Overview

Use this wiki plugin, introduced in Tiki3, to produce formatted reports from ADOdb SQL databases.

The plugin content is parsed as a description of how to run and display a database query. If only an SQL statement (and parameters) is given, a default view of the returned data will be prepared. A report has a fairly restricted format, consisting of zero or more levels of grouping (supported by GROUP BY or ORDER BY clauses in the query's SQL) with a table generated per group. The table can have headers and footers, and links can be inserted which may take you to other web pages.

Example:

```
{DBREPORT(dsn="mysql://user:password@server/database")} SQL { SELECT * FROM Members ORDER BY City, Postcode } GROUP [City] "Members in [City]" :heading{text-size:150%;} GROUP [Postcode] "Within postcode [Postcode]" :heading{text-size:120%;} TABLE HEADER :heading CELL :{text-align:center;} "First Names" CELL :{text-align:center;} "Surname" ROWS :even :odd <"http://site.com/member.php?id=[MemberID]"> CELL "[First Name] [Middle Name]" CELL [Surname] {DBREPORT}
```

Report Definition

A report definition is written in a simple "micro-language" that declares zero or more levels of grouping (supported by ORDER BY clauses in the query's SQL) and generates a table for each group. The tables may have headers and footers, and links can be generated for most visible elements will can take you to other web pages. If only the SQL (and parameters) are provided, a default view of the returned data will be prepared.

The report generator language is designed to be very human-friendly, and tries to adhere to the 'natural language' approach of SQL which tries to avoid the use of constructs such as recursive brackets or semicolons to finish statements. Indenting and line breaks are not important, so definitions can be formatted for readability.

Much of the definition can simply be omitted, and the report generator will fill in what it can. If the TABLE definition is left out, for example, then the report generator will internally generate one which has a column for each field returned by the query, and even has a header row.

A general outline of a report definition looks like this:

```
SQL { ... sql query ... } PARAM (zero or more SQL parameters) GROUP (zero or more group breaks) TABLE (zero or one data table) HEADER (zero or more header rows) CELL (one or more header cells) ROW (one or more data rows) CELL (one or more row cells) FOOTER (zero or more footer rows) CELL (one or more footer cells)
```

Parameters

Query an ADOdb database and display results (does not work with PDO)
Introduced in Tiki 3.
Go to the source code
Preferences required: wikiplugin_dbreport

Parameters	Accepted Values	Description	Default	Since
(body of plugin)		report definition		
`audit`		Create a log entry containing information about the SQL call.	0	21.2
`audit_csv`	text	If set, a CSV file will be created or appended with information about the SQL call performed.		21.2
`dsn`	url	A full DSN (Data Source Name) connection string. Example: `mysql://user:pass@server/database`		3.0
`db`	text	The name of a DSN connection defined by the Wiki administrator.		3.0
`debug`	(blank) 1 0	Display the parsed report definition (not displayed by default)		3.0
`wiki`	(blank) 1 0	Parse wiki syntax within the report (not parsed by default)		3.0

Either the `dsn` or the `db` parameter should be specified, not both, since they represent alternate ways of determining how to connect to the database.

This example uses a Wiki DSN to connect, and debugs the report definition.

```
{DBREPORT(db="database" debug="true")} ... report definition ... {DBREPORT}
```

This example uses a specific DSN to connect, and is probably what you will use most often.

```
{DBREPORT(dsn="mysql://user:password@server/database")} ... report definition ... {DBREPORT}
```

Keywords

Eight keywords are defined; **SQL**, **PARAM**, **GROUP**, **TABLE**, **HEADER**, **ROW**, **FOOTER**, **CELL**. They are case insensitive, but uppercase is encouraged for readability. Most commands take a sequence of parameters, terminated by the next keyword. Parameters come in five forms: Fields, Variables, Text, Styles, and Links

