# Building SQL Commands

By Davin Church and Douglas Neman
Creative Software Design
(Last updated January 2020)

## Introduction

SQL databases are powerful tools for storing and processing most kinds of data and they work well when integrated into APL business applications. APL has had its own data filing system that works fine for many programmers' needs, but when systems become large or complex, a commercial database system will often provide much better capabilities for the programmer's use.

However, the syntax of the SQL language is sometimes complex and verbose, particularly from the standpoint of a long-time APL programmer. I feel that a toolkit is needed to help the APL programmer with such complexities by assisting him in building SQL language commands in a flexible application environment.

Granted, there's nothing in the SQL language that cannot be constructed relatively easily by concatenation of text. But SQL can be rather picky about formatting and grammar and it can be a chore to watch out for lots of little details, especially when you may have to do it in hundreds to thousands of places in a large application. In addition, it is sometimes necessary to write code that can run against multiple databases with slightly different command syntax, or the database system being used may have to change over time. In order to avoid rewriting the code every time, a toolkit could make the necessary adjustments at run-time to operate with whatever the database system in use requires.

For instance, whenever a user can enter their own optional "selection criteria" as input, how many times do you want to write the expression "`,(0≠⍴where)/' Where ',where`" to include optional restrictions? Wouldn't it be nicer if you had a simple subroutine to do that kind of work for you? And while it's at it, how about giving it 0 to *n* separate terms for that **Where** clause and have it string them together with the word "**And**" between each one? Or even build each of those terms internally given a list of field names to compare with and the data (supplied by the application user, possibly with "wildcard" characters) to compare them to (each field optionally, of course, since not all such selection data will normally be entered every time)? And don't forget what happens to your SQL commands when the user enters a "quote" character in their data if you don't include extra code everywhere to double potential quotes. (An unmatched quote is also a prime method that hackers use to break into web sites.) You must also be careful to handle any negative numbers that would require extra negative-sign formatting or large or fractional numbers that require a more-than-usual number of digits. Well, those are some of the kinds of things a toolkit can do automatically for you.

Herein is described just such a toolkit that you can include in your application (as a namespace) and use to dynamically build SQL language commands on-the-fly, based on static, user-entered, or program-adjustable data. It also makes short and simple commands even easier and more readable to include in your code. So if you've got an application that needs more than a few simple, unconnected tables (which is why you're using SQL in the first place, right?), here's a way to quickly and easily write very readable code to produce SQL commands.

# SQL Language Overview

SQL is primarily a data-manipulation language, and there are a few major commands used for that purpose. Many commercial SQL databases also have other commands for defining and managing the structural organization of the databases, but since those are not as consistent among vendors and are not often used in application code, they are not currently included in this package.

*Note:* SQL commands are normally provided as a single line of text, but wrapping the statements makes it easier for us to read here. (Most SQL databases also allow for statements to be spread over multiple lines in one fashion or another, too.)

(In this document, parameters/arguments/results are shown in *italics*, and SQL keywords are shown in proper-case **boldface**. Optional data is shown in *[italic square brackets]*. APL names and code are shown in an `APL font`.)

*Note:* This document mentions a few descriptions of simple SQL commands or clauses, but it otherwise assumes that the APL programmer has access to the full SQL documentation for his database so that they can construct statements appropriate for their application's needs. This document is not a tutorial on the SQL language.

The most-used SQL command is **Select**, for reading data, and it usually looks something like this, at most:

> **Select** *fieldnames*
> **From** *tablenames*
> **Where** *restrictions*
> **Order By** *sortfields*
> **Group By** *groupfields*
> **Having** *grouprestrictions*

Of these "clauses" (major parts of the SQL statement), only the **Select** and **From** clauses are required. The **Where** clause is almost always used as well (unless you're reading an entire table). The remainder of the clauses are available for more complex queries. All clauses that <u>are</u> present must be specified in the above order.

Also useful, since you need a way to get data into a database table, the Insert clause is usually structured like this:

> **Insert Into** *tablename [(fieldnames)]*
> **Values (***dataitems***)**

To modify individual items of data in specific records:

> **Update** *tablename*
> **Set** *field* **=** *value…*
> **Where** *restrictions*

And to remove entire records from the tables:

> **Delete From** *tablename*
> **Where** *restrictions*

Of course, there are a few other forms these commands can take, and a number of other statement types as well, but these are the major ones, and are the ones currently supported by this toolkit.

*Note:* **Insert**, **Update**, and **Delete** operations can also often be done via your programmatic SQL interface rather than with text commands. This can sometimes be faster and easier in code than building text commands and they can usually support non-printing characters and other kinds of special data (including APL symbols or encoded data).

These commands, once constructed as text, are passed to your favorite database using whatever interface you have available (via ADO, ODBC, or whatever). We shall concentrate here on simply constructing such commands using this toolkit.

# Toolkit Organization

These tools are distributed via a GitHub repository named SQLFns, in a namespace nominally called SQL. The namespace may be referenced, renamed or moved as desired by the APL developer. The functions are then called using the usual namespace-reference syntax. Because that namespace syntax varies depending on the environment, this document will leave off the namespace references and give examples of calling the functions directly.

So, to build a **Select** statement with all six clauses in it, you'd call six functions and simply catenate the results together, as follows:

```
cmd←Select fieldnames
cmd,←From tablenames
cmd,←Where restrictions
cmd,←OrderBy sortfields
cmd,←GroupBy groupfields
cmd,←Having grouprestrictions
```

An Insert statement would be constructed like this:

```
cmd←[fieldnames] Insert tablename
cmd,←Values dataitems
```

An Update statement can be specified as:

```
cmd←Update tablename
cmd,←fieldnames Set datavalues
cmd,←Where restrictions
```

And to Delete records:

```
cmd←Delete tablename
cmd,←Where restrictions
```

Each function is described below, but you should be able to see how each SQL statement is constructed one piece at a time with the given functions. Of course, they don't have to be built on separate lines – they could just be strung together with parentheses and commas if the line is short enough for your comfort.

There are a few other "utility-class" SQL-related tools included as well. For instance, one function makes simple **Select** commands (very common in applications) even easier to code. A cover function has been provided to build the command all at once (usually with either short constants or variables as individual arguments):

```
      cmd←Get fieldnames tablenames restrictions
```

This function will build an entire **Select** command for you, handling your quick and simple data retrievals in a single line of readable code, which could be passed directly to a data-fetch function, such as:

```
      user←,ADO.Get Get '*' 'Users' ('ID' Is loginname)
```

# Functions & Syntax

## And

*sql ← [expression(s)] And expression(s)*

This is a handy little function to join zero or more text expressions together with the word "And" inserted between them. Either argument may be an empty text vector, a simple text vector, or a nested vector of zero or more text vectors. All such vectors, with empties removed, will be joined together with the **And** conjunction. This is especially useful when the expressions to be joined together are not known in advance. This function is also used internally by `From`, `Is`, `Where`, and similar routines.

*Note:* Only use plain text in these expressions. Name-value pairs intended for use with `Is`/`Where`/etc. will not be processed properly. Name-value pairs that have already been processed by `Is` are now in plain text form and can be used with this function.

There is an additional utility feature available in this function. When joining SQL phrases together, it is often necessary to also enclose the result in parentheses, but only if the result is non-empty (to prevent SQL errors). If any *expression* term provided in the arguments is exactly the text value `'()'`, then any non-empty result of this function will be enclosed in parentheses.

### Examples

```
      'foo = 4' And 'goo = 7' 'hoo = ''bye'''
foo = 4 And goo = 7 And hoo = 'bye'
      cmd←cmd And 'item = ',⍕item
      cmd←cmd And where1 where2 where3
      '()' And 'goo = 7' 'hoo = ''bye'''
(goo = 7 And hoo = 'bye')
```

## Cmd

*sql* ← *Cmd* *argument argument …*

*sql* ← *Cmd* *[functioncode] argument argument …*

This is a cover function that is designed to be called from utilities rather than directly, as direct calls are more easily and effectively made by calling other functions themselves. The purpose of this function is to be able to call one of several other commands by supplying data only, rather than by calling the subroutines directly. There are some cases, such as when defining SQL data structures within other definitions, that only data may be specified. In such cases various subroutines may need to be called dynamically by the utilities handling that data, so those utilities can call *Cmd* instead and let the data itself determine the processing needed.

The primary use for *Cmd* is to call `Get` monadically with the *argument* list given. The first syntax above is the simplest and used for that purpose. The entire *argument*-list is passed as a single *argument*-list to *Get* and its results are returned from *Cmd*. (If `Get` needs to be called dyadically, see options below.)

*Cmd* is designed to be able to call `Get`, `Del`, `Ins`, or *Upd* upon demand. `Get` can be called by default without any special *functioncode*, but if a different subroutine is needed supply a *functioncode* as the first term of the *argument*-list. A *functioncode* always starts with the APL "Del" character ("∇") and is followed by the name of the subroutine desired (from the above list, including `Get` itself if desired). The named function is then called in place of `Get` and the remaining *arguments* are passed mondically to that function instead.

Occasionally, particularly in the case of `Get`, it will be needed to pass a left argument to the named subroutine. To use this feature, use the *functioncode* term as described above. (In this case the `Get` name cannot be assumed.) In addition, supply the APL "Alpha" character ("α") immediately to the left of the function name, to indicate that the function has one left argument. This will cause the first *argument* from the *argument*-list to be unnested and passed to the function as the left argument, while the remainder of the *argument*-list continues to be passed as the function's right argument. (More esoteric argument-options are also available – consult the function comments for details if necessary.)

