MEASURE FACTORY OVERVIEW

A Diver® Platform Add-On

Abstract

Measure Factory is an automated rules engine. It allows you to define custom rules to apply to your data sets to compute your key performance measures. This overview describes the factory build process.

Dimensional Insight, Inc.

www.dimins.com

Dimensional INSIGHT

Contents

Measure Factory Overview

Introduction

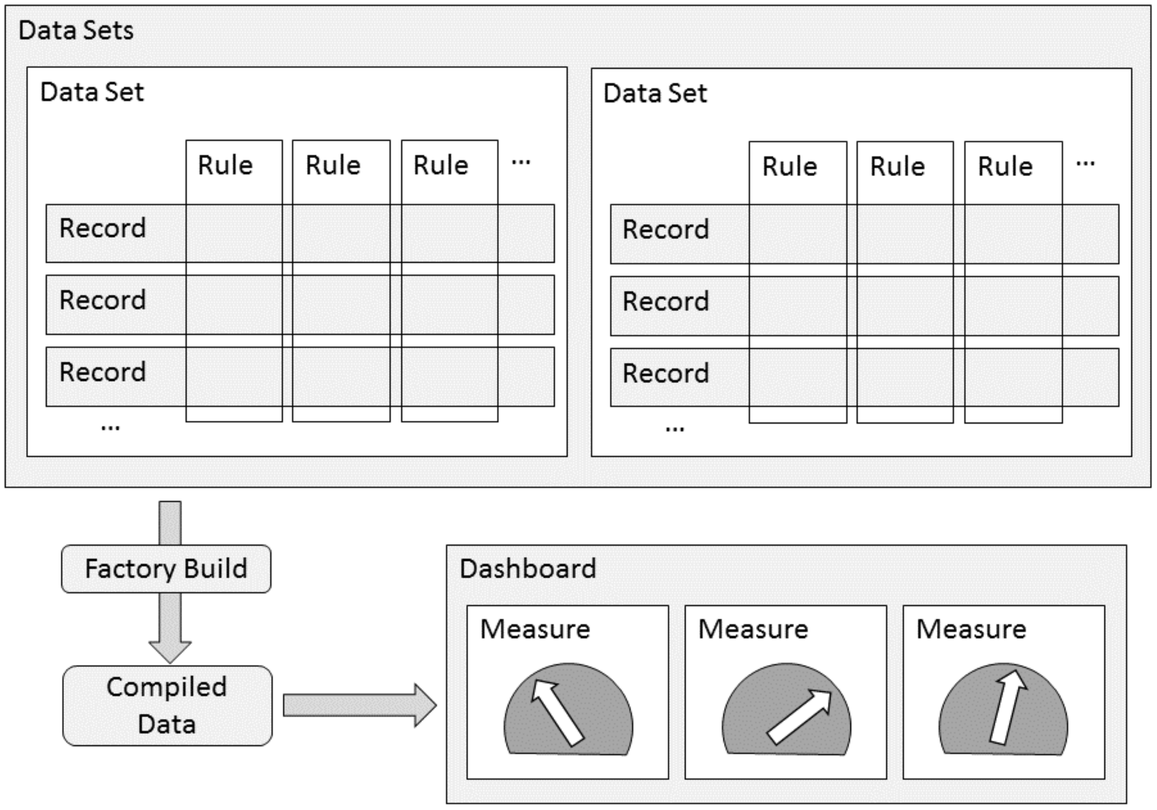
Measure Factory is an automated rules engine. It allows you to define custom rules to apply to your data sets to compute your key performance measures. Measure Factory relies on the Spectre data engine and its columnar data structure known as a cBase, as well as the Diver Platform ETL tool, Data Integrator.

This document provides an overview of how the Measure Factory works. It explains each of the rule types and describes the data flow, from the source tables to the generated cBases and cPlans. It also explains how the processed results are used to render dashboards.

DATA SETS, RULES, MEASURES, and VIEWS

Key elements of a Measure Factory are the data sets, the defined rules, the measures that are calculated, and views or slices of the final data used when displaying dashboards.

Figure 1 - Overview



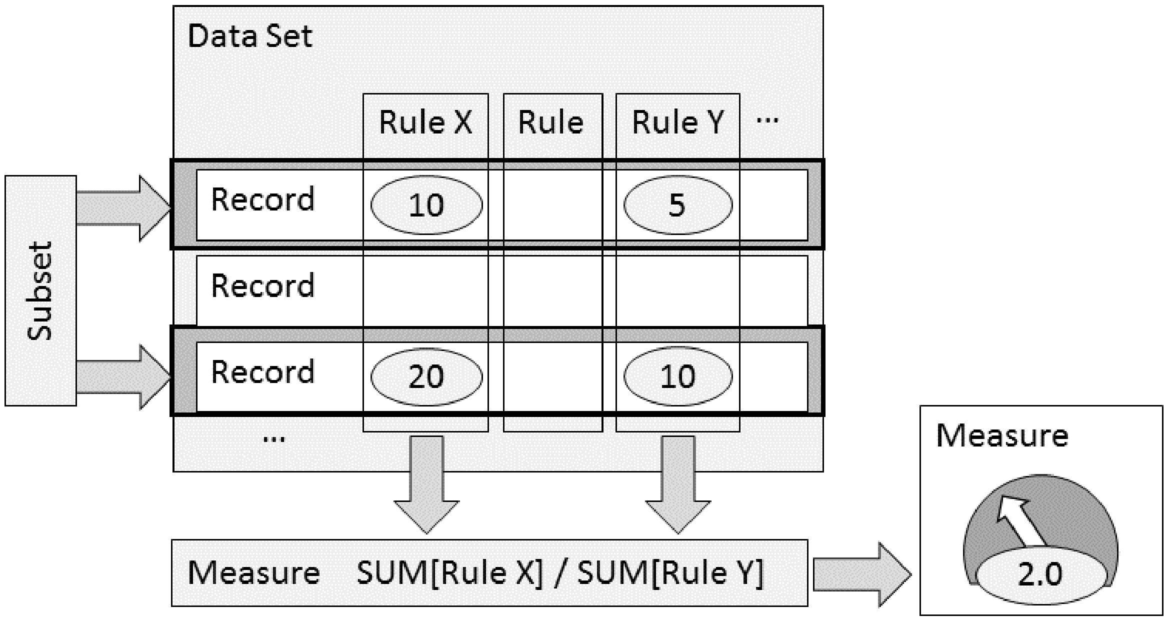
A data set is a collection of records, each of which usually describes a particular kind of object, event, or relationship. For instance, the Accounts data set has one record for each account.

