

Ignition 7.9 User Manual — SQL in Ignition

Generated 2026-10-12 from
<https://www.docs.inductiveautomation.com/docs/7.9/>

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

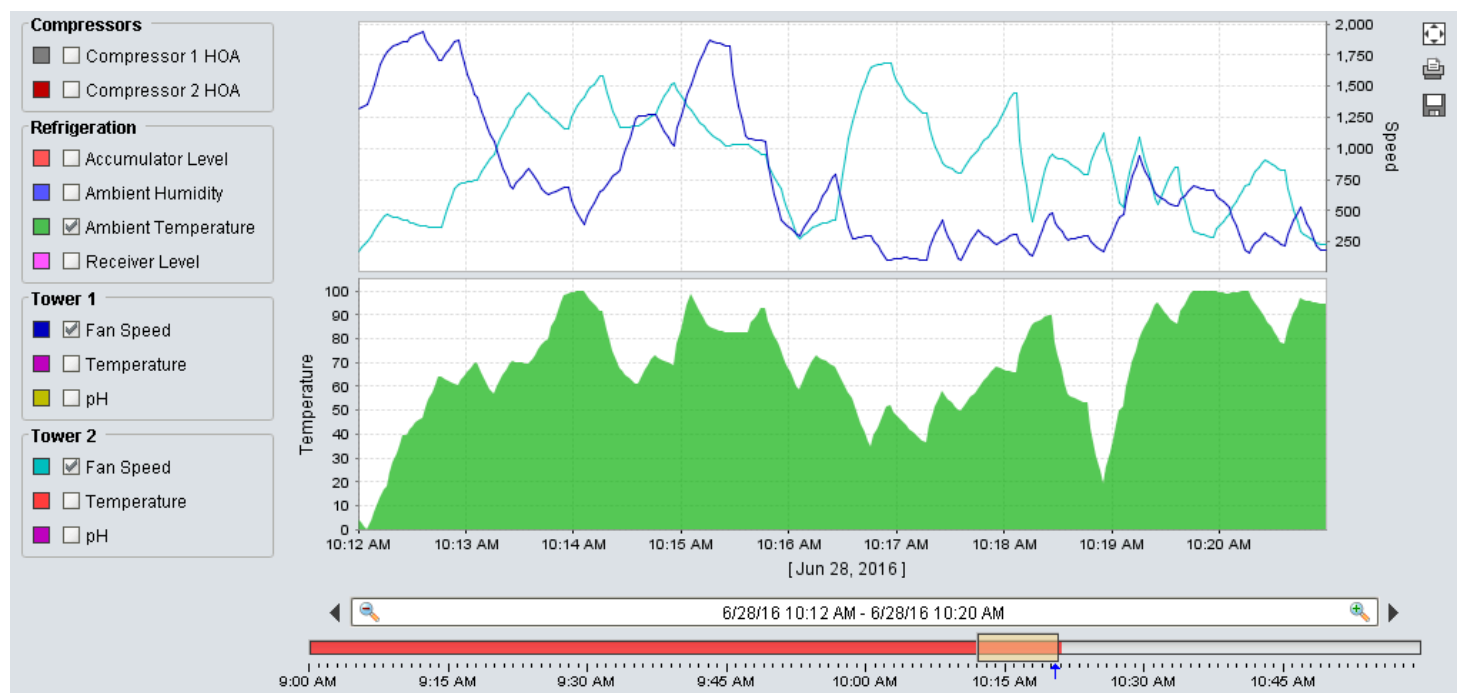
SQL in Ignition

Leveraging Databases

In addition to all the normal HMI functionality you would expect, Ignition has the ability to connect to databases, and this can greatly increase the functionality available to you! You can use databases to store history, create easy to search lists and configurations, and retrieve data from ERP or other systems. When you start using SQL, you can expand your project from a simple HMI to a project that brings your whole process together. The best part is that Ignition connects to as many databases as you want, all from one central location. Because the database lives outside of Ignition, we don't maintain any control or rules over what you can do with your data.

Displaying Data

You can easily display information from your databases on a window along with anything else in Ignition. You can show parts lists, step sequences, **realtime or historical charts**, add the ability to search for inventory, or anything else you can think of.



What is SQL?

SQL stands for "Structured Query Language" and is the backbone of most modern relational databases. It's often referred to as "S.Q.L." or "Sequel," but both are correct and widely recognized. This language (different than the [Python Scripting Language](#)) allows you to write out requests or "queries" against the existing data to **view, add, edit, or remove** the information you want in a simple format.

Everything in a relational database is based around tables. Tables store the basic information for any system and can be combined together to make very efficient queries to retrieve your data.

	id	name	description	location	qty
▶	576	Mold 1	Mold for part # 1523	Warehouse A	23
	577	Mold 2	Mold for part # 8825	Warehouse A	12
	578	Bin 1	Finished Goods Bin 1	Warehouse B	9
	579	Bin 2	Scrap Bin 1	Warehouse B	14

```
SELECT *  
FROM inventory  
JOIN location on location.name = inventory.location;
```

SQL Queries

SQL queries are crucial to Ignition's database-centric model. Queries can **show what is available** or **alter data in the databases**, and some companies have positions just dedicated to running databases and creating queries. Anywhere Ignition is fetching data, you can choose to use your own custom queries to get exactly what you want out of the database. You can make your queries as simple or complex as you like. If your database is large, you might have a whole team dedicated to creating these queries for you and Ignition will happily execute them.

SQL - Select Statement - Select Data from a Table

```
SELECT * FROM mytable
```

SQL - Inner Join Statement - Selects Records that have Matching Values

```
SELECT users.id, users.firstname, users.lastname, roles.name as 'rolename'  
FROM users  
INNER JOIN mapping ON users.id = mapping.userid  
INNER JOIN roles ON mapping.roleid = roles.id  
WHERE roles.name = 'Administrator'
```

Database Connections

Any SQL query you use needs a **Database connection**, but Ignition simplifies all that by creating database connections in the Gateway instead of in the clients. This means from one central location you can manage all your database connections, and you don't have to worry about planning around adding clients in the future. Any special rules or connection restrictions are taken care of in the Gateway.

Using SQL in Ignition



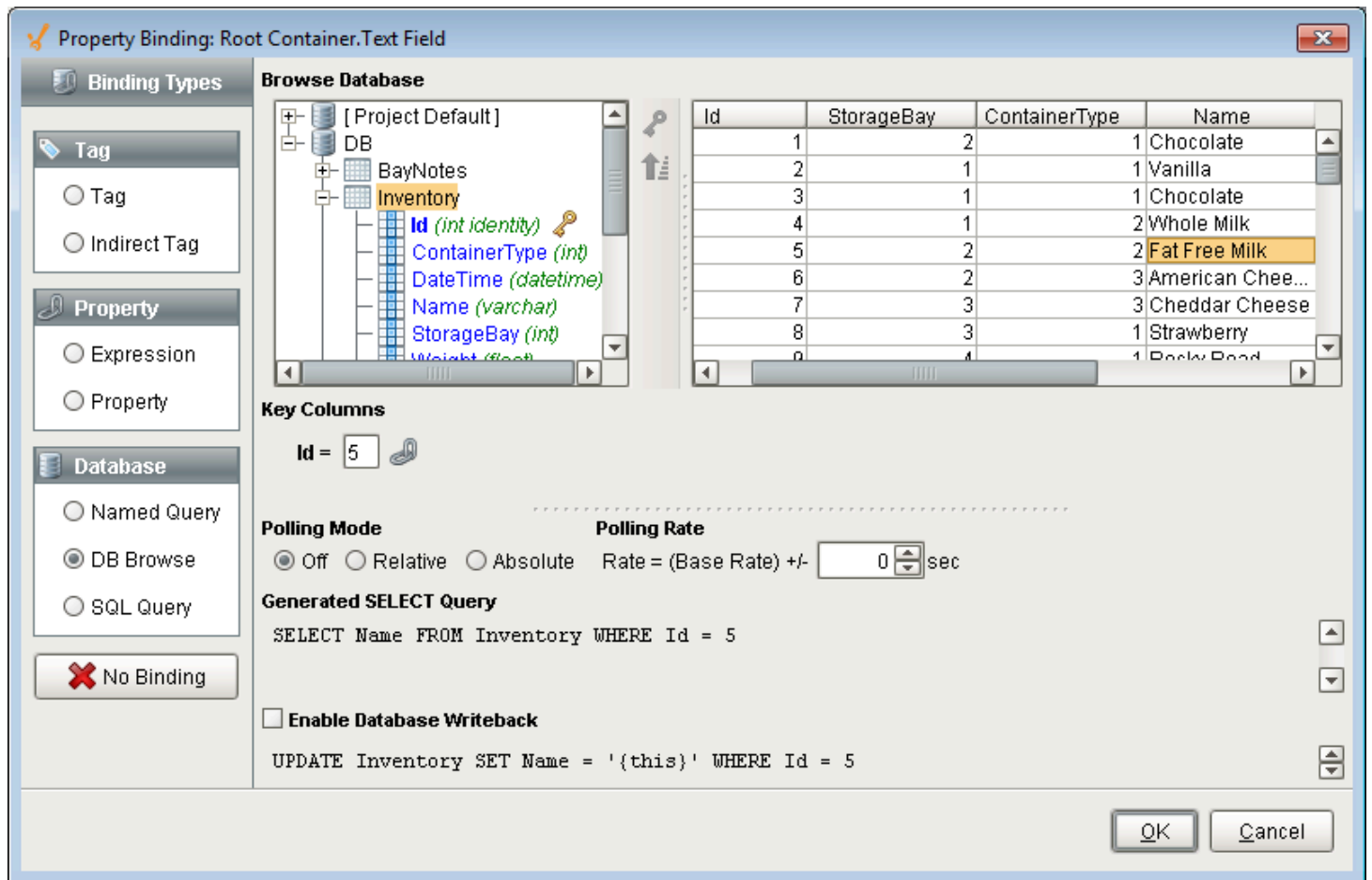
INDUCTIVE
UNIVERSITY

Querying Data from Database

[Watch the video](#)

There are many types of queries, and many ways to use them in Ignition. Some provide an easy to use builder to automatically store or fetch data, and some allow you to completely customize your queries.

Queries in Bindings Ignition's binding system offers a lot of flexibility in how database data can be used in a binding. The **Named Query binding** allows you to select one of the Named Queries that were previously built for that project, offering a very secure method of pulling data from the database. The **DB Browse** builder provides an interface that will build the query based on the data in the table that was selected. This allows even users with little SQL knowledge to pull data from the database. Finally, the **SQL Query** option will accept a straight query, so that a query specific to that binding can be written directly in the binding. When binding to a basic (non-dataset) data type, you can use the Writeback feature directly to send any changes back to the database.



Queries in Scripting

Ignition offers a number of built-in scripting functions for you to use to query your databases. This makes it very simple to view data, create dynamic scripts that use real data, and more. You can pull individual pieces of information, return whole tables of data, or update your database directly. Depending on the type of query and the sort of results you want, you will use different functions. The following functions are the ones you will use most, and all of them can use a special placeholder (?) to allow for dynamic query building.

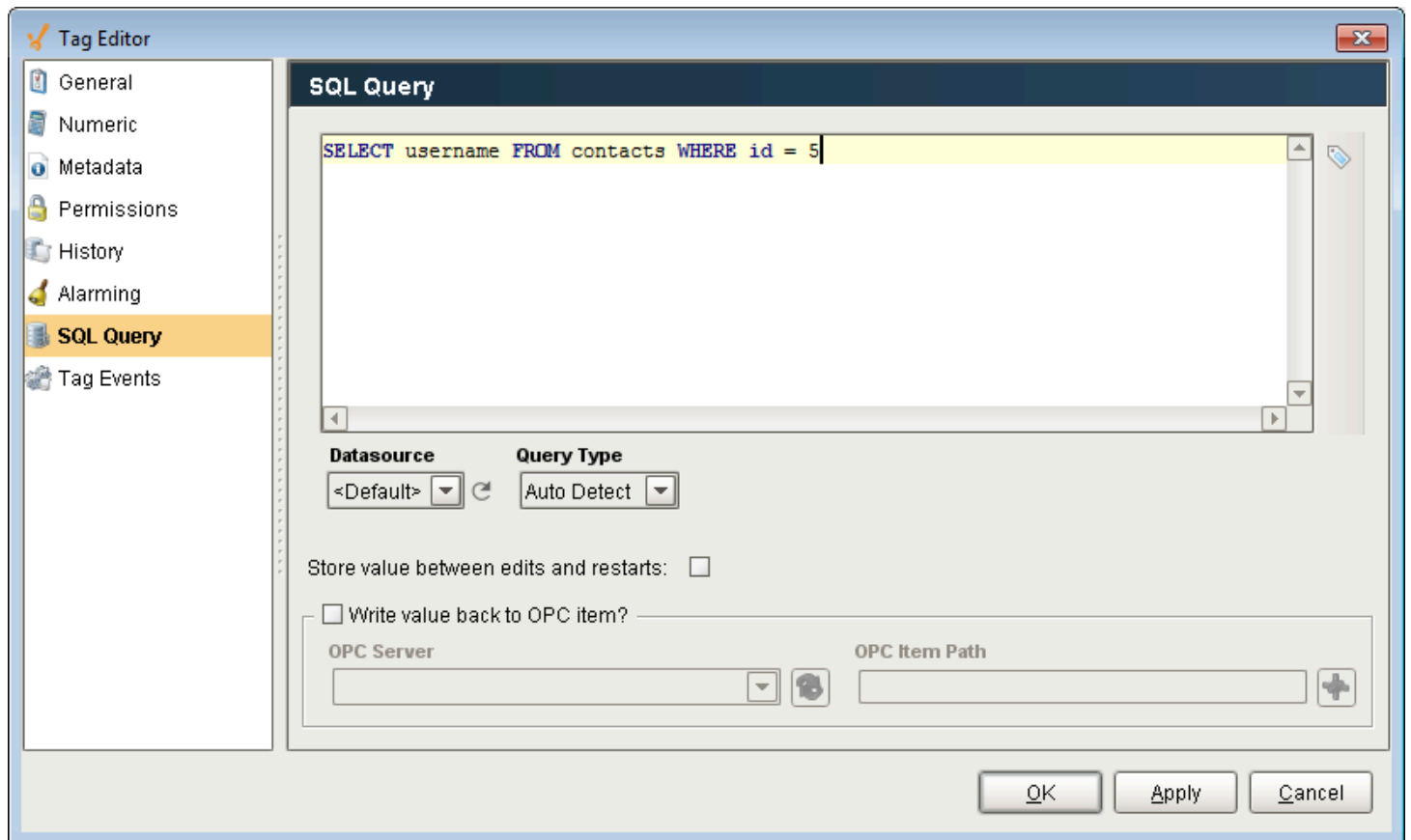
Scripting Function	Description
<code>system.db.runNamedQuery()</code>	Used to run a previously setup Named Query from within a script.
<code>system.db.runPrepQuery()</code>	Used to run basic SELECT queries to fetch whole datasets. This can be used to populate tables, or to sift through the data to do your own calculations.
<code>system.db.runPrepUpdate()</code>	Used to run queries that change the data in the database. Usually used on input form windows to update your database records.
<code>system.db.runScalarPrepQuery()</code>	Used when you want only one value from your results. Perfect for fetching a single value like the highest ID, or first timestamp of a result set.

Each of the different functions takes in different arguments (values) and provides slightly different options and functionality. For example, the `runPrepUpdate()` can return the auto-generated key from insert queries. This can be extremely helpful and eliminate the need to hit the database multiple times if you are using linked tables.

You can find examples of each of these and all the other database functions in the **system.db** section of the appendix.

Queries in Tags

Ignition offers **Query Tags**, which can run queries and return the result as a Tag value, giving all of the projects in the Gateway access to the same Database values.



Queries in Reports

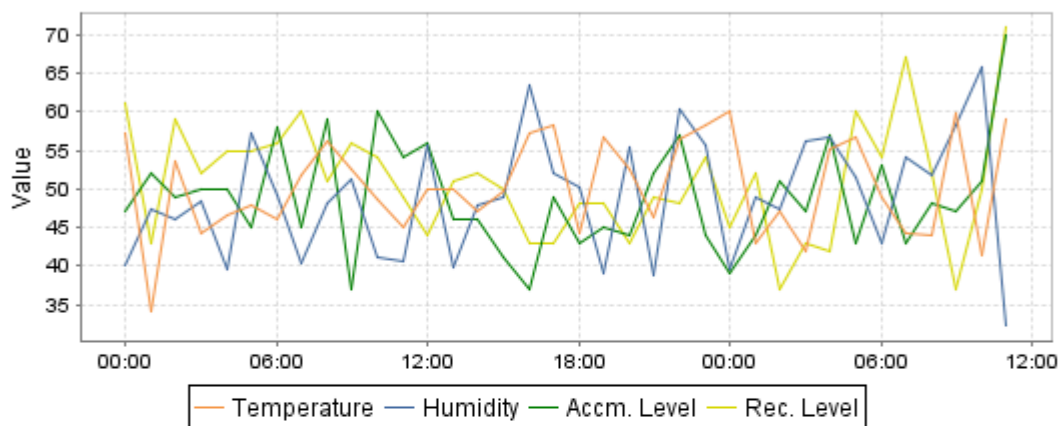
You can leverage queries to access data from all database connections to [create rich reports](#), from simple data logging to complex reports with grouped charts and datasets.

Tag History Report

A report that queries tag history and displays a graph and table

Report Start: 6/27/2016 12:00 AM

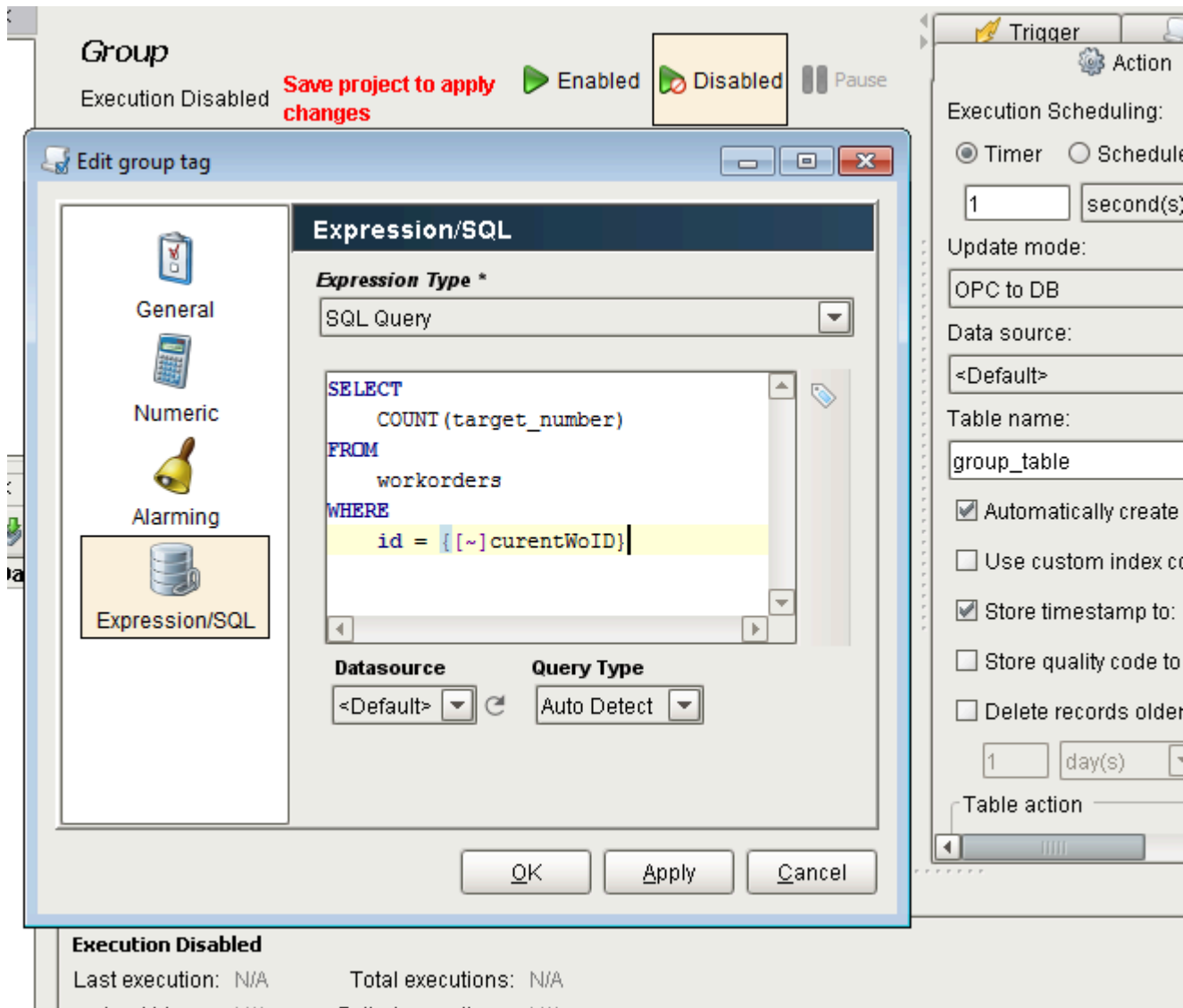
Report End: 6/28/2016 11:00 AM



Date / Time	Temperature	Humidity	Accm. Level	Rec. Level
6/27/2016 12:00:00 AM	57.34	40.14	47	61
6/27/2016 1:00:00 AM	33.98	47.33	52	43
6/27/2016 2:00:00 AM	53.61	46.15	49	59
6/27/2016 3:00:00 AM	44.3	48.5	50	52
6/27/2016 4:00:00 AM	46.57	39.59	50	55
6/27/2016 5:00:00 AM	47.9	57.16	45	55
6/27/2016 6:00:00 AM	46.04	49.16	58	56
6/27/2016 7:00:00 AM	51.87	40.42	45	60
6/27/2016 8:00:00 AM	56.24	48.07	59	51
6/27/2016 9:00:00 AM	52.42	51.17	37	56
6/27/2016 10:00:00 AM	48.74	41.01	60	54
6/27/2016 11:00:00 AM	44.89	40.67	54	49
6/27/2016 12:00:00 PM	49.85	55.73	56	44
6/27/2016 1:00:00 PM	49.82	39.84	46	54

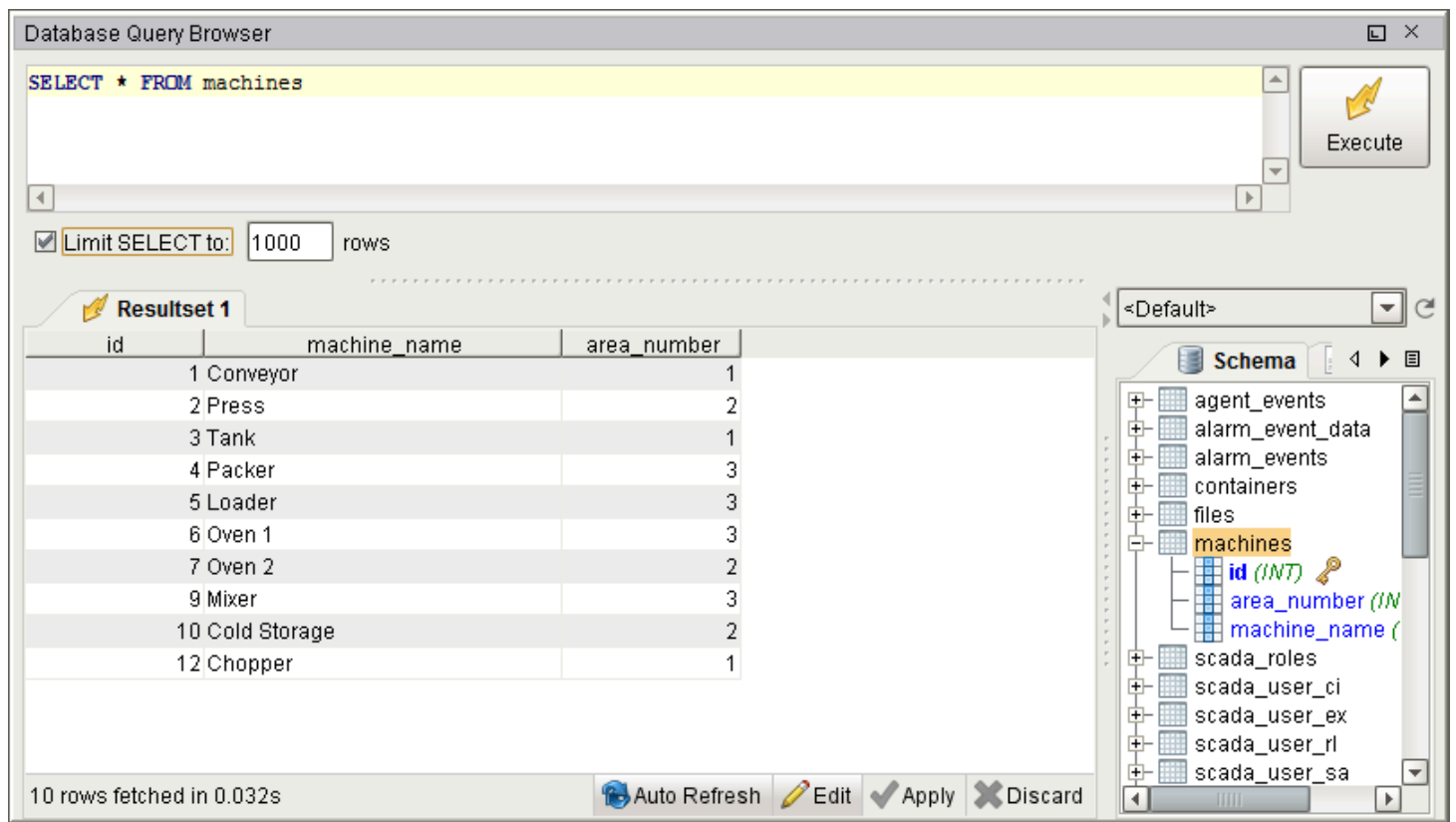
Queries in Transaction Groups

While **Transaction Groups** are great at storing Tag data to a Database automatically, the built-in Expression Items can execute a SQL Query within the Transaction Group.



Database Query Browser

The **Database Query Browser** offers an easy to use environment to run queries in for testing. Here, queries can be tested to figure out what values get returned, or data can be updated through raw queries, or the Database Query Browsers easy to use GUI editor.



Auto Generated Queries

Many systems within Ignition utilize a database connection, but the queries that are executed are constructed automatically by the system and do not require you to build the queries manually. These systems such as the **Tag Historian**, the **Alarm Journal**, or the **Database User Source** are very easy to setup and use since each system will automatically generate the necessary tables in the database, insert the relevant data, and even has prebuilt tools to extract the data. However, it is important to note that while these systems can automatically generate queries for you using the various components, these systems are simply storing data in a database which you can manually query out by building your own SQL queries.

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Writing SQL Queries

SQL Tables

! INFO

For information about databases and how to get connected, see the [Database Connections](#) section.

The foundation of every database system is a table. Every database consists of one or more tables, which store the database's data/information. Each table is identified by a name (for example `Customers` or `Orders`), and consists of column definitions and rows of data.

The database table columns have their own unique names and have pre-defined data types. Table columns can have various attributes defining the column functionality (such as the primary key, index, default value, and so on).

While table columns describe the data types, the table rows contain the actual data for the columns.

ID	Name	Address
1	Safeway	123 Safeway Lane
2	Costco	456 Costco Way
3	Target	789 Target Ave

Current common databases are almost all Relational Databases. This means that their tables can relate to each other by including an ID in one table that matches the key of another. These are called foreign keys.

Primary Key

A primary key is a way to uniquely identify each row in a table. While it is possible to create a database table without a primary key, it is highly recommended to configure one for each table. A primary key is comprised of either a single column, or set of columns. When multiple columns are specified as a primary key, this is known as a **composite primary key**. No two distinct rows in a table can have the same value (or combination of values) in those columns.

While Primary Keys can be configured in several ways, they typically meet the following criteria:

- **Integer Datatype:** The datatype of the key column is typically an integer, and not a varchar. The primary key is only an identifier to a specific row in a table, so an integer datatype can easily be used. Some databases support a UID or a UUID (Universally Unique Identifier) that looks like a character string, but is something specially made for primary keys.
- **Automatically Incrementing:** The value of the primary key increments as rows are added. The key is usually configured to automatically increment in the database so that external applications (such as Ignition) don't have to figure out the next available value when inserting a new row.
- **Statically Defined:** Any row that is inserted must fill in these value(s) without creating duplicates. Configuring the primary key as automatically incrementing means that the database will **automatically** handle this criteria.
- **Non-NULL:** NULL (empty) values should not be present in the primary key. This column (or columns) will usually not allow NULL values.

Index

Indexes speed up the querying process by providing swift access to rows in the data tables, similarly to the way a book's index helps you find information quickly within that book. Indexes are extremely important when querying large sets of data. You should create an index for the set of columns you use commonly in a `WHERE` clause. For example, you should add an index on the timestamp column of a historical table when querying the table by a start and end date. Ignition does this automatically when it creates tables for Tag History or Transaction Groups.

Foreign Key

A Foreign Key is a referential constraint between two tables. The foreign key identifies a column or a set of columns in one (referencing) table that refers to a column or set of columns in another (referenced) table. The columns in the referencing table must be the primary key in the referenced table. For example, you might have a list of suppliers with an integer ID column. Then your invoices would use the supplier's ID instead of the name. These linked tables can save a lot of space because you don't have to include everything about the supplier in each invoice.

Example

Supplier (**SupplierNumber**, Name, Address, Type) Invoices (InvoiceNumber, **SupplierNumber**, Text, Cost)

Null Value

NULL is a special marker used in SQL to indicate that a data value does not exist in the database. This way it is clear that there is no data, instead of guessing if a value of 0 is correct or just missing data. By definition, NULL is not equal to anything, even other NULL values. Every database has a function to identify if a value is NULL, usually called `isNULL()` or something similar.

Comments

Comments can be added to any SQL query just like with scripting. Single line comments are done with two dashes and a space: `--`

SQL - Single Line Comment

```
-- This is a single line comment in a SQL query.  
SELECT * FROM my_table
```

You can also do multi line comments by wrapping text within a forward slash and an asterisk: `/* text */`

SQL - Multi Line Comment

```
/* A multi line comment  
can span multiple lines. The  
comment will stop when it is closed  
with another asterisk and forward slash.*/
```

SQL Queries

SQL queries (or statements) are used to create, maintain, and view relational databases like MySQL, SQLServer, Oracle, etc. They follow a specific format and use just key words to determine the structure of the query. Unlike most coding languages, SQL does not rely on newlines or start/end markers for it's format, each query is a single line of code. You will often see SQL queries split over several lines, but that is just to make them easier to read.

! INFO

You might notice a lot of CAPITALIZED words in SQL queries. While these key words are not case sensitive, it is still common practice for people to capitalize them in a query. Things like `SELECT`, `FROM`, `WHERE` (and a few others) are almost always capitalized because they have a

special meaning. Try not to have table or column names that use any of these special words. You will see this capitalization format in examples throughout this user manual and other online references.

Select Command

The SELECT statement is used to select data from a database. The result is returned as a data set, called the result set. This is true even if there is only one value returned. The syntax for a SELECT statement is as follows:

SQL - The Select Command

```
SELECT myColumn FROM myTable
```

Select Examples

The "*" character can be used to specify all columns from a database table. While this is the easiest way to retrieve results from a table, this is not the recommended approach.

SQL - Selecting All Columns

```
SELECT * FROM Customers
```

The recommended approach is to instead specify only the columns that are required for a query. There are several reasons for this, but performance would be the main one: less columns in a statement means less work for the database, and the resulting data set in Ignition will use less memory.

SQL - Selecting a Single Column

```
SELECT Name FROM Customers
```

SQL - Selecting Multiple Columns

```
SELECT Name, Address FROM Customers
```

Where Clause

The WHERE clause is used in conjunction with other commands to extract only those records that fulfill a specified criterion. The WHERE clause usually goes near the end of the query followed by a condition that the values must meet to be returned.

Pseudocode - Where Clause

```
SELECT myColumn FROM myTable WHERE condition
```

The WHERE clause can use various operators for its condition, with the basic operators being:

Operator	Description
#ERROR!	Equal to.
<>	Not equal to.
>	Greater than.
<	Less than.
>=	Greater than or equal to.
<=	Less than or equal to.

Where Examples

Only return customers from CA.

SQL - Simple Where Clause

```
SELECT * FROM Customers WHERE State = 'CA'
```

Only return users over a specified age.

SQL - Select Users over 25

```
SELECT * FROM Users WHERE Age > 25
```

Insert Into Command

The INSERT INTO statement is used to insert a new row in a table. If any columns have default values or are auto-incrementing, they can be omitted from the INSERT query.

Pseudocode - The Insert Command

```
INSERT INTO myTable (column1, column2)
VALUES ('Value1', 'Value2')
```

Insert Examples

SQL - Insert Values into Columns

```
INSERT INTO Customers (Name, Address, City, State, Zip, Country, Phone)
VALUES ('Inductive Automation', '90 Blue Ravine', 'Folsom', 'CA', '95630', United
States, '1-800-266-7798')
```

If inserting a value into every column of the table, the columns do not need to be listed on the INSERT INTO statement. The values just need to be listed in the same order as the columns in the table. The table in the query below has four columns: id, first name, last name, and title.

SQL - Inserting to all Columns

```
INSERT INTO Users
VALUES (5628, 'Bob', 'Smith', 'Project Manager')
```

Update Command

The UPDATE statement is used to update existing records in a table. If a WHERE clause is not used, **all rows in the table will be updated**. As a result, the UPDATE statement should be used in conjunction with a WHERE clause in most cases. Many official management tools like SQLServer's Management studio will not allow UPDATE commands without a WHERE clause.

Pseudocode - The Update Command

```
UPDATE myTable SET myColumn = 'myValue'
```

Update Examples

SQL - Updating All Rows in a Single Column

```
UPDATE Customers SET Name = 'Inductive Automation'
```

SQL - Updating a Single Column for a Single Row

```
UPDATE Customers SET Address = '2110 21st Street' WHERE ID = 1
```

Delete Command

The DELETE statement is used to delete records in a table. **NEVER** run a delete command without a WHERE clause. It will delete **ALL** records from that table. Many official management tools like SQLServer's Management studio will not allow DELETE commands without a WHERE clause.

Pseudocode - The Delete Command

```
DELETE FROM myTable WHERE myColumn = value
```

Delete Examples

SQL - Deleting Based on Column Value

```
DELETE FROM Customers WHERE Name = 'Inductive Automation'
```

SQL - Deleting Rows Based on an ID Column

```
DELETE FROM Customers WHERE id < 538
```

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

SQL Select Options

While the SELECT command in its basic form can be very simple to use, the SELECT statement can be used with other statements or in certain ways that allow you to bring in exactly the data you need.

Selecting Static Values

Static values can be inserted into a resultset returned from a SELECT query as another column. Simply use the static value as a column to select, and the query will return a column where the name of the column is the static value, and every row in that column will return that same static value.

Pseudocode - Selecting Two Columns From a Table, and a Third Column with a Value of 10 for every row

```
SELECT column1, column2, 10 FROM table
```

Select Distinct Command

The SELECT DISTINCT statement works much like a SELECT statement works, in that it selects data from a database. However, SELECT DISTINCT will only return distinct or different values, not duplicates.

Pseudocode - The Select Distinct Command

```
SELECT DISTINCT column FROM table
```

Examples

This can be useful for getting a better idea of the range of values in a particular column.

SQL - Select Distinct Countries

```
SELECT DISTINCT country FROM Customers
```

Order By Clause

The ORDER BY keyword is used to sort the result-set by a specified column set of column. The ORDER BY keyword sorts the records in ascending (ASC) order by default. If you want to sort the records in a descending order, you can use the DESC keyword.

Pseudocode - Order By Clause

```
SELECT column1, column2 FROM table ORDER BY column2 DESC
```

Examples

SQL - Ordering by One Column

```
SELECT * FROM Customers ORDER BY Name ASC
```

You can use multiple columns to sort, this will sort by state first, and for each state the rows will be sorted by name.