### Examples

```
     Cmd '★' 'People' ('ID' Is id)
Select ★ From People Where ID = 3
     Cmd '∇Get' '★' 'People'
Select ★ From People
     Cmd '∇αGet' '??' '★' 'People'
??Select ★ From People
     Cmd '∇Upd' 'Students' (1 2⍴'First' 'Davin') 'Id = 3'
Update Students Set "First" = 'Davin' Where Id = 3
     Cmd '∇Del' 'People' ('ID' Is 3)
Delete From People Where ID = 3
```

## Del

*sql* ← *Del table restrictions*

This is just a simple cover function to make it easier to delete records from a table. A single *table* name is required, as is a *restriction* term. The *table* name is passed to the `Delete` function and the *restriction* is passed to the *Where* function, and the results are catenated together into the final SQL statement result. The two terms may be in any form acceptable to those subroutines.

### Examples

```
     Del 'People' 'ID = 3'
Delete From People Where ID = 3
     Del 'People' (('First' 'Davin') ('Last' 'Church'))
Delete From People Where "First" = 'Davin' And "Last" = 'Church'
```

## Delete

*sql* ← *Delete tablename*

This will construct the beginning of a SQL **Delete From** statement. Provide a single *tablename* as the argument.

**Warning:** Remember to always use a **Where** clause with this SQL statement so you don't accidentally delete your entire database table!

All field and table names used in these functions are sent through *Q* for quoting if they are reserved words or contain improper characters.

### Examples

```
     Delete 'Classes'
Delete From Classes
```

## Empty

*newdata* ← *default Empty rawdata*

This utility function's only goal is to operate on empty APL arrays (vectors or matrices). If the right argument is not empty (~0∊⍴*rawdata*) then it (*rawdata*) is simply returned without any changes whatsoever.

If the right argument is empty, then the function's result is the left argument (*default*) instead of the right argument. This gives you the option of returning some other (arbitrary) value or array in place of an empty array.

This is often used when a matrix is returned from a SQL command but it contains no rows at all. The APL prototype of the array (one column or the entire matrix) may be inappropriate for further processing that would otherwise be able to handle empties. `Empty` can then be used to replace it with an empty array of the expected data type so that processing can continue normally.

This could also be used to replace an empty array with a non-empty one containing default or explanatory values, if desired.

## Examples

```
      ⊃¯1 Empty ADO.Get Get 'Id' 'Users' ('Name' Is 'nobody')
¯1
```

# From

*sql* ← `From` *tablelist*

*sql* ← *memory* `From` *tablelist*

*tables* ← *parse* `From` *tablelist*

This function provides one of the most important features of the toolkit. In general, it creates a SQL **From** clause, listing the table or tables to be processed in the SQL statement. All the functions that take table names call it. In its simplest form, it provides only trivial functionality… listing the table name or names after the **From** keyword. However, it can also do much more – so let's describe this one by example.

If only a single table is to be processed, then it just lists that table (quoted as necessary) along with any alias name provided (see below). But if multiple tables are to be joined together (as frequently happens even in slightly complex database applications), then it makes your code <u>much</u> easier.

All field and table names used in these functions are sent through `Q` for quoting if they are reserved words or contain improper characters.

## Basic Needs

Let's start with the basics. To select from a single table, just list its name in the argument, such as:

```
      From 'Table1'
From Table1
```

If you'd like to assign an alias (such as "`t1`") to that table (for easier use in specifying the remainder of the SQL statement), use an assignment arrow with the alias name first, like this:

```
      From 't1←Table1'
From Table1 As t1
```

To select from more than one table at once, un-joined, just list them as a nested vector of names (aliases are still fine to use in any example, if you want them):

```
      From 'Table1' 'Table2' 'Table3'
From Table1, Table2, Table3
```

In some SQL implementations, you can further refine the connections among the tables by using the **Where** clause to perform an "implicit join". But this is an old (and often not as well optimized) syntax and is no longer generally recommended. (It can be used if needed by just providing a list of table names to `From`, as above.) Implicitly-joined tables function as if they were Cross(Cartesian)-joined and the **Where** clause is used to restrict which of the many possible record combinations are actually returned.

However, most SQL systems encourage (and some may require) you to use the explicit joining syntax directly in the **From** clause for best operation. This grammar can be quite long and dizzying, so let's have *From* build that for us. (This is where it starts getting fun.)

Let's say we want to have Table1 joined to Table2 where F1 (in Table1) is equal to F2 (in Table2). We'd specify that using *From* with:

```
      From 'Table1(F1)→Table2(F2)'
From Table1 Inner Join Table2 On Table1.F1 = Table2.F2
```

(Note the "Branch Arrow" pointing from the starting table to the joined table. There are other symbols that may be used there instead, which we'll mention later, but this is the most commonly-used one.)

If we've got a two-field join, it looks more like:

```
      From 'Table1(F1,F3)→Table2(F2,F4)'
From Table1 Inner Join Table2
   On Table1.F1 = Table2.F2 And Table1.F3 = Table2.F4
```

As you might notice, the generated clause is getting pretty long (and the text is wrapped in this document), and the table names are listed three times apiece. I wouldn't want to have to do that by hand throughout a large application.

Now let's say that there's some additional data needed from Table3, joined from another field in Table2. Now you're looking at something like:

```
      From 'Table1(F1,F3)→Table2(F2,F4)' 'Table2(F5)→Table3(F6)'
From (Table1 Inner Join Table2 On Table1.F1 = Table2.F2 And
   Table1.F3 = Table2.F4) Inner Join Table3 On Table2.F5 = Table3.F6
```

It's getting to look a bit complicated, isn't it? The *From* argument is shorter and noticeably easier to read and write in the function.

## Join Memory

But even this simpler format isn't something I'd want to be typing over and over in the code. I'd like to simplify it even further by teaching *From* how my database structure is organized for this application. This is where the second syntax definition above comes in. Providing a left argument "*memory* command" to *From* tells it that we're managing its built-in "join *memory*".

```
      '+' From 'Table1(F1,F3)→Table2(F2,F4)'
      '+' From 'Table2(F5)→Table3(F6)'
```

It now knows that when joining Table1 to Table2, fields F1 & F3 are to be linked to fields F2 & F4, and Table2 can also be joined to Table3 on fields F5 and F6. Now when we specify any of these tables to be joined together, we no longer have to specify the fields on which they're joined. So our function call is now just:

```
      From 'Table1→Table2→Table3'
From (Table1 Inner Join Table2 On Table1.F1 = Table2.F2 And
   Table1.F3 = Table2.F4) Inner Join Table3 On Table2.F5 = Table3.F6
```

OK, **now** it's getting as easy as I like to see it! This is especially helpful when you have to work with the same group of joined tables in many places in the application, so you don't have to laboriously list the same specifications over and over.

To create such definitions, you can use the following *memory* codes:

| | |
|---|---|
| + | Add join definition(s) to *memory*. |
| − | Remove join definition(s) from *memory*. |
| ← | Erase all join definitions from *memory*. Add any new definitions to the now-empty *memory* if they are provided at the same time; otherwise leave the *memory* empty until a "+" command is used. |
| ? | Query the list of join definitions currently in *memory*.<br>The right argument may contain one or more separate and unadorned table names whereupon only joins involving any of those tables will be returned. Otherwise all the definitions in *memory* are returned. |

Occasionally, there may be more than one way to join two tables together, both memorized, and `From` won't be able to choose one if only table names are specified. If needed, you can indicate which alternate possibility to use by listing either or both of the indexes to specify a particular choice.

Also be aware that the default join definitions are ordered, so you cannot use a definition of `'A→B'` to supply a default when `'B→A'` is requested. A separate `'B→A'` definition is required for that use.

## Join Types

In the examples above I've only been using "Inner Joins", shown above using the APL "Branch Arrow" ("→"). SQL actually defines several different types of joins that can be performed. `From` has symbols that can indicate these alternate types, for whenever you'd like to use them. Here are your choices:

| | |
|---|---|
| → | Inner Join *(very common)* |
| > | Left [Outer] Join *(also common)* |
| < | Right [Outer] Join |
| ≠ | Full [Outer] Join |
| ∘ | Cross (Cartesian product) Join *(no join fields)* |
| = | Natural Join *(no join fields)* |

## Advanced Features

### *Custom Join Criteria*

Joining tables on matching field names should suffice for nearly all situations. However, a need occasionally arises to include other conditions when joining tables together. For example, Table1.F1 may need to be joined to Table2.F2, but Table2 also requires a year restriction. In SQL this might look something like this:

```
From Table1 Inner Join Table2
  On Table1.F1 = Table2.F2 And Table2.FYear > 2000
```

The simplified syntax that `From` uses (as described above) does not allow for this sort of criteria to be specified directly. However, there is a way to add custom join conditions like this to the standard field-to-field joins. To signify that custom join conditions are to be used, nest the from/to table name pair an extra level deeper (to depth 3), and append one or more extra conditions (as nested items) to the end of the usual specification. (This can only be used as part of a pair of table names and not a longer chain.) Be careful to nest it the further extra level if only one unnested join condition would otherwise be used, so that `From` doesn't interpret the item containing criteria as a separate join. For instance, to specify the above example, use:

```
      From ⊂'Table1(F1)→Table2(F2)' 'Table2.FYear > 2000'
From Table1 Inner Join Table2
  On Table1.F1 = Table2.F2 And Table2.FYear > 2000
```

There can be more than one such custom criteria term after the tables are listed, and they are all included in the join's **On** phrase by "**And**"ing the terms together, in the same way that multiple field names are handled. Each additional criteria term can be constructed by calling *Is*, or can be provided in any structure suitable for use by the *Where* utility. This includes the ability to pass a further-nested pair of items so *Where* knows to call *Is* internally on the pair of arguments. (See *Where* and *Is* for detailed descriptions of acceptable data structures.)