Each record in a data set has a number of facts, or individual pieces of information, associated with it. In the Measure Factory a particular kind of fact shared by all the records of a data set is called a *rule*. For instance, the Accounts data set may have rules such as Account ID and Admit Date.

The main purpose of the Measure Factory is to add new rules to a collection of data sets, and to make it easier to create and manage a large number of rules. The Measure Factory automates the processing of the rules, so the people working with the factory can focus on the correctness of the rules, without needing to worry about the implementation of those rules in terms of the data flow.

The factory output is a set of generated tables which support the data displays. Typically, the data is displayed in summary form, saying for instance something like *the number of Accounts with Admit Dates occurring this month is 286*. The data can be summarized in many ways, and each kind of summarization is called a *measure*. A measure is typically the result of applying some mathematical expression to a specific set of rule values from a collection of records in one or more data sets.

Figure 2 - Measure



Measures have associated data which helps to support information rich displays, such as a description, a flag indicating whether the measure is *better* when it goes up or down, and a set of preferred columns to see when analyzing the measure.

Measures may also have a *view*, which is an abstraction over the rules available in a data set. The view assigns specific rules to abstract rule concepts. For instance, the abstract concept of Date may be specifically assigned to Admit Date for the Admissions measure, but it may be assigned to Discharge Date for the Discharges measure. This allows the dashboard to show the Admissions and Discharges measures together over a general time range (such as year-to-date), even though the two measures have a different concrete notion of which date rule is relevant to them.

RULE TYPES

Measure Factory operates on data according to user define rules. There are several rule types available.

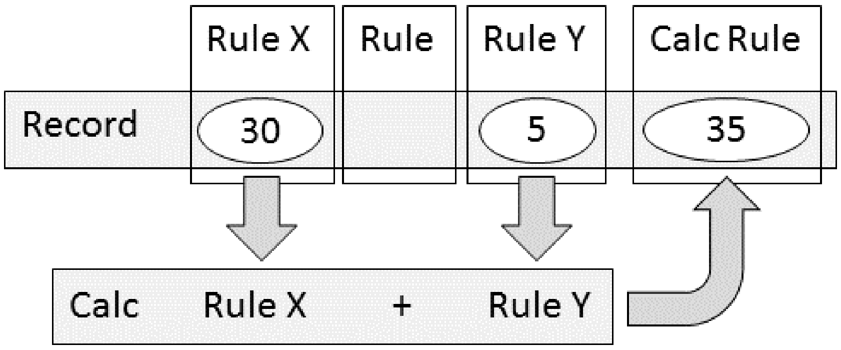
Source Rules

Each data set begins by loading a cBase from disk. That cBase is usually produced by extracting data from one or more external data stores or systems. The columns in the cBase are automatically made available as rules in the data set. These are called *source rules*. There are other rule types, for rules that are added during the factory build process.

Calc Rules

A calc rule adds information to each record of a data set by applying a mathematical expression to a subset of other rules in the same data set.

Figure 3 - Calc Rule



For instance, a rule "Admission" might be defined by the expression:

```
value("Patient Type") = "Inpatient" and value("Admit Date") != null
```

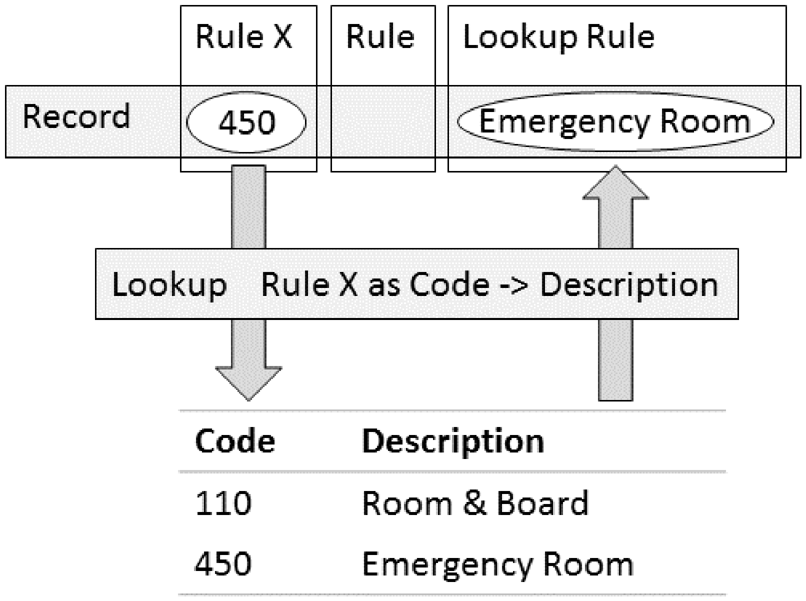
The script snippet to define such a rule would simply be:

```
calc-rule "Admission" `value("Patient Type") = "Inpatient" and
value("Admit Date") != null`
```

Lookup Rules

A lookup rule gets information from a lookup file, by matching certain values from each record with values on a particular row of the lookup file.