SQL - Ordering by Multiple Columns

```
SELECT * FROM Customers ORDER BY State ASC, Name DESC
```

Limiting Rows Returned

SELECT commands can have the number of rows that the query returns limited using a special keyword. The keyword differs between database providers but the effect is the same, limiting the number of rows returned to a value that you specify.

Database	Keyword	Example
MS SQL Server/ MS Access	SELECT TOP value/percent	Note that the SELECT TOP command is unique in that you can specify a percentage value instead of an exact number of records. <pre>SELECT TOP 200 column1, column2 FROM table</pre> <pre>SELECT TOP 10 PERCENT column1, column2 FROM table</pre>
MySQL	Limit value	<pre>SELECT column1, column2 FROM table LIMIT 200</pre>
Oracle DB	ROWNUM ≤ value	Note that the ROWNUM command is unique in that you can use it to identify the row number for any reason, not just limiting the number of rows returned. <pre>SELECT column1, column2 FROM table WHERE ROWNUM ≤ 100</pre>

Aliases

In a SQL query, aliases are used to give columns or even tables a temporary name for that query. Simply place the keyword AS after a column or table, followed by the alias name. If the alias is two words, it needs to be encapsulated in single quotes.

Pseudocode - Aliasing Columns and the Table

```
SELECT column1 AS a, column2 AS 'b c' FROM table AS t
```

Examples

This can be really useful when the table has complex column names.

SQL - Aliasing Complex Column Names

```
SELECT id AS 'Badge Number', name AS 'Employee Name', dob AS Birthday FROM employees
```

This can also be useful when using multiple tables in a query, such as with a JOIN.

SQL - Joining Columns from Two Tables with Aliases

```
SELECT * FROM Contacts AS co JOIN Customers AS cu ON cu.ID = co.CustomerID
```

Union Command

The UNION operator is used to combine the results of two different SELECT statements. This differs from a JOIN in that there does not have to be a relationship between columns. However, both SELECT statements need to select the same number of columns with similar data types in a similar order. So if my first statement selects an int column and then a string column, the second statement needs to do the same. The name of the columns in the resultset will take the name of the columns from the first SELECT in the UNION.

Pseudocode - Union Two Tables

```
SELECT stringCol, intCol FROM table1
UNION
SELECT stringCol, intCol FROM table2
```

Examples

By default, the UNION operator will only select distinct values between the two tables.

SQL - Union of Users and Customers

```
SELECT username FROM users
UNION
SELECT name FROM customers
```

To select all values from both tables, we can use UNION ALL instead.

SQL - Union All

```
SELECT jobTitle FROM jobs
UNION ALL
SELECT position FROM employees
```

Static values can be used in a UNION to help differentiate the rows from each table.

SQL - Differentiating Between Users and Customers

```
SELECT 'User' AS Type, username FROM users
UNION
SELECT 'Customer', name FROM customers
```

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

SQL Where Clauses

Overview

The SQL WHERE clause is utilized to restrict the number of rows impacted by a query. A WHERE clause is commonly utilized in two scenarios:

- In conjunction with a SELECT statement to filter the number of rows returned
- As part of an UPDATE or DELETE statements to restrict which rows are manipulated by the query.

In either scenario, the syntax of a WHERE clause is used the same, and can have multiple predicates:

Pseudocode - Where Clause Syntax

```
SELECT * FROM table WHERE column = value
```

Where-Clauses and Data Manipulation

In most cases, statements that modify the content of a database table via UPDATE or DELETE **should** include a WHERE clause: otherwise the manipulation will be applied to every row. To demonstrate, you typically want to avoid queries that look like the following

Pseudocode - Never do this!

```
UPDATE table SET column = 'This was a horrible mistake'
```

When manually modifying a database table, a good habit to develop involves first writing a SELECT statement. If you can successfully write a WHERE clause that only returns the results you need to modify, then you can simply change the rest of your query to manipulate the table. Thus, we could write a query like the following:

Pseudocode - Select-Statement with a Where Clause

```
SELECT * FROM table WHERE id in (100,101,150,174)
```

If we receive only the results we need to modify in the query above, we can make a simple modification to our query to now delete just the rows we want.

Pseudocode - Delete-Statement with a Where Clause

```
DELETE FROM table WHERE id in (100,101,150,174)
```

And Operator

The AND operator allows you to specify two or more conditions in a WHERE clause, where each condition must be true for the row to be returned.

Pseudocode - Using And to Specify Multiple Conditions

```
SELECT column1, column2, column3 FROM table WHERE column1 > value AND column2 < value AND column3 = value
```

Example

This helps to narrow down the result set even further by adding in additional conditions that must be met. This will only return rows for customers from Germany who are also over 20 years old.

SQL - Customers from Germany Over 20

```
SELECT * FROM customers WHERE country = 'Germany' AND age > 20
```

Or Operator

The OR operator allows you to specify two or more conditions in a WHERE clause, but only one of the conditions need to be true for the row to be returned.

Pseudocode - Using Or to Specify Multiple Conditions

```
SELECT column1, column2, column3 FROM table WHERE column1 > value OR column2 < value OR column3 = value
```

Example

The OR operator can help pull in data from two different subsets in the table. This will only return rows for customers from Germany or customers who are over 20 years old.

SQL - Customers from Germany or Customers Over 20

```
SELECT * FROM customers WHERE country = 'Germany' OR age > 20
```

Not Condition

The NOT condition allows you to specify a condition that must not be met for the row to be returned.

Pseudocode - Using Not to Specify a Condition that Shouldn't be met

```
SELECT column1, column2, column3 FROM table WHERE NOT column1 = value
```

Example

This can be useful for finding all data other than a certain subset. This will return all customers who are not from Germany.

SQL - Customers that are not from Germany

```
SELECT * FROM customers WHERE NOT country = 'Germany'
```

Between Operator

The BETWEEN condition allows you to specify a range of values separated by an AND that the value must be in for a condition to be true. The value can be numbers, text or dates and is inclusive of the first and last values in the range.

Pseudocode - Using Between to Specify a Range of Values

```
SELECT column1, column2, column3 FROM table WHERE column1 BETWEEN value1 AND value2
```

Examples

SQL - Customers that are Between the Ages of 20 and 40

```
SELECT * FROM customers WHERE age BETWEEN 20 AND 40
```

Note that the BETWEEN operator would work similarly to using a greater than or equal condition and a less than or equal condition.

SQL - Customers that are Between the Ages of 20 and 40 with no Between Operator

```
SELECT * FROM customers WHERE age >= 20 AND age <= 40
```

Like Condition

The LIKE condition allows you to specify a condition that must meet a certain pattern. Typically used to compare to string values, the pattern can be built using a combination of characters and the two wildcard values.

- % - Used to specify any number of any characters including zero characters.
- _ - Used to specify exactly one character.

Pattern Examples

Pattern	Meaning	Possible Matches
%a%	Values that have an 'a' in them.	a', 'Inductive Automation', 'almost', 'create'
a'	Values that have an 'a' with exactly one character before and after the 'a'.	bat', 'cat', 'can'
_a%	Values that have an 'a' as the second character.	da', 'saw', 'catcher'
a%t'	Values that start with 'a' and end with 't'.	about', 'at'
%a%%_%'	Values that contain an 'a' with at least 3 other characters after it.	trains', 'airplane', 'canteen'
%a%a%	Values that contain at least two 'a' characters in them.	Inductive Automation', 'separate', 'apart'

Once the pattern has been constructed, it can be used with the LIKE operator to find values that match the specified pattern.

Pseudocode - Using Like to Specify a Pattern of Values

```
SELECT column1, column2, column3 FROM table WHERE column1 LIKE '%a%'
```

Example

The LIKE operator can be used to find all values that match a criteria, such as all countries with 'land' in that name

SQL - Customers that are not from Countries with 'land' in the Name

```
SELECT * FROM customers WHERE country LIKE '%land%'
```

In Condition

The IN operator allows you to specify a subset of values, with the condition that the return match at least one of them. Using an IN operator is similar to using multiple OR operators for the same column.

Pseudocode - Using IN to Specify Multiple Values

```
SELECT column1, column2, column3 FROM table WHERE column1 IN (value1, value2, value3)
```

Examples

The IN can be used as a shorthand way of writing out multiple conditions for the same column separated by OR operators. This would select all values where the country is either Germany, France, or USA.

SQL - Customers from Germany or France or USA

```
SELECT * FROM customers WHERE country IN ('Germany', 'France', 'USA')
```

This would be similar to doing something like this.

SQL - Customers from Germany or France or USA

```
SELECT * FROM customers WHERE country = 'Germany' OR country = 'France' OR country = 'USA'
```

The real power of the IN operator is that instead of specifying static values, an entirely new query can be run to compare values against.

SQL - Customers from Countries that users are also in

```
SELECT * FROM customers WHERE country IN (SELECT country FROM users)
```

Combining Multiple Operators

Multiple AND and OR operators can be combined to specify multiple different conditions that need to be met in order for a particular row to be returned. Additionally, each condition can be simple using the mathematical operators or complex using the conditions listed above. When using AND and OR operators in a WHERE clause, the AND will take precedence, evaluating first before the OR. In the pseudocode below, the row will be returned if either both the first and second conditions are met, or the third condition is met.

Pseudocode - Using Multiple Where Clause Operators

```
SELECT column1, column2, column3 FROM table WHERE column1 > value AND column2 < value OR column3 = value
```

However, the order at which the operators get evaluated can change by placing parentheses around the conditions which should be evaluated first. In the pseudocode below, the row will be returned if both the first condition is met, and either the second or third condition is met.

Pseudocode - Using Multiple Where Clause Operators with Parentheses

```
SELECT column1, column2, column3 FROM table WHERE column1 > value AND (column2 < value OR column3 = value)
```

Examples

We can use complex conditions with different operators to find all customers who are over the age of 50 in a country that has 'land' in the name, or any customers in Germany or France.

SQL - Multiple Complex Conditions

```
SELECT * FROM customers WHERE country LIKE '%land%' AND age > 50 OR country IN ('Germany', 'France')
```

Using parentheses in the same query can drastically change what valid return conditions are. Here, the customer must both be from a country with 'land' in the name, as well as either over 50 or from Germany or France.

SQL - Multiple Complex Conditions

```
SELECT * FROM customers WHERE country LIKE '%land%' AND (age > 50 OR country IN ('Germany', 'France'))
```

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

SQL Table Joins

Overview

The SQL JOIN allows you to run a single SELECT statement that references multiple tables. This can be used for more advanced filtering, as well as combining data from multiple tables in a single result set. The process of joining two tables involves stating both tables in the query, and then specifying that a column from one table relates to another in some way.

Joins may look imposing at first, but they are simply SELECT statements that utilize columns from multiple tables.

The JOIN keyword works in conjunction with a SELECT statement. However, there are some key concepts that must be addressed when attempting to use the JOIN keyword.

Specifying Each Column

When listing column in a statement that uses the JOIN keyword, you must denote which table each column is being retrieved from. You can do this by using an Alias, or with the fully qualified column name. This prevents ambiguous columns in the context of the query, and makes it easier for you to use other keywords in the statement: i.e., adding WHERE clauses that apply to multiple tables.

Pseudocode - Fully Qualified Column Name

```
table_name.column_name
```

Declare the Relation

After the JOIN keyword, you must state which columns from each table relate to each other. This is accomplished by stating the name of the table, using the ON keyword, and then stating that a column on the first table is equal to a column on the second table. The columns specified are typically primary keys for their respective tables.

In the example below, we're stating that the values in **some_column** on **tableA** should be associated with matching values in **some_other_column** on **tableB**.

Pseudocode - Using the Join Keyword

```
FROM
    TableA
JOIN tableB ON tableA.some_column = tableB.some_other_column
```

The order you present the columns **after** the ON clause does not matter: A = B is equivalent to saying B = A, so we could switch the columns listed with no distinguishable impact on the resulting query

Combining the Concepts

Altogether, a basic JOIN looks like the following:

Pseudocode - Join Syntax

```
SELECT
    tableA.column,
    tableB.column
FROM
    tableA
JOIN tableB ON tableA.identity_column = tableB.identity_column
```

Joins with Three or More Tables

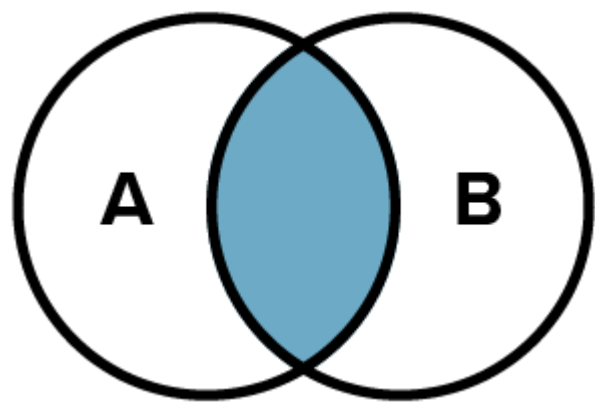
Joins can even be done between three or more tables. The syntax is similar, with each new table relation declared below the first.

Pseudocode - Join Syntax

```
SELECT
    tableA.column,
    tableB.column,
    tableC.column
FROM
    tableA
JOIN tableB ON tableA.identity_column = tableB.identity_column
JOIN tableC ON tableA.identity_column = tableC.identity_column
```

Join

The standard JOIN, also referred to as INNER JOIN, will only return rows where the joined columns contain matching values. If one of the joined columns contains a value that is not present in the in the other, then the row is not represented in the result set. You would use JOIN when you only want results that are represented in both tables.



This section will demonstrate the various uses of the JOIN keyword. For the sake of clarity, the queries will run against tables that look like the following:

Products Table

id	product_name
1	Apples
2	Oranges
3	Grapes
4	Plums

Inventory Table

id	product_id	quantity	product_vendor
1	1	15	Apple Corp
2	2	25	Orange Ya-Glad
3	3	56	Grape Escape
4	5	45	Banana Solutions

Join in Action

In this demonstration, only rows that pertain to Apples, Oranges, and Grapes are being returned. We're using a JOIN between **products.id** and **inventory.product_id**, so our results will only contain rows that have matching values from both of those columns. Our result set does not contain any information on products with **products.id** values of 4 or **inventory.product_id** values of 5, because those values are not present in **both** of the joined columns.

SQL - Joining Products and Inventory

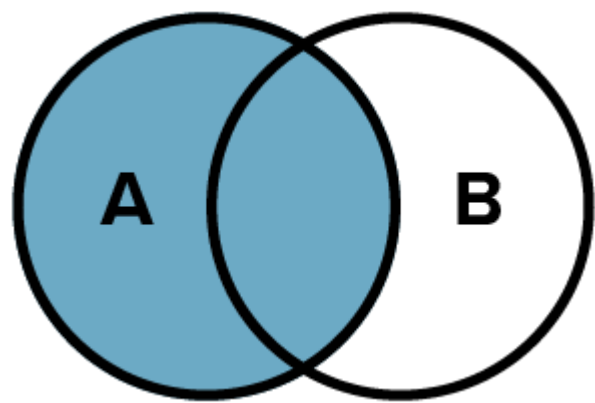
```
SELECT
    products.id
    ,inventory.product_id
    ,products.product_name
    ,inventory.product_vendor
    ,inventory.quantity
FROM
    products
JOIN inventory ON products.id = inventory.product_id
```

Example Results

id	product_id	product_name	product_vendor	quantity
1	1	Apples	Apple Corp	15
2	2	Orange	Orange Ya-Glad	25
3	3	Grapes	Grape Escape	56

Left Join

Return all rows from the left-most table (**table A** in the diagram), even if there are no matches on the right-most table (**table B**). If there isn't a matching record in the right table, then NULL values are returned.



Left Join in Action

Here we see all rows returned from our products table (since it is the left-most table in our query). In row 4, columns that are being populated via the inventory table (**product_id**, **product_vendor**, and **quantity**) contain NULL values, because there isn't a row on the inventory table that matches with a **product_id** value of 4. The query must return something in this case, so it returns NULL for these columns.

```
SQL - Left Joining Products and Inventory
```

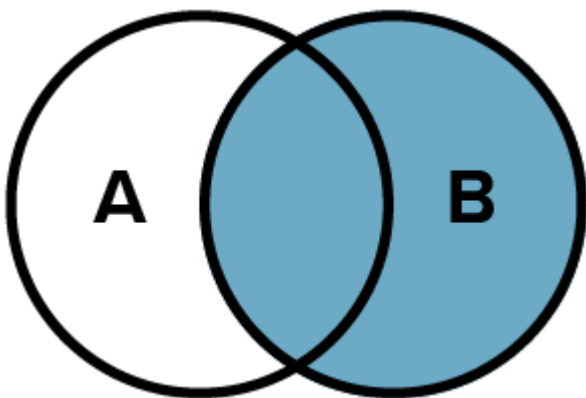
```
SELECT
    products.id
    ,inventory.product_id
    ,products.product_name
    ,inventory.product_vendor
    ,inventory.quantity
FROM
    products
LEFT JOIN inventory ON products.id = inventory.product_id
```

Example Results

id	product_id	product_name	product_vendor	quantity
1	1	Apples	Apple Corp	15
2	2	Orange	Orange Ya-Glad	25
3	3	Grapes	Grape Escape	56
4	NULL	Plums	NULL	NULL

Right Join

Return all rows from the right-most table (table B), even if there are no matches on the left-most table (table A). If there isn't a matching record on the left table, then NULL values are returned.



Right Join in Action

When using a RIGHT JOIN, all rows will be returned from the inventory table. The products table does not have a row that contains an **id** value of 5, so the **id** and **product_name** columns will show NULL

values in our result set.

SQL - Right Joining Products and Inventory

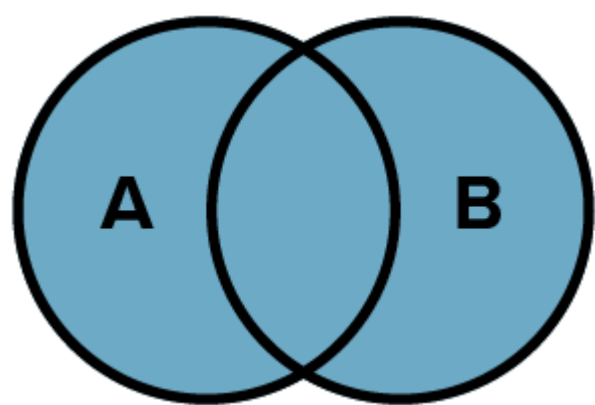
```
SELECT
    products.id
    ,inventory.product_id
    ,products.product_name
    ,inventory.product_vendor
    ,inventory.quantity
FROM
    products
RIGHT JOIN inventory ON products.id = inventory.product_id
```

Example Results

id	product_id	product_name	product_vendor	quantity
1	1	Apples	Apple Corp	15
2	2	Orange	Orange Ya-Glad	25
3	3	Grapes	Grape Escape	56
NULL	5	NULL	Banana Solutions	45

Full Join

The FULL JOIN returns all rows from both tables, regardless if there are matching values in the joined columns. You would use a FULL JOIN in cases where you want to show all applicable records from both tables, and synchronize the data across both tables via the joining columns where possible.



Full Join in Action

Note that we're using the same query as the standard JOIN, but we've prepended "FULL" to the last line of the query. Note the NULL values returned in cases where our product ID is not fully represented on both tables.

SQL - Full Joining Products and Inventory

```
SELECT
  products.id
  ,inventory.product_id
  ,products.product_name
  ,inventory.product_vendor
  ,inventory.quantity
FROM
  products
FULL JOIN inventory ON products.id = inventory.product_id
```

Example Results

id	product_id	product_name	product_vendor	quantity
1	1	Apples	Apple Corp	15
2	2	Orange	Orange Ya-Glad	25
3	3	Grapes	Grape Escape	56
4	NULL	Plums	NULL	NULL
NULL	5	NULL	Banana Solutions	45

Full Joins in MySQL

MySQL does not have an equivalent FULL JOIN. However, you can emulate one by utilizing a LEFT JOIN, RIGHT JOIN, and the UNION keyword. For the sake of simplicity, we will return all columns in the following example, but you would still want to specify individual columns in both SELECT query.

SQL - Full Join in MySQL using Left Join, Right Join, and Union Keyword

```
SELECT * FROM products
LEFT JOIN inventory ON products.id = inventory.product_id

UNION ALL

SELECT * FROM products
RIGHT JOIN inventory ON products.id = inventory.product_id
WHERE products.id IS NULL
```

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

SQL Common Functions

Functions are available in most SQL databases, and can provide some helpful utility to any queries you may be calling in Ignition.

This page contains some commonly used SQL functions that some databases contain. The exact functions available and usage depends on the database, so always check your database's documentation for a more complete list of available functions.

Using Column Values

It is important to understand that when calling these functions, you generally use a column name instead of a static number. For the sake of simplicity, the tables below demonstrate how to use the functions with static values, but they usually are switched for column names, ie:

Pseudocode - Passing a Column to a Function

```
SELECT SUM(downtime_duration) FROM downtime_events
```

Example Table

Some of the functions on this table are better demonstrated when used in conjunction with a table (i.e., using the AVG() function with a single value isn't too interesting). Thus, the following table contains sample data that the functions on this page will utilize if necessary.

Products Table

id	product_quantity	product_name	date_added	date_updated	or
1	100	apples	Mon Jan 29 00:00:00 PST 2018	Mon Jan 29 12:00:00 PST 2018	Cæ
2	24	oranges	Mon Feb 13 00:00:00 PST 2017	Mon Feb 13 09:00:00 PST 2017	Flø
3	56	grapes	Mon Mar 07 00:00:00 PST 2016	Mon Mar 07 05:00:00 PST 2016	Cæ

Numeric Functions

Function	Description	Example	Output
ABS(value)	Returns the absolute value of the passed number or column.	<pre>SELECT ABS(-3.5)</pre>	3.5
AVG(value)	Takes the values of a single numeric column, and returns an average. A WHERE clause may be used in the same statement to filter out some of the rows on the table.	<pre>SELECT AVG(product_quantity) FROM products</pre>	60
CEILING(value)	Returns the next greatest integer value based on the argument provided. Thus, CEILING(10.1) would return 11.	<pre>SELECT CEILING(10.1)</pre>	11
COUNT(value)	Returns a row count. Typically takes either a single column, *, or 1. Regardless of which row is passed, the function will return the number of rows on the table that meet the criteria of any WHERE clauses.	<pre>SELECT COUNT(*) FROM products</pre>	3
FLOOR(value)	Returns the next smallest integer value based on the argument provided. Thus, FLOOR(10.9) would return 10.	<pre>SELECT FLOOR(10.9)</pre>	10
MAX(value)	Returns the largest value from the specified column.	<pre>SELECT MAX(product_quantity) FROM products</pre>	100

Function	Description	Example	Output
MIN(value)	Returns the smallest value from the specified column.	<pre>SELECT MIN(product_quantity) FROM products</pre>	24
ROUND(value, decimal_places)	Returns a number rounded to a certain number of decimal places. Takes two parameters. The first is the number to round to, and the second is the number of decimal places to round to.	<pre>SELECT ROUND(1.234, 1)</pre>	1.2
SUM(value)	Takes the value of a single numeric column, and returns the sum. A WHERE clause may be used in the same statement to filter out some of the rows on the table.	<pre>SELECT SUM(product_quantity) FROM products</pre>	180

String Functions

Function	Description	Example
CONCAT(value1,valu2,...valueN)	Concatenates multiple strings or values. Some databases may require you to convert each value to a string before concatenating.	<pre>SELECT CONCAT(product_name, ':', product_quantity) FROM products</pre>
LOWER(value)	Converts a string to lowercase.	<pre>SELECT LOWER('MAKE Me smALL')</pre>
LTRIM(value)	Removes leading space from a string.	<pre>SELECTCT LTRIM(' Take a little off the left')</pre>
REPLACE(ornal_string, target_string, replacement_string)	Searchings a string for a substring (target_string), and replaces the substring with the replacement_string.	<pre>SELECT REPLACE('Who is awesome', 'Who is', '')</pre>
RTRIM(value)	Removes leading space from a string	<pre>SELECT RTRIM('Take a little off the right ')</pre>
SUBSTRING(ornal_string, character_index, [length])	Extracts a substring from another string based on character index. Takes two parameters: the original string, and the character index	<pre>SELECT SUBSTRING('This is my string!' , 9, 5)</pre>

Function	Description	Example
	to start at. An optional third parameter can specify the number of characters to extract. Character index is one-based, so the first character in the string rests at index 1.	
TRIM(value)	Removes both leading and trailing space from a string.	<pre>SELECT TRIM(' Trim Both Sides ')</pre>
UPPER(value)	Converts a string to uppercase.	<pre>SELECT UPPER('super size me')</pre>

Date Functions

There are many date and time functions for each database (MySQL, MSSQL, Oracle, etc), but they all vary wildly. These examples work in most databases:

Function	Description	Example	Output
CURRENT_TIMESTAMP()	Returns the current date and time, as reported by the database.	SELECT CURRENT_TIMESTAMP ()	Returns the current time
TIMEDIFF(date1, date2)	Returns a difference between two dates. Assumes that date1 is the most recent datetime.	SELECT TIMEDIFF(date_updated, date_added) FROM products WHERE id =1	Thu Jul 01 12:00 PST 1970

Logic Functions

Function	Description	Example	Output
COALESCE(value1, value2,...valueN)	Returns the first non-null expression.	<pre>SELECT COALESCE(NULL, 'Pick me!')</pre>	Pick me!
ISNULL(expression)	Returns true if an expression is NULL.	<pre>SELECT ISNULL(NULL)</pre>	True
		<pre>SELECT ISNULL(14)</pre>	False
NULLIF(expression1, expression2)	Compares two expressions. If they are equal to each other, then the function returns a NULL. If the two expressions are not equal, the first expression passed to NULLIF() is returned.	<pre>SELECT NULLIF(100, 100)</pre>	NULL
		<pre>SELECT NULLIF(100, 3)</pre>	100

Group By Clause

The GROUP BY statement is used in conjunction with the aggregate functions to group the result-set by one or more columns. This way you can find the MIN, MAX, Average, COUNT, etc., for each group of rows. Grouping can also be done for multiple columns, with precedence going in the order that they are listed.

Pseudocode - Passing a Column to a Function

```
SELECT SUM(column1) FROM table GROUP BY column2
```

Example

Here, we are grabbing the sum of product quantity for each origin state.

SQL - Simple Group By

```
SELECT SUM(product_quantity), origin_state, FROM products_table GROUP BY  
origin_state
```

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

SQL Stored Procedures

For the uninitiated, Stored Procedures are a series of predefined SQL statements that are configured and stored in the database, and can be easily called from another application, such as Ignition, with a single statement. Conceptually, they are very similar to a scripting function: parameters may be passed to them, they can utilize looping and conditional logic, and can return a value.

Stored Procedure Syntax

Stored procedures are created and maintained in your database. As a result, the creation of a stored procedure is outside the scope of the Ignition User Manual: the commands used to create a stored procedure vary per database.

In regard to calling a Stored Procedure, the syntax can also differ.

SQL Server

The EXEC command is utilized to execute a stored procedure.

Pseudocode - Executing a Stored Procedure in SQL Server

```
EXEC dbo.myStoredProcedure
```

Parameters may be passed to the Stored Procedure. SQL Server's documentation has more details on utilizing parameters with Stored Procedures.

Pseudocode - Executing a Stored Procedure in SQL Server with Parameters

```
EXEC dbo.myStoredProcedure @myParam = 100, @AnotherParameters = 'Hello'
```

MySQL

MySQL uses the CALL command to execute a Stored Procedure. Note the parentheses characters at the end, as they must be present even when the Stored Procedure is not utilizing parameters.

Pseudocode - Executing a Stored Procedure in MySQL

```
CALL myStoredProcedure()
```

If parameters are defined, they can be passed via the parenthesis, similar to how functions work in Python.

Pseudocode - Executing a Stored Procedure in MySQL with Multiple Parameters

```
CALL myStoredProcedure(100, 'Hello')
```

For information on the creation of a stored procedure, as well as proper SQL syntax to call a Stored Procedure, reference your database's documentation. Alternatively, if you have a database administrator, they can typically help with the creation and execution of a Stored Procedure.

Calling Stored Procedures in Ignition

There are several locations in Ignition where Stored Procedures may be utilized from.

SQL Query Bindings

Instead of typing a query directly into a [SQL Query binding](#), a Stored Procedure may be executed instead. Assuming a MySQL database contains a Stored Procedure named 'return_all_bays', we can call the procedure on a binding with the following:

SQL - Calling a Stored Procedure in MySQL

```
CALL return_all_bays()
```

As with all bindings, Tag  and Property  references may be utilized by clicking the appropriate icons in the binding window.

Property Binding: Root Container.Power Table 1

Binding Types

Tag

☐ Tag

☐ Indirect Tag

☐ Tag History

Property

☐ Expression

☐ Property

Database

☐ Named Query

☐ DB Browse

☒ SQL Query

SELECT Query

```
CALL return_bay_data(  
    {Root Container.Current_Id.intValue},  
    {Root Container.Selected_Area.intValue}  
)
```

Database Connection

Polling Mode ☒ Off ☐ Relative ☐ Absolute

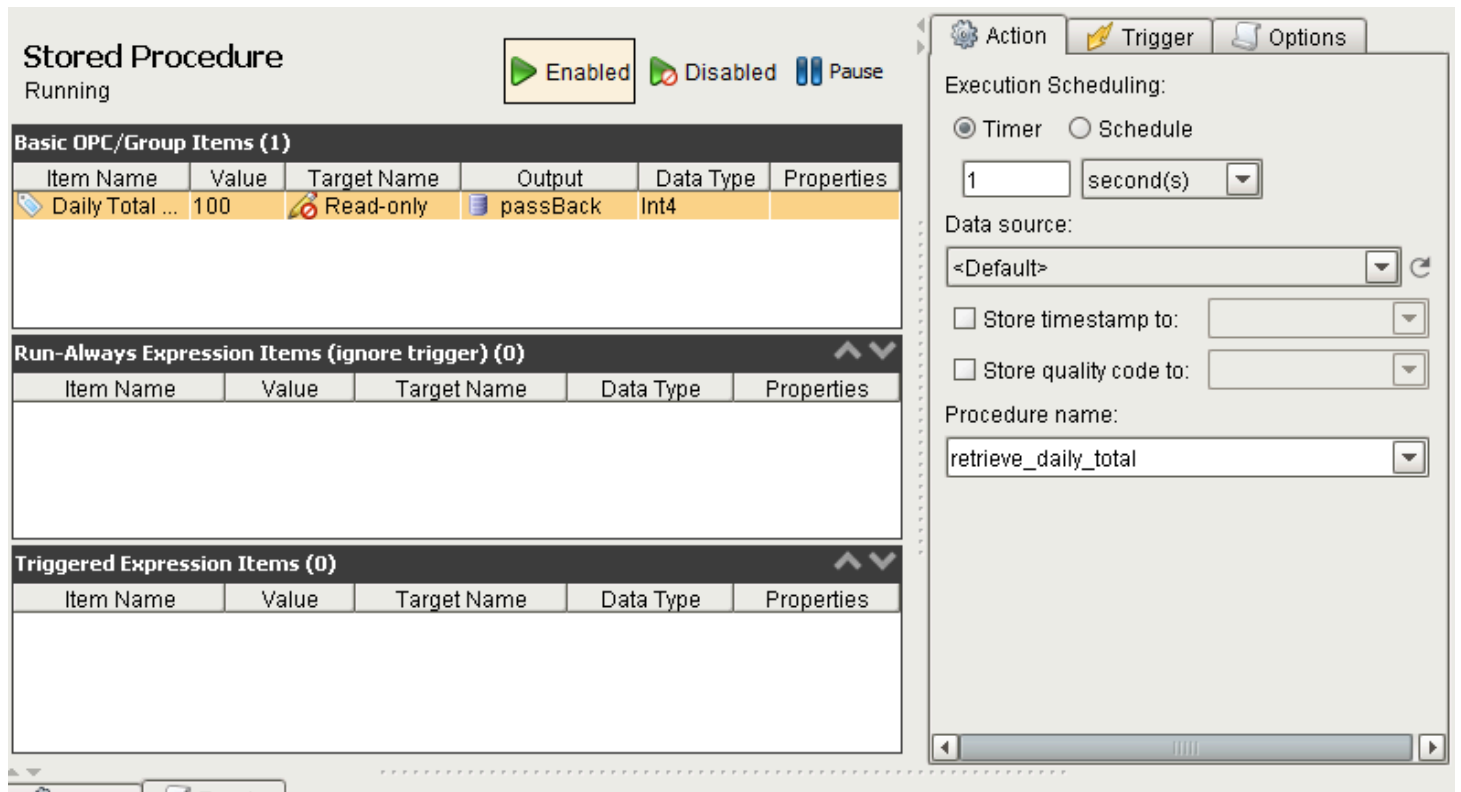
Polling Rate Rate N/A sec

Retain Rows ☐ false

Stored Procedure Groups

One of the easiest ways to utilize Tags with a Stored Procedure is to use the **Stored Procedure Group**. Parameters can be easily assigned to each item in the group, and utilize all of the features of a **Transaction Group**, such as scheduled execution and triggers.

Each item in the group is linked to a specific parameter in the Stored Procedure. Any IN or INOUT parameters can write directly to the Tags, while new values can be fed into OUT and INOUT parameters allowing you to easily move data from Tags into the database with the Stored Procedure.



Stored Procedures in Python Scripts

There are a few ways to call a Stored Procedure from a script in Ignition.

Using createSProcCall and execSProcCall

The recommended approach to calling a Stored Procedure from a Python script in Ignition typically involves two main steps:

1. Calling `system.db.createSProcCall` to create a call context, or object that effectively represents the impending stored procedure call. This object can be used to specify parameter values that will be passed to the Stored Procedure.
2. Using `system.db.execSProcCall` to execute the Stored Procedure.

Once the Stored Procedure has been executed, the call context generated in step #1 can be used to reference any values that were returned.

Other System Functions

Technically, most other system functions in the "db" library, such as `system.db.runPrepQuery`, can be used to call a Stored Procedure. We generally recommend against this, as `system.db.createSProcCall` and `system.db.execSProcCall` are better suited to work with Stored Procedures and have some additional functionality not found in the other db functions.

