Adding Custom Data to the Oracle® Solaris 11.4 StatsStore and System Web Interface

Part No: E61819
August 2018

Adding Custom Data to the Oracle Solaris 11.4 StatsStore and System Web Interface

Part No: E61819

Référence: E61819

Accès aux services de support Oracle

Les clients Oracle qui ont souscrit un contrat de support ont accès au support électronique via My Oracle Support. Pour plus d'informations, visitez le site `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` ou le site `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` si vous êtes malentendant.

Contents

Tables

Examples

Using This Documentation

- **Overview** – Describes how to provide statistics and performance data in Oracle Solaris
- **Audience** – Application developers who want Oracle Solaris users to be able to use the Oracle Solaris System Web Interface and CLI to get information about the activities and performance of the application
- **Required knowledge** – Experience administering Oracle Solaris systems

Product Documentation Library

Documentation and resources for this product and related products are available at http://www.oracle.com/pls/topic/lookup?ctx=E37838-01.

Feedback

Provide feedback about this documentation at http://www.oracle.com/goto/docfeedback.

♦ ♦ ♦ CHAPTER 1

Adding Data to the Oracle Solaris StatsStore

The Oracle Solaris StatsStore statistics store includes events from sources such as SMF services, FMA, and the auditing system and includes statistics from sources such as `kstat`, `dtrace`, `proc`, and some applications.

This chapter provides an overview of how you can add data from your application to the StatsStore.

Administrators and users of your application retrieve data to help maintain the application and diagnose problems. For information about how your users can retrieve data from the StatsStore, see *Using Oracle Solaris 11.4 StatsStore and System Web Interface*.

Adding Data: Steps and Best Practices

Following are the steps for adding data to the StatsStore and making that data easy to use in the Oracle Solaris System Web Interface.

▼ How to Add Data

1. **Decide what data to provide.**

   Provide only data that users are likely to want in order to use the application as effectively as possible or to troubleshoot problems. Adding all available data to the statistics store might cause administrators to spend more time finding the data they really need.

   - What are the most meaningful metrics for this application?
   - What information are users most likely to need to diagnose a performance degradation?
   - Can you provide useful data about how the application is interacting with users, with other applications, or with the operating system?

2. **Decide the best form in which to provide the data.**

In most cases, you should provide counter data. See “Type of Data to Provide” on page 15 for more information.

3. **Define the statistics.**

   Create metadata files that define your statistics as described in “Adding Resources and Statistics to the Statistics Store Namespace” on page 25.

   - Give each statistic a descriptive SSID (`id`) and an informative description.
   - Set the value of `stability` to `stable` so that all users can browse the data. Similarly, be very judicious about specifying authorizations required to read data. Anyone who is authorized to use your application should be able to browse the data about the application.
   - Specify any resources, partitions, and topology mappings that are needed.

   See the `ssid-metadata`(7) man page for best practices for naming resources, statistics, and partitions.

4. **Modify your application to update values for the statistics that you created in metadata.**

   Use the interfaces described in “Interfaces for Providing Statistic Values” on page 17.

   If you are not using static resources, your application might need to create resources as well. See “Adding Resources Dynamically” on page 29.

5. **Record data values.**

   Recording data values causes the values to be stored in the statistics store so that users can retrieve current and historical values. When your application writes data values, each value overwrites the previous value unless the values are being requested.

   - Users can record values by displaying a sheet that contains those statistics in the System Web Interface while the application is writing the values.
   - Users can record values by running the `sstore capture` command while the application is writing the values.
   - An application can record values by calling `sstore_data_read()` while the application is writing the values.
   - To persistently record values whenever the values are written, create and enable a collection as described in “Creating a Collection” on page 30.

6. **Create charts and graphs of the data.**

   How will your users use this information? Decide how the information should be organized into groups of charts and graphs for most effective use in the System Web Interface. What type of visualization is most appropriate for this data?

Typically, each statistic is shown on a separate visualization. Related statistics can be shown on the same visualization if the statistics are in the same units and can be shown in the same time scale.

Related events can be included on the same visualization with other data for visualizations with a time axis.

Will your users want to see some system-provided statistics along with your application statistics? You can include visualizations for those statistics on the same sheet with your application statistics for easier visual correlation.

Create metadata files that describe your visualizations as described in "Creating Visualizations" on page 32. Give each visualization and sheet a useful title and description.

Type of Data to Provide

This section discusses the kinds of data that are most useful for analyzing performance issues.

This section also discusses how to organize related data.

- An application might have multiple components that each need to report the same statistics.
- A statistic might need multiple components to fully explain the total value.
- Statistics might benefit from topology mapping to enable administrators to more easily find the data they need.

Data Type

Counts are the most efficient type of data to provide. For example, you might provide the number of transactions or number of bytes read. Count data will be monotonically increasing and not very useful for troubleshooting. You can use the built-in operators to show the count data as another type of data such as the rate of change or percent utilization. See Chapter 4, "Performing Operations on Statistic Values" in *Using Oracle Solaris 11.4 StatsStore and System Web Interface* for descriptions of the built-in operators.

Components of the Application

Does the application have subcomponents for which users might want separate data? Does the application provide the same data about different parts of the application? Define these subcomponents or parts as resources and provide the same statistics for each resource. For

example, `nscd` provides the same set of statistics for resources such as `ipnodes`, `networks`, and `services`. The `nscd` StatsStore resources are the resources for which `nscd` provides caching.

For a class that has resources defined, any class-level statistics (statistics that are defined directly on the class) should also be defined on each resource.

Class-level statistics are a useful way to combine data across all resources so that users do not need to apply operations to the SSIDs. For example, for a class that has resources defined, the class-level statistic could be the sum or average of that same statistic for each resource.

The `//:class.cpu//:stat.usage` class-level statistic shows the total usage of all CPUs in the system, and the `//:class.cpu//:res.id/0//:stat.usage` resource statistic shows the total usage for one CPU. The value of `//:class.cpu//:stat.usage` is equal to the value of `//:class.cpu//:res.id/*//:stat.usage//:op.sum`. The `nscd` application does not provide any class-level statistics because users do not want to combine all positive hits, for example, for `ipnodes`, `networks`, `services`, and other `nscd` resources.

Another reason to define resources for your application is to use the resources to provide topology mapping so that users can access the statistics data in different ways.

Do not define resources that are not interesting to users. For example, CPUs are exposed as resources, but DIMMs are not.

Components of a Statistic

Can the value of a statistic be subdivided into useful parts such that the sum of the values of the parts accounts for one hundred percent of the value of the statistic? Define a partition for such a statistic. If the sum of the values of the parts does not equal the value of the unpartitioned statistic, do not create a partition for the statistic.

For example, `//:class.cpu//:stat.usage` shows the total usage of all CPUs in the system, and `//:class.cpu//:stat.usage//:part.mode` shows the total `idle`, `inter`, `kernel`, `stolen`, and `user` usage of all CPUs in the system. Similarly, `//:class.cpu//:res.id/0//:stat.usage` shows the total usage of CPU 0, and `//:class.cpu//:res.id/0//:stat.usage//:part.mode` shows the total `idle`, `inter`, `kernel`, `stolen`, and `user` usage of CPU 0. The value of `//:class.cpu//:res.id/0//:stat.usage` is equal to the sum of the values of all parts of the `//:class.cpu//:res.id/0//:stat.usage//:part.mode` partition.

Partitions of a statistic should have approximately equal capacity. If the capacities vary greatly, consider providing a normalized statistic. For example, users should be able to easily see that a 1GB network card is 90% utilized and a 10GB network card is 10% utilized.

Topology Map

Will topology mapping of some statistics benefit your users? You need resources to provide topology. Users can list the resources of a class to explore the topology without capturing the data.

Interfaces for Providing Statistic Values

Use one of the following interfaces to write data values to the statistics store from your applications:

- `sstore_data_attach`
- `sstore_data_update`
- `sstore_data_bulk_update`

All of these interfaces are available for both C and Python. See the `libsstore`(3LIB), `sstore_data_attach`(3SSTORE), `libsstore`(python), and `sstore`(3rad) man pages.

All of these interfaces perform the following tasks:

- Enable any static resources in any of the specified SSIDs that are not currently enabled.
- Mark affected resources as actively provided.

Use the `sstore_data_attach_histogram()` interface to record the number of times your statistic values occur within predefined ranges or intervals. The `sstore_data_attach_histogram()` interface can be used only with one of the three interfaces listed above.

See the `libsstore`(3LIB) man page for the complete list of statistics store interfaces. Interfaces in the `libsstore` library provide the following capabilities:

- List statistic and event identifiers
- Update values for statistics
- Read statistic and event value data and metadata
- Add resources and resource-specific metadata to the statistics store
- Remove resources
- Add resource topology to the statistics store by using the metadata argument of `sstore_resource_add()`

To update data statistic values periodically, consider using an SMF periodic service, probably with the `sstore_bulk_update()` interface. For more information about periodic services, see Chapter 3, "Creating a Service to Run Periodically" in *Developing System Services in Oracle Solaris 11.4*.

Note that updating statistic values does not add the updated value to the statistics store. The statistics store accumulates data values only when requested by a client. The requesting client can be the `sstore capture` command or `sstore_data_read()`. The following clients use `sstore_data_read()` to request statistic values:

- A sheet that contains the statistic in the System Web Interface
- An application that calls `sstore_data_read()`
- An enabled collection

Updated statistic values continue to be written to the statistics store for a short time after the last request or until the application that is updating the values stops providing updates.

Comparing Methods for Providing Statistic Values

Whether you use the `sstore_data_attach()` or `sstore_data_update()` API depends on the data type, the frequency of additions, and the importance of the time stamp of each value.

TABLE 1 `sstore_data_attach()` and `sstore_data_update()` Comparison

Feature	`sstore_data_attach()`	`sstore_data_update()`
Architecture	Writes statistic values to memory space that is shared between the application and the statistics store.	Writes statistic values directly to the statistics store.
Statistic data type	Integer, especially a counter	Any type
Initial data value	0	Specified value
Time stamp	When the data is requested by the statistics store client	When the data is collected by the application

Use `sstore_data_attach()` if the statistic values are integer and if the time the value was collected is not needed. Statistic value time stamps in this case are the time that the statistics store retrieves the values from the shared memory area. The `sstore_data_attach()` method is also recommended if you need to make large numbers of data updates per time period.

Use `sstore_data_update()` if the statistic values are not numeric or if the exact time the data was collected is needed. You must use the `sstore_data_update()` method if multiple statistic

values must be updated simultaneously such that each statistic value must have the same time stamp.

See the following sections for more information about `sstore_data_attach()`, `sstore_data_update()`, `sstore_data_bulk_update()`, and `sstore_data_attach_histogram()`. See also the `sstore_data_attach`(3SSTORE) man page and the `sstore_data_read`(3SSTORE) man page.

The `sstore_data_attach()` Interface

The `sstore_data_attach()` interface creates a shared memory region between `sstored` and the client process. Call `sstore_data_attach()` only one time during the runtime of the client process.

The `sstore_data_attach()` interface supports only integer statistic values, particularly counters. The `sstore_data_attach()` interface creates a shared memory region with one counter for each statistic. The values in this shared memory region are initialized to 0. To update the statistics store, update the shared memory region array element for that statistic. For `sstore_data_attach()`, updating statistic values typically means incrementing the counter for that statistic.

When these statistic values are requested, `sstored` attempts to read from the shared memory region once each second, regardless of how quickly the application updates the value.

See Chapter 3, “Adding Simple Data Values to the Statistics Store” for example applications that use `sstore_data_attach()`.

The `sstore_data_update()` Interface

The `sstore_data_update()` interface supports any data type for statistic values, including string and other large data types. Each time the statistics store client requests a value update, call `sstore_data_update()` to update the values of the specified SSIDs with the given values.

See Chapter 6, “Adding Any Type of Data to the Statistics Store” for example applications that use `sstore_data_update()`.

The `sstore_data_bulk_update()` Interface

The `sstore_data_bulk_update()` interface enables an application to provide a value pair (time stamp, value) for statistics. Statistic values do not need to be provided in real time because the application provides the time stamp along with the value.

An example of a case when `sstore_data_bulk_update()` is needed is for a third-party application that cannot be modified to talk to `sstored` directly. In such a case, statistic values can be read from a proxy such as a log file.

The `sstore_data_bulk_update()` interface supports any data type for statistic values, including string and other large data types.