Figure 4 - Lookup Rule



A common use of a lookup rule is to convert codes into text descriptions. For instance, a lookup rule could convert Revenue Code to Revenue Description. It would look in the Revenue lookup file to find the Code for each record, and then copy the corresponding Description into the new rule in the data set. The script snippet to define such a rule might be:

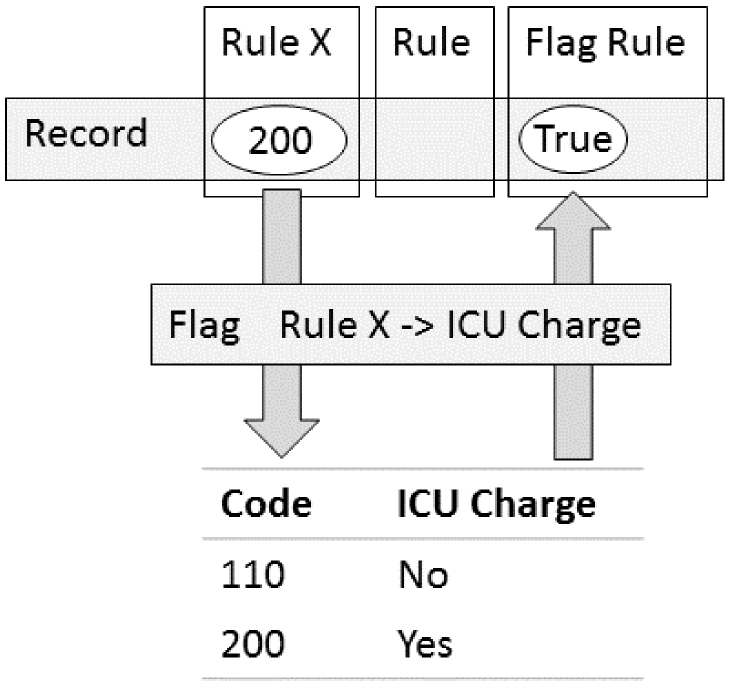
```
lookup "Revenue Descriptions" {
  date rule="Posting Date"
  key "Revenue Code"
  lookup-rule "Revenue Description"
}
```

In some cases, a lookup rule could be implemented as a calc rule, but lookups have advantages over calc rules. Lookup rules are easier to maintain: you can modify the lookup values by editing the lookup table directly, and the lookup table can be generated by an external process. Lookup rules can also store the *effective date range* for certain mapped values. That means that if the mapping between the *code* and *description* changes over time, then the lookup table can reflect that change and return the mapped value appropriate to the time associated with the record. For example, if Revenue Code *123* meant *Emergency Room - Other* for transactions that occurred before Jan 1, 2015, but it meant *Urgent Care* for transactions on or after that date, then the lookup table can be set up to ensure that transactions display the description corresponding to their effective date.

Flag Rules

A flag rule is similar to a lookup rule, except that only one rule may be used as the *key* in the lookup, and the resulting values must be Boolean (true or false). Although this rule type has tighter restrictions, the configuration is simpler.

Figure 5 - Flag Rule



For instance, a flag rule could convert Revenue Code to ICU Charge, indicating which Revenue Codes qualify as ICU charges. Like the lookup rule, this uses an external (flag table) file to map the values. The script snippet to define such a rule might be:

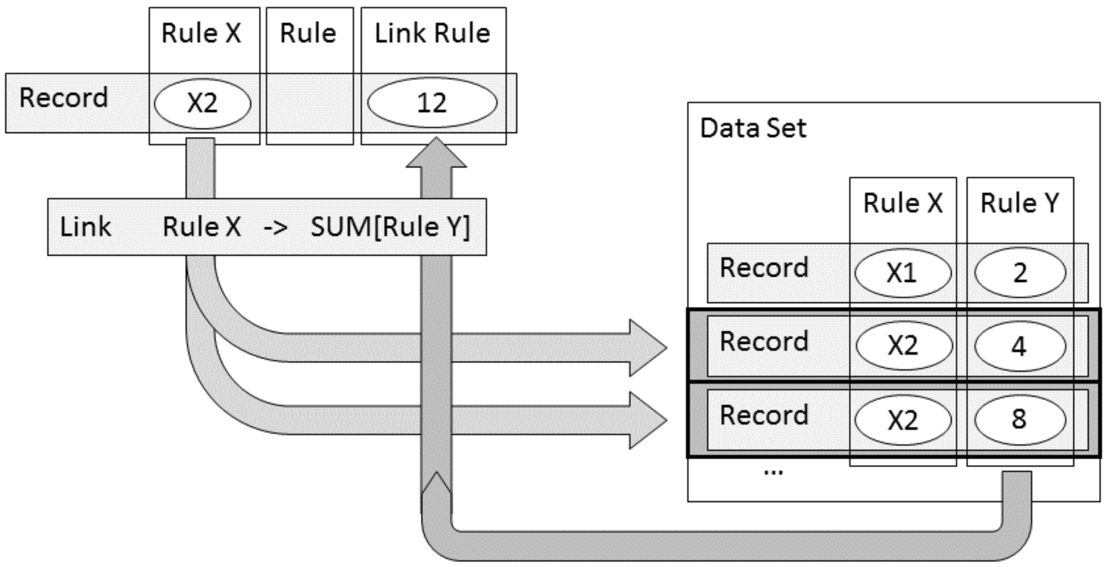
```
flag-table "Revenue Flags" {
  date rule="Posting Date"
  key rule="Revenue Code"
  flag-rule "ICU Charge"
}
```

Similar to lookup rules, flag rules can use an *effective date range*, in case the mapping changes over time.

Link Rules

A link rule is used to move data from one data set to another. To use a link rule, there must be a key: a rule in one data set that matches a rule in the other. For instance, both the Accounts and Charges data sets may have *Account ID* in them. The link rule connects records from the two data sets which share the same value for the selected key. Then it applies a summarizing mathematical expression to the records in one data set, and places the result in the corresponding records in the other data set.

Figure 6 - Link Rule



For instance, a rule *Has ICU Charge* on the Accounts data set might be defined as whether or not there exist any Charge records for the same Account ID where the ICU Charge rule is true. The script snippet for such a rule might be:

