EXT: SQL-based Data Provider

Extension Key: dataquery
Language: en
Keywords: forAdmins, forIntermediates, tesseract

Table of Contents

Introduction

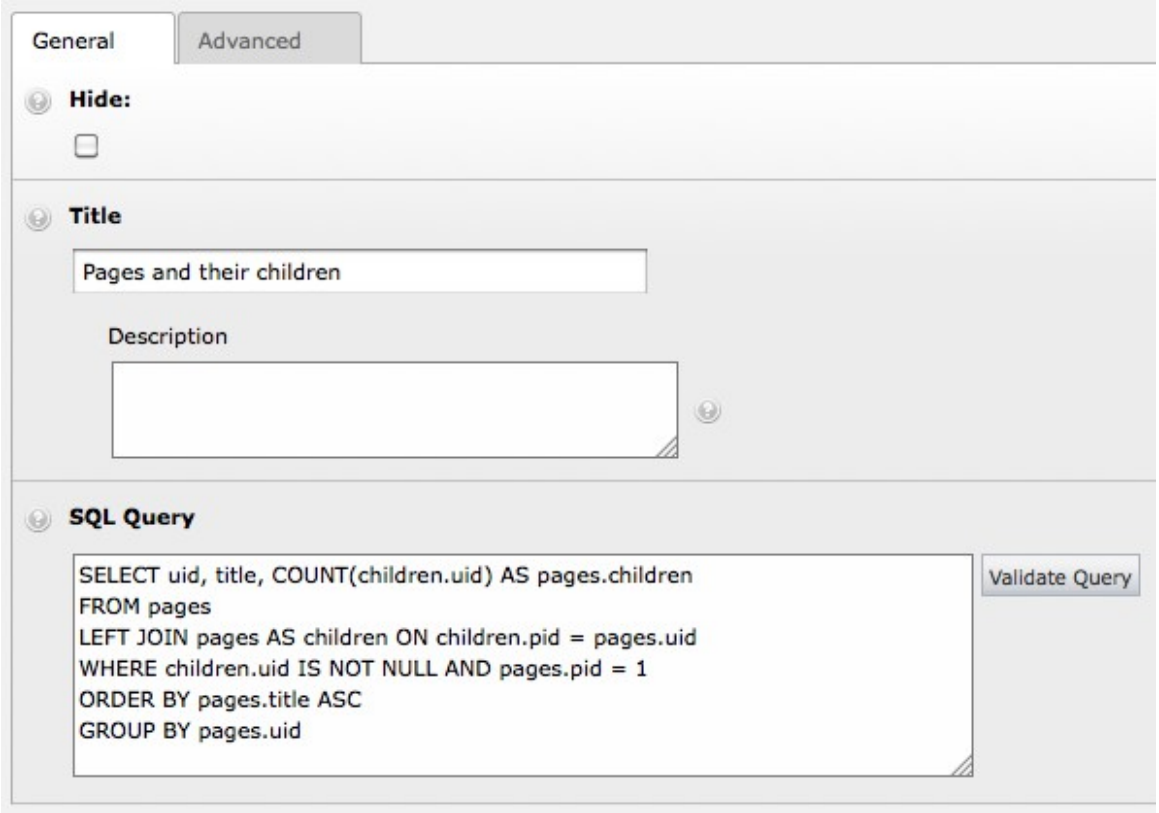
What does it do?

The dataquery extension is a Data Provider for the Tesseract framework. It can be used to query the local TYPO3 database and return the resulting recordset in a standardized data structure (SDS) of type "recordset" or "idlist".

It is designed to make it easy to write SQL by automating the handling of all the TYPO3 standard mechanisms, like enable fields and language overlays.

Screenshot

Here's a view of a Data Query record in the TYPO3 BE:



Questions?

If you have any questions about this extension, you may want to refer to the Tesseract Project web site (http://www.typo3-tesseract.com/) for support and tutorials. You may also ask questions in the TYPO3 English mailing list (typo3.english).

Keeping the developer happy

If you like this extension, do not hesitate to rate it. Go the Extension Repository, search for this extension, click on its title to go to the details view, then click on the "Ratings" tab and vote (you need to be logged in). Every new vote keeps the developer ticking. So just do it!

You may also take a step back and reflect about the beauty of sharing. Think about how much you are benefiting and how much yourself is giving back to the community.

Installation

Installation is pretty straightforward, but the extension is useless on its own. It must be installed as part of the Tesseract project. Currently the only extension making use of "dataquery" is the Display Controller ("displaycontroller").