Statistic updates must be provided in chronological order: Each update must have a time stamp that is more recent than the time stamp of any update that was previously provided. Any update that has a time stamp that is not more recent than the time stamp of the previous update is ignored. The `sstore_data_bulk_update()` interface writes to the statistics store all the statistics value pairs that are newer than the most recently provided data for that statistic.

Statistics store clients such as `sstore capture`, `sstore export`, or `sstore_data_read()` return the time stamps provided by `sstore_data_bulk_update()` for each statistic value. If a consumer requests data for a time for which the application has not yet provided data, the value `SSTORE_VALUE_NODATA_YET` is returned.

The metadata for any statistic for which data will be provided by `sstore_data_bulk_update()` must define `min-update-interval`, which is the minimum number of seconds between any two bulk updates. Consumers such as the System Web Interface use this value to set expectations about the update frequency of a statistic.

The `sstore_data_attach_histogram()` Interface

Like `sstore_data_attach()`, the `sstore_data_attach_histogram()` interface creates a shared memory region between `sstored` and the client process. Call `sstore_data_attach_histogram()` only one time during the runtime of the client process.

The `sstore_data_attach_histogram()` interface creates a shared memory region with an array of counters for each statistic: one counter for each interval defined for the histogram. To update the statistics store, call `sstore_histogram_quantize()` to update the appropriate interval counter in the shared memory region. The statistic value can be any data type; `sstore_data_attach_histogram()` stores a count of the values. See the `sstore_histogram_init`(3SSTORE) man page for more information about interfaces used to store data in a histogram and the types of histograms that are supported by the statistics store.

Latency is a good example for a histogram. If you capture or export a statistic such as `//:class.disk//:stat.io-completions//:part.latency` or `//:class.scheduler//:stat.cv-signal//:part.latency`, you see a long list of intervals. The value of each interval is the number of times the value of the statistic is in that interval. The following example shows that

since system boot, the number of I/O operations that completed in approximately one-half to one second is 191324.

```
$ sstore export //:class.disk//:stat.io-completions//:part.latency
...
                    524288: 112392.0
                    1048576: 191324.0
...
```

Removing Resources and Statistics

The `sstore_resource_remove()` interface performs the following tasks:

- Decrements the internal reference count.
- Removes the resource from the statistics store namespace if both of the following criteria are met:
  - The reference count is 0.
  - No statistics associated with the resource have an active provider.

Troubleshooting Providing Statistics

If your statistic values do not update as expected, check for the following issues:

- The class, resource, statistic, or event that you are updating is not defined in a JSON file in `/usr/lib/sstore/metadata/json/`.
- The identifier names used in the application that provides the statistic values do not match the names defined in the metadata files.
- The type of the data specified in the application does not match the data type specified in the metadata files.
- The statistics store service has not been restarted to read new or changed metadata.
- The caller of the application does not have the `solaris.sstore.write` authorization and is not the authorized user for write operations on that namespace.
- An identifier that your application attempts to update is already updated by a different application.

  A statistic cannot have multiple providers. If one application initializes a statistic and then another application initializes the same statistic while the first application is still actively writing values for those statistics, the second application will receive an error from `sstore_data_attach()`, `sstore_data_update()`, or `sstore_data_bulk_update()`.

Check the application log file for error messages.

If the application is managed by an SMF service, use the `svcs -Lv` command to check the log file for that service.

If you attempt to retrieve data values before the event or statistic is initialized, you receive an error message that the identifier is not valid.

If you attempt to retrieve data values after the event or statistic is initialized but before any value has been provided, the identifier is valid but no value is shown.

How to Force a Reread of all Metadata

If you make a metadata change that is not read by restarting the `sstore:default` service, you can force a reread of all metadata by destroying the entire statistics repository.

Caution - This action destroys all historical data.

1. **Disable the statistics store service.**

```
$ svcadm disable sstore:default
$ svcs sstore
STATE          STIME    FMRI
disabled       13:03:54 svc:/system/sstore:default
```

2. **Destroy the statistics store repository.**

```
$ zfs destroy rpool/VARSHARE/sstore
```

If the repository still exists, remove it:

```
$ rm -fr /var/share/sstore/repo
```

Note - Removing or modifying individual directories or files within the statistics store repository is not supported. Removing or modifying metadata directories or files that were delivered with Oracle Solaris is not supported.

3. **Reread the statistics metadata.**

```
$ svcadm enable sstore:default
$ svcs sstore
STATE          STIME    FMRI
```

```
online         13:04:46 svc:/system/sstore:default
```

If the service is not online, check the service log file.

```
$ svcs -Lx sstore
```

♦ ♦ ♦ CHAPTER 2

Defining Custom Statistics

All classes, resources, and statistics must be initialized with metadata. Metadata provides descriptions of statistics store identifiers and also provides information such as type, units, and authorization. A class specifies resources and statistics that are valid for that class.

This chapter shows how to add metadata to do the following tasks:

- "Adding Resources and Statistics to the Statistics Store Namespace" on page 25, including "Defining Resources" on page 27 and "Defining Partitions" on page 29, describes how to define new class instances, resources, statistics, and partitions. Typically you will define a new instance of the existing `app` class.
- "Mapping Topology" on page 30 describes how to map a statistics store identifier to another identifier name that might be easier for administrators to use.
- "Creating a Collection" on page 30 describes how to define a new collection.
- "Creating Visualizations" on page 32 describes how to define new sheets and visualizations to display in the System Web Interface.
- "Authorizing Access to Resources and Statistics" on page 36 describes how to authorize users to collect and view resources and statistics.

Adding Resources and Statistics to the Statistics Store Namespace

Many resources, such as disks or other devices, are added to the statistics store automatically when the devices are added to the system. System statistics such as kstats and information about services and faults are included in the statistics store by default.

You can use one of the following methods to add your custom resources and statistics to the statistics store namespace:

- JSON text files. See "Using Statistics Store Metadata Files" on page 26.

- The `sstore_resource_add()` statistics store library interface. See "Adding Resources Dynamically" on page 29.

Using Statistics Store Metadata Files

By default, permission to create and modify files in the `/usr/lib/sstore/metadata/` directory is restricted to `root`.

JSON files delivered as part of the Oracle Solaris OS are in `/usr/lib/sstore/metadata/json/solaris`. Your custom metadata files can go in either the `/usr/lib/sstore/metadata/json/site` directory or the `/usr/lib/sstore/metadata/json/vendor` directory.

JSON files in `/usr/lib/sstore/metadata/json/` describe classes, resources, and statistics. These statistics are automatically added to the statistics store namespace on system start and any other time the statistics store service restarts.

To add statistics to the statistics store namespace, take the following steps:

1. Create one or more files in `/usr/lib/sstore/metadata/json/`. Do not modify files that you did not create.
2. Run the `soljsonfmt` tool on the `.json` file to check for JSON syntactic errors. See the `soljsonfmt`(1) man page for more information.
3. Run the `soljsonvalidate` tool on the `.json` file to check for JSON semantic errors. The `soljsonvalidate` tool reports deviation from defined schemas, such as a missing `id`. See the `soljsonvalidate`(1) man page for more information.

   All matching `.json` files are opened and checked for JSON syntax errors. No validation is performed if syntax errors are found.

You can describe each class and statistic in a separate file, or you can describe multiple components in one file by creating a more complex object using JSON array syntax. Resource definitions are part of the class definition. A single statistic definition can be used by multiple classes and resources. For example, many devices might count how many times they are interrupted. The same `stat.interrupt` statistic could be used with each of the different device resources.

The first character of a *class-name* must be an alphanumeric character; other characters in a *class-name* must be alphanumeric characters or the hyphen character (`-`). For alphabetic characters, best practice is to use lower case characters only. Examples of *class-name* values include `app`, `cpu`, `dev`, `io`, `link`, `pg`, `pset`, `svc`, `system`, and `zone`. The `//:class.solaris/` and `//:class.s/` class names are reserved.

A `class` and `res` pair is a *canonical* resource name. Each name space must be unique on each system. For example, a system cannot have more than one CPU with ID 0. If a given name can be used by multiple resources, then that name is not a suitable namespace identifier.

The `/` and `:` characters are allowed in resource names. The following `fmri` resource uses both `/` and `:` in the name. Resource names can contain any characters except the reserved three-character sequence `//:`. As much as possible, resource names are the same as names used in the related administrative commands.

```
//:class.svc//:res.fmri/system/identity:node
```

The metadata for a class defines how resources in that class can be named. Resources can use different naming schemes within a single class, as shown in the following examples from the `kstat` class:

```
//:class.kstat//:res.disk/sd/sd0/0
//:class.kstat//:res.device_error/sderr/sd0,err/0
//:class.kstat//:res.misc/pci-ide/fm/0
//:class.kstat//:res.zones/cpu/sys_zone_0/0
```

You can use metadata to provide a description of each component, provide topology mappings, and provide aliases. See the `ssid-metadata`(7) man page for more information about statistics store metadata files.

The statistics store reads all the data without regard to how the data is organized into different files. For documentation purposes for other developers and administrators, if you describe multiple components in a single file, all the components described in one file should be related.

Classes can define in metadata other classes in which their resources appear.

The `/usr/lib/sstore/metadata/json-schema/` directory contains JSON schema files that describe the format of the metadata files in `/usr/lib/sstore/metadata/json/`.

The properties `sensitive` and `expensive` restrict the use of statistics as described in "Restricting Access to Sensitive Data" on page 38 and "Restricting Capture of Data that is Expensive to Capture" on page 39.

Defining Resources

Some statistics apply to the entire class, and no separate resources need to be defined within that class. For example, the `//:class.app/solaris/sysstat/sysconf` system configuration statistics are for the entire system. No resources are defined for that class.

Some statistics have a value for the entire class and separate values for individual resources within the class. For example, `//:class.cpu//:res.id/0//:stat.usage` gives usage statistics

for one CPU in the system, while `//:class.cpu//:stat.usage` gives usage statistics for all CPUs in the system combined.

If you know the resources in the class at the time you define the class, define those resources as `static-instances` in the class `json` file as described in "Adding Static Resources" on page 28.

If some instances of the class are not known when you define the class, you can add those resources dynamically as described in "Adding Resources Dynamically" on page 29.

A single class can have some resources that are statically defined and others that are added dynamically.

A resource inherits class metadata, and you can add metadata that only applies to the resources.

Adding Static Resources

Resources are defined in a `static-instances` element in the class metadata file. The `name` and `namespace` properties are required. The `name` is the unique name of the particular resource. The `namespace` groups similar resources and matches the value of a `resource-name` in the `namespaces` element. You can add metadata for the resources in the `instance-metadata` element.

The following partial example of a class metadata file shows the required `namespaces` and `static-instances` elements and the optional `instance-metadata` element:

```
"instance-metadata": {
    "description": "Instances of util2",
    "stability": "stable"
},
"namespaces": [
    {
        "name-type": "string",
        "resource-name": "inst"
    }
],
"static-instances": [
    {
        "name": "inst1",
        "namespace": "inst"
    },
    {
        "name": "inst2",
        "namespace": "inst"
    }
```

```
]
```

See “Collect Data for Statically Allocated Resources” on page 57 for a complete example.

In the following excerpt from `class.svc.json`, one resource is defined. Other resources (services) are added dynamically.

```
    "namespaces": [
        {
            "name-type": "string",
            "resource-name": "fmri"
        }
    ],
...
    "static-instances": [
        {
            "name": "system/svc/restarter:default",
            "namespace": "fmri"
        }
    ]
```

Adding Resources Dynamically

If you need to create new resources from within your application, use the `sstore_resource_add()` interface. See the `sstore_resource_add`(3SSTORE) man page for details. Typically you should not need to add resources from within your application. Instead, you should define any needed resources as `static-instances` in the class `json` file and use your application only to add statistic values.

If your application will use the `sstore_resource_add()` interface to create resources dynamically, then you do not need to specify static resources in the metadata. You still need to provide resource namespace rules in the class `json` metadata file.

An example where resources are added dynamically is the `cpu` class, where the number of CPUs on the system is not known until the system is running.

See “Collect Data for Dynamically Allocated Resources” on page 65 for an example of how to add resources dynamically.

Defining Partitions

To have partitioned statistics, your application must have resources.

