IMS 2015
12th International Mathematica Symposium

Implementation of custom data source for simple evaluation of large experiment sets

Václav Čejka
VÚTS a.s., Svárovská 619, Liberec, Czech Republic

Abstract

For couple of years Mathematica provides wonderful data sources from many different areas (from AdministrativeDivisionData to ZIPCodeData, taken by alphabet). Cloud integration now provides the way to share custom know-how and data. However in our company we missed the opportunity to create our own data source based on our input. Such data can arise as the result of some large set of experiments and their evaluation. In our case typically some measurements are performed with different parameter settings and environment conditions. Such measurements should be stored in some general structure with a symbolic way of extraction of data. Then some analysis are performed and again their results are stored to be symbolically retrieved later.

Introduction

Both research work and engineering praxis is often based on smaller or bigger set of experiments that must be analysed. It is very important to have some system of organization of data, otherwise the research is soon lost in the mess whenever the research crosses some limits in amount of data or time. Proper research must be also well documented for other colleagues.

Analysis of the experiments often follows similar procedures and similar ways of visualization of results. For a researcher it is important to focus on the idea he or she follows, not on repeated creation of new and new procedures for data extraction and visualization in some specific way. There is often a risk he / she gets lost in syntax elements, long and hard-to-read procedures etc., and the examined idea vanishes. There are many examples from history, and also from our own history, when the researcher was very close to prove some interesting idea, but he missed it because the important facts were hidden too long in the rest of data.

Proper research often contains long list of analysed data and experiments. These data must be stored easily and retrieved also easily. Twenty experiments can be stored in *Mathematica* Lists or some files. Hundreds of experiments should be handled with more care. Its said that Edison had to test 3000 different material and procedures before he (and his team) found the way to construct working bulb. Many experiments in engineering, pharmaceutical research, genome research etc. contain similar big sets of experiments.

Very convenient way of extraction of already known data is implemented in `-Data` methods in *Mathematica*. Basic limit is in fact these data sources are read-only and there is no prepared way of creating new data sources, committing data in them etc. In this article we want to suggest one possible implementation that should overcome this limit. This implementation is not perfect nor complete, but should server as the inspiration for future research in this area. One of the possible future goal might be opened system of private data sources operated by some powerful data extraction engine like Wolfram Alpha.

Methods

Any such system must be based on some form of database engine. We chose mathematic built-in `DatabaseLink`` as a data access layer. Arbitrary database supported by this engine can be used as the data layer. Because direct work with `DatabaseLink`` is quite difficult and too low-level for common work, high-level application level was implemented inspired by the syntax of *Mathematica* built-in `-Data` commands. All functionality is hidden in `SignalData` command. We had to add new syntax allowing not only data extraction, but also configuration of the data source and committing new data to it. For the sake of systematic analysis, we had to suggest general structure of the database.

Using `SignalData[]` command with no parameters, list of names of data sources configured in the local computer is given. Each data source is fully configured by some connection string, but using this connection string for each access to the data source is too complicated. For this reason connection string have names and `SignalData[name]` opens the data source of given name (if it is not opened) and returns the *Mathematica* connection object. In all functions, only name of the data source can be used instead of the whole connection string. Calling `SignalData[name]` with new name automatically creates new empty data source on default database. By default, connection remains opened until it is manually closed using `SignalData[name, "Close"]` command, or the kernel is shutted down. Usually, it is not needed to close the data source connection during work. By default, file database HSQL is used, which is very fast (memory based) database. Its main limit is that only single user can open the database at the moment. Often it is not the real limit, but if more users need to access the data source at once, other server-based databases like MySql can be used.

Structure of stored data

Data source is the database with specific data structure. It contains experiments described by filename with original data (and internally with unique integer ID). This filename can typically contain relative path to the database files (for HSQL databases), but can contain also absolute path. Max. allowed path length is 250 characters. The path must be unique, because it is used for searching.

We come from machine construction environment, so following examples will be focused on this area. However the whole system is general and can used in any other area of work. We use the system especially for evaluation of the measurements, but it can contain also results from symbolic analysis or any other kind of work.

Experiment refers to the whole experiment, not to particular measured data. Different properties can be defined for the experiment, like parameters set on the measured machine, date / time of experiment, location of experiment, temperature etc.