The only condition is that "dataquery" must be installed **after** "datafilter", because it adds a field to the latter. If this order is not respected in your installation, either modify the extension list yourself (in `typo3conf/localconf.php`) or uninstall and reinstall "dataquery".

"dataquery" requires TYPO3 4.3 or above.

The extension provides to configuration options:

- **Cache limit**: Data Query comes with its own cache system, which is detailed later. The elements cached by Data Query may be quite large and may actually crash the database system. As such this option provides a way to limit the size of elements written to the cache table. Setting this option to 0 is equivalent to having no limit, but you should be cautious about this. See the chapter about cache for more details.

Compatibility

Since version 1.7.0, TYPO3 4.5 or more is required.

Upgrading to 1.5.0

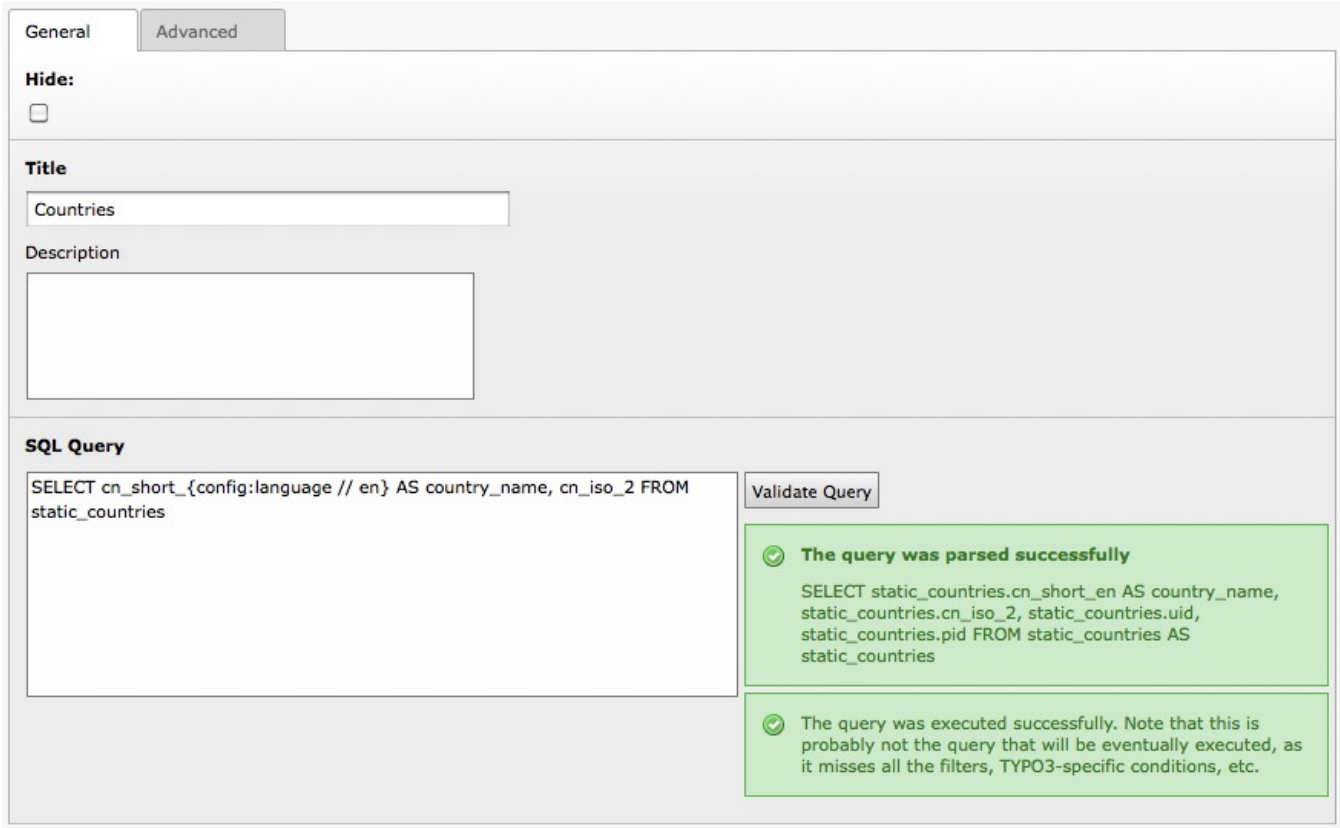
Note that the "Debug" flag was removed from the extension configuration options as of version 1.5.0. Data Query now refers to the calling controller to get the status of the debug flag.

User manual

Defining a query

To create a new SQL query using Data Query, create a new "Data Queries" record. This is divided into two tabs.

