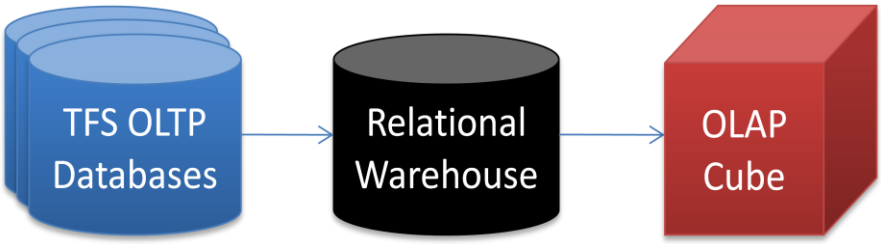


Figure 1

TFS data flow; current online data is illustrated on the far left, with historical data to its right

TFS's OLTP Database

TFS begins with an Online Transaction Processing (OLTP) store that contains all "live" data. The OLTP store contains multiple databases. Each TFS tool has its own database or set of databases, such as Work Item Tracking, Source Control, and Build. Unless you're writing a custom tool that will need its own database, you probably won't need to know about these individual databases—you just need to know the existence of an online store that contains multiple databases.

This store is designed to provide high transaction speed and data integrity. Additionally, much of the data is stored in normalized tables so the same information isn't duplicated in multiple places, which helps with data integrity and performance. However, the normalization means that the information you might want for a report is spread across many tables. If you've ever tried to make sense out of the OLTP tables used by TFS, you know what I mean—it's not easy to browse these databases. The schemas are hard to understand, and it's difficult to know which tables to join.

Query performance is also an issue. The more tables you include in a join, the slower the query runs. Additionally, you can imagine that online users might not be happy if your reporting and sleuthing slows down updating work items.

TFS Relational Warehouse

Fortunately, there is an easy way to get around these issues, using a data warehouse designed specifically for queries instead of for transactions. TFS uses a relational warehouse named, naturally, TFSWarehouse, that has a schema that is much easier to understand because it is optimized for queries and reporting instead of transactions. Additionally, this database can be on a completely different server so queries won't slow down TFS.

Data is transferred into this warehouse using TFS warehouse adapters. There is one adapter for each tool, such as Work Item Tracking, Build, and Source Control. These adapters run periodically (the default setting is every hour) to update the data in the warehouse.

For more information, see "Understanding the Data Warehouse Architecture" on MSDN.

OLAP Cube

The final database isn't a relational database at all. Instead, it's an Online Analysis Processing (OLAP) database. You access this database through Microsoft SQL Server Analysis Services and it even has its own query language named MDX. This type of database is very useful for analyzing historical data and calculating values—we'll come back to this and explain more after a discussion of some concepts that might be new to you.

Dimensions, Facts, Stars, and Cubes

The warehouse is organized using what is known as a star schema, which makes it easier to answer common questions you would pose when analyzing data. Before we get into the details of the star schema, it helps to talk a little about the types of questions you might want to ask, and how you would want queries to work. Here are some example questions you might want to ask:

- How many hours of work are remaining for a specific area of a project?
- How many bugs were opened and closed during the last week?
- Can I see a graph that shows how the remaining work has changed over the last month, plotted daily?

These are just a few examples of the type of information project managers like to see so they can judge the health and progress of a project.

Questions like these can be broken down into dimensions and facts, which are concepts central to the idea of a star schema, as you'll see soon. Facts are values, such as a number of hours, and dimensions are the parameters you're using to control what you see. Quite often, you'll also want aggregated facts, such as the total number of remaining hours for each area you've selected to view.

If you look at these questions, you'll notice they each have several different dimensions associated with them. For example, the first question uses the Area dimension, as well as Project and Work Item Type (the latter isn't directly in the question, but the question is probably asking how many hours are left in the Task work items).

You could rephrase the first question to make it explicit that dimensions and facts are used. It might look something like this:

> For a specific list of values in the Area dimension, and for a specific value in the Project dimension, and for the value Task in the Work Item Type dimension, show me the sum of the Remaining Hours fact.

Figure 2 shows a view of the facts and dimensions for Work Item History.

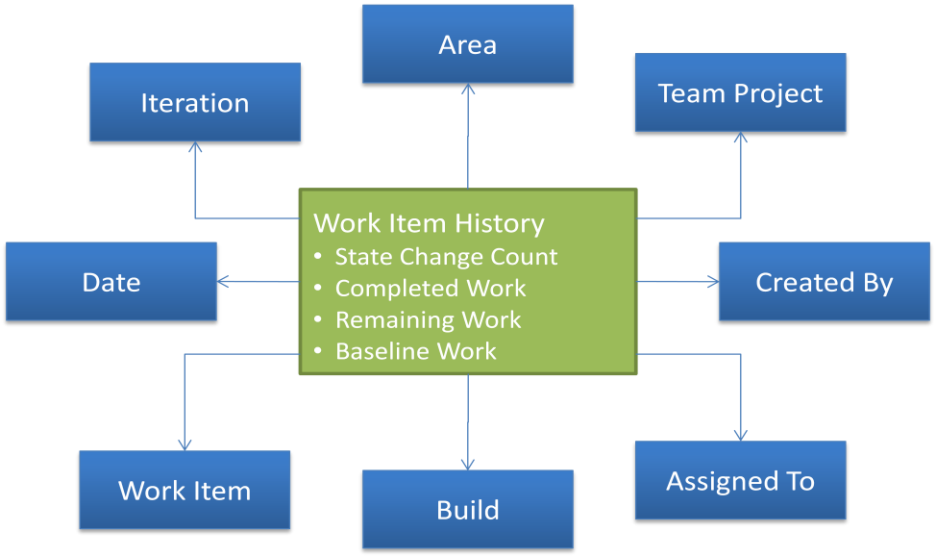


Figure 2

A subset of the star for Work Item History

The example in Figure 2 includes more dimensions and facts in the actual table, but the illustration should give you an idea of why it is referred to as a star schema.

In the example illustrated in Figure 2, Remaining Work and Completed Work are two of the facts, while Team Project, Date, Iteration, and Area are all dimensions.

TFS's relational warehouse uses different tables to store different collections of facts. For example, Work Item History in Figure 2 is the name of a table that a number of facts used to save historical information about a work item. Likewise, each dimension also has a table. Fact tables have foreign keys that link them to the different dimensions.

Rows are added to the Work Item History table each time a work item changes in the OLTP store. These rows are linked to specific values in each dimension table, allowing you to slice the data using these different dimensions.

The TFSWarehouse relational warehouse actually contains multiple stars; in fact, it also uses another schema referred to as a snowflake that is based on a star. Again, we won't get into the details of this here because, as you'll see, you probably won't need to delve into that level of detail.

Table 1 lists the different fact tables that are at the center of each star in TFS.

Table 1: Fact Tables available in the TFS relational warehouse

Fact table	Fact table
Build Changeset	Load Test Summary
Build Coverage	Load Test Transaction
Build Details	Run Coverage
Build Project	Test Result
Code Churn	vRelated Current Work Items
Current Work Item	vWork Item with Result
Load Test Counter	Work Item Changeset
Load Test Details	Work Item History
Load Test Page Summary	

Table 2 lists all the dimensions (which are implemented in tables).

Table 2: Dimensions available in the TFSWarehouse relational database

Dimension	Dimension	Dimension
Area	Iteration	Result
Assembly	Load Test Counter Dimension	Run
Build	Load Test Page Summary Dimension	Run Result
Build Flavor	Load Test Scenario	Team Project
Build Quality	Load Test Transaction Dimension	Test Category

Build Status	Machine	Today
Changeset	Outcome	Tool Artifact Display Url
Date	Person	Work Item
File	Platform	

Not all dimensions are used by all fact tables. You can use Microsoft SQL Server Management Studio to explore these relationships; you can find what facts are in a fact table and which dimensions a fact table uses. Figure 3 shows an example dimension as viewed in Management Studio. You'll notice it has multiple, redundant pieces of information about a date. These extra pieces of information are used for hierarchies, which are described later in this article, and help with slicing and dicing of the data in queries.

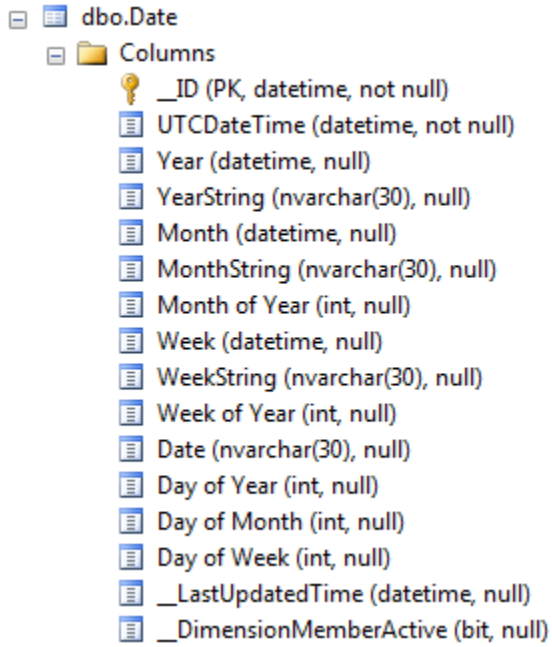


Figure 3

The Date dimension as viewed in Management Studio

Figure 4 shows an example of a fact table; in this case, Work Item History. You can see that it contains foreign keys; each foreign key represents a dimension. The other columns represent facts that are being stored in this table. Therefore, each row contains a set of facts and is connected to specific values in the different dimension tables.