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Database Query Browser



INDUCTIVE
UNIVERSITY

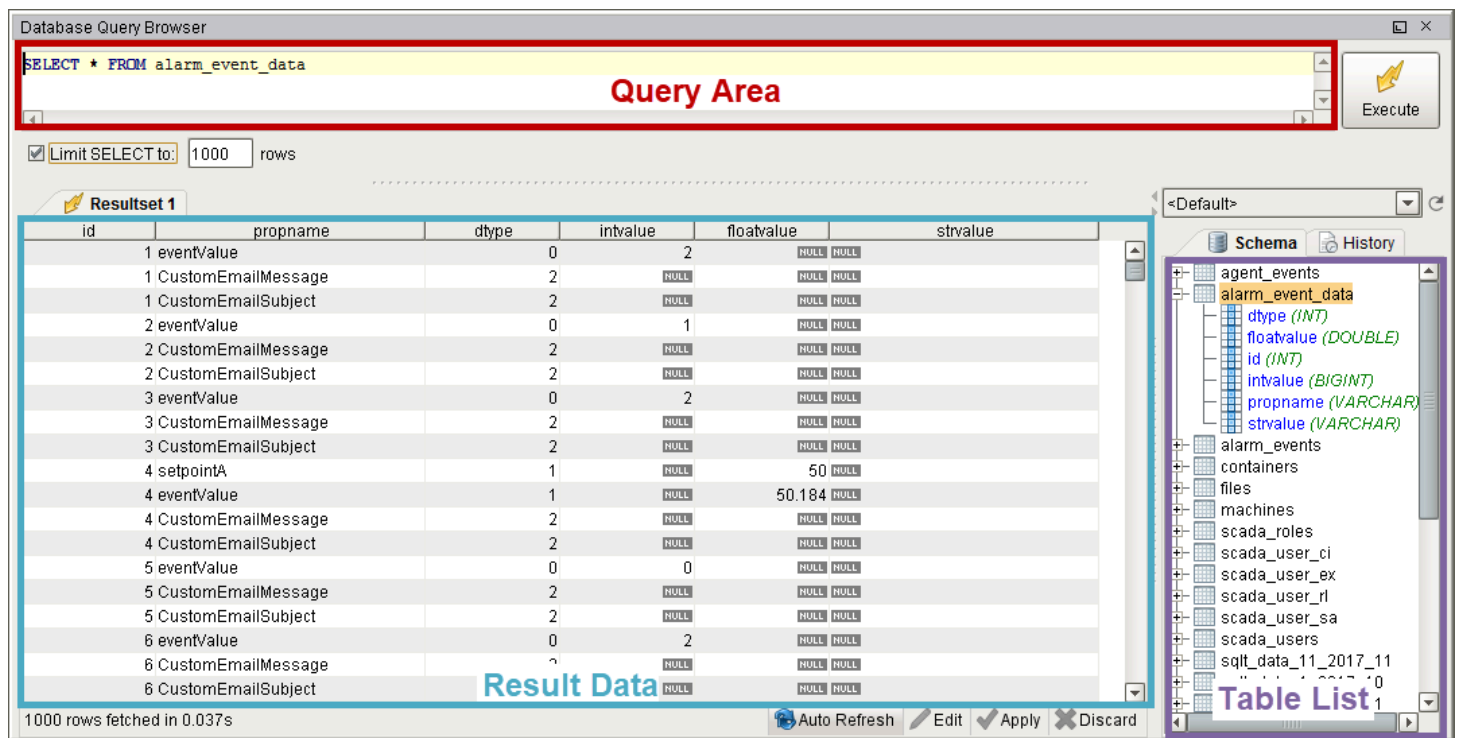
Using the Query Browser

[Watch the video](#)

The Database Query Browser Panel

The **Database Query Browser** is a very convenient tool that lets you query any database connected to Ignition, and interact with tables. Because Ignition is so heavily integrated with databases, it is very common in the course of project design the need to inspect the database directly, or to experiment with a SQL query to get it just right.

The Database Query Browser is found in the tools menu and has a three basic parts to it. The **Query Area** is where a query can be written and then executed against the selected database connection. This can accept any type of query, but if the query would update the database, the Designer needs to have its Gateway **Communication Mode** set to read/write first. The database connection that the query is executed against can be chosen from the dropdown, below the **Execute** button. The **Result Data** space is where the results of the executed query will appear. If a SELECT query was run, then the table data will be shown. If an UPDATE, INSERT, or DELETE query was run, then the number of rows affected will be displayed instead. Finally, the **Table List** will display all of the tables in the specified database connection. The tables can be expanded to show the columns and their data types for that table. This can help when writing queries in the Query Area. Additionally, when a table is double clicked, the Query Area will automatically be populated with a `SELECT * FROM` the table.

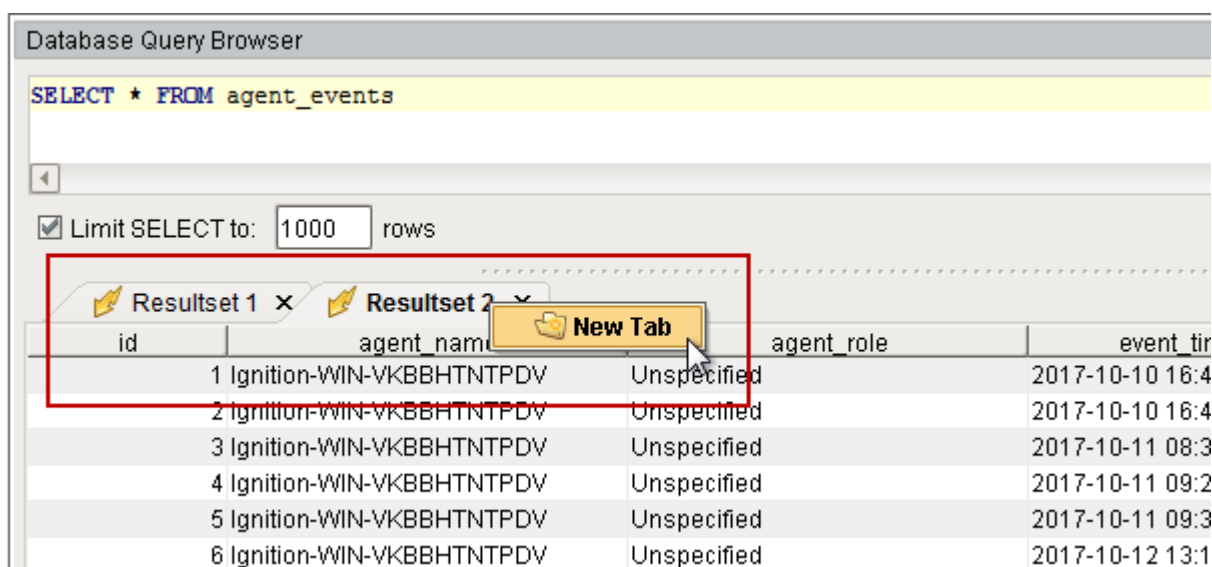


Features of the Query Browser

The Database Query Browser has a few features that can help manage and build any SQL query.

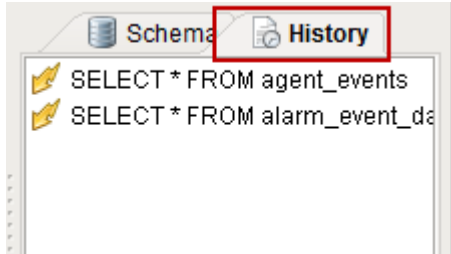
Multiple Resultsets

The Query Browser allows you to make multiple tabs of results, so that multiple queries can be run and their results compared to each other. Simply right click on the **Resultset** area and select **New Tab**. Your queries will be saved for each tab.



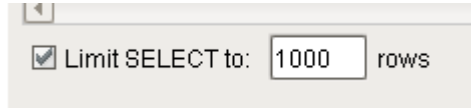
Query History

At the top of the Table List, there is a second tab labeled **History**. This will switch the Table List to instead show the history of queries run in the query browser. Double clicking one of the entries will push the query into the Query Area (for the selected tab). This allows you see what queries have been run previously to help you tweak your current query.



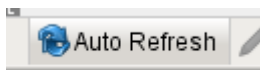
Select Limit

By default, any SELECT statement is limited to 1000 rows. This is to help the queries return quickly, however, it may not always be wanted. This can be turned off or adjusted by either clicking the checkbox or manipulating the row number located underneath the Query Area on the left. It is recommended to leave this on unless you know the result set size. It's better to use a count query than to return all results to see the result count. IE: `SELECT COUNT(*) FROM table...`



Auto Refresh

The Query Browser also lets you monitor a database table for changes by using the **Auto Refresh** button. This is often convenient when designing Transaction Groups. As the group runs, you can view the table that it is targeting with Auto Refresh turned ON to watch how the group is altering the table. Simply click the Auto Refresh button at the bottom of the Query Browser to periodically rerun the query in the Query Area. Make sure to include an ORDER BY clause so your results show up in the order you want.



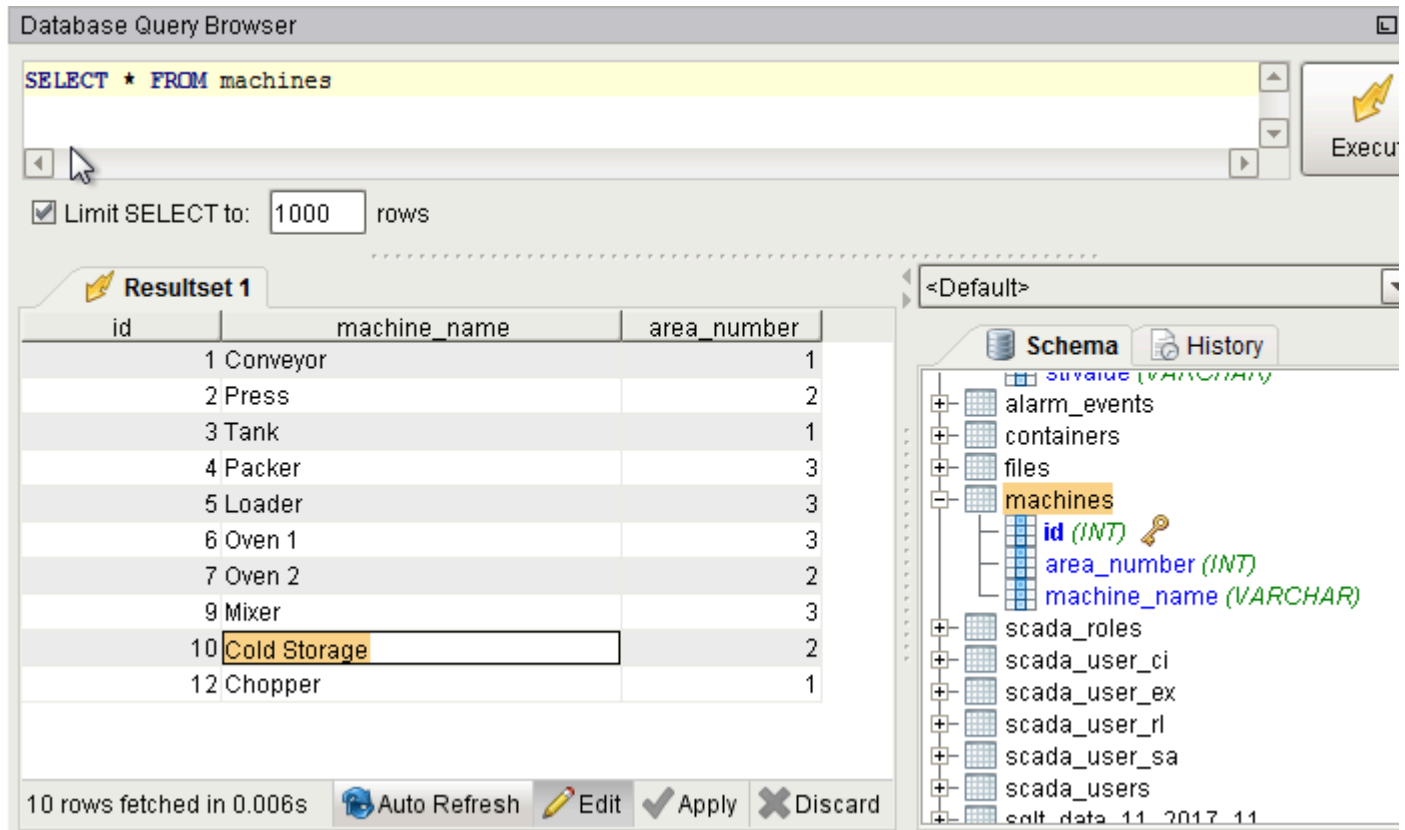
Editing the Table in the GUI

In addition to editing the table data using INSERT, UPDATE, and DELETE statements, the data can also be edited within the Result Data. Simply click the Edit button at the bottom of the Query Browser window.

! INFO

Editing in this way requires that the data be obtained from a single table with a simple query. The table must also not be tied to other tables.

Once the Edit button is clicked, the values in the table can be edited by double clicking on the value and entering a new one.



The screenshot shows the Database Query Browser interface. The query bar contains the SQL statement `SELECT * FROM machines`. Below the query bar, there is a checkbox labeled "Limit SELECT to:" with a value of "1000" rows. The "Resultset 1" tab is active, displaying a table with the following data:

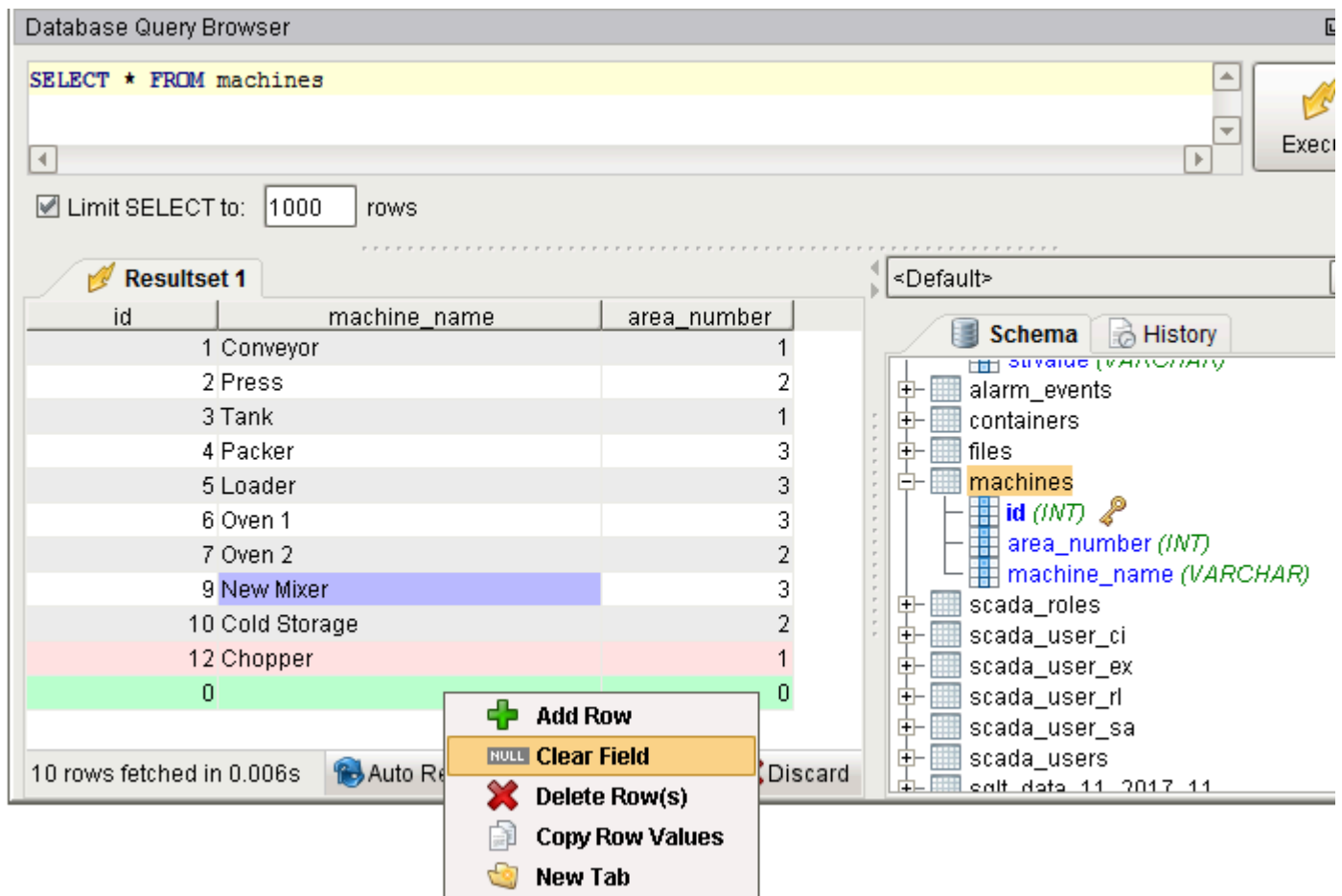
id	machine_name	area_number
1	Conveyor	1
2	Press	2
3	Tank	1
4	Packer	3
5	Loader	3
6	Oven 1	3
7	Oven 2	2
9	Mixer	3
10	Cold Storage	2
12	Chopper	1

The "Cold Storage" row is highlighted. The bottom status bar indicates "10 rows fetched in 0.006s" and includes buttons for "Auto Refresh", "Edit", "Apply", and "Discard". On the right side, there is a "Schema" pane showing a tree view of the database structure, including tables like "alarm_events", "containers", "files", "machines", "scada_roles", "scada_user_ci", "scada_user_ex", "scada_user_rl", "scada_user_sa", "scada_users", and "scit_data". The "machines" table is expanded, showing its columns: "id (INT)" with a primary key icon, "area_number (INT)", and "machine_name (VARCHAR)".

Right clicking on a row also provides a few options:

- **Add Row:** Will add a new row to the table for data to be entered into.
- **Clear Field:** Will clear out the value in the selected cell so that it will be NULL. This is different than simply deleting the value out and leaving it empty.
- **Delete Row(s):** Will delete the selected row or rows from the table.
- **Copy Row Values:** Will copy the row values in a comma separated form to the clipboard.
- **New Tab:** Will Create a new Resultset tab for a new query to be run in.

When editing values, cells will highlight depending on what is being done to them. Green cells are new, and typically indicate a new row was added. Red cells are marked for deletion, and will be deleted when the changes are confirmed. Blue cells are cells that have had values changed during editing.



After making edits to the table data, the changes then either need to be applied or discarded. This gives you the opportunity to revert the table to the way it was before the current edit session, or apply the changes and rewrite the table appropriately. Simply click the corresponding button at the bottom of the Query Browser next to the Edit button. Make sure your Gateway **Communication Mode** is set to Read/Write before Applying your changes.



This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Query Builder

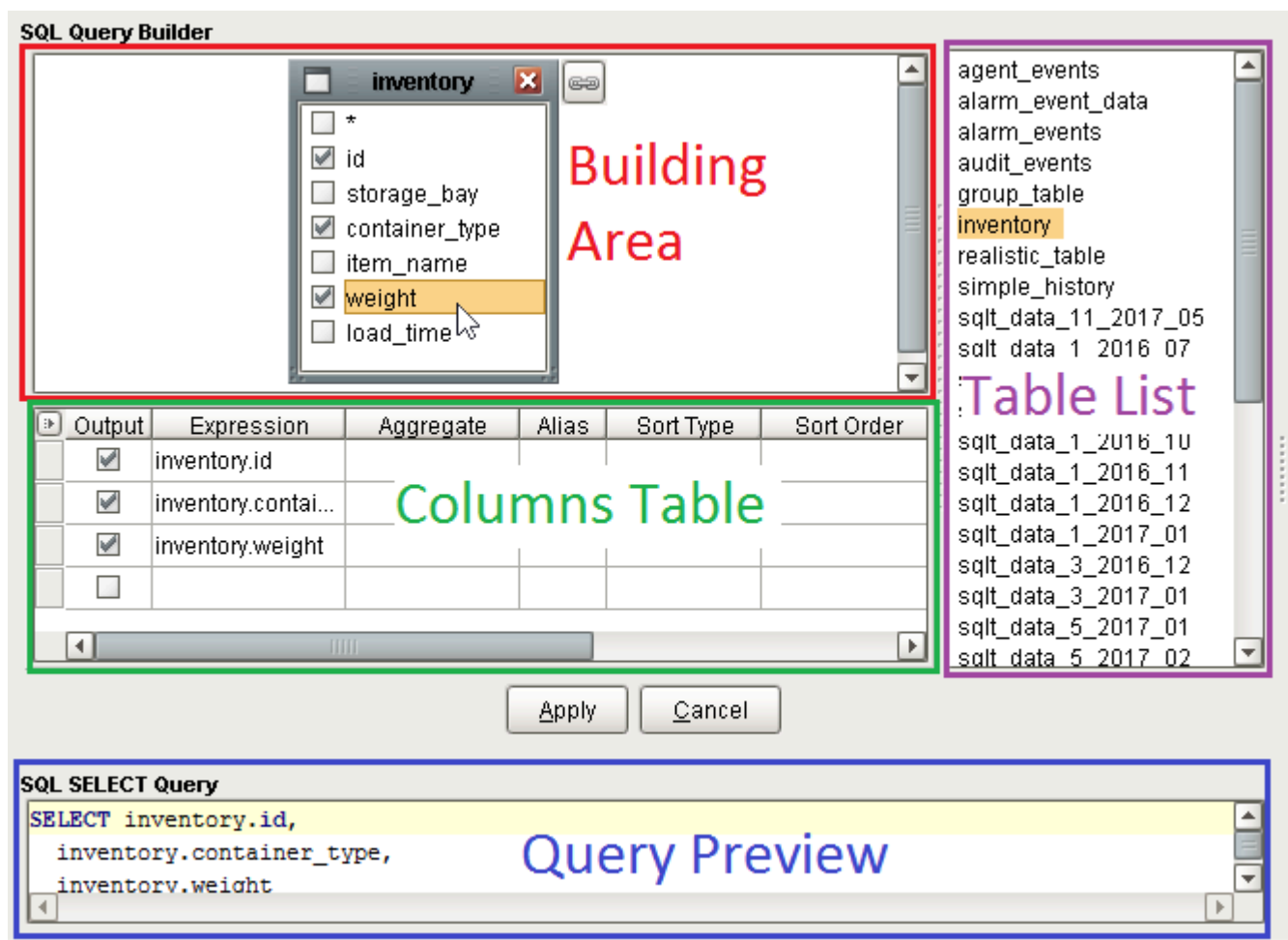
Crafting Queries with the Query Builder

Many places in Ignition that allow for SQL queries have a link to the SQL Query Builder tool. The Query Builder is a powerful Drag-and-Drop query building GUI that allows you to make complex queries from your connected databases. While a basic understanding of SQL helps make the most of the Query Builder tool, most people will have no problem creating effective queries after a brief tutorial. Additionally, the Query Builder does go over many advanced features of SQL that may be unfamiliar. We suggest looking up how these work in your favorite SQL resource guide as this covers how to use them, not what they do.

Builder Anatomy

Once opened, the Query Builder has the following items:

- **Building Area:** Visualizes the table relationships of all tables included in the query. Tables are typically dragged in from the **Table List**.
- **Columns Table:** Shows which columns from all tables are referenced in the query.
- **Table List:** Shows all database tables in the selected Database Connection.
- **Query Preview:** Shows a preview of the query that will be created once the **Apply** button has been pressed.



Opening the Query Builder

The Builder can be accessed from several different resources in Ignition, notably when using Named Queries and setting up Report Data. To use the Builder, click the **Show Builder** button.

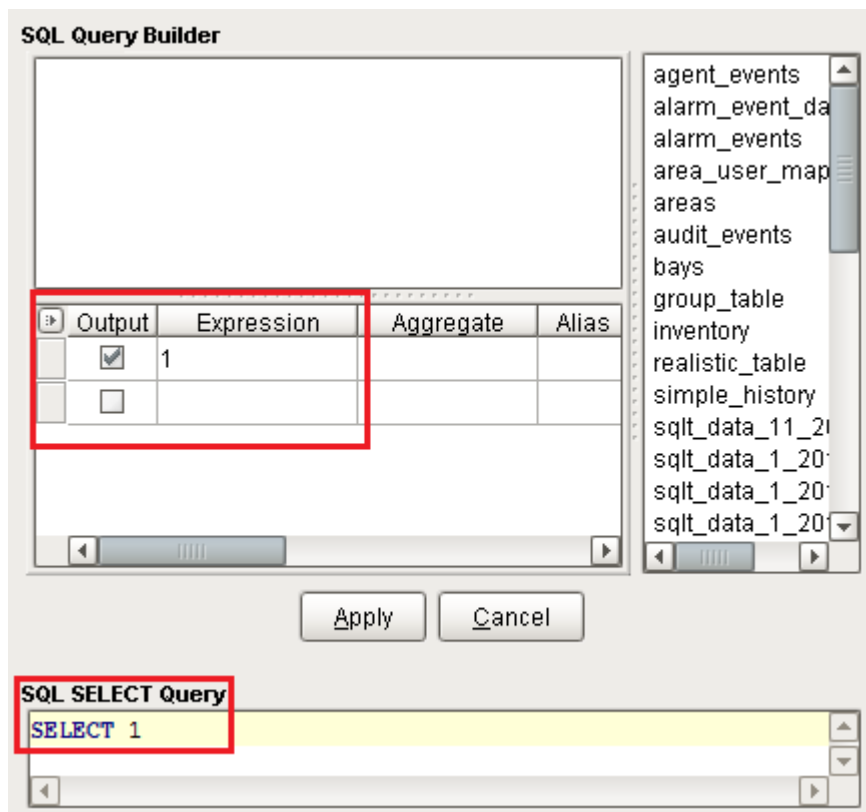


Clicking this button will open the Builder workspace with an empty query. If a query was already written before the button was pressed, then the Builder will attempt to load the query into the Builder's interface. For example, if the following query was typed before pressing the button:

SQL - Select Statement - Loading into the Builder's Workspace

```
SELECT 1
```

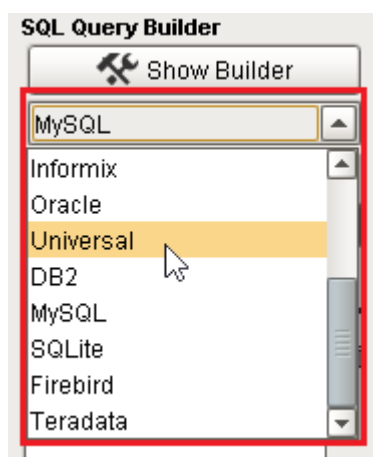
The following would be loaded into the Builder's workspace. Note that the **Building Area** is empty, because we're not querying from a table.



The Builder attempts to parse your query based on the **Syntax Parser**.

Using the Syntax Parser

When starting with a pre-existing query, the Syntax Parser tells the Builder which implementation of SQL your query's syntax is using. In most cases, this can be left with the default value of **Universal**. When set to a specific implementation of SQL, this allows the Builder to understand/accept implementation-specific keywords and syntax.

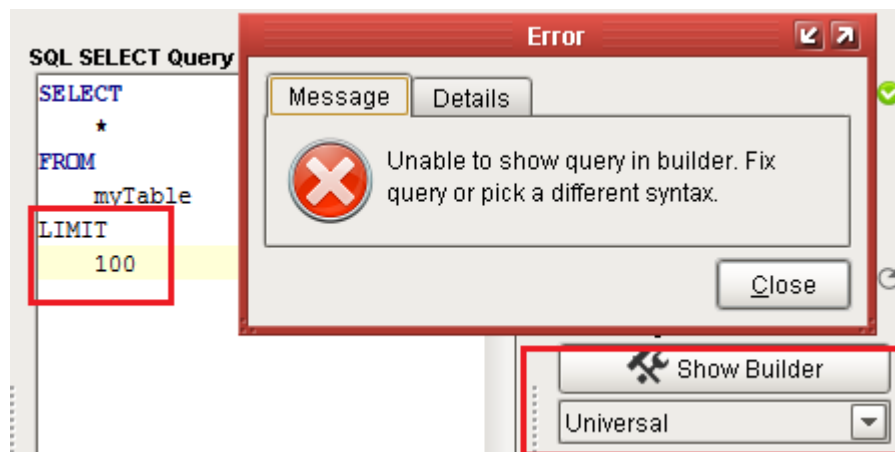


Syntax Parser in Action

In MySQL, query results may be limited with the **LIMIT** keyword:

```
SELECT
    *
FROM
    myTable
LIMIT
    100
```

Attempting to open the Builder while using **LIMIT**, and the Syntax Parser is configured to a syntax that does recognize the **LIMIT** keyword, will result in an error:

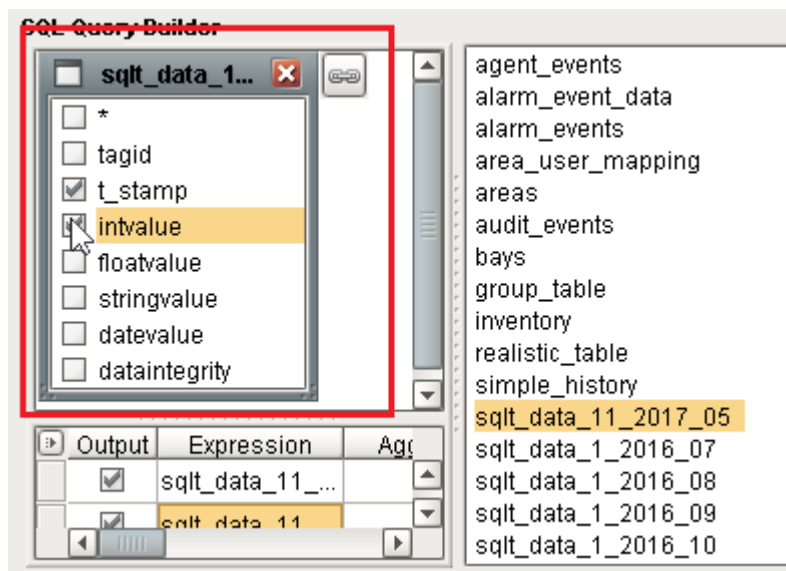


In these scenarios, you can switch to a syntax that supports the query, or remove the offending lines.

Using the Builder

Once the Builder is open, (and if you have tables in your database connection), you can start building a query by dragging and dropping a table from the Table List into the Building Area. From here, you can select which columns on the Table Object you want to bring into the query, or drag and drop them into the Columns Table below.

The * option at the top of the table is special. If the * character is selected, then all columns from the table will be included in the results just like a "SELECT * " query.



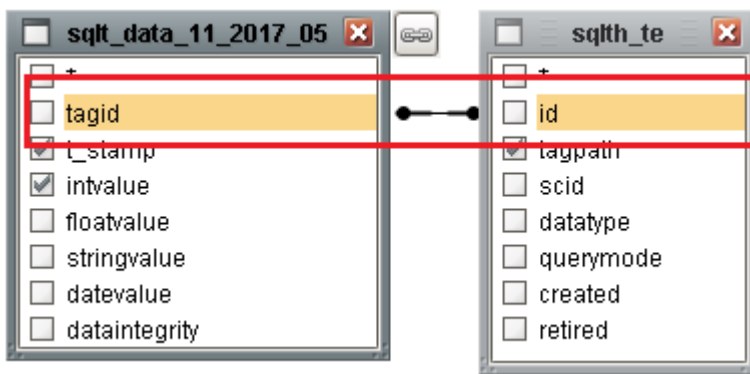
Query Properties

Right clicking on the empty space in the Building Area and selecting properties will bring up the Query Properties window that allows you to customize how the query works. How it looks and what it contains can vary, depending on what syntax parser you have selected.

MySQL Query Properties	MSSQL Query Properties	PostgreSQL Query Properties

Joining Tables in the Query Builder

Multiple tables may be added to the Building Area. Once there, the Builder can JOIN the two tables by dragging from one column in a table on to another column on a different table.

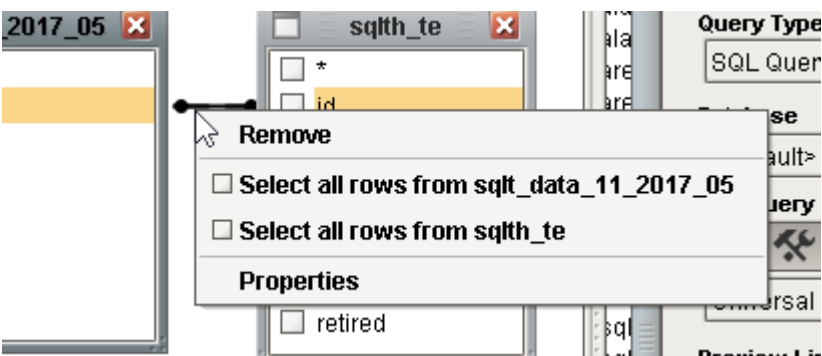


Note that "tagid" and "id" are now linked together. This means the resulting query will JOIN the two tables based on the linked columns. Much like a SQL JOIN, the joining columns do not need to be selected to be utilized in the JOIN.

Joins Right-Click Menu

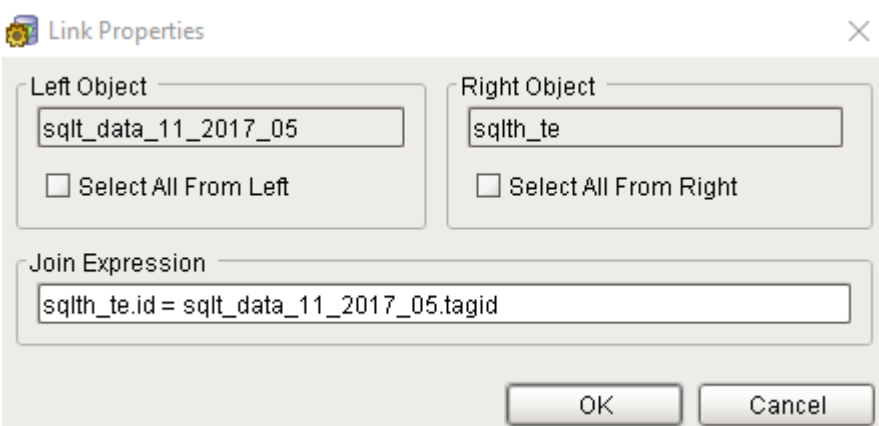
Right-clicking on the link will create a menu that allows you to remove the **JOIN**, as well as access properties of the JOIN.

Additionally, you can select all rows from one of the joining tables, which effectively creates a **RIGHT OUTER JOIN** or **LEFT OUTER JOIN**, depending on which table you select.



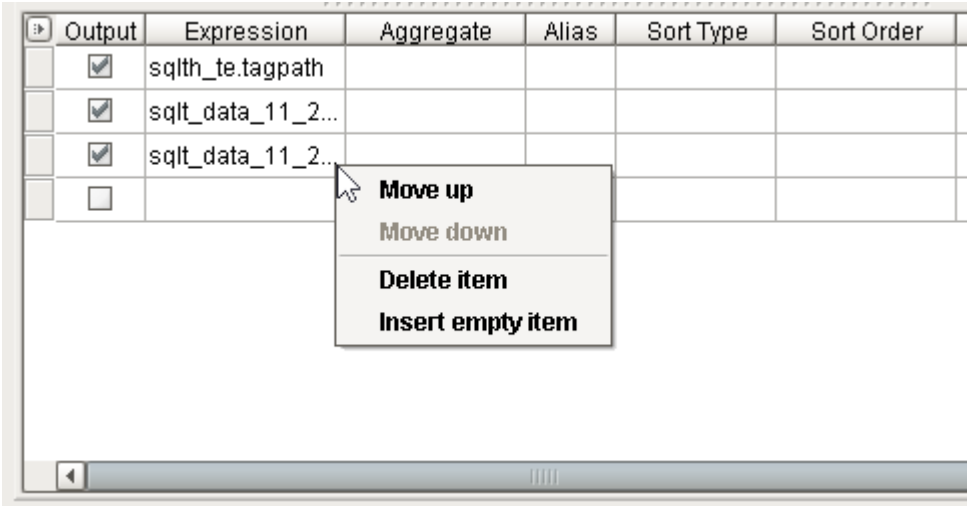
Join Properties

Clicking on **Properties** on the Right-Click Menu allows you to view the relation between each table, as well as apply **LEFT** or **RIGHT JOINS**, as mentioned above.



Columns Table

The Columns Table allows you to modify individual columns in the query. Typically, this allows you to add aggregates or aliases to each column from the Builder. Each row represents an expression, or combination of functions, columns, variables, and constants that will ultimately be a single column in the resulting query.



Right-Clicking on an Item

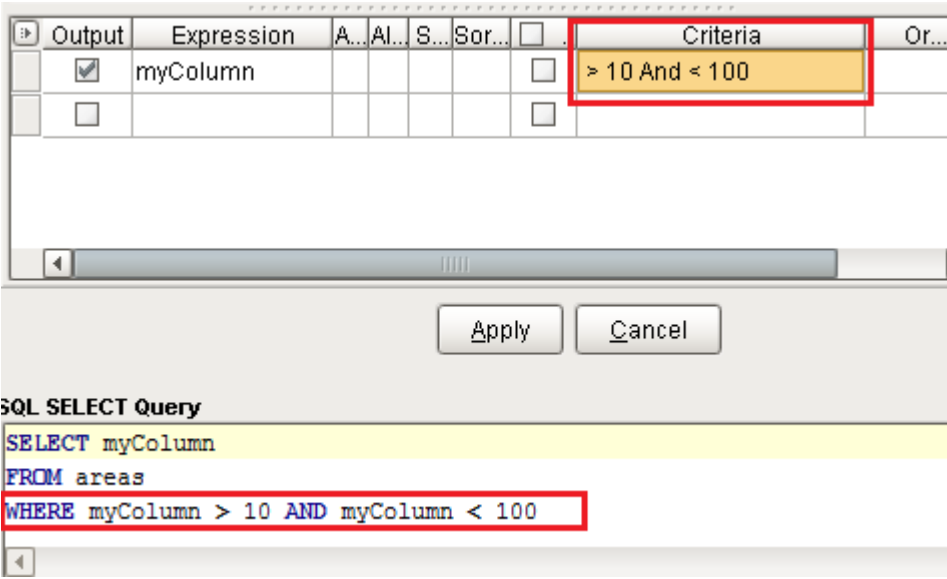
Right-clicking on a row of the Columns Table brings up a menu with the following options:

Option	Description
Move up	Moves the selected row up in the table. This means the column will appear in the query results prior to the other listed columns.
Move down	Similar to move up, but moves the row down, or towards the end of the listed columns.
Delete item	Removes the row from the Columns Table, effectively removing it from the resulting query.
Insert empty item	Adds a new row to the Columns Table with blank values.