Partitions are a `stat-mapping` schema, and the SSID has a `stat-mapping` component instead of a `stat` component. Partitions are named in a `partitions` element within an `aggregations` or `instance-metadata` element in the metadata file, as shown in the following example:

```
"$schema": "//:stat-mapping",
"id": "//:class.app/util2//:stat-mapping.errors",
"instance-metadata": {
    "partitions": [
        "inst"
    ]
},
"transforms": [
    {
        "match": "//:class.app/util2//:res.inst/(inst[1-2])//:stat.errors$",
        "replace": "//:class.app/util2//:stat.errors//:part.inst(\\1)"
    }
]
```

In the `transforms` section, each `match` represents a statistic that is already defined in the `class` and `stat` metadata, and the paired `replace` maps that statistic to one of the partitions named in the `partitions` element. The value of the partitioned statistic is equal to the sum of the values of the matching statistics for each resource.

See Chapter 5, “Separating Data Into Partitions” for an example. See also the "Statistic Mapping" section in the `ssid-metadata`(7) man page.

Mapping Topology

See mapping information in the `ssid-metadata`(7) man page.

Creating a Collection

A collection is a convenient way to show multiple statistics using one SSID, as described in “Representing Sets of Statistics and Events” in *Using Oracle Solaris 11.4 StatsStore and System Web Interface* and “Using Collections” in *Using Oracle Solaris 11.4 StatsStore and System Web Interface*.

Use one of the following methods to create a collection:

- Create a JSON file in the `/usr/lib/sstore/metadata/collections/` directory and restart the `sstore` service. The rules for creating a collection JSON file are documented in the `ssid-collection.json`(5) man page.
- Use the API described in the `sstore_collection_alloc` (3SSTORE) man page.

After the collection is created, you can disable or enable the collection by using the following commands:

- `sstoreadm disable-collection`
- `sstoreadm enable-collection`

Note - Statistics in an enabled collection are recorded persistently. Recording too many statistics persistently can degrade system performance, especially if those statistics are expensive to record.

The `user1.db-rw.json` JSON file in the `/usr/lib/sstore/metadata/collections/` directory includes some user-created statistics in a collection:

```
{
    "$schema": "//:collection",
    "description": "DB rw",
    "enabled": false,
    "id": "db-rw",
    "ssids": [
        "//:class.app/db1//:stat.reads",
        "//:class.app/db1//:stat.writes",
        "//:class.app/db2//:stat.reads",
        "//:class.app/db2//:stat.writes"
    ],
    "user": "user1"
}
```

This collection is not enabled by default. Instead of persistently recording these statistics, this collection provides a convenient way to access these statistics by using one SSID.

Only `user1` can change this collection by using the System Web Interface or the `sstoreadm` command.

Run the `soljsonfmt` tool on the `.json` file to check for JSON syntactic errors. Run the `soljsonvalidate` tool on the `.json` file to check for JSON semantic errors. See “Using Statistics Store Metadata Files” on page 26 for more information. Make sure you are in a directory where you have write privilege because the `soljsonfmt` command creates a temporary file.

```
# soljsonfmt /usr/lib/sstore/metadata/collections/user1.db-rw.json
#
```

Restart the `sstore` service. Now you are able to use the collection.

```
# svcadm restart sstore:default
# sstore list //:class.collection//:collection.name/user1/db-rw
IDENTIFIER
//:class.collection//:collection.name/user1/db-rw
```

You cannot use the `sstore list` command to list the statistics that belong to the collection. Use the `sstore info` command instead.

```
# sstore info //:class.collection//:collection.name/user1/db-rw
Identifier: //:class.collection//:collection.name/user1/db-rw
  ssid: //:class.app/db1//:stat.reads
  ssid: //:class.app/db1//:stat.writes
  ssid: //:class.app/db2//:stat.reads
  ssid: //:class.app/db2//:stat.writes
 state: disabled
  uuid: 1ca6f562-e00d-42ed-b288-8047345507b6
 owner: user1
 cname: db-rw
crtime: 1464997510390334
```

You can use the collection SSID to easily record all the statistics that are in the collection:

```
# sstore capture //:class.collection//:collection.name/user1/db-rw
TIME                VALUE IDENTIFIER
2016-06-03T16:54:20 33352207 //:class.app/db1//:stat.reads
2016-06-03T16:54:20 33316406 //:class.app/db1//:stat.writes
2016-06-03T16:54:20 16438126 //:class.app/db2//:stat.reads
2016-06-03T16:54:20 16370542 //:class.app/db2//:stat.writes
```

Creating Visualizations

Visualizations enable administrators to view data graphically in the System Web Interface, as described in Chapter 2, “Using Oracle Solaris System Web Interface” in *Using Oracle Solaris 11.4 StatsStore and System Web Interface*.

Sheet and Visualization Design Best Practices

Visualizations are organized into groups, which are organized into sections, which are on a sheet.

- Define the purpose of the sheet. When will an administrator use this sheet? How will an administrator use this sheet to learn more about this application or subsystem?

A sheet should be self explanatory and guide the administrator through the troubleshooting process. A sheet should help an administrator discover contributors to a particular problem and lead the administrator to specific actions.

- Determine the information needed to achieve the purpose. What information is most key? What information is related?

  Resource utilization and saturation and number and type of errors typically are key information. Can you show utilization rather than raw usage numbers?

  Time-series visualisations provide the most data dense visualisation that include historical data. Other types of visualizations should only be used if the time-series visualisation does not suit the underlying data.

  Consider using `//:op.top` to avoid switching to Pareto visualization automatically when defining a time-series visualization.

  Are historical comparisons of data values useful? If so, should historical comparisons be shown by default?

  Is the required information available from existing statistics, do new statistics need to be created, do existing statistics need to be partitioned or mapped to achieve the purpose of this sheet?
- Determine the layout of the sheet into sections and groups. What flow should the administrator likely to follow to troubleshoot a specific problem or explore potential problems with this application or subsystem?
  - The most key information should be first, at the top of the sheet. To aid troubleshooting, provide information about potential problems or errors first, instead of starting with basic status information.
  - Sections should guide the flow of problem investigation. Based on what the most key information in the first section shows, what should the administrator look at next? Group the key indicators for the most common symptoms of problems with this subsystem into sections.
  - Related information should be in the same group.
  - Find the right balance. Organizing data into multiple groups and sections can make data easier to understand and use and reduce clutter on the screen, but too many groups and sections add to the clutter and make problems harder to investigate. Also, each section and group title takes screen real estate that visualizations could use.
- Use the purpose of the sheet to create a brief description of the sheet.
  - What data does this sheet provide?
  - What kinds of failures can this sheet help diagnose?
  - If possible, mention diagnosis specifics such as how visualizations are related and used together. What are typical steps to take to troubleshoot the system represented by this sheet?
  - Include links to additional information.

- Create a brief description for each section.
  - What is the purpose of this section? Which symptoms can be diagnosed here?
  - What steps should an administrator take here?
  - Include links to additional information.
- Create a brief description for each visualization.
  - How can the data in this visualization be used to diagnose a problem?
  - How can an administrator compare the data values in this visualization to expected values?
  - Mention specifics such as how data is partitioned, what related data can be shown, what you can learn from this data.
  - Include links to additional information to help troubleshoot the problem, including references to visualizations on other sheets and how those would be helpful.

  If you cannot clearly express the purpose of the visualization, perhaps the visualization is not needed to investigate problems with this subsystem.
- Decide what type of visualization will make the data easiest to understand and use.
  - Add operations to the SSIDs to show the most meaningful data. Do you want to show resource utilization, resource saturation, error counts, rate, top five values? Do you want to filter the data or convert to a different unit of measurement?
  - How important is historical data? To show historical data, use one of the time series type of visualizations. Consider whether the data identifiers should be in an enabled collection and collected persistently.
  - Is a stacked time series or Pareto chart better than a regular time series graph for this data? Is a gauge, bar chart, or pie chart better for what you want to show? Do you need a histogram to show which values occur most frequently?
  - Is data that is represented by different SSIDs related closely enough to show in the same visualization? For example, reads and writes of the same resources might be shown in a single visualization. To show two different SSIDs in one visualization, both SSIDs must be partitioned identically (or not partitioned), must be in the same units in a similar range of possible values, and must be able to be shown at the same time scale.
  - Are certain events useful to include with this data? Events can be shown as points on a time graph.

How to Create a Visualization

To create a visualization, use one of the following methods:

- Use the System Web Interface as described in "How to Create a Visualization by Using the System Web Interface" on page 35.

- Create a JSON file in the `/usr/lib/webui/analytics/sheets/vendor` or `/usr/lib/webui/analytics/sheets/site` directory and restart the `webui/server` service.

  Use one of the following methods to create the JSON file:

  - Export the file from the System Web Interface. In the System Web Interface, create a new sheet or copy and modify an existing sheet. To copy a sheet, open the sheet and select the Duplicate & Edit option from the Sheet Actions menu. Modifications that you make in the System Web Interface are saved in your user preferences file (`/var/user/`*user-name*`/webui/preferences/solaris.json`) and only you can view them. From the Sheet Actions menu, select the Export option to save the JSON description of the sheet in the `/usr/lib/webui/analytics/sheets/vendor` or `/usr/lib/webui/analytics/sheets/site` directory so that all users can view it.
  - Write the JSON code yourself using the following resources:
    - The `analytics`(5) man page
    - The `/usr/lib/webui/analytics/sheets/analytics-import.schema.json` JSON schema file
    - Sheet definitions in `/usr/lib/webui/analytics/sheets/solaris`
    - Examples in later chapters in this guide

▼ How to Create a Visualization by Using the System Web Interface

This procedure describes all steps, starting with creating a new sheet. You can skip some of these steps by modifying an existing sheet. Open a sheet that is similar to what you want, and select the Duplicate & Edit option from the Sheet Actions menu. Then open the duplicate sheet and modify the sections, groups, and visualizations as necessary.

1. **Create a sheet.**

   At the top of the Sheets page, select the Add Sheet button. Give the new sheet a useful name and description.

2. **Add a section to the sheet.**

   From the Sheet Actions menu, select Add Section. From the Section Actions menu, select the Rename option to give the section a meaningful name.

3. **Add a group to the section.**

   From the Section Actions menu, select Add Group. From the Group Actions menu, select the Properties option and give the group a useful name and description.

4. **Add a visualization to the group.**

Use one of the following methods to add a visualization:

- **From the bottom of the Group Properties pop-up, select Add Visualization.**
- **From the Group toolbar, select the + icon.**

Give the visualization a useful name.

From the Visualization Actions menu, select the Properties option and give the visualization a useful description.

5. **Add a statistic or event to the visualization.**

   Use one of the following methods to add a statistic or event:

   - **From the Visualization Properties pop-up, select Statistics or select Events and select the + icon.**
   - **From the Visualization Actions menu, select the Add Statistic or Event option.**

   Follow the prompts in the dialog. See the System Web Interface help for more information.

   From the Visualization Actions menu, select the Visualization Type option to select the type of chart or graph that is most appropriate for this statistic.

   From the Visualization Actions menu, select the Set Time Range → Custom option to select the period length that is most appropriate for monitoring this statistic. Be sure to select Ending Now.

Authorizing Access to Resources and Statistics

By default, any user can read and record any data in the statistics store. Anyone who is authorized to use your application should be able to browse the data about the application. Some other operations, such as reading sensitive data, are restricted. Table 2, “Statistics Store Operation Authorizations,” on page 37 provides information you need to authorize access to restricted statistics store operations. The listed operations can be performed by any user that has the associated authorization. See the `sstore-security`(7) and `sstore-authorized-user`(7) man pages for more information. The `root` user or role has all `solaris` authorizations. Most users do not have these authorizations. You might need to assign alternative authorizations to enable a daemon or application to manipulate certain statistics store data.

You can specify a particular authorization to grant access to any user who has that authorization, or you can authorize specified users. The authorization applies to the node where the

authorization is specified in the metadata and to any non-topological descendant nodes. For example, if you specify an authorized user for a class, that user can perform the specified operation on any statistics in that class. If you specify an authorized user for a statistic but not for the class, that user can perform the specified operation only on that statistic, not on other statistics in that class.

You can authorize access for any user who has a specified authorization.

- Imply a specific required authorization by setting the `sensitive` or `expensive` property to `true`.
- Specify a required authorization as the value of an `sau_`*`op_name`*`_auth` property. See the table for values of *op_name*.

You can authorize access to an operation for specified users.

- Specify a list of user names as the value of an `sau_`*`op_name`*`_username` property. See the table for values of *op_name*. An authorized user can be a human user or a daemon. A user is also called a client.