- **Fields** are enclosed in **[square brackets]**, and denote database field names.
- **Variables** begin with a **$dollar** sign and must be followed by whitespace. They are used to pass wiki variables to the query parameters. They cannot be used to compose strings. Variables may also be enclosed in **[$square brackets]** (since they are very similar to fields), in which case the dollar sign must be inside the square brackets. Variables used within text must be inside square brackets in order to separate them from the rest of the text.
- **Text** is enclosed in **"double quotes"**, and denote text strings to be displayed in the report. They can include fields and variables (which will insert the value of the field into the text at that point) and be followed by style names. (to set the style of the text)
- **Styles** begin with a **:colon** and can specify the CSS 'class' and CSS 'style' parameters which are applied to HTML elements generated by the report. They can have both **:class{css:style;}** parts, or omit either. They denote the visual style which should be applied to the keyword or text which immediately precedes them.
- **Links** are enclosed in , and may contain Strings, Variables, and Fields which are used to compose the link's URI, and styles which are used to format the link. Just like styles, Links are applied to the text or keyword directly before them.

SQL Queries and Parameters

The query to be run by the database engine is defined by the SQL keyword, like so:

```
SQL { SELECT * FROM table WHERE field1=? AND field2=? AND field3=? } PARAM "value1" PARAM "id" $value2 PARAM [$value3]
```

Parameters bind values to the '?' slots in a query.

- The first PARAM associates the literal string "value1" to the first query parameter.
- The second PARAM associates a string composed of the literal "id" concatenated with the contents of the PHP variable "value2" to the second query parameter.
- The third PARAM associates the variable "value3" to the third query parameter. It just uses the square bracket form.

The order of the parameter statements determines which slot they are bound to.

Table fields **cannot** be used as SQL parameters, either directly or within strings, because no data has been obtained at the time of the binding. (The query hasn't run yet!) Attempts to use strings such as "[field1]" will bind the literal text (square brackets and all) to the parameter, which is probably not what you are after.

SQL statements are processed by the database server, and must adhere to it's syntax, which can vary. Some common differences between server engines are:

- The semicolon at the end of the SQL statement may be optional or mandatory.
- Field names (especially those with special characters like spaces) may be specified differently. For example, field names are often enclosed in square brackets in Microsoft Access, but MySQL uses backtick quotes like `this`. They are rarely case sensitive.
- SQL Subqueries are supported in most engines, but not all.
- Wildcard characters for LIKE statements may be different.

SQL Queries can be placed in **{curly brackets}** or within **"double quotes"**. Only closing brackets at the 'top level' of the SQL end the statement, so curly brackets can be put in quoted string like this:

```
SQL { SELECT "{curly}" as field1, '{brackets}' AS field2 }
```

Double quotes were the original syntax, but it was discovered that escape sequences became unwieldy; Reporting definition strings are escaped, but so are literal strings in SQL statements. This double-escaping meant a backslash '\' within an SQL literal was encoded by *four* backslashes '\\\\' in the report definition.

For example, an SQL statement that included quotes within strings like this:

```
SQL { SELECT "back~bs~~bs~slash" AS slash, "double~bs~"quote" AS quote }
```

Would have to be escaped as follows if using double quotes around the entire statement;

```
SQL " SELECT ~bs~"back~bs~~bs~~bs~~bs~slash~bs~" AS slash, ~bs~"double~bs~~bs~~bs~"quote~bs~" AS quote
"
```

Curly braces allow SQL strings to be cut-and-pasted directly into the report definition without having to escape quotes or backslashes. Double-quoted SQL is retained as a 'backup' method in case some system does actually use brackets.

Variables

Variables can come from three sources, in the following order of precedence. This order is chosen so that more 'intrinsic' variables cannot be overridden by more 'external' variables, while still allowing a useful mix.

Source	*Description*	*Example*
Global	PHP Global variables which are set by the Wiki software (or other plugins) each time a page is requested.	$user
Session	PHP Session variables persist across multiple page requests, and can be set by the wiki software, other plugins, or other PHP scripts running on a site. These variables are stored by the server.	$wysiwyg
Request	Variables can be passed in the URL as request parameters, (or just typed into the browser's location bar by the user) or by POST data from forms.	$search

Many global variables are provided by the Wiki system. Here are some useful ones:

Variable	*Description*	*Example*
$user	The name of the logged-in wiki user	"admin"
$group	The wiki group that the logged in user belongs to.	"Registered"
$page	The wiki page the report is on.	"PluginDBReport"
$tiki_p_admin	Whether the current user is a wiki administrator.	"1"