Field Description

Each column in the Columns Table allows you to modify the resulting query in some way. The following is a description of each column:

Option	Description
Output	Specifies if the row should appear in the resulting query. Disabling a row on the table removes the column from the results.
Expression	The column from one of the tables in the Building Area that this row represents. Left-clicking on this field will create a dropdown list of possible rows.
Aggregate	Allows you to aggregate the expression. Blank by default (no aggregate), configuring this column allows you to do things like sum column, or return the average. Possible values are: - Blank (no aggregation will be performed) - Avg - Count - Max - Min - Sum You can also add the Distinct keyword to any non-blank option. For example, you could enter "Sum Distinct" or "Count Distinct".
Alias	Creates an Alias, or alternative name for the column. Effectively adds the SQL "AS" keyword to the column, which allows you to rename the resulting column.
Sort Type	Sorts the results based on this column. Enabling a Sort Type is similar to added an ORDER BY to your query. Possible values are: - Blank (no sorting on this column will be performed) - Ascending - Descending
Sort Order	When a Sort Type has been specified on multiple rows, this determines which row the query will sort on first.
Grouping	Allows you to group the results. Adds a GROUP BY statement to the resulting query. Enabling grouping on one of the columns makes the Criteria for column appear.
Criteria for	A dropdown list that sets how you want the Criteria column to apply to the grouping. Does not apply if both Grouping and the WHERE clause Criteria

Option	Description
	are not being used. - For Values will place the WHERE clause specified in the Criteria column first, filtering out rows that don't apply, and then applying the grouping. - For Groups will instead change the WHERE clause into a HAVING clause that is used to filter after the grouping has taken place.
Criteria	Allows you to add a WHERE clause. Supports the use of both the OR and AND keywords for multiple conditions. Example: assuming a column named "myColumn", we could limit the results of our query to rows where myColumn has a value greater than 10 AND less than 100:  SQL SELECT Query <pre>SELECT myColumn FROM areas WHERE myColumn > 10 AND myColumn < 100</pre>
Or...	Allows you to add an additional condition to a WHERE clause. Will separate each grouped condition with "(" characters to maintain logic.

CAUTION

The Query Builder is a third party tool that we brought into Ignition and while we go over how to use it here, you can also check out the [Active Query Builder's documentation](#) for more information on how this feature works.

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Named Queries

New in 7.9.4



INDUCTIVE
UNIVERSITY

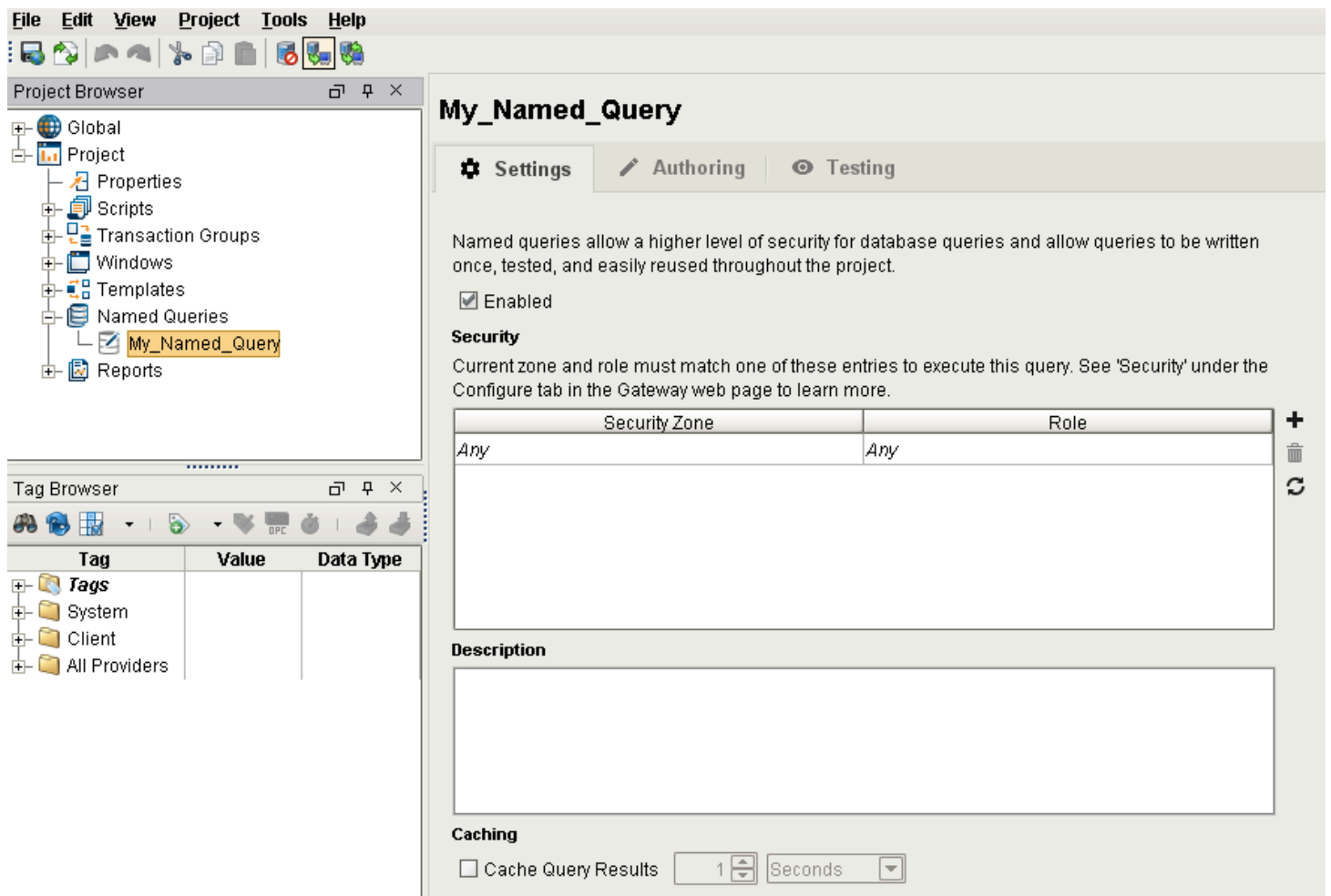
Named Query Overview

[Watch the video](#)

What are Named Queries?

Named Queries are preconfigured queries that may be invoked elsewhere in a project. Named Queries are conceptually very similar to Project Scripts; defined in a single location, and then referenced in multiple places throughout the project. When executing a Named Query, parameters may be passed to return a dynamic result set. This way, a query may be written once, and then called from multiple locations throughout the project.

Named Queries have their own workspace in the **Project** section of the **Designer**.



Secure Query Execution

While clients may request data from a Named Query, the actual execution of the query always takes place on the Gateway. Clients simply specify which query should run, and pass parameters that the Gateway will use. Additionally, the Gateway has an opportunity to **restrict access** to the query based on Security Zone and/or User Role. This provides a single interface to restrict access to the queries, and better protect your data. Additionally, queries cannot be modified by a Client other than by passing variables into it. This creates a very secure method to control what queries are being run against your database.

Where are Named Queries Used?

Named Queries may be used in multiple locations in Ignition. They are used in all the same places that a normal SQL query would be used. Here are a few examples.

Bindings

A **Named Query binding** type has been added to leverage named queries for any component property bindings. Parameters may be bound to Tags or other properties on the same window,

allowing your users to modify the resulting dataset by manipulating other components similar to the original SQL Query binding type. You can also select and update a query to simulate a bi-directional binding to the database.

✂ Property Binding: Main Window (1).closable

Binding Types

Tag

☐ Tag

☐ Indirect Tag

Property

☐ Expression

☐ Property

Database

☒ Named Query

☐ DB Browse

☐ SQL Query

☒ No Binding

SELECT Query

Path

Sales_KPI

Parameters

Type	Name	Data Type	Value
Value	myValueX	String	{Root Container.styl...
Value	myValueY	Int4	{ItemName}

Query

SELECT column_a,
column_b
FROM my_table

Polling Mode

Polling Rate

☒ Off ☐ Relative ☐ Absolute

Rate = (Base Rate) +/-

0

 sec

UPDATE Query

☐ Enabled

Path

Parameters

Query

OK

Cancel

Reporting

A **Named Query Data Source** has been added to **Reports**. Report Parameters may be used by the Named Query to generate dynamic reports.

Report Overview

Data

Design

Preview

Schedule

Parameters

BayNumber

Data Sources

1 Query - named_query

Path

Sales_KPI

Parameters

Type	Name	Data Type	Value
Value	myValueX	String	
Value	myValueY	Int4	

Query

```

SELECT column_a,
       column_b
FROM my_table
WHERE column_x = :myValueX
      AND column_y = :myValueY

```

Data Key

named_query

Query Type

Named Query

Preview Limit

100

Nested Queries

named_query

Scripting

The `system.db.runNamedQuery()` function may be used to call a Named Query from any Python Script. This provides a secure method to execute a query from any location in Ignition.

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Named Query Workspace

New in 7.9.4

Named Query Workspace Overview

Named Queries have a dedicated workspace inside of the Project Section of the Designer. This workspace allows for the creation and testing of Named Queries. Once created, the Named Query may be called from another resource, such as a [datasource in a report](#) or a component using a [Named Query binding](#).

Named Queries are created by **right-clicking** the Named Query item in the Project Browser. Like other resources in the Project Browser, Named Queries can be organized in folders, which creates a unique path to the query and helps keep your queries organized.

NOTE

Named Queries are referenced by path, so renaming the Named Query or any parent folders will require you to update the path on any other resources that are using it.

Also like other resources, multiple Named Queries may be opened in the same Designer session. Tabs at the bottom of the Designer allow for easy swapping between Named Queries.

Named Query Icons

Named Queries in the Project Browser use the following Icons.

Icon	Description
	Disabled
	Enabled and open in the current Designer session
	Enabled, and closed in the current Designer session

Workspace Sections

The Named Query workspace contains three sections (or tabs). A description of each section is listed below.

Settings

This section contains configuration properties and security for the selected Named Query. The following properties are available:

 Settings

 Authoring

 Testing

Named queries allow a higher level of security for database queries and allow queries to be written once, tested, and easily reused throughout the project.

☒ Enabled

Security

Current zone and role must match one of these entries to execute this query. See 'Security' under the Configure tab in the Gateway web page to learn more.

Security Zone	Role
Office	Any Administrator Operator Supervisor

Description

Query that grabs the contact information for customers.

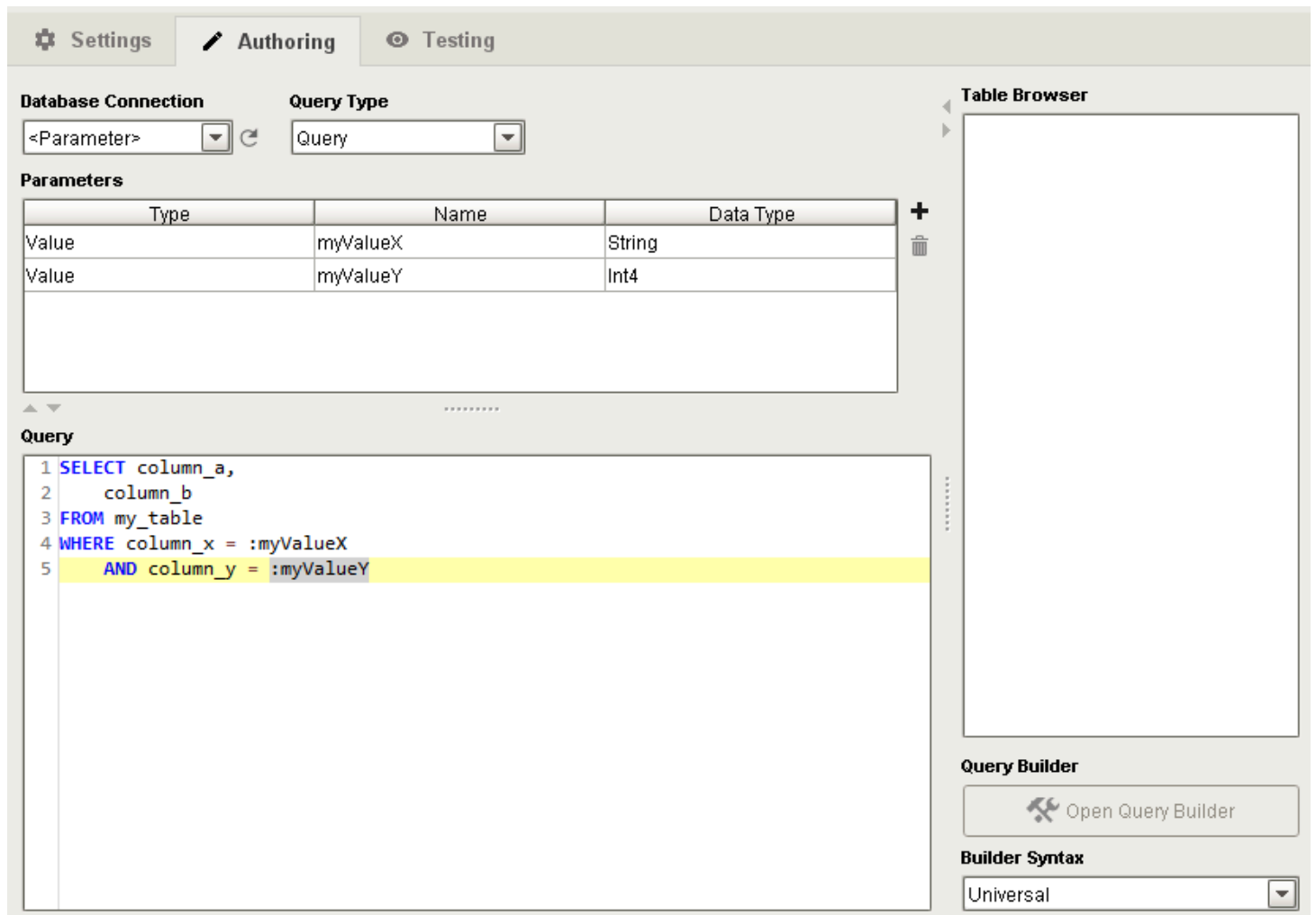
Caching

☐ Cache Query Results 1 Seconds

Item Name	Description
Enabled	Determines if the Named Query is enabled or disabled. A disabled Named Query may not be executed.
Security	Specifies a combination of Security Zones and Roles that may call the Named Query. Only roles in the projects user source will be available in the Role dropdown. Multiple rows may be configured to account for granular access restrictions (i.e., requests from Administrator roles originating from the Office security zone area could be allowed, while requests from same users in the plant floor zone could be denied). If the request does not match any of the specified zone and role combinations, then the query will not run. Additionally, if a ScalarQuery type has a Fallback value configured, that Fallback value will not be returned either: the query will not execute due to security settings, so there is never a chance for other errors to occur. Either the Security Zone or the Role (but not both) may be left blank. This means it is available to all objects of that type (i.e., with a blank Role and the "office" Security Zone, all roles in the "office" zone are valid).
Description	Allows you to give the Named Query a description.
Caching	Allows the Gateway to cache the results of the query. See the Named Query Caching page for more details.

Authoring

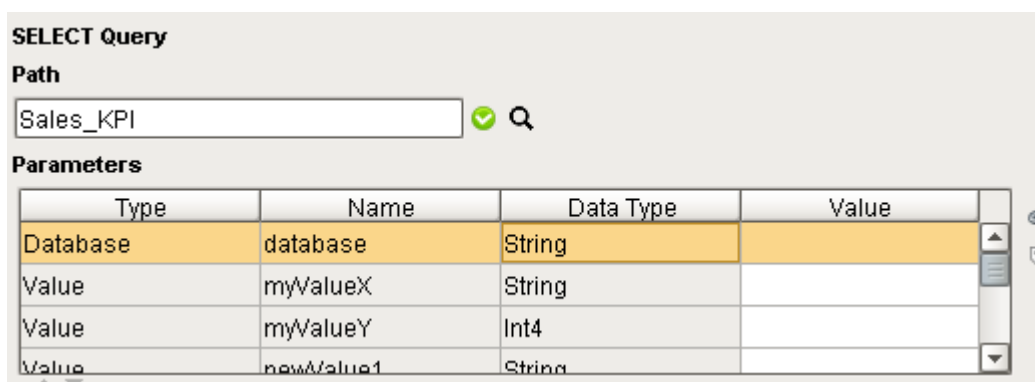
This section is where the query and parameters are created. There is also a Table Browser and [Query Builder](#) that can be used to help you to create your query.



Database Connection

The database connection the Named Query should run against. In addition to a list of the database connections configured in the Gateway, this dropdown contains two unique values: **<Default>** and **<Parameter>**.

- <Default>**: The query will execute against the project's default database connection.
- <Parameter>**: The query expects the database connection name to be passed in as a parameter when called. This allows you to use a dynamic database connection. Note: a parameter for the database connection does not need to be manually created in the Parameters section. Instead, the Named Query will have a special "Database" type parameter available when called. Below is an image of a Named Query binding that is utilizing the **<Parameter>** connection type.



Query Type

The type of query to execute. The following options are available:

- **Query**: Allows SELECT Queries and returns a full dataset. This type should be selected when running a SELECT statement that returns multiple rows or columns.
- **ScalarQuery**: Allows SELECT queries and returns a single value. This type should be used when running SELECT statements that only return a single value. The very first cell returned will be the only output.

This type is special in that a **Fallback** value may be defined. The Fallback value will be returned if the Named Query would return an error. Note, the **Fallback** parameter will not be returned if the request does not meet the security requirements.

New in 7.9.10 The Fallback value will now also be returned if there is no value returned from the query. Note, that this differs from a null value being returned in that the null value is deliberate.

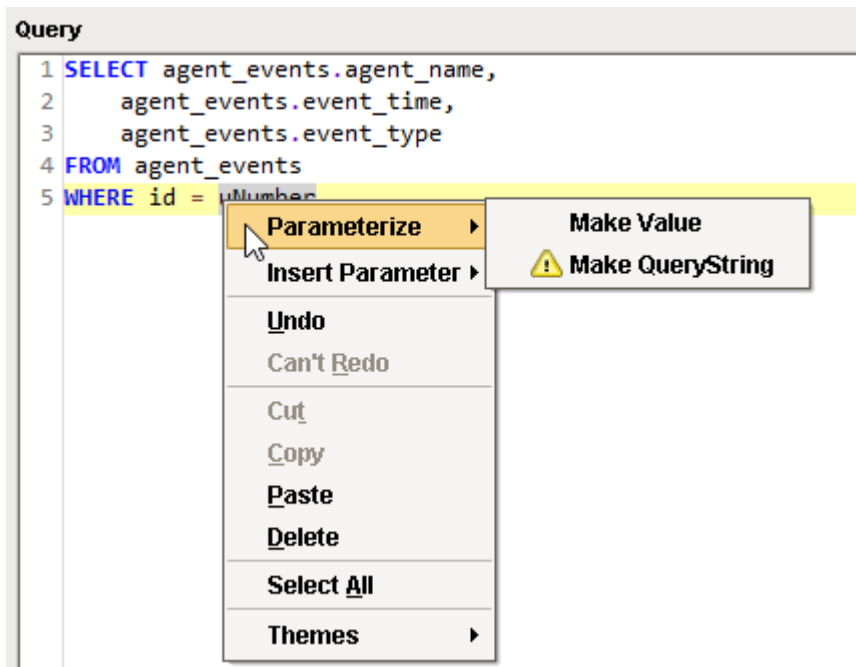
- **UpdateQuery**: Allows all **UPDATE** types of queries (queries that mutate or otherwise modify rows on a table) and returns the number of rows affected by the query. This type should be selected when modifying the database in some way, such as when running an INSERT, UPDATE, or DELETE query. |

Parameters

A table of the parameter names and types that will be used in the query. These parameters have three types, Value, QueryString, and Database. Most commonly, the **Value** type is used and can be accessed by using the **:paramName** notation. More details on this field may be found on the [Named Query Parameters](#) page.

Query

The query that will execute when the Named Query is called. You can type directly into this field or use the Table Browser on the right to get started. **Right-Clicking** inside this field will cause a popup menu to appear:



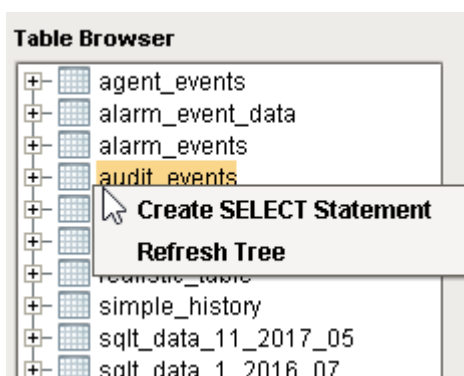
Most of the items on this menu are self-explanatory, but a few require special mention:

Parameterize: Contains two sub-menu items, which are detailed below. Note, that the sub-items will be **disabled** unless you right-click on some text that does not reference a parameter, table name, or column name in the query.

- **Make Value:** Turns the selected text into a value-type parameter. The new value-type parameter will appear in the **Parameter** table above the query.
- **Make QueryString:** Turns the selected text into a QueryString. The new QueryString will appear in the **Parameter** table above the query. Note that QueryStrings are susceptible to SQL injection attacks. Because of this, the **Make Value** option is recommend over this option.
- **Insert Parameter:** Quickly creates a reference to the selected parameter. This menu is an alternative to dragging-and-dropping from the Parameters table or typing the name of the parameter.

Table Browser

Provides a list of the tables in the selected **Database Connection**. Tables may be dragged into the Query field to quickly insert the name of the table. Additionally, **right-clicking** on a table in the list will cause a popup menu to appear:



- **Create SELECT Statement** will populate the Query field with a SELECT statement targeting the selected table or selected row of the table.
- **Refresh Tree** will refresh the Table Browser.

Query Builder

Opens the **Query Builder**, which provides an easy way to create SQL queries using a drag-and-drop interface. This button will be disabled if the **Database Connection** property is set to **<Parameter>****. This is the same Query Builder used in other places like the reporting data page.

Builder Syntax

Specifies the syntax the **Query Builder** should use. Contains syntax for many popular databases, and has a Universal selection that should work in most scenarios.

Testing

This section allows you to test your query without leaving the workspace. Fill in your values and click the **Execute Query** button to see your results.

Test Parameters

Type	Name	Data Type	Value
Database	database	String	
Value	myValueX	String	
Value	myValueY	Int4	

☐ Use Sample Size Limit

100



Execute Query

Export to CSV

Results

Item Name	Description
Test Parameters	Allows you to manually supply test values to the parameters to the queries. The table is populated by the Parameters field on the Authoring section.
Use Sample Size Limit	When checked, allows you to set the maximum number of rows the query will return while testing. This will not effect anything outside of this tab. This property is only enabled when the Query Type on the Authoring section is set to Query .
Execute Query	Runs the Named Query using the parameter values listed above. The Results area will display any results returned by the query.
Export to CSV	Exports the results of the query to a CSV file. The button becomes available after results are returned. Note that this will only return the values shown, check the Sample Size Limit when using this button.
Results	The results returned by the Named Query when testing. Populated by the Execute Query button.

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Named Query Parameters

New in 7.9.4

Parameters allow you to make Named Queries dynamic. They act as placeholders you can pass values into when requesting the query to execute. Other resources in Ignition can then pass arguments into the parameters. The exact implementation depends on what resource is requesting the Named Query, such as a [Named Query binding](#), a [Named Query Data Source](#), or the [system.db.runNamedQuery\(\)](#) function. Check out the [Named Query Example](#) page for a complete example on passing parameters into a named query.

Types of Named Query Parameters

There are three types of Parameters in Named queries. Each varies in usage.

Value

The Value type should be used whenever a Named Query needs a dynamic WHERE clause. These act like values passed to a prepared statement, meaning they can never be used to parameterize column or table names. However, they are resilient to SQL injection attacks.

SQL Query - Using a Parameter

```
SELECT * FROM mytable
WHERE name = :myParam
```

QueryString

QueryStrings are more flexible than the Value type in that they can be used to parameterize column and table names. However, their values are never sanitized, which causes them to be more susceptible to SQL injection attacks. When using QueryStrings, it is best to avoid situations where the user can manually type in the value that will be passed to the Name Query. Additionally, if you are using a QueryString for a string in the where clause, you would need to provide quotation marks.

```
SELECT {myColumnName} FROM mytable  
WHERE name = '{myName}'
```

Database

Database type parameters cannot be created manually. Instead, it is automatically created when the **Database Connection** dropdown on the **Authoring** section is set to **<Parameter>**. Additionally, this parameter is not used in the body of the query. This type allows you to parameterize the database connection when the Named Query is called. This way the Named Query can run against multiple database connections specified by the resource that made the request.

Parameters while Authoring a Named Query

Creating Parameters

New parameters can be created in the Authoring section of a Named Query by clicking the add button (+) next to the Parameters table.

The screenshot shows the 'Authoring' tab of a tool. At the top, there are three tabs: 'Settings', 'Authoring', and 'Testing'. Below the tabs, there are two dropdown menus: 'Database Connection' (set to '<Parameter>') and 'Query Type' (set to 'Query'). Below these is a 'Parameters' table with three columns: 'Type', 'Name', and 'Data Type'. The table has two rows: one with 'Value', 'myValueX', and 'String', and another with 'Value', 'myValueY', and 'Int4'. To the right of the table is a '+' button and a trash icon. Below the table is a 'Query' text area containing the text: '1 SELECT id, quantity FROM myTable Where id = :myValueX'.

Type	Name	Data Type
Value	myValueX	String
Value	myValueY	Int4

Query

```
1 SELECT id, quantity FROM myTable Where id = :myValueX
```

i NOTE

In Named Queries, Parameters are referenced by their name, so renaming the Parameter will require you to update it on any other resources that are using it.

Each parameter has three properties represented by different columns in the table. These may be edited by double-clicking on the cell you wish to modify:

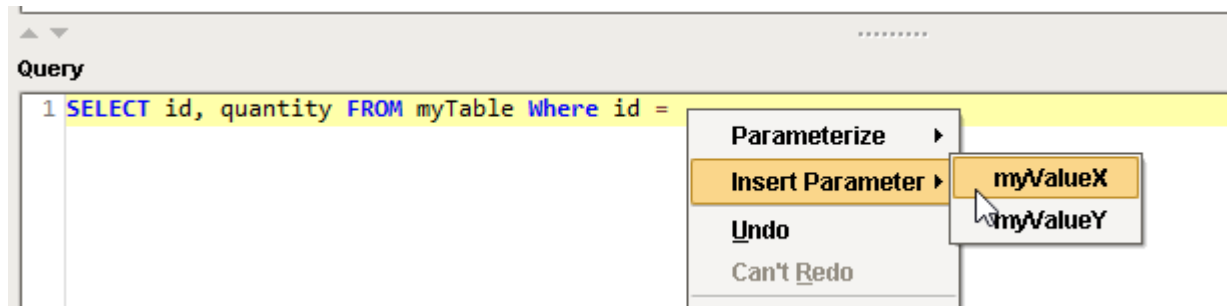
- **Type:** Changes the type between Value and QueryString.
- **Name:** Determines the name of the parameter, and how it will appear in the query. Names are **not** case-sensitive and must be unique. Additionally, they may only use letters, numbers, dashes and underscores.
- **Data Type:** Specifies the datatype of the parameter. The Type of the parameter determines which data types are available. Note, that QueryStrings may only be configured as strings, where as Value-type parameters have more types available.

Byte Arrays

New in 7.9.9 As of Ignition 7.9.9, parameters in a named query may now be configured with a byte array datatype, which can in turn be used to pass files into a named query.

Using Parameters in the Query

Once created, parameters can be inserted into the Query field by **drag-and-drop** from the Parameter table onto the Query field, or by using the **right-click menu** in the Query field.



Additionally, the parameters may be typed in manually, but the correct syntax must be used.

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Named Query Caching

New in 7.9.4

Named queries can opt-in to caching the results on the Gateway. This means if another request to the same Named Query comes in, the Gateway can return the cached result instead of having the database executing the query again. This will use more memory on the Gateway (to maintain the results), but could result in less queries running against the database.

Named Query caching is disabled by default, but can be enabled on the **Settings** section of each Named Query. If caching is enabled, the spinner and dropdown fields set the **lifespan** of the cache. Once the lifespan expires, the cache is invalidated. After the cache is invalidated, the next request for the Named Query will force the Gateway to re-execute the query and build a new cache.

Caching is especially useful for tables that are not updated often like recipe or inventory tables. Tables that update often like historical storage tables are bad candidates.

Settings

Authoring

Testing

Named queries allow a higher level of security for database queries and allow queries to be written once, tested, and easily reused throughout the project.

☒ Enabled

Security

Current zone and role must match one of these entries to execute this named query. See 'Security' under the Configure tab in the Gateway web page to learn more.

Security Zone	Role
Any	Any

Description

Caching

☒ Cache Query Results 5 Seconds

When Caching is Disabled

Update queries are not allowed to cache their results. With UPDATE or DELETE statements, the work will already have been performed after the first execution, and INSERT statements typically utilize different parameters each execution. Thus if the Caching setting is disabled, then this typically means that the **Query Type** in the **Authoring** tab is set to an **Update Query**.

Caching

☐ Cache Query Results 1 Seconds

Settings

Authoring

Testing

Database Connection

<Default>

Query Type

Update Query

Scenario

Consider the following:

- A Named Query is created. This runs a query to fetch data from an arbitrary table.
- A component in a project is configured to call the Named Query on a window.
- **Client_A** navigates to the window, which triggers a request for the Named Query be executed.
- Several seconds later, **Client_B** opens the same window, and needs the same results.

In this scenario, if caching was enabled on the Named Query, then **Client_B** would not cause another query execution (assuming both **A** and **B** passed the same values to the parameters). This would result in less network traffic between the Gateway and the Database, and less work for the Database. An example with two clients isn't exciting, but the same scenario with ~50 clients would mean a huge potential performance boost, especially if those requests were polling at a 5 second rate, and the cache period was configured to 5 seconds.

Considerations

The following are considerations that should determine whether or not caching the results of a Named Query is helpful.

Gateway Memory

Each time the Named Query is called, if the arguments passed differ from those used to create the current cache, a new cache is created. This means Named Queries that are frequently called with varying parameter values will create multiple caches. If the results are large datasets, this can result in a large amount of the Gateway Memory being tied up maintaining these caches. In this scenario, you will want to monitor the memory usage of the Gateway.

This is especially important to consider when dealing with queries that accept a timestamp parameter that uses an expression like `now()` that will return time to the current second. If 5 clients are opened with each one 1 second apart, they will all have different timestamps and create separate cache entries.

How often the Database Values are Updated

Once a cache is created, the Named Query will not look for any changes made to the database table until the query next executes. This means changes, such as adding a new row to the table, will not appear in the Named Query's results until the cache is invalidated. Data that changes often may not be a good candidate for caching. In this case, the lifespan of the cache should be set to a short amount of time, depending on how often the data may get updated.

How often the Named Query is Executed

If a Named Query is called frequently and there are not many updates to the table data, then there can be a huge performance benefit to caching the results. Fewer calls to the database result in less network traffic and better overall performance.

Cached Query Updates and Designer Values

Cached query results have some notable interactions in the Designer. Specifically:

- **Named Queries results executed in the Designer never cache:** This is because the Designer uses the live version of the query in the Designer instead of the saved Gateway version. This also prevents bad results being stored in the cache, and then appearing in the Clients.
- **Making changes to a Named Query and saving will invalidate all caches for that Named Query:** When a change is made to a Named Query and saved, the new query is pushed to the Gateway, which means all current caches are immediately outdated.

Scripting Functions to Clear Cache from a Named Query

New in 7.9.6

You can also clear the Named Query cache using the following scripting functions.

- **`system.db.clearNamedQueryCache`** - Clears the cache of a single Named Query.
- **`system.db.clearAllNamedQueryCaches`** - Clears the caches of all Named Queries.

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Named Query Conversions

New in 7.9.4



INDUCTIVE
UNIVERSITY

Named Query Conversion
Example

[Watch the video](#)

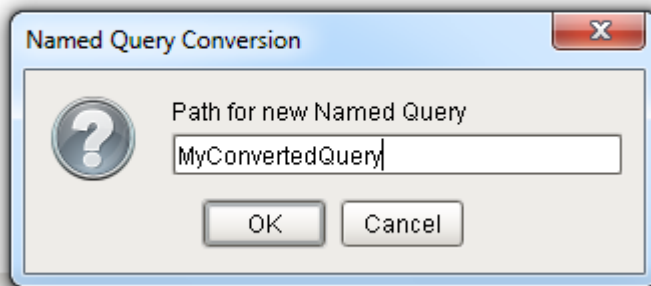
Converting SQL Queries to Named Queries

If you have a SQL Query binding on a component, it is actually really easy to convert it to a Named Query. Simply open up the binding and click the **Convert to Named Query** button.

A screenshot of the Ignition binding configuration dialog for a SQL Query. The dialog has a left sidebar with three sections: 'Tag' (with options Tag, Indirect Tag, Tag History), 'Property' (with options Expression, Property), and 'Database' (with options Named Query, DB Browse, and SQL Query, which is selected). The main area is titled 'SELECT Query' and contains a text box with the SQL query: `SELECT * FROM containers WHERE ContainerType = '{Root Container.Dropdown.selectedStringValue}'`. Below the query text box is a 'Database Connection' field. At the bottom, there are settings for 'Polling Mode' (Off, Relative, Absolute), 'Polling Rate' (Rate N/A, 0 sec), and 'Retain Rows' (false). A red rectangle highlights the 'Convert to Named Query' button at the bottom left of the main area. 'OK' and 'Cancel' buttons are at the bottom right.