TABLE 2 Statistics Store Operation Authorizations

Property *op_name*	**Authorization**	**Authorized Operation**	**Interface**
`read_sensitive`	`solaris.sstore.read.sensitive`	Read a sensitive statistic or event.	`sstore_data_read()`, `sstore_batch_data_read()`, `sstore_info_read()`, `sstore_batch_info_read()`, `sstore_namespace_list()`, `sstore_batch_namespace_list()`, `sstore export`, `sstore info`, `sstore list`
`capture_sensitive`	`solaris.sstore.capture.sensitive`	Record a sensitive statistic or event.	`sstore_data_read()`, `sstore_batch_data_read()`, `sstore capture`
`capture_expensive`	`solaris.sstore.capture.expensive`	Record an expensive statistic or event.	`sstore_data_read()`, `sstore_batch_data_read()`, `sstore capture`
`update_res`	`solaris.sstore.update.res`	Add a resource to a class.	`sstore_resource_add()`
`update_res`	`solaris.sstore.update.res`	Deactivate a resource that was created by a previous `sstore_resource_add()` call.	`sstore_resource_remove()`
`write`	`solaris.sstore.write`	Provide statistic or event data.	`sstore_data_attach()`, `sstore_data_update()`
`delete`	`solaris.sstore.delete`	Purge statistic or event data.	`sstoreadm purge`

Property *op_name*	Authorization	Authorized Operation	Interface
`config`	`solaris.sstore.configure`	Update a collection created by another user.	`sstore_collection_write()`, `sstore_collection_set_state()`, `sstore_collection_update_ssid()`, `sstore_collection_delete()`

Restricting Access to Sensitive Data

To restrict access to sensitive data, mark the data sensitive by specifying the `sensitive` property with the value `true`. Statistics and events that have the `sensitive` property set to `true` require a user to have the `solaris.sstore.read.sensitive` authorization to read the data and have the `solaris.sstore.capture.sensitive` authorization to capture the data. A user that has the `solaris.sstore.read.sensitive` authorization can export data values of any statistic in the statistics store. A user that has the `solaris.sstore.capture.sensitive` authorization can record data values of any statistic in the statistics store except statistics that are expensive to capture as described in "Restricting Capture of Data that is Expensive to Capture" on page 39.

To enable access by other users who need to read or record this data, specify an alternative authorization that is more targeted to this data, or specify particular users that are authorized to access this data.

To specify an alternative authorization that enables a user to access particular sensitive data, specify the `sau_read_sensitive_auth` property or the `sau_capture_sensitive_auth` property with the alternative authorization as the value. The value of these properties can be a list of authorizations.

To enable a specified user to access particular sensitive data even if the user has none of the required authorizations, specify the `sau_read_sensitive_username` property or the `sau_capture_sensitive_username` property with the user name as the value. The value of these properties can be a list of user names.

Note - Even if you specify alternative authorizations or authorized users, you must still set the `sensitive` property to `true`.

EXAMPLE 1 Specifying Which Users Can Read Particular Sensitive Data

The following partial metadata for a sensitive statistic enables the following users to export values of this statistic:

- Any user that has the `solaris.sstore.read.sensitive` authorization

- Any user that has the `solaris.system.sysevent.read` authorization
- The `authorizeduser1` user
- The `authorizeduser2` user

```
{
  "sensitive" : "true"
  "sau_read_sensitive_auth" : "solaris.system.sysevent.read"
  "sau_read_sensitive_username" : "authorizeduser1" "authorizeduser2"
}
```

EXAMPLE 2 Specifying Which Users Can Record Particular Sensitive Data

The following partial metadata for a sensitive statistic enables the following users to record values of this statistic:

- Any user that has the `solaris.sstore.capture.sensitive` authorization
- Any user that has the `solaris.system.sysevent.write` authorization
- The `authorizeduser1` user

```
{
  "sensitive" : "true"
  "sau_capture_sensitive_auth" : "solaris.system.sysevent.write"
  "sau_capture_sensitive_username" : "authorizeduser1"
}
```

Restricting Capture of Data that is Expensive to Capture

If capturing certain statistic data has a high cost in system resources, you might want to restrict who can capture that data. For example, using DTrace scripts to record statistic data often has a high cost in system resources.

To restrict who can capture data that is costly to capture, mark the data costly by specifying the `expensive` property with the value `true`. Statistics and events that have the `expensive` property set to `true` require a user to have the `solaris.sstore.capture.expensive` authorization to capture the data. A user that has the `solaris.sstore.capture.expensive` authorization can record data values of any statistic in the statistics store except statistics that are sensitive as described in "Restricting Access to Sensitive Data" on page 38.

To enable access by other users who need to record this data, specify an alternative authorization that is more targeted to this data, or specify particular users that are authorized to record this data.

To specify an alternative authorization that enables a user to record particular expensive data, specify the `sau_capture_expensive_auth` property with the alternative authorization as the value. The value of this property can be a list of authorizations.

To enable a specified user to record particular expensive data even if the user has none of the required authorizations, specify the `sau_capture_expensive_username` property with the user name as the value. The value of this property can be a list of user names.

Note - Even if you specify alternative authorizations or authorized users, you must still set the `expensive` property to `true`.

EXAMPLE 3 Specifying Which Users Can Record Particular Expensive Data

The following partial metadata for a statistic that is expensive to record enables the following users to record values of this statistic:

- Any user that has the `solaris.sstore.capture.expensive` authorization
- Any user that has the `solaris.system.sysevent.write` authorization
- The `authorizeduser1` user

```
{
  "expensive" : "true"
  "sau_capture_expensive_auth" : "solaris.system.sysevent.write"
  "sau_capture_expensive_username" : "authorizeduser1"
}
```

EXAMPLE 4 Specifying Users Who Can Record Data that is Sensitive and Expensive

A statistic could be both sensitive and expensive. The following partial metadata for a statistic restricts the ability to capture values of this statistic to the following users:

- Any user that has the `solaris.sstore.capture.sensitive` authorization
- Any user that has the `solaris.sstore.capture.expensive` authorization
- Any user that has the `solaris.system.sysevent.write` authorization
- The `authorizeduser1` user

```
{
  "sensitive" : "true"
  "expensive" : "true"
  "sau_capture_sensitive_auth" : "solaris.system.sysevent.write"
  "sau_capture_sensitive_username" : "authorizeduser1"
  "sau_capture_expensive_auth" : "solaris.system.sysevent.write"
  "sau_capture_expensive_username" : "authorizeduser1"
```

```
}
```

Authorizing the Ability to Add and Remove Resources and Statistic and Event Data

You can specify who can add resources to a class and deactivate those resources, and who can add and purge statistic and event data values.

A user who has the `solaris.sstore.update.res` authorization can add a resource to any class in the statistics store. To enable other users to add resources in a specific class, set the `sau_update.res_auth` property on the class to specify an alternative authorization, or set the `sau_update.res_username` property on the class to authorize particular users. These users are also able to deactivate resources that they created.

A user who has the `solaris.sstore.write` authorization can add or update any statistic or event data values. A user who has the `solaris.sstore.delete` authorization can purge any statistic or event data values from the statistics store. To enable other users to add, update, or purge specific statistic or event data, set the `sau_write_auth` and `sau_delete_auth` properties to specify an alternative authorization, or use the `sau_write_username` and `sau_delete_username` properties to authorize particular users.

Authorizing the Ability to Configure a Collection

Configuring a collection includes adding statistics and events to the collection, removing statistics and events from the collection, enabling or disabling the collection, and deleting the collection. A user who has the `solaris.sstore.configure` authorization can configure any collection. The user who created the collection can configure that collection.

To enable other users to configure a particular collection, set the `sau_config_auth` property on the class to specify an alternative authorization, or set the `sau_config_username` property on the class to authorize particular users.

♦ ♦ ♦ CHAPTER 3

Adding Simple Data Values to the Statistics Store

This chapter shows a simple example that demonstrates the most common steps of adding statistics to the statistics store and viewing the values.

The example described in this section creates a class for an application and associates statistics directly with that application. These statistics are sometimes called **class statistics**.

- "Populate the Statistics Store Namespace" on page 43 shows how to create the class and statistic metadata.
- "Create an Application that Writes Statistic Values" on page 46 shows how to use the `sstore_data_attach()` interface to add values for the statistics.
- "Update and View Statistic Values" on page 48 shows how to make the data available to administrators.
- "Create a Graph to Visualize the Statistic Values" on page 50 shows how to display the data on a sheet that administrators can view in the System Web Interface.

Populate the Statistics Store Namespace

In this example, the application that produces the statistics is a custom application named `util1`.

In the `/usr/lib/sstore/metadata/json/site/` directory, create the file `class.app.util1.json` with the following content to define the class for the `util1` application. Note that files in `/usr/lib/sstore/metadata/json/site/` are owned by `root`.

```
{
    "$schema": "//:class",
    "description": "Example using count data and sstore_data_attach()",
    "id": "app/util1",
```

```
    "stability": "stable",
    "stat-names": [
        "//:stat.reads",
        "//:stat.writes",
        "//:stat.errors"
    ]
}
```

If you might change the semantic meaning of a statistic (for example, the type of the statistic), set the value of `stability` to `unstable`.

To define the statistics for this example, create the file `stat.util1.json` with the following content in the `/usr/lib/sstore/metadata/json/site/` directory:

```
[
    {
        "$schema": "//:stat",
        "description": "reads",
        "id": "//:class.app/util1//:stat.reads",
        "stability": "stable",
        "type": "counter",
        "units": "operations"
    },
    {
        "$schema": "//:stat",
        "description": "writes",
        "id": "//:class.app/util1//:stat.writes",
        "stability": "stable",
        "type": "counter",
        "units": "operations"
    },
    {
        "$schema": "//:stat",
        "description": "errors",
        "id": "//:class.app/util1//:stat.errors",
        "stability": "stable",
        "type": "counter",
        "units": "errors"
    }
]
```

Run the `soljsonfmt` tool on the `.json` file to check for JSON syntactic errors:

```
# soljsonfmt class.app.util1.json stat.util1.json
```

Run the `soljsonvalidate` tool on the `.json` file to check for JSON semantic errors:

```
# soljsonvalidate class.app.util1.json stat.util1.json
```

Use the -v option to see a list of each element and the schema against which that element was validated.

You must restart the `sstore:default` service to see the new class SSID.

```
$ sstore list //:class.app/util1
Warning (//:class.app/util1) - lookup error: no matches found
```

Restart the statistics store:

```
$ svcadm restart sstore:default
```

Ensure that `sstore:default` and other services are online:

```
$ svcs -x
```

Check whether the metadata files imported correctly:

```
$ svcs -Lx sstore
```

The log file should show no errors and should show that the start method exited with status 0.

Try again to list the new class SSID:

```
$ sstore list //:class.app/util1
IDENTIFIER
//:class.app/util1
```

Show the metadata for the class:

```
$ sstore info //:class.app/util1
 Identifier: //:class.app/util1
  stability: stable
    $schema: //:class
description: Example using count data and sstore_data_attach()
         id: app/util1
 stat-names: //:stat.reads
 stat-names: //:stat.writes
 stat-names: //:stat.errors
```

Except for the `stat-names` statistic names shown in the class information, you cannot get any information about statistics until you provide values for the statistics as shown in “Update and View Statistic Values” on page 48.

```
$ sstore list //:class.app/util1//:*
Warning (//:class.app/util1//:*) - lookup error: no matches found
$ sstore list //:class.app/util1//:stat.*
Warning (//:class.app/util1//:stat.*) - lookup error: no matches found
$ sstore list //:class.app/util1//:stat.reads
Warning (//:class.app/util1//:stat.reads) - lookup error: no matches found
```

Create an Application that Writes Statistic Values

The application in this section does nothing other than generate values for the statistics in this example. This functionality needs to be integrated into your real application.