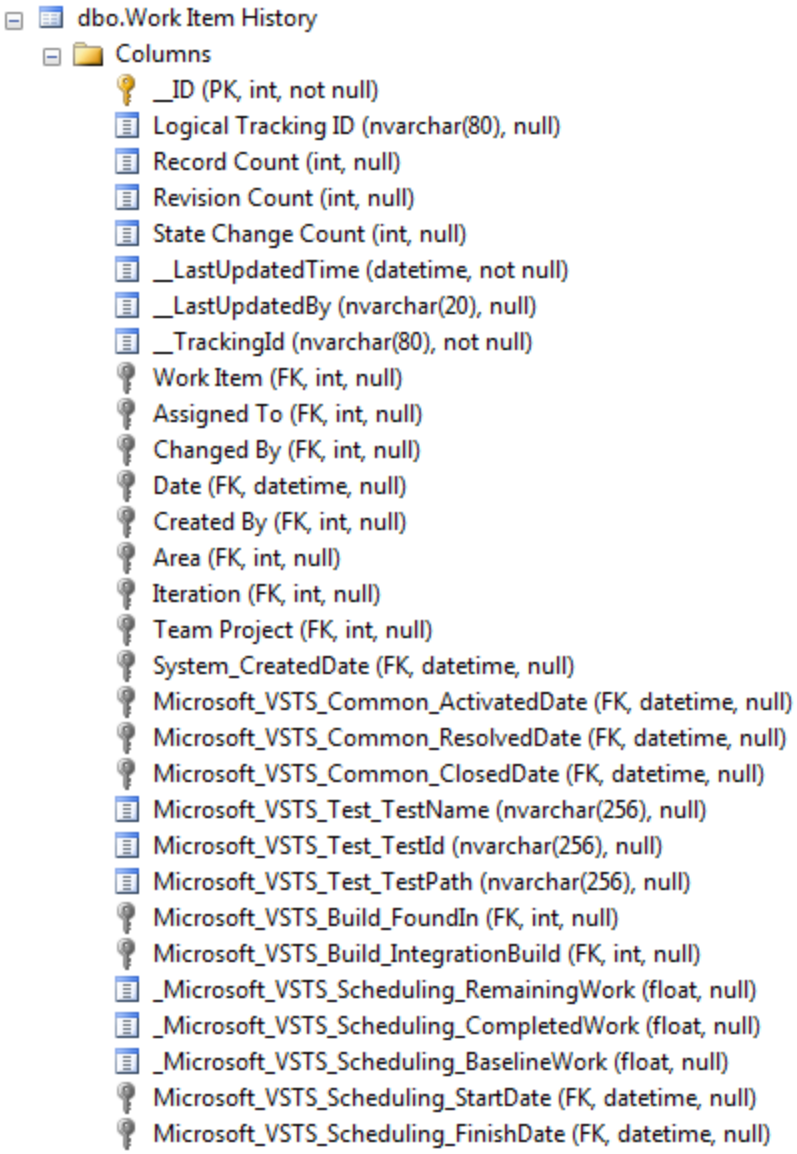


Figure 4

The Work Item History fact table as viewed in Management Studio

TFS OLAP Cube

The star schema in the relational warehouse certainly helps. However, it doesn't go far enough. Many reports aggregate values; the way values are aggregated changes depending on which dimensions you decide to use to filter and categorize the data you get back (this will make more sense when we start to work with queries in Visual Studio). Because there can be millions of rows in the facts table, aggregations can become slow very quickly. This is where a "cube" can make a huge difference both in performance and the type and form of data you can retrieve. Figure 5 summarizes the advantages of the different stores used by TFS. In this diagram, ETL refers to Extract, Translate, and Load, which is the process used to convert from the OLTP schema into the star schema of the warehouse. Also, process

refers to the extra processing that Microsoft SQL Analysis Services performs on the data in the relational warehouse, such as precalculating aggregated values.

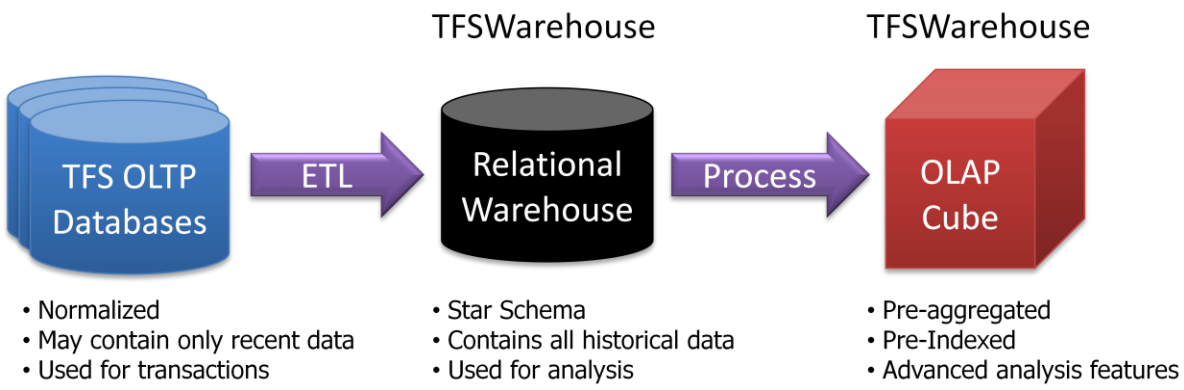


Figure 5

Characteristics of TFSWarehouse stores

Dimension Hierarchies

The advantage of a cube becomes even clearer when you add yet another ingredient to the pot—hierarchical dimensions. Let's begin with a common example. If you look at dates, you can group them in different ways, such as by year, month, or date. In fact, if you look at Figure 3 again, you'll notice this table contains a column for each of these groupings plus more. These groupings have a clear hierarchy. The year contains 12 months and each month contains a number of days, as shown in Figure 6.

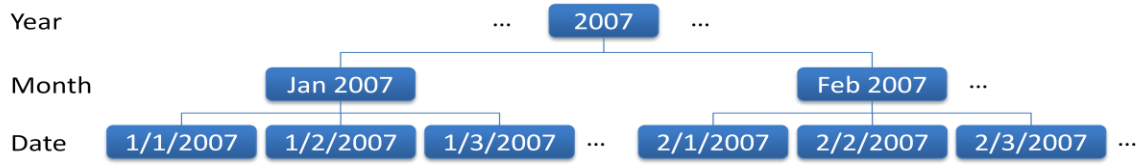


Figure 6

Example of the Year Month Date hierarchy in TFS

Hierarchies like this can be very useful when aggregating values because you can answer questions that would otherwise be very hard to answer. For example, how does the percentage of bugs opened versus bugs closed this month compare with the results from one month ago? In other words, if we're currently in the month of April, we would want to compare these values with the values from January. Amazingly enough, SQL Server Analysis Services (SSAS) precalculates a number of aggregated values at the different levels of hierarchies like this one, as long as the hierarchy is defined in the cube. It also makes it very easy to move to previous and next nodes at a specific level in this hierarchy. For example, the month before Jan 2007 is Dec 2006. It is obvious, but it's hard to write a query that does this type of moving around with standard SQL, whereas it's very easy with SSAS and its MDX query language.

Figure 7 shows an example of Date hierarchies as shown in the query builder we'll be using later. This figure shows three different hierarchies. The first hierarchy really isn't a tree because it's just a flat list of dates. The second hierarchy is organized by year, then month, and finally date, just as shown in Figure 6.

The last hierarchy is very similar, except it uses the week number in the year instead of month, so weeks go from 1 to 52.

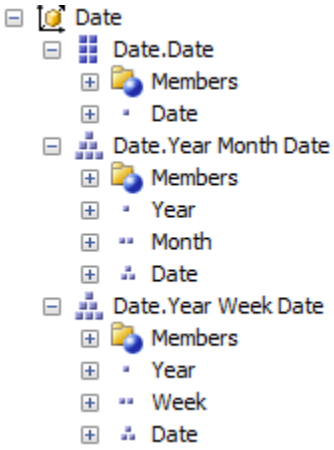


Figure 7

The cube in TFS has three different hierarchies for Date (each small dot next to the right-most nodes shown represents its level in the hierarchy; three dots [arranged in a triangle] are below the node with two dots

Getting Your Computer Set Up

That's enough theory for now. In this section, we'll get your computer set up so you can start working with the data in the relational warehouse and the cube.

Installing Required Tools

First you'll need to make sure you have all the required software installed on your computer. This article assumes you have TFS completely installed on a server that you can access, so this section is just about installing the tools you'll need to communicate with TFS's data stores and create reports. You will need the following:

- Visual Studio 2005 Professional Edition or Visual Studio 2005 Team System
- SQL Server Client Tools:
  - Management Tools (optional)
  - Business Intelligence Development Studio
  - SQL Server Books Online (optional)
- Visual Studio 2005 SP1
- Visual Studio 2005 SP1 Update for Windows Vista (if you are using the Windows Vista operation system)
- SQL Server 2005 SP2

The Business Intelligence Development Studio (which is part of the SQL Server Client Tools) will install the tools you need inside Visual Studio to create and customize TFS reports.

> **Note:** If you're running Visual Studio 2005 on a computer that is running the Microsoft Windows Vista operating system, you will need to launch Visual Studio under administrator permissions to be able to work with reports.

Creating a Report Server Project

After you install all the tools, you'll see a new set of project types appear under Business Intelligence Projects in Visual Studio's **New Project** dialog box, as shown in Figure 8. After you name your project and choose a location, click **Report Server Project** under **Templates**, and then click **OK**.