General tab



The "Hide" field is currently not actively used (i.e. the Display Controller does not check for it). It can be used to indicate that a given query is obsolete or should not be used.

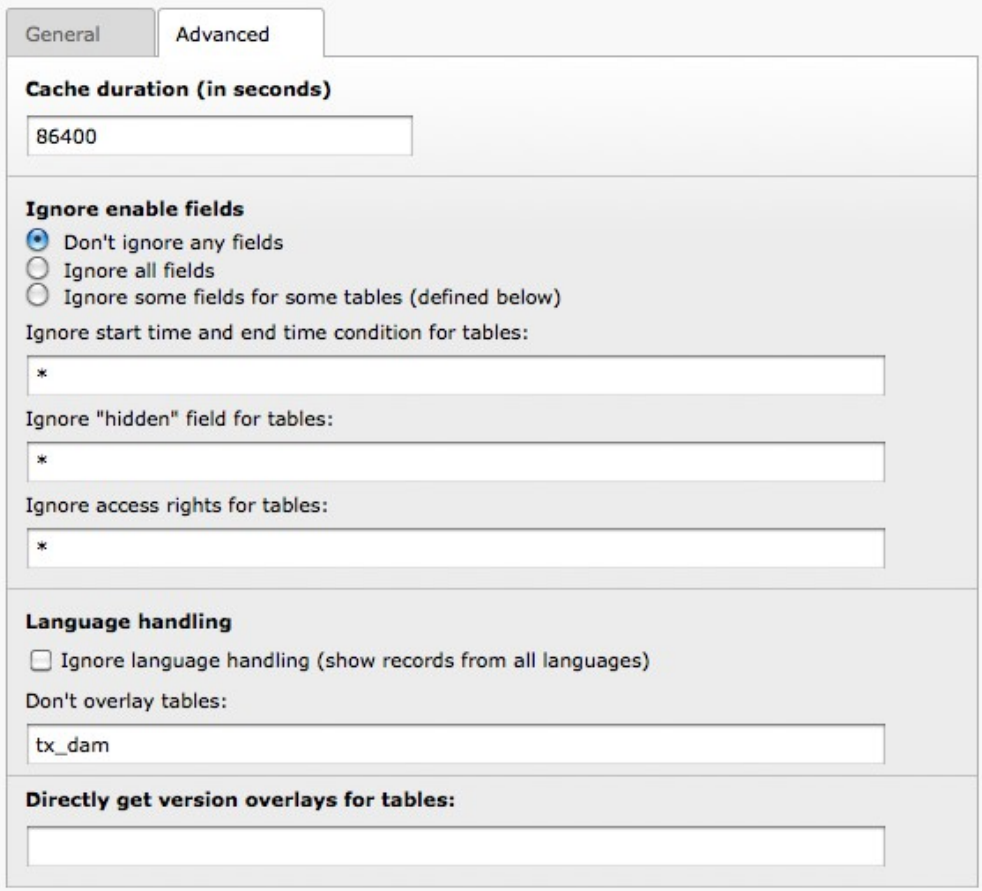
The "Title" field is mandatory and should be a short summary of what the query does. The "Description" field can be used for entering more information.

The action happens in the "SQL Query" field, where the actual query is written. Refer to the "Writing queries" chapter for a detailed explanation of the syntax allowed in this field (not all of the SQL elements can be used).

Next to the "SQL query" field is a "Validate Query" button, which can be used to verify that the query is alright. It does two checks:

- first, it parses the query and rebuilds it. The internal query parser may come up with errors at that point. If it doesn't the success message will display the resulting query (as in the screenshot above). Note that here TYPO3 mechanisms (see below) and Data Filters are not applied. It is only the "raw" query, as typed in the text field, that is parsed.
- next, the query is executed. This makes it possible to catch SQL errors early on. Note that a condition of "LIMIT 1" is applied to the query before executing it, to avoid draining server resources needlessly.