Signal represents single measured quantity / channel / signal of the experiment. It is uniquely described by relation to the experiment and property `"Quantity"` (internally also by unique integer ID). Property `"Quantity"` is the only property created on default for each `SignalData` data source. It is integer number determining uniquely single measured signal in experiment, i.e. force or acceleration of some specific part of machine. It can be directly channel number from measurement settings or any other well defined enumeration.

Different properties can be again defined for signals, like their mean values, amplitudes of their 1st harmonics, relation to machine part or some manually defined parameter specific for this measurement, that should be used for further processing. Usually only some values are stored to database for some signals, different quantities will have different properties stored. But for the sake of simplicity all signals have the same definition with the same properties.

By default the measured signal is not stored to the database, only filename of the experiment. Usually only desired properties are calculated from the original data during import and they are stored to the database. But sometimes even full signals need to be stored. They are typically not full original signals, but i.e. couple of periods from original signal with noise reduced, statistical curves like mean course from multiple cycles, or derived signals, like calculated forces etc. For each signal quantity in 1 experiment 1 signal data can be stored, or better said referenced by database.

As was mentioned, any number of properties can be defined for experiments and signals. For single data source, property names for signals and experiments together must be unique to avoid ambiguity in accessing property of experiments and property of signals. Properties has some default attributes and named values can be defined for them.

Attributes are predefines basic characteristics of the property. There are defining attributes defining the property. `“PropertyName”` is used as unique key for the property. It is used for further retrieving data from the database, so it should be well understandable, not an abbreviation. It is also used as name of the database column, so further restrictions may apply for different databases. Each property has some `“DataTypeName”` and for some data-types also `“DataLength”`. These attributes define physical type of underlying database representation. They must be provided when property is defined and they cannot be easily changed. Last defining attribute is called `“Nullable”` and defines, if the property can hold Null values (default status) or only specific values of given datatype.

There are also optional, but important descriptive attributes. They can be changed anytime and are used for better understanding and automatic generation of output values when data are extracted from the data source. These attributes are `“Units”`, `“Label”` and `“Description”` of the property.

It might be useful to represent some specific values of some property by user defined description. For example default property `"Quantity"` is the enumeration of integers, but it would be nice to assign text for it. For some boolean properties interpretation by custom string would be also cute. This is the purpose of named values. Generally any string of 25 characters can be assigned to any combination of property (given by its name), value (given as string of 15 characters) and optionally `“Quantity”` (Integer or `Null`, if assignment is not dependent on `“Quantity”`). Value has limit of 15 characters, because obviously it has no meaning of using it on String properties, it is even dangerous, because it cannot be decided, what is true value and what is named value. It should not be used on float values because different possible representations of float values. For boolean properties values `"True"` and `"False"` should be used. Value `"Null"` can be used for any nullable property. Internally all values retrieved from the database will be converted to string using `ToString` command and then compared. In conditions, string values are supposed to be named values and are searched in database to be translated to their true value.

Named values should be used with care, too many of them may be very slow. Use them only for booleans and enumerations.

There is also a special use of named values together with property units and labels of signals. Many, maybe most signals have different units and labels for different quantities stored in the database. Imagine even basic statistics like mean value, standard deviation, minimum and maximum - they have the same units as original signal, so the attribute `"Units"` cannot be used directly for all signals of different quantities. Giving string of form `"`somestring`"` in `"Units"` and `"Label"` attributes force the engine to search for named value "somestring" for given property and quantity.

Implementation

Configuration

Generally, `SignalData` has always the data source name as the first parameter. All commands are implemented as the second parameter of type `String`. Next parameter is usually either identification of the manipulated entity, i.e. property or named-value, or `All` keyword for accessing all entities. In many cases last parameter is the condition reducing the list of influenced entities. For configuration of the data source, following commands can be used.