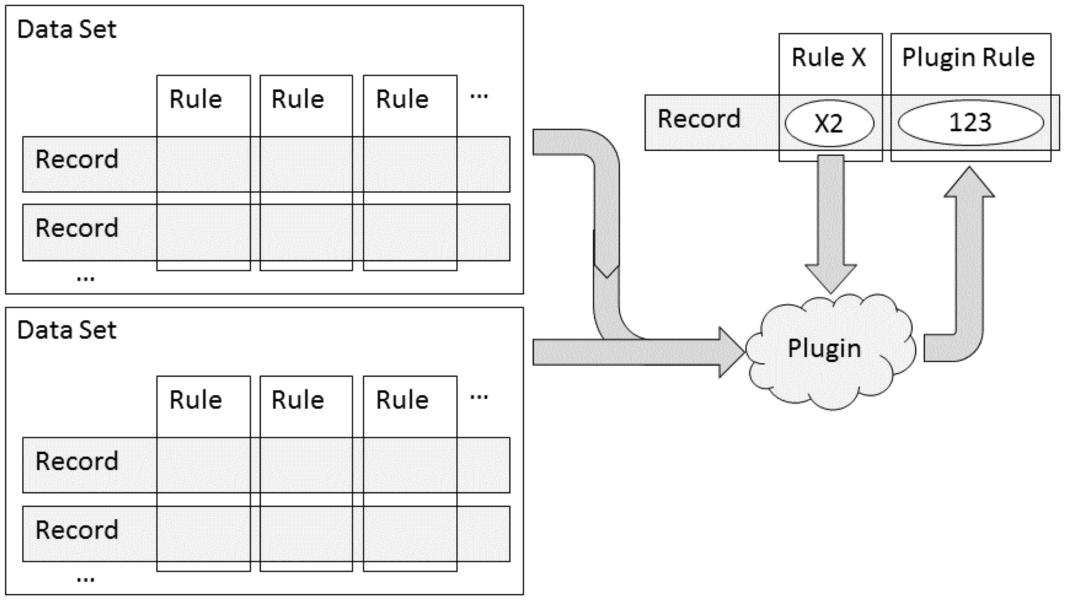
```
link "Charges" {
  key "Account ID"
  link-rule "Has ICU Charge" `count() > 0` filter=`value("ICU Charge")`
}
```

One of the challenges that the Measure Factory overcomes is the tracking of rule dependencies across links. For instance, a rule in the Accounts data set might depend on a link from Charges, which might depend on a different link back to Accounts. When a factory becomes large and complex, it can be difficult to follow the trail of dependencies from one rule to another, especially as the data needs to flow back and forth between data sets, but because the Measure Factory takes care of that problem the user can focus on one rule at a time. As long as the inputs to a rule are correct and the rule definition is correct, the user can be confident that the rule will produce correct results. Being able to validate rules in isolation from one another allows users to distribute data governance among multiple people, and helps to produce a scalable data solution.

Plugin Rules

Plugin rules are available to produce rules that are not possible using the other rule types. A plugin rule executes an external process to determine the rule values, providing that external process with data being computed during the factory build process, and then joining the result of that process back into the data set. This allows the user to inject more complicated logic into the factory while still taking advantage of the automatic management of the data flow within Measure Factory.

Figure 7 - Plugin Rule



For example, to compute whether an Admission is a *Readmission*, we may need to determine if that encounter occurred within a certain number of days after the previous encounter. This requires looking at data outside of each individual account record (that is, the prior records), and none of the other rule types can accommodate that kind of query. If we install a plugin which can handle readmission calculations, then we only need to invoke that plugin to compute the Readmission rule. The script snippet for that rule might look like this:

```
plugin "Readmission" {
  input "Accounts" {
    column "Account ID"
    column "Admission"
    column "Admit Date"
    column "Discharge Date"
    column "DRG"
    column "MRN"
  }
  dimension "Account ID"
  plugin-rule "Readmission"
}
```

The Integrator script would encompass whatever steps are required to perform the readmission logic and determine the value for the Readmission rule.

DATA FLOW

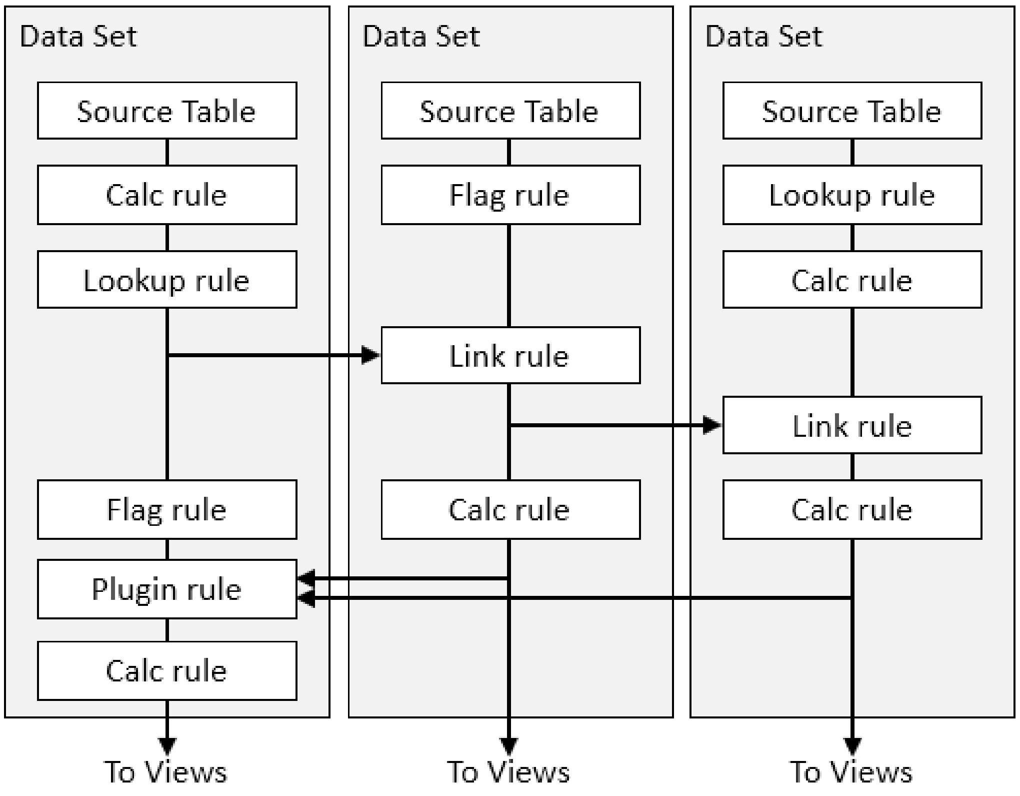
A major feature of the Measure Factory is its ability to automatically manage the application of rules, so that the user configuring the factory can focus on the rule definitions and need not manually track the flow of data through the process. The process consists of the following elements:

- Swim Lanes
- Rule Application
- The Build Plan
- Build Scripts
- Dive Scripts and Link Rules
- Integrator Scripts and Plugin Rules

Swim Lanes

The Measure Factory processes data using a *swim lane* method, in which each data set is built up from its source table, rule by rule, until all of the rules are added. At that point the records are complete and the data sets are ready to be turned into views. Most of the time, a data set will stay in its swim lane, but for certain rule types (link and plugin) it may be necessary to transfer data from one lane to another.