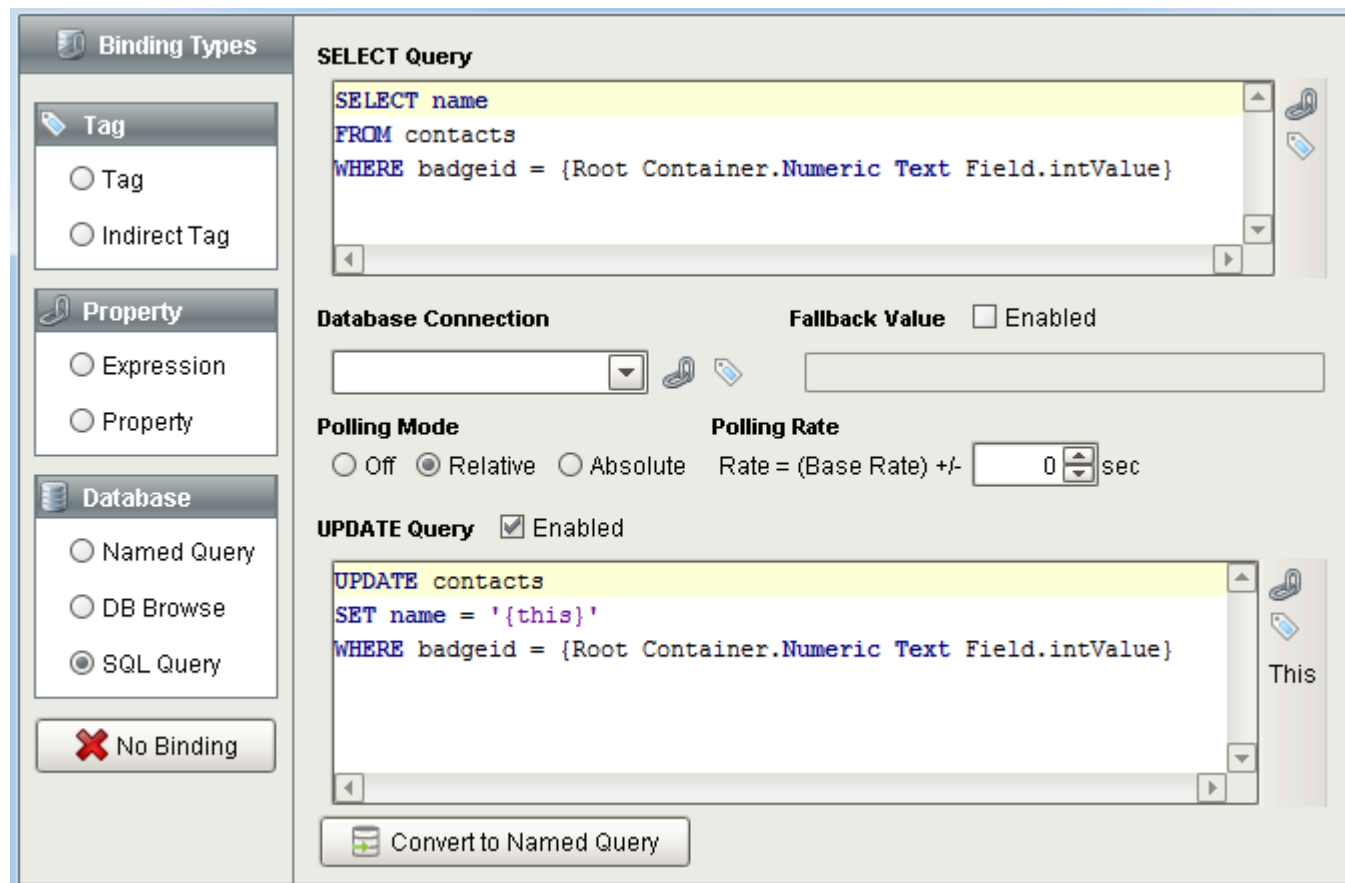
A popup will then appear, and have you put in a Path to the new Named Query.

```
ContainerType = '{Root Container.Dropdown.selectedStringValue}'
```

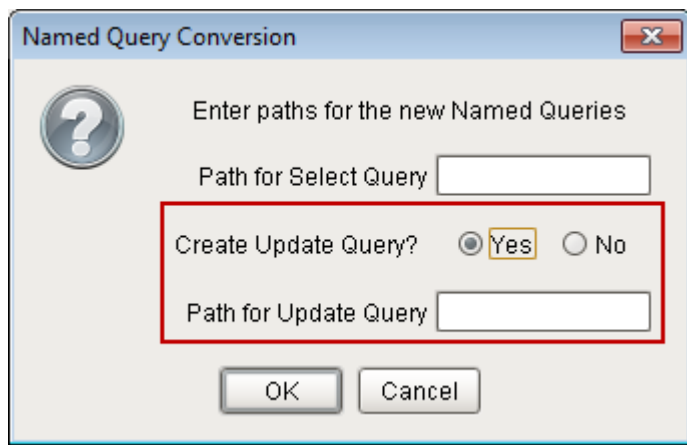


Converting with an Update Query

New in 7.9.6



In 7.9.6 and above, converting a SQL Query Binding to a Named Query will allow the option to create two Named Queries: one for the Select Query in the binding, and another for the Update Query.



Named Query Conversion

Enter paths for the new Named Queries

Path for Select Query

Create Update Query? ☒ Yes ☐ No

Path for Update Query

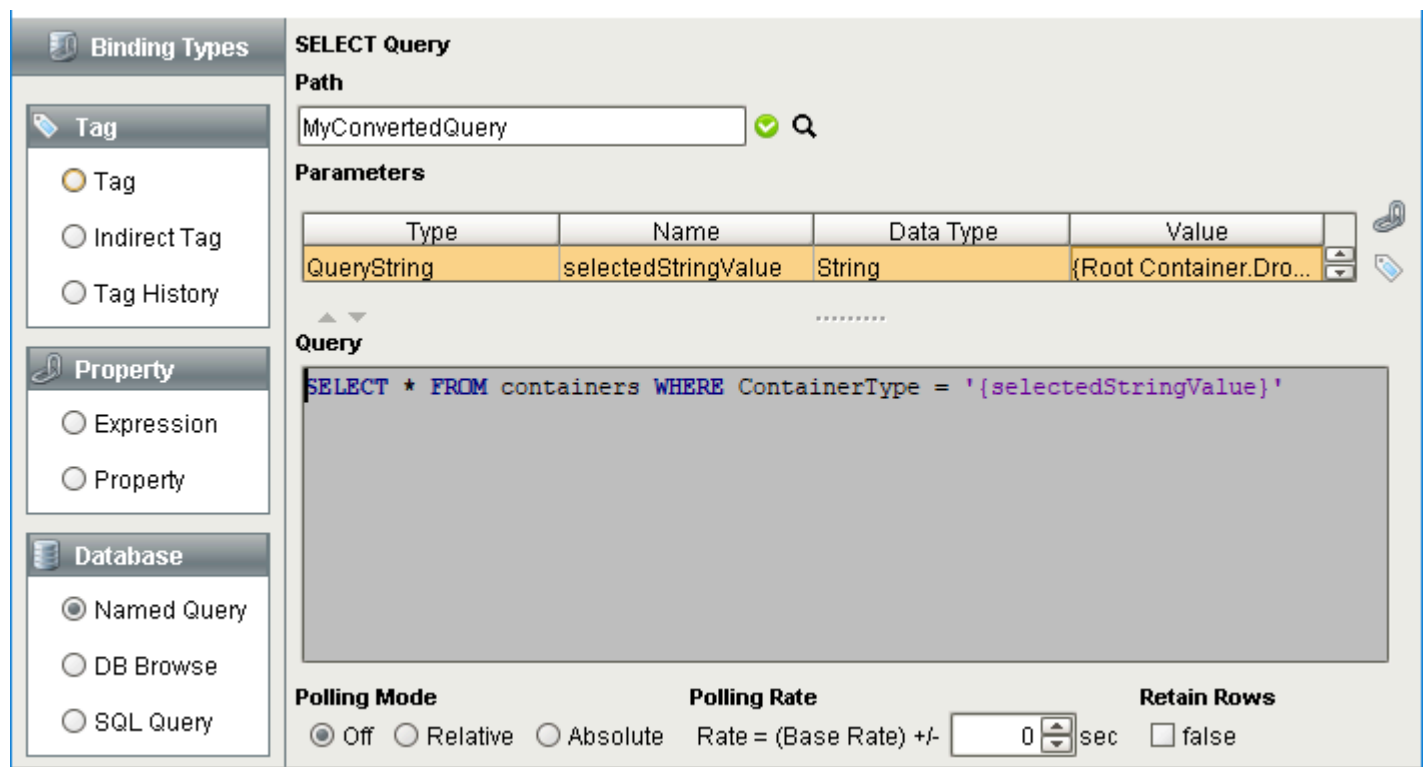
OK Cancel

WARNING FOR 7.9.5 AND PRIOR

When converting a SQL Query Binding that contains an UPDATE Query in 7.9.5 and prior, an Update query will not be generated. Thus, it was possible to lose the Update Query upon conversion. If using these versions of Ignition, it is highly recommended to manually create the Update Named Query before pressing the Conversion button. As of 7.9.6, this is no longer an issue, and the **Convert to Named Query** button may freely be used.

After confirming the conversion, the Window will show the query/queries. Click the **OK** button to save this change.

When converting in this manner, all parameters will be created at **QueryStrings**. It is highly recommended that you modify your new Named Queries so that these values become **Value Parameters** instead, as mentioned later on this page.



Binding Types

- Tag
 - ☒ Tag
 - ☐ Indirect Tag
 - ☐ Tag History
- Property
 - ☐ Expression
 - ☐ Property
- Database
 - ☒ Named Query
 - ☐ DB Browse
 - ☐ SQL Query

SELECT Query

Path
MyConvertedQuery  

Parameters

Type	Name	Data Type	Value
QueryString	selectedStringValue	String	{Root Container.Dro...}

Query

```
SELECT * FROM containers WHERE ContainerType = '{selectedStringValue}'
```

Polling Mode ☒ Off ☐ Relative ☐ Absolute

Polling Rate Rate = (Base Rate) +/- sec

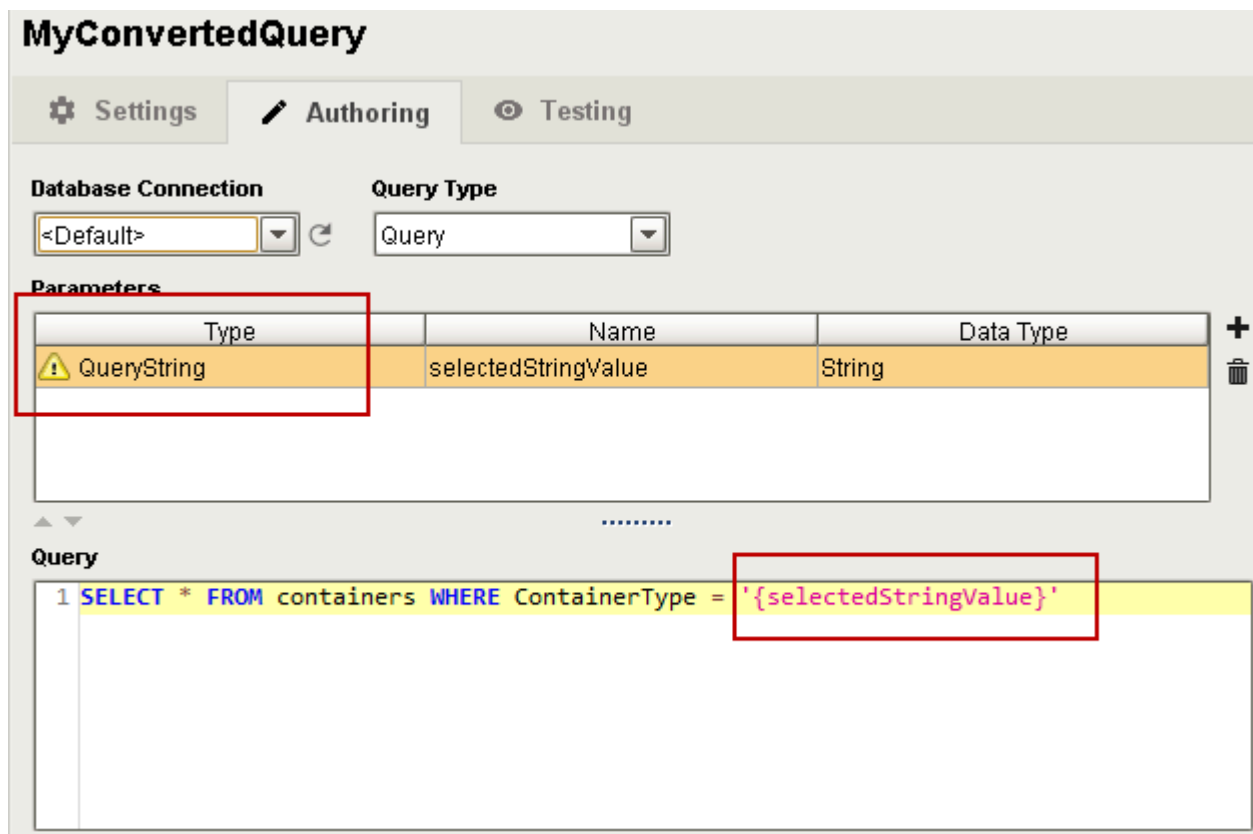
Retain Rows ☐ false

Modifying Converted Query to Use Parameters

When converting a query to a Named Query, it is **strongly** recommended that you go back into the query and convert the parameter from a **QueryString** type to a **Parameter** type. To do this, you first need to change all QueryString type parameters to the Parameter type. This is as simple as selecting Parameter from the dropdown under Type.

Now that your parameter(s) types have been modified, the second thing you need to do is modify syntax in the query to use the parameter(s). The Parameter and QueryString types are referenced differently in the query, so you will need to ensure that you modify how the parameter is referenced. In addition because the Parameter type works like a prepared statement, it does not need any quotation marks around any string type parameters like a QueryString would, so all quotation marks around parameters should be removed from the query. See the images below for an example.

From this:



The screenshot shows the 'MyConvertedQuery' interface with three tabs: Settings, Authoring, and Testing. The 'Authoring' tab is active. Under 'Database Connection', the dropdown is set to '<Default>'. Under 'Query Type', the dropdown is set to 'Query'. The 'Parameters' section contains a table with one parameter:

Type	Name	Data Type
QueryString	selectedStringValue	String

The 'Type' column header and the first row are highlighted with a red box. Below the parameters table is a 'Query' section with a text area containing the following SQL query:

```
1 SELECT * FROM containers WHERE ContainerType = '{selectedStringValue}'
```

The parameter placeholder in the query is highlighted with a red box.

To this:

MyConvertedQuery

⚙ Settings

✎ Authoring

👁 Testing

Database Connection

<Parameter> 

Query Type

Query 

Parameters

Type	Name	Data Type
Value	selectedStringValue	String

+

🗑

Query

1 SELECT * FROM containers WHERE ContainerType = :selectedStringValue

Once the syntax in the query has been modified, test it out in the **Testing** section at the top to make sure everything works correctly. If so, then you have successfully converted to a Named Query.

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Using Named Queries - Example

New in 7.9.4

A Named Query Example

Here we will go over the steps necessary to put together a basic named query. This example will create a Named Query that uses a single parameter to run a select query, and then add a Table to a window and create a binding that uses our new Named Query.

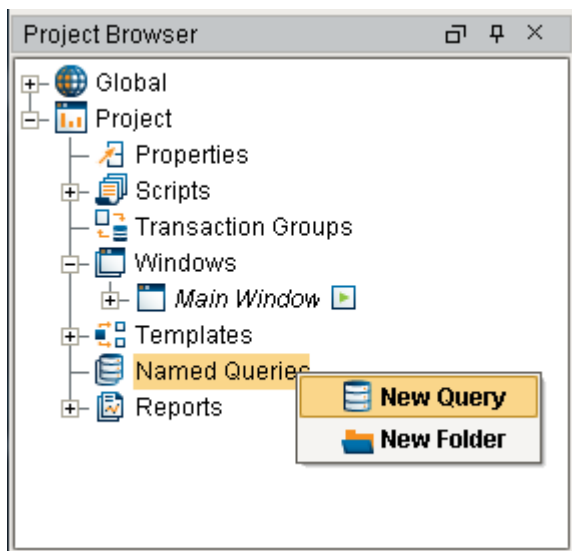
We are going to be querying a table in the database that holds information about products stored in storage bays. The database table is named "containers" and has the following structure.

id	StorageBay	ContainerType	ItemName	Weight	Time
1	1	Jug	Vanilla	25.2	2017-06-25 15:58:47
6	1	Mason Jar	Chocolate	12.3	2017-06-26 16:05:27
18	2	Tray	Swiss Cheese	88.8	2017-06-25 01:21:31
22	3	Tray	Cheddar Cheese	54.7	2017-06-25 03:52:16
23	3	Basket	Strawberry	36.8	2017-06-25 15:56:03
31	3	Jug	Whole Milk	80.1	2017-06-27 09:51:31
32	3	Jug	Fat Free Milk	76.9	2017-06-27 09:52:52

If you want to follow along with the example, feel free to make a database table that looks similar to this one and add as many rows of data as you would like, otherwise, you can use your own and substitute in the proper column names from your database.

Creating a Named Query and Adding Security

1. Start by opening up the Designer and [loading a project](#).
2. Locate the Named Query section of the project browser, right click on it, and select the New Query option.



3. You should now have a fresh Named Query that you can rename whatever you want. It is named **FirstNamedQuery** for this example.

The screenshot shows the 'FirstNamedQuery' settings window. The 'Settings' tab is active, displaying a table for security zones and roles. The 'Tag Browser' is also visible on the left.

FirstNamedQuery

Settings | **Authoring** | **Testing**

Named queries allow a higher level of security for database queries and allow queries to be written once, tested, and easily reused throughout the project.

☒ Enabled

Security

Current zone and role must match one of these entries to execute this named query. See 'Security' under the Configure tab in the Gateway web page to learn more.

Security Zone	Role
Any	Any

Description

Caching

☐ Cache Query Results

4. To setup security on the Named Query, set required Security Zone and Role combinations in the Security table of the Settings tab. You can leave this blank if you don't have **roles** or **zones** set up yet.
5. If multiple security combinations are required, use the plus (+) button to add additional rows.

Security

Current zone and role must match one of these entries to execute this query. See 'Security' under the Configure tab in the Gateway web page to learn more.

Security Zone	Role
Any	Administrator
Office	Any Administrator Guest Manager Operator





Building the Query

1. Click on the **Authoring** tab. Here is where we do most of the work.
2. Under **Database Connection**, we need to select a database connection that this named query will use. I selected **<Default>**.
3. For the **Query Type**, we can decide what type of query this will be. For this example, I am running a select query that will return a dataset, so I selected **Query**.
4. In the Parameters section, we can decide on a list of parameters that will be used in this query. My query is fairly simple, and will only use a single parameter. Click the plus (+) button to add a new parameter and set the following values:
 - **Type:** Parameter
 - **Name:** BayNum
 - **Data Type:** Integer
5. In the Query section below, we will construct our query using the Table Browser.
 - i. Right click on the containers table in the Table Browser on the right, and click on **Create SELECT Statement**. This will populate our query field with a basic select all statement.
 - ii. Type into the **Query** field and add the following WHERE clause: **"WHERE StorageBay = "**
 - iii. Now drag the **BayNum** parameter from the Parameters table to the end of the query you just typed. Notice **":BayNum"** will be added at the end of the query.

Settings
Authoring
Testing

Database Connection
Query Type

<Default>
Query

Parameters

Type	Name	Data Type
Value	BayNum	Int2

Query

```

1 SELECT * FROM containers WHERE StorageBay = :BayNum

```

Using the Query

1. In the Project Browser, create or open a **Main Window**.
2. Drag a Dropdown List component onto the window.
3. The Dropdown List is where we will be able to select a Bay Number to use as our Named Query's BayNum Parameter. Use the **Dataset Viewer** to set the **Data** property of the Dropdown to look like this:

Value	Label
1	Bay 1
2	Bay 2
3	Bay 3

4. Now drag a **Table** component onto the window. We can setup a Named Query binding on the Data property.
5. Click on the binding icon for the **data** property and select the **Named Query** binding type.
6. Set the Path property by clicking on the magnifying glass (🔍) button and selecting your new Named Query from the list. In my case it is **FirstNamedQuery**. Alternately, we can type the name in.

- Highlight your **BayNum** Parameter (in the Parameters table) and click on the property binding () icon. We want to select the **Selected Value** property of our Dropdown List.
- Finally, we want to ensure the Polling Mode is set to **Off**. This means the query will not run continuously, but will only run when it changes such as when a new bay is selected from the dropdown.

 Property Binding: Root Container.Table X

Binding Types

 **Tag**

☐ Tag

☐ Indirect Tag

☐ Tag History

Property

☐ Expression

☐ Property

Database

☒ Named Query

☐ DB Browse

☐ SQL Query

SELECT Query

Path

FirstNamedQuery  

Parameters

Type	Name	Data Type	Value
Value	BayNum	Int2	{Root Container.Dropdown.selectedValue}

Query

```
SELECT * FROM containers WHERE StorageBay = :BayNum
```

Polling Mode ☒ Off ☐ Relative ☐ Absolute **Polling Rate** Rate = (Base Rate) +/- sec **Retain Rows** ☐ false

OK Cancel

- Click the **OK** button to save your binding, and put your Designer into **Preview Mode** to test it. We can then make a selection (i.e., Bay 3) with our dropdown, and see our table populate with data.

id	StorageBay	ContainerType	ItemName	Weight	Time
22	3	Tray	Cheddar Cheese	54.7	Jun 25, 2017 3:5...
23	3	Basket	Strawberry	36.8	Jun 25, 2017 3:5...
31	3	Jug	Whole Milk	80.1	Jun 27, 2017 9:5...
32	3	Jug	Fat Free Milk	76.9	Jun 27, 2017 9:5...

Bay 3 

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Queries in Scripting

Overview

In addition to using a binding, queries can be called from a Python script. This typically involves using one of Ignition's built-in [scripting functions](#). This page presents several approaches to interacting with a database from a Python Script. See the [Scripting](#) section for more information about using Python.

Named Queries

Named Queries may be called from a Python script using the [system.db.runNamedQuery](#) function. Named Queries can execute as a Prepared Statement when not utilizing [Query String parameters](#), so they can offer a secure approach to calling a query from a script.

Additionally, Named Queries are incredibly easy to call from a script, since you only need to know their name, and don't have to bother with SQL syntax in the script. This is another way to protect your database from other users in the Designer, as you can [Protect Named Queries](#).

Python - Calling a Named Query

```
namedQuery = "Add New Order"
parameters = {"accountId":123, "productName":"Bananas"}

system.db.runNamedQuery(namedQuery, parameters)
```

For more information, refer to the [Named Queries](#) section.

Prepared Statements vs Standard Statements A prepared statement is a feature in a database that executes a parameterized query. One of the main reasons to utilize a prepared statement is because they are resilient to SQL Injection Attacks. Because of this, we **highly recommend** you utilize Prepared Statements over Standard Statements, but Named Queries are the most secure. Especially in cases where the users can type values that will be included in the query.

Prepared Statements typically involve passing two pieces of information to the database:

- A query string that contains placeholders to represent where arguments will be passed in later. These are represented in the query as question mark characters ("?")
- A series of arguments to replace the placeholder characters in the static query.

Standard Statements vs Prepared Statements

Typical SQL insert queries look like the following:

SQL - Standard Statement

```
INSERT INTO orders (account_id, product_name) VALUES (123, 'Bananas')
```

A Prepared Statement instead looks like the following. Note, that the placeholders do not require quotation marks even when a string will be passed in.

SQL - Prepared Statement

```
INSERT INTO orders (account_id, product_name) VALUES (?, ?)
```

How do I Call a Prepared Statement?

Prepared Statements can be called from a script using specific functions. Typically, they contain "Prep" in the name such as [system.db.runPrepQuery](#), or [system.db.runPrepUpdate](#). When in doubt, take a look at the sub pages in the system.db section.

There are typically two required parameters with these functions: a string literal that represents the query, and a list of parameters.

Python - Calling an Insert Query as a Prepared Statement

```
query = "INSERT INTO orders (account_id, product_name) VALUES (?, ?)"
args = [123, "Bananas"]
system.db.runPrepUpdate(query, args)
```

Notable Prepared Statement Functions

- [system.db.runPrepQuery](#): Should be used when SELECT statements are used, as it returns a dataset containing the results. This function is effectively "read-only", as it does not manipulate

data in the database.

- **system.db.runPrepUpdate**: Should be used when manipulating the database in some way. When executing a statement that utilize INSERT, UPDATE, or DELETE, the runPrepUpdate function should be called. Note that it does return the number of rows affected, so the return value can be used to keep track of how much of an impact the query had.
- **system.db.runScalarPrepQuery**: Like runPrepQuery above, but only returns the first column of the first row: i.e. a single value is returned instead of a full dataset. This is useful when using an aggregate function of some sort to return a count or total, as it saves your script the work of extracting the value from the full dataset that runPrepQuery normally returns.

Standard Statements

Queries can be called as a Standard Statement (a statement that that isn't a Prepared Statement) by using the **system.db.runQuery** and **system.db.runUpdateQuery** functions. However, these are susceptible to **SQL Injection attacks**, and should be avoided where possible: especially when users have access to a keyboard and can directly type values that will be used in the query.

Calling a Standard Statement involves building the entire query as a single string, and passing the string on to our Standard Statement functions.

Python - Calling an Insert Query as a Standard Statement

```
query = "INSERT INTO orders (account_id, product_name) VALUES (%i, '%s')" % (123, "Bananas")
system.db.run(query)
```

Notable Standard Statement Functions

- **system.db.runQuery**: Executes a SELECT statement, returning a result set as a dataset.
- **system.db.runUpdateQuery**: Executes a statement that manipulates the database in some way. Should be used with INSERT, UPDATE, and DELETE statements.
- **system.db.runScalarPrepQuery**: Similar to runQuery, except only a single value is returned: the first column of the first row. Generally used in conjunction with SELECT statements that contain an aggregate function.

Stored Procedures

If your database administrator has already configured Stored Procedures for you to use, then they can easily be called from a Python Script. Using Stored Procedures in a script typically involves two main

steps:

1. A SProcCall object is created with the [system.db.createSProcCall](#) function. The SProcCall object contains several functions that can be used to register parameters, and access the results set returned by the Stored Procedure after it has been executed.
2. The [system.db.execSProcCall](#) function must be used to execute the Stored Procedure.

Python - Creating and Executing a Stored Procedure

```
# Create a SProcCall object, which will be used to configure parameters on the  
Stored Procedure, and then executed.  
myCall = system.db.createSProcCall("insert_new_order")  
  
# Register parameters on the SProcCall object.  
myCall.registerInParam(1, system.db.INTEGER, 123)  
myCall.registerInParam(2, system.db.VARCHAR, "Bananas")  
  
# Execute the Stored Procedure.  
system.db.execSProcCall(myCall)
```

Take a look at the [SQL Stored Procedures](#) page for more details.

Transactions

A SQL Transaction can also be executed from a script. For the unfamiliar, a Transaction is a batch of statements that will be executed together, and either succeed or fail as a group. Note, that the statements executed in the Transaction are not visible by other connections in the database until you commit them.

Transactions typically involve several steps:

1. Call [system.db.beginTransaction](#). This returns a **transaction identifier** that can be used with other statements. Using this identifier is how you specify that a statement should be included in the transaction.
2. Start calling other statements with other functions, such as [system.db.runPrepUpdate](#). The function's "tx" parameter will be passed the **transaction identifier**.
3. Commit or Rollback the transaction. Use [system.db.commitTransaction](#) to commit, and [system.db.rollbackTransaction](#) to rollback. These options are essentially the same as applying or canceling the results of the queries. Committing will make the updated results available to other database connections.

4. Close the Transaction once you're done, which can be accomplished with the `system.db.closeTransaction` function. This invalidates the **transaction identifier**.

Python - The SQL Transaction Workflow

1) Begin the transaction. This returns a transaction identifier that we can use with other statements.

```
transactionId = system.db.beginTransaction(timeout = 5000)
```

2) Now we can execute statements. Because we want these to run as part of the transaction, we need to include our identifier.

```
query = "INSERT INTO orders (account_id, product_name) VALUES (?, ?)"
```

```
args = [123, "Bananas"]
```

```
system.db.runPrepUpdate(query, args, tx = transactionId)
```

3) We can continue to add statements, but in this case we'll commit them. We could instead rollback if there was an issue with our previous statement.

```
system.db.commitTransactions(transactionId)
```

4) We're done, so close the Transaction.

```
system.db.closeTransaction(transactionId)
```

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Ignition Auto-Generated Tables

Overview

Ignition has a lot of systems built in that will query the database automatically without requiring you to build a query. These systems automatically create the necessary tables in the database, insert the relevant data, and even query it back out for you. However, because this data is all stored in simple database tables, it can be manually accessed using queries to customize how you see the data.

CAUTION

These tables are setup in very specific ways that the system understands. Altering them in any way may make it so the system can no longer automatically interact with the tables. While it can be useful to manually query out the data, we advise caution when attempting to alter the data or tables in any way. We recommend taking a backup of the database tables before making manual changes to them, with the understand that manually altering the data or tables is done at your own risk.

Tag History

The **Tag History** system utilizes six different tables in the database. Each of the tables are described later on this page.

Table Name	Table Description	Column References
sqlt_data_x_x_x	This table stores the raw Tag data. There will be multiple tables that fit this format depending on the name of the Gateway and the date. (sqlt_data_1_2018_01 This table is storing data from the Gateway with an id of 1, for the year 2018, for the month of January)	sqlt_data_x_x_x.tagid = sqlth_te.id
sqlth_te	This table stores the non-data details of each Tag.	sqlth_te.scid = sqlth_scinfo.id
sqlth_scinfo	This table stores scan class information.	sqlth_scinfo.drvid = sqlth_drv.id
sqlth_sce	This table stores start and end times for scan classes.	sqlth_sce.scid = sqlth_scinfo.id
sqlth_partitions	This table stores start and end times for each sqlt_data table.	sqlth_partitions.drvid = sqlth_drv.id
sqlth_drv	This table stores information about the drivers of the historical data.	none

sqlt_data_x_x_x

This is the central table that stores the core Tag values. The system stores data in the corresponding xxx value column and leaves the others set to NULL. By default, the partition size is set to 1 month and the table name will appear as sqlt_data_1_yyyy_mm. When you first start storing historical data with this system, there will be only one table.

Column Name	Data Type	Notes
tagid	int	Unique id of the Tag. References sqlth_te.
intvalue	int	Holds the value of the Tag if it is datatype 0, NULL otherwise.
floatvalue	double	Holds the value of the Tag if it is datatype 1, NULL otherwise.
stringvalue	string	Holds the value of the Tag if it is datatype 2, NULL otherwise.
datevalue	date	Holds the value of the Tag if it is datatype 3, NULL otherwise.
dataintegrity	int	Quality of the Tag for this timestamp. 192 is Good Quality, anything else is bad. See Tag Quality Overlays.
t_stamp	long	Unix Timestamp (milliseconds since epoch) for this value.

sqlth_1_data

This is the data table for historian providers that have disabled multiple partitions (by unchecking the "Enable Partitions" setting). The structure of this table matches the sqlt_data_x_x_x table mentioned above.

sqlth_drv

This table defines the historical providers that are storing data to the database.

Column Name	Data Type	Notes
id	int	Unique id of the driver.
name	string	Name of the driver. This is usually the project name.
provider	string	Name of the Tag provider.

sqlth_partitions

This table defines the "partitions" (tables) that are used to store data, and what time frames they cover. Partitioning in the history system splits data across multiple tables in a way that is compatible with all database systems, making certain maintenance tasks easier. The query system does not expect any particular partition configuration, it simply consults this table for table-to-time associations, and then queries the resulting tables for data.

Column Name	Data Type	Notes
pname	string	The name of the table that contains this partition's data.
drvid	int	The id of the driver that owns this data table. Partitions are created per driver to keep data separate. References sqlth_drv.
start_time	long	Unix Timestamp (milliseconds since epoch) for the earliest time covered by this partition.
end_time	long	Unix Timestamp (milliseconds since epoch) for the end time covered by this partition.
blocksize	int	The size, in milliseconds, of time covered by each entry. This is used by "pre-processed" partitions and would be 0 for normal data partitions.
flags	int	Additional flags that affect how the partition is used. 1 = No seed query support. The system will not execute "bounding value" (or "seed") queries against the table. Useful for database engines that do not support indexing (such as MySQL Archive engine), as these operations can become very time intensive.

sqlth_sce

A record of executions of scan classes, defined in sqlth_scinfo. This table is primarily used to determine when the system was running correctly. The end_time of an entry is updated, as long as it falls within 2*rate of the last execution. If not, a new entry is made, and data queried will be "stale" during the gap.

Column Name	Data Type	Notes
scid	int	Id of the scan class execution entry. References sqlth_scinfo
start_time	long	Unix Timestamp (milliseconds since epoch) for the first execution of this scan class.
end_time	long	Unix Timestamp (milliseconds since epoch) for the latest execution of this scan class.
rate	int	The rate (in milliseconds) of execution. There are no entries for the 'Execute on Value Change' option.

sqlth_scinfo

This table defining scan class information for executing Historical Tags.

Column Name	Data Type	Notes
id	int	Unique id of the Scan class.
scname	string	Name of the scan class. " <i>exempt</i> " for 'Execute on Value Change' option.
drvid	int	The driver this scan class uses. References sqlth_drv.

sqlth_te

This table stores the core Tag definitions. The sqlt_data_x_x_x tables use these ids to reference values

Column Name	Data Type	Notes
id	int	Unique id of the Tag.
tagpath	string	Path of the tag in the Tag Provider. ie: Folder1/tag1
scid	int	The scan class this Tag is storing values with. References sqlth_scinfo.
datatype	int	The type of value for this Tag. 0: int, byte, short, boolean 1: float, double, long 2: string 3: date
querymode	int	Which internal mode to use for returning data.
created	long	Unix Timestamp (milliseconds since epoch) for when the Tag was created.
retired	long	Unix Timestamp (milliseconds since epoch) for when the Tag was retired (deleted/renamed). This value is NULL while the Tag is active.

Alarm Journal

The **Alarm Journal** system utilizes two different tables in the database:

Table Name	Table Description	Column References
alarm_events*	This table stores every event (active, cleared, acknowledged) that happened to any alarms that fit within the Journal filter parameters. Each row is a new event	alarm_events.id = alarm_event_data.id
alarm_events_data*	This table stores unique information pertaining to each event. Each row is a specific property of a specific event, so alarm events with multiple properties will have multiple rows in the table.	none

See the [Journal Properties and Tables](#) page for more information regarding all of the columns in the tables.

 INFO

*The names of the tables are completely configurable in the Journal settings in the Gateway. The default table names are used in the table.

Authentication

The [Database Authentication](#) system utilizes six different tables in the database:

Table Name	Table Description	Column References
scada_users*	This table stores each user contained within the user source, along with basic user information. Each row is a new user.	none
scada_roles*	This table stores all of the possible roles within the user source. Each row is a new role.	none
scada_user_rl*	This table stores a mapping of users to roles. Each row is a user and a paired role, so users with multiple roles will have multiple rows in the table.	scada_users_rl.user_id = scada_users.id scada_users_rl.role_id = scada_roles.id
scada_user_sa*	This table stores a list of all upcoming schedule adjustments for each user. Each row is a new schedule adjustment, so users with multiple schedule adjustments will have multiple rows in the table.	scada_user_sa.user_id = scada_users.id
scada_user_ci*	This table stores a list of all contact information items for each user. Each row is a new contact information item, so users with multiple contact information items will have multiple rows in the table.	scada_user_ci.user_id = scada_users.id
scada_user_ex*	This table stores a list of all extra properties for each user, with properties and values stored 1 for 1. Each row is a new property and value pair, so users with multiple extra properties will have multiple rows in the table. Extra properties are added by modules that want to associate data with a user, such as the Voice Notification Module, which adds a Security PIN setting.	scada_user_ex.user_id = scada_users.id

! INFO

*The prefix of the tables are configurable in the User Source settings in the Gateway. The default prefix of "scada_" is used in the table

Audit Log

The **Audit** system utilizes one table in the database:

Table Name	Table Description	Column References
AUDIT_EVENTS*	This table stores each auditable event (save, publish, edits, etc.) that has happened for each project or system that has auditing enabled. Each row is a new event.	none

 CAUTION

*The names of the tables are completely configurable in the Audit settings in the Gateway. The default table names are used in the table.

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Common SQL Tasks

This section contains examples for items we identified as "common" tasks. When first starting out with **SQL in Ignition**, many users are looking for examples in order to grasp concepts, or examples to possibly use in their projects. Additionally, this section aims to demystify some of the more complex or abstract tasks that our users may encounter when working with SQL in Ignition.

The examples in this section document several types of SQL tasks that may also touch many other areas of Ignition. While these examples are typically focused on a single concept or end result, they can easily be expanded or modified. In essence, they serve as a great starting point for users new to Ignition, as well as experienced users that need to get acquainted with a new or unfamiliar feature.

Below is a list of common SQL tasks related to using SQL in Ignition.

Filter Data in a Table

The **Filter Data in a Table** section explains how using a simple Dropdown box and making some modifications to the SQL query can allow users to dynamically filter data coming into the table to only show a specific subset of data, or all of the data. This section also includes variations of the SQL query, thus returning different filtered results to the table.

Add Data to a Database

A common method of **adding data to a database** from within a client is using a Button component that executes an SQL query. This section provides an SQL script showing how to collect relevant properties and insert them into a database, as well as how to setup your components on a window.

Edit Data in a Database

Editing data in a database is very similar to how we add data to a database. Users can edit data within a table on the screen in realtime, and the changes are then pushed back to the database table.

