SQLTags

Guide to Driving External SQLTags

The goal of this document is to outline the fundamental concepts of how SQLTags work, and the steps required to create your own SQLTags driver.

SQLTags, introduced into Inductive Automation software in 2007, turns any SQL database into a high-performance tag database. It is a representation of hierarchical tag data stored and communicated through a set of static-schema tables in the database. Each tag is executed by a driver, who updates the database with its current value. Providers monitor the tags for value and configuration changes, and use the tag data for any purpose – realtime status, historical logging, etc.

Originally, SQLTags provided a way for FactorySQL, an OPC client, to share tag data with FactoryPMI, a Java based visualization system, which couldn't access OPC directly. Multiple FactorySQLs could act as tag drivers to drive data from various data sources or locations into a single database, monitored by FactoryPMI, thus creating a unified aggregate tag model.

Ignition, released in 2010, introduced a new form of internal SQLTag provider, more appropriate for the "all-in-one" architecture. However, this is just another element in the toolkit. SQLTags in their traditional form continue to play an important role in enabling highly distributed architectures. They exist in the form of the "Database Provider", which acts like FactoryPMI, and the "Database Driving Provider", which plays the role previously filled by FactorySQL. Using a combination of these "realtime providers", it is possible to create architectures identical to those created with FactorySQL and FactoryPMI.

In addition to creating distributed architectures, the "external", or database-centric, SQLTags providers offer an excellent point of extensibility for the Ignition platform. Given the open, accessible nature of SQL databases, creating a custom SQLTags driver is one of the easiest ways to make data available to the Ignition platform from a custom/3rd party application.

Basic Concepts and Data Flow

SQLTags operate through 9 tables created in the database.

Tag Configuration Tables

1) **sqlt_core** – The core tag information table, has one entry per tag. Defines fundamental properties like data type, as well as the current value of the tag. Is monitored by the provider to determine value and configuration changes.

2) **sqlt_meta** – Provides additional properties for tags. Only consulted when tag configuration has changed.

3) **sqlt_as** – Provides alert state configuration for tags which utilize alerting.

4) **sqlt_perm** – Provides custom permission settings for tags set to use them.

Operations Tables

5) **sqlt_sc** – Contains the definitions of scan classes, which dictate how tags are executed.

6) **sqlt_sci** – Contains an entry for each scan class from sqlt_sc, for each driver currently driving tags. Used to verify that drivers are properly executing.

7) **sqlt_drv** – Contains an entry for each SQLTags driver. Only really used for browsing, not necessary for drivers who will be creating tag configurations themselves.

8) **sqlt_err** – Contains errors that have occurred executing tags. May be used by drivers to record errors, but is now generally considered deprecated with Ignition.

9) **sqlt_wq** – The "write queue". All write requests are entered into this table, where the driver will detect and execute them. The result will be written back by the driver, and will be noticed by the provider.

Tag Execution Concepts

Polling – Many operations require polling of the database by either the driver or the provider. To ensure efficiency, all polling operations utilize indexed timestamp fields. This allows the database to do very little work when nothing has changed.

Tag Configuration – Tags are configured by inserting or modifying the appropriate entries in the configuration tables above. Configuration change is signaled to the provider by updating the "configchange" of sqlt_core to be the current time. Deleting a tag works by setting it's "deleted" column and then "touching" config change. This will inform all drivers and providers to remove the tag from memory. At some point later, a daemon will delete the tag information from the database.

Tag Execution, drivers – Each tag has a "drivername" property that indicates which driver is responsible for executing it. Other drivers and providers with different names will treat the tag as an "external" tag – a tag driven by a different entity – and will only monitor its value.

Tag Execution, scan classes – Each tag is assigned to a scan class. The idea is that scan classes will define how often the tag should execute, as well as provide more advanced options like leased and driven execution. In reality, the tag driver is free to execute tags as it desires, but it is important to understand how the scan classes and the sqlt_sci table are expected to work, as that is how the provider will verify that the tags are being executed.