Keep in mind that these custom criteria terms will be added to any field-based matching, whether explicitly specified or assumed from defaults. If the custom criteria needs to be used <u>in place of</u> default join conditions, or there are no default join conditions, then list an empty pair of parentheses after each table name to indicate that only the custom criteria is to be used.

Very, very rarely, a join needs only these custom criteria terms without any field-based matching, and the join needs to be memorized, and especially if more than one such join between those tables needs memorization. A problem arises here because calling *From* to bring up the memorized join condition requires the use of the named fields but in this case there are no named fields. *From* provides yet another advanced feature for use in such rare cases. You may specify one or more field names in the field-based join criteria so they can be found in the memorized list but will <u>not</u> be included in the resulting SQL command. This very special case is handled by "commenting" the field names (placing an APL "lamp" symbol ["⍝"] in front of each name). Retrieving the memorized version of the join could then be done either with or without the "lamp" symbols, and it can be distinguished from other very similar memorized custom joins. Here is what the syntax would look like (if it wasn't being memorized):

```
    From ⊂'Table1(⍝F1)→Table2(⍝F2)' 'Table2.FYear > 2000'
From Table1 Inner Join Table2 On Table2.FYear > 2000
```

### *For Utility Use Only*

There is a third syntax available that is just for use by other utility functions. Specify a left argument of `'⊂'` to request that the right argument be parsed in the normal fashion, but a nested array of the parsed and processed data should be returned instead of a spelled-out SQL command. This may be used if a utility program needs to analyze an unknown argument to find out what *tables* and fields it will reference. (Names are returned exactly as specified and aliases are ignored.) The returned structure will contain one row per join (or unjoined *table*):

| | |
|---|---|
| [;1] | "From" *table* name |
| [;2] | Join type character scalar (a space character if not joined to a second *table*) |
| [;3] | "To" *table* name, if joined |
| [;4] | Nested matrix of field names used for this join (empty if unjoined), one row per join criteria (usually a field-pair comparison), in the structure: |

| | <u>Normal joins</u> | <u>Custom join criteria</u> |
|---|---|---|
| [;1] | "From" field name | `''` |
| [;2] | `'='` | `'⍵'` |
| [;3] | "To" field name | Custom data (*Where*-compatible) |

# Get

*sql* ← *[selectoptions]* `Get` *fields tables [restrictions [ordering [grouping [having]]]]*

This is just a simple cover function to make it as easy as possible to retrieve data without any more typing than necessary. The left argument of `Get` is optional and if present is normally just passed as the left argument to the `Select` function call. The right argument is a two- to six-item nested vector, each of which is passed to the appropriate subroutine and the results are catenated together into a whole **Select** statement. This is most often used with a three-item argument. For details on the structure of the individual items, see the descriptions of `Select`, `From`, `Where`, `OrderBy`, `GroupBy`, and `Having` (respectively).

A `Get`-specific *selectoptions* may also be included that is a character vector containing one or more `'?'` characters. It is not passed on to `Select`, as is usual for these options. This option will simply prepend those characters to the (beginning of the) SQL command as a typing convenience so that they need not be catenated explicitly. This is useful for passing to database-access functions that allow such prefixes to specify special operations.

## Advanced usage

`Get` can also take a nested matrix with two to four columns, where each row of the matrix is a separate query in the above order. `Get` will generate a **Select/Union** SQL command as its result. The usual SQL restrictions on using **Union** apply, in that the number of *fields* selected in each **Select** must match and their corresponding types must be compatible. The *grouping* and *having* clauses are not permitted in a **Union**. Only one *ordering* clause is permitted, so each row's *ordering* must either be identical or at most one row may include an *ordering* parameter. By default, duplicate rows from the separate queries may be returned from the resulting SQL command, which will be generated with the **Union All** operator. This may be changed to use the **Union** operator instead to remove duplicate rows by including a `Get`-specific *selectoptions* `'∪'` (a mnemonic for "unique") in the left argument. Be advised that use of this duplicate-removal request may require more processing time for the database to perform.

## Examples

```
     Get ('Name' 'Phone') 'People' 'ID = 3'
Select Name, Phone From People Where ID = 3
     Get '*' 'People' ('ID' Is id)
Select * From People Where ID = 3
     Get '*' 'People' (('First' 'Davin') ('Last' 'Church'))
Select * From People Where "First" = 'Davin' And "Last" = 'Church'
     '??' Get '*' 'People'
??Select * From People
     Get 2 2⍴'State' 'Customer','State' 'Supplier'
Select State From Customer Union All Select State From Supplier
```

# GroupBy

*sql* ← `GroupBy` *field(s)*

Simply provide zero or more *field* names as a nested vector of names. Like most similar functions in this toolkit, commas are inserted automatically as needed, and an empty vector is returned when the argument is empty.

When calling `Select` with a variable containing specific *field* names to return, it is often convenient to use the same variable (or portion thereof) to supply to `GroupBy` so that the names need not be repeated.

All field and table names used in these functions are sent through `Q` for quoting if they are reserved words or contain improper characters.

## Examples

```
      GroupBy 'Department' 'Manager'
Group By Department, Manager
```

# Having

*sql* ← `Having` *[group-restriction]...*

Provide zero or more *group-restrictions* in any of the following forms:

- A single text vector containing an entire **Having** clause (or an empty vector).
- Zero or more phrases to be `And`-ed together to produce the **Having** clause.
- Zero or more phrases, as above, any of which may be a two-item nested vector containing a *formula-value* pair to be passed to `Is` to turn it into a single *restriction* expression.
- A two-column matrix containing *formulas* in `[;1]` and *values* in `[;2]`, each row of which is to be passed to `Is` for *restriction* formatting.

This list of expressions is very similar to the argument to `Where`, except that there is an additional assumption being made for `Having`. `Where` most often takes simple field names in its comparisons, but the SQL syntax of **Having** requires it to use a "summary" formula for each comparison, so it cannot take plain field names there. Therefore, the *formula* argument(s) for `Having` will never be "quoted" as field names since they are already expected to have non-alphanumeric characters in them. (You'll have to use `Q` or `Math` yourself, as needed to preserve system-independence, to build quoted field names into the formulas.) Also, since "<", ">", "=", and "⋆" are often legitimate characters in a SQL formula, they are disallowed as special `Having` (which uses `Is`) comparison operations – use the "Not" ("~") APL symbol to invert other relational tests instead, if needed.

Note that SQL requires the use of a **Group By** clause before a **Having** clause can be used. `Having` will not produce a **Having** clause at all if no expressions are provided.

## Examples

```
     Having 'Count(*) > 100'
Having Count(*) > 100
     Having ('Sum(PayRate)≥' 100000) ('Count(Distinct Manager)~' 1)
Having Sum(PayRate) >= 100000 And Count(Distinct Manager) <> 1
```

# Ins

*sql* ← `Ins` *table values*

*sql* ← `Ins (`*table fields*`)` *values*

*sql* ← `Ins` *table* `((`*field value*`) (`*field value*`) …)`

*sql* ← `Ins` *table* `((`*n*`,2)⍴`*field value field value …*`)`

This is just a simple cover function to make it easier to insert records into a table. If values are being inserted for all *fields* in the *table*, in table-order, then use the first syntax above. Otherwise, use one of the other three optional syntax descriptions to supply the *field* names in the same order as the *values*. Nesting and shape are important when using these alternate forms.

The *table* name (and *fields*, if specified) are passed to the `Insert` function and the *values* are passed to the `Values` function. Those results are catenated and returned as the final SQL statement result.

## Examples

```
     Ins 'Students' ('Davin' 'Church' 'M')
Insert Into Students Values ('Davin', 'Church', 'M')
     Ins ('Students' ('First' 'Last')) ('Davin' 'Church')
Insert Into Students ("First", "Last") Values ('Davin', 'Church')
     Ins 'Students' (('First' 'Davin') ('Last' 'Church'))
Insert Into Students ("First", "Last") Values ('Davin', 'Church')
     Ins 'Students' (2 2⍴'First' 'Davin' 'Last' 'Church')
Insert Into Students ("First", "Last") Values ('Davin', 'Church')
```

# Insert

*sql* ← *[fields]* `Insert` *tablename*

This generates the beginning of a SQL **Insert** command. Only one *tablename* is permitted. If no left argument is given, the SQL command will expect values for every field in the table, in the table-defined order. Otherwise, specify a left argument of nested *field* names to be included in the command. As previously mentioned, this is normally used with the `Values` function.

All field and table names used in these functions are sent through `Q` for quoting if they are reserved words or contain improper characters.

## Examples

```
     Insert 'Students'
Insert Into Students
     'Name' 'Address' 'Phone' 'Sex' Insert 'Students'
Insert Into Students (Name, Address, Phone, Sex)
```

# Into

*sql* ← `Into` *table*

This is used to build the **Into** clause of a **Select**…**Into** SQL Command, which is not commonly used. The **Into** keyword is also used in the **Insert Into** command, but the `Insert` function will generate that keyword automatically so `Into` is not used for that purpose.

A single table name is provided for the SQL **Into** phrase.

All field and table names used in these functions are sent through `Q` for quoting if they are reserved words or contain improper characters.