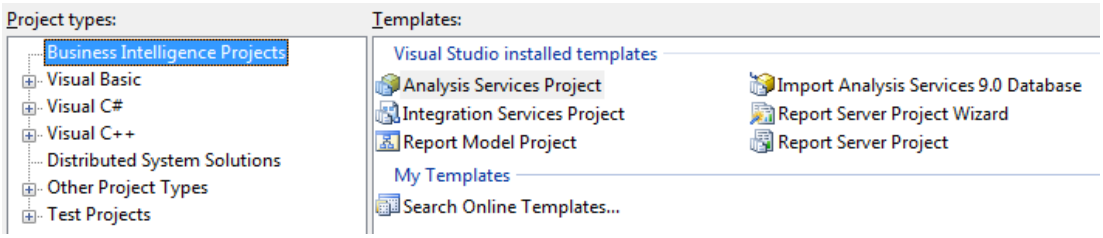


Figure 8

The Business Intelligence Projects project type appears in Visual Studio after you install all the correct tools

Your new project should contain only two empty folders, as shown in Figure 9.

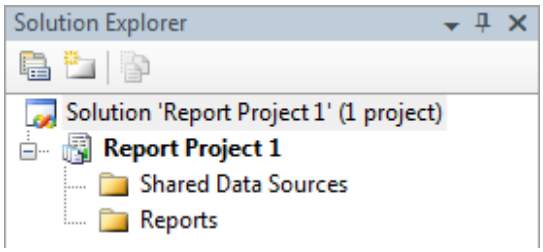


Figure 9

A new empty report project should look like this

Creating the Data Sources

The next step is to add two data sources: one that connects to the relational warehouse and one that connects to the cube. To add the relational data source, do the following:

1. Right-click the Shared Data Sources folder, and then click **Add New Data Source**.
2. On the **General** tab, type **TfsReportDS** in the **Name** text box. Many of the reports in TFS expect this data source name, so the name is actually important when you're writing reports for TFS.
3. In the **Type** combo box, click **Microsoft SQL Server**.

4. Create the connection string for connecting to the SQL Server instance that is hosting the data warehouse. It's easiest to click the **Edit** button and enter appropriate information in the fields. You'll want to select the **TFSWarehouse** database. You will need to make sure your administrator has given you access rights to the database.
5. Click **OK**.

Next, you'll need to create a data source that connects to the cube. To do this, do the following:

1. Right-click the Shared Data Sources folder, and then click **Add New Data Source**.
2. On the **General** tab, type **TfsOlapReportDS** in the **Name** text box. Many of the reports in TFS expect this data source name, so the name is actually important when you're writing reports for TFS.
3. In the Type combo box, click Microsoft SQL Server Analysis Services.
4. Create the connection string for connecting to the SQL Server instance that is hosting the data warehouse. It's easiest to click the **Edit** button and enter the appropriate information in the fields. You'll want to select the **TFSWarehouse** database. You will need to make sure your administrator has given you access rights to the database.
5. Click **OK**.

Your project should now have the two data sources, as shown in Figure 10.

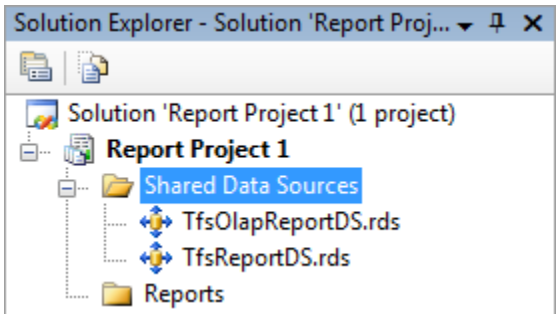


Figure 10

After you set up your two data sources, you'll see something like this in Solution Explorer

Adding a Report

If you add a report the usual way, a wizard that walks you through creating a report appears. This article is bypassing this wizard so we can jump directly into working with queries. To add a report, do the following:

1. Right-click the Reports folder, point to **Add**, and then click **New Item**.
2. In the **Add New Item** dialog box, click **Report**, give it any name you want, and then click **Add**. The example in this article uses the name "Test Report."

You should now see your report in Solution Explorer under the Reports folder. When you open the report (it usually automatically opens when you add a new report), you'll see a dialog box with three tabs: **Data**, **Layout**, and **Preview**.

Building a Simple Query

We'll be working with the **Data** tab of the report in this section. To create a new dataset attached to the cube, do the following:

1. In the **Dataset** combo box, click **New Dataset**. This opens the **Dataset** dialog box.
2. In the **Name** text box, type a name. The example for this article uses the name dsTest.
3. In the **Data source** combo box, click **TfsOlapReportDS (shared)**. This connects this query to the cube instead of the relational warehouse.
4. Click **OK** to create this dataset.

At this point, you'll see a query window that is very different than other query windows you may have seen before, as shown in Figure 11.

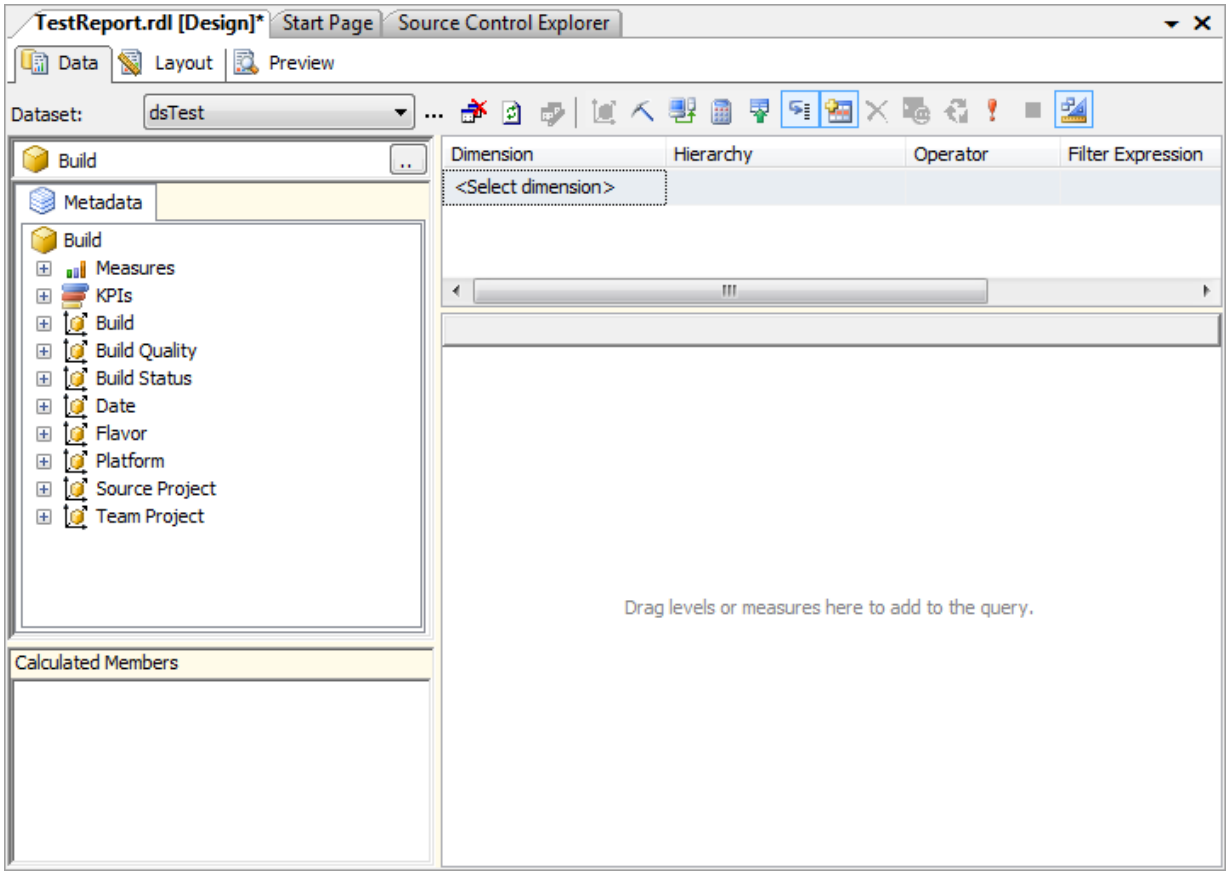


Figure 11

This editor appears after you create a new dataset that is connected to the cube; the list on the left will be longer if you're not using the Enterprise version of SQL Server

This window consists of four main areas: **Metadata**, **Calculated Members**, query results, and the dimensions/filters. We'll ignore the **Calculated Members** area for now, but as you'll see later, it's very useful for doing calculations on the returned data.

Before we create a query, the **Metadata** area you see in Figure 11 shows a combination of measures, KPIs (key performance indicators, which we'll ignore in this article), and dimensions. Right now, the list of dimensions is restricted to a subset named Build. The Enterprise edition of SQL Server supports "perspectives" that allow you to restrict your view of dimensions to a subset that are relevant to the type of query you want to build. The standard edition supports only one perspective named Team System. If you see **Build** above the **Metadata** tab, click the ellipsis button (...) to the right of **Build**, and then click **Team System** in the **Cube Selection** dialog box.

Now you should see a very long list of dimensions, as shown in Figure 12.

