The MariaDB CONNECT plugin Handler

Version 1.06.0007

Olivier Bertrand: 1, venelle d'en haut, 85330 Noirmoutier en l'Ile, France
Phone: 1(33) 2.51.39.57.84 – Cell: 1(33) 6.70.06.04.16
Email: bertrandop@gmail.com

Table of Content

Introduction

CONNECT is not just a new "YASE" (Yet another Storage Engine) that provides another way to store data with additional features. It brings a new dimension to MariaDB, already one of the best products to deal with traditional database transactional applications, into the world of business intelligence and data analysis, including NoSQL facilities. Indeed, BI is the set of techniques and tools for the transformation of raw data into meaningful and useful information. And where is this data?

"It's amazing in an age where relational databases reign supreme when it comes to managing data that so much information still exists outside RDBMS engines in the form of flat files and other such constructs. In most enterprises, data is passed back and forth between disparate systems in a fashion and speed that would rival the busiest expressways in the world, with much of this data existing in common, delimited files. Target systems intercept these source files and then typically proceed to load them via ETL (extract, transform, load) processes into databases that then utilize the information for business intelligence, transactional functions, or other standard operations. ETL tasks and data movement jobs can consume quite a bit of time and resources, especially if large volumes of data are present that require loading into a database. This being the case, many DBAs welcome alternative means of accessing and managing data that exists in file format."

This has been written by Robin Schumacher[1]. What he describes is known as MED (Management of External Data) enabling to handle data not stored in a DBMS database as it were stored tables (Federated Database System). An ISO standard exists that describe one way to implement and use MED in SQL by defining foreign tables for which an external FDW (Foreign Data Wrapper) has been developed in C.

This is a rather complex way to achieve this goal and MariaDB does not support the ISO SQL/MED standard but, to cover all these needs, possibly in transactional but mostly in decision support applications, features the new CONNECT plugin storage engine that provides an extended support of MED and NOSQL in a much simpler and powerful way.

The main features of CONNECT are:

1. No need for additional SQL language extensions.
2. Embedded wrappers for many external data types (files, data sources, virtual).
3. NoSQL query facilities for JSON, XML, and HTML files, and using Json UDF's.
4. NoSQL new data type MONGO accessing MongoDB collections as relational tables.
5. Read/Write access to external files of most commonly used formats.
6. Direct access to external data sources via ODBC, JDBC and MySQL or MongoDB API.
7. Only used columns are retrieved from external scan.
8. Push-down WHERE clauses when appropriate.
9. Support of special and virtual columns.
10. ~~Parallel execution of multi-table tables.~~
11. Supports partitioning by sub-files or by sub-tables (enabling table sharding).
12. Support of MRR for SELECT, UPDATE and DELETE.
13. Provides remote, block, dynamic and virtual indexing.
14. Can execute complex queries on remote servers.
15. Provides an API that allows writing additional FDW in C++.

This makes MariaDB featuring one of the most advanced support of MED and NoSQL, without the need of complex additions to the SQL syntax (foreign tables are "normal" tables using the CONNECT engine).

Giving MariaDB easy and natural access to external data enables to use all the power of its functions and its handling of the SQL language for developing business intelligence applications.

---

[1] Robin Schumacher, Vice President Products at DataStax and former Director of Product Management at MySQL. He has over 13 years of database experience in DB2, MySQL, Oracle, SQL Server and other database engines.

The CONNECT MariaDB handler

The present document describes the MariaDB new CONNECT plugin handler and gives many examples of using it.

This new handler enables MariaDB to access external local or remote data (MED). This is done by defining tables based on different data types, in particular files of various format, data extracted from other DBMS or products (such as Excel or MongoDB) via ODBC or JDBC, or data retrieved from the environment (for example DIR, WMI, and MAC tables)

This handler supports table partitioning, MariaDB virtual columns and permits defining "special" columns such as ROWID, FILEID and SERVID.

Version	Maturity	Distributed with	Remark
1.05.0003	gamma	MariaDB 10.0	Soon to be replaced by 1.06
1.06.0002	GA	MariaDB 10.1, 10.2 and 10.3	
1.06.0003	Beta	Source	Contains the MONGO table type.
1.06.0004	GA	MariaDB all versions	Deprecated.
1.06.0005	GA	MariaDB all versions	MONGO enabled only for MariaDB 10.2, 10.3.
1.06.0006	GA	MariaDB all versions	MONGO available only in MariaDB 10.2, 10.3.

Maturity of version 1.06.0003 is specified as "beta" when compiled from source with the MONGO table type because this is a new table type not yet thoroughly tested. However, this does not mean much because no precise definition of maturity exists. As a matter of facts, because CONNECT handles many table types, each type has different maturity depending on whether is old and well tested, not so much tested or newly implemented. This will be indicated when applicable.

New Feature of the new CONNECT Version

This version introduces a new table type MONGO. It enables users to access MongoDB collections and to regard them as relational tables. This table type and the possibility to specify some columns a JSON objects, coupled with the use of JSON functions, makes MariaDB one of the most complete tool to handle NOSQL information.

See the MONGO table type for a detailed description of MONGO in version 1.06.0006.

Loading the CONNECT handler

The CONNECT handler must be enabled like any other plugin, for instance using the INSTALL SONAME command.

```
INSTALL SONAME 'ha_connect.[so | dll]';
```

To be visible the *ha_connect.dll* or *ha_connect.so, ha_connect.so.0, ha_connect.so.0.0.0* libraries must be placed in the standard MariaDB plugin directory, which is automatically done when using the standard binary distribution.