Figure 8 – Data Swim Lanes

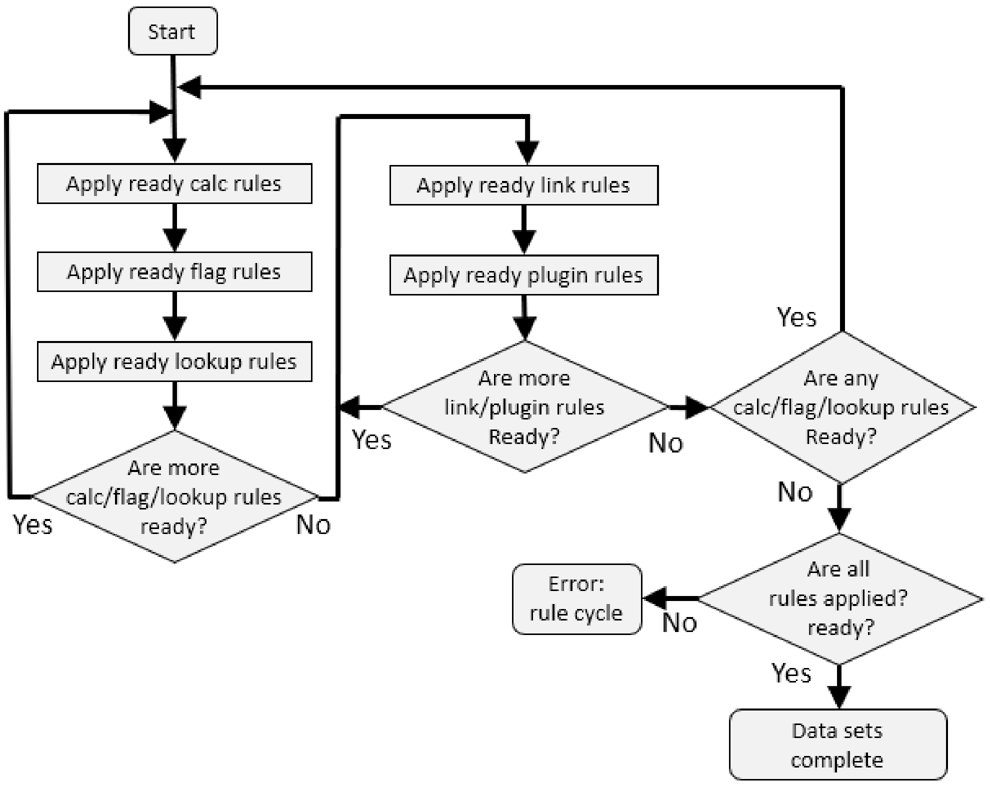


Rule Application

The Measure Factory has an algorithm that decides the order in which to process rules. A rule is considered ready if it does not depend on a rule which hasn't yet been applied. The algorithm cannot apply a rule unless it is ready. First, it applies all of the calc, flag, and lookup rules which are ready. When there are no more ready calc, flag, and lookup rules, check to see if there are link or plugin rules that need to be applied. If so, apply any link or plugin rules that are ready, and then return to the beginning, to process more calc, flag, and lookup rules. Here is an outline of that algorithm:

1. For each calc rule, apply it if it is ready.
2. For each flag rule, apply it if it is ready.
3. For each lookup rule, apply it if it is ready.
4. If any rules were just added in 1-3, go back to 1.
5. For each link rule, apply it if it is ready.
6. For each plugin rule, apply it if it is ready.
7. If any rules were just added in 5-6, go back to 5.
8. If any calc, flag, or lookup rules are ready, go back to 1.
9. If any rules are not yet applied, error "rule cycle".
10. The data sets are complete.