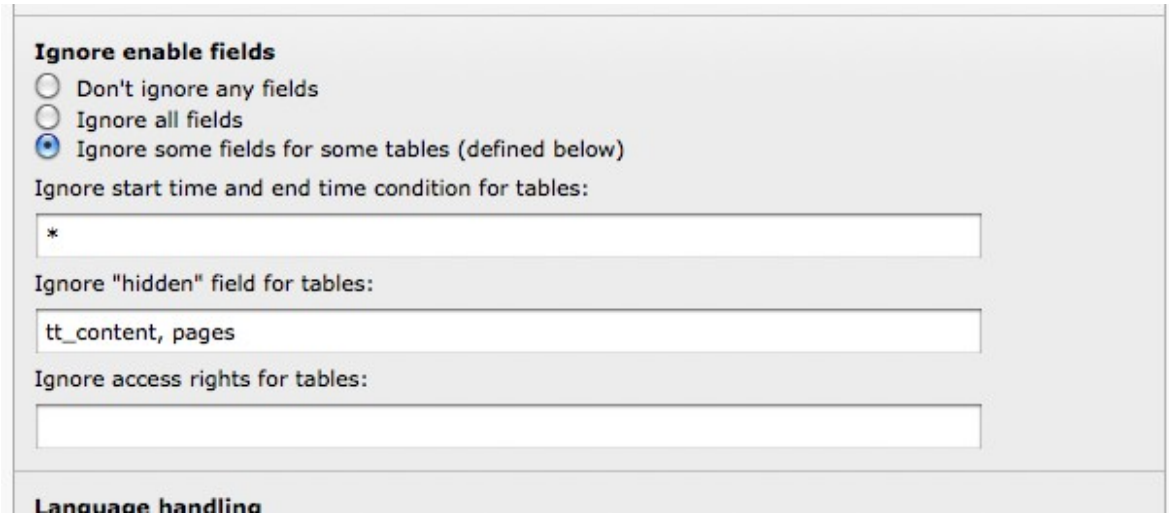
Advanced tab



The "Cache duration" field defines the period during which the result of the query must be kept in cache. The default is 86400 seconds (1 day). Setting a value of 0 disables the cache. See also the "Caching" chapter below.

By default, "dataquery" transparently handles all of TYPO3's enable fields, i.e. hidden or disabled flag, start and end time, and fe_groups access rights. This makes it very convenient to use, but may be undesired in some specific situations. In such a case, it is possible to completely disable this behavior by choosing the "Ignore all fields" options from the "Ignore enable fields" setting.

It is also possible to make more detailed choices. When choosing the "Ignore some fields for some tables" option, the next three text fields can be used to precisely define which enable field should be skipped and for which tables. Consider the following setup:



The meaning is the following:

- the start and end time enable fields should be ignored for all tables (using the "*" wildcard)
- the hidden field should be ignored for tables "tt_content" and "pages" (comma-separated list of table names)

- the access rights should not be ignored for any table (field is left blank)

If you used aliases for table names in your SQL query, you should use the same aliases in the ignore enable field settings.

On top of this “dataquery” also transparently handles everything related to language overlays. Again it is possible to disable that behavior by checking the “Language handling” box. On a finer level, it's possible to give a list of table for which language overlays (and conditions) should not be applied (in the above screenshot, records from table "tx_dam" will be taken in the default language and kept as is).

Data Query also handles versioning transparently, as soon as it is used within a workspace. This behavior cannot be switched off. However it is sometimes necessary to influence which records are taken when handling workspaces: the lives ones (that will be overlaid) or directly the ones that contain the modified version. This is the point of the "Directly get version overlays for tables" field. For any table listed there (comma-separated list), the records will be selected among the overlays and not the live ones. This if often necessary for handling JOINs correctly in workspaces. If you don't use workspaces, just ignore this option. For more details, see the "Limitations" section later.

See the “Behind the scene” chapter for more details about what Data Query handles automatically.

Note: language and version overlays are not applied when the type of Data Structure returned is "idlist" (e.g. when "dataquery" is called as a secondary provider via the Display Controller). The reasoning is that the list of ids returned will always be handled by further processes which will take care of language and version overlays.

Writing queries

As is discussed later, there's quite a lot of magic going on behind the scenes to make the users' lives easier. This comes at some cost: it is not possible to write queries in any arbitrary way. In effect, Data Query will accept only a sub-set of SQL commands. It also adds a few of its own for increased convenience. All the rules that pertain to writing queries are described below.

Important: all keywords must be written in *upper case*, as they appear below.

Allowed keywords

The following SQL keywords are allowed:

- SELECT
- FROM
- INNER JOIN
- LEFT JOIN
- RIGHT JOIN
- WHERE
- GROUP BY
- ORDER BY
- LIMIT
- OFFSET

On top of these keywords, it is possible to use:

- aliases (using "AS"), see "Advanced uses of aliases" in the "Behind the scenes" chapter below.
- the "ON" clause inside a JOIN.
- DISTINCT at the beginning of the list of SELECTed fields. See more about the use of DISTINCT below.
- ASC, DESC in the ORDER BY part.
- any function call in the list of SELECTed fields. More details in "SQL function calls" below.
- the alternate OFFSET syntax (i.e. you can use `LIMIT x,y` or `LIMIT y OFFSET x`).