Report Groups

Grouping (also called "Section Breaks" in other reporting systems) can be done based on one or more fields, and text may be generated for the title of each grouped section. Several levels of grouping can be done, each on many fields, but this really only affects the visual appearance of the report. Here are some examples:

```
GROUP [city] [state] "City of [city] in [state]"
```

Produces one group per unique combination of city and state. The order of the fields is irrelevant, but make sure they come first. Any fields specified after the text will be interpreted as part of the text. For multi-level display, something like this:

```
GROUP [state] "State: [state]":h1 GROUP [city] "City: [city]":h2
```

Will produce two levels of grouping, showing a list of cities within each state group. Styles are applied to differentiate the levels.

```
GROUP [city]
```

Would create one level of grouping without any text heading, but will also put records from cities of the same name (but in different states) into the same group.

Data Tables

For each unique group, a table is generated that will contain at least one record. (Otherwise the group would not be created by the data) This table can be given a style.

```
TABLE :style
```

The table may also have headers and footers, which can be used to title columns or to show group totals. Fields can be used, not just text. The header will be generated using the same record as first row of the table, and the footer using the last record. The HEADER and FOOTER and ROW keyword can be followed by one or more styles, although this is really only useful for ROW in which case the styles will be used in 'round robin' order for successive database records, which is useful for visually delineating records. This example has alternating white/grey rows for the records:

```
TABLE HEADER :black_row CELL "First Field" :white_text CELL "Second Field":white_text ROW :grey_row :white_row
CELL "[field4]" CELL "[field5]" FOOTER CELL "Total" CELL "[sum_field]"
```

Multiple Header, Footer and Row 'lines' can be specified by simply adding more keywords. Cells can also span rows and columns by specifying ROWSPAN and COLSPAN keywords followed by a single number after the CELL keyword. You can also use the SPAN keyword instead of COLSPAN for rows. (And instead of ROWSPAN for Columns if they are ever implemented)

```
SQL "SELECT * FROM Categories" TABLE HEADER CELL SPAN 3 :heading "Categories" ROW CELL :heading "id:" CELL
:heading [CatID] CELL ROWSPAN 2 :heading "]" ROW CELL SPAN 2 [Category]
```

Spans work the same was as in HTML tables, (since that is how the data tables are ultimately generated) and can produce some quite unexpected results if you get the spans just a little wrong. Refer to a good HTML coding manual for details on how spans work. Remember that spanning cells move other cells 'out of the way' and do not replace them.

Styles and Formatting

Styles can be specified in two parts, which relates to the way HTML "style sheets" work. HTML elements can be given a "class", in which case their style information is found by looking in active stylesheets, and they can have "inline" style information (such as colours, borders, and margins) specified directly. Classes are the preferred method, since HTML pages are composed of hundreds (or thousands) of "tags" and having a short name attached to each is more efficient than having long strings of style information repeated over and over again, and the visual style of the page can be changed by using a different style sheet rather than re-writing all the HTML tags.

Style Syntax

Style definitions in reports always begin with a ':' colon immediately followed (with no intervening whitespace) by a name which corresponds to a CSS class, and/or a string in {curly braces} which is used as inline style information. You can have whitespace inside the braces, but it's discouraged because it makes the code harder to read, and will also appear in the HTML.

The following example shows a useful mixture of inline style information on the GROUP and TABLE, and style classes used for the ROW and CELL definitions that will generate efficient pages.

```
GROUP :{border-top:1px;border-bottom:1px;} TABLE :{margin-top:1em;margin-bottom:1em;margin-left:2em;margin-right:2em;} HEADER :heading CELL :{text-align:center} "Field 1" :{color:blue;} CELL :{text-align:center} "Field 2" :{color:red;} ROW :odd :even CELL [Field1] CELL [Field2]
```

Remember that a GROUP with no actual fields or text will still produce a 'DIV' element in the output HTML which can be used to advantage, as in the example above.

Report Keywords and HTML Elements

When using Cascading Style Sheets, it can be important to know how the various parts of the report are rendered into HTML, so that you know which "tag class" will be used for which report elements. The following table lists the translations;

Report Element	HTML Tag
GROUP	DIV
TABLE	TABLE
HEADER	TH
FOOTER	TH
ROW	TR
CELL	TD

Text	SPAN

Generally you should try to use style *classes* for ROW and CELL elements, as many of them will be generated for an average report. However, it's fine to specify *inline CSS* for the GROUP, TABLE, HEADERs and FOOTERs since relatively few of them are generated, and they tend to be the elements you want to control most with alignment, borders, and margins.