## Examples

```
     (Select '*'),(Into 'EmpCopy'),From 'Employees'
Select * Into EmpCopy From Employees
```

# Is

*sql* ← *field(s)* `Is` *value(s)*

This function is conceptually simple but it is also the most flexible and deceptively useful function in this toolkit. It is also one of the largest and most potentially complicated utilities here, so we'll describe it by example, too.

All field and table names used in these functions are sent through `Q` for quoting if they are reserved words or contain improper characters.

## Basic Needs

At its most basic level, given a *field* name on the left and a *value* on the right, it generates a term of the SQL **Where** clause that restricts records to those where that *field* has that *value*. Of course, it can tell the difference between text and numeric *values*, so it trims trailing spaces from text *values* and encloses them in the proper quote marks, and it makes sure numeric *values* are formatted reasonably including negative and floating point values. For example:

```
      'Name' Is 'Davin    '
Name = 'Davin'
```

Of course, it doubles the quote marks in any text it encloses in quotes, which provides another minor convenience on the side… there's no need to pre-check the users' input to see if they've used quote marks as data, and if you're working with a web site this also prevents "SQL Injection" hacking attacks of the web site's character fields (input boxes, URL parameters, POST data, and cookies).

Just the formatting by itself is handy enough, avoiding lots of typing and reading of code (doubled quote marks, commas, parentheses, numeric formatting, etc.), and you can pass its results to other functions such as `Where`, `Having`, and `Get`. But these other functions will also take *name*-*value* pairs directly and will call `Is` internally to do the appropriate formatting so you don't have to be constantly typing in the function name, either, such as:

```
      Where ⊂'Name' 'Davin    '
Where Name = 'Davin'
```

In addition, you may supply more than one *value* to be used with a single *field* name. In such a case, `Is` will use that list of *values* as alternatives, any one of which may be matched by SQL. These situations will generate a parenthesized list of expressions **Or**-ed together, or more commonly, a SQL **In** expression, such as:

```
      'Choice' Is 2 3 5 7
Choice In (2, 3, 5, 7)
```

If a text *value* contains an asterisk ("*") character, which is equivalent to the APL "Star" symbol ("⋆"), representing a "wild card" that can match any characters at all, then `Is` will generate a SQL **Like** operator (using **Like**'s "%" symbol) instead of just an "=" operator, like this:

```
      'Name' Is 'Da⋆'
Name Like 'Da%'
```

And multiple values can be checked this way, too:

```
      'Name' Is 'Fred⋆' 'Bob' 'Rob⋆'
(Name = 'Bob' Or Name Like 'Fred%' Or Name Like 'Rob%')
```

One thing you'll want to watch out for is matching any of a list of scalar text *values*. Since a list of scalars is the same as a simple vector in APL, `Is` can't tell that you wanted multiple *values*. To tell it so, just nest the *values* appropriately to indicate a list of choices. The easiest way to do this is with the "Ravel-Each" function ("`,¨`"), like this:

```
      'Flag' Is ,¨'ABCX'
Flag In ('A', 'B', 'C', 'X')
```

Since the input to `Is` is often going to be a value entered by a user, and such values may often be left blank to mean "everything", `Is` will normally ignore *values* of `''`, or any vector composed entirely of spaces or asterisks (so you don't have to trim them yourself), such as:

```
      'Name' Is '' ⍝ (an empty vector is returned)
```

In most cases, you should use this feature when the SQL clause being constructed may or may not include terms for certain elements. This is typically much more robust and easier to manage than using `:If` commands or selective compression to conditionally include or exclude various elements.

However, this can present a bit of a problem when you're actually trying to <u>match</u> a blank *value*, so the logic for matching a <u>list</u> of *values* is adjusted to continue to ignore empty vectors in the list but not listed vectors that are composed entirely of spaces. One way to match a specific blank *value* (often you know this ahead of time), is to simply nest a space character. The same "Ravel-Each" function will do the trick here, and the space may also be combined with other scalar characters if desired. So one way to check for an explicitly blank *value* is:

```
      'Name' Is ,¨' '
Name = ''
```

But even this makes certain constructions awkward to code. So a simpler way has been provided to match all-blank or empty *values* (without affecting any other *values*) with a special symbol that can be added to override this skip-all-blanks rule. See the description of the "Zilde" symbol ("⍬"), below, for more information.

There are also needs for matching a special kind of value called a **Null**. In SQL, a **Null** value does not equal (or not-equal) <u>any</u> specific value. To look for records that have **Null** values, SQL requires a special syntax. `Is` can generate that syntax if you give it `⎕UCS 0` or `⎕NULL` as an argument.

```
      'Name' Is ⎕UCS 0
Name Is Null
```

Notice that that is not the same as just using `'Null'`:

```
      'Name' Is 'Null'
Name = 'Null'
```

## <u>Operators</u>

Now (no we're still not finished), this is all well and good for matching specific values, which is what you normally want to do with SQL **Where** clauses. But sometimes you want a different kind of test to be made, such as a "not equals". If you'll simply include the APL "Not" symbol ("~") or the "Not Equals" symbol ("≠") as part of the *field* name, the test will be changed from "equals" to "not equals". Here's how that looks:

```
      'Amount~' Is 0
Amount <> 0
```

And it can be used with lists, too:

```
      'Qty~' Is ⍳5
Qty Not In (1, 2, 3, 4, 5)
```

And that's not the only alternate test you can do, because sometimes you want an inequality comparison done. For that, you would use other reserved symbols in the *field* name:

```
      'Qty≥' Is 100
Qty >= 100
      'Qty<' Is 500
Qty < 500
```

Or even **Between** two values (however your database defines "between"; it requires a pair of values):

```
      'Qty→' Is 100 500
Qty Between 100 And 500
```

Sometimes you need to make a test against a subquery in another table rather than against constant data. For this, the "∊" operator is provided. When this is used with a *field* name, the right argument must be a SQL **Select** command (subquery), which may in turn be generated with these utilities. This text value is enclosed in parentheses and used as-is (as if the "⎕" option symbol, below, was specified) in a "*field* **In** (**Select** …)" phrase. (Optionally, a nested vector may be provided in `Get` form and `Is` will call `Get` to format it into text before proceeding.) Alternatively, the "∊" operator may be specified without any field name at all and an "**Exists** (**Select** …)" phrase is generated instead.

```
      'Name∊' Is Get 'FirstName' 'Employees'
Name In (Select FirstName From Employees)
      '∊' Is Get '⋆' 'Employees' ('Pay>' Is 100000)
Exists (Select ⋆ From Employees Where Pay > 100000)
```

Your choices of operators which you can use (anywhere within the *field* name) are:

| | |
|---|---|
| = | Equals *(default if none specified)* |
| ≠ | Not Equals |
| < | Less Than |
| > | Greater Than |
| ≤ | Less Than or Equal To |
| ≥ | Greater Than or Equal To |
| → | **Between** *(exactly two values required)* |
| ∊ | **In** (a subquery), or (a subquery value) **Exists** |
| ~ | Not: Invert any of the other tests |

Granted, you shouldn't need most of these very often, but they're handy when you do.

## Option Symbols

In addition (and no, we're still not finished yet), there are some other symbols you can use in the *field* name for special processing. The simplest of these is the APL "Take" symbol ("↑") to indicate that the test is to be made case-insensitive (upper-case), for SQL fields that are not pre-defined that way:

```
      'Name↑' Is 'Davin'
Upper(Name) = 'DAVIN'
```

Also, as mentioned above, `Is` does not normally allow comparison to empty vectors or character *values* composed entirely of spaces. To avoid this, you may nest a space character as described above to have such a *value* compared directly. Occasionally, though, you'll have special circumstances that make it more complicated to code such a nested *value*, such as when you don't know ahead of time whether the *value* is empty or not but it should be compared in either case. For such circumstances, and also for the simplest of comparisons against a constant blank in the database, a special symbol may be included in the *field* name to indicate that empty/space *values* are to be specifically compared. Use the APL "Zilde" symbol ("⍬") to force that comparison:

```
      'City⍬' Is ''
City = ''
```

Sometimes a SQL field needs to be compared directly to another *field* or to a computed value (formula). (Use `Q` on any *field* name to ensure proper SQL quoting.) For that, use an APL "Quote-Quad" symbol ("⍞"):

```
      'Name⍞' Is Q 'Manager'
Name = Manager
```

You may be using this option, for instance, to supply a "substitution parameter" to SQL (an argument value to be specified later). The syntax for this varies depending on the database and interface, but it may be as simple as listing a "?" character:

```
      'Name⍞' Is '?'
Name = ?
```

But what if you need to later supply such values with imbedded "wildcard" characters (a more advanced need), and thus you need the expression generated with a **Like** operator instead of "="? That's when you use an APL "Star" character ("⋆"), but in the *field* name instead of the data *value*:

```
      'Name⍞⋆' Is '?'
Name Like ?
```

If you need such fixed values prepared for use with the **Like** operator (such as when preparing commands with both "⍞" and "⋆" as in the example above), `Is` has a special utility feature to do that, too. Just call it with the data to be prepared as the right argument and <u>just</u> the APL "Star" ("⋆") character <u>without a *field* name</u> as the left argument. The **Like**-encoded value will be returned.