This section goes over the different ways to edit data in the database depending if you're using a **Table** component or **Power Table** component.

Refreshing SQL Data on a Component

Sometimes, it may be unnecessary to have a table constantly updating and requerying the database for data. By using the **system.db.refresh** function, we can turn Polling "Off" on our query, and have a button that allows us to manually update the table with new data when the Button is pressed. Alternatively, we can add the **system.db.refresh** function at the end of a script to refresh the newly entered data automatically without pressing a button.

Editing Multi-Selected Rows from Table

Tables have the ability to select and **edit multiple rows** at a time. Oftentimes, this can be used to manipulate multiple rows simultaneously, such as deleting all of the selected rows at once.

Storing PDF Files in a Database

The database is a powerful tool and can allow you to store files such as PDFs in a database table. The **Storing PDF Files in a Database** section explores how we can take a PDF file and store it in the database table, as well as how to pull it back out and display it in the **PDF Viewer** component.

Simple Database Editor

It may be beneficial to have a frontend to a database table built right into an Ignition project that allows you to control the data in the table without having to go into the database's built-in frontend, such as MySQL workbench. In the **Simple Database Editor** section, you will learn how to build a simple database table editor on a window that can add, edit, and delete data from the database.

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Filter Rows in a Table



INDUCTIVE
UNIVERSITY

Filter Data in Table

[Watch the video](#)

Creating the Components

Filtering table data by using a Dropdown box is possible if the **Table** component's **Data** property is bound to a SQL query. We can create a dynamic `WHERE` clause that will allow us to select all the data or a specific subset of it. To implement this solution, you must have a **Table** component and a **Dropdown List** component on a window. With a Dropdown component, we can create an option for each way of filtering the data, each with a unique value associated with it. An additional "all" entry can also be added, with its own unique value.

We can then use the value of the Dropdown to drive the query. When you select a different option from the Dropdown component the binding gets re-evaluated on the table's data binding resulting in the query executing the WHERE clause with the new parameters. You may want to turn Polling Mode off on the tables Data property binding in order to limit the periodic querying of the database.

Here we can use a value of 1 to show only entries which match that filter column value.

Pseudocode - Filtering for Area 1

```
SELECT * FROM table WHERE filterColumn = 1
```

While the following query will return everything, but will still evaluate the query's WHERE clause.

Pseudocode - Return All Rows

```
SELECT * FROM table WHERE 0=0
```

In this case $0=0$ will always evaluate as true, therefore, the query will return every row.

We can combine the logic of each WHERE with an OR into a single query, and substitute in our Dropdown value. The SQL Query binding would then look something like this:

SQL - Where Clause Combined with OR Condition

```
SELECT * FROM table WHERE filterColumn = {dropdown value} OR 0 = {dropdown value}
```

This way, the WHERE clause will only be true when the filter column matches with our selected option, or it will return all rows, if we setup our "all" option to have a value of 0.

Example - Filtering Data on an Area Number

We can put together an easy example on how this might work with data that stores what machines are in which areas. Our data table should look something like this:

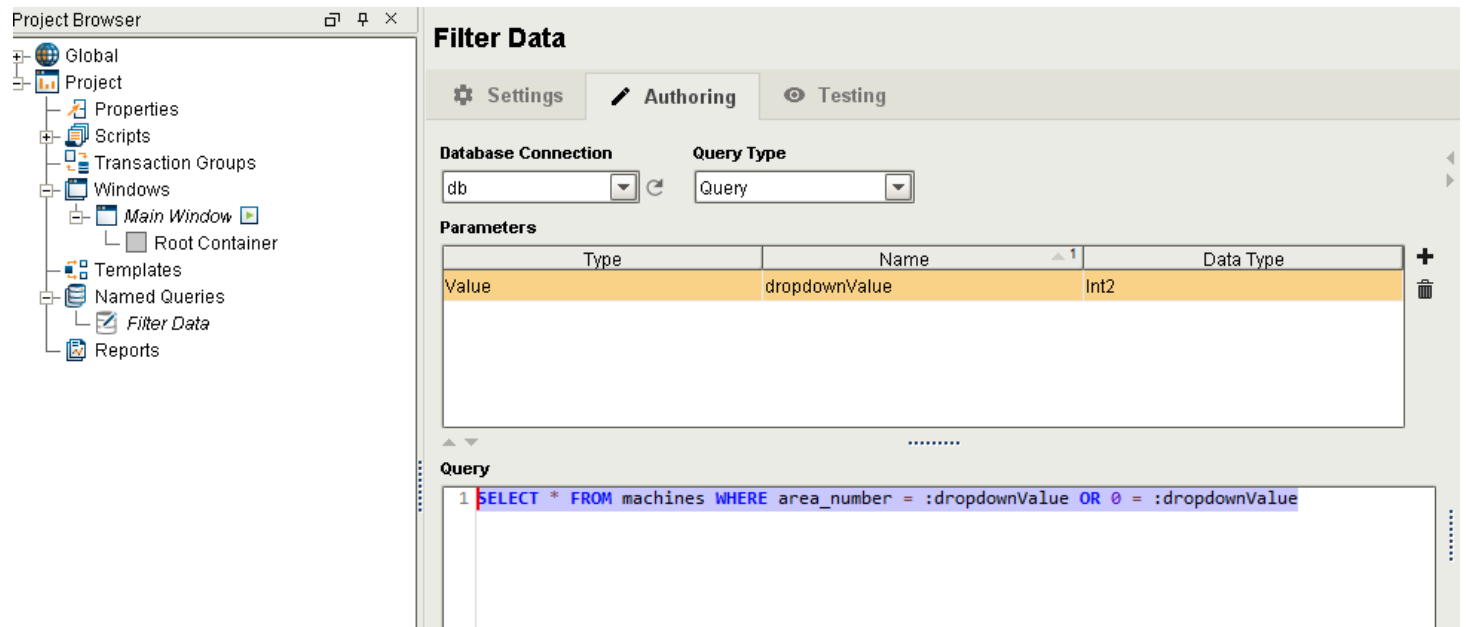
id	machine_name	area_number
1	Conveyor	1
2	Press	2
3	Tank	1
4	Packer	3
5	Loader	3
6	Oven 1	3
7	Oven 2	2
8	Wrapper	1
9	Mixer	3
10	Cold Storage	2
11	Dryer	2

We can then put together the query and components necessary to get this working:

1. Create a new Named Query. Set up security to fit your needs, and name it appropriately. For more information on creating Named Queries, see [Using Named Queries - Example](#).
 - i. Create a single Value type Parameter that is an Int2 data type. I called mine dropdownValue
 - ii. Add in the combined query that we went over above, but use the machine table name and column names.

SQL - Selecting Values from a Table and Filtering on a Dropdown

```
SELECT * FROM machines WHERE area_number = :dropdownValue OR 0 = :dropdownValue
```



1. Create a new Main Window and add a Power Table component and a Dropdown component to the window.
 - i. On the Dropdown's **Data** property, create a dataset that looks like this: |Value|Label| |---|:---|
|1|Area 1| |2|Area 2| |3|Area 3| |0|All Areas|
2. On the Power Table component, create a Named Query Binding on the **Data** property.
 - i. Select the Named Query that was created in step 1.
 - ii. For the parameter, bind its value to the **Selected Value** property of the Dropdown component, and then click the OK button.

Property Binding: Root Container.Power Table

Binding Types

Tag

☐ Tag

☐ Indirect Tag

☐ Tag History

Property

☐ Expression

☐ Property

Database

☒ Named Query

☐ DB Browse

☐ SQL Query

Other

☐ Cell Update

☐ Functions

☒ No Binding

SELECT Query

Path

Filter Data ☐

Parameters

Type	Name	Data Type	Value
Value	dropdownValue	Int2	{Root Container.Dropdown.selectedValue}

Query

```
SELECT * FROM machines WHERE area_number = :dropdownValue OR 0 = :dropdownValue
```

Polling Mode

☒ Off ☐ Relative ☐ Absolute

Polling Rate

Rate = (Base Rate) +/- sec

Retain Rows

☐ false

3. Put the Designer into preview mode and try out the Dropdown Filter. You will see that selecting a different value in the dropdown filters the data coming back so that only certain rows are shown.

id	machine_name	area_number
1	Conveyor	1
2	Press	2
3	Tank	1
4	Packer	3
5	Loader	3
6	Oven 1	3
7	Oven 2	2
8	Wrapper	1
9	Mixer	3
10	Cold Storage	2
11	Dryer	2

All Areas

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Inserting Data into a Database



INDUCTIVE
UNIVERSITY

Add Data to Database

[Watch the video](#)

Inserting Data on a Button Press

A common way to insert information into a database is to execute a SQL query after the user presses a Button. The button can run a script to collect the information it needs and then execute a SQL INSERT statement to push that data into the correct Database table. A script executed on the Button's **actionPerformed** event handler would collect the relevant properties and insert them into a database. The script would look like this:

Pseudocode - Collects Data and Insert into the Database

```
value1 = {component property reference}
value2 = {tag value}
value3 = {static value}

query = "INSERT INTO table (col1, col2, col3) VALUES (?, ?, ?)"
args = [value1, value2, value3]
system.db.runPrepUpdate(query, args)
```

Notice that the script isn't limited from taking values from any one place, grabbing tag values and property values and inserting them into a table. It is important to note that using the runPrepUpdate function will require that certain **Client Permissions** have been enabled.

Example - Inserting Values

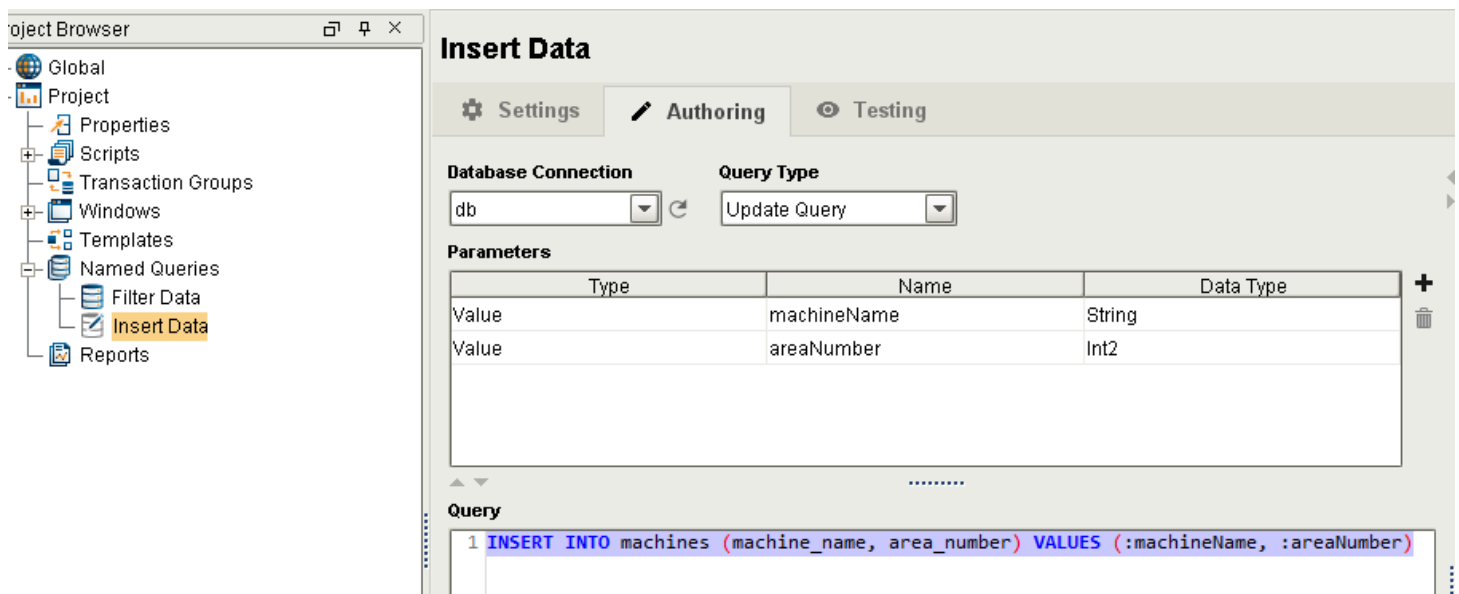
Say we have a table called machines in a database with three columns: an id column, a machine_name column, and an area_number column. We can build a query and a script that will

insert the data into the database:

1. Create a new Named Query. Set up security to fit your needs and name it appropriately. For more information on creating Named Queries, see [Using Named Queries - Example](#).
 - i. Set the Query Type to Update Query.
 - ii. Create two Value type Parameters.
 - a. The first will be a string data type and will be for the name of the machine so it can be called machineName.
 - b. The second will be an Int2 data type and will be for the area number so it can be called areaNumber.
 - iii. Create the insert query.

SQL - Inserting Values from Components on the Window

```
INSERT INTO machines (machine_name, area_number) VALUES (:machineName, :areaNumber)
```



2. On a Main Window, add a Button component, a Text Field component, and a Dropdown Component.
 - i. On the Dropdown's Data property, create a dataset that looks like this:

Value	Label
1	Area 1
2	Area 2
3	Area 3

- ii. On the Button's Text property, change the value to something like "Submit"



The image shows a light beige rectangular area containing three UI elements. At the top is a dropdown menu with a white background and a small downward arrow on the right; it displays the text 'Area 1'. Below the dropdown is a white rectangular text input field. At the bottom is a button with a light beige background and a thin black border, containing the text 'Submit' in a dark font.

3. Right Click on the Button and select Scripting. Navigate to the Script Editor tab on the actionPerformed Event Handler.

- i. Copy in this script, which will pull in the value from the Dropdown and the Text Field and insert them into the table using the Named Query we built in step 1.

Python - Insert Values into a Database Table

```
# Grab the area number and machine name from the components we added to the window.
areaNum = event.source.parent.getComponent('Dropdown').selectedValue
machineName = event.source.parent.getComponent('Text Field').text

# A call to our Named Query, inserting the two parameters using dictionary syntax.
system.db.runNamedQuery("Insert Data", {"machineName":machineName,
"areaNumber":areaNum})
```

4. Test it out by selecting an area, entering in a machine name, and clicking the submit button. You can check out the new rows using the [Database Query Browser](#) to select the values from that table.

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Updating the Database through the Power Table



INDUCTIVE
UNIVERSITY

Edit Data in Database

[Watch the video](#)

Updating Table Data

Table components can do more than show data from a database. A properly configured Table can make the data of the Table accessible to the Client and allow the user to edit the data in realtime. This page shows two examples using the **Table** and **Power Table** components editing data in a database in realtime.

Suppose your database had a Table like this:

Id	UserName	FirstName	LastName	Notes
1	JS	John	Smith	Likes bikes
2	LJ	Luke	Johnson	Lives in town
3	PB	Peter	Burke	Enjoys cooking

Using a Power Table Component

Let's create a script on a **Power Table** component and the **onCellEdited** extension function that will write to the cell in the database.

DANGER

This function will not work using default [Client Permission](#) settings. The Legacy Database Access will need to be enabled for this to work.

1. Drag a Power Table component to your window, and bind the **Data** property of the Power Table using a SQL Query Binding Type and the query below.

SQL - Selects All Records

```
SELECT * FROM users
```

1. Right-click on the **Power Table** and select **Customizers > Table Customizer**.
2. Select the **Editable** checkbox for each of the columns that we want to edit (i.e., same as the above example). Note, the **id** column should not be editable. Click the **OK** button.
3. Right-click the **Power Table** and select **Scripting**.
4. Under the Extension Functions folder, select the **onCellEdited** function, and check the **Enable** box. We can put a script in here that will grab the id column value of the row that we modify, and then use that and the new value that we entered to update the Database table. The extension function makes this easy, because it provides variables for the column name of the row we are editing, the row number that was edited, as well as the new value. Paste the following script in the scripting area.

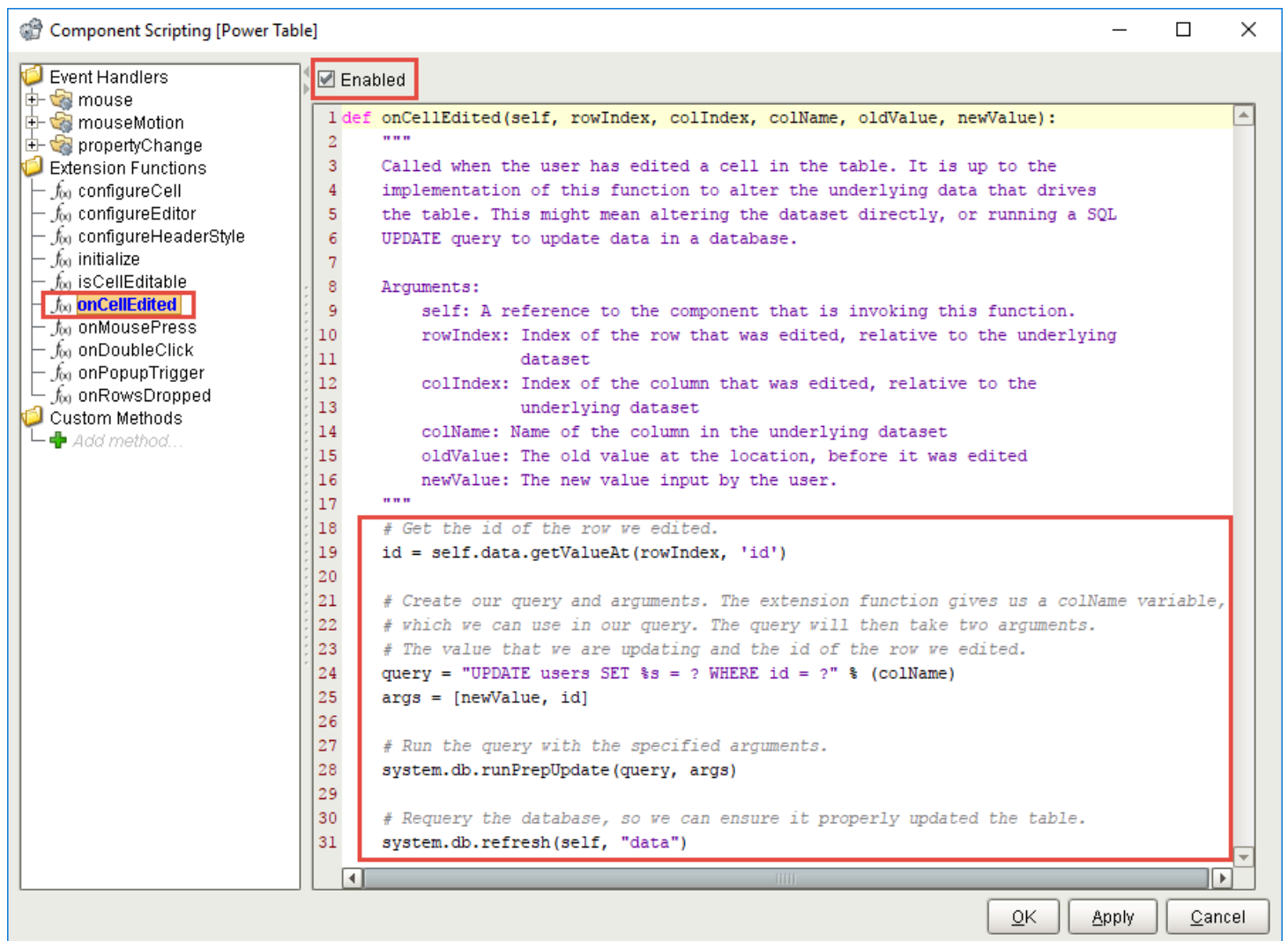
Python - Writes to a Cell in the Database

```
# Get the id of the row we edited.
id = self.data.getValueAt(rowIndex, 'id')

# Create our query and arguments. The extension function gives us a colName
variable,
# which we can use in our query. The query will then take two arguments.
# The value that we are updating and the id of the row we edited.
query = "UPDATE users SET %s = ? WHERE id = ?" % (colName)
args = [newValue, id]

# Run the query with the specified arguments.
system.db.runPrepUpdate(query, args)

# Requery the database, so we can ensure it properly updated the table.
system.db.refresh(self, "data")
```



6. Now, let's test it out. Put the Designer in **Preview Mode**, select the cell you want to edit, hit enter to commit the change, or tab to the next cell to make additional edits.

id	UserName	FirstName	LastName	Notes
1	JS	John	Smith	Likes bikes
2	LJ	Luke	Johnson	Lives in town
3	CP	Casey	Peters	Loves Ignition

Using a Table Component

Let's create another script like the example above, but this time we'll use a **Table** component's **cellEdited** event that will write to the cell in the database when we update it on the component. The script will be slightly different.

DANGER

This function will not work using default [Client Permission](#) settings. The Legacy Database Access will need to be enabled for this to work.

1. Drag a Table component to your window. With the Table component selected, bind the **Data** property of the Table using a SQL Query Binding Type and the query below.

SQL - Select All Records

```
SELECT * FROM users
```

id	UserName	FirstName	LastName	Notes
1	JS	John	Smith	Likes bikes
2	LJ	Luke	Johnson	Lives in town
3	PB	Peter	Burke	Enjoys cooking

2. Right-click on the **Table** and select **Customizers > Table Customizer**.
3. Select the **Editable** checkbox for each of the columns that we will want to edit. Note, the **id** column should not be editable. Click the **OK** button.

Table Customizer

Column Configuration Background Color Mapping

	id	UserName	FirstName	LastName	Notes
Header					
Hide?	☐	☐	☐	☐	☐
Editable	☐	☒	☒	☒	☒
Sortable	☒	☒	☒	☒	☒
Horiz Align	Auto	Auto	Auto	Auto	Auto
Vert Align	Center	Center	Center	Center	Center
Hdr Horiz Align	Center	Center	Center	Center	Center
Prefix					
Suffix					
Number Format	###0.## %	###0.## %	###0.## %	###0.## %	###0.## %
Date Format	MMM d, yyyy h:mm a	MMM d, yyyy h:mm a	MMM d, yyyy h:mm a	MMM d, yyyy h:mm a	MMM d, yyyy h:mm a
Boolean?	☐	☐	☐	☐	☐
Progress Bar?	☐	☐	☐	☐	☐
Progress Bar Range	Min: 0 Max: 100	Min: 0 Max: 100	Min: 0 Max: 100	Min: 0 Max: 100	Min: 0 Max: 100
Hide Text Over P-Bar?	☐	☐	☐	☐	☐
P-Bar Color					
P-Bar Background					
Translation List Column					
Translation List	(none)	(none)	(none)	(none)	(none)

OK Cancel

4. Right-click the **Table** and select **Scripting**.

5. Create a script in the **cell > cellEdited** event handler using the script provided below. In this script, we have variables that contain the row number that was edited, the column number that was edited, and the new value. Because we do not have the name of the column that was edited, we must first grab the list of columns using the **getColumnHeaders()** system function. We can then use the column number that was edited to find the name of the column that changed and use it in the query.

Python - Writes to a Cell in the Database When Component is Updated

```

# Get the id of the row we edited.
id = event.source.data.getValueAt(event.row, 'id')

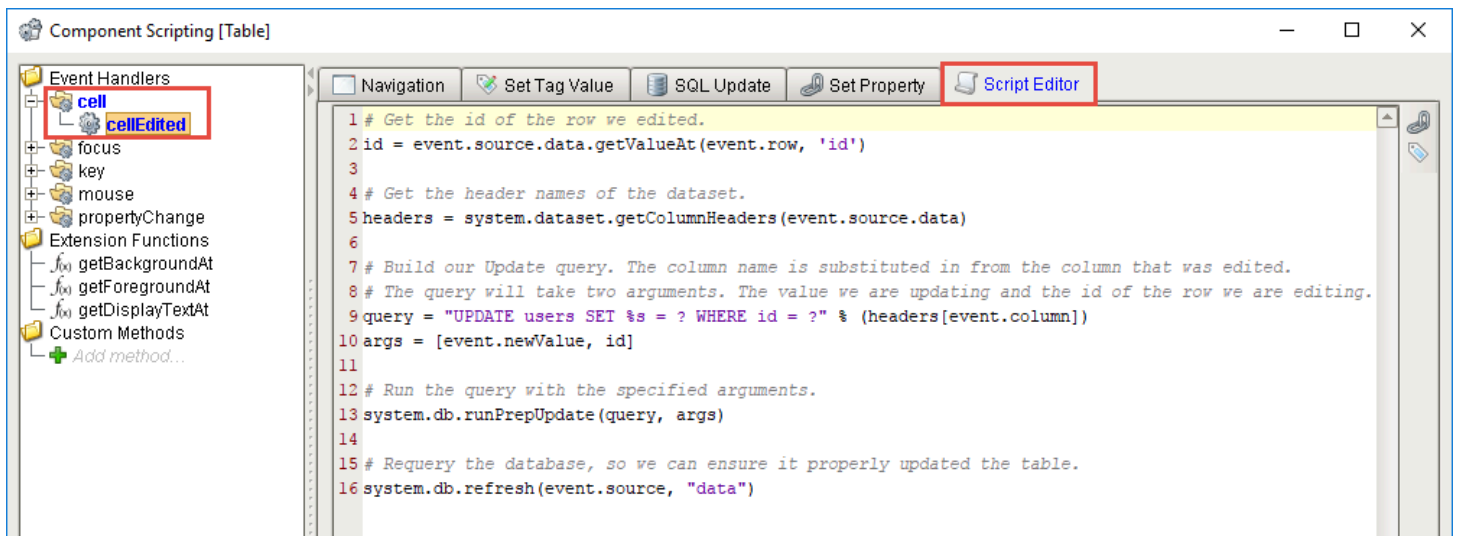
# Get the header names of the dataset.
headers = system.dataset.getColumnHeaders(event.source.data)

# Build our Update query. The column name is substituted in from the column that
was edited.
# The query will take two arguments. The value we are updating and the id of the
row we are editing.
query = "UPDATE users SET %s = ? WHERE id = ?" % (headers[event.column])
args = [event.newValue, id]

# Run the query with the specified arguments.
system.db.runPrepUpdate(query, args)

# Requery the database, so we can ensure it properly updated the table.
system.db.refresh(event.source, "data")

```



6. Now, you're ready to test it out! Put the Designer in **Preview Mode**, select the cell you want to edit, hit enter to commit the change, or tab to the next cell to make additional edits.

id	UserName	FirstName	LastName	Notes
1	JS	John	Smith	Likes bikes
2	LJ	Luke	Johnson	Lives in town
3	EF	Ellen	Frank	Loves Ignition

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Refreshing a SQL Query



INDUCTIVE
UNIVERSITY

Refreshing SQL Data on
Component

[Watch the video](#)

Refreshing a Query

The SQL query that populates a property on a component will refresh its data periodically if the **Polling Mode** is set to either Relative or Absolute. However, there are times when you want the data to query the database once when the window is opened, and then retain explicit control over each subsequent refresh. The Table data can be requered using `system.db.refresh`. There are two main ways of doing this: either placing the script on a button, or at the end of some code.

Refresh After Edits

If edits are being made to the Table data, or additional rows are being added, it can be a good idea to call the `system.db.refresh()` function after manipulating data. This way, the data in the Table will automatically refresh with the newly entered data, saving the user the hassle of clicking a refresh button.

Pseudocode - Refreshing a Table after an Edit

```
doWork()  
  
system.db.runPrepUpdate("INSERT INTO table")  
  
system.db.refresh(component, "propertyName")
```

Refresh on a Button

We can use a Button component with a script on it to refresh the component with the query binding on it. Typically the button is placed close by the query bound component, and will say something like "Refresh Data". The script is actually fairly simple, and can be placed on the **actionPerformed** event of the Button.

1. Drag a **Button** component onto a window that has a Power Table or Table component querying a Database table.
 - i. Change the Button's Text property to say "Refresh."
2. On the Power Table component, open the **data** property binding and set the polling mode to **off**.
3. Right click on the Button and select **Scripting**.
 - i. Select the **actionPerformed** Event Handler and navigate to the Script Editor tab.
 - ii. Add in this script, which will force the Table's data property to refresh.

Python - Refreshing the Database when the Button is Pressed

```
# Will force the Power Table's Data property to run the query again.  
system.db.refresh(event.source.parent.getComponent('Power Table'), "data")
```

4. Try it out by adding new data to the Database table and then clicking the Refresh Button.

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Editing Multi-Selected Rows from a Table



INDUCTIVE
UNIVERSITY

**Deleting Multi-Selected Rows
from Table**

[Watch the video](#)

Editing Multiple Table Rows

A common user experience is to select multiple rows of a table and then edit all of those rows from the database at once, such as deleting all of them. When a user selects more than one row in a table, there is a special function called `getSelectedRows()` available on both the **Table** and **Power Table** components that returns the row indexes of the table as a Python list. This list can be iterated through in order to delete the selected rows from the database.

Example - Deleting Selected Rows

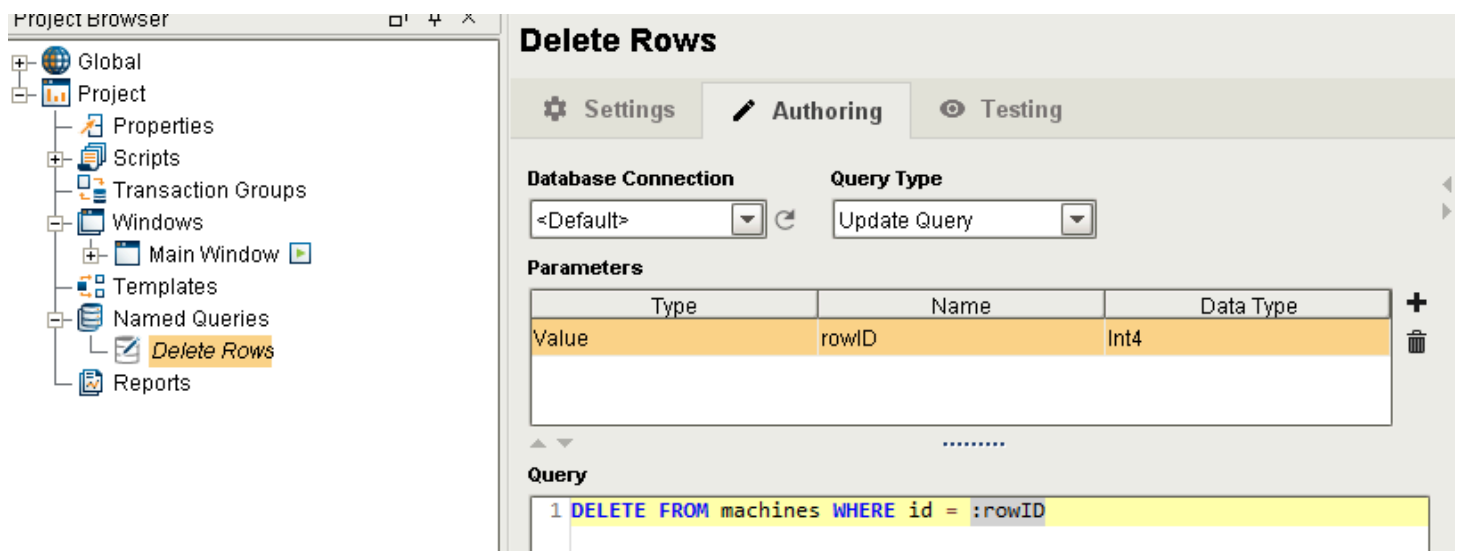
We can use the list of selected rows to delete them from the database. Start with a table in the Database table that looks like this:

id	machine_name	area_number
1	Conveyor	1
2	Press	2
3	Tank	1
4	Packer	3
5	Loader	3
6	Oven 1	3
7	Oven 2	2
8	Wrapper	1
9	Mixer	3
10	Cold Storage	2
11	Dryer	2

1. Create a new Named Query that will be used to delete rows of data. [Setup Security](#) to fit your needs, and name it appropriately. For more information on creating Named Queries, see [Using Named Queries - Example](#).
 - i. Set the Query Type to **Update Query**.
 - ii. Create a single Value type Parameter that is an Int4 data type. This will hold the id of the row, so we can name it rowID.
 - iii. Create the Delete Query.

SQL - Delete Row with Matching ID

```
DELETE FROM machines WHERE id = :rowID
```



2. Create a second Named Query as in step 1, but this one will be used to select the data into a table. 2. Set the Query Type to Query, no parameters, and add a basic select query.

SQL - Selecting from a Table

```
SELECT * FROM machine
```

3. On a new Main Window, add a Power Table component and a Button component.
4. On the Power Table's **Data** property, set up a Named Query binding to the Select Query that was made in step 2.
 - i. Ensure that the Selection Mode property is set to **Multiple Interval**.
5. On the Button, change the Text property to say "Delete".
 - i. Right click on the Button and select Scripting.
 - ii. Select the actionPerformed Event Handler and navigate to the Script Editor Tab.
 - iii. Here we need to call the **getSelectedRows()** function on the Power Table to determine what rows are selected, and then loop through those to grab the value of the id column in each row and delete the row based on that id.

Python - Looping Through the Selected Rows and Deleting Them

```
# Get the data from the table and assign it to the variable called data.
data = event.source.parent.getComponent('Power Table').data

# Get the rows of the data that the user has currently highlighted.
rows = event.source.parent.getComponent('Power Table').getSelectedRows()

# Iterate through each row of the list that is associated with the rows variable.
for row in rows:

    # Get the value associated with the current row and the the column called
"id".
    id = data.getValueAt(row, "id")

    # Run the query to delete from the database, where tableName is the name of
the database table.
    system.db.runNamedQuery("Delete Machine Rows", {"rowID":id})

# Refresh the table data.
system.db.refresh(event.source.parent.getComponent('Power Table'), "data")
```

6. You can test it out by putting the Designer into Preview Mode, selecting a few rows, and then clicking the Delete Button.

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Storing Files in a Database



INDUCTIVE
UNIVERSITY

Storing Files in a Database

[Watch the video](#)

Storing and Displaying Files in a Database

Ignition can store different types of files into a database by storing the raw file bytes into a special database column. Ignition can also pull these file bytes out and display certain files within a Client.

! INFO

Each database has different column types that are used to store files, so it is important to check with your database documentation to see which data type the column would need to be set to for it to accept file bytes. For example, in MySQL, the datatype that accepts PDF bytes is a **LongBlob** datatype, so you will need to set the PDF Data column to the **LongBlob** datatype. MS SQL accepts the **Varbinary** datatype, so you'll need to set the PDF Data column to a **Varbinary** datatype.

Example - PDF File

One of the most common file types that is stored is PDF files. This allows you to store each PDF file within the central database where each client will have access to it, instead of placing the file in a shared drive that all Client computers have access to.

Uploading PDF Files to the Database

With a simple script on a Button component, we can store a PDF file into the database so that any user can view it later. This part does not use a Named Query, because the Named Query Parameters

do not have a data type that allows us to pass in the raw file bytes. We can instead use `system.db.runPrepUpdate` to call a query from the script.

This example requires that you have a table with a byte array column in it. For example: MySQL user the BLOB data type and MSSQL uses the varbinary() data type.

DANGER

This function will not work using default [Client Permission](#) settings. The Legacy Database Access will need to be enabled for this to work.

1. Add a Button component to a new Main Window.
2. Change the Text property to say Add File.
3. Right click on the Button and select Scripting.
 - i. Navigate to the Script Editor Tab of the actionPerformed Event Handler. Here we can put a script that will grab the file bytes using the file path and the `system.file.readFileAsBytes` function, and then insert that into the database, along with a user selected file name.