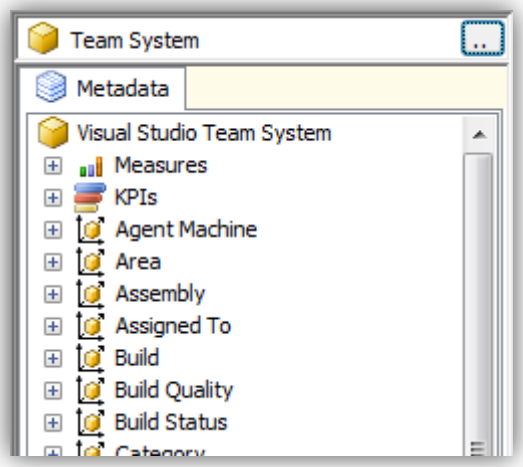


Figure 12

This is what you see when you view the entire cube instead of a perspective

Currently, the query results window doesn't shows anything. We can change that by dragging a measure from the **Metadata** area into the query results area. To do this, do the following:

1. Expand the **Measures** branch of the tree.
2. Open the Current Work Item folder.
3. Drag the **Current Work Item Count** measure into the query results area, as shown in Figure 13.

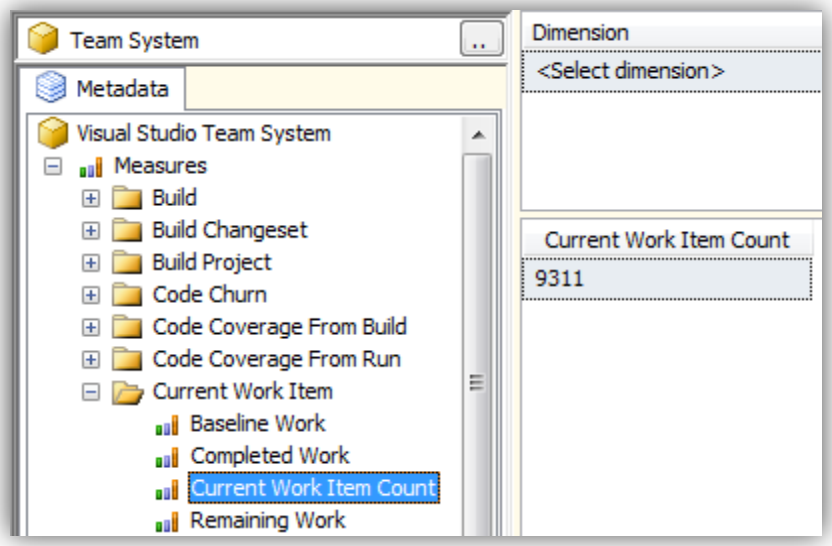


Figure 13

Dragging a measure into the results area will show an aggregate value

You should now see something like Figure 13, but the value you see under **Current Work Item Count** will almost certainly be different. This value is the total number of work items in the cube. And you may have noticed that this number appeared almost instantly, which is a result of the cube precalculating a number of aggregated values.

Adding Some Dimensions

Having the total number of work items is fun, but it is not very useful. Let's say you want to see how many work items are in a specific project. To do this, do the following:

1. Drag the **Team Project** dimension into dimension area above the query results area.
2. Click the **Filter Expression** cell, and then click the down arrow in the box that looks like a combo box.
3. Select a project that you want to view, as shown in Figure 14.

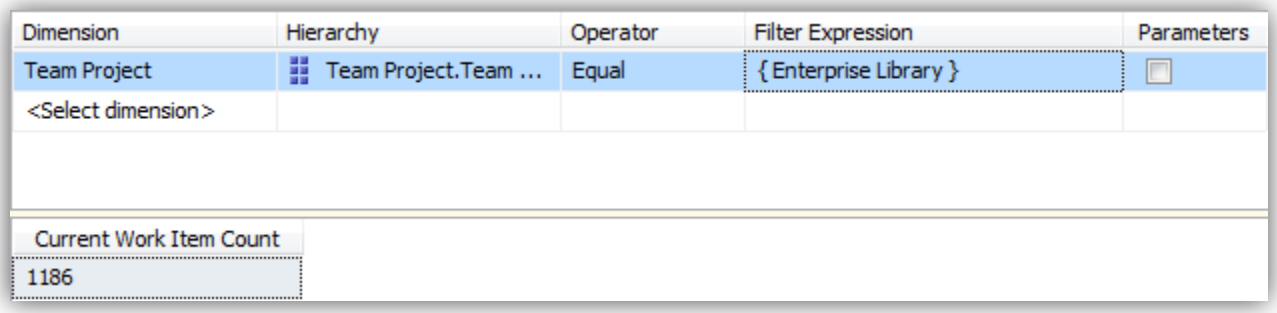


Figure 14

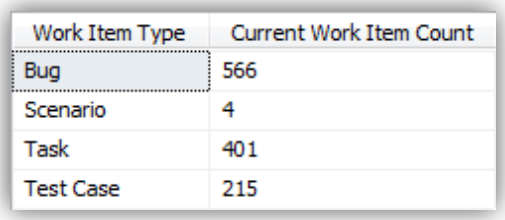
Results of filtering for only the Enterprise Library project in the patterns & practices group at Microsoft

You'll notice that the number is now lower (assuming you have more than one project on your TFS server) because it now shows the number of work items just in that one project.

Showing Multiple Rows

So far, we've seen only a single number returned from the query. This isn't very useful. After all, it's hard to graph a single number and have it look very interesting. The next step is to add a dimension attribute as a new column in the results area. To do this, do the following:

1. In the **Metadata** area, scroll down to the **Work Item** dimension, and then expand it.
2. Drag the **Work Item.Work Item Type** attribute into the results area. Before you release the mouse button, notice that there is a red line that shows where the column will be placed in the results. In this case, it will only place the column to the left of the current column.



Work Item Type	Current Work Item Count
Bug	566
Scenario	4
Task	401
Test Case	215

Figure 15

The results now show how many work items of each type are in the project

The result will be a list of work item types used in your project, along with the number of each type. The example results look like Figure 15, but your results will certainly look different.

Building a Bug Rate Report

With these fundamentals, we'll create a real report. We'll use the updated Bug Rates report I created for the MSF group at Microsoft as an example. However, we won't create the full report—just enough so you'll know how to add all the details or understand what you find in the existing report. Figure 16 shows what the full report looks like. As you can see, there are six different parameters that control what you see in the report. For the report we'll build here, we'll only add one parameter so you can see how it's done.

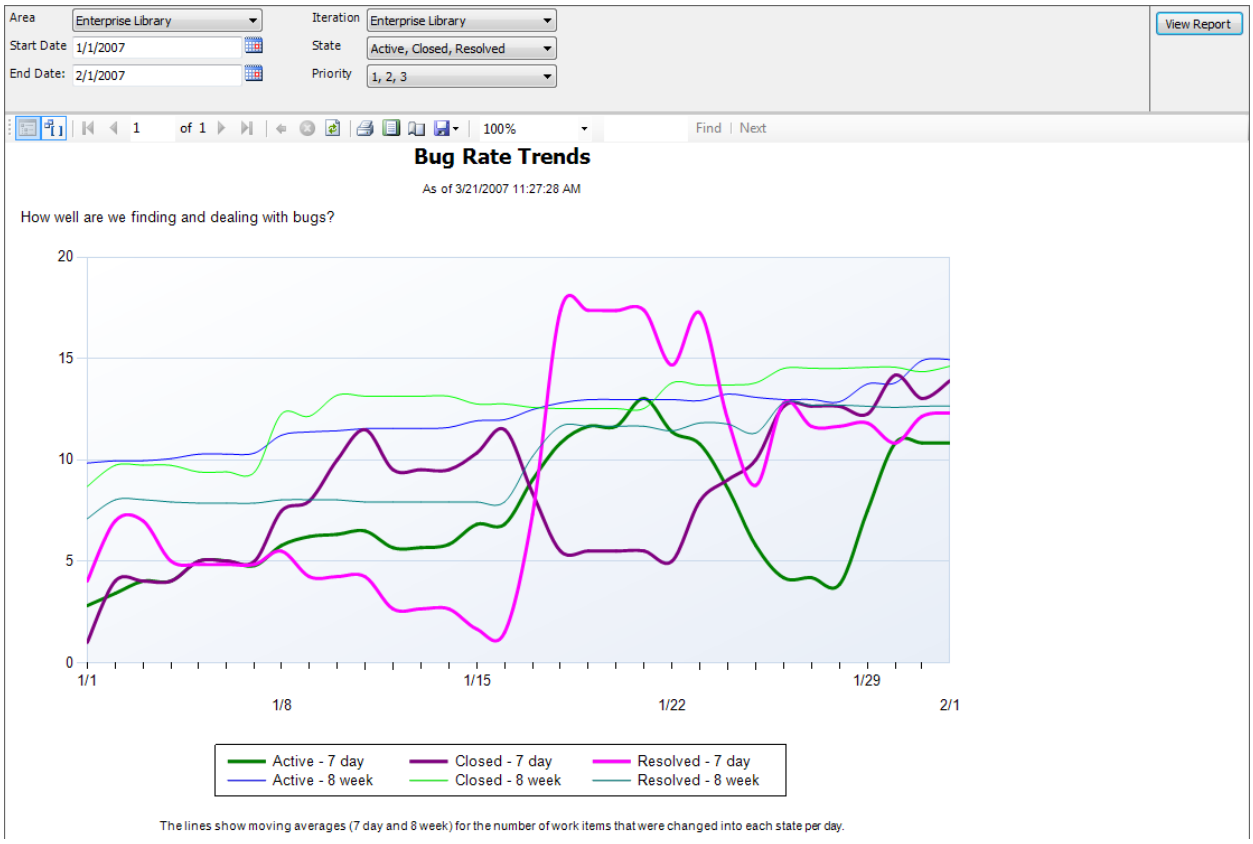


Figure 16

The newer Bug Rates report shipped by MSF looks like this when run inside Visual Studio; there are six parameters that drive what you see

Let's think about what type of data we'll need to create this chart. We're graphing historical data; in this case, we're graphing data over a period of a month. The data is the number of bugs in each state on each day of the report. To make things more interesting, this graph is also using a rolling average to smooth the lines. The bold lines use a 7-day moving average, and the thin lines use an 8-week moving average.