If you want to specify a **Like** operator with a fixed value, you can do that, too, but then `Is` won't convert "⋆" into "%" characters and do its other **Like**-operator processing, assuming that you've done that already:

```
      'Description⋆' Is '⋆too%many_odd\\ities'
Description Like '⋆too%many_odd\\ities'
```

Ok, there may be some places where you really want to put an asterisk ("*") into the database and search on it, rather than using "⋆" as a "wildcard" character. If you have a particular field where that's needed, you can specify the *field* name with an imbedded APL "Logarithm" symbol ("⍟") to indicate that **Like**-operator processing is to be skipped:

```
      'Name⍟' Is 'me⋆too'
Name = 'me⋆too'
```

And once in a while you really need to compare a SQL computation or formula to a value and you need the *field* name to be a calculation and avoid being treated as a simple name. For that, include the APL “Execute” symbol (“⍎”) in the *field* expression and it won’t treat it as an invalid SQL *field* name. However, be warned that since “=”, “<”, “>”, and “★” are often valid characters in a SQL expression, those symbols cannot also be used to indicate special Is commands at the same time as “⍎”.

```
      'Left(Name,5)⍎' Is 'Davin'
Left(Name,5) = 'Davin'
```

Alternatively, as a more advanced option, you may ask Is to call the Math utility for you to generate a SQL computation expression. To do that, the *field* name should specify the *expression* to be used as the left argument to Math, and that *expression* must be the first item of a nested vector (used together as one *field* name). (The nested structure itself indicates the use of a Math *field* and no particular option symbol is required.) This first nested item is passed to the left argument of Math (as the *expression* to use) and the remainder of the vector is passed as the right argument to Math. (See Math for further details on these arguments.) The result from Math is then used by Is as the computed *field* name, and will not be further processed by Q (so it won’t be quoted as an invalid SQL *field* name), as if the “Execute” option symbol (“⍎”) were specified. Please note that this method generates a Math left argument to Is, unlike the “Divide” option symbol (“÷”) below which generates a Math right argument to Is (see below).

In addition to being able to specify a Math *expression* in place of a *field* name, Is also allows you (if you need it) to specify a Math *expression* as a SQL-computed *value* for comparison. To indicate this, use the APL “Divide” symbol (“÷”) as part of the *field* name. When this is done, the *value* item in the right argument of Is must be a nested vector. The first item of this nested vector is used as the left argument (*expression*) for Math and the remainder of the nested vector is used as the right argument (substitution *values*). The result from Math is then used as a computed comparison value and is included in the SQL output as-is, as if the APL “Quote-Quad” option symbol (“⍞”) was used in the *field* name.

So, to recap this section… in addition to the comparison operator symbols, you may also specify any of the following symbols within the *field* name for special handling:

| | |
|---|---|
| ↑ | Upper case (case-insensitive) check |
| ⍬ | Force a check against an empty (or all-blank) value |
| ⍞ | Literal as-is *value* specification |
| ★ | Force the use of the **Like** operator;<br>or if no field name is included, **Like**-encode the right argument |
| ⍟ | Prevent the use of the **Like** operator |
| ⍎ | Use the *field* name as an un-quoted computation expression |
| ÷ | Pass a nested *value* to Math to generate a computed comparison formula. |
| ⎕ | Commonly used when the *field* name is part of a nested vector indicating that the Math utility should be called to generate a computed *field* expression. |

## Combining Conditions

And **finally** (aren't you glad?), one last major syntax that you'll use a lot… You can specify multiple *field* names and *values* as arguments to perform multiple tests at one time. You may specify more than one *field* name in the left argument (along with any of the fancy options above on each one), along with a matching-length (nested) vector of *values* in the right argument to be paired (as "Each" would have done) with each of the names. Do not specify "Each" ("¨") yourself, though, because *Is* will automatically combine all the tests you requested together into a single phrase with "**And**" between each one, so that all the tests will have to match at once. So you might say:

```
      'Name' 'Pay' 'State' Is 'Davin' 100000 'TX'
Name = 'Davin' And Pay = 100000 And State = 'TX'
```

And of course, you can get as complicated as you want specifying many different options (that will never happen in real code) to produce crazy **Where** clauses.

```
      'Name' '↑State' '~Pay<' 'Manager≠' '⍎Left(Desc,10)⊛' 'Hours→' Is
        ('Davin' 'Zz⋆') 'tx' 99999 ('Bob' 'Fred') 'Exactly⋆it' (10 20)
(Name = 'Davin' Or Name Like 'Zz%') And Upper(State) = 'TX' And
   Pay >= 99999 And Manager Not In ('Bob', 'Fred') And Left(Desc,10) =
   'Exactly⋆it' And Hours Between 10 And 20
```

Of course, nobody needs to use all this stuff, and only occasionally will anyone need to use any of the really fancy stuff at all. But isn't it nice to know that on the few occasions that something special is required that you can just plug in an APL symbol and have it construct the right statement (with the right syntax) for you?

## Special "Or" Conditions

SQL conditions are usually built solely out of "**And**"ed phrases, and the construction of *Is* obviously reflects this with its automatic "**And**"ing of multiple terms. However, there are occasions where "**Or**" conditions need to be specified, and *Is* can help with those, too.

In the most trivial case, when a single *field* may be one of several *values*, *Is* just works as described above. Give it several *values* along with a single *field* name and it will generate the correct SQL expression. If all these are single *values* (without wildcards) and are "equals" or "not equals" comparisons, then *Is* will generate an "**In**" or "**Not In**" clause for SQL as usual. If the required conditions prevent this use, then *Is* will create a composite condition with several checks "**Or**"ed together and enclosed in parentheses.

Some situations, though, require for two conditions to be "**Or**"ed together that are based on different fields or comparisons. For this, you may just call *Is* twice and join the results together using *Or* (see "*Or*" below). For example:

```
      ('Manager' Is 1) Or 'Pay>' Is 100000
Manager = 1 Or Pay > 100000
```

Often such composite conditions will need to be used with others and thus will need to be enclosed in parentheses. *Or* has a special feature that will handle this for you, too, so you may wish to write the code like this before combining it with other terms:

```
      '()' Or ('Manager' Is 1) ('Pay>' Is 100000)
(Manager = 1 Or Pay > 100000)
```

But of course, APL can let you call *Is* twice quite easily by using "Each" ("¨") with the pairs of arguments, making this expression even simpler. You don't normally use Each with *Is* because it can generate "**And**"ed conditions for you automatically, but in this case it is useful because you want to create "**Or**"ed conditions, thusly:

```
      '()' Or 'Manager' 'Pay>' Is¨1 100000
(Manager = 1 Or Pay > 100000)
```

More rarely, you may encounter a need for an even more complicated composite condition. You may need to ask for alternative "**Or**" conditions, each of which is composed of a series of "**And**"ed conditions. This may be accomplished using the above method by passing extra-nested arguments to *Is*:

```
      '()' Or 'Manager' ('Pay≥' 'Pay<') Is¨1 (100000 200000)
(Manager = 1 Or Pay >= 100000 And Pay < 200000)
```

Or you may use yet <u>another</u> *Is* syntax to make it even easier (especially when the requested data is not known in advance). For this feature, *Is* is invoked with a list of *field* names as the left argument but a <u>matrix</u> of data *values* as the right argument. The number of columns in this matrix must match the number of *field* names given (one column for each field) and multiple composite conditions are then generated by *Is*. Each row of the data matrix produces an "**And**"ed clause as normal for *Is*, but then an "**Or**" is inserted between row-results so that any one whole row of the data matrix can cause SQL to match the combined condition regardless of the other rows. Here's an example:

```
      'City' 'State' Is 2 2⍴'Dallas' 'TX' 'Washington' 'DC'
((City = 'Dallas' And State = 'TX') Or
   (City = 'Washington' And State = 'DC'))
```

Granted, this fancy option should seldom be needed, but it's available if you do wish to use it. Don't forget that since *Is* specifically skips any checks for empty vectors (either `''` or ⍬), it is possible to also use this to generate composite phrases with only some of the *field* names in each part of the test. That might look like this:

```
      'City' 'State' Is 2 2⍴'Dallas' '' '' 'DC'
(City = 'Dallas' Or State = 'DC')
```

# Math

*sql* ← *expression* `Math` *value(s)*

Part of the job of this toolkit is to analyze SQL field names and determine if they are reserved or restricted names, and if so, enclose them in the appropriate quotation marks for proper processing. This is handled by the companion *Q* utility function which is called automatically from companion utilities such as *Is*, *Select*, *Where*, etc. However, sometimes field names alone are not sufficient in some SQL commands and SQL functions or other *expressions* need to be invoked within the SQL command. For instance, calls to functions such as **Max( )**, **RTrim( )**, or **Left( )**, or other sorts of calculations, might be needed for some applications.

*Q* can do this quoting work on an individual field name, but is helpless when the field name is imbedded in a function call or calculation. To perform this sort of work, *Q* must instead be independently called on the field name by itself and then the rest of the calculation text must be constructed around it before passing it to the companion utility functions for use. With all the quotes, parentheses, commas, and other syntactical work that needs to be performed during this process, coding is annoyingly detailed and prone to typographical errors. This is also counter to this toolkit's goal of making the programming easier. Therefore, *Math* is provided to do most of this detailed work for you, relieving you of much of this coding drudgery and potential for mistakes.

`Math` is invoked with the text of a SQL calculation or *expression* as the left argument, except that the APL "Quad" character ("⎕") is used wherever a field name or other data *value* is to be substituted. The right argument contains the field name(s) (or other data) to be substituted in place of those Quad characters. The number of Quad characters in the left argument should match the number of *values* in the right argument, and they will be replaced in order from left to right. (If there is only one *value* argument provided, it need not be nested.) Don't forget to include all the usual syntactical SQL requirements in the *expression*, such as paired parentheses around function arguments and commas between them.