Figure 9 – Rule Algorithm

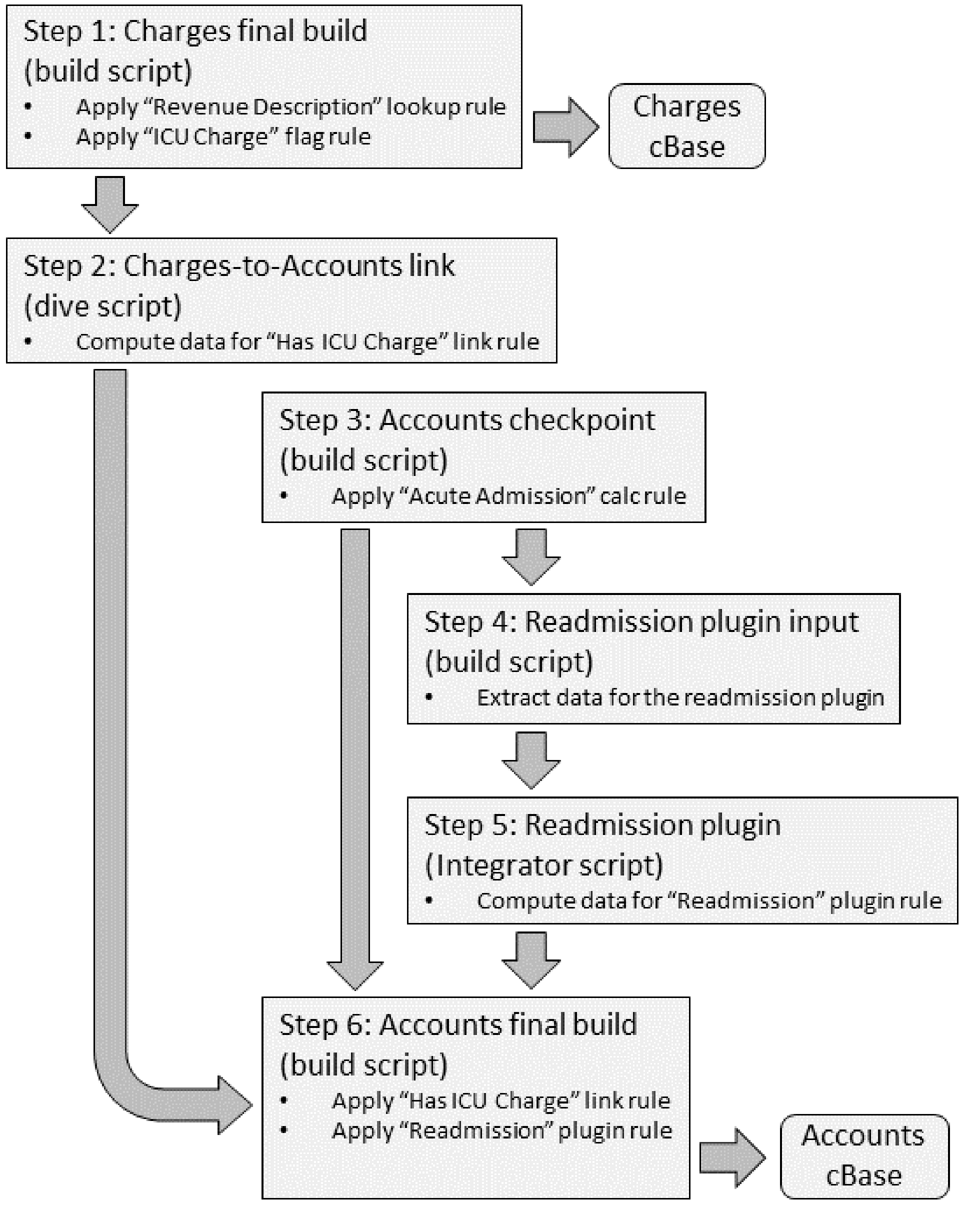


Each phase of the algorithm is applied to all data sets, so if the Measure Factory is running in the phase 1-4 loop and a data set becomes blocked waiting for a link rule, the Measure Factory will continue processing ready rules in other data sets until the condition in phase 4 completes and it goes to phase 5.

The Build Plan

The rules are implemented using a combination of Dive, Build, and Integrator scripts. The dive and build scripts are run as if by the `spectre dive` and `spectre build` commands, and the Integrator scripts are run by Data Integrator. The running of one of those scripts is a step in the build process. The Measure Factory produces an ordered list of these steps (a *build plan*) and then runs each script to produce the data set cBases. The scripts are made available after the factory build in the `/factory-output/__internal` directory, so they can be used to diagnose problems or to manually track the data flow if that becomes necessary.

Figure 10 - The Build Plan


Step 1: Charges final build
(build script)
• Apply "Revenue Description" lookup rule
• Apply "ICU Charge" flag rule
Charges cBase
Step 2: Charges-to-Accounts link
(dive script)
• Compute data for "Has ICU Charge" link rule
Step 3: Accounts checkpoint
(build script)
• Apply "Acute Admission" calc rule
Step 4: Readmission plugin input
(build script)
• Extract data for the readmission plugin
Step 5: Readmission plugin
(Integrator script)
• Compute data for "Readmission" plugin rule
Step 6: Accounts final build
(build script)
• Apply "Has ICU Charge" link rule
• Apply "Readmission" plugin rule
Accounts cBase

Build Scripts

All of the rules are added to a data set in Build scripts. It's possible for an entire data set to be made using a single Build script. But rules in other data sets may require the build to be broken up into separate scripts, so that the intermediate data can be used in a Dive or Integrator script.

If a Build script is the last script needed for a data set, it is called the *final* build script, and its filename would be something like *006_Accounts_final.build*. Build scripts which are used to save intermediate data for plugin or link rules are called *checkpoints*, and a checkpoint filename would be something like *003_Accounts_checkpoint.build*.

A factory-generated Build script might look like this (slightly abbreviated here):

```
build {
  cbase-input "/data_sources_build/accounts.cbase"
  output "006_Accounts_checkpoint.cbase"

  // Calculation rule "Outpatient":
  add "Outpatient" `not value("Inpatient")`

  // Flag table "Location Flags":
  lookup {
    text-input "/config/flag_table_Location Flags.txt"
    [...]
    column "NICU Location Key"
  }

  // Lookup "Admit Type":
  lookup {
    text-input "/config/lookup_table_Admit Type.txt"
    [...]
    column "Admit Type"
  }

  // Link to data set "Charges":
  lookup {
    cbase-input "002_Accounts_Charges.cbase"
    [...]
    column "Has ICU Charge"
  }

  // Plugin "Readmission":
  lookup {
    cbase-input "004_Accounts_Readmission/output.cbase"
    [...]
    column "Readmission"
  }
}
```

This Build script would be saved as *006_Accounts_final.build*, would be step 6 in the build plan (as shown in Figure 10), and would depend on having steps 2 (charges link) and 5 (readmission plugin) run before it. The Measure Factory would effectively run the Build script like this:

```
spectre build 006_Accounts_final.build
```

Dive Scripts and Link Rules

Dive scripts are necessary to apply link rules. In a link rule, records from the source data set are summarized and then the resulting table is joined with the target data set. The Dive script performs the summarization and the subsequent Build script performs the join. So, there are two phases to a link rule:

- **Phase 1**—Compute the summary data from the source data set
- **Phase 2**—Join the summary data into the target data set

The phase one is done in a Dive script. A factory-generated Dive script might look like this:

```
dive {
  cplan {
    input "data-sets/Charges.cbase"

    // Link rule "Has ICU Charge":
    calc "__r_Has ICU Charge" `count() > 0` filter=`value("ICU Charge")`
  }
  window {
    dimension "Account ID"
    column "__r_Has ICU Charge"
  }
}
```

The Measure Factory would effectively run the Dive script like this:

```
spectre dive 001_Accounts_Charges.dive --cbase-output
001_Accounts_Charges.cbase
```

In this example, the Dive script takes as input the final Charges cBase, which indicates that a previous build step in the build plan finished all of the Charges rules.