Initially, we'll write a query that returns the raw data without smoothing. For this query, we'll need to think about which measures and dimensions we're going to need. We want results returned for each day of the date range we'll be using, so we'll want to include the Date.Date dimension. Recall from Figure 7 that there are several dimension hierarchies for Date, but we don't need anything other than just the date.

We're really going to want to start over with a new query. You can delete the existing query if you want (to do this, click the **Delete Selected Dataset** toolbar button [it's the button with the red X on it]). To create a new query, do the following:

1. In any event, create a new dataset named **dsBugRates** using the **TfsOlapReportDS** dataset.
2. Make sure you're viewing the **Team System** perspective instead of **Build**.
3. In the **Metadata** tree, expand the **Measures** branch.

4. Open the Work Item History folder (because we want historical data instead of current data).
5. Drag the **Cumulative Count** measure into the query results area.
6. Drag the **Team Project** dimension into the dimension/filter area, and then select the project you want to view.
7. Finally, drag the **Date.Date** dimension into the query results area. To find this dimension, expand the **Date** dimension in the **Metadata** area.

At this point, you should see a number of rows in the result area, with one row for each date. Assuming you're using a project with some history, this result set could be quite long. Before adding more dimensions, which will cause the result set to grow even larger, it's a good idea to add some filters. To do this, do the following:

1. Drag the **Date.Date** dimension into the dimension/filter area, as shown in Figure 17.
2. Click in the **Operator** column for **Date.Date**, and then click **Range (Inclusive)** in the combo box.
3. Click in the **Filter Expression** column for **Date.Date**, and then click a start date in the left combo box and an end date in the right combo box. The example in Figure 17 shows a date range for the month of January 2007.

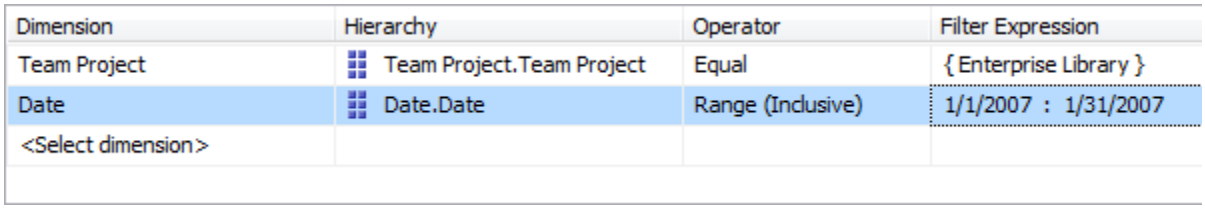


Figure 17

Filtering based on a date range

You should now see only a month of work items. But at this point, we're looking at all work items; yet for a report named Bug Rates, we should really be looking at only the Bug work item type. You can add another filter to do just this. To do this, do the following:

1. In the **Metadata** area, expand the **Work Item** dimension.
2. Drag the **Work Item.Work Item Type** dimension into the filter area.
3. Click in the **Filter Expression** cell and select the check box for the Bug work item type (or whatever name you're using in your project). You can select check boxes for more than one work item type if you have more than one you use for bugs.

You'll notice that the query results window will update after you've added this filter and the numbers will most likely be different. In fact, they should be lower, assuming you have other types of work items in your project.

Thinking About What to Retrieve

So far, the results have shown the total number of bugs that were in your project on each day shown. Because TFS doesn't have a way to delete work items, this number should increase over time, so plotting this information really isn't very useful.

At this point, it's good to stand back and think about what you're trying to accomplish with the report. When you're looking at bug reports, you might want to look at how the totals change over time, which is the query we have so far. Or, you may want to look at the rate of change over time. The latter means you want to see how many new bugs are added, resolved, closed, etc., each day of the report. Because this is referred to as a Bug Rates report, the word "rates" implies we're interested in looking at the trend; In other words, we want to view the number of bugs that changed into each state instead of the total number of bugs in each state on a particular day.

Fortunately, the cube has a measure named State Change Count that we can use to get this information. To do this, do the following:

1. Click and drag the **Cumulative Count** header out of the result area. This removes this column from the query. You'll notice the results area is now be blank because we're not asking for any measure.
2. In the **Measures** branch of the **Metadata** area, open the Work Item History folder, and drag the **State Change Count** measure into the result area.

At this point, you should see a very different set of numbers. Chances are you'll see the numbers going lower and higher throughout the month. You'll also notice that some dates will be missing, which is the case when no work items changed state in your project on that date.

Adding the State Dimension

Now let's break down the state changes for each day into the different states. The Bug work item can have many states. Active, Resolved, and Closed states are supported out-of-the-box, but your work item might have more, or different, states. These states are defined in the Work Item, so we'll drag the state dimension and add it as a column in the results. To do this, do the following:

1. In the **Metadata** area, expand the **Work Item** dimension.
2. Drag the **Work Item.State** dimension into the result area. You'll notice the red vertical line will allow you to drop it either to the left or right of the date. Drop it on the right side so we'll see the results grouped by date first and state second, as shown in Figure 18.

Date	State	State Change Count
1/1/2007	Active	3
1/2/2007	Active	6
1/2/2007	Closed	3
1/2/2007	Resolved	5
1/4/2007	Active	5
1/4/2007	Resolved	1
1/5/2007	Active	9
1/5/2007	Closed	2
1/5/2007	Resolved	6
1/7/2007	Active	8
1/8/2007	Active	2

Figure 18

Part of the results returned after the query with Date and State dimensions runs

That's all the data you need at the moment to be able to create a graphical report.

Adding a Graph

All of your work so far has been on the **Data** tab of the Report Designer window of Visual Studio. There are two other tabs in this window that allow you to edit and preview reports. In this section, you'll add a graph to the report and set it up so it shows the results from your query. To do this, do the following:

1. Make sure the Toolbox panel in Visual Studio is visible.
2. In the Toolbox panel, click the **Chart** item.
3. Click and drag inside the **Layout** area to create a new chart.
4. Drag the bottom right adorner (corner) on the graph to make it whatever size you want.

At this point, you'll see something like Figure 19. When the chart is selected, as it is in the figure, you'll see three drop zones for data, series, and category fields. These will be explained later in this article.

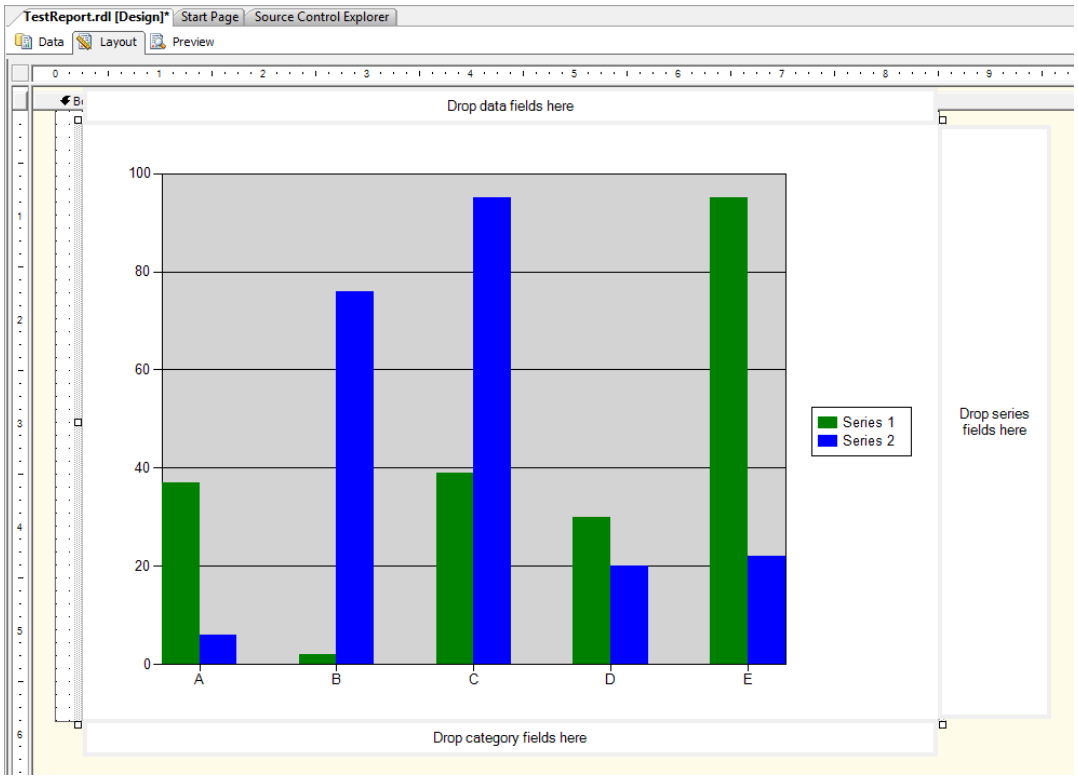


Figure 19

An empty chart, when selected, has several drop zones, which you'll use to add data you want graphed

Before you add any data, change the type of chart so it shows lines instead of bar graphs. To do this, do the following:

- Right-click the chart, point to **Chart Type**, point to **Line**, and then click **Simple Line**.

Now it's time to give the chart some data to plot. To do this, do the following:

1. Make sure you have the Datasets panel open in Visual Studio (to do this, click **Datasets** on the **View** menu).
2. Under the **Report Datasets** branch, expand the **dsBugRates** dataset.
3. On the **Layout** tab, double-click the chart. This will display the three data drop zones.
4. Drag the State_Change_Count column from the **Datasets** panel into the **Drop data fields here** drop zone of the chart.
5. Drag the **State** column into the **Drop series fields here** drop zone.
6. Drag the **Date** column into the **Drop category fields here** drop zone.
7. At the top of the Report Designer window, click the **Preview** tab.