When substituting individual data values from the right argument, a simple unnested text name (as one of the data *values*) is taken to be a SQL field name and is processed with `Q` before substitution into the *expression*. This should satisfy most needs. However, if it is useful, other types of data may also be specified for substitution items, as follows:

| **Item data structure** | **To be used as** | **Formatted with** |
|---|---|---|
| Single number | Constant numeric value | `⍕` (Converted to text) |
| Simple (unnested) character vector | SQL field name | `Q` (SQL-quoted) |
| Nested ("⊂") character vector | SQL text constant | `∆Q` (Text-quoted) |
| Doubly-nested ("⊂⊂") character vector | Pre-formatted expression | (Unmodified) |

As noted, the depth at which character data is enclosed changes the way that it is processed before substitution. (Unnested text for the entire right argument is assumed to be a single-item nested vector, so the first enclosure does not change the meaning of the single value.)

**Note:** Do not fail to enclose character values an extra time if they're to be used as SQL text constants. Otherwise, the wrong kind of quotes may be generated which can be difficult to spot when reviewing by eye.

There is also a shortcut syntax available for the simplest of cases. If all that is needed is a single function call with no other operation to be performed, then the usual suffix of syntax parentheses, Quads, and any commas may be left out and will be assumed, leaving only the SQL function name in the *expression*. The call to the function will be generated by appending a comma-separated list of arguments enclosed in parentheses to the end of the supplied function name (made of letters and digits only). The number of arguments is determined by the number of *value* items supplied.

Since the purpose of this utility will often be to provide its results to companion utilities, if any special `Is` function codes are included in the *expression* then they will also appear in the output from `Math`, so any codes can be passed directly to the companion functions in this way. `Is` may itself call `Math` automatically if a special code and data structure is used to signal its use (*see also:* `Is`).

Of course, use of this utility is not at all necessary, and writing ordinary text processing code shouldn't be too difficult in many applications. But if you wish to make the SQL code portable (by using `Q`) and you don't want to deal with writing and reading complicated expressions and fixing frequent typing errors, then you should find `Math` to be very helpful.

## Examples

```
      'RTrim' Math 'field'
RTrim(field)
      'RTrim' Math 'table.field'
RTrim("table".field)
```

```
      'RTrim' Math 'funny%field'
RTrim("funny%field")
      'Left' Math 'field' 10
Left(field,10)
      'Left(⎕,⎕)' Math 'field' 10
Left(field,10)
      'Concat' Math 'city' 'state'
Concat(city,state)
      'Concat(⎕,⎕)' Math 'city' 'state'
Concat(city,state)
      'Concat(Concat(⎕,⎕),⎕)' Math 'city' 'state' 'zip'
Concat(Concat(city,state),zip)
      'Concat(Concat(RTrim(⎕),⎕),RTrim(⎕))' Math 'city' 'state' 'zip'
Concat(Concat(RTrim(city),state),RTrim(zip))
      'Left(⎕,Length(⎕)-1)' Math 'field' 'field'
Left(field,Length(field)-1)
      'Concat(Concat(⎕,⎕),⎕)' Math 'city' (⊂', ') 'state'
Concat(Concat(city,', '),state)
      'IfNull(⎕,⎕)' Math (⊂⊂'Null()') (⊂'-n/a-')
IfNull(Null(),'-n/a-')
```

# Null

*cleandata* ← *default* `Null` *rawdata*

This is a utility function for pre-processing data that is returned from SQL Select commands. Most databases are prone to return Null values for data that does not have any real value. Such Null values may be expressed in Dyalog APL as the special value `⎕NULL`. These Null values usually make APL data processing code much more complicated when it has to take them into account. Most of the time, a simple value will suffice to replace these Nulls for processing purposes. For instance, a numeric Null could be treated as a `0` or a character Null could be treated as `''`, and produce the desired results from the processing program. This function allows such substitutions to be easily done on the returned database data so that your APL code can then handle it much more simply.

The right argument *rawdata* is the data as it is typically returned from the database, potentially containing Null values. This is usually a nested array of values, but may also be an unnested singleton.

The left argument *default* (which is required) is the unnested value to substitute into the data in place of each Null value found.

The result is the same shape as the input data, but its depth or data representation may have been changed due to the replacement of Null values.

*Note:* This routine may also be used on other data structures that may contain `⎕NULL` values, such as those being returned from external programs.

## Examples

("Num2" and "Char2" are being returned as Null values in these examples.)

```
      0 Null ADO.Get sGet ('Num1' 'Num2') 'Table1'
111 0
      ]DISPLAY '' Null ADO.Get Get ('Char1' 'Char2') 'Table1'
.→---------.
↓.→----..⊖.|
||Davin|| ||
|'-----''-'|
'∊---------'
      x←ADO.Get Get ('Num2' 'Char2') 'Table1'
      x[;1]←0 Null x[;1]
      x[;2]←'' Null x[;2]
      ]DISPLAY x
.→-----.
↓    .⊖.|
| 0  | ||
|    '-'|
'∊-----'
```

## Nulls

*cleandata* ← *defaults* `Nulls` *rawdata*

This is a cover function for the `Null` utility function. `Null` may be used on simple values or on arrays of values, but it has the disadvantage that all the Nulls found must be replaced by a single type of Null value *default*. In many applications data is organized by columns and each column might need to use a different *default* value for any Nulls found in that column. Therefore, the `Nulls` cover function was written to handle that one common, specific case so that applications can be more readable.

The right argument *rawdata* should be a matrix of data as it is typically returned from the database, potentially containing Null values.

The left argument *defaults* (which is required) is a vector of (often nested) values to substitute into the data. There should be one item in this vector for each column in *rawdata*. The `Null` utility is then called on a column-by-column basis, and Null values in each column are replaced by the *default* value from the corresponding item of the left argument.

The result is the same shape as the input data, but its depth or data representation may have been changed due to the replacement of Null values.

*Note:* This routine may also be used on other data structures that may contain `⎕NULL` values, such as those being returned from external programs.

### Example

```
      ]DISPLAY 0 '' Null ADO.Get Get ('Num2' 'Char2') 'Table1'
.→-----.
↓    .⊖.|
| 0  | ||
|    '-'|
'∊-----'
```

## Or

*sql* ← *[expression(s)]* `Or` *expression(s)*

This is a handy little function to join zero or more text expressions together with the word "Or" inserted between them. Either argument may be an empty text vector, a simple text vector, or a nested vector of zero or more text vectors. All such vectors, with empties removed, will be joined together with the **Or** conjunction. This is especially useful when the expressions to be joined together are not known in advance. This function is also used internally by *`Is`* and its parents (*`Where`*, *`Get`*, etc.). It can also be used to combine *`Is`* restrictions together if you have an unusual restriction, but you may need to enclose such expressions in parentheses to have them take precedence over any **And**-ed expressions. (*`Is`* can also do this internally with some very fancy arguments.)

*Note:* Only use plain text in these expressions. Name-value pairs intended for use with *`Is`*/*`Where`*/etc. will not be processed properly. Name-value pairs that have already been processed by *`Is`* are now in plain text form and can be used with this function.

There is an additional utility feature available in this function. When joining SQL phrases together, it is often necessary to also enclose the result in parentheses, but only if the result is non-empty (to prevent SQL errors). If any *expression* term provided in the arguments is exactly the text value `'()'`, then any non-empty result of this function will be enclosed in parentheses.

### Examples

```
      'foo = 4' Or 'goo = 7' 'hoo = ''bye'''
foo = 4 Or goo = 7 Or hoo = 'bye'
      '()' Or 'goo = 7' 'hoo = ''bye'''
(goo = 7 Or hoo = 'bye')
```

# `OrderBy`

*sql* `←` `OrderBy` *field(s)*

Provide zero or more *field* names to be used as sorting criteria for the SQL command. SQL will sort the selected records by the *fields* in their listed order before returning the results. Normally, each field is sorted in ascending order, but a descending order could be specified instead by including the APL "Grade Down" symbol ("⍒") as part of the *field* name. An APL "Grade Up" symbol ("⍋") could also be used for ascending sorting if desired, but that is optional since it is the default direction.

When calling `Select` with a variable containing specific *field* names to return, it is often convenient to use the same variable (or portion thereof) to supply to `OrderBy` so that the names need not be repeated.

All field and table names used in these functions are sent through `Q` for quoting if they are reserved words or contain improper characters.

## Examples

```
     OrderBy 'LastName' 'FirstName'
Order By LastName, FirstName
     OrderBy '⍒Age'
Order By Age Desc
```

## Q

*sql* ← *[options]* Q *name(s)*

This is a utility function that most of the main routines call internally. It only needs to be called directly if SQL *names* (of fields, tables, etc.) are to be formatted outside of those main routines. It can also be used if you're building your own SQL specialty commands from scratch to maintain portability. Its purpose is to check the *names* provided and determine if they either (a) are reserved words, or (b) contain characters that are otherwise illegal for use in SQL names. If either of those conditions exist, then the *names* are enclosed in quotation marks so that SQL will recognize them as non-standard *names*. *Names* may be a nested array and the result is returned in the same shape.

This function provides one of the major points of flexibility of this toolkit – the ability to port code from one SQL database system to another that has different rules of syntax. `Q` will perform its checks for reserved words and illegal characters based on information found in the `Config` namespace variable (see below), which is to be copied, modified or defined to represent the database currently in use. It also uses that same variable to tell what kind of "quotation" marks to use around such *names*. (For instance, Microsoft's old Access database system uses square brackets "[]" to surround invalid *names*.)