`SignalData[dsrc, "GUI"]`	opens graphics interface for configuration
`SignalData[dsrc, "Property"]`	returns list of all defined properties
`SignalData[dsrc, "Property", propname]`	returns attributes of given property
`SignalData[dsrc, "Property", All]`	returns list of all defined properties with their attributes

`SignalData[dsrc, "Property", AddTo, rules]`	adds new property with attributes given by rules
`SignalData[dsrc, "Property", SetAttributes, rules]`	changes attributes of property
`SignalData[dsrc, "Remove", propname]`	remove property
`SignalData[dsrc, "Remove", prop1, prop2, ...]`	removes more properties
`SignalData[dsrc, "Property", Names, cond]`	returns list of properties with attributes satisfying given condition
`SignalData[dsrc, "Property", All, cond]`	returns list of properties and their attributes with attributes satisfying given condition

Usually, the user does not need to use textural form of configuration commands, although it can be useful in some auto-generation code. Most often graphical configuration tool is used, see the example on Fig. 1. Following example shows the procedure for retrieving filtered list of properties holding some condition:

```
SignalData["TestDb", "Property", All, ! StringMatchQ["DataTypename", "*INT*"] &&
"IsSignalProperty"] // TableForm
```

```
"PROPERTY2"    "IsSignalProperty" → True    "DataTypeName" → "DOUBLE"    "DataLength" → Null    "
```

Properties | Named values

Add | Delete | MEANVAL | Apply changes to Db

Exper./Signal	Name	Type	Length	Nullable	Units	Label	Description
Experiment	ELONGATION	INTEGER		☑	%	Elongation	Roving nominal elongation
Experiment	ANGLE	INTEGER		☑	deg	Angle	
Experiment	ZONELENGTH	INTEGER		☑	mm	Zone length	Length of the zone between rollers
Experiment	SENSORPOS	INTEGER		☑		Position of sensor	Position of force sensor
Experiment	ROVING	INTEGER		☑		Used roving	Roving used for the measurement
Experiment	AMPLIFICATION	INTEGER		☑		Amplification	Amplification used on Waweon electronics
Experiment	DATUM	DATE		☑		Date of measurement	Date of measurement
Experiment	MARKS	VARCHAR	2147483647	☑	index	Marks	Positions of marks on the loop
Experiment	VELOCITY	INTEGER		☑	mm/s	Feeding velocity	Feeding velocity
Experiment	NUMBER	INTEGER		☑		Experiment number	Experiment number
Experiment	LOOPS	INTEGER		☑		Loops	Number of loops in measurement
Experiment	LASTLOOPFULL	BOOLEAN		☑		Last loops	Is last loop finished in the measurement?
Experiment	ELONLOOPS	INTEGER		☑		Elongated loops	Number of loops still longer than previous loop
Experiment	STAB	INTEGER		☑			
Signal	QUANTITY	INTEGER		☑			
Signal	FIRSTVAL	DOUBLE		☑			First value of signal
Signal	MEDIANVAL	DOUBLE		☑			Median of signal
Signal	MEANVAL	DOUBLE		☑			Mean value of signal

Fig. 1: Example of the graphical configuration screen.

Adding and updating data in the database

For adding and updating data in the data source, “Commit” command has been defined. Following table shows possible notations.

`SignalData[dsrc, "Commit", "Experiment" → name, rules]`	commit data of experiment (not signal) identified by experiment name or ID
`SignalData[dsrc, "Commit", "Experiment" → name, "Quantity" → q, rules]`	commit data of experiment and / or signal identified by experiment name or ID and quantity
`SignalData[dsrc, "Commit", "SignalID" → id, rules]`	commit data of experiment and / or signal identified by signal ID
`SignalData[dsrc, "Commit", id, "File" → expr]`	create internally compress file in database folder and join it to (somehow) identified signal

For committing data to the database, the affected record must be uniquely identified. There are following possibilities:

- Storing experiment properties to the new experiment - experiment name must be provided
- Updating experiment properties of the existing experiment
  - existing experiment name or ID is provided. For non-existing experiment name new experiment is automatically created and stored to the database
  - `"SignalID"` of existing signal that belongs to the experiment is given. Such a way experiment name / ID is not used even if provided.
- Storing signal properties to the new experiment - same as storing experiment properties but also `"Quantity"` must be provided
- Storing signal properties to the new signal of existing experiment - existing experiment name or ID together with `"Quantity"` of new signal must be provided
- Updating signal properties of the existing signal
  - existing experiment name or ID is given together with signal `"Quantity"` must be provided. For non-existing experiment name or `"Quantity"` new experiment and signal is automatically created and stored to the database
  - `"SignalID"` of existing signal. Such a way experiment name / ID and Quantity is not used even if provided.