Tag Monitoring – Both providers and drivers generally monitor tag value and configuration changes. In general, the entities will monitor tags whose "drivername" isn't equal to their own, which for providers means all tags, since providers don't have a driver name. Monitoring occurs by selecting the tag values (or any information desired) from the appropriate table where one of the indexed timestamp columns is greater than the last checked time. The provider/driver will then store that time in memory as the last check, and will use it in the next poll.

Implementing a Simple Driver

This section will describe the process to implement a very simple driver in order to expose data from an external data source. The driver will manage the tag configuration, that is – it will insert the tags as necessary. It is assumed that the driver will be reflecting internal tag structure into the database, and not loading tag configuration from there. Therefore, it will not store configuration information using meta properties.

Once the tags are inserted, it will update their values. Additionally, the driver won't implement complex scan classes – all tags will be executed under a single scan class. The driver is not constrained to updating tags at the rate of the scan class – it can update them at any rate it wants. The scan class, however, is a necessity and is used to verify that the driver is running.

Note: Despite the fact that this section is presented before the reference information, it assumes a familiarity with that information. Therefore, it will be beneficial to first become familiar with the table structures and enumeration values in that section.

Step 1: Setting up the database and Ignition
In order to get started, you'll need the tables to exist in the database, and a tag provider in Ignition to watch your tags. While you will only need a "Database Provider" to monitor the tags, only the "Database Driving Provider" creates the table. Since the Driving Provider also acts as a standard provider, you can just use that. One caveat: The "Driving Provider" is provided by the SQL Bridge module. If you do not have the SQL Bridge module installed, you'll need to create the tables through a different means, such as by using FactorySQL, or by exporting the tables from a different database.

Step 2: Create the required supporting elements
Create a new scan class entry in the sqlt_sc table, with a mode=0, lorate=1000, staletimeout=10000 and configchange=CURRENT_TIMESTAMP (or appropriate keyword for your database).

As mentioned above, the lorate is arbitrary, and does not constrain how often the values may be updated. However, we must make note of the rate, because it will affect how often we update the sqlt_sci table.

Note the id of the new scan class for use when creating the tag.

Create a new entry in the sqlt_drv table for your driver name. Leave ipaddr and port blank, as the driver does not support TCP/IP browsing.

Step 3: Create the Core Tag
Create a row for your tag in the sqlt_core table, with the appropriate name, path, and drivername values that you want. Set the correct datatype for your tag. For tagtype, use the value of 1, for "DB Tag". You will be modeling your tags as "db static tags", which to the providers appear to be SQLTags whose values can be updated from an outside source.

Set enabled to 1 (true), and set scanclass to the id created in the step above.

Since the tag meta properties contained in the sqlt_meta table are normally used by the driver to define the tag, and in this simple example we already know exactly how to execute the tags, it is not necessary to enter anything in this table.

Set configchange of sqlt_core to CURRENT_TIMESTAMP to notify the watching providers that the tag has changed.

Step 4: Execute the Tag
Begin collecting data for the tag. To notify the system of a value change, update the appropriate value column (intvalue, floatvalue, datevalue, stringvalue) in sqlt_core, and "touch" valuechange (set to CURRENT_TIMESTAMP).

In order to tell the provider that execution is occurring correctly, you must update an entry in sqlt_sci that corresponds to your scan class and driver. Set lastexecrate=scan class rate. After creating the row, update lastexec to be the current time, and nextexec to be (current time + scan class rate), and continue to update the row with these values at each scan class interval. If you fail to update these values within the stale timeout of the scan class, the provider will determine that your driver is not running, and will set all tags from your provider to "stale quality."

Advanced Topics

The goal of this paper is to illustrate how to drive simple values from a 3rd party application, for example, from a python application collecting data from serial. It therefore bypasses much of the normal work flow of SQLTags, where the user browses tags through the driver by way of a TCP/IP mechanism, and then defines tags in the designer, with properties for alarming, scaling, etc. which the driver then executes.