C Version

Create the file `util1.c` with the following content to increment the counter statistics in this example. See the `libsstore`(3LIB) and `sstore_data_attach`(3SSTORE) man pages for information about the `libsstore` C interfaces and types.

```
/*
 * Example program to provide statistics values using sstore_data_attach().
 */

#include <libsstore.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>

#define NUM_STATS   3

/* libsstore handle */
sstore_handle_t hdl;

/* statistic identifiers */
char *ids[NUM_STATS] = {
    "//:class.app/util1//:stat.reads",
    "//:class.app/util1//:stat.writes",
    "//:class.app/util1//:stat.errors"
};

/* structure where values are stored */
struct mystats {
    uint64_t reads;
    uint64_t writes;
    uint64_t errors;
};
```

```
int main()
{
    int iterations = 500;
    struct mystats *stats;

    /* Allocate a libsstore handle. */
    if ((hdl = sstore_alloc()) == NULL) {
            (void) printf("Failed to allocate handle.");
            return (-1);
    }

    /*
     * These statistics already have metadata in a common location,
     * so sstore knows how to create them. sstore_data_attach() will
     * create a shared-memory region between sstore and this program.
     */
    if (sstore_data_attach(hdl, (const char **)&ids,
        (uint64_t **)&stats, NUM_STATS) != ESSTORE_OK) {
            (void) fprintf(stderr,
                "sstore_data_attach() failed because %s\n",
                sstore_err_description(hdl));
            return (-1);
    }

    /*
     * Update the values in the structure.
     * The new values will be stored when sstore reads them.
     */

    while (iterations-- > 0) {
        stats->reads += rand() % 6;
        stats->writes += rand() % 5;
        stats->errors += rand() % 2;
        sleep(1);
    }

    /*
     * Free the libsstore handle.
     * The statistics are marked as not being actively provided.
     */
    sstore_free(hdl);

    return (0);
}
```

Compile this sample application:

```
$ cc -lsstore -o util1 util1.c
```

Python Version

Create the file `util1.py` with the following content to increment the counter statistics in this example. See the `libsstore`(python) man page for information about the Python `libsstore` library, the `SStore` class, and the `data_attach()` method.

```
#!/usr/bin/python

# Example program to provide statistics values using Python data_attach().

import time
import random
from libsstore import SStore

ssids = [
    "//:class.app/util1//:stat.reads",
    "//:class.app/util1//:stat.writes",
    "//:class.app/util1//:stat.errors"
]

# Get an instance of the SStore class.
ss = SStore()

# Set up the shared memory region.
try:
    stats = ss.data_attach(ssids)
except:
    print("data_attach() failed. Reason {0}".format(
        ss.err_description))
    exit(1)

# Update statistics every second.
for i in range (500):
    stats[0] += random.randint(2,6)
    stats[1] += random.randint(1,4)
    stats[2] += random.randint(0,1)
    time.sleep(1)
```

Update and View Statistic Values

Run the application:

```
$ ./util1
```

List the new statistics. The `sstore` command lists them in alphabetical order:

```
$ sstore list //:class.app/util1//:stat.*
IDENTIFIER
//:class.app/util1//:stat.errors
//:class.app/util1//:stat.reads
//:class.app/util1//:stat.writes
```

Show metadata for the new statistics:

```
$ sstore info //:class.app/util1//:stat.*
 Identifier: //:class.app/util1//:stat.errors
    $schema: //:stat
description: errors
         id: //:class.app/util1//:stat.errors
  stability: stable
      units: errors
       type: counter

 Identifier: //:class.app/util1//:stat.reads
    $schema: //:stat
description: reads
         id: //:class.app/util1//:stat.reads
  stability: stable
      units: operations
       type: counter

 Identifier: //:class.app/util1//:stat.writes
    $schema: //:stat
description: writes
         id: //:class.app/util1//:stat.writes
  stability: stable
      units: operations
       type: counter
```

Record values. If the program has finished running, you only see the last value that was recorded in the shared memory space.

```
$ sstore capture //:class.app/util1//:stat.*
TIME                VALUE IDENTIFIER
2016-06-01T16:04:14 252 //:class.app/util1//:stat.errors
2016-06-01T16:04:14 1191 //:class.app/util1//:stat.reads
2016-06-01T16:04:14 959 //:class.app/util1//:stat.writes
2016-06-01T16:04:14 252 //:class.app/util1//:stat.errors
2016-06-01T16:04:14 1191 //:class.app/util1//:stat.reads
2016-06-01T16:04:14 959 //:class.app/util1//:stat.writes
^C
```

Re-run the program, and record the values while the program is running:

```
$ ./util1 &
$ sstore capture //:class.app/util1//:stat.*
TIME                VALUE IDENTIFIER
2016-06-01T14:36:54 2 //:class.app/util1//:stat.errors
2016-06-01T14:36:54 3 //:class.app/util1//:stat.reads
2016-06-01T14:36:54 4 //:class.app/util1//:stat.writes
2016-06-01T14:36:54 2 //:class.app/util1//:stat.errors
2016-06-01T14:36:54 3 //:class.app/util1//:stat.reads
2016-06-01T14:36:54 8 //:class.app/util1//:stat.writes
2016-06-01T14:36:55 3 //:class.app/util1//:stat.errors
2016-06-01T14:36:55 3 //:class.app/util1//:stat.reads
2016-06-01T14:36:55 12 //:class.app/util1//:stat.writes
^C
```

The `export` command can show values that were recorded in the past. The `export` command prints all the requested values for one statistic and then all the values for the next statistic:

```
$ sstore export -t 2016-06-01T16:58:20 -p 3 //:class.app/util1//:stat.*
TIME                VALUE IDENTIFIER
2016-06-01T16:58:20 244 //:class.app/util1//:stat.errors
2016-06-01T16:58:21 245 //:class.app/util1//:stat.errors
2016-06-01T16:58:22 246 //:class.app/util1//:stat.errors
2016-06-01T16:58:20 1144 //:class.app/util1//:stat.reads
2016-06-01T16:58:21 1148 //:class.app/util1//:stat.reads
2016-06-01T16:58:22 1152 //:class.app/util1//:stat.reads
2016-06-01T16:58:20 712 //:class.app/util1//:stat.writes
2016-06-01T16:58:21 715 //:class.app/util1//:stat.writes
2016-06-01T16:58:22 717 //:class.app/util1//:stat.writes
```

Create a Graph to Visualize the Statistic Values

Create the following sheet metadata file named `util1.json` in the directory `/usr/lib/webui/analytics/sheets/site/`. Each graph or visualization must be in a group, each group must be in a section, and each section must be in a sheet. Each visualization, group, section, and sheet must have a unique name: They cannot all be named “util1 statistics,” for example. The following file defines two visualizations in one group: one visualization for the read and write operations counts, and one visualization for the error count.

```
{
    "$schema": "file:///analytics-import.schema.json",
    "v1": {
        "groups": [
            {
                "description": "Reads, writes, and error counts for the util1 example",
                "uniqueName": "util1 statistics Group",
```

```
                "visualizations": [
                    "util1 operations",
                    "util1 errors"
                ]
            }
        ],
        "sections": [
            {
                "groups": [
                    "util1 statistics Group"
                ],
                "uniqueName": "util1 statistics Section"
            }
        ],
        "sheets": [
            {
                "description": "Statistics for the util1 data_attach example.",
                "sections": [
                    "util1 statistics Section"
                ],
                "tags": [
                    "data_attach",
                    "memory map"
                ],
                "uniqueName": "util1 statistics"
            }
        ],
        "visualizations": [
            {
                "description": "Count of errors from util1",
                "ssids": [
                    "//:class.app/util1//:stat.errors"
                ],
                "style": "time-series",
                "uniqueName": "util1 errors"
            },
            {
                "description": "Counts of read and write operations for util1",
                "ssids": [
                    "//:class.app/util1//:stat.reads",
                    "//:class.app/util1//:stat.writes"
                ],
                "style": "time-series",
                "uniqueName": "util1 operations"
            }
        ]
    }
}
```

Run the `soljsonfmt` tool on the `.json` file to check for JSON syntactic errors. Run the `soljsonvalidate` tool on the `.json` file to check for JSON semantic errors:

```
# soljsonfmt util1.json
# soljsonvalidate /usr/lib/webui/analytics/sheets/analytics-import.schema.json
 util1.json
```

Restart the `webui/server:default` service and ensure the service is online:

```
$ svcadm restart svc:/system/webui/server:default
$ svcs webui/server
STATE          STIME    FMRI
online         14:02:22 svc:/system/webui/server:default
```

Ensure the new sheet file was successfully read:

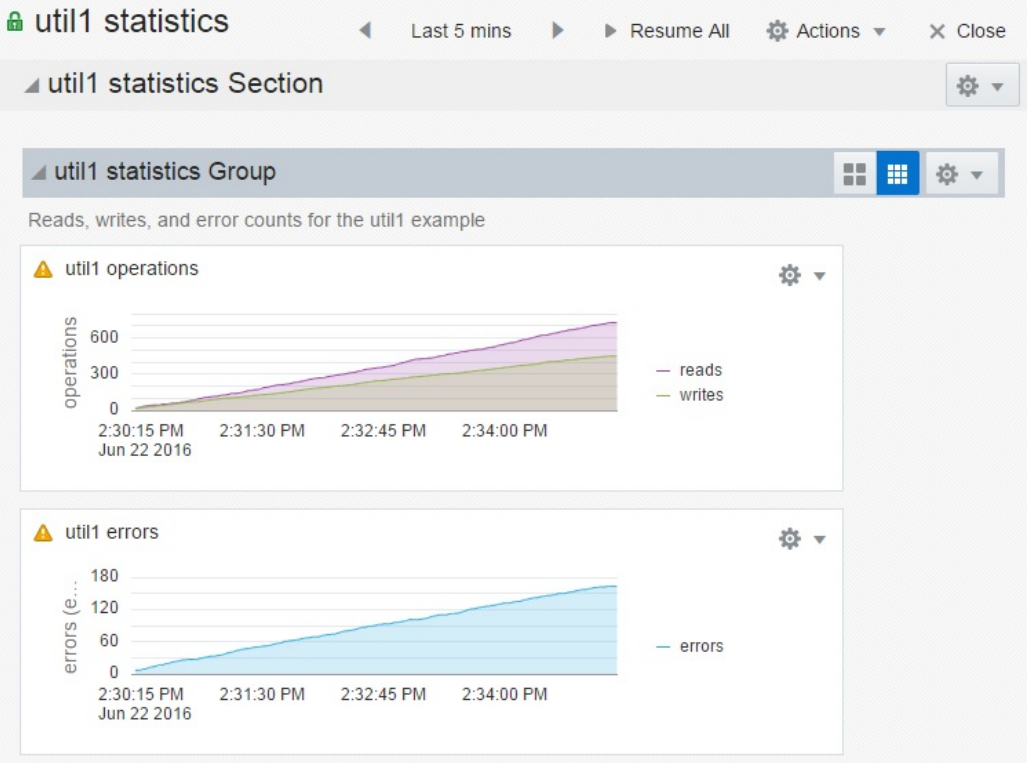
```
$ svcs -Lx webui/server
[ 2016 Jun 22 14:02:20 Executing start method ("/lib/svc/method/svc-webui-server
 start"). ]
Importing preference files
Found     : 14 files
Imported  : 1 updated/new files

Successfully imported 1 files, removed 0 files
Import succeeded
Starting Apache webserver
Apache start completed
[ 2016 Jun 22 14:02:22 Method "start" exited with status 0. ]
```

When administrators open the Sheets view in the System Web Interface, they see a new sheet named "util1 statistics". If the `util1` program is not running, the "util1 operations" and "util1 errors" visualizations show only the last values that were recorded. When the `util1` program is started, the graphs show the values updating as they are recorded.

The following figure shows the values of `uniqueName` specified in the `util1.json` sheet definition file displayed as the names of the sheet, section, group, and visualizations. The statistic labels in the legend of each visualization were specified in `description` elements in the `stat.util1.json` statistic definition file, and the units of the y-axis were specified in the `units` elements. The `reads` and `writes` statistics must be the same units to display on the same visualization.