The following is **not** supported:

- static values in the SELECT part of the query
- using brackets around field names. Thus don't write "`SELECT (title) AS headline FROM tt_news`", but "`SELECT title AS headline FROM tt_news`"

Using DISTINCT

Using the DISTINCT keyword in your queries requires to pay attention to a couple of peculiarities:

1. It is very likely that the DISTINCT field in your query is the actual primary key in your query. Assuming this, Data Query does not automatically add the "uid" field to this query (if it exists) to avoid messing up the effect of DISTINCT. Thus it is your responsibility to explicitly designate a "uid" in this case, using aliases. Example:

   ```
   SELECT DISTINCT name AS uid FROM fe_users
   ```

2. In a more complex scenario using several fields with DISTINCT, you should build a concatenated string with all fields involved to be used as uid. Example:

   ```
   SELECT DISTINCT name, username, CONCAT(name, username) AS uid FROM fe_users
   ```

Furthermore, please note that using DISTINCT will disable language overlays (because of the limitations discussed in "Behind the scenes" below).

Non-standard SQL keywords

Data Query is open to support non-standard SQL keywords for increased ease of use. Obviously these are not handled inside

the SQL query, but during processing of the recordset.

- MAX

The MAX keyword can be used inside a JOIN statement and represents a limit applied to the joined records. It is useful – for example – to get a single record for each join. Example:

```
SELECT pages.uid, pages.title, tt_content.uid, tt_content_header FROM pages
LEFT JOIN tt_content ON tt_content.pid = pages.uid MAX 1
```

The above query will return the first content element found for each page, instead of all of them.

Join rules

The following join types are recognized: INNER, LEFT and RIGHT.

Implicit table joins are recognized and transformed into INNER joins. So the following query:

```
SELECT * FROM tt_news,tx_dam WHERE …
```

would become:

```
SELECT * FROM tt_news INNER JOIN tx_dam WHERE …
```

SQL function calls

It is possible to use any SQL function in the SELECT part of the statement. Function calls are expected to be used with an alias. If this is not the case, an alias will be automatically added. So the following query:

```
SELECT FROM_UNIXTIME(tstamp, '%Y') FROM tt_content...
```

would be transformed into:

```
SELECT FROM_UNIXTIME(tstamp, '%Y') AS function_1 FROM tt_content...
```

Automatically generated aliases are called "function_" plus the position of the function inside the SELECT part (i.e. the first function is "1", the second is "2", etc.).

Note: this feature has been tested with many different functions, but there might be some particular where the SQL parser gets it wrong. If you should stumble on such an issue, don't hesitate to report it.

Expressions in queries

It is possible to use expressions inside a query. Consider the following:

```
SELECT cn_short_{config:language} FROM static_countries
```

The expression refers to the config.language TypoScript value. If that value is "en", the query will select the "cn_short_en" field. If it's "fr", it will be "cn_short_fr".

This is a very convenient way to dynamically modify a query.

Expressions can be used anywhere inside a query, but should **not** be used in the WHERE clause. Instead Data Filters should be used in this case.

For more information on expressions, please refer to the manual of the "expressions" extension.

Comments

It is possible to comment lines in the query by starting them with a "#" or "//" marker. Example:

```
SELECT * FROM tt_content
#WHERE header like '%foo%'
```

The second line in this query will be ignored.

Mandatory field "uid"

All queries must have a field called "uid" (this is necessary for all the processing that happens afterward, see "Behind the scenes"). It can be either the real "uid" of a given table or some other field using an alias, e.g.

SELECT FROM_UNIXTIME(tstamp, '%Y') AS uid FROM tt_content

If no explicit "uid" field is found in the SELECT statement, Data Query will try to add one automatically, which may have unexpected results. As such it is useful to use the "Validate Query" button and look at the query that Data Query rebuilt, to see if it really matches what you expected.

Additional SQL in Data Filter

Data Query adds an "additional SQL" field to the Data Filters. This makes it possible to directly add some SQL which will be added to the WHERE part of the query. This field can be used for SQL-specific conditions which cannot be expressed with standard Data Filter syntax.

Expressions can be used in the "additional SQL" field too.

Behind the scenes

To make it as easy and flexible as possible, a lot of magic goes on behind the scene in dataquery.

Query parsing and query building

The query that is entered in the "SQL Query" field is parsed by Data Query into its individual components. It is split along the various allowed keywords and each part can have whatever specific treatment is necessary. This is particularly true for the SELECT part. Each field is isolated and stored in an array. If the "*" wildcard is used, it is replaced by an explicit list of all fields from the given table.