If a name contains the APL "Execute" symbol ("⍎"), then that is a signal to suppress the quoting of names. This allows the selective insertion of calculations where names would normally be expected (e.g. `'⍎PERCENT_COMPLETE/100'`). Or, such a calculation can be nested an extra level, which also suppresses any quoting.

If a *name* to be quoted is simply "⋆", or ends in "`(⋆)`", then that *name* is never quoted automatically. This allows the use of "`Select ⋆` …" or "`Select ⋆.⋆` …" or "`Select Count(⋆)` …" or similar phrasing, without having to specify that the field *name* term is not to be quoted. This should account for the most-commonly used special *names* that are to be left un-quoted, so the special "Execute" symbol need not be used with these.

An optional left argument may be provided of a period ("`.`") to signal `Q` that the *names* provided may be SQL compound names. Compound names are often used to distinguish between the same *name* being referred to in more than one table, such as "Departments.Name" versus "Employees.Name". Since a period is not a valid *name* character, `Q` would normally enclose these in quotes. But if a left argument of `'.'` is provided (which is used automatically by these companion functions) then the *names* will each be divided at the "." character and each portion will have a separate determination to see if it needs to be quoted by itself.

## Examples

```
      Q 'Union' ⍝ In SQL Server
"Union"
      Q 'Union' ⍝ In Access
[Union]
      '.' Q 'Logic.From'
Logic."From"
```

## See

*prettysql* ← `See` *sql*

This is a utility function meant solely to make life easier for the programmer. All it does it take the incoming SQL command (usually in a single line) and reformat it to wrap long lines (at `⎕PW`) and insert line breaks and indentations at appropriate places so that it's more easily read by human eyes.

While some SQL databases may accept the command in this form, it is not recommended to automatically reformat the commands before passing them to SQL unless you first examine the result by eye. This is because the reformatting is not always perfect and it may introduce unintentional changes (such as breaking a line in the middle of a quoted string) that will cause the SQL command to not operate properly.

Just use it to help you read the long SQL commands you generate and you'll find that it makes reading SQL commands much more pleasant.

*Note:* Future improvements are needed and likely to be forthcoming.

### Examples

```
     See Get '⋆' 'Students' ('State' Is 'TX')
Select ⋆
    From Students
    Where
        State = 'TX'
```

## Select

*sql* ← *[options]* `Select` *field(s)*

This function generates the **Select** clause for a SQL command. The right argument is the nested list of *field* names (or a single unnested *field* name) to be selected. The names will be processed through `Q` to quote anything reserved or otherwise non-standard. If a particular *field* name is a formula instead of a name, include an APL "Execute" symbol ("`⍎`") as part of the text and it will be used exactly as-is (but without the "`⍎`") without any automatic quoting. Alternatively, *field* names nested an extra level deeper than aliased names are also not quoted with `Q`. Compound names are permitted (see `Q`, above), and neither "`⋆`" (as a *field* name) nor any *field* name ending in "`(⋆)`" is ever quoted automatically.

It may sound like overkill to provide a nested list of *fields* all the time, but it can be quite beneficial in a number of circumstances. For instance, if you name some of your *fields* in a pattern, such as numbering them or using the names of the months or days of the week, then these names might be easier to construct (using "`¨`", for instance) than to type in one at a time. Or, you may have a list of *fields* that you need to modify on occasion, adding or removing *fields* to retrieve based on decisions that the code needs to make at run-time, including the possibility that the user may have some say in what *fields* should be retrieved. And of course, being able to automatically quote problem names and separate each name with a comma in the appropriate places without typographical errors is also handy.

Plus, once you have the *field* names listed in a variable, you can make use of that variable further on. For instance, you may often find yourself sorting or grouping by the first few names in the list, so you can use "Take" ("↑") on the same list of names to pass some of them again to `OrderBy` or `GroupBy` without having to retype them. And once you've fetched your data, you can use the *field* name list as an argument to `Split`, or to look up which columns of your resulting data matrix contain items of interest without depending on fixed subscripts which may change over time as your code changes.

*Fieldname* aliases are also permitted in one of three ways:
1) Specify the *field* name as a text vector of the form *alias←fieldname*;
2) Nest the *field* name to contain two names within a single item, the *field* followed by the *alias*;
3) Provide the *field(s)* as a two-column nested matrix with each *alias* (or a `''`) in the second column.

Computed *field* expressions (those specified with a "⍎" symbol) may also be conveniently constructed using the `Math` utility and then passed to `Select`. On rare occasions, these may need to be specified as a special (and complicated) data structure rather than calling `Math` directly. Should that be required, please refer to the function comments for a description of how to construct such an argument.

*Options* are zero or more (nested, if more than one) "words" to be inserted into the **Select** clause, after the word "**Select**" but before the *field* names begin. This allows the use of SQL keywords such as "**Distinct**" or "**Top** *n*" to be included in the command. In addition to "prefix words", if one of those *option* words is a text "Execute" symbol ("⍎"), then this suppresses calling the `Q` subroutine for <u>any</u> of the *fields* in the right argument, as if the "Execute" symbol ("⍎") were specified in each *field* name. (The "⍎" symbol itself is not included in the resulting SQL command.)

## Examples

```
     Select '⋆'
Select ⋆
     'Distinct' Select 'T1.F1' 'T2.F2' 'T3.F3'
Select Distinct T1.F1, T2.F2, T3.F3
     Select 'me←MyFullName'
Select MyFullName As me
     'Distinct' Select 'Item'
Select Distinct Item
     Select 'Used-100'
Select "Used-100"
     Select '⍎Used-100'
Select Used-100
     '⍎' Select 'Used-100' 'Left+100'
Select Used-100, Left+100
     Select '⍎RTrim(⎕)' Math 'MyAddress'
Select RTrim(MyAddress)
```

## Set

*sql* ← *fields* `Set` *newvalues*

*sql* ← `Set` `(`*field newvalue*`)` `[(`*field newvalue*`)]`...

*sql* ← `Set` *fields*`,[1.5]`*newvalues*

This function will generate a **Set** clause for a SQL **Update** statement. It can be given zero or more pairs of *field* names and *newvalues* to assign to them. Three syntax forms are supported for convenience: separate lists of *field* names and *newvalues* on the left and right; a nested vector of *field*-*newvalue* pairs; or a two-column nested matrix of *field*-*newvalue* pairs.

In addition to the usual text and numeric values, three special values may also be specified:

| | | |
|---|---|---|
| `⎕NULL` | A special null object | Set *field* to a SQL **Null** value |
| `⎕UCS 0` | The ASCII NUL character | Set *field* to a SQL **Null** value |
| `⎕UCS 127` | The ASCII DEL character | Set *field* to its pre-defined SQL **Default** value |

If you need to specify a formula (e.g. "CURDATE()") instead of a *newvalue*, supply the text of the formula, but enclose it (again) to nest it an extra level deeper than usual. If the formula requires *field* names to be imbedded within it, enclose it in the same way but supply a nested vector of items instead of a nested scalar to call the `Math` utility internally. The first item of the nested vector becomes the left argument to `Math` and the remainder becomes its right argument.

All field and table names used in these functions are sent through `Q` for quoting if they are reserved words or contain improper characters.

*Note:* Unlike `Is`, `Set` does not ignore empty values or values composed entirely of spaces. Empty values can be explicitly set without any special concerns.

### Examples

```
      'Name' Set 'Davin'
Set Name = 'Davin'
      'Name' 'Age' 'DateDied' Set 'Davin' 99 ⎕NULL
Set Name = 'Davin', Age = 99, DateDied = Null
      Set ('Name' 'Davin') ('Age' 99) ('DateDied' ⎕NULL)
Set Name = 'Davin', Age = 99, DateDied = Null
      Set 3 2⍴'Name' 'Davin' 'Age' 99 'DateDied' ⎕NULL
Set Name = 'Davin', Age = 99, DateDied = Null
      'Started' Set ⊂'CURDATE()'
Set Started = CURDATE()
      'Initials' Set 'Left(⎕,2)' 'ShortName'
Set Initials = Left(ShortName,2)
```

# Split

*`Split` namedarray*

*names `Split` array*

*namesandtypes `Split` array*

`Split` is a data processing function that can take a nested array of data that you've retrieved from your database and assign it to variables in your workspace for more comfortable processing.

The *namedarray* argument is a nested matrix of database records, but with the variable *names* added as the top row of the matrix. Since SQL field *names* are sometimes placed at the tops of columns of a data array, this form is quite suitable for using those field *names* as APL variable names (with minor automatic conversions as necessary). Each column of the array is assigned into a (nested) vector of data items named after the field *names* at the top of the column.

If the field *names* are not already part of the data, or you wish to use different *names* for the variables, you may instead specify the variable *names* to use as the left argument to `Split` and leave only data records in the right argument. In this case, the *array* may be either a matrix or a vector. If it's a matrix, then the data is processed as above, with each variable holding a (nested) vector of data items. If the *array* argument is itself a nested vector, then you're assumed to be splitting apart only a single database record and so the variables will all get simple (unnested) values.

When there are no data records returned from a database call, this function will produce variables containing empty vectors. Often this is detected in the application by checking the rho of the database result or the rho of one of the output variables and a separate path is taken to handle the empty result as a special case. Or the results may simply be processed in the normal fashion if the code produces accurate results whether the *array* is empty or not. Occasionally, though, specific output variables are processed in such a way that the data type of the variable is critical to proper operation (e.g. a numeric calculation). Since empty data *arrays* do not preserve the proper data types on a column-by-column basis, and therefore the empty variables that `Split` creates will all have the same data type rather than each having its own correct data type.