FIGURE 1 Graphs Showing Updating Values



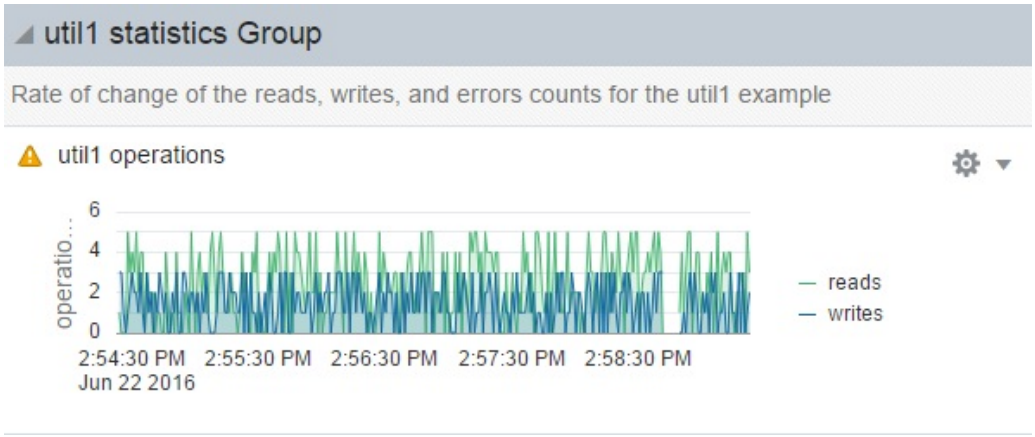
What do your users need to know about these statistics? An ever-increasing total count of operations over time might not be very useful. Edit the sheet definition file to add `//:op.rate` to each statistic as shown:

```
{
    "$schema": "file:///analytics-import.schema.json",
    "v1": {
        "groups": [
            {
                "description": "Rate of change of the reads, writes, and errors counts
 for the util1 example",
                "uniqueName": "util1 statistics Group",
                "visualizations": [
                    "util1 operations",
                    "util1 errors"
```

```
                ]
            }
        ],
        "sections": [
            {
                "groups": [
                    "util1 statistics Group"
                ],
                "uniqueName": "util1 statistics Section"
            }
        ],
        "sheets": [
            {
                "description": "Statistics for the util1 data_attach example.",
                "sections": [
                    "util1 statistics Section"
                ],
                "tags": [
                    "data_attach",
                    "memory map"
                ],
                "uniqueName": "util1 statistics"
            }
        ],
        "visualizations": [
            {
                "description": "Rate of change of errors count from util1",
                "ssids": [
                    "//:class.app/util1//:stat.errors//:op.rate"
                ],
                "style": "time-series",
                "uniqueName": "util1 errors"
            },
            {
                "description": "Rate of change of read and write operations count for
 util1",
                "ssids": [
                    "//:class.app/util1//:stat.reads//:op.rate",
                    "//:class.app/util1//:stat.writes//:op.rate"
                ],
                "style": "time-series",
                "uniqueName": "util1 operations"
            }
        ]
    }
}
```

Restart the `webui/server:default` service and ensure the service is online. The System Web Interface will close and request that you log in again. Restart the `util1` program, and now you see a graph of the rate of change of the statistics.

FIGURE 2 Graph Showing Rate of Change of Statistics



♦ ♦ ♦ CHAPTER 4

Specifying Resources

The example in this chapter builds on the example in Chapter 3, “Adding Simple Data Values to the Statistics Store”. The previous chapter showed only statistics that apply to the entire class. This chapter shows specifying resources, both statically and dynamically.

Collect Data for Statically Allocated Resources

The examples described in this section are similar to the previous example. The difference is that resources are defined for the applications in these examples. For an application, a resource typically is an instance of the application. The examples described in this section define three resources for the application: Customers, Products, and Orders. The same statistics that were associated directly with the application in the previous example are also associated with each resource in these examples.

These examples focus on the differences between these examples and the previous example. For more complete descriptions of these files and procedures, see Chapter 3, “Adding Simple Data Values to the Statistics Store”.

Add Resources to the Class Metadata

In the `/usr/lib/sstore/metadata/json/site/` directory, create the file `class.app.util2.json` with the following content to include the three static resources for the `util2` application. In this version, the description and the name of the application are updated . The `stat-names` element is the same as in the previous example. The following elements are new:

- `instance-metadata`

- `namespaces`
- `static-instances`

Note that the value of `resource-name` in the `namespaces` element matches the value of `namespace` in the `static-instances` element.

```
{
    "$schema": "//:class",
    "description": "Example util1 plus statically-allocated resources",
    "id": "app/util2",
    "instance-metadata": {
        "description": "Instances of util2",
        "stability": "stable"
    },
    "namespaces": [
        {
            "name-type": "string",
            "resource-name": "inst"
        }
    ],
    "stability": "stable",
    "stat-names": [
        "//:stat.reads",
        "//:stat.writes",
        "//:stat.errors"
    ],
    "static-instances": [
        {
            "name": "Customers",
            "namespace": "inst"
        },
        {
            "name": "Products",
            "namespace": "inst"
        },
        {
            "name": "Orders",
            "namespace": "inst"
        }
    ]
}
```

Create the file `stat.util2.json`, which is the same as the `stat.util1.json` file except that `util1` is changed to `util2` in each `id` value.

```
[
    {
        "$schema": "//:stat",
```

```
        "description": "reads",
        "id": "//:class.app/util2//:stat.reads",
        "stability": "stable",
        "type": "counter",
        "units": "operations"
    },
    {
        "$schema": "//:stat",
        "description": "writes",
        "id": "//:class.app/util2//:stat.writes",
        "stability": "stable",
        "type": "counter",
        "units": "operations"
    },
    {
        "$schema": "//:stat",
        "description": "errors",
        "id": "//:class.app/util2//:stat.errors",
        "stability": "stable",
        "type": "counter",
        "units": "errors"
    }
]
```

Restart the `sstore:default` service and view the `util2` statistic metadata.

```
$ svcadm restart sstore:default
$ sstore info //:class.app/util2
       Identifier: //:class.app/util2
       namespaces: {'0': '{'name-type': 'string', 'resource-name': 'inst'}'}
          $schema: //:class
      description: Example util1 plus statically-allocated resources
               id: app/util2
instance-metadata: {'description': 'Instances of util2', 'stability': 'stable'}
 static-instances: {'0': '{'name': 'Customers', 'namespace': 'inst'}', '1': '{'name':
 'Products', 'namespace': 'inst'}', '2': '{'name': 'Orders', 'namespace': 'inst'}'}
        stability: stable
       stat-names: //:stat.reads
       stat-names: //:stat.writes
       stat-names: //:stat.errors
```

Modify the Application to Save Statistic Values for Each Resource

The `util2.c` file is the same as the `util1.c` file except for the description at the top of the file and the following changes:

- Change `NUM_STATS` from 3 to 9.
- The `ids` array has two changes:
  - Change `util1` to `util2` in the class name of each SSID.
  - Add the three resource instances, each with all three statistics.
- The `while()` loop updates nine statistics instead of three.

Note that the `stats` structure is unchanged and the `sstore_data_attach()` call is unchanged.

```
/*
 * Sample program to use sstore_data_attach() to provide values for
 * statistics of statically allocated resources.
 */

#include <libsstore.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>

#define NUM_STATS   9

/* libsstore handle */
sstore_handle_t hdl;

/* statistic identifiers */
char *ids[NUM_STATS] = {
    "//:class.app/util2//:res.inst/Customers//:stat.reads",
    "//:class.app/util2//:res.inst/Customers//:stat.writes",
    "//:class.app/util2//:res.inst/Customers//:stat.errors",
    "//:class.app/util2//:res.inst/Products//:stat.reads",
    "//:class.app/util2//:res.inst/Products//:stat.writes",
    "//:class.app/util2//:res.inst/Products//:stat.errors",
    "//:class.app/util2//:res.inst/Orders//:stat.reads",
    "//:class.app/util2//:res.inst/Orders//:stat.writes",
    "//:class.app/util2//:res.inst/Orders//:stat.errors"
};

/* structure where values are stored */
struct mystats {
    uint64_t reads;
    uint64_t writes;
    uint64_t errors;
};

int main()
```

```
{
    int iterations = 500;
    struct mystats *stats;

    /* Allocate a libsstore handle. */
    if ((hdl = sstore_alloc()) == NULL) {
            (void) printf("Failed to allocate handle.");
            return (-1);
    }

    /*
     * These statistics already have metadata in a common location,
     * so sstore knows how to create them. sstore_data_attach() will
     * create a shared-memory region between sstore and this program.
     */
    if (sstore_data_attach(hdl, (const char **)&ids,
        (uint64_t **)&stats, NUM_STATS) != ESSTORE_OK) {
            (void) fprintf(stderr,
                "sstore_data_attach() failed because %s\n",
                sstore_err_description(hdl));
            return (-1);
    }

    /*
     * Update the values in the structure.
     * The new values will be stored when sstore reads them.
     */

    while (iterations-- > 0) {
        stats[0].reads += rand() % 6;
        stats[0].writes += rand() % 4;
        stats[0].errors += rand() % 2;
        stats[1].reads += rand() % 8;
        stats[1].writes += rand() % 4;
        stats[1].errors += rand() % 2;
        stats[2].reads += rand() % 9;
        stats[2].writes += rand() % 5;
        stats[2].errors += rand() % 2;
        sleep(1);
    }

    /*
     * Free the libsstore handle.
     * The statistics are marked as not being actively provided.
     */
    sstore_free(hdl);

    return (0);
```

```
}
```

View Statistic Values for Statically Allocated Resources

Compile and run the `util2` application.

```
# cc -lsstore -o util2 util2.c
# ./util2 &
# sstore capture //:class.app/util2//:*//:*
TIME                VALUE IDENTIFIER
2016-06-12T04:34:28 2 //:class.app/util2//:res.inst/Customers//:stat.errors
2016-06-12T04:34:28 5 //:class.app/util2//:res.inst/Customers//:stat.reads
2016-06-12T04:34:28 5 //:class.app/util2//:res.inst/Customers//:stat.writes
2016-06-12T04:34:28 2 //:class.app/util2//:res.inst/Orders//:stat.errors
2016-06-12T04:34:28 8 //:class.app/util2//:res.inst/Orders//:stat.reads
2016-06-12T04:34:28 12 //:class.app/util2//:res.inst/Orders//:stat.writes
2016-06-12T04:34:28 3 //:class.app/util2//:res.inst/Products//:stat.errors
2016-06-12T04:34:28 9 //:class.app/util2//:res.inst/Products//:stat.reads
2016-06-12T04:34:28 6 //:class.app/util2//:res.inst/Products//:stat.writes
```

Create a Graph to Visualize Resource Statistics

Create the following sheet metadata file named `util2.json` in the directory `/usr/lib/webui/analytics/sheets/site/`:

```
{
    "$schema": "file:///analytics-import.schema.json",
    "v1": {
        "groups": [
            {
                "description": "Rate of change of the reads, writes, and errors counts
 for the util2 example",
                "uniqueName": "util2 statistics Group",
                "visualizations": [
                    "util2 writes",
                    "util2 Customers"
                ]
            }
        ],
        "sections": [
            {
                "groups": [
```

```
                    "util2 statistics Group"
                ],
                "uniqueName": "util2 statistics Section"
            }
        ],
        "sheets": [
            {
                "description": "Resource statistics for the util2 example.",
                "sections": [
                    "util2 statistics Section"
                ],
                "tags": [
                    "data_attach",
                    "static resources"
                ],
                "uniqueName": "util2 statistics"
            }
        ],
        "visualizations": [
            {
                "description": "Rate of change of writes count for all resources",
                "ssids": [
                    "//:class.app/util2//:res.*//:stat.writes//:op.rate"
                ],
                "style": "time-series",
                "uniqueName": "util2 writes"
            },
            {
                "description": "Rate of change of reads, writes, and errors count for
 Customers",
                "ssids": [
                    "//:class.app/util2//:res.inst/Customers//:stat.*//:op.rate"
                ],
                "style": "time-series",
                "uniqueName": "util2 Customers"
            }
        ]
    }
}
```

This file defines a sheet with two visualizations. One visualization shows the rate of change of write operation counts for each resource. The second visualization shows the read, write, and error count rates of change for one resource.