The phase two of a link rule is done in a Build script, and is usually combined with the application of other rules.

Integrator Scripts and Plugin Rules

Integrator scripts are necessary to run plugin rules. These scripts are not generated by the factory build process, but are prepared separately and then located in the `/plugins` folder of the factory project. There are three phases needed to execute a plugin, which can be associated with three or more steps in the build plan:

- **Phase 1**—Extract data from a data set to form the plugin's input
- **Phase 2**—Run the plugin Integrator script, producing the plugin's output
- **Phase 3**—Join the plugin's output back into a data set

Phase one (extracting data) requires one step per plugin rule input. A plugin rule input script snippet might look like this (abbreviated here):

```
build {
  cbase-input "003_Accounts_checkpoint.cbase" {

    // Remove columns that are not needed by the plugin:
    remove "Facility"
    [...]
  }
  output "005_Accounts_Readmission/Accounts.cbase"
}
```

The Measure Factory would effectively run the Build script like this:

```
spectre build 004_Accounts_Readmission_Accounts.build
```

Note that while this would be step 4 of a build plan (as shown in Figure 10), the result of this Build script is placed in a folder called `005_Accounts_Readmission`(in `factory-output/_internal`). That's where the plugin rule Integrator script will expect to find it when it is run as step 5 of the build plan.

In this example, the script merely removes columns from the Accounts data that the Readmission plugin did not request. It may also rename columns as well, so that the resulting cBase has the same column names that the plugin script expects.

Phase two of a plugin rule is done by running the plugin rule's Integrator script, something like this:

```
cd 005_Accounts_Readmission
integ /plugins/Readmission/readmission-script.int
```

Note that the plugin rule script is run with the *working directory* set to the folder created for that step in the build plan. The plugin is expected to produce a single cBase, named `output.cbase`, in that folder.

Phase three of a plugin rule is done in a Build script, and is usually combined with the application of other rules.

CALCULATING MEASURES

Once the factory build plan is executed, most of the factory processing is complete.

Data Set cBases

After all of the rules have been applied, the Measure Factory will have created final cBases for each data set. These can be found in `/factory-output/__internal/data-sets`. Each data set cBase will have the same number of records as were present in the source table for that data set. It will also have a column for almost every rule (including the source rules, which came from columns in the source table).

The only rules for which there will be no column in the data set cBase are calc rules which were not used by any other rule type. These calc rules are omitted from the cBase because they are easy for the Spectre engine to compute later, and omitting them reduces the size of the cBase on disk and may improve data query performance by allowing the Spectre engine to use a smaller amount of memory during queries.

Once the data set cBases are created, the factory is almost ready to be queried. The next step is to apply the view aliases, so that when measures are computed across a time range, each measure uses the particular date column associated with that measure.

For instance, we can calculate the Admissions and Discharges measures directly against the data set cBase, but if we wanted to see those measures over time by diving on a date dimension, we would have to choose either Admit Date or Discharge Date. We dive on the abstract Date dimension and let each measure choose its own actual date. To understand how this works, we'll need to understand multi-level dives.

Multi-Level Dives

A multi-level dive is a query against two or more tables simultaneously, in which data from those tables is joined together into a single result. Suppose we have the following 2 tables (Accounts and Charges):

Accounts		Charges		
Account ID	Length of Stay	Charge ID	Account ID	Charge Amount
A_01	4	C_01	A_01	110.00
A_02	2	C_02	A_02	260.00
A_03	7	C_03	A_02	80.00
A_04	1	C_04	A_03	135.00
	C_05	A_03	25.00	

And in our query we want, for each account, to see the ratio of the Total Charge Amount divided by the Length of Stay. Neither table alone can provide this result, but we can query the two tables independently and then merge the results to get our answer. We select the Length of Stay by Account ID from the Accounts table, and the Total Charge Amount by Account ID from the Charges table:

Accounts		Charges	
Account ID	Length of Stay	Account ID	Total Charge Amount
A_01	4	A_01	110.00
A_02	2	A_02	340.00
A_03	7	A_03	160.00
A_04	1		

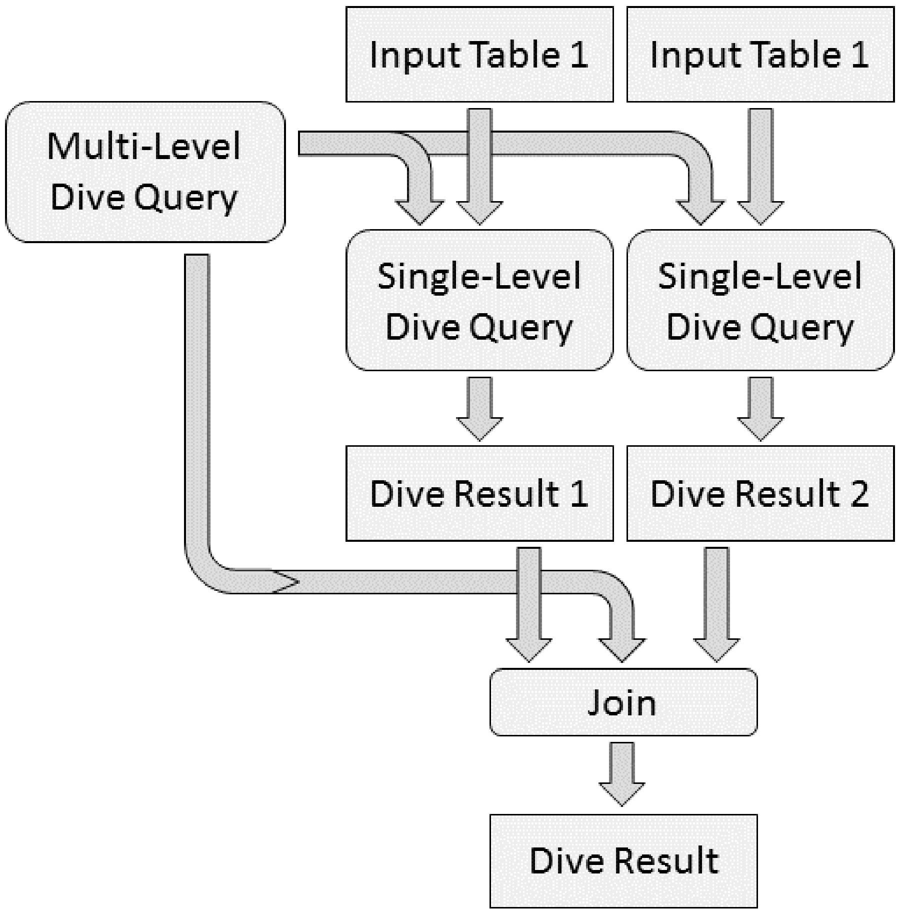
Then we line up the Account IDs and compute the ratio:

Account ID	Ratio
A_01	27.50
A_02	170.00
A_03	22.88
A_04	N/A

Multi-level dives work by first dividing the query up into parts that can be resolved by each table individually, then lining up the dimension values and combining the results to reach the answer to the query.