Python - Uploads PDF Files to a Database Using a Button

```
# Find the path to the PDF file.
path = system.file.openFile("pdf")

# Check to ensure that the user actually selected a filepath.
if path != None:

    # Read the file as bytes.
    data = system.file.readFileAsBytes(path)

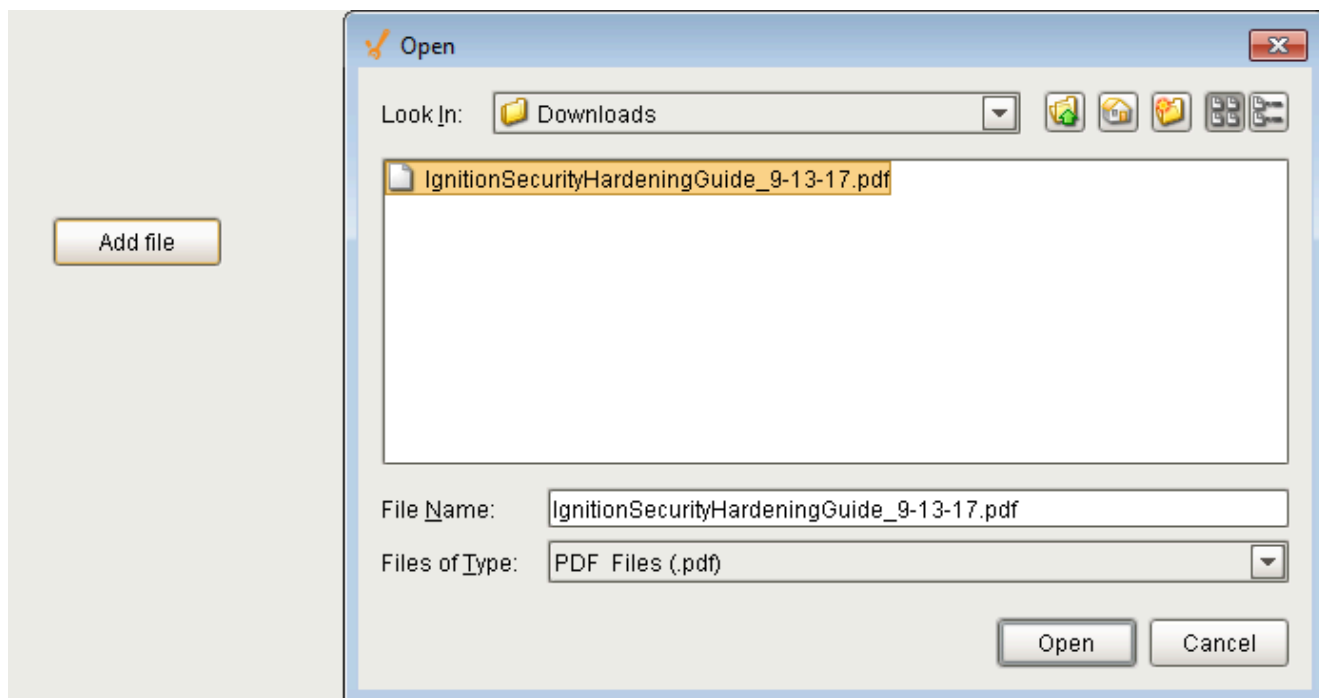
    # Ask the user to enter in a filename.
    # Will grab just the filename and extension from the path as a suggestion.
    name = system.gui.inputBox("Enter a name for the file", path.split('\\')[-1])

    # Check to ensure that the user entered a name for the file.
    if name != None:

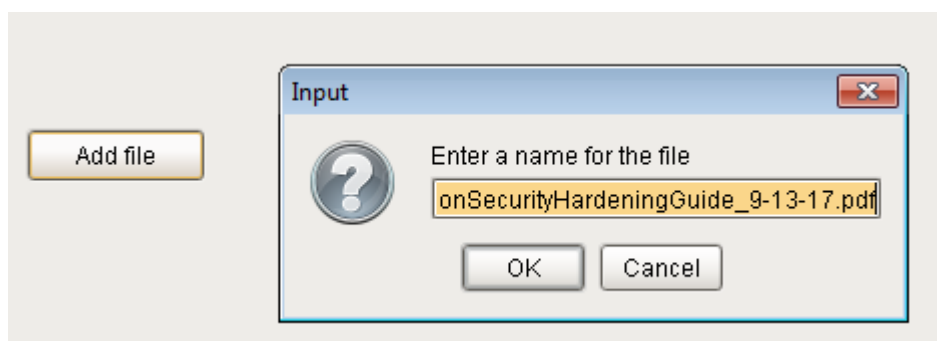
        # Insert the data and name into the database.
        system.db.runPrepUpdate("INSERT INTO files (fileName, fileBytes) VALUES
(?,?)", [name,data])
```

4. Test out the script by putting the Designer into Preview Mode and clicking the Button.

i. First, select the file to load.



ii. Then enter in a file name.



Displaying PDF Files from the Database

Ignition can render a PDF document inside the **PDF Viewer** component, which is a part of the **Reporting Module**. To view PDF files in the Client, your Ignition server must have the Reporting Module installed. Once the module is installed, you can load the bytes from the database into the PDF Viewer component.

1. Create a new Named Query that will be used to select the file names and ids for a Dropdown selection. Set up security to fit your needs, and name it appropriately. For more information on creating Named Queries, see [Using Named Queries - Example](#).
 - i. With no Parameters, add a query to select all the files in our files table.

SQL - Selecting all Files

```
SELECT id, fileName FROM files
```

2. Create a second Named Query, same as in step 1. This will be used to grab the file name and bytes after the user has chosen a file.
 - i. Add a single Value type Parameter, "fileID" that will be an Int4 data type.
 - ii. Add the query to select the file name and bytes based on the selected ID.

SQL - Selecting a File based on an ID

```
SELECT fileName, fileBytes FROM files WHERE id = :fileID
```

3. On the Main Window, add a Dropdown component and a PDF Viewer component.
4. On the Dropdown component, add a Named Query binding to the Data property.
 - i. Set the binding to the Named Query that was made in step 1.
 - ii. Place a small refresh Button next to the Dropdown that will refresh this query. See [Refreshing a SQL Query](#) for more information for how to refresh on a button.
5. Right click on the Dropdown component and select Scripting.
 - i. Navigate to the Script Editor Tab of the actionPerformed Event Handler.
 - ii. This script will take any new selected value and use it in the Named Query we made in step 2 to get the file name and bytes. We can then load the bytes into the PDF Viewer.

Python - Displays PDF Files from a Database Using the PDF Viewer

```
# Check to see if the property that changed was the Selected Value property.
if event.propertyName == "selectedValue":

    # Run the query to grab the file name and bytes using the new selected ID
    value.
    system.db.runNamedQuery("Read File", {"fileID":event.newValue})

    # Grab the file bytes and name from the same row.
    bytes = data.getValueAt(0, "fileBytes")
    name = data.getValueAt(0, "fileName")

    # Load the bytes into the PDF Viewer component.
    event.source.parent.getComponent('PDF Viewer').loadPDFBytes(bytes, name)
```

6. Place the Designer into Preview Mode, and try selecting one of the stored files.



This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Simple Database Editor

Building a Database Table Editor

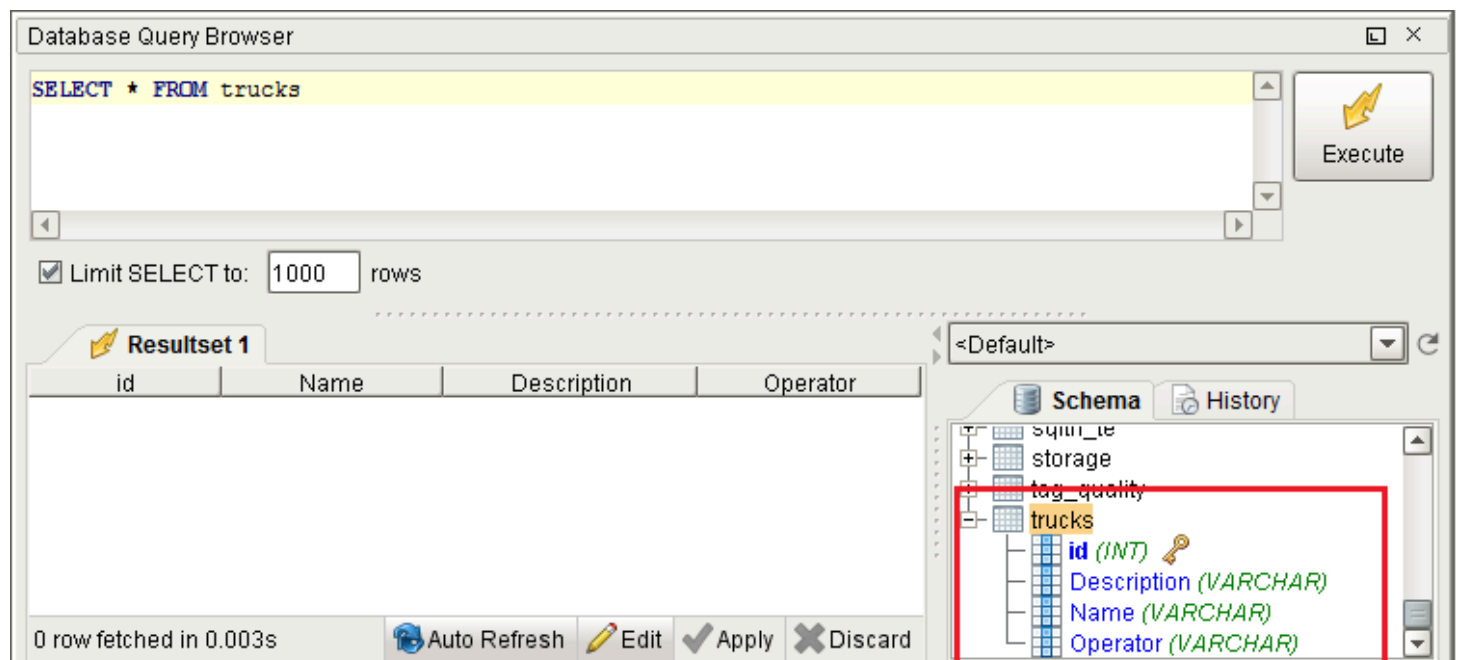
Building a Simple Database Table Editor in Ignition is actually quite easy. Using a few simple components and some scripting, we can easily make a window that will allow users to add, edit, and delete the data from a particular database table. Before we get started, there is a table that has already been created in a database that is used for this example. You can either create this table and follow along, or alter the example to fit your table.

The table is called **trucks**, and it has four columns.

- **id**
- **Name**
- **Description**
- **Operator**

The **id** column is our primary key, it is non null, and it auto increments. All other columns are varchar(45)

That is it. There is no need to add any data, as we can add data when testing our tool.



Adding the Components

This example consists of a Table component that will display the data, Button components to add and delete rows from the database table, Text Fields where we can enter in new values, and an Update Details Button that will push those values to the database table.

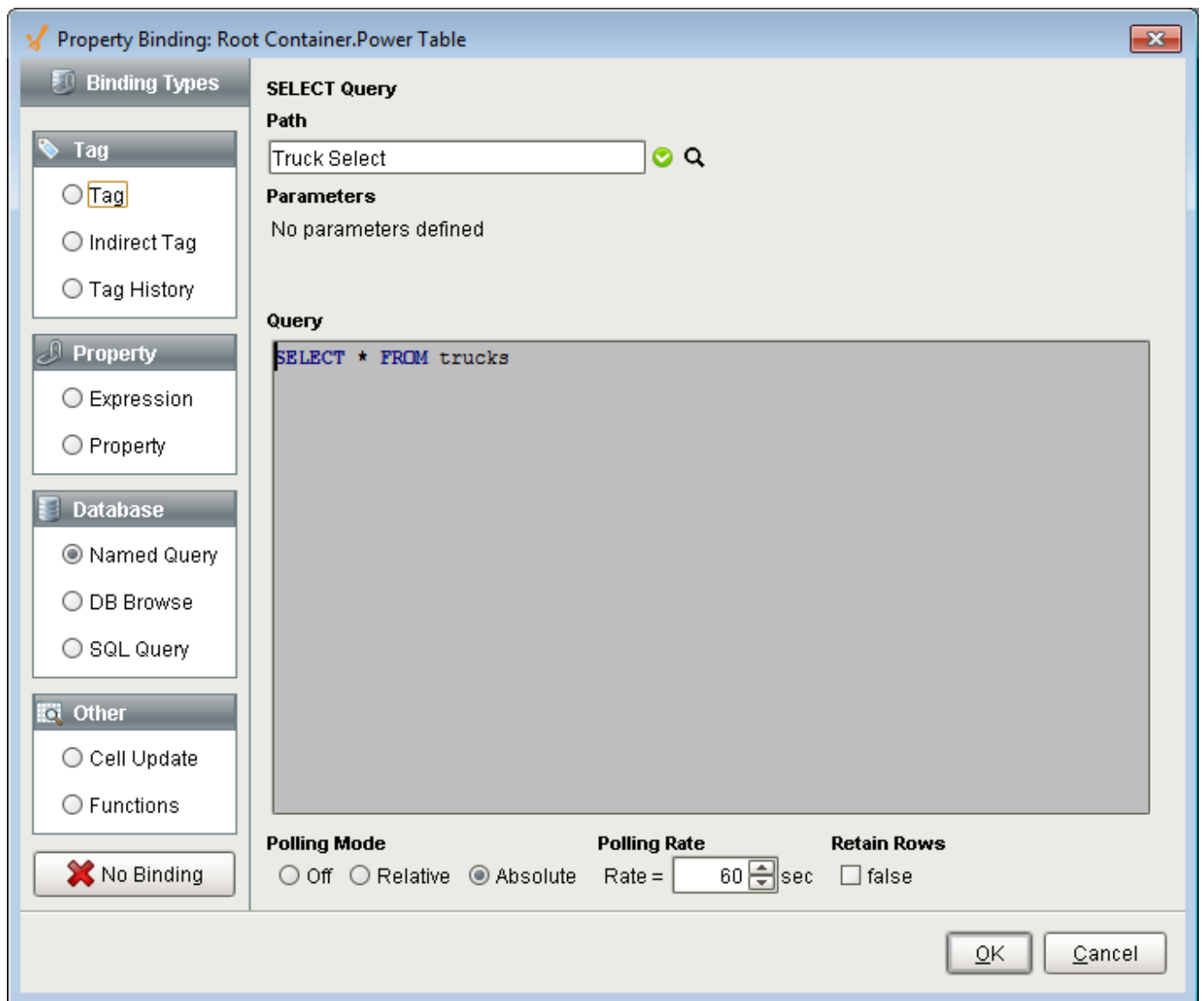
Let's start off by going to the **Designer** and adding the necessary components to the window.

1. Add a **Power Table** component to the window.
 - i. We want to query the table in the database that we made earlier, so let's add a **SQL Query Binding** on the **Data** property of the Table.
 - ii. Add the following code under the **SELECT Query** area.

SQL - Selects all Trucks

```
SELECT * FROM trucks
```

- iii. Set the **Polling Mode** to **Absolute** and the **Polling Rate** to **60 seconds**, so that it periodically updates if left open.
 - iv. Since we want to use the secure Named Query system, click the convert to Named Query button under the polling mode, and call the query "Truck Select"



- v. With the Table still selected, open the **Table Customizer** by right clicking on the Power Table and going to **Customizers > Table Customizer**.
- vi. Set the **Hide** property of the **id** column to **true**. We don't need to show that column to our users. Click **OK** to close the Table Customizer.
- vii. In the **Property Editor**, make sure the **Selection Mode** property of the Power Table is set to **Multiple Interval**.

2. Add three **Button** components to the right of the **Table**.

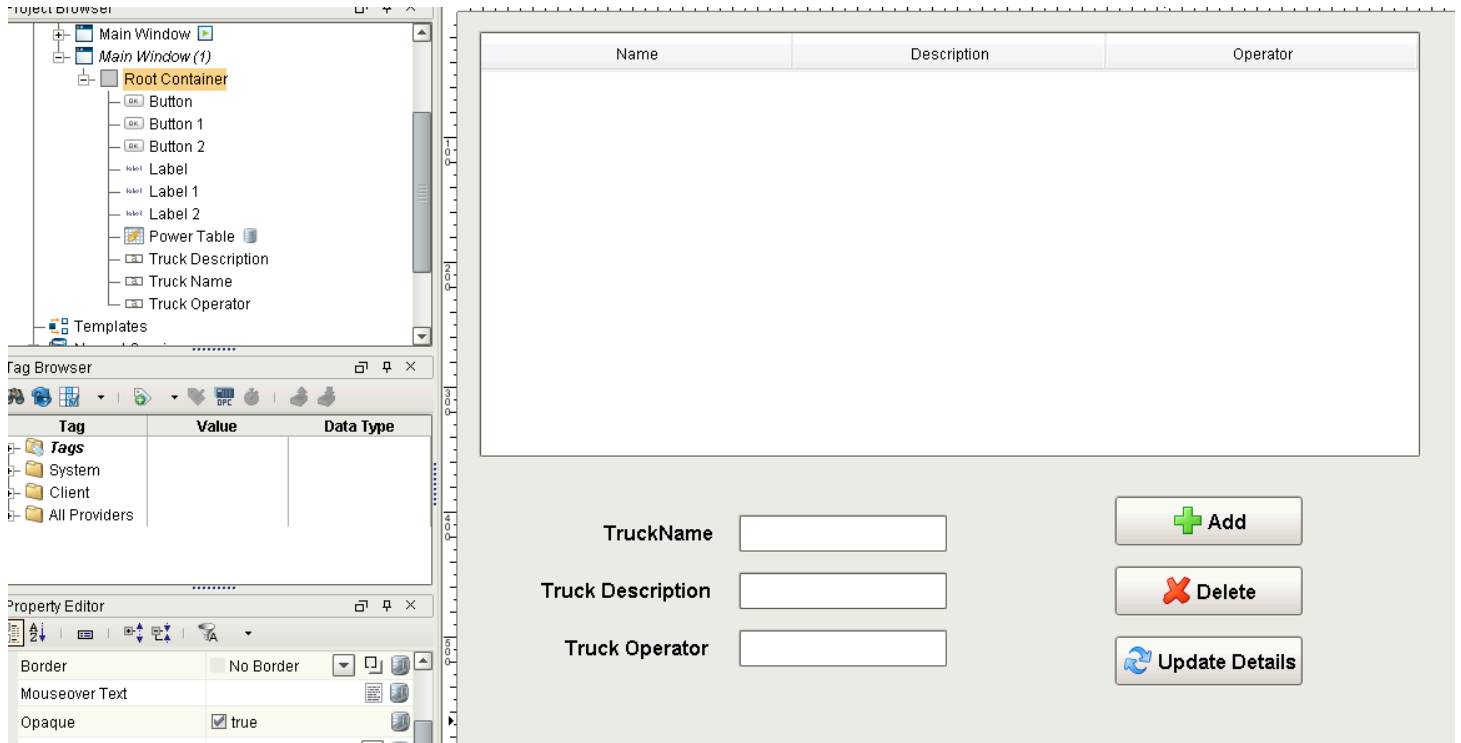
- i. Change the **Text** property on one Button to say "**Add.**"
- ii. Change the **Text** property of the second Button to say "**Delete**".
- iii. Change the **Text** property of the final Button to say "**Update Details**"
- iv. Optionally, you can also add an image to the **Image Path** of each button such as a green plus and a red X, and resize them as necessary.

3. Add three **Label** components and three **Text Field** components under the **Table**.

- i. Pair up a Label with each Text Field.

- ii. Rename the Text Fields: **Truck Name, Truck Description, Truck Operator**.
- iii. Change the **Text** property of each Label to match the name of their corresponding Text Fields.

Excellent! Now we have our window configured with all of the components we need. Now we can start adding scripts that will alter the components.



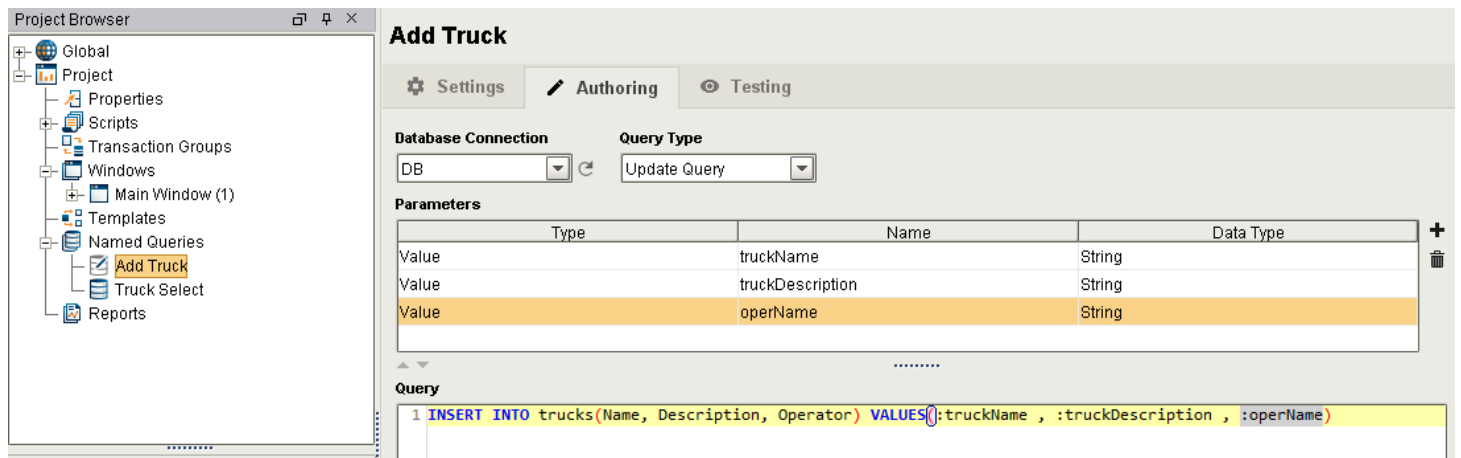
Add a Truck

First, we can add a script to our **Add** button that will allow us to add a row to the Table. We can use the Text Fields that we added so the user can enter in values to insert into the table.

1. Create a new Named Query which we will use to insert new data. Set up **security** to fit your needs and name it appropriately. For more information on creating Named Queries, see [Using Named Queries - Example](#).
 - i. Set the Query Type to Update Query.
 - ii. Create three Value type Parameters.
 - a. The first will be a string data type and will be used for the name of the truck so it can be called "truckName".
 - b. The second will be a string data type and will be used for the description of the truck so it can be called "truckDescription".
 - c. The third will also be a string data type and will be used for the operators name so it can be called "operName".
 - iii. Create the insert query.

SQL - Adding a Truck

```
INSERT INTO trucks(Name, Description, Operator) VALUES(:truckName ,
:truckDescription , :operName)
```



2. Back on our window with components, right click on the **Add Button** and select Scripting. Select the **actionPerformed** Event Handler and navigate to the **Script Editor** tab.

i. Here we can add some code that will pull the values from the three Text Field components and then use those values in the Named Query we made in step 1.

Python - Adds a New Blank Row and Refreshes the Table

```
# Grab the values from the Text Fields.
name = event.source.parent.getComponent('Truck Name').text
description = event.source.parent.getComponent('Truck Description').text
operator = event.source.parent.getComponent('Truck Operator').text

# Use those values in the Add Truck Named Query.
system.db.runNamedQuery("Add Truck", {"truckName":name,
"truckDescription":description, "operName":operator})

# Refresh the table to immediately bring in the new row.
system.db.refresh(event.source.parent.getComponent("Power Table"), "data")
```

3. Now we can test out our script and query by putting the Designer into **Preview Mode**, adding some data to the **Text Fields**, and clicking the **Add Button**. You should see a row populate the table. Try adding a few rows to fill our table with data. Make sure you have the designer Communication Mode set to Read/Write to test. Go back to **Design Mode** before moving on.

Name	Description	Operator
Green Monster	It is so green!	Sarah
Little Truck	The smallest truck.	Joe
Big Truck	This truck is big!	James
Fire Truck	Puts out fires. Does not transport fire.	Dale
Ice Cream Truck	Sells ice cream and various other treats.	Amy
Toy Truck	Transports toys, not a truck that is a toy.	Jane
Blue Truck	A truck that is painted blue.	Anne

TruckName

 Add

Truck Description

 Delete

Truck Operator

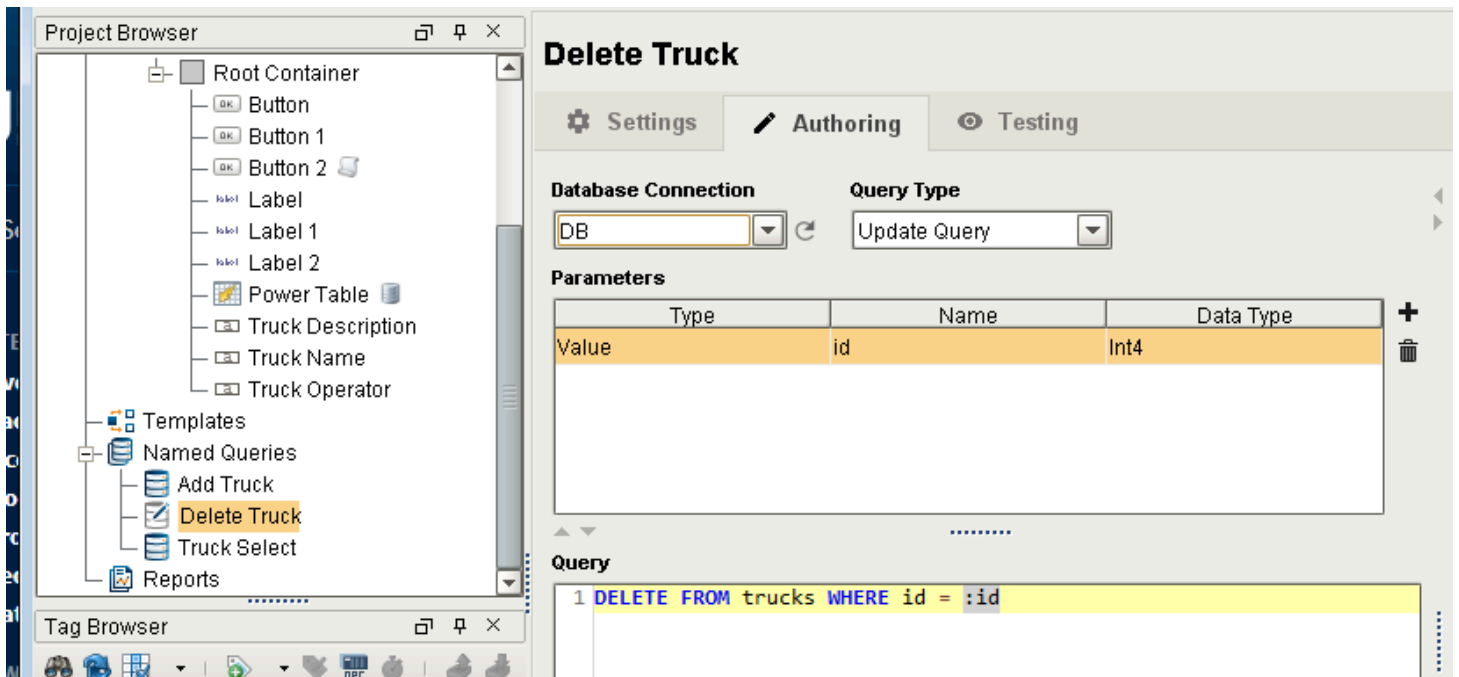
 Update Details

Delete a Truck

Now that we can add rows, let's add a script to our **Delete button** that will delete rows. Since our users can select multiple rows, our script needs to take that into account, and delete all of the rows that the user selected. It will also have a **Message Box** popup if there are no rows selected, informing the user that they need to select at least one row. We also want to add a confirmation before the rows actually get deleted.

1. Create a new Named Query which we will use to insert new data. Set up [security](#) to fit your needs and name it appropriately. For more information on creating Named Queries, see [Using Named Queries - Example](#).
 - i. Set the Query Type to Update Query.
 - ii. Create one Value type Parameters.
 - a. It will be a Int4 data type and will be used for the id of row so it can be called "id".
 - iii. Create the delete query.

```
DELETE FROM trucks WHERE id = :id
```



2. Back on our window with components, right click on the **Delete Button** and select Scripting. Navigate to the **Script Editor** tab on the **actionPerformed** Event Handler. Here we can add some code that will delete the user's selected rows. First we grab the Power Table's selected rows, and check to make sure rows are actually selected. If there are selected rows, then we have a popup that confirms the user wants to delete rows, and it shows the number of rows that will be deleted. Once the user confirms, the script will then loop through the selected rows and delete each one.

Python - Deletes the Row or Rows

```

# Get the selected row or rows.
selRows = event.source.parent.getComponent('Power Table').getSelectedRows()

# Check to see that a row is actually selected.
if len(selRows) > 0:

    # If a row is selected, ask for confirmation before deleting the rows.
    if system.gui.confirm("Are you sure you want to delete " + str(len(selRows)) +
" row(s)?", "Are You Sure?", 0):

        # If the user gave confirmation, we need to loop through all of them.
        for row in selRows:

            # For each selected row, we need to get the value of the id column in
that row.
            id = event.source.parent.getComponent('Power
Table').data.getValueAt(row, "id")

            # Use the id of the row to delete it from the database table.
            system.db.runNamedQuery("Delete Truck", {"id":id})

            # Refresh the table after deleting all selected rows
            # to immediately remove the selected rows from the Power Table.
            system.db.refresh(event.source.parent.getComponent('Power Table'), "data")

# If the user said no to the delete.
else:
    system.gui.messageBox("User canceled the delete.", "Delete Canceled")

# If no row is selected, have a popup message that asks the user to select a row.
else:
    system.gui.messageBox("Please select at least one row.", "Select A Row")

```

- Now we can test our **Delete button** by putting the Designer into **Preview Mode** and clicking the **Delete button**. When one or more rows are selected and the **Delete button** is pressed, a popup will confirm we want to delete the rows before deleting them. (You can select multiple rows with **Shift-Click** or **Control-Click**). When no rows are selected and the **Delete button** is pressed, a **Message Box** will pop up, informing the user to select a row first. You can deselect rows in the Table by clicking on the Table and pressing the **Escape** key. Be sure to go back to Design Mode before moving on.

Confirming Row Selection to Delete

Row Deletion Results

Name	Description	Operator
Green Monster	It is so green!	Sarah
Little Truck	The smallest truck.	Joe
Big Truck	This truck is big!	James
Fire Truck	Puts out fires. Does not transport fire.	Dale
Ice Cream Truck	Sells ice cream and various other treats.	Amy
Toy Truck	Transports toys, not a truck that is a toy.	Jane
Blue Truck	A truck that is painted blue.	Anne

Are You Sure?

Are you sure you want to delete 2 row(s)?

Yes No

TruckName

Truck Description

Truck Operator

 Add

 Delete

 Update Details

Name	Description	Operator
Green Monster	It is so green!	Sarah
Little Truck	The smallest truck.	Joe
Fire Truck	Puts out fires. Does not transport fire.	Dale
Toy Truck	Transports toys, not a truck that is a toy.	Jane
Blue Truck	A truck that is painted blue.	Anne

TruckName

Truck Description

Truck Operator

 Add

 Delete

 Update Details

Populate Text Fields

Now that we can add and delete rows from our Table, we need a way to edit the data contained in them. However, before we can create a script on our **Update Details button**, we first need to populate the Text Fields with the current values of the selected row. We can do this by adding a simple expression to each of the text fields.

1. On the Truck Name Text Field, add an expression binding to the Text property.
 - i. The expression will grab the dataset and will grab the value at the selected row of the Name column. The expression is wrapped in a try so that if no rows are selected, it will return an empty string.

Expression - Grabbing the Selected Row's Name Value

```
try({Root Container.Power Table.data}[{Root Container.Power Table.selectedRow}, "Name"], "")
```

2. On the Truck Description Text Field, add an expression binding to the Text property.

- i. The expression will grab the dataset and will grab the value at the selected row of the Description column. The expression is wrapped in a try so that if no rows are selected, it will return an empty string.

Expression - Grabbing the Selected Row's Description Value

```
try({Root Container.Power Table.data}[{Root Container.Power  
Table.selectedRow}, "Description"], "")
```

3. On the Truck Operator Text Field, add an expression binding to the Text property.

i. The expression will grab the dataset and will grab the value at the selected row of the Operator column. The expression is wrapped in a try so that if no rows are selected, it will return an empty string.

Expression - Grabbing the Selected Row's Operator Value

```
try({Root Container.Power Table.data}[{Root Container.Power  
Table.selectedRow}, "Operator"], "")
```

4. Click on a row to see the fields below populate. Press the ESC key to empty the selection.

Name	Description	Operator
Green Monster	It is so green!	Sarah
Little Truck	The smallest truck.	Joe
Fire Truck	Puts out fires. Does not transport fire.	Dale
Toy Truck	Transports toys, not a truck that is a toy.	Jane
Blue Truck	A truck that is painted blue.	Anne

TruckName	Fire Truck	
Truck Description	Puts out fires. Does not tr	
Truck Operator	Dale	

Update Data

The last part of setting up this window is to add a script to the **Update Details button** that will pull the data from the Text Fields and insert it into the selected row of the database Table. Additionally, the script will check to ensure that a row selected, and that only one row is selected. We wouldn't want to

confuse users who may have accidentally selected multiple rows at a time, since our button will only update a single row.

1. Create a new Named Query which we will use to update a row of data. Set up **security** to fit your needs and name it appropriately. For more information on creating Named Queries, see [Using Named Queries - Example](#).

- i. Set the Query Type to Update Query.

- ii. Create four Value type Parameters.

- a. The first will be a string data type and will be used for the truck name of row so it can be called "truckName".
- b. The second will be a string data type and will be used for the truck description of row so it can be called "truckDescription".
- c. The third will be a string data type and will be used for the operator name of row so it can be called "operName".
- d. The fourth will be a Int4 data type and will be used for the id of row so it can be called "id".

- iii. Create the update query.

SQL - Deleting a Truck

```
UPDATE trucks SET Name = :truckName , Description = :truckDescription ,  
Operator = :operName WHERE id = :id
```

The screenshot shows the Power BI Desktop interface with the 'Update Truck' query configuration. The 'Project Browser' on the left shows a hierarchy of components: Button, Button 1, Button 2, Label, Label 1, Label 2, Power Table, Truck Description, Truck Name, Truck Operator, Templates, Named Queries, Add Truck, Delete Truck, Truck Select, Update Truck (highlighted), and Reports. The 'Update Truck' query configuration window is open, showing the 'Settings' tab. The 'Database Connection' is set to 'DB' and the 'Query Type' is 'Update Query'. The 'Parameters' table lists four parameters: truckName (String), truckDescription (String), operName (String), and id (Int4). The 'Query' text area contains the SQL statement: `1 UPDATE trucks SET Name = :truckName , Description = :truckDescription , Operator = :operName WHERE id = :id`.

Type	Name	Data Type
Value	truckName	String
Value	truckDescription	String
Value	operName	String
Value	id	Int4

2. Back on our window with components, right click on the **Update Button** and select Scripting. Navigate to the **Script Editor** tab on the **actionPerformed** Event Handler.

- i. Here we can add some code that will update the users selected row with data pulled from the Text Fields. First we check to ensure that only one row is selected. Next the script asks them to confirm that the user wants to update the selected row. If the user confirms, the script then pulls in the values of all three Text Fields and uses those values along with the selected rows

id to update the database table with new data. It will then refresh the table after updating so that the new data can be brought into the table.

Python - Deletes the Row or Rows

```
# Check to make sure only one row is selected.
if len(event.source.parent.getComponent('Power Table').getSelectedRows()) ==
1:

    # If a row is selected, ask for confirmation before updating the row.
    if system.gui.confirm("Are you sure you want to update the selected row?",
"Are You Sure?", 0):

        # Grab the values from the text fields.
        name = event.source.parent.getComponent('Truck Name').text
        description = event.source.parent.getComponent('Truck
Description').text
        operator = event.source.parent.getComponent('Truck Operator').text

        # Grab the selected row.
        selRow = event.source.parent.getComponent('Power Table').selectedRow

        # Using the selected row, we need to get the value of the id column in
that row.
        id = event.source.parent.getComponent('Power
Table').data.getValueAt(selRow, "id")

        # Run a query that will update the values of the row matching the id.
        system.db.runNamedQuery("Update Truck", {"truckName":name,
"truckDescription":description, "operName":operator, "id":id})

        # Refresh the table to immediately show the updated data.
        system.db.refresh(event.source.parent.getComponent('Power Table'),
"data")

    # If the user said no to the update.
    else:
        system.gui.messageBox("User canceled the update.", "Update Canceled")

# If there more than one row selected, or no rows selected.
else:

    # Have a popup asking the user to select one row.
    system.gui.messageBox("Please select one row to edit.", "Select One Row")
```

3. Now that we added this last bit of code, we can test it by selecting a row, adding data to each **Text Field**, and clicking the **Update Details button**. You should see the data populate the row of

the Table, since the table in the database has been updated. Additionally, you should notice how the Text Fields will automatically update when selecting a new row. Try adding a few rows of valid data to your Table.