For storing new experiment to the database without setting any property or creating any signal only string, typically filename or path to original measured data, must be given. Function returns internal ID identifying the record in the database. Using this ID instead of string in further operations is slightly faster.

Record can be created and new value can be stored all at once. When signal property is stored for some `" Quantity "`, also internal unique ID identifying signal record in the database is returned. When this signal ID is used later in commit, experiment does not need to be identified again, it is already identified by this signal ID.

For commiting some types of expression, special wrappers must be used, see examples below (`SQLDateTime`, `SQLExpr`, `SQLBinary`).

Each signal can be acompanied by any data stored to external file (in database folder), just commit the expression to `"File"` property.

Following example creates or updates experiment property `"PropertyExp"` to value 7 for experiment of given name. Signal property `"PropertySig"` of this experiment for `"Quantity"` equal to 1 of type DateTime is set to current date. Another property is set to *Mathematica* expression - this property must be of type string or equivalent. For this signal, also file is created and another *Mathematica* expression is stored to it. Content of the file can be accessed using `"File"` property. As a result, unique ID for the experiment and signal is returned.

```
SignalData["TestDb", "Commit", "Experiment" -> "a3.uff", "PropertyExp" -> 7, "Quantity" ->
1, "PropertySig" -> SQLDateTime[DateList[]], "PropertyExpr" → SQLExpr[x^2 + y^2], "File"→
Range[100]]
```

```
{"Experiment" -> 2, "SignalID" -> 1}
```

Extraction of the data from the data source

There are two basic way of data extraction - command based and combinatorial. Command based extraction uses `"Experiment"` command and some condition to retrieve list of data satisfying given condition in raw form, as they are in the database. Using `"ResultType"` option they can be extracted in either tabular form typical for databases, or as list of rules, more convenient way for *Mathematica* users. Output can be limited to only some properties using `"ShowColumns"` option and the sorting can be defined using `"SortingColumns"` option. `"ShowColumns"` can also use expression supported by underlying database for generation of artificial columns. For example value `"Property1 + 2 * Property2 AS ArtificialProperty"` gives property called `"ArtificialProperty"` composed from algebraic combination of original properties.

Experiments can be filtered based on some condition. Conditions typically contain property names, comparisons and logical operand. Operand `MemberQ` and string matching `StringMatchQ` are also supported. Condition is held (using `Hold`) not to be evaluated outside of database engine. If you need to evaluate some part of condition before database engine is launched, you can surround it by `Evaluate`. Example of the call with its output follows:

```
SignalData["TestDb", "Experiment", (! MemberQ[{3, 4}, "PROPERTYSIG"]) && ("PROPERTYSig" +
"PropertyExp" > 10)] // TableForm
```

```
"a3.uff"    "PROPERTYEXP" → 8    "QUANTITY" → 1  "FILE" → "SigData2_1.sgdat"  "PROPERTYSIG" → 2
```

More advanced way of extracting data is direct composition of stored information to computable data, or other word direct analysis of dependences between properties. Instead of `"Experiment"` keyword the dependent property or expression or their list is given, than optional property or expression used for the independent variable is given. Of course condition can supplied at the end. The command creates tuples of independent and dependent variables. If they can be interpreted using some supported method, the result is given in this interpreted form. Examples will be given in below.

Properties given as string can again contain any database-supported expression. Furthermore these strings can be part of any *Mathematica* expression, which is mapped on each row of resulting dataset. Also database aggregation functions can be used in x and y columns with some restrictions:

- aggregation functions cannot be used in the condition at this moment (maybe in future)
- named values will not be translated for extracted data in this case (but named values can be still used in condition)

- if aggregation function is used for some *y* column, it must be used for all *y* columns (restriction of SQL - database language - syntax). Also no generated column like "`Prop1 * Sum(Prop2)`" is allowed. But you can calculate i.e. "`2 * Avg(Prop1 * Prop2)`"
- because different databases implement different sets of aggregation functions, only basic generally accepted aggregation functions, i.e. `Count`, `Sum`, `Avg`, `Max` and `Min`.