At this point, you should see an actual report running in Visual Studio, using data from your TFS server, and it should look something like Figure 20.

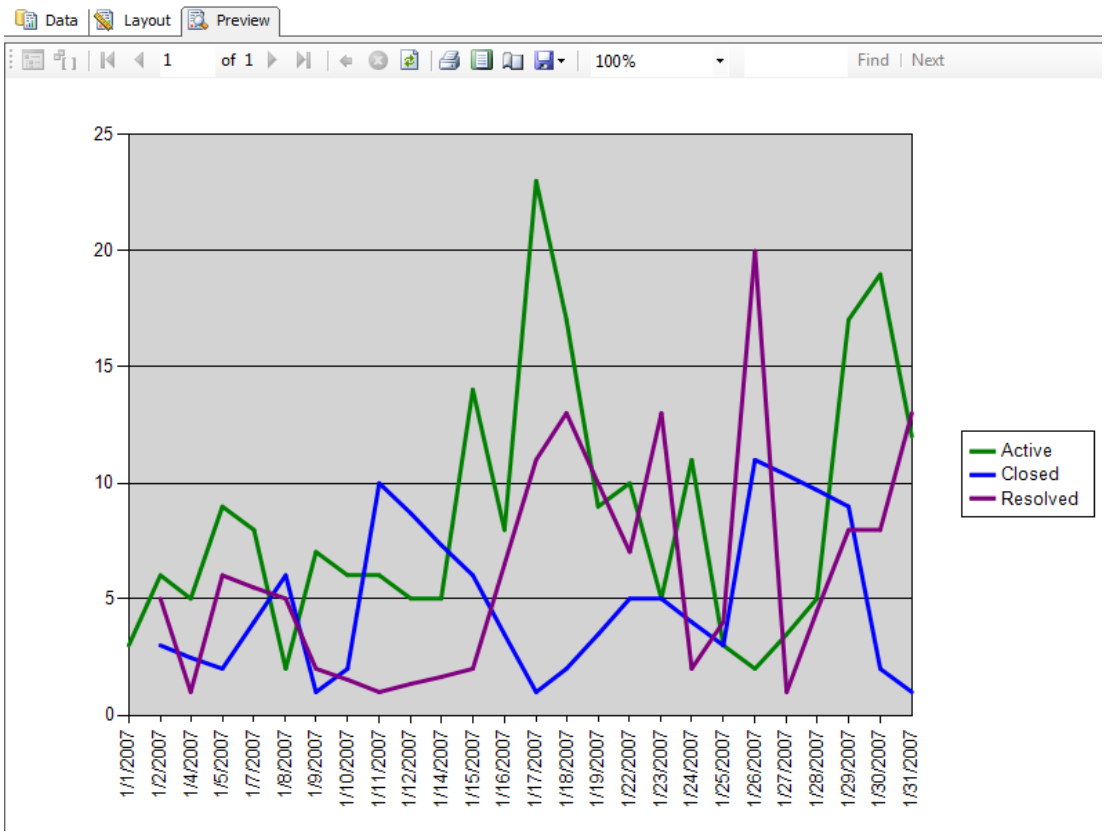


Figure 20

The report showing state change count by date (but missing some dates)

Look at this graph carefully. Do you notice anything wrong? Do you see anything missing? Look at the dates at the bottom. There are dates missing from this graph. Why?

The query we created only shows rows where the measure results are not null. Some of the dates had no work items change states. For example, you'll notice that Saturdays are missing (Sundays appear because we have testers in India, in a very different time zone). How do you get all days to show up? You can add another column, such as Cumulative Count, that won't be null. To do this, do the following:

1. In the **Metadata** area, expand the **Measures** node.
2. Expand the Work Item History folder.
3. Drag the **Cumulative Count** measure into the result area (the order of this column doesn't matter, but it's a little easier to add it to the very right).

Now your results will include rows for days when all state change values are null. If you click the Preview tab again, the report should update to show all days of the range you selected.

Adding Calculated Members

This article mentioned earlier that we would eventually switch over to using a rolling average to smooth out the lines. We can do this by creating a new column in the results that is calculated using values in the cube.

At the bottom of the **Data** tab is an area labeled **Calculated Members**. This is where we'll create a definition for our rolling average. The syntax will look a little odd because it's actually using a snippet of a query language specifically designed to work with cubes. This query language is named MDX, which stands for Multi-Dimensional eXpressions, and is designed specifically to work with cubes, using knowledge of dimensions and measures. To create a definition for our rolling average, do the following:

1. Right-click in the **Calculated Members** area, and then click **New Calculated Member**. This opens the **Calculated Member Builder** dialog box.
2. In the **Name** text box, type **Rolling Average**.
3. Enter the following in the **Expression** text box:

   Avg([Date].[Date].CurrentMember.Lag(6): [Date].[Date].CurrentMember, [Measures].[State Change Count])
4. Click the **Check** button to make sure the expression is valid.
5. Click **OK** to finish creating this calculate member.
6. Drag the **Rolling Average** member into the result area.
7. Switch to the **Layout** tab, and then drag the Rolling_Average column from the **dsBugRates** dataset into the top drop area of the graph.
8. Right-click the **State Change Count** in the top drop area, and then click **Delete** to remove the old column from the graph.

The preceding expression does a 7-day rolling average of the values, and I'll explain the syntax a little later. At this point, you should see a set of numbers that changes a little more smoothly than the raw State Change Count numbers. Likewise, the graph should be a little smoother now. Figure 21 shows the result of running this on our server.

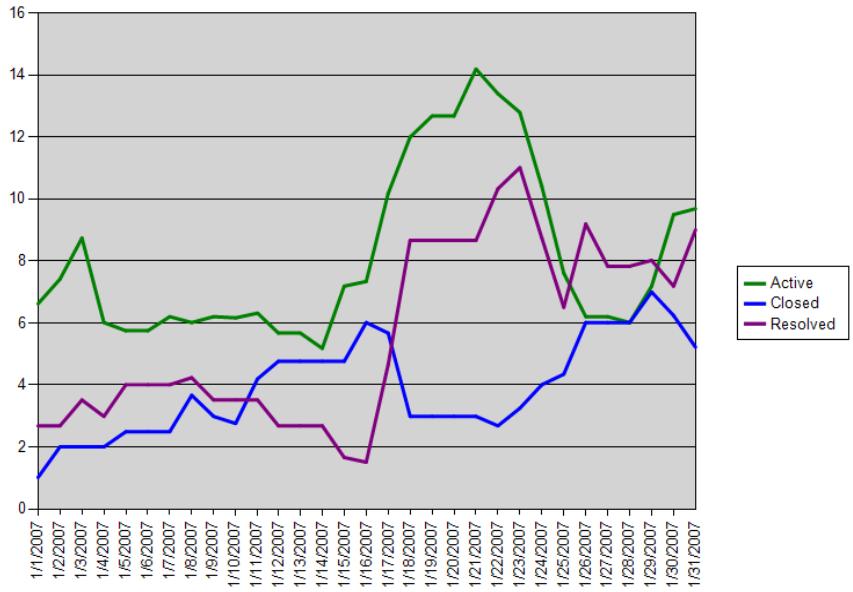


Figure 21

Graph with a 7-day rolling average; notice that the lines are a little smoother

You can make the lines even smoother by changing the chart type. Right now, it's a Simple Line graph, but if you change it to a Smooth Line graph, you won't see any kinks in the graph. To do this, do the following:

- On the **Layout** tab, right-click the chart, point to **Chart Type**, point to **Line**, and then click **Smooth Line**.

A Snippet of MDX

Let's take a look at the expression that calculated the rolling average:

```
Avg(
    [Date].[Date].CurrentMember.Lag(6): [Date].[Date].CurrentMember,
    [Measures].[State Change Count]
)
```

This expression is using a small piece of MDX. What I want to emphasize here is that you can go a long way with just a small amount of MDX knowledge. Let's take a look at the different parts of this expression.

First, **Avg** is a function (and there are many more) that takes two parameters. The first parameter is actually a "set" of dimension values, where a set can contain zero or more items. And the second parameter is a fact we want used in the expression.

This expression uses values from the Date.Date dimension (see Figure 7), and you're probably wondering why there are square brackets around each name in this dimension. The square brackets in this specific case are optional. However, names are allowed to have spaces in them, such as for **State**

Change Count, and in these cases, the square brackets make it possible for the parser that processes the expression to know where a name starts and ends.

All queries you build using the cube generate MDX behind the scenes, so the MDX expression here is combined with the rest of the MDX for the query. As Analysis Services processes an MDX query, it performs calculations for each row returned by the query. **CurrentMember** in the preceding expression refers to the current "instance" of a dimension for the row being calculated. **CurrentMember** is attached to the Date.Date dimension, so **CurrentMember** is the date, such as 1/1/2007, in the row being processed.

After the **CurrentMember** is another function, **Lag**, which moves between values at the same level in the tree. **Lag** will move to the value that is six before the current value. For example, if **CurrentMember** is 1/10/2007, **Lag(6)** will return 1/4/2007, which is six days earlier. Finally, the colon says to use a range. In other words, it indicates the following expression:

[Date].[Date].CurrentMember.Lag(6):[Date].[Date].CurrentMember

The preceding expression returns a set that contains a week worth of dates, ending with the date from the current row being processed.