In a second step Data Query makes sure that every necessary field is indeed selected. This means it will add a "uid" field if it finds none. It will also add language-related fields necessary for the overlay process.

In the WHERE part, conditions are added to match all enable fields of the given table, as defined in its TCA. Conditions for language overlays are also added if necessary. The same goes for workspaces: these are not directly supported (in the sense that the preview will not be correct), but Data Query at least ensures that all records selected belong to the live workspace.

This means that the user does not have to worry about all these TYPO3-specific fields when writing a query. For example, a simple query like:

```
SELECT uid, title FROM tt_news
```

will be transformed into:

```
SELECT tt_news.uid, tt_news.title, tt_news.pid AS tt_news$pid, tt_news.sys_language_uid AS
tt_news$sys_language_uid FROM tt_news AS tt_news WHERE tt_news.deleted=0 AND tt_news.t3ver_state<=0 AND
tt_news.hidden=0 AND (tt_news.starttime<=1257863340) AND (tt_news.endtime=0 OR
tt_news.endtime>1257863340) AND (tt_news.fe_group='' OR tt_news.fe_group IS NULL OR tt_news.fe_group='0'
OR (tt_news.fe_group LIKE '%,0,%' OR tt_news.fe_group LIKE '0,%' OR tt_news.fe_group LIKE '%,0' OR
tt_news.fe_group='0') OR (tt_news.fe_group LIKE '%,-1,%' OR tt_news.fe_group LIKE '-1,%' OR
tt_news.fe_group LIKE '%,-1' OR tt_news.fe_group='-1')) AND (tt_news.sys_language_uid IN (0,-1)) AND
tt_news.t3ver_oid = '0'
```

All conditions coming from filters are also added (see "Queries and Data Filters" below), as well as the additional SQL, if defined.

The ORDER clause also requires a special handling. See "Translations, ordering and limits" below.

Translations, ordering and limits

Translations are handled in TYPO3 using a concept called "overlays". In this system records are first fetched in the default language and then overlaid with their translations, coming from the same table (most commonly) or from a different table (as is the case for the "pages" table) depending on the chosen translation paradigm.

While this system has a lot of advantages, it also makes it difficult to select a limited number of records in a non-default language, sorted alphabetically. Indeed records are originally selected in the default language. So alphabetical ordering cannot take place simply during the SQL query. Furthermore some records may not exist as translations, and others exist only in non-default languages. This means that the total number of records cannot be known in advance for translations. Again the limit cannot be applied directly to the SQL query.

Data Query tries to make this part simpler too. When the current language is not the default, it will get the records in the default language, then get all the translations and perform the overlays as appropriate. For the ordering, Data Query will check if the fields being ordered on are alphanumeric fields or not. If the fields are not alphanumeric (e.g. date fields or integer fields) they can be ordered in the SQL statement, as overlays will not have an influence on them. On the other hand if at least one of the fields selected for ordering contains alphanumeric data, it will not be ordered using SQL. The ordering is done after overlays have been applied.

NOTE: for Data Query to work its magic, every field needs to be defined in the TCA. Be careful with the traditional "sorting" field, because it is normally not defined in the TCA, so Data Query will not consider it as an integer field. Adding a TCA definition for the "sorting" field solves this issue.

Limits are always applied on the resulting recordset and not in the SQL, except if they are explicitly defined using the LIMIT keyword in the SQL statement. So limits should be defined using data filters, unless there's a good reason to use the SQL LIMIT.

The drawback of this approach is that it requires far more computing power and memory than if the task could be delegated to the database (hoping that the database is optimized for such operations). It is the main reason why Data Query has its own caching mechanism (see related chapter), to avoid repeating operations.

Versioning

Pretty much the same can be said for the management of versioning (i.e. content created in workspaces). Getting the correct version of records is about actually getting:

- the live version of records, overlaying them with any existing change, delete those that do not exist in the workspace anymore, move those that were moved in the workspace
- the records that were created in the workspace and exist only there for the time being

All this makes it impossible to correctly apply a limit, since after getting the original records with the SQL statement, you may end up with less records (those that were deleted). This particular case is not taken into account by Data Query, so it is possible to stumble upon an unexpected number of records. This could be improved in the future if need arises. It has not been done now, because it would add yet more processing and does not seem to be mission-critical.

Limitations

The overlay mechanism described above adds other limitations beyond the simple number of records eventually returned by Data Query. Indeed it is not possible to build a multi-lingual full-text search. For a detailed explanation, please refer to the "Text search" section of the "Limitations" chapter in the manual of the "overlays" extension.