More generally, a multi-level dive is deconstructed into multiple single-level dives in which each dive includes only the dimensions present in that level's source table, and then those dive results are joined.

Figure 11 - Multi-Level Dives



This process can be used to construct the abstract Date dimension. When a query asks for the value of two measures which use different dates, the Measure Factory divides the query into different views, each of which has a specific date dimension selected, then combines the results.

View cBases

Each view in the factory is based on a particular data set, but with dimension aliases applied. For instance, the Admissions view is the Accounts data set where Admit Date is aliased to Date. To get the data ready for multi-level queries against the views, the Measure Factory takes the data set cBases and copies them to view cBases, adding the aliased dimensions along the way.

The process of copying a data set cBase to a view cBase is another step in the build plan, and is accomplished by a Build script. A view build script might look like this (abbreviated here):

```
build {
  // This view is based on data set "Accounts"
  cbase-input "data-sets/Accounts.cbase"
  output "views/Admissions.cbase"

  // "Date" is aliased from "Admit Date":
  add "Date" `value("Admit Date")`

  // Rename non-dimension rules:
  rename "Admission" "Rule: Admission (Accounts)"
  [...]
}
```

The view cBases can be found in `/factory-output/__internal/views`. There are two differences between the view cBases and the data set cBases. First, aliasing the Date dimension in view cBases allows them to be used in multi-level dives, to provide values for measures that use different dates. Second, rules in the view cBase are renamed with the pattern:

```
Rule: <rule name> (<data set name>)
```

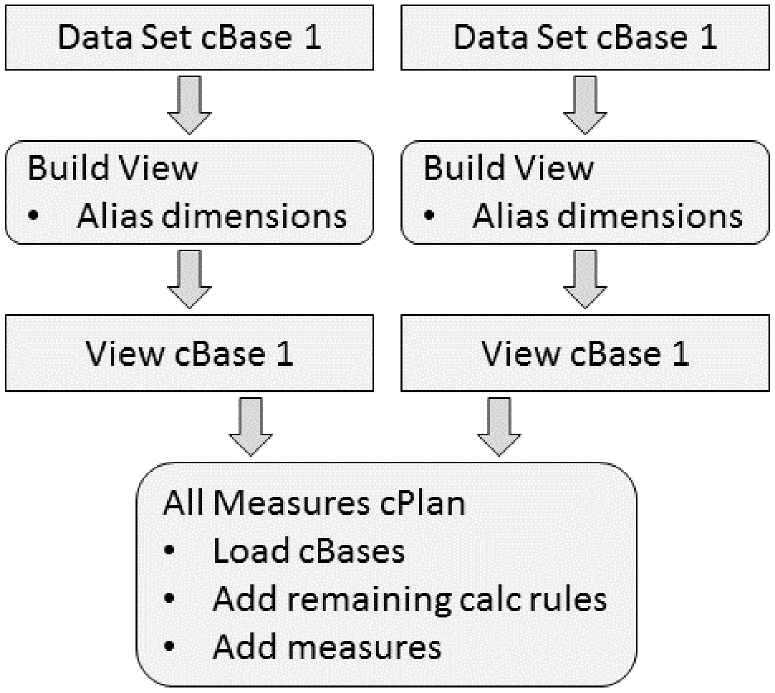
Renaming the rules is necessary in preparation for being merged in multi-level dives. Rule names are unique within their individual data sets and there is no requirement for rules in two different data sets to have different names (with the exception of dimensions, discussed below). Also, later cPlans will add measures, and the Measure Factory needs to make sure measure names won't collide with rule names either.

The All Measures cPlan

The first point at which all the data is finally gathered into a single entity is the all measures cPlan, which can be found at `/factory-output/__internal/all-measures.cplan`. Here, all the view cBases are merged together and the measure calculations are finally implemented. Creation of the all measures cPlan has three phases:

- **Phase 1—**Load the view cBases
- **Phase 2—**Add calc rules that were not written into the view cBases
- **Phase 3—**Add measure calculations

Figure 12 - Creation of the All Measures cPlan



In phase 1, the views are loaded. A view may be loaded like this:

```
        cbase-input "views/Admissions.cbase" name="Admissions"
```

Not all rules must be renamed, however. The factory configuration may specify a number of dimensions, which can span multiple data sets by having a rule in each data set. Rules which will become dimensions are left with their name unchanged. This is possible because there is a requirement that dimensions and measures not have the same name.

Recall that the data set cBases do not have columns for every rule. Calc rules that were not used by other rules are not saved into the cBase. Those rules are added in phase 2 of the all measures cPlan. For instance:

```
        dimension "Rule: ED Admission (Accounts)" `value("Rule: Admission
      (Accounts)") and value("Rule: ED Account (Accounts)")`
```

Note that the rule is added with the dimension tag. This means that if you open up *all-measures.cplan* in ProDiver you can dive on the rule. That's not usually employed in dashboards, but can be very useful to help validate the data or diagnose issues.

Finally, in phase 3, measure calculations are added. For instance:

```
    calc "Admissions" `input("Admissions", count())` filter=`value("Rule:
Admission (Accounts)")` format="#,#"
```

The Admissions measure here is the count (thus `count()`) of records in the accounts data set for which the Admission rule is true (thus the filter). It uses the Admissions view, in which Date is an alias for Admit Date (thus the call to the input() function).