Name	Description	Operator
Green Monster	It is so green!	Sarah
Little Truck	The smallest truck.	Joe
Fire Truck	Puts out fires. Does not transport fire.	Dale
Toy Truck	Transports toys, not a truck that is a toy.	Jane
Yellow Truck	A truck that is painted yellow.	Abby

TruckName

Yellow Truck

Truck Description

A truck that is painted yell

Truck Operator

Abby

 Add

 Delete

 Update Details

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

Basic SQL Troubleshooting

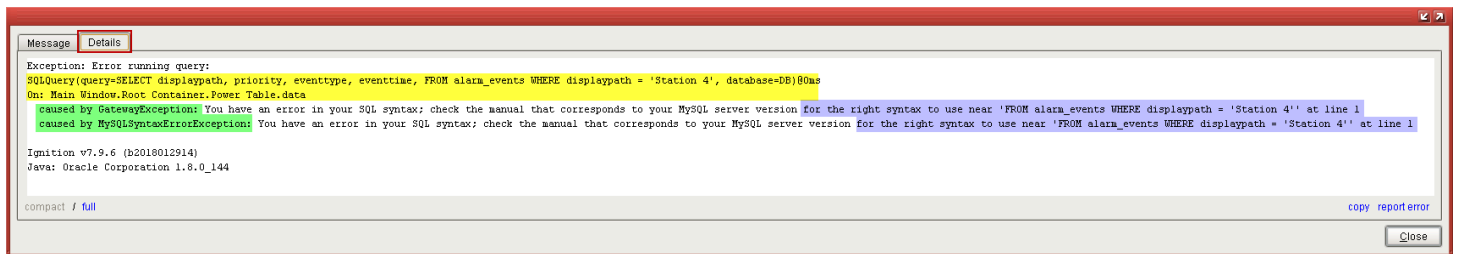
Learning when something is wrong with your database queries and learning how to resolve the problem is key to getting the best possible use out of the database connection to Ignition. Typically, the biggest problem that users face with queries is the data taking too long to load on the window, but an error in a query can also be difficult to track down. This section details what to look for when the query fails to execute, as well as what to do when facing slow queries and how to optimize them within the project to help return data to the window as quickly as possible.

Error Message Box

If a query fails to execute, chances are there was an error box that popped up. Resist the urge to close it! The Error Box usually contains a lot of useful info that can help troubleshoot why the query is failing. When looking at the error message, there are a few things to look for. First, you will want to click on the **Details** tab in the upper left corner of the Error Box. This will bring up the details of the error, which contains the information we need to track down the error.

In the image below, you can see the yellow highlighted part shows us what our query is, as well as where it is executing from. This helps to ensure we are looking at the correct error message for the query we are trying to fix. Next, you want to look for the words '**caused by**' in the text of the error details, which tell us the reason for the error. You can see in the image below, there are two of '**caused by**' messages highlighted in green. The first one is from the Gateway, and the second one is from the database (in this case, a MySQL database).

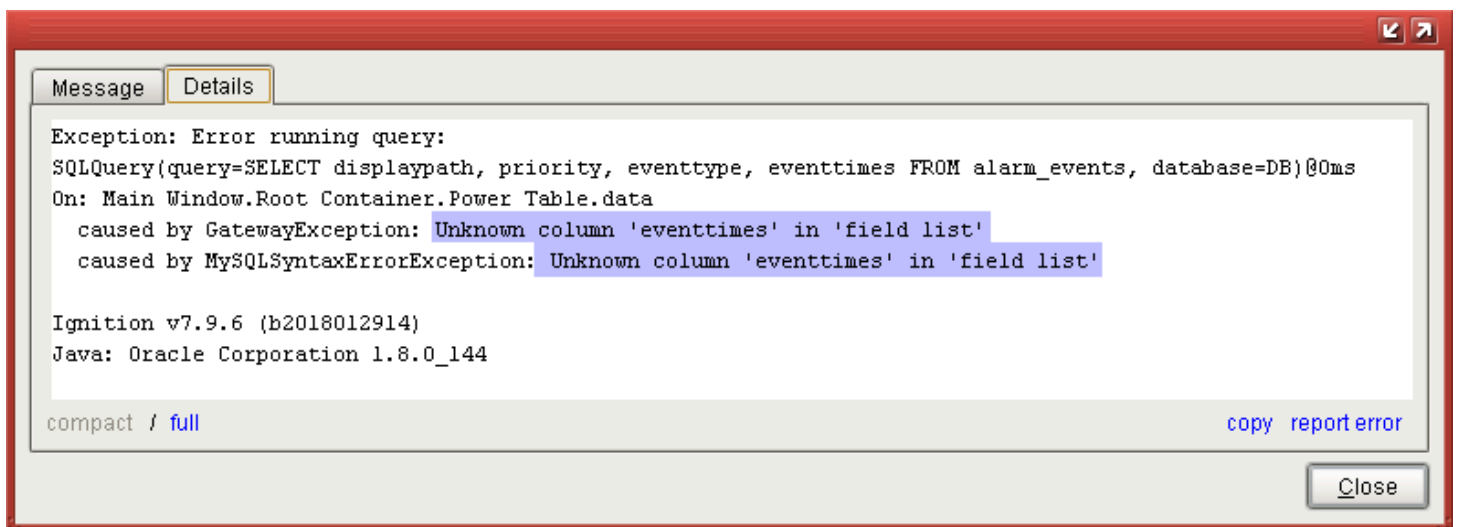
After locating the '**caused by**', the following message will help to pinpoint what the error is. In this example, we have an error with our syntax in our SQL statement. The most helpful portion of the error is at the end of the message highlighted in purple. This lets us know where the syntax error is in the SQL statement. The trick is to look immediately before the quoted query. In the image below, it starts to quote the original query with '**FROM alarm_events WHERE.....**', so in my original query, I need to look at what was right before the '**FROM**'. You can see in my original query highlighted in yellow, there is a comma right before the '**FROM**', which is incorrect, as the last column in the **SELECT** statement should not have a comma after it.



Other Common SQL Errors

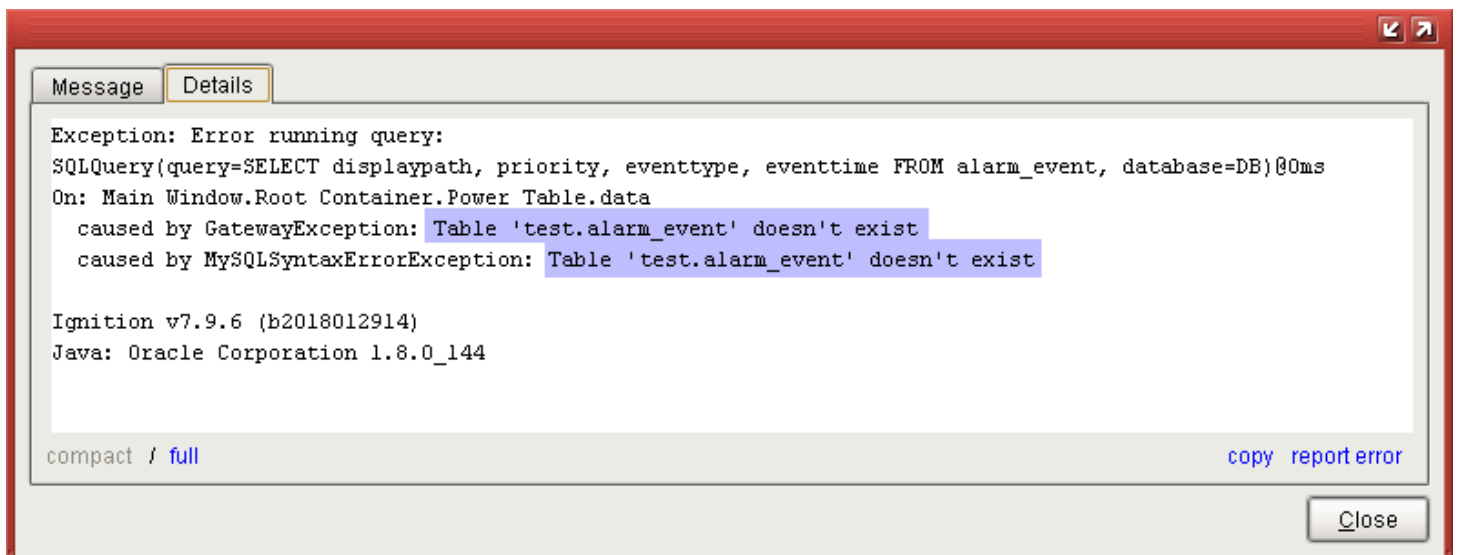
Unknown Column

Typically, the name of the column is wrong because of a misspelling. In this case, the correct spelling was 'eventtime', but there was an accidental 's' added to the name.



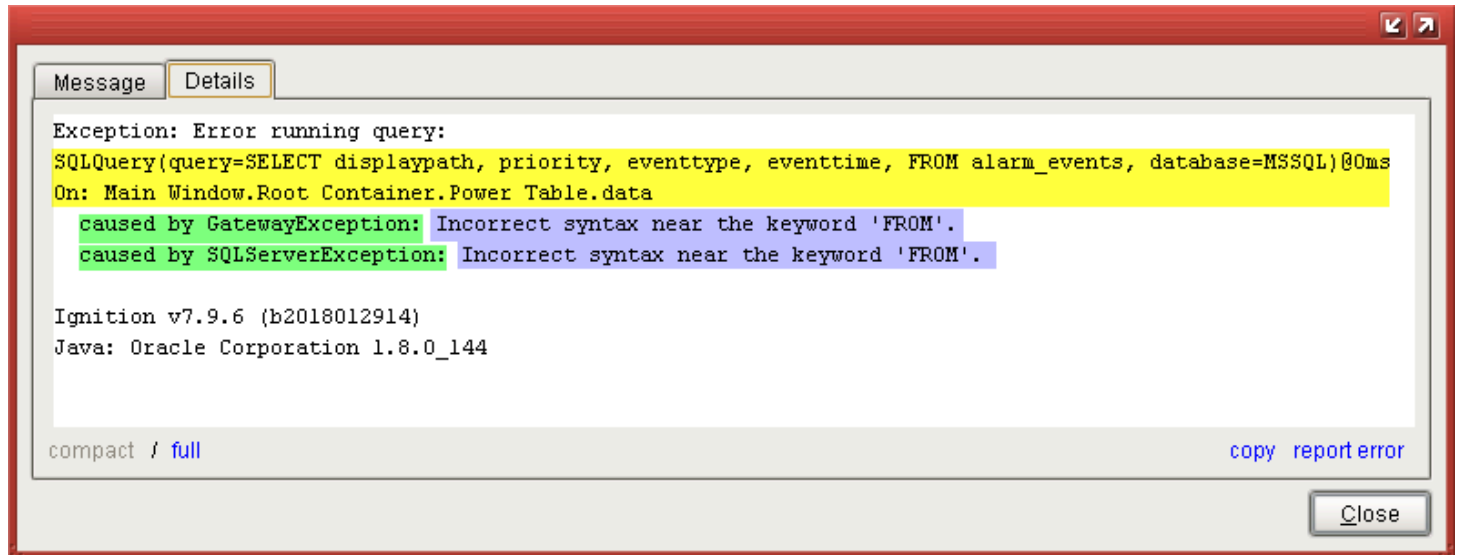
Unknown Table

If the table is not found in the database, it may also be because of a misspelling. Here, the correct table name should be 'alarm_events'.



Any Database Error

The previous error examples have all been from a MySQL database, but the same principles apply to any database. Simply locate the 'caused by', and look at the message afterwards. In this MSSQL syntax error, instead of displaying everything after the syntax error, it only displays the part immediately after. The problem in this query is an extra comma before the word 'FROM'.



Checking the Database Connection

Checking the Database Connection in the Gateway can also be useful to ensure there is a valid connection and there is nothing blocking the execution of queries. The Status page of the Gateway can be used to determine if queries are taking too long, while the Database Connection page can be used to alter the settings of the database to better handle the number of queries running from the Gateway to the database. See the [Slow Queries](#) page for more information on changing these settings.

Testing Query Results

Using the [Database Query Browser](#) is a good way to test out a query before actually running it on a component. This can help you see what results will be returned so that the query can be modified to narrow down how the query should be formatted.

Checking for Slow Queries

There are several ways that slow running queries can cause improper behavior. See [Slow Queries](#) for more details.

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

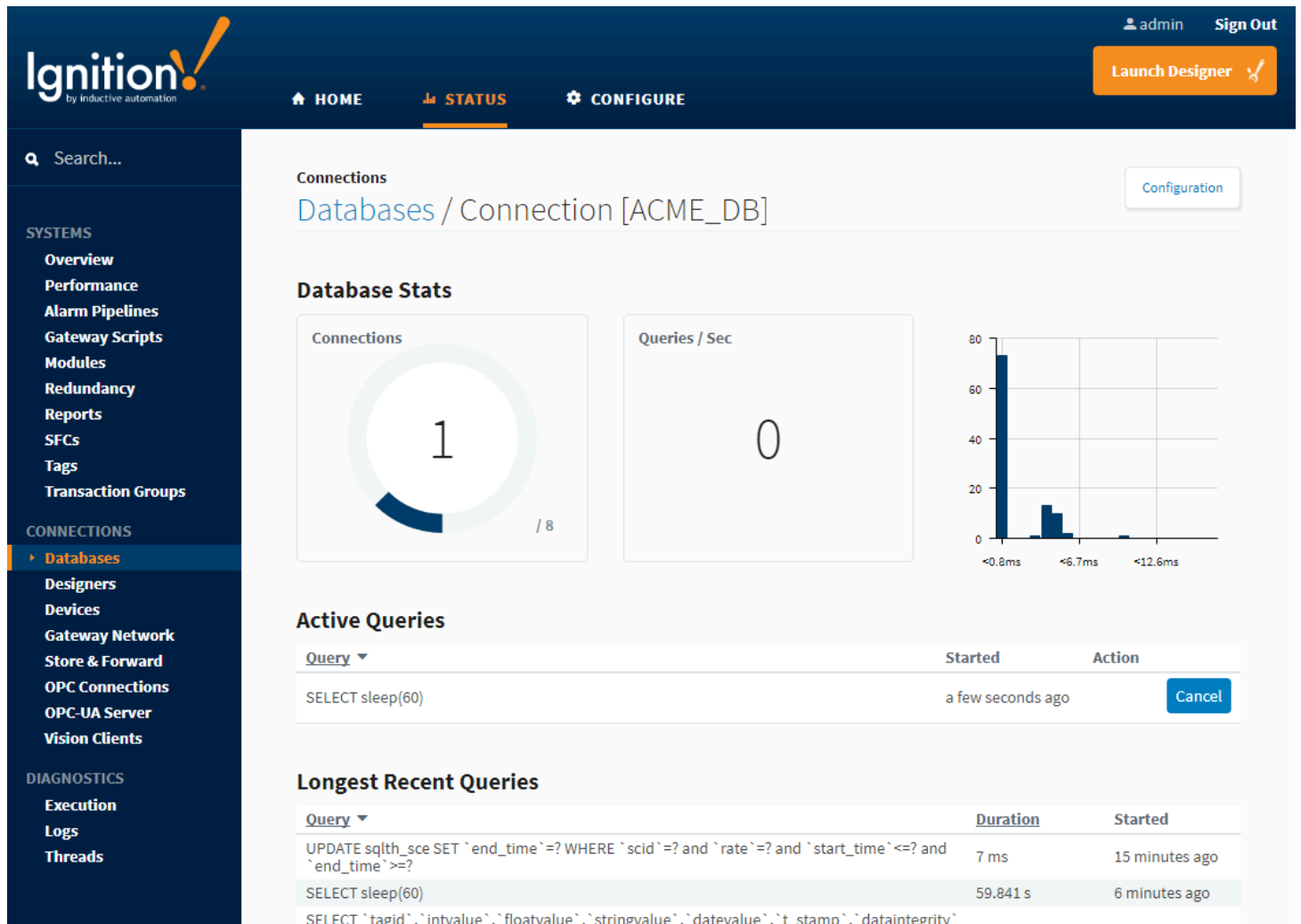
For documentation on the latest stable release, see the [latest version](#) (8.3).

Slow Queries

Slow running queries can be a big problem. Not only can data take a long time to display on the screen, but it can end up slowing down the whole client. Here, we take a look at some of the things that you can do when your project has a slow running query.

Identify the Slow Query

The first step in dealing with slow queries is identifying which query is actually running slowly. It is typically obvious when a window is opened and a component takes time before database data is displayed. Chances are, the query is on that component somewhere. However, it may also be a good idea to check the queries that are being run against that **database connection in the Gateway**. From the Gateway webpage, navigate to **Status** → **Connections** → **Databases** to see a list of all database connections. Clicking on the **Details** button to the right of the database connection will show all of the currently running queries, the most recent long running queries, as well as some basic metrics for that connection. Here, you can get a good idea of any queries that may be a little slower than the others.



Execute Against the Database Management Software

The next step in identifying the type of slow query we are dealing with is to run the query directly within the database management software. By cutting out Ignition, we can determine if the query is actually running slowly, or if there is a problem within Ignition that is making the query run slow.

Fast Query in the Database Management Software

If the query runs quickly in the database management software, then there are a few things we can take a look at that may help out.

Check the Database Connection

First, check to make sure the database connection is valid, and there are no warnings associated with it. You may want to go into the connection settings for that database and take note of the value of the

Max Active property, which determines the maximum number of active connections to the database as well as the **Max Wait** property, which is the number of milliseconds to wait for a connection to come available. To get to the database connections settings, go to **Configure** page of the Gateway webpage and select **Databases** → **Connections**, locate your database, and hit the **edit** button. Here, you can check the database settings. Open **Advanced Properties** at the bottom of the screen, and you'll find the **Max Active** property as well as the **Max Wait** property.

Check Currently Executing Queries

Back in the **Status** section of the Gateway Webpage on the **Database Connection** page mentioned above, we can see a list of currently running queries, as well as how many of the active connections are being used. If the max number of connections is being used, it may be that there are so many queries running that each query needs to wait for an active connection to open up. If this is the case, you may want to increase the amount of active connections to the database or take steps to [reduce the query load on the database](#).

Slow Query in the Database Management Software

If the query also runs slow within the database management software, then the query is just a slow query. Unfortunately, nothing within Ignition can speed up the execution of that query, so you would want to instead take a look at what the query is doing. If the query is pulling in lots of data, you can try breaking the query down into smaller queries, or writing the query in a more efficient way.

For particularly large tables, it may also be helpful to add an index to one of the table's columns. Indexes are something that the user can't see, but help the database speed up data retrieval. However, adding an index to a table will increase the amount of time that an update to the table takes, because the index also needs to be updated. For this reason, it is recommended to only make indexes on columns that are frequently searched against.

To make an index, most database management software have built in interfaces that allow you to customize the index on each table. An index can be made for a single column, or a combination of columns in the table. Talk to your Database Administrator about adding or updating table indexes.

Alternately, you may need to take a look at the database system as a whole. As the size of the database grows, you may need to update the hardware resources available to it. If the database is installed on a server with another system like Ignition, be aware that although the two systems are now sitting next to each other, they now have to share the hardware resources available to them, which may cause issues for both systems. In many cases, it is often better to have the database run on a separate server, which gives it ample room to grow.

This is documentation for **Ignition 7.9**, which is no longer actively maintained.

For documentation on the latest stable release, see the [latest version](#) (8.3).

SQL Query Volume Optimization

Overview

Leveraging a SQL Database can drastically improve the quality of a project, but improper database consideration while designing can lead to poor performance later on. This page contains some best practices and considerations when incorporating SQL queries.

Optimizing Individual Queries

Optimizing individual queries to run faster and be more efficient is very difficult to do properly and can vary widely depending on a number of factors such as the way the tables are setup in the database and what data is being pulled out. Because each database can vary widely from another, there isn't any general way of improving the efficiency of your queries. Instead we recommend becoming more familiar with the SQL language as well as how the data is setup in your system and exactly what data you want to retrieve. This knowledge can help you build better queries.

Your company may also have a database administrator who would manage the database system for your company and would be familiar with figuring out the best way of retrieving data. They may be able to help you retrieve the data that you need.

Always Think Large

Consider how many instances of a query may be running at any given point in time. A single **SQL Query Binding** will be called for each instance of the window that is open, so if 50 clients are all looking at the same window, 50 separate queries will be called. If the binding is configured to poll, then 50 queries will poll at the rate specified for this single binding. This is already a fair amount of work without factoring in other systems, such as Tag Historian.

To provide context, you can always check the **Status Section** of the Gateway Webpage to check current throughput of each database connection.

Ignition-WIN-VKBBHTNTPDV

USER MANUALSUPPORT

admin

Sign Out

Launch Designer

Ignition!by inductive automation

HOMESTATUSCONFIGURE

Search...

SYSTEMS

OverviewPerformanceAlarm PipelinesGateway ScriptsModulesRedundancyReportsSFCsTagsTransaction Groups

CONNECTIONS

DatabasesDesignersDevices

Connections

Databases

Configuration

Valid Connections

1 / 1

Total Throughput

0.3queries/sec

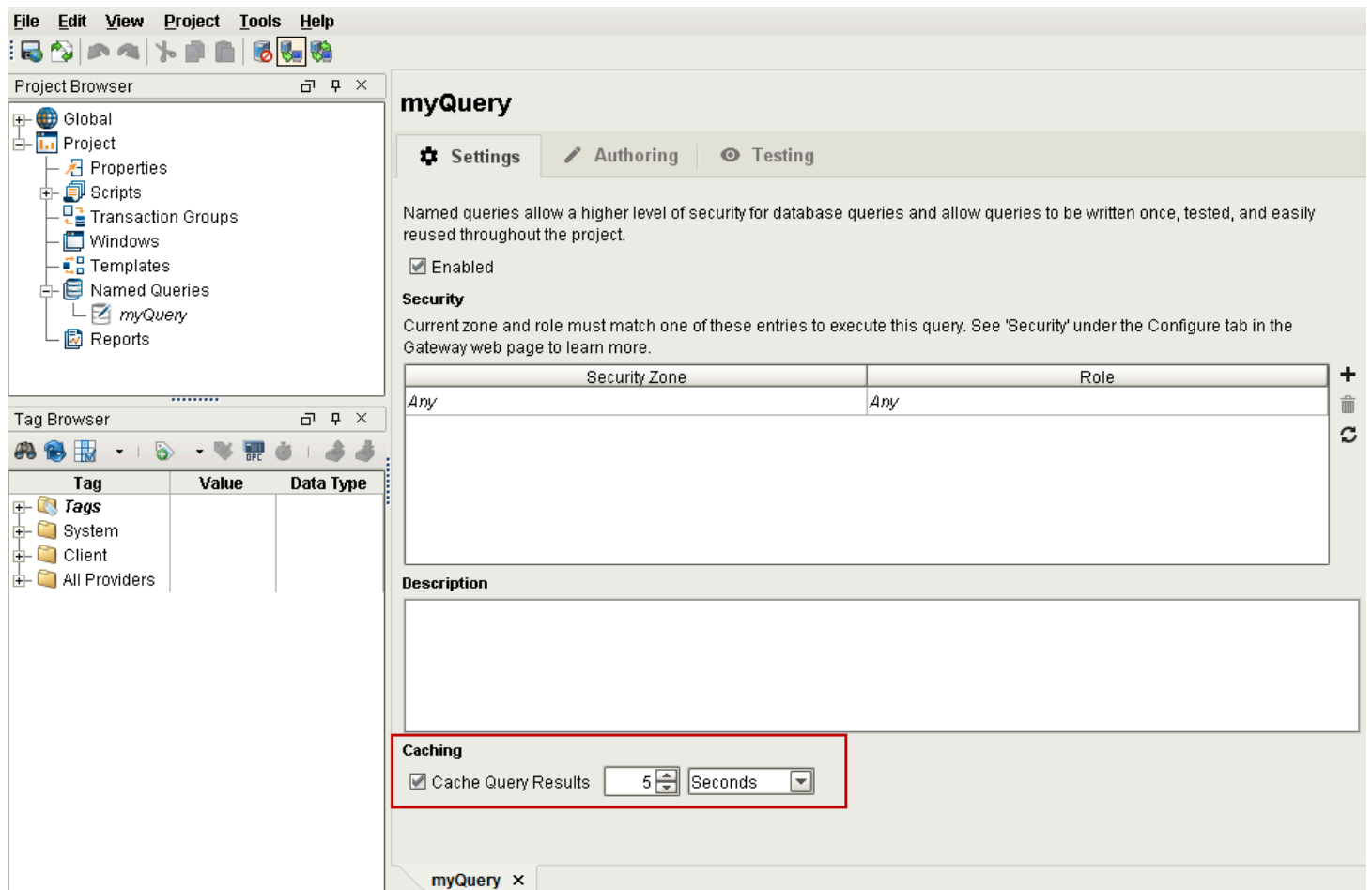
Name	Driver	Status	Connections	Throughput	
DB	MySQL ConnectorJ	Valid	0 / 8	0.3 queries/sec	Details

Use Cached Named Queries When Possible

Most resources in Ignition that can request a query will not cache the results for use by other resources: a SQL Query Binding that returns a result set will only do so for the one component, and can't be utilized by other resources in the same project. Thus, if two clients have the same window, the same query must be fired twice for both bindings to receive information from the database which is wasteful.

Named Queries are the exception. They can **cache the resulting dataset** for use by other resources in the project, as well as other instances of the same project. In the previously mentioned scenario, one client would trigger the Named Query to execute, and the other client would simply utilize the cached result set, reducing the number of queries running against the database.

If your project contains queries that poll slowly, or results sets that aren't frequently modified, then a Named Query with caching enabled is an efficient alternative to a SQL Query or DB Browse binding.



Use the Expression Language to Consolidate Multiple Queries

If multiple resources (such as multiple Tags, or multiple components) need separate values from the same database table, or a window contains multiple components that are all querying data from the same table, such as multiple Numeric Labels, it may be more efficient to have a single query run and fetch the large portions of the table, and then distribute the individual values to each component. This typically involves having some property or Tag query for several rows of data from a database table, and then using expression bindings or Expression Tags to retrieve individual values from the query.

Individual values may be retrieved from a dataset via the Expression Language: either an Expression binding on a property, or an Expression Tag. Here are two commonly used approaches to extracting a single value from a dataset in the Expression Language:

- The [Expression Language's Dataset Access](#) syntax. You may want to use the try() function in case the dataset is empty:

Pseudocode - Expression Language Dataset Access Syntax

```
{dataset}[rowIndex,columnIndex]
```

- The lookup function:

Pseudocode - Expression Language Lookup Function

```
lookup({dataset}, lookupValue, noMatchValue)
```

Restrict the Number of Query Tags in a UDT

Each Query Tag in a UDT will run a separate query per instance of the UDT. Assuming a scan class of one second, if a UDT definition contains 5 Query Tags, and there are 5 instances of that UDT, then there will be 25 queries executing every second.

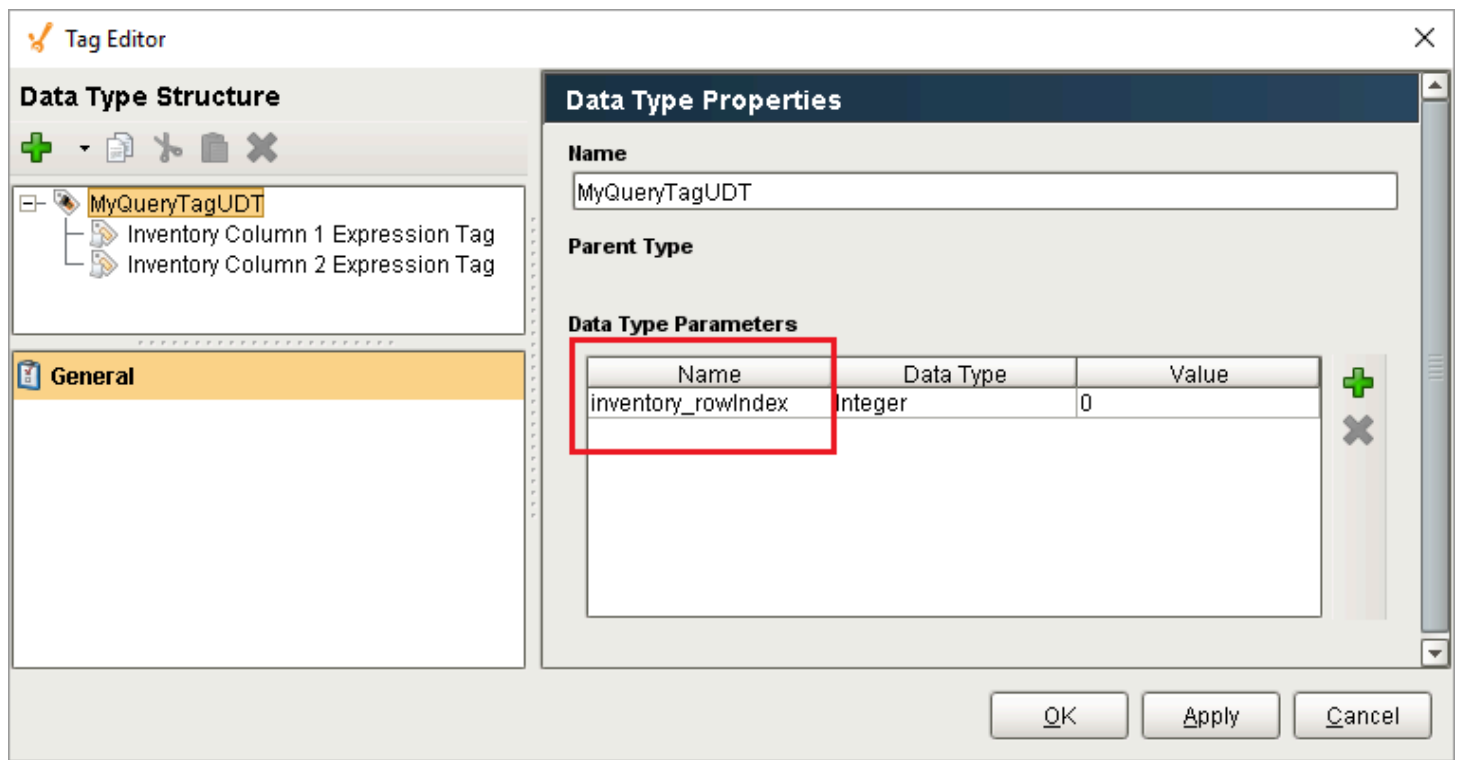
As mentioned above, the Expression Language can be used to reduce load on the database if multiple Query Tags are retrieving data from the same database table. Furthermore, UDT parameters can be utilized in the Expression Tags, so new UDT Instances can easily be configured to look up the appropriate values.

Single Database Tables

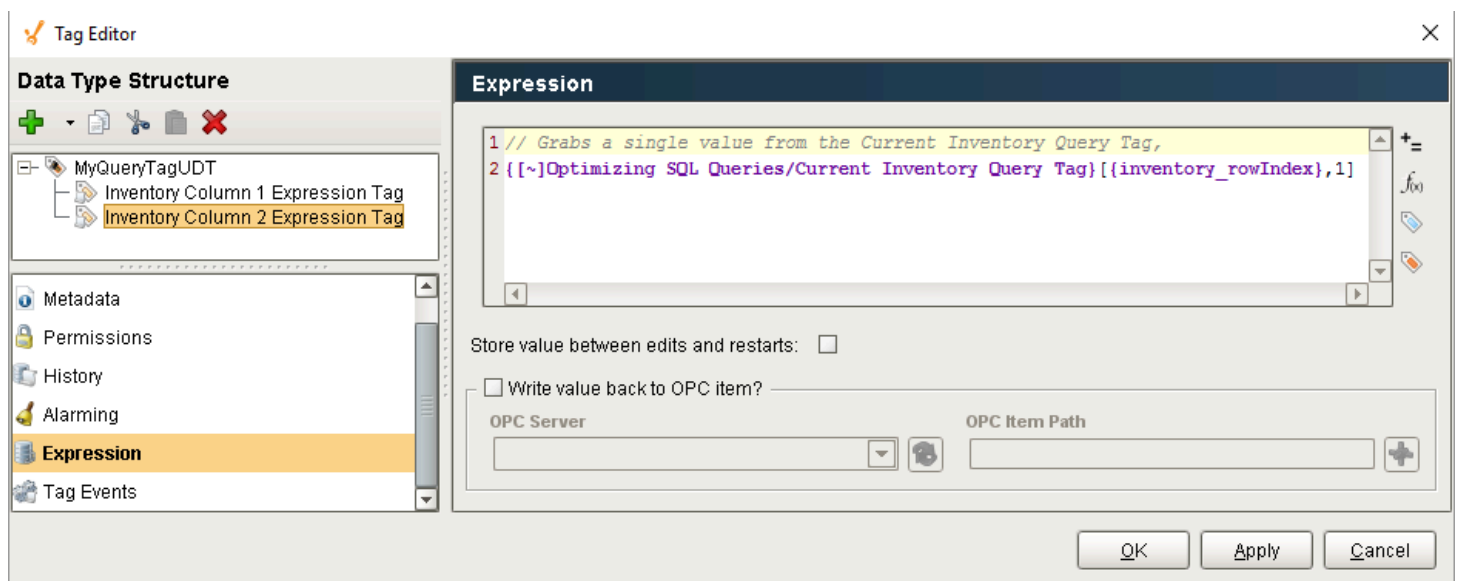
Below, the Tag named **Current Inventory Query Tag**, as the name implies, is a Query Tag retrieving multiple rows of data from a database table. We see that the highlighted **UDT Instance 2** contains two members: **Inventory Column 1 Expression Tag** and **Inventory Column 2 Expression Tag**, which are simply Expression Tags that are referencing individual cells from the **Current Inventory Query Tag**.

Tag Name	Value	Type
Current Inventory Query Tag	Dataset [10R x 3C]	DataSet
UDT Instance 1		MyQueryTagUDT
Inventory Column 1 Expression Tag	1	Int4
Inventory Column 2 Expression Tag	Bay 2	String
UDT Instance 2		MyQueryTagUDT
Inventory Column 1 Expression Tag	1	Int4
Inventory Column 2 Expression Tag	Bay 3	String
UDT Instance 3		MyQueryTagUDT
UDT Instance 4		MyQueryTagUDT
UDT Instance 5		MyQueryTagUDT

The UDT definition can use a parameter to specify an individual row in the Query Tag that each instance should focus on.



Each Expression Tag could use an expression like the following to look up individual values:



Multiple Database Tables

To add values from a separate database table, we simply need a separate Query Tag. In the image below, a new Tag named **Maintenance Query Tag** has been added, which is querying from a separate Database table. To incorporate this new data into our UDT instances, new Expression Tags have been added (**Maintenance Column 1 Expression Tag**, and **Maintenance Column 2 Expression Tag**) that simply reference specific values in the new Query Tag. Now, regardless of how many UDT instances exist in the Tag provider, we only have two Tags that are executing queries against the database.

For each separate table, we need to incorporate a single new Query Tag to collect all of the rows we want to show, add index parameters to the UDT definition, and add Expression Tags to our UDTs.

Tag Browser		
Historical		
Optimizing SQL Queries		
Current Inventory Query Tag	Dataset [10R x 3C]	DataSet
Maintenance Query Tag	Dataset [9R x 6C]	DataSet
UDT Instance 1		MyQueryTagUDT
UDT Instance 2		MyQueryTagUDT
Inventory Column 1 Expression Tag	1	Int4
Inventory Column 2 Expression Tag	Bay 3	String
Maintenance Column 1 Expression Tag	1	Int4
Maintenance Column 2 Expression Tag	item1	String
UDT Instance 3		MyQueryTagUDT