Special Classes

Nine style class names are treated specially when applied to Text, Fields and Variables. Instead of creating a text HTML element, they generate the named tag. These special classes do not apply to TABLE, GROUP, ROW, CELL, or Links keywords, since they must generate their own special tags.

Style	Generated HTML
:b	**Bold tag**
:i	*Italic tag*
:u	Underline tag
:h1	**Heading 1 tag**
:h2	**Heading 2 tag**
:h3	**Heading 3 tag**
:h4	**Heading 4 tag**
:h5	**Heading 5 tag**
:h6	**Heading 6 tag**

You can also specify inline style for any of these classes, which is especially useful for the heading tags. This example turns the text into a level 1 heading, but also controls the leading margin.

"Heading Text" :h1{margin-top:6px;}

However, this syntax does not allow you to set a class for the tag, only inline style. This is a limitation, but not a large one... if you have enough control over the stylesheet to declare heading tags with specific style classes, then you can just define new classes with the full style you want.

Computed Style Names

Style classes and inline CSS don't have to be static strings. You can use variables or database fields to compose the style names. Indeed, you can use quite complicated SQL statements to determine what style to apply to individual rows.

The following example shows a variety of styles composed from variables and fields;

```
SQL { SELECT 'h3' as style1, 'b' as style2, 'caption' as style3, 'font-weight:bold;' as style4, '2' as level } TABLE ROW CELL "URL Style" :[$url-style] CELL "Field Style 1" :[style1] CELL "Field Style 2" :[style2] CELL "Field Style 3" :[style3] CELL "Field Style 4" :{[style4]} CELL "Heading" :h[level]
```

Note that variables (from sources such as URL parameters) have to be enclosed in [square brackets] in the same way as within text strings.

Multiple Classes

If you absolutely have to, you can specify multiple classes for an element by using an 'escaped space' (a space preceded by a single backslash) between the class names, which will translate through to the HTML attribute as a normal space;

```
"Dual" :class1\ class2
```

Multiple classes are rarely used, but are legal CSS, and can be useful. They allow several 'subsets' of classes (perhaps one group that sets alignments, another group that sets colours, and a third which sets whitespace breaking) to be combined on a per-tag basis, rather than having to create style classes for every possible combination.

They could even be used with computed styles to allow very tight control over field formatting, eg:

```
CELL [text] :[align]\ [color]\ [whitespace]{text-weight:[weight];}
```

Web Links

Content in after a GROUP, HEADER, FOOTER, ROW, CELL or Text generates a hyperlink. The link can include fields and variables and even have a style.

A common case is when you want to be able to click on a record and be taken to a detail page (or editor page) for that entry. The entire URL must be provided if you want an absolute link, (ie; starting with "http://") otherwise it will be interpreted by the browser as a relative link from the location of the current page, and will be appended to the current URL path by the browser in the normal way. For example: This would take you to the relevant page on another site when the group's title text was clicked;

```
GROUP [state] [city] "[city] [state]" <"http://site.com/"[state]"/"[city]".html">
```

But the following code would take you to a relative page on the same server when the record's row was clicked.

```
TABLE ROWS :grey_row :white_row <"database/details.php?id="[field4]> CELL "[field4]" CELL "[field5]"
```

In the following example, two links are used to manage each entry, in addition to the table row link. Note that the final link would enclose the "delete" text, but not the separating " ";

```
TABLE ROWS :grey_row :white_row <"database/details.php?id="[field4]:row_link> CELL "[field4]" CELL "[field5]" CELL "\[edit\]":normal <"edit.php?id="[field4]> " " "\[delete\]":normal <"delete.php?id="[field4]>
```

Also note that the link style will be superseded by any CSS properties of the text style "normal" because of the order in which links and style are applied to text. Links wrap around text, which may have formatting.