Restart the `webui/server:default` service and ensure the service is online and the visualization definitions were successfully read:

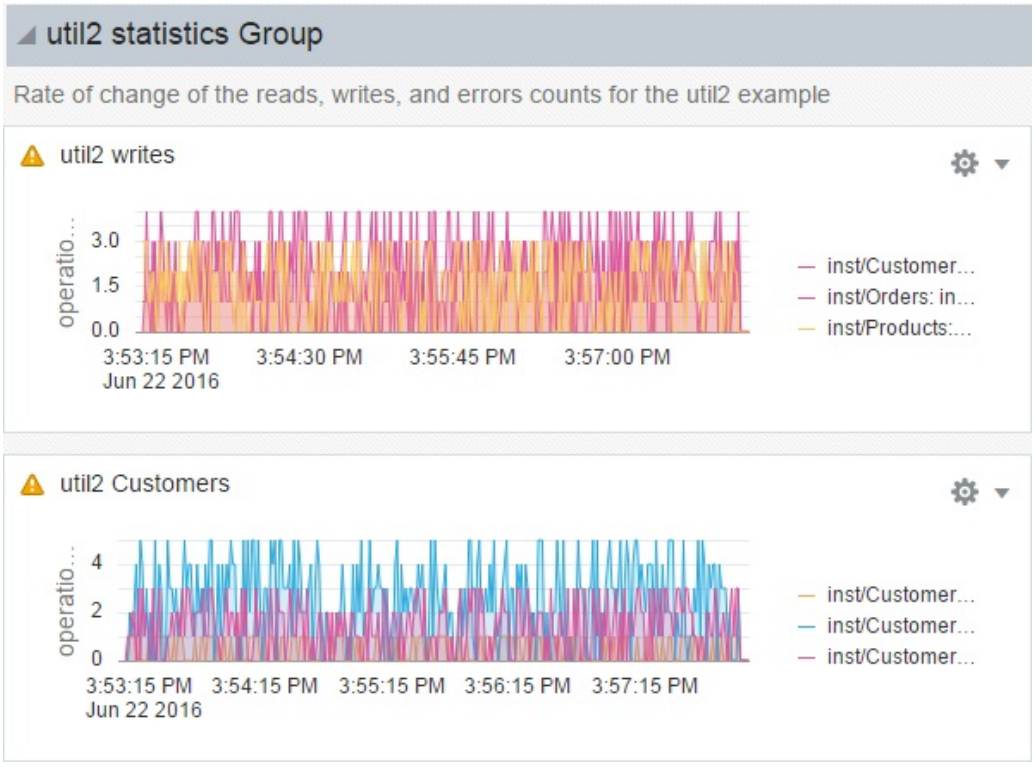
```
$ svcadm restart svc:/system/webui/server:default
```

```
$ svcs webui/server
STATE          STIME    FMRI
online         15:47:13 svc:/system/webui/server:default
$ svcs -Lx webui/server
```

When administrators open the Sheets view in the System Web Interface, they see a new sheet named “util2 statistics,” shown in the following figure. The visualization named “util2 writes” shows the rate of change of the writes count for the Customers resource, the Orders resource, and the Products resource. The visualization named “util2 Customers” shows the rate of change of the reads, writes, and errors counts for the Customers resource.

You can create partitions and use mapping to enable administrators to choose how they want to view the data. Instead of providing multiple visualizations with different views of the data, you can provide a single visualization that enables administrators to display the data in different ways. See Chapter 5, “Separating Data Into Partitions” for more information.

FIGURE 3 Graph Showing Statistics for Resources



Collect Data for Dynamically Allocated Resources

The example described in this section is similar to the previous example. The difference is that in this example you do not know ahead of time how many or which application instances will be configured on the system. The resources in this example are dynamically allocated in the application using the `sstore_resource_add()` interface.

The `sstore_resource_add()` interface protects against multiple applications providing data for the same resources as described in the following comparison:

- To allocate resources statically, you must assume the `root` role; the metadata files and directories are owned by `root`. If you add conflicting metadata, you will get errors when you restart the `sstore` service.
- To allocate resources dynamically, you must call `sstore_resource_add()` to add the resource metadata to the statistics repository. If you try to add metadata that is already in the statistics store, you will get an error return.

Note that a single class can have both statically-allocated resources and dynamically-allocated resources. For example, the `svc` class has one static resource: the default restarter service `system/svc/restarter:default`. Other services are added dynamically as the services are installed on the system.

Modify the Metadata to Omit Resource Names

The following table provides an overview of elements to define in your class metadata. Some optional elements, such as `events`, are not shown. This table is focused on resources. In the "No Resources" column, all statistics are class statistics.

TABLE 3 Class Metadata Elements

No Resources	Statically Defined Resources	Dynamically Defined Resources
■ `schema` ■ `description` ■ `id` ■ `stability` ■ `stat-names`	■ `schema` ■ `description` ■ `id` ■ `instance-metadata` ■ `namespaces` ■ `stability` ■ `stat-names` ■ `static-instances`	■ `schema` ■ `description` ■ `id` ■ `namespaces` ■ `stability` ■ `stat-names`

In the `/usr/lib/sstore/metadata/json/site/` directory, create the file `class.app.util3.json` with the following content. If you copy the file `class.app.util2.json`, the primary changes you need to make are:

- Delete the `instance-metadata` element.
- Delete the `static-instances` element.

The `instance-metadata` and `static-instances` elements are not used for dynamically allocated resources.

The `stat-names` element is the same as in the previous two examples. The `namespaces` element is still needed to define the type of the resources.

You might also want to update the description of the application and the name of the application in the `id`.

```
{
    "$schema": "//:class",
    "description": "Example util1 plus dynamically-allocated resources",
    "id": "app/util3",
    "namespaces": [
        {
            "name-type": "string",
            "resource-name": "inst"
        }
    ],
    "stability": "stable",
    "stat-names": [
        "//:stat.reads",
        "//:stat.writes",
        "//:stat.errors"
    ]
}
```

Create the file `stat.util3.json`, which is the same as the `stat.util2.json` file except that `util2` is changed to `util3` in each `id` value.

```
[
    {
        "$schema": "//:stat",
        "description": "reads",
        "id": "//:class.app/util3//:stat.reads",
        "stability": "stable",
        "type": "counter",
        "units": "operations"
    },
    {
```

```
        "$schema": "//:stat",
        "description": "writes",
        "id": "//:class.app/util3//:stat.writes",
        "stability": "stable",
        "type": "counter",
        "units": "operations"
    },
    {
        "$schema": "//:stat",
        "description": "errors",
        "id": "//:class.app/util3//:stat.errors",
        "stability": "stable",
        "type": "counter",
        "units": "errors"
    }
]
```

Restart the `sstore:default` service to add the new metadata to the statistics repository.

```
$ svcadm restart sstore:default
```

View the `util3` statistic metadata.

```
$ sstore info //:class.app/util3
 Identifier: //:class.app/util3
 namespaces: {'0': '{'name-type': 'string', 'resource-name': 'inst'}'}
    $schema: //:class
description: Example util1 plus dynamically-allocated resources
         id: app/util3
  stability: stable
 stat-names: //:stat.reads
 stat-names: //:stat.writes
 stat-names: //:stat.errors
```

Modify the Application to Create Resources Dynamically

The `util3` version of this example application calls `sstore_data_attach()` in the same way as in the `util2` version and updates the statistic values in the same way. The `stats` structure is the same.

The `util3` version of the `ids` array is the same as the `util2` version except that the SSIDs are not statically listed in the application. In the `util3` version, the `ids` array is built dynamically as resources are added.

How to Add Resources Dynamically

Use this procedure when your application detects that a new resource has come online.

1. **Store an SSID for the resource in an array.**

   This resource SSID has the following form:

   `//:class.`*class*`//:res.`*resource*

   For example, store the following SSID in an array named `res_ids`.

   ```
   //:class.app/util3//:res.inst/Customers
   ```

   This array of resource SSIDs is an argument to `sstore_resource_add()`.

2. **Call `sstore_resource_add()` to add the resource metadata to the statistics repository.**

   For example, if you had added Customers, Products, and Orders resources to the `res_ids` array, you could make the following call:

   ```
   sstore_resource_add(hdl, (const char **)res_ids, 3)
   ```

   An optional fourth argument enables you to include additional metadata for the resource. See the `sstore_resource_add`(3SSTORE) man page for details.

3. **Build the statistics array.**

   Use the `res_ids` resource SSID array and the `stats` structure to dynamically create the `ids` statistics array. As in previous versions of this example, the `ids` statistics array is the array that you pass to `sstore_data_attach()` and use for updating the values of the statistics.

♦ ♦ ♦ CHAPTER 5

Separating Data Into Partitions

The example in this chapter builds on the examples in the previous chapters. This chapter shows separating the statistic data into partitions. Partitioning statistic data enables you to see either a total value or a breakdown (parts) of a total value. For example, one administrator might need to know only that an error occurred, while another administrator needs more information about where errors occurred.

Add Partition Metadata

The class metadata file does not need to change. Use the `class.app.util2.json` file shown in "Add Resources to the Class Metadata" on page 57.

In this example, the statistics metadata file is modified to add a statistic that will be used in creating partitions.

Modify the Statistics Metadata File

The statistics metadata file is the same as the `stat.util2.json` file shown in "Add Resources to the Class Metadata" on page 57 except that you need to add an element for the `activity` statistic that will be used to partition and `reads` and `writes` statistics. Modify the `stat.util2.json` file to have the following content:

```
[
    {
        "$schema": "//:stat",
```

```
        "description": "reads",
        "id": "//:class.app/util2//:stat.reads",
        "stability": "stable",
        "type": "counter",
        "units": "operations"
    },
    {
        "$schema": "//:stat",
        "description": "writes",
        "id": "//:class.app/util2//:stat.writes",
        "stability": "stable",
        "type": "counter",
        "units": "operations"
    },
    {
        "$schema": "//:stat",
        "description": "errors",
        "id": "//:class.app/util2//:stat.errors",
        "stability": "stable",
        "type": "counter",
        "units": "errors"
    },
    {
        "$schema": "//:stat",
        "description": "reads or writes",
        "id": "//:class.app/util2//:stat.activity",
        "stability": "stable",
        "type": "counter",
        "units": "operations"
    }
]
```

Create a Statistic Mapping File

Create a new `stat-mapping` metadata file with the following content in `stat-mapping.util2.json` in the `/usr/lib/sstore/metadata/json/site/` directory:

```
[
    {
        "$schema": "//:stat-mapping",
        "description": "map util2 instance errors to aggregate class errors",
        "id": "//:class.app/util2//:stat-mapping.errors",
        "instance-metadata": {
            "partitions": [
                "inst"
            ]
```

```
        },
        "transforms": [
            {
                "match": "//:class.app/util2//:res.inst/(Customers|Products|Orders)//:
stat.errors$",
                "replace": "//:class.app/util2//:stat.errors//:part.inst(\\1)"
            }
        ]
    },
    {
        "$schema": "//:stat-mapping",
        "description": "map util2 instance reads and writes to aggregate class
 activity",
        "id": "//:class.app/util2//:stat-mapping.activity",
        "instance-metadata": {
            "partitions": [
                "inst",
                "type"
            ]
        },
        "transforms": [
            {
                "match": "//:class.app/util2//:res.inst/(Customers|Products|Orders)//:
stat.(reads|writes)$",
                "replace": "//:class.app/util2//:stat.activity//:part.inst(\\1)//:part.
type(\\2)"
            }
        ]
    }
]
```

In the first stanza, the `errors` statistic for each instance (statically allocated `Customers`, `Products`, and `Orders`) is mapped to a class-level `errors` statistic. The `partitions` element indicates that the statistic is partitioned by resource (`inst`). The `(\\1)` expression in the `replace` value matches the `(Customers|Products|Orders)` expression in the `match` value. The `//:class.app/util2//:res.inst/Customers//:stat.errors`, `//:class.app/util2//:res.inst/Products//:stat.errors`, and `//:class.app/util2//:res.inst/Orders//:stat.errors` values are aggregated in the `//:class.app/util2//:stat.errors` statistic. The split between resource instances is shown by the `//:class.app/util2//:stat.errors//:part.inst` SSID.

In the second stanza, the `reads` and `writes` statistics for each instance are mapped to the new `activity` statistic that you added to the `stat.util2.json` file. The `(\\2)` expression in the `replace` value matches the `(reads|writes)` expression in the `match` value. Total reads and writes from all three instances are aggregated in the `//:class.app/util2//:stat.activity` statistic. Splits between instances are shown by the `//:class.app/util2//:stat.activity//:`

`part.inst` SSID. Splits between statistics are shown by the `//:class.app/util2//:stat.activity//:part.type` SSID.

View Partitioned Statistic Values

The following output shows the values of all resource instance statistics and partitioned statistics:

```
$ ./util2 &
$ sstore capture //:class.app/util2//:*//:*
TIME                VALUE IDENTIFIER
2016-06-12T05:19:54 2 //:class.app/util2//:res.inst/Customers//:stat.errors
2016-06-12T05:19:54 10 //:class.app/util2//:res.inst/Customers//:stat.reads
2016-06-12T05:19:54 7 //:class.app/util2//:res.inst/Customers//:stat.writes
2016-06-12T05:19:54 3 //:class.app/util2//:res.inst/Orders//:stat.errors
2016-06-12T05:19:54 15 //:class.app/util2//:res.inst/Orders//:stat.reads
2016-06-12T05:19:54 13 //:class.app/util2//:res.inst/Orders//:stat.writes
2016-06-12T05:19:54 3 //:class.app/util2//:res.inst/Products//:stat.errors
2016-06-12T05:19:54 13 //:class.app/util2//:res.inst/Products//:stat.reads
2016-06-12T05:19:54 7 //:class.app/util2//:res.inst/Products//:stat.writes
2016-06-12T05:19:54  //:class.app/util2//:stat.activity//:part.inst
                    Customers: 17.0
                    Orders: 28.0
                    Products: 20.0
2016-06-12T05:19:54  //:class.app/util2//:stat.activity//:part.type
                    writes: 27.0
                    reads: 38.0
2016-06-12T05:19:54  //:class.app/util2//:stat.errors//:part.inst
                    Customers: 2.0
                    Orders: 3.0
                    Products: 3.0
^C
```

Partitioned values are shown as real numbers.