The next steps for creating a more robust driver would be to potentially monitor configuration changes to the tags, and to monitor write requests to the tags. As it stands, the user could potentially try to modify the tags, which would either be ignored or perhaps in worse case cause your driver to re-insert the tags as it expects to see them. Any write requests will go into the sqlt_wq table, and will time out.

Table Reference

Keep in mind that most of the properties in these tables will not be used for the simple driver example specified in this document. In general, all integer time values are in milliseconds.

sqlt_core

Column	Data Type	Notes
id	integer	Auto-incrementing, unique id for the tag
name	string	Name of tag
path	string	Folder path, in form of "path/to/"
drivername	string	Name of driver responsible for executing tags.
tagtype	integer / TagType enum	The type of tag – ie. OPC, DB, etc.
datatype	integer / DataType enum	The type of data provided by the tag.
enabled	integer (0 or 1)	Whether the tag is enabled for execution.
accessrights	integer / AccessRights enum	Access permissions for the tag.
scanclass	integer	ID of the scan class for the tag.
intvalue	integer	Value column used if tag has integer data.
floatvalue	double	Value column for float/real data.
stringvalue	string	Value column for string data.
datevalue	datetime	Value column for date data.
dataintegrity	integer / DataQuality enum	Current quality of the value.
deleted	integer (0 or 1)	Whether the tag is deleted or not.
valuechange	datetime	The last time that the value changed.
configchange	datetime	The last time that the tag's config changed.

sqlt_meta

Column	Type	Notes
tagid	integer	ID of tag that the property belongs to.
name	string	The well-known property name.
intval	integer	Value, if property has integer type.
floatval	double	Value, if property has float type.
stringval	string	Value, if property has string type.

sqlt_as

Column	Type	Notes
id	integer	Unique id of alert state
statename	string	Name of alert state
severity	integer / Severity Enum	
low	double	Low setpoint
high	double	High setpoint
flags	integer / Alert Flags	Flags that dictate how the state acts.
lotagpath	string	Path to tag that provides low setpoint, if low driven flag is set.
hitagpath	string	Path to tag that provides high set-point, if high driven flag is set.
timedeadband	double	Time deadband value
timedbunits	integer / TimeUnits enum	Time deadband units.

sqlt_perm

Column	Type	Notes
tagid	integer	ID of the tag the per-mission belongs to.
rolename	string	Name of the role that this permission is applied to
accessrights	integer / Access Rights enum	Access rights for the given role on the given tag.

sqlt_drv

Column	Type	Notes
name	string	Name of the tag driver
ipaddr	string	Address of browse server, blank or null if browsing isn't available.
port	integer	Port of browse server

sqlt_sc

Column	Type	Notes
id	integer	Auto-incrementing unique id.
name	string	Name of the scan class
lorate	integer	The slower rate to run at, in milli-seconds. Only rate used if scan class mode is "direct".
hirate	integer	Higher rate, in MS. Only used if scan class is "driven" or "leased".
drivingtagpath	string	Path to tag to watch if mode is "driven".

comparison	integer / Comparison enum	Operation to apply to driving tag in driven mode.
comparevalue	double	Value to compare driving tag to for driven mode.
mode	integer / Scan class mode enum	The mode of the scan class.
staletimeout	integer	Time, in milliseconds, before scan class is determined to not be running.
leaseexpire	datetime	The time that the lease should expire, if using “leased” mode.
configchange	datetime	The last time that the scan class has been modified.
deleted	integer (0 or 1)	Whether the scan class has been deleted.