Links may have multiple segments of text, with fields inside or outside the quotes. Multiple styles may be specified, but only the last one is applied. For example, the following Links are all equivalent:

```
<"edit.php?id="[field4] :link_style> < :link_style "edit.php?id=" [field4] > <"edit.php" :ignored_style "?id=" [field4] :link_style>
```

The first example is the recommended form, because it is most clear that the style always applies to the whole link, and not just parts of it.

URL Encoding

One subtle issue to note is the concept of URL encoding; some characters are not allowed in URLs, such as spaces. Other characters such as question marks, ampersands, plus and equals signs, and colons have special meanings. When building a URL from database fields you need to take care that the special characters are encoded appropriately.

This is most important for text fields that might contain these special characters, but not so much for numeric fields, which end up with the same encoding whichever way you do it.

Imagine you want to create links from database entries to search engines. The database table contains three fields, [Title], [Engine] and [Find]. You want to display the [Title] field in the report, and when you click on it you want to perform a search in some [Engine] for the [Find] term. Here's what the table could contain;

Title	Engine	Find
Simple Google	http://www.google.com/search?q=	Simple Search
Simple Yahoo	http://au.search.yahoo.com/search?p=	Simple Search
Complex Google	http://www.google.com/search?q=	Long & Complex search?
Complex Yahoo	http://au.search.yahoo.com/search?p=	http://www.google.com/search?q=

Note the final two entries in particular, which although weird, represent perfectly legal searches. The correct report code for this example is:

```
SQL "SELECT Title, Engine, Find FROM Searches" TABLE ROW CELL [Title] <"[Engine]"[Find]>
```

Note how the [Engine] field is inside a pair of double quotes, while the [Find] field is not. This tells the report generator that the [Engine] field is not to be encoded, to be used "as-is". On the other hand, the [Find] field is to be URLEncoded, otherwise special characters will mess up the URL. The URLs generated for these examples will be as follows:

Title	Generated Link URL

Simple Google	http://www.google.com/search?q=Simple+Search
Simple Yahoo	http://au.search.yahoo.com/search?p=Simple+Search
Complex Google	http://www.google.com/search?q=Long+%26+Complex+search%3F
Complex Yahoo	http://au.search.yahoo.com/search?p=http%3A%2F%2Fwww.google.com/search%3Fq%3D

Here is what would happen to the 'Complex Google' link for all the other (wrong) ways of doing it:

Bad Link Code	**Generated Link URL**	**Notes**
<"[Engine][Find]">	http://www.google.com/search?q=Long & Complex search?	
The [Find] field has not been encoded. The ampersand in the search term will truncate the "q=" parameter.		
<[Engine]"[Find]">	http%3A%2F%2Fwww.google.com/search%3Fq%3DLong & Complex search?	
The [Engine] field was encoded messing up the "http://" and "?q=" parts, while not encoding the [Search] field. This is an invalid URL.		
<[Engine][Find]>	http%3A%2F%2Fwww.google.com/search%3Fq%3DLong+%26+Complex+search%3F	
Both parts are encoded, creating a valid relative URL to a page that does not exist		

The first link would mostly work, but would only end up searching google for "Long", because ampersands indicate the beginning of the next parameter. The second link is completely messed up. The final link would be treated by the browser as a 'relative link' to a page within the site, ie, it would attempt to jump to a page like "http://your.site.com/wiki/http%3A%2F%2Fwww.google.com..." because the "http://" at the beginning has been encoded. This page would almost certainly not exist, and you would get a "Not Found" error.

So in conclusion, put fields into quoted strings when you want the field used 'as is' with no encoding, such as for complete URLs stored in a database. But in cases where you compose a URL that contains parameters, leave the parameter fields outside of the quotation marks so they are encoded.

Useful Link Styles

It can sometimes be distracting to have every link in a table underlined, so don't forget about the "text-decoration" CSS property which can be set to "none".

When creating clickable cells and rows, it's often useful to choose a CSS class that incorporates a 'hover' style and which changes the mouse pointer so that the users have some visual feedback. (Otherwise they won't necessarily know that the row is clickable unless they actually try it.) TikiWiki's 'odd' and 'even' row classes (usually) change the color of the row the user is hovering over, but should be extended to set the mouse pointer as in the following example;

```
{DBREPORT(db="tiwi1")} SQL "SELECT * FROM tiki_preferences" TABLE HEADER :heading CELL "name" CELL "value" ROW :even{cursor:pointer;} :odd{cursor:pointer;} <"http://www.google.com/search?q="[name]> CELL [name] CELL [value] <"http://www.google.com/search?q="[name]:{display:block;text-decoration:none;}> {DBREPORT}
```

When putting links into table cells, if the link text is going to be the only content in the cell then you can use CSS to 'expand' it so that it occupies the entire cell, by setting the "display" property to "block". This has some slight advantages over making the table cell itself clickable. Namely, the cell remains a 'normal' hyperlink.