If aggregation function is used for *x* column (property), the there is only 1 x-axis point in the result (or none). If aggregation function is used for some *y* column but not for *x* column, *x* column will be used for grouping results, so aggregation functions on *y* columns will be applied on sets of values for which *x* is the same. For example is *x* is `"Quantity"` and *y* is `Avg(Prop1)`, average value separately for each quantity is calculated, like in following example.

```
SignalData["TestDb", "Avg(PropertyExp)", "Quantity", "Quantity" != Null]
```

Contingention tables can be made by providing list of *x*-properties. In this case data will be extracted with *x*-axis of the first property in the list multiple times for each value of other properties present in the database. The following example will extract all "`PropertySig`" values based on values of "`PropertyExp`" for each value of "`Quantity`". The result will have depth = 3.

```
SignalData["TestDb", "PropertySig", {"PropertyExp", "Quantity"}, True]
```

Results

The described system was used in our research many times in many different circumstances. Here we would like to show some basic and very common applications.

The first example comes from the set of about 60 measurements on some developed device. Force was measured in different cycles with different setup of the measurement device. Parameters of the measurements were position of the sensor in the measurement zone given by enumeration with some defined named values, length of the measurement zone, angle in which the sensor is oriented with respect to the analysed material, and the elongation applied to the measured flexible material by measurement device. In this case mean values of forces in each cycle were stored in files related to signals. `SignalData` command extracts the content of the file and values of particular parameters. There is no independent variable defined in this case. Only quantity mean force is extracted using named value. Only filtered list of measurement parameter is extracted and data are grouped according to unique value parameters - in this case length of the measurement zone and the position sensor has constant value in all extracted data, so they are shown in `PlotLabel`. Angle and Elongation parameters vary and their specific combinations are shown in the legend. Very simply the command was adapt for different scenarios and resulting plots were visually compared quickly.

```
SignalData["RoteLoops", {"File", "Angle", "Elongation", "ZoneLength", "SensorPos"}, "",
("Quantity" == "MeanForce") && (3<"Elongation"<65) && ("ZoneLength"<50) && ("SensorPos"==
"Ss1")]
```

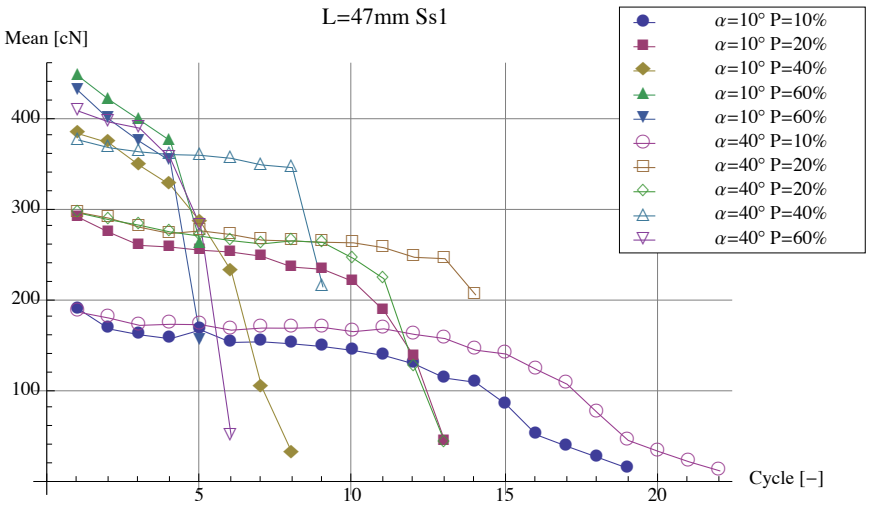

Fig. 2: Example of the plot based on the result of `SignalData` call

Conclusion

One possible implementation of user-defined data source was shown. It is fully working and can be easily used for any set of repeated experiments with some unified but still quite general structure. It has been used in our company for 5 years already and we found no fundamental limitation of it.

We believe some implementation of user-defined and user-handled datasources would be very useful in a general and official form. Together with new cloud integration and Wolfram Alpha algorithms, it could create very strong information system for specific research both in academy and industry.

Acknowledgments

This research has been supported by Czech Ministry of Education, Youth and Sports in research program L012 - NPU I - National program of sustainability.