In other words, the **Avg** expression in the earlier expression averages the values from the set of seven days, ending in the date from the current row. But what is it averaging? After all, there are different facts it could be averaging. The answer is in the second parameter, which tells the expression which fact, or "measure," to use in calculating the average. The expression uses [**Measures].[State Change Count]**, which means that it will return the average of **State Change Count** over a one-week period, ending in the date of the current row.

Finding the Names of Measures and Dimensions

You may have also noticed that the name **[Measures].[State Change Count]** doesn't directly match the hierarchy shown in the **Measures** branch of the tree in Figure 13. So how do you get the correct name? Figure 22 shows how you can use the Metadata tree to get the full name for a measure.

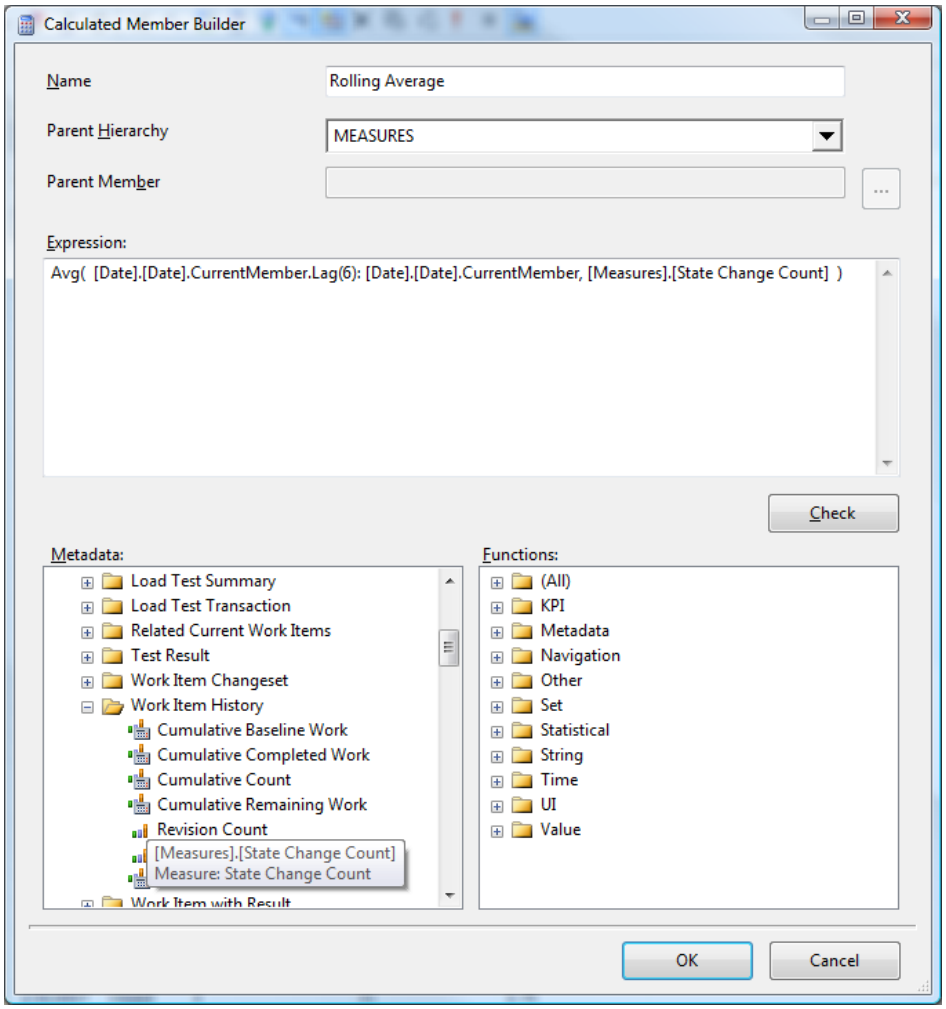


Figure 22

Hovering over a member in the Metadata area displays the MDX name and a short description

If you were creating this expression from scratch instead of using the provided expression, it's often easier to use the two trees at the bottom of the **Calculated Member Builder** dialog box. You can find a name by hovering over an item, as shown for **State Change Count** in Figure 22. You can also double-click any node of the tree to add the text to the expression text box. The left tree provides access to all the measures and dimensions, while the right tree provides access to various functions and properties.

Adding Parameters

This report is all very nice, except it's not very general. In other words, you have to modify the report in Query Designer to change the date range. Obviously, you don't want to have every user of your report load it into Visual Studio's Report Designer just to change the date range. What you really want is to allow users of the report to simply select the range of dates.

You can use parameters to perform this type of operation. You'll be adding two parameters to this report: start dates and end dates. These parameters will control the date range shown in the report.

Additionally, you probably don't want the project hard-coded into the report, so you can use the same report in any project.

Making Start and End Dates Parameters

At this point, the report query contains three filters that you'll probably want to convert into parameters, and the process takes a few steps. The first step is to mark which expression values you want to make into parameters. In the Query Design window, select the two check boxes in the row that contains the Date dimension, as shown in Figure 23.

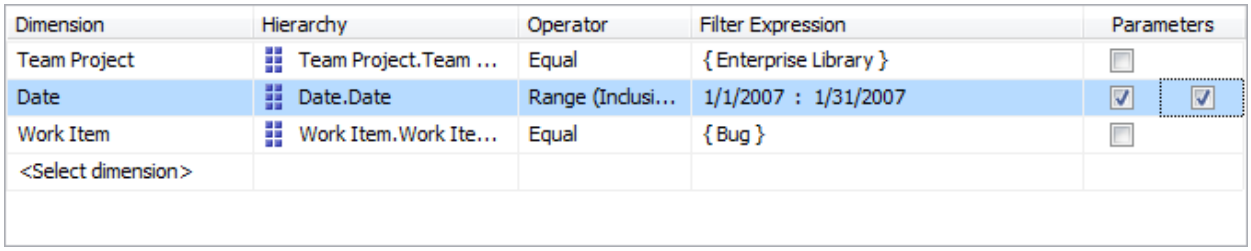


Figure 23

Select the two check boxes in the Parameters column for the Date dimension to make them parameters

There are two check boxes in this row because the filter is over a range of dates. Therefore, there is a check box for the two extremes of the filter.

Selecting these two check boxes doesn't do anything immediately, so you need to click the Layout tab and then click the **Data** tab again. After this, you'll see two new datasets in the **Dataset** combo box: **FromDateDate** and **ToDateDate**. In case you're wondering about these strange names, the **From** and **To** come from the **Range** operator, while the **DateDate** comes from the Hierarchy column. As you'll recall, Date.Date means that we're using the Date dimension, and we're looking at the Date slicing of that dimension, instead of another slicing like [Year Month Day].

If you select one of these datasets, you'll see a bunch of really ugly code in place of the filter area we've been working with. This ugly code is raw MDX. Fortunately, you're not going to need these two raw MDX datasets. In addition to creating these two datasets, selecting the two check boxes in the Parameters column also created two report parameters, which is where you'll do the actual work.

Before we go any further, you can click the **Preview** tab to run the report. You'll see there are two parameters that allow you to set the start and end dates, as shown in Figure 24. However, there are a couple of problems with these parameters. First, they appear as combo boxes that have a very long list of dates instead of appearing as a calendar control. And second, the labels show the internal names and aren't very user friendly.

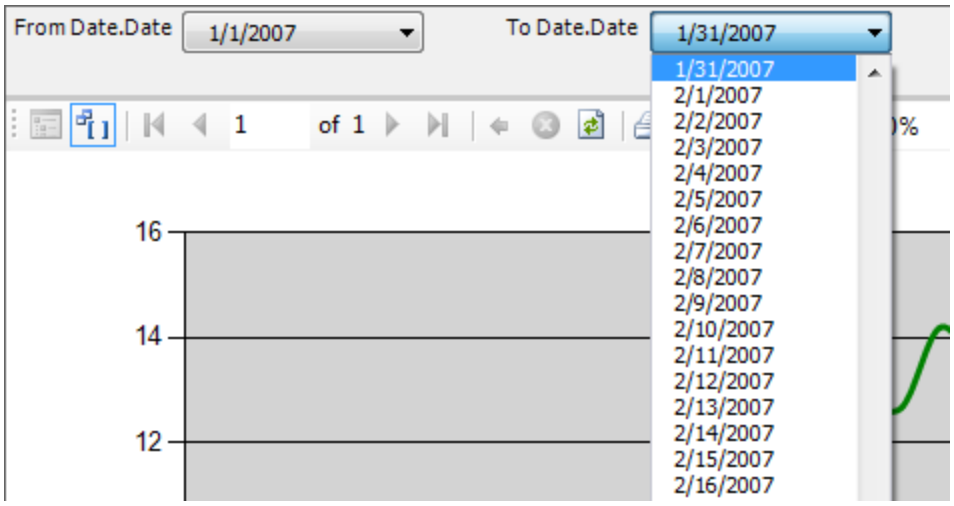


Figure 24

The two date parameters appear as very long list boxes

The date parameters appear as a long list instead of a calendar control because they're defined as a String instead of a **DateTime** type in the reports parameter dialog box. To see this, click **Report Parameters** on the **Report** menu.

Unfortunately, you can't just change the parameter type to **DateTime** and have it work. Why not? Well, this gets a little tricky. The Query Builder interface builds an MDX query behind the scenes that assumes any values coming from parameters will be strings and not any other type. Additionally, the strings often have to be in a very specific format, as you'll see soon. The solution is to create a new set of parameters that will be the visible set, with the data type of **DateTime**. The **FromDateDate** and **ToDateDate** parameters will then be hidden and receive their values from the two visible parameters, using some expressions to format the strings correctly.