Let's change the above example so that each cell becomes clickable rather than the whole row;

```
{DBREPORT(db="tiwi1")} SQL "SELECT * FROM tiki_preferences" TABLE HEADER :heading CELL "name" CELL "value" ROW :even :odd CELL :{cursor:pointer;} <"http://www.google.com/search?q="[name]> [name] CELL [value] <"http://www.google.com/search?q="[value]:{display:block;}> {DBREPORT}
```

We format the [name] so that the entire cell becomes a link. The HTML that the report generator creates must use an 'onclick' event for the CELL elements rather than an anchor tag, (this is a shortcoming in HTML) so we need to set the style to explicitly change the mouse pointer for user feedback. The text stays black, and hovering over the cell does not show the link destination in the status bar of the browser.

The [value] cell instead wraps a normal hyperlink around the text content. The text is coloured blue, gets an underline. But in normal circumstances the link would be confined to just the text, which could be a problem if the text content was an empty string, but we still wanted to click on it. (This is actually quite common with tables generated out of a database) This is why we want to use CSS to 'display' the link as a 'block', which causes it to expand to fill the entire cell.

Examples

Finally, here is a complicated example that shows nearly every feature in use simultaneously, and in all their legal forms. (Unreal names are used for clarity)

```
{DBREPORT( 'dsn'=>'mysql://user:password@server/database' )} SQL { SELECT * FROM table WHERE field1=? AND field2=? } PARAM "0001" PARAM $user GROUP [field1] "Level 1 [field1]":{font-size:150%;} <"group.php?id="[field1]> GROUP [field2] [field3] "Level 2 [field2] [field3]":{font-size:120%;} TABLE :table_style HEADER :heading CELL "First Field" CELL "Second Field" ROWS :even{cursor:pointer;} :odd{cursor:pointer;} <"record.php?id="[field4]> CELL "[field4]" CELL "[field5]" FOOTER :row_style CELL "Total" :{font-weight:bold;} CELL "[sum_field]" {DBREPORT}
```

The following example is a real report in use on the author's wiki site. The current wiki user determines what records are visible. Notice that the SQL code is actually slightly longer than the display definition, and more complicated.

```
{DBREPORT(db="members")} SQL { SELECT Members.MemID, CONCAT_WS(' ', Members.FirstName, Members.Surname) AS Name, CONCAT_WS(' ', Members.Address1, Members.Address2) AS Address, Members.`Suburb or City`, Members.State, Members.PostCode, Members.`Hub Postcode`, Members.Phone, Members.Mobile, Members.Email, Members.Status, Postcodes.Hub, Members.Experience, Categories.Category, FeesPaid.PayDate FROM Members INNER JOIN Postcodes INNER JOIN ( SELECT MembersFeesRecord.MemID, Max(MembersFeesRecord.FeeID) AS FeeID, Max(MembersFeesRecord.DatePd) AS PayDate FROM MembersFeesRecord GROUP BY MembersFeesRecord.MemID ) AS FeesPaid INNER JOIN Categories ON Postcodes.Postcode = Members.`Hub Postcode` AND Members.CatID = Categories.CatID AND FeesPaid.MemID = Members.MemID WHERE Members.Status='Yes' AND Postcodes.Abbrev = ( SELECT Abbrev FROM Postcodes INNER JOIN Members ON Postcodes.Postcode = Members.`Hub Postcode` WHERE CONCAT('bsol',MemID) = ? ) ORDER BY Members.MemID; } PARAM $user TABLE HEADER CELL :heading ROWSPAN 2 "ID" CELL :heading ROWSPAN 2 "Name" CELL :heading ROWSPAN 2 "Category" CELL :heading ROWSPAN 2 "Address" CELL :heading "Phone" CELL :heading "Mobile" CELL :heading "Email" CELL :heading "Paid" HEADER CELL :heading COLSPAN 4 "Comments" ROW CELL :heading ROWSPAN 2 [MemID] CELL ROWSPAN 2 [Name] CELL ROWSPAN 2 [Category] CELL ROWSPAN 2 "[Address] [Suburb or City] [State] [PostCode]" CELL :{white-space:nowrap;} [Phone] CELL :{white-space:nowrap;} [Mobile] CELL :{white-space:nowrap;} [Email] CELL :{white-space:nowrap;} [PayDate] ROW CELL :{color:#404060;vertical-align:top;width:50%;} SPAN 4 [Experience] {DBREPORT}
```