To deal with this problem when it is important to the application, `Split` can accept an additional argument to indicate which columns should be treated as numbers and which should be treated as characters. The additional argument can only be used when it is called dyadically, as in the third syntax choice above. To use this feature, pass the left argument as a matrix rather than a vector. It should then contain two rows, where `[1;]` contains the field/variable *names* as usual and `[2;]` should contain data *type* indicators. *Type* indicators may be the numeric field type codes used by ADO (e.g. 129=Character, 3=Integer, …). (If using the separate *`ADOGet`* utility, for example, it can easily produce such *type* column headers by using the special "**??Select**…" syntax.) Alternatively, *type* indicators may be specified by using `0` for numeric columns and `''` for text columns, for easier use when the *types* are entered manually.

## Examples

```
     Split ADO.Get '?',Get '★' 'Classes'
     'name' 'addr' 'phone' Split ADO.Get Get
       ('FullName' 'Address' 'Phone#') 'Members' (⊂'Club' club)
     x←ADO.Get '??Select Name, Address, Phone From Customers'
       ⋄ x[1 2;] Split 2 0↓x
     ('name' 'addr' 'phone',[.5] '' '' 0) Split ADO.Get Get
       ('FullName' 'Address' 'Phone#') 'Members' (⊂'Club' club)
```

# Upd

*sql* ← *Upd* *table setvalues restrictions*

This is just a simple cover function to make it easier to update records in a table. The *table* name is passed to the `Update` function, the *setvalues* parameter is passed monadically to the `Set` function (if nested), and the *restriction* is passed to the `Where` function. The results are catenated together into the final SQL statement result. The terms may be in any form acceptable to those subroutines.

## Examples

```
     Upd 'Students' (('First' 'Davin') ('Last' 'Church')) ('Id' Is 3)
Update Students Set "First" = 'Davin', "Last" = 'Church' Where Id = 3
     Upd 'Students' (2 2⍴'First' 'Davin' 'Last' 'Church') 'Id = 3'
Update Students Set "First" = 'Davin', "Last" = 'Church' Where Id = 3
     Upd 'Students' ('First' 'Last' Set 'Davin' 'Church') ('Id' Is 3)
Update Students Set "First" = 'Davin', "Last" = 'Church' Where Id = 3
```

# Update

*sql* ← *Update* *tablename*

This generates the beginning of a SQL **Update** command. Provide a single *tablename* as the argument.

All field and table names used in these functions are sent through *Q* for quoting if they are reserved words or contain improper characters.

## Examples

```
     Update 'Students'
Update Students
```

## Values

*sql* ← `Values` *data*

Used in the SQL **Insert** command, this function provides one or more sets of *data* values to be added to a table. (Not all databases accept multiple **Values** statements on a single **Insert** command.) A nested vector in *data* provides the values for a single record to be added. A nested matrix in *data* provides multiple records at one time (one row per record), for databases that support that multi-record syntax. The items in this vector (or columns in a matrix) must be in the same order as the *field* names you provided to the `Insert` function, or in the table-default order if you provided no explicit *field* names.

In addition to the usual text and numeric values, three special values may also be specified:

| | | |
|---|---|---|
| `⎕NULL` | A special null object | Set *field* to a SQL **Null** value |
| `⎕UCS 0` | The ASCII NUL character | Set *field* to a SQL **Null** value |
| `⎕UCS 127` | The ASCII DEL character | Set *field* to its pre-defined SQL **Default** value |

If you need to specify a formula (e.g. "CURDATE()") instead of a *data* item, supply the text of the formula, but enclose it (again) to nest it an extra level deeper than usual.

### Examples

```
     Values 'Davin' 'Church' 'M'
Values ('Davin', 'Church', 'M')
     Values 2 3⍴'Davin' 'Church' 'M','Genny' 'White' 'F'
Values ('Davin', 'Church', 'M'), ('Genny', 'White', 'F')
     Values 'Davin' ⎕NULL (⊂'CURDATE()')
Values ('Davin', Null, CURDATE())
```

## Where

*sql* ← `Where` *restriction(s)*

The SQL **Where** clause is used in several different SQL commands (most commonly **Select**) to restrict the database records which are to be acted upon. The *restriction* argument is flexible in what it can accept – you can provide it any of the following structures:

- An empty vector, in which case no **Where** clause is generated at all.

- A simple text vector, containing an entire **Where** clause (except without the **Where** keyword) to be used directly in the SQL command. This can be a single *restriction*, or it can be multiple *restrictions* already joined with **Ands** and **Ors** as desired. And since this is the syntax generated by the `Is` function, you can therefore also call the `Is` function directly and pass its result on to `Where`.

- A nested vector of text vectors of *restriction* operations, which will be joined together with **Ands** by `Where` (using the `And` function) and then used as the composite *restriction* clause. Any of these could have been generated by the `Is` function if desired.

- A nested vector of text vectors of *restriction* operations, as above, any one or more of which may be specified as a further-nested two-item vector *name-value* pair. Any such nested pair will be passed to `Is` for formatting before being **Anded** into the composite *restriction* clause as above.

- A two-column nested matrix of *name-value* pairs, each pair (row) of which will be passed to `Is` for formatting before being **Anded** together into the composite *restriction* clause as above.

All field and table names used in these functions are sent through `Q` for quoting if they are reserved words or contain improper characters.

## Examples

```
     Where 'Name = ''Davin'' And Age > 18 And State In
        (''TX'', ''OK'', ''AR'', ''LA'')'
Where Name = 'Davin' And Age > 18 And State In ('TX', 'OK', 'AR', 'LA')
     Where 'Name = ''Davin''' 'Age > 18' 'State In
        (''TX'', ''OK'', ''AR'', ''LA'')'
Where Name = 'Davin' And Age > 18 And State In ('TX', 'OK', 'AR', 'LA')
     Where ('Name' 'Davin') ('Age>' 18) ('State' ('TX' 'OK' 'AR' 'LA'))
Where Name = 'Davin' And Age > 18 And State In ('TX', 'OK', 'AR', 'LA')
     Where 'Name' 'Age>' 'State' sIs 'Davin' 18 ('TX' 'OK' 'AR' 'LA')
Where Name = 'Davin' And Age > 18 And State In ('TX', 'OK', 'AR', 'LA')
     Where 'Name' 'Age>' 'State',[1.5] 'Davin' 18 ('TX' 'OK' 'AR' 'LA')
Where Name = 'Davin' And Age > 18 And State In ('TX', 'OK', 'AR', 'LA')
```

# Config

This is a namespace variable used internally by nearly all the functions in this toolkit, but it is not usually accessed directly by programs. It may be used to globally configure any particulars about the database in use, such as how and when to quote names with odd characters and what keywords are reserved by your particular SQL implementation. It is a nested vector of items that you may modify as desired before running these functions. Note that none of the contents are error-checked, so modify them carefully if needed. A default configuration is included in the workspace, and alternate variables might also be available for your particular database – just assign them to this variable.

The structure of this vector is as follows:

[1] This description (for self-documentation only)
[2] Valid characters that may be used in names (in addition to alphanumerics)
[3] Quotation marks used to surround reserved/invalid names (e.g. `'""'` or `'[]'`)
[4] Quotation marks used to surround text values (e.g. `''''''`)
[5] External wildcard character used to request SQL **Like** expressions (e.g. `'⋆'`)
[6] (reserved)
[7] (reserved)
[8] (reserved)
[9] List (nested vector) of SQL reserved words for this version of SQL

## `From_Defaults`

This is a namespace variable used by the *`From`* function to remember how tables are typically joined together. Do not directly (manually) change this variable. Use *`From`* dyadically to both read and modify it.

# Subroutines

The following general-purpose subroutines are also required for various portions of this toolkit. They are included in this namespace for its own use but are normally distributed in various other "family" toolkits elsewhere for more general use. Since they are often quite helpful, feel free to use them directly in your application as well if desired.

| | |
|---|---|
| `∆Q` | Enclose string(s) in quotes (doubling internal quotes as needed) or other symbols. |
| `∆clean` | Clean up executable APL code for analysis by blanking out strings and comments. *(Only needed for `See`.)* |
| `∆csv` | Change an array of values into simple comma-separated, potentially-quoted text string(s). |
| `∆cut` | Cut a vector into nested pieces at a delimiter, without empties. |
| `∆cuts` | Cut a vector into nested pieces at a delimiter, including empties. |
| `∆dlt` | **D**eletes **L**eading and **T**railing spaces (or zeros/prototypes/`∆dlt`-prototypes) from the rows of any array. |
| `∆dmu` | **D**eletes leading, trailing, and **MU**ltiple imbedded spaces (or zeros/prototypes/`∆dmu`-prototypes) from the rows of any array. *(Only needed for `See`.)* |
| `∆dtr` | **D**eletes all **TR**ailing spaces (or zeros/prototypes/`∆dtr`-prototypes) from the rows of any array. |
| `∆sew` | Join together a list (or array) of values into text string(s) separated by delimiters. |
| `∆uc` | Convert lower case text to upper case. (Text may be any rank or depth.) |

# Conclusion

If you use this toolkit to build all the SQL commands in your application then it'll be faster and easier to write, much simpler to read, more flexible, with shorter code, fewer typos, and be much more portable to other database or interface implementations. I hope you have fun with it!