Start by adding two new parameters and hiding the two existing parameters. To do this, do the following:

1. On the **Report** menu, click **Report Parameters** (this command is unavailable when the **Preview** tab is active).
2. Under the **Parameters** list, click **Add**.
3. In the **Name** box, enter **FromParameter**.
4. In the **Data type** box, enter **DateTime**.
5. In the **Prompt** box, enter **Start Date**.
6. In the **Default values** area, click the **Non-queried** option button, and then enter the following expression into the text box to the right of the button:
   =DateAdd("m",-1,Today())
7. Click the up arrow to the right of the **Parameters** list box so this new parameter is above the **FromDateDate** and **ToDateDate** parameters in the list.

8. Repeat these steps to add a second parameter with **Name** set to **ToParameter**, the **Prompt** of **End Date**, and the following expression:
   =Today()

You can see what this looks like by clicking the **Preview** tab. You'll see two sets of parameters. The new set of parameters will show a nice pop-up calendar you can use to select start and end dates. However, at this point they're not connected to the query. By the way, you'll soon see why these two new parameters needed to be added above the two parameters created by the Query Builder (it's to control calculation order, which is top to bottom).

The next step is to modify the two auto-generated parameters so they convert the values from the two new parameters into correctly-formatted strings. To do this, do the following:

1. Open the **Report Parameters** dialog box (from the **Report** menu).
2. Click the **FromDateDate** parameter.
3. In the **Available values** section, click the **Non-queried** option button.
4. In the **Default values** section, enter the following text into the expression text box:
   ="[Date].[Date].&[" + CDate(Parameters!FromParameter.Value).ToString("s") + "]"
5. Repeat for **ToDateDate**, except using the following expression:
   ="[Date].[Date].&[" + CDate(Parameters!ToParameter.Value).ToString("s") + "]"
6. Click the **Preview** tab.

The values in the **FromDateDate** and **ToDateDate** parameters now have values that look something like the following:

[Date].[Date].&[2007-03-16T00:00:00]

Why this strange value? This value is formatted in MDX. The first **Date** is the name of the dimension. The second **Date** indicates to use the date slicing instead of one of the other slice options available (such as [Year Month Day]). The ampersand character says this is an actual value, and the value at the very end is the date formatted in a non-locale specific format. Whew!

You can learn what format to use for strings like this with the help of the Metadata browser in the Query Builder window. Navigate down the dimension tree you want, and then drill down into the Members/All branch of the tree. When you hover the mouse over one of the values, a tooltip appears that shows an example of what the strings need to look like for that dimension.

Now that this is all working, you can hide the two auto-generated parameters and delete the two ugly MDX queries. To do this, do the following:

1. Open the **Report Parameters** dialog box.
2. Click the **FromDateDate** parameter, and then select the **Internal** check box.
3. Click the **ToDateDate** parameter, and then select the **Internal** check box.

4. Close the dialog box.
5. In the **Dataset** combo box, click the **FromDateDate** dataset.
6. Click the **Delete Selected Dataset** toolbar button (it's the button with the red X on it).
7. Repeat to delete the **ToDateDate** dataset.

To summarize, you start creating custom parameters by selecting the check box(es) in the Parameters column of the filter area in the Query Designer window. This creates both one or more datasets and one or more report parameters. If you want to delete the auto-generated MDX, you'll need to create a new set of parameters and hide the old set. Additionally, you'll modify the old set to format the values from the new set into the correct MDX format as a string.

Using the "Default" Project

The query you created is specific to a single project, which doesn't make it a very general report. Ideally, this report should show results for the current project. When you deploy a report to TFS, you'll be deploying it to a single project on the server, so the current project is the project that contains the report you've deployed. How can you get this information? You can get it by using expressions and a value from the **Globals** collection named **ReportFolder**.

The solution shown here isn't ideal, but it works. One issue is that we need a project parameter that you'll want to be visible when you're developing, but you'll want to hide it before you deploy the report. You need this because the **ReportFolder** value is empty when you run the report inside Visual Studio—it only has a value when the report is running on SQL Server Reporting Services. To make your report automatically use the project that contains your report, do the following:

1. In the **dsBugRates** dataset, select the check box in the Parameters column for the Team Project dimension.
2. Switch to the **Layout** tab to generate the parameter and dataset.
3. Open the **Report Parameters** dialog box.
4. Click the **Add** button, and then create a parameter named **ExplicitProject** (you can leave all the other settings as they are).
5. Move this parameter to the top of the **Parameters** list.
6. Move the **TeamProjectTeamProject** parameter to the second place in the **Parameters** list.
7. Select the **TeamProjectTeamProject** parameter and make the following changes:
    - Select the **Internal** check box.
    - Clear the **Multi-value** check box.
    - Select the **Allow null value** and **Allow blank value** check boxes.

- In both the **Available values** section and the **Default values** section, select the **Non-queried** option button.
- Place the following expression into the Default values expression text box:

```
="[Team Project].[Team Project].[" +
IIF(LEN(Globals!ReportFolder) > 0,
   SPLIT(Globals!ReportFolder,"/").GetValue(
      IIF(split(Globals!ReportFolder,"/").Length > 1, 1, 0)),
   Parameters!ExplicitProject.Value
) + "]"
```

8. On the **Data** tab, click the **TeamProjectTeamProject** dataset in the **Dataset** combo box, and then delete it by clicking **Delete Selected Dataset** toolbar button.

At this point, if you switch to the **Preview** tab, you'll see an error, which is actually fine in this case. Figure 25 shows what you'll see.

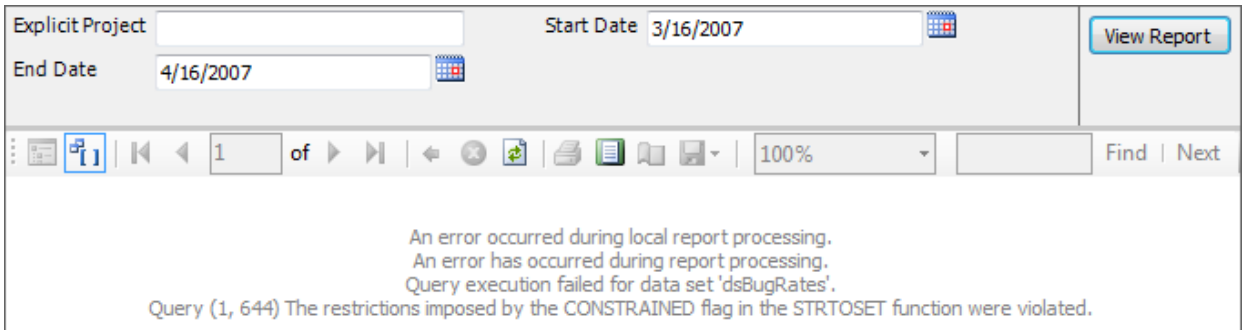


Figure 25

Error message that appears if you switch to the Preview tab

In the **Explicit Project** parameter text box, type your project name, and then click **View Report** to see the report for that project.

When you're ready to deploy the project, you'll want to set the **ExplicitProject** parameter to Internal before you place it on the server.

Publishing Reports

Speaking of deploying the project, now is probably a good time to look at how you deploy a report to the TFS Reporting Services server. It's actually very easy, but it's not necessarily obvious. Visual Studio has the ability to deploy a report directly to a project of your choice, after you set it up. To do this, do the following:

1. In Solution Explorer, right-click your report project, and then click **Properties**. This displays the Properties window.
2. Make sure **OverwriteDataSources** is set to **False**. This will preserve the data sources that were created for your project on the reporting server.

3. For **TargetDataSourceFolder**, enter the name of your TFS project, because it is the name of the folder that TFS creates on the report server for your project's reports.
4. For **TargetReportFolder**, enter the name of your TFS project, followed by any child folders if you're using folders to organize reports.
5. For **TargetServerURL**, this should be something like http://*<server name>*//reportserver, as shown in Figure 26.

Figure 26

An example of settings that allow deploying a report to a TFS project

After you set up the deployment options, you can right-click a report in Solution Explorer and then click **Deploy** to add the report to the server (or update it).

Parameters from a Dataset

Quite often, when you select the check box in the Parameters column in the Query Builder, a new dataset will be created for you to populate the values for a parameter. The code that is generated is custom MDX that is hard to maintain. I prefer to replace these generated queries with my own versions that use the Query Builder so they're easier to read and maintain.

As an example, you might want to add a **State** parameter to the bug rate report so you can limit the states that are graphed.

Filters

Sometimes you'll get more data back from a query than you really want. This often happens when you add new columns and end up receiving rows with null values in the old columns. Usually, the query results don't include rows that have null values in all the columns, so adding a column that has values can show other rows that weren't returned before because the values were null.

How do you delete these extra rows? You delete them through filters, which are applied to the dataset ***results***. In other words, the filter removes rows from the dataset before passing them on to the report.

Filters are added to a dataset using in the properties dialog box for a dataset. To open this dialog box, click the ellipsis button (…) that is to the right of the dataset combo box, and then click the **Filters** tab.

You'll want to add a filter that looks something like the following (where the expression depends on the actual column you want to filter):

=IsNothing(Fields!Current_Work_Item_Count.Value)=False

One very important thing to notice is the equal sign in front of the value. Without the equal sign, the value will be treated as a string. However, the **IsNothing** function returns a Boolean operator, so you'll get a type mismatch error.

Further Reading

This article just scratches the surface of SQL Reporting Services and Analysis Services. We plan to continue to provide more information on our Web site. In addition, the following resources on MSDN may be helpful:

- Visual Studio Team System Process Templates and Tools
- "Get More Out of SQL Server Reporting Services Charts"
- "Using Extended Field Properties for an Analysis Services Database"
- "Team Foundation Server Data Warehouse"