Developing a Report

There are two major parts to developing a report;

- Writing the SQL query which retrieves the records from the database.
- Writing the report definition which displays the records.

It's usually best to develop the SQL query first, and then think about the best way of arranging it on the screen later, once you can actually see some example data.

Developing the SQL

The hardest part will probably be developing the SQL query. Most of the examples given have been simple "SELECT * FROM tablename" queries, but many real-world cases are quite complicated with multiple INNER JOINs, ON, WHERE, GROUP BY and ORDER BY clauses, concatenation of fields, and more. The SQL statement can easily be longer than the rest of the report definition.

For performance reasons, it's best to only retrieve the fields that you intend to use in the report.

While you can develop the SQL directly in the Wiki page editor, it may be better to do it with a proper database front-end, (Such as Microsoft Access, or phpMyAdmin) since it's likely to provide more help and better error messages. Once the query is working properly, just copy-and paste it into the report definition, but be careful of the following:

- The query shouldn't contain any "double quoted" strings, because they will be interpreted as the closing quotes for SQL report keyword. The SQL specification encourages the use of 'single quotes' for the purpose of string in queries, but double-quotes are allowed. If you have to use double-quotes, (say, if you are composing a text string that must contain quotes for output) then prefix them with the '\' escape character.
- Be aware of the '\' escape character, which is used to prefix any characters like quotes, and itself, which sometimes need to be treated literally rather than syntactically.
- Some query designers (most notably Microsoft Access) do not generate standard SQL, but their own special variant. For example, Microsoft Access uses [square brackets] to surround field names that contain spaces, and these need to be converted to `backticks`. (also called `backquotes`, usually the top-left button on the keyboard)

Developing the Display Definition

You do not actually have to specify a TABLE part of the report definition, in which case a simple 'default' table will be generated which has each field in it's own column, in the order that they come from the query. You can use this in combination with the **debug** parameter to the plugin to see the code for the TABLE section that the report generator creates from the data. For example, if we were to write the following Wiki code:

```
{DBREPORT(db="members" debug="true")} SQL "Select FirstName, MiddleNames, Surname FROM People"
{DBREPORT}
```

Then the report generator would, at the top of the report, give us back a report definition like this:

SQL "Select FirstName, MiddleNames, Surname FROM People" TABLE HEADER :heading CELL "FirstName" CELL "MiddleNames" CELL "Surname" ROW :even :odd CELL [FirstName] CELL [MiddleNames] CELL [Surname]

You could then copy-and-paste this definition back into the original, and use it as a starting point to re-arrange the fields, change the heading names or formatting, etc.

Concatenating Fields into Text Strings

Concatenating fields together for display can either be done in the SQL query, (Using a statement like "CONCAT(field1,field2) AS NewFieldName") or in the display part of the definition. (Using a Text string like "[Field1] [Field2]") and there are some subtleties you should be aware of, mostly to do with the whitespace between fields, and the behavior of the SQL CONCAT function, when fields have null values.

For example, consider the common case where you need to concatenate a person's name out of three seperate fields; FirstName, MiddleNames, and Surname. Many people may not have any middle names, in which case this field could be NULL. We will also consider cases where the FirstName is also NULL, for various reasons. (This is more common than you think)

Imagine we want, for each person, a line on the report that gives their full name followed by a full stop.