On the other hand Data Query goes beyond "overlays" limitation regarding table joins. Data Query effectively "de-joins" tables, overlays their records separately and joins them again after that. This process is not entirely transparent however. In general database relations created in a workspace will be built – quite obviously – with the primary keys of the overlay records (and not the live ones). In such a case, in order to be able to perform the join in SQL, it is necessary to query the overlay records and not the live ones. This is the reason for the "Directly get version overlays for tables" option described above.

Queries and Data Filters

The main idea that underlies the whole Tesseract concept is to have libraries of elements that can be reused and combined in different ways. Part of this flexibility comes from the use of Data Filters. Via a controller (like the Display Controller) a given Data Query can be set in relation with a Data Filter. This makes it possible to reuse a Data Query and change it dynamically via the filters.

Technically the filter structure created by the Data Filter is passed to the Data Query by the controller using the `tx_tesseract_dataprovider::setDataFilter()` method from the base provider interface. The filter structure is then translated into SQL by Data Query and added to the base query from the "SQL query" field.

Using aliases in filters

It is normally not possible to use aliases in the WHERE clause. However Data Query will recognize aliases used in Data Filters and map them to the original field they represented. Imagine the following query:

```
SELECT FROM_UNIXTIME(tstamp, '%Y') AS year FROM tt_content
```

with the following Data Filter:

```
year = date:Y
```

(which would select all content element edited during the current year). This will be (correctly) interpreted as:

```
SELECT FROM_UNIXTIME(tstamp, '%Y') AS year FROM tt_content WHERE (FROM_UNIXTIME(tstamp, '%Y') = 2010)
```

(assuming the current year is 2010), instead of:

```
SELECT FROM_UNIXTIME(tstamp, '%Y') AS year FROM tt_content WHERE (year = 2010)
```

which would cause a SQL syntax error.

Array values from filters

Imagine setting up a group of checkboxes like:

```
<input type="checkbox" name="tx_myext[foo][]" value="bob" />
<input type="checkbox" name="tx_myext[foo][]" value="alice" />
```

Next imagine a filter like:

```
fe_users.name like gp:tx_myext|foo
```

The value returned will be an array. This is handled by Data Query by creating a LIKE condition for each value and concatenating all these conditions with a "OR" logical operator. So the above example would result in the following SQL

condition (assuming both checkboxes were checked):

```
(fe_users.name LIKE '%bob%' OR fe_users.name LIKE '%alice%')
```

It's not possible to change the logical operator to "AND" (this didn't seem useful after thinking quite a bit about it; the whole reasoning is outside of the scope of this manual; if you have a use case for this, please open a feature request on Forge).

Data Query output

"dataquery" returns a standardized data structure of type "recordset". Refer to the "tesseract" manual for more details on such data structure.

Advanced uses of aliases

Aliases can be used in the SQL query to modify where the data is stored in the Data Structure (as explained above). It is possible to "move" a field from one table to another. Let's take a look at the SQL query that appears in the introductory screenshot:

```
SELECT uid, title, COUNT(children.uid) AS pages.children FROM pages LEFT JOIN pages AS children ON children.pid = pages.uid WHERE children.uid IS NOT NULL AND pages.pid = 1 ORDER BY pages.title ASC GROUP BY pages.uid
```

If you try to execute it as is you will get several SQL errors. It is indeed not correct, but will be by the time Data Query has rewritten it. Anyway the important point here is to look at the alias used for `COUNT(children.uid)`: "pages.children". What this will do is to "move" the "children" column to the "pages" table.

The result of the above query will be something like:

	pages$uid	pages$title	pages$children
0	1	My first page	2
1	5	Some other page	0
...			

To Data Query all fields now seem related to the "pages" table. This will result in a Data Structure with no subtables. This is often very convenient as it makes it easier to use the results (in "templatedisplay" for example) and can save unnecessary loops on subtables.

Caching

As explained above, Data Query performs some rather intensive calculations. In order to avoid repeating them needlessly, it implements it own caching mechanism. Cache is written to the `tx_dataquery_cache` table. Entries are keyed to the page they are related to (`page_id` field) and to the Data Query record being executed (`query_id` field). Furthermore every entry in this table is identified by a unique hash constructed from the following components:

- the current filter applied to the query, if any
- the list of primary keys provided by the secondary provider, if any
- additional, optional parameters (this feature is currently not used)
- the current FE language (`$GLOBALS['TSFE']->sys_language_content`)
- the current FE user groups, if any

The last two parameters ensure that the cache is stored correctly with respect to language and FE user rights, since all this is handled automatically by Data Query (as described above). The Data Structure gets stored into cache as a serialized array. Storage will be aborted if the size of the resulting string exceeds the cache limit, as defined in the extension's configuration (see "Installation" above). This mechanism ensures that the cache table does not grow out of control, as this could not only slow the system, but even crash the database server.