The following output shows that the total of all reads and writes (17 + 28 + 20 or 27 +38 shown in the preceding output) is stored in the `//:class.app/util2//:stat.activity` statistic, and the total errors from all instances is stored in the `//:class.app/util2//:stat.errors` statistic:

```
$ sstore export -t 2016-06-12T05:19:53 -p 1 //:class.app/util2//:stat.*
TIME                VALUE IDENTIFIER
2016-06-12T05:19:54 65.0 //:class.app/util2//:stat.activity
2016-06-12T05:19:54 8.0 //:class.app/util2//:stat.errors
```

Create a Graph to Visualize Partitioned Statistic Values

Create the following sheet metadata file in `/usr/lib/webui/analytics/sheets/site/` and then restart the `webui/server:default` service:

```
{
    "$schema": "file:///analytics-import.schema.json",
    "v1": {
        "groups": [
            {
                "description": "util2 reads, writes, and errors partitioned by resource
 or operation",
                "uniqueName": "util2 partitioned statistics Group",
                "visualizations": [
                    "util2 operations summed",
                    "util2 operations not summed",
                    "util2 errors"
                ]
            }
        ],
        "sections": [
            {
                "groups": [
                    "util2 partitioned statistics Group"
                ],
                "uniqueName": "util2 partitioned statistics Section"
            }
        ],
        "sheets": [
            {
                "description": "Partitioned statistics",
                "sections": [
                    "util2 partitioned statistics Section"
                ],
                "tags": [
                    "data_attach",
                    "partitions"
                ],
                "uniqueName": "util2 partitioned statistics"
            }
        ],
        "visualizations": [
            {
                "description": "Sum of reads and writes for each resource",
                "ssids": [
                    "//:class.app/util2//:stat.activity"
```

```
                ],
                "style": "time-series",
                "uniqueName": "util2 operations summed"
            },
            {
                "description": "reads and writes for each resource",
                "ssids": [
                    "//:class.app/util2//:stat.activity//:part.type(reads,writes,
sum=false)"
                ],
                "style": "time-series",
                "uniqueName": "util2 operations not summed"
            },
            {
                "description": "util2 error counts",
                "ssids": [
                    "//:class.app/util2//:stat.errors"
                ],
                "style": "time-series",
                "uniqueName": "util2 errors"
            }
        ]
    }
}
```

The sheet defined in this file includes two visualizations to display values of the `activity` statistic so that you can compare the results. You probably will define a single visualization for each statistic that you think will be most useful to most administrators. Administrators can use the System Web Interface to change the visualization if necessary.

All three visualizations defined in this metadata file have an `inst` choice on the partitions list. The visualization for `//:class.app/util2//:stat.activity` also has a `type` choice. This information comes from the mapping file.

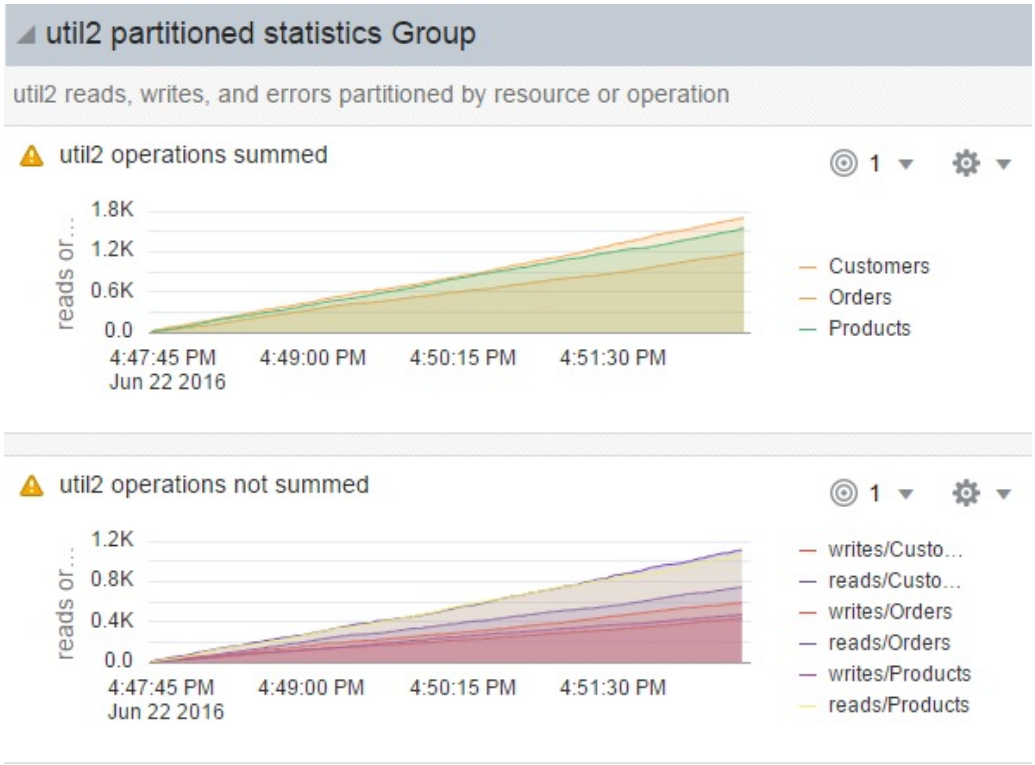
In the first `activity` visualization ("operations summed"), if no partition is selected, the graph shows a single line labeled "reads or writes" that represents the sum of all reads and writes from any resource.

In the second `activity` visualization ("operations not summed"), if no partition is selected, the graph shows two lines. The line labeled "reads" is the sum of all reads performed by any resource; the line labeled "writes" is the sum of all writes performed by any resource. The first `activity` visualization shows this same display if the user selects `type` from the partitions list.

The following figure shows both `activity` visualizations when `inst` is selected from the partitions list. In the first visualization, the three lines represent the sum of all reads and writes from each resource. In the second visualization, the six lines represent the number of reads from each resource and the number of writes from each resource. Looking back at the sheet metadata

file, the reads and writes are summed by default. Specifying `sum=false` effectively resulted in partitioning by both resource and data type at the same time.

FIGURE 4 Graph Showing Statistics Partitioned by Resource



♦ ♦ ♦ CHAPTER 6

Adding Any Type of Data to the Statistics Store

This chapter shows an `sstore_data_update()` example for updating statistic data values. Instead of using a shared memory area as `sstore_data_attach()` does, `sstore_data_update()` adds statistic values directly to the statistics store. The statistic values can be any data type.

Create the Class and Statistic Definition Files

In the `/usr/lib/sstore/metadata/json/site/` directory, create the file `class.app.example.json` with the following content to define the class for the `example` application.

```
{
    "$schema": "//:class",
    "description": "example of sstore_data_update()",
    "id": "app/example",
    "stability": "stable",
    "stat-names": [
        "//:stat.one",
        "//:stat.two"
    ]
}
```

To define the statistics for this example, create the file `stat.example.json` with the following content in the `/usr/lib/sstore/metadata/json/site/` directory:

```
[
    {
        "$schema": "//:stat",
        "description": "example stat one",
        "id": "//:class.app/example//:stat.one",
        "stability": "stable",
        "type": "counter",
        "units": "calls"
```

```
    },
    {
        "$schema": "//:stat",
        "description": "example stat two",
        "id": "//:class.app/example//:stat.two",
        "stability": "stable",
        "type": "counter",
        "units": "calls"
    }
]
```

Run the `soljsonfmt` tool on the `.json` file to check for JSON syntactic errors. Run the `soljsonvalidate` tool on the `.json` file to check for JSON semantic errors:

```
# soljsonvalidate class.app.example.json stat.example.json
# soljsonfmt class.app.example.json stat.example.json
```

You must restart the `sstore:default` service to see the new class SSID.

```
$ svcadm restart sstore:default
```

Ensure that `sstore:default` and other services are online:

```
$ svcs -x
```

Check whether the metadata files imported correctly:

```
$ tail `svcs -L sstore`
```

The log file should show no errors and should show that the start method exited with status 0.

List the new class SSID:

```
$ sstore list //:class.app/example
IDENTIFIER
//:class.app/example
```

Show the metadata for the class:

```
$ sstore info //:class.app/example
 Identifier: //:class.app/example
    $schema: //:class
         id: app/example
description: example of sstore_data_update()
 stat-names: //:stat.one
 stat-names: //:stat.two
  stability: stable
```

Create an Application that Updates Statistic Values

Create the file `data_update.c` with the following content to update the statistic values in this example. Differences between this `sstore_data_update()` program and the `sstore_data_attach()` program described in "Create an Application that Writes Statistic Values" on page 46 include the following:

- This `sstore_data_update()` program has no `mystats` structure. The `mystats` structure is used for the memory map in the `sstore_data_attach()` program.
- The handle to the statistics store is declared but no memory is allocated.

```
/*
 * Example program data_update.c:
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <libsstore.h>

#define NUM_STATS   2

const char *ssids[NUM_STATS] = {
    "//:class.app/example//:stat.one",
    "//:class.app/example//:stat.two"
};

int main()
{
    sstore_handle_t hdl;
    sstore_value_t vals[NUM_STATS] = {0};
    int i, j;

    if ((hdl = sstore_alloc()) == NULL) {
        fprintf(stderr, "Failed to alloc libsstore handle\n");
        return (-1);
    }

    /* Alloc the sstore_value_t's */
    for (i = 0; i < NUM_STATS; i++) {
        if ((vals[i] = sstore_value_alloc()) == NULL) {
            fprintf(stderr, "Failed to alloc sstore_value_t\n");
            goto end;
        }

        vals[i]->sv_type = SSTORE_VALUE_NUMBER;
    }
```

```
    /* Update the stats every second */
    for (i = 0;; i++) {
        char *id, *desc;

        for (j = 0; j < NUM_STATS; j++) {
            vals[j]->sv_value.num += j + 1;
        }

        if (sstore_data_update(hdl, ssids, vals,
            NUM_STATS) != ESSTORE_OK) {
                fprintf(stderr, "Failed to update stats. "
                    "Reason: %s\n", sstore_err_description(hdl));
                break;
        }

        /* Check warnings */
        while (sstore_warning_next(hdl, &id,
            &desc) != SS_WARN_OK) {
                fprintf(stderr, "failed to update stat for %s "
                    "because %s\n", id, desc);
        }

        sleep(1);
    }

end:
    for (i = 0; i < NUM_STATS; i++) {
        sstore_value_free(vals[i]);
    }

    sstore_free(hdl);
    return (0);
}
```

Compile the application:

```
$ cc -lsstore -o data_update data_update.c
```

Record Statistic Values

List the new statistics:

```
$ sstore list //:class.app/example//:stat.*
IDENTIFIER
//:class.app/example//:stat.one
```

```
//:class.app/example//:stat.two
```

Show information about the new statistics:

```
$ sstore info //:class.app/example//:stat.*
 Identifier: //:class.app/example//:stat.one
    $schema: //:stat
description: example stat one
         id: //:class.app/example//:stat.one
  stability: stable
       type: counter
      units: calls

 Identifier: //:class.app/example//:stat.two
    $schema: //:stat
description: example stat two
         id: //:class.app/example//:stat.two
  stability: stable
       type: counter
      units: calls
```

Run the `data_update` application and record the statistic values:

```
$ sstore capture //:class.app/example//:stat.*
TIME                VALUE IDENTIFIER
2016-05-25T23:21:02 2 //:class.app/example//:stat.one
2016-05-25T23:21:02 4 //:class.app/example//:stat.two
2016-05-25T23:21:03 3 //:class.app/example//:stat.one
2016-05-25T23:21:03 6 //:class.app/example//:stat.two
2016-05-25T23:21:04 4 //:class.app/example//:stat.one
2016-05-25T23:21:04 8 //:class.app/example//:stat.two
...
```

Index

R

S

T

U

V