Now consider the following three examples:

{DBREPORT(db="members")} SQL "SELECT CONCAT(FirstName,' ',MiddleNames,' ',Surname,'.') AS Name FROM People" TABLE ROW CELL "[Name]" {DBREPORT}

This looks right, but you need to know that the SQL CONCAT statement is defined to return NULL *if any of it's parameters are NULL*. So in cases where the person has no middle name, their entire name will come out blank! Not what we want at all! Although there are cases where this behavior is preferable, such as when adding prefix or suffix strings to a field, (like a dollar sign to money fields) and we want the entire field to remain blank if the involved field is null.

{DBREPORT(db="members")} SQL "SELECT FirstName, MiddleNames, Surname FROM People" TABLE ROW CELL "[FirstName]_[MiddleNames]_[Surname]." {DBREPORT}

This is better, and we will see records for people who have NULL fields. However, we will also have some slight visual issues with extra spaces around fields that may have NULL values. Examine the following table of result fields and the concatenated string, where spaces in the output are represented with underscores:

FirstName	MiddleNames	Surname	Output Text
John	Qubert	Public	John_Qubert_Public.
Ronnie	NULL	Corbett	Ronnie__Corbett.
Ronnie	Barker	NULL	Ronnie_Barker_.
Nobody	NULL	NULL	Nobody__.

Note how the spaces (represented as underscores) persist, even when some fields are null. In many cases this won't even be noticed, because extra whitespace in HTML pages is 'parsed away' by the browser, but this can lead to some interesting visual 'quirks', such as the last entry where a space will be visible between the "Nobody" and the trailing full stop. This issue becomes much more obvious if we were to use actual underscores, or other visible characters.

To get perfect output, with no extra spaces, we instead should use the SQL CONCAT_WS function, like so:

{DBREPORT(db="members")} SQL "SELECT CONCAT_WS(' ',FirstName,MiddleNames,Surname) AS Name FROM People"

TABLE ROW CELL "[Name]." {DBREPORT}

CONCAT_WS (WS stands for "with separators") works slightly differently from CONCAT, in that the first parameter is used as the separator to use between fields, and NULL fields are skipped during the concatenation process. The text that is produced for our example data now looks like this:

FirstName	MiddleNames	Surname	Output Text
John	Qubert	Public	John_Qubert_Public.
Ronnie	NULL	Corbett	Ronnie_Corbett.
Ronnie	Barker	NULL	Ronnie_Barker.
Nobody	NULL	NULL	Nobody.

Finally this looks correct.

Nearly any effect can be produced with appropriate use of each of these techniques. Here is a more complicated example that composes an likely e-mail address for people based on their names and a 'server' field.

{DBREPORT(db="members")} SQL "SELECT CONCAT(CONCAT_WS('_',FirstName,MiddleNames,Surname),'@',Server) AS Email FROM People" TABLE ROW CELL "[Email]" {DBREPORT}

And here is what would be produced, given some example data:

FirstName	MiddleNames	Surname	Server	Output Text
John	Qubert	Public	NULL	
Ronnie	NULL	Corbett	google.com	Ronnie_Corbett@google.com
Ronnie	Barker	NULL	msn.com	Ronnie_Barker@msn.com
Nobody	NULL	NULL	yahoo.com	Nobody@yahoo.com

Note how there are never extra underscore characters where they're not needed to separate names, and how the entire string is blank in cases where there is no Server.

Security

When used correctly, security is quite good. Parameterized queries ensure that badly formed SQL due to funny requests can't happen.

However, since this extension passes SQL strings to the database, hacks are possible such as malicious users changing the query to "DROP TABLE *" or worse, if they can edit the report definition.

- Make sure pages with reports have their permissions set properly, so that general users of the page do not have edit privileges. If you can edit the page, you can change the query, or copy the report to another page and hack the query there.
- You could use a Wiki DSN (setup using the DSN Administrator, and specified using the "db=>" parameter to the plugin rather than a "dsn=>" string) instead of specifying the full connection string. This way, even if the page's permissions are incorrectly set, the DSN permissions will prevent unauthorized users from running the report, or from seeing how to connect to the database using other software. However, users authorized to access the DSN can write a new page with a report, and do anything the database allows, so you must trust them entirely.
- The most secure method is therefore to specify an entire DSN on each use of the report plugin, and then lock down the permissions so that authorized users can only view the page. That means that database reports will probably inhabit their own pages, and cannot be mixed with generally editable wiki content.