Every time Data Query needs to execute a query, it will first look if it has an existing, up to date Data Structure in the cache. If it does, it will get the cached data and unserialize it. Otherwise, it will calculate a new Data Structure.

The duration of the cache is defined for each query as shown in the "User Manual". The default value is 86400 seconds (= 1 day). Setting a value of "0" will disable caching for the query (see below).

When to avoid caching

The basic reasons for using the Data Query cache is to avoid calculating the whole Data Structure again if it has already been done. This can be – for example – because a given search pattern has recently been already used. Quite typically it is also

used when paginating through results. As Data Query handles the limits itself (as explained above) it will re-use the same Data Structure when paginating through records (the limit is not part of the cache key hash; all records are stored in the cache).

There are however circumstances when this caching mechanism is useless. When using a cached Display Controller (pi1), the resulting content will be put into the TYPO3 cache along with the rest of the page. When that page is called up again, it is served from the TYPO3 cache. Data Query is not called at all, its cache is not needed. In such a case, the cache duration of relevant queries should be set to "0" in order to avoid bloating the Data Query cache table with useless entries.

Cleaning up the cache

Whenever a query is modified the cache should be cleared, to ensure that old data will not be served anymore. Data Query hooks into TYPO3's cache clearing mechanism to simplify this task:

- when executing the "Clear all caches" command, the `tx_dataquery_cache` table will be emptied
- when executing the "Clear page cache" command, all `tx_dataquery_cache` entries related to that page will be deleted

However expired cache entries are **never** deleted, as TYPO3 does not provide an automatic way to do this. Instead you should seriously consider using an extension such as "cachecleaner" that makes it easy delete expired records from any database table on a regular basis.

Developer's Guide

Hooks

There are two hooks that can be used to manipulate the Data Structure produced by dataquery. They are similar but one is called before the structure is stored into cache, and the second one is called every time (i.e. either when the structure was freshly generated or when it was read from cache).

- **postProcessDataStructure**: called every time
- **postProcessDataStructureBeforeCache**: called when a structure has been newly generated and is about to be stored into cache (note that this hook is called also if the structure is **not** written to cache)

Both hooks receive as arguments the full Data Structure as well as a back-reference to the calling `tx_dataquery_wrapper` object. They are expected to return a complete Data Structure even if they did not perform any change.

Skeleton code for both hooks can be found in `samples/class.tx_dataquery_sample_hook.php`.

Another hook is available for manipulating the tables and fields information:

- **postProcessFieldInformation**: this hook is inside `tx_dataquery_wrapper::getTablesAndFields()`, a method which is called when "dataquery" provides Data Consumers with a list of available tables and fields while working within the TYPO3 backend (this is how, for example, "templatedisplay" knows which fields to map). This hook can be used when the data structure has been modified by one of the above hooks and such changes need to be known in the backend too. This is generally the case when the hooks change the tables and fields structure, so that these changed elements can be mapped properly.

Finally a hook can be used during cache hash calculation, for manipulating the parameters used to calculate the hash:

- **ProcessCacheHashParameters**: this hook is called inside `tx_dataquery_wrapper::calculateCacheHash()`. It receives as arguments the current cache parameters (an associative array) and a back-reference to the calling object (an instance of `tx_dataquery_wrapper`). It is expected to return the full array of cache parameters, whether it modified them or not.
  Classes using this hook must implement interface `tx_dataquery_cacheParametersProcessor`.

Known problems

The following problems have been identified:

- RIGHT JOINs will not produce a correct Data Structure (i.e. the orphan elements on the right side of the join do not appear).

If you have any issues, please refer to the Tesseract Project web site (http://www.typo3-tesseract.com/). You may also post your problems to the TYPO3 English mailing list (typo3.english), so that others may benefit from the answers too. For bugs or feature requests, please open an entry in the extension's bug tracker on Forge (http://forge.typo3.org/projects/extension-dataquery/issues).

To-Do List

The roadmap for the evolution of this extension can be found on Forge: http://forge.typo3.org/projects/roadmap/extension-dataquery

For feature requests, please open a report on Forge issue tracker: http://forge.typo3.org/projects/extension-dataquery/issues