sqlt_sci

Column	Type	Notes
sc_id	integer	The id of the scan class represented.
drivername	string	The driver executing this instance.
lastexec	datetime	Last time that the scan class executed.
lastexecrate	integer	The rate of the scan class at last execution.
lastexecduration	integer	Time, in ms, that the scan class took to execute.
lastexecopcwrites	integer	Writes to OPC performed during last execution.
lastexecopcreads	integer	Value updates from OPC processed in last execution.
lastexecdbwrites	integer	Writes to DB performed during last execution.
lastexecdbreads	integer	Value updates from the database processed during the last execution.
lastexecdelay	integer	The delay between when the scan class should have ran and when it actually ran for the last execution.
avgexecduration	integer	The average duration time of the scan class, in ms.
execcount	integer	The number of times the scan class has executed.
nextexec	datetime	The next time that the scan class should execute.

sqlt_wq

Column	Type	Notes
id	integer	Auto-incrementing unique id for the write operation
tagid	integer	ID of the tag to write to.
intvalue	integer	Value, if tag has integer data type.
floatvalue	double	
stringvalue	string	
datevalue	datetime	
responsecode	integer / Write Response enum	The state of the write request. When created, the reponse code should be set to 2 – Pending
responsemsg	string	Write error if operation failed.
t_stamp	datetime	The time that the write request was created.

sqlt_err

Column	Type	Notes
objectid	integer	ID of the object with the error
objectype	integer / Object Type enum	The type of object. Used with objectid to identify the item that caused the message.
lifecycleid	integer / Lifecycle Enum	When the message was generated
msgtype	integer / Message Type enum	
errormsg	string	The primary message
stack	string	Additional error information
t_stamp	datetime	When the message was generated.

Enum Reference

Enums are well-known values that are stored as integers in the database.

Tag Type

0	OPC Tag
1	DB Tag
2	Client Tag
6	Folder Tag

Data Type

0	Int1
1	Int2
2	Int4
3	Int8
4	Float4
5	Float8
6	Boolean
7	String
8	DateTime
9	DataSet

Data Quality

0	Bad Data from OPC
4	CONFIG_ERROR
8	NOT_CONNECTED
12	DEVICE_FAILURE
16	SENSOR_FAILURE
20	Bad, showing last value
24	COMM_FAIL
28	OUT_OF_SERVICE
32	WAITING
64	UNCERTAIN
68	UNCERTAIN, showing last value
80	SENSOR_BAD
84	LIMIT_EXCEEDED
88	SUB_NORMAL
28	SERVER_DOWN
192	Good Data
216	Good, with local over-ride
256	OPC_UNKNOWN
300	Config Error

301	Comm Error
310	Expr Eval Error
330	Tag exec error (fsql)
340	Type conversion error
403	Access Denied
404	Not Found
410	Disabled
500	Stale
600	Unknown (loading)
700	Write Pending

Access Rights

0	Read Only
1	Read/Write
2	Custom

Scan Class Modes

0	Direct
1	Driven
2	Leased

Comparison Mode

0	Equal
1	Not Equal
2	Less Than
3	Less Than Equal
4	Greater Than
5	Greater Than Equal

Alert Flags

0x01	Low Exclusive
0x02	Low Infinite
0x04	High Exclusive
0x08	High Infinite
0x10	Any Change
0x20	Low Driven
0x40	High Driven

Write Response

0	Failure
1	Success
2	Pending

Meta Properties

While these properties aren't used in this document, they are included here for completeness.

Property Name	Data Type	Default
OPCServer	String	""
OPCItemPath	String	""
OPCWriteBackServer	String	""
OPCWriteBackItemPath	String	""
ScaleMode	Integer	0
RawLow	Double	0
RawHigh	Double	100
ScaledLow	Double	0
ScaledHigh	Double	10
ClampMode	Integer	0
Deadband	Double	0.0001
FormatString	String	"#,##0.##"
EngUnit	String	""
Tooltip	String	null
EngHigh	Double	100
EngLow	Double	0
Documentation	String	""
Expression	String	""
ExpressionType	Integer	0
AlertMode	Integer	0
AlertAckMode	Integer	1
AlertSendClear	Integer	1
AlertMessageMode	Integer	0
AlertMessage	String	""
AlertDeadband	Double	0.0001