System Variables

CONNECT defines twelve system variables:

Name	Type	C-Type	Default	Description
xtrace	session	set	0	Console trace values
type_conv	session	enum	YES	TEXT conversion to VARCHAR (no, yes, or skip)
conv_size	session	integer	8192	VARCHAR size when converted from TEXT

indx_map	global	Boolean	OFF	Enable file mapping for index files
work_size	session	uint	64M	Size of the allocation work area
use_tempfile	session	enum	AUTO	Using temporary file for UPDATE/DELETE
exact_info	session	Boolean	OFF	Return exact values to info queries
cond_push	session	Boolean	ON	Enabling *cond_push* for CONNECT
json_grp_size	session	integer	10	Max number of rows for JSON aggregate functions
json_null	session	string	<null>	Representation of JSON null values
jvm_path	global	string	NULL	Path to JVM library
class_path	global	string	NULL	Java class path
java_wrapper	session	string	See →	Java wrapper (default: wrappers/JdbcInterface)

connect_xtrace
This variable can be set to trace selected parts of the CONNECT execution. Possible values are:

Name	Value	Description
	0	No trace
YES	1	Basic trace
MORE	2	More tracing
INDEX	4	Index construction
MEMORY	8	Allocating and freeing memory
SUBALLOC	16	Sub-allocating in work area
QUERY	32	Constructed query send to external server
STMT	64	Currently executing statement
HANDLER	128	Creating and dropping CONNECT handlers
BLOCK	256	Creating and dropping CONNECT objects
MONGO	512	Mongo tracing

Console tracing can be set on the command line or later by names or values, for instance:

```
set global connect_xtrace=0;                // No trace
set global connect_xtrace='YES';            // By name
set global connect_xtrace=1;                // By value
set global connect_xtrace='QUERY,STMT';     // By name
set global connect_xtrace=96;               // By value
set global connect_xtrace=1023;             // Trace all
```

Set connect_xtrace to 0 (default) to stop tracing or to other values if a console tracing is desired. Note that to test this handler, MariaDB should be executed with the --console parameter because CONNECT prints some error and trace messages on the console[2].

connect_work_size
The connect_work_size variable permits allocating a larger memory sub-allocation space when dealing with very big tables if sub-allocation fails. Its minimum value is 4194304 and maximum value depends on the machine physical memory size. It must be specified in numeric when modified with the SET command. If the specified value is too big and memory allocation fails, the size of the work area remains but the variable value is not modified and should be reset.

connect_exact_info
This variable tells whether the CONNECT engine should return and exact record number value to information queries. It is OFF by default because this information can take a very long time for variable record length big tables or for remote tables, especially if the remote server is not available.

[2] In some Linux versions, this is re-routed into the error.log file.

It can be set to ON when exact values are desired, for instance when querying the repartition of rows in a partition table.

Other variables usages will be explained later where it applies.

Creating and dropping "CONNECT" Tables

Create Table statements for "CONNECT" tables are standard MySQL create statements specifying "engine=CONNECT". There are several additional table, column and index options specific to CONNECT.

Table options:

This is the list of table options that can be specified when creating or altering CONNECT tables. Some are standard MariaDB options but most of them are specific to the CONNECT engine. Because CONNECT implements many table types, only the most current options are used directly. Other options must be specified in the OPTION_LIST string and are described with the table type to which they apply.

(This list is prone to be added more options in future versions of the handler)

Table Option	Type	Description
ENGINE	String	Must be specified as CONNECT.
TABLE_TYPE	String	The external table type: DOS, FIX, BIN, CSV, FMT, XML, JSON, INI, DBF, VEC, ODBC, JDBC, MONGO, MYSQL, TBL, PROXY, XCOL, OCCUR, PIVOT, ZIP, VIR, DIR, WMI, MAC and EOM. Defaults to DOS, MYSQL, or PROXY depending on other options.
FILE_NAME	String	The file (path) name for all table types based on files. Can be absolute or relative to the current data directory. If not specified, this is an Inward table and a default value is used.
XFILE_NAME	String	The file (path) base name for a table index files. Can be absolute or relative to the data directory. Defaults to the file name.
TABNAME	String	The target table name for ODBC, JDBC, MONGO, MYSQL, PROXY or catalog tables or top node name for XML tables.
TABLE_LIST	String	The comma separated sub-table list of a TBL table.
DBNAME	String	The target database for ODBC, JDBC, MONGO, MYSQL, catalog, and PROXY based tables. The database concept is sometimes known as "Schema".
DATA_CHARSET	String	The character set used in the external file or data source.
SEP_CHAR	String	Specifies the field separator character of a CSV or XCOL table. Also, used to specify the Jpath separator for JSON tables.
QCHAR	String	Specifies the character used for quoting some fields of a CSV table or the identifiers of an ODBC/JDBC table.
SRCDEF	String	The source definition of a table retrieved via ODBC, JDBC or MySQL API or used by a PIVOT table.
COLIST	String	The column list of OCCUR tables or $project of MONGO tables.
FILTER	String	To filter an external table. Currently MONGO tables only.
MODULE	String	The (path) name of the DLL or shared lib implementing the access of a non-standard (OEM) table type.
SUBTYPE	String	The subtype of an OEM table type.
CATFUNC	String	The catalog function used by a catalog table.
OPTION_LIST	String	Used to specify all other options not yet directly defined.
CONNECTION	String	Specifies the connection of ODBC, JDBC, MONGO or MYSQL tables.
MAPPED	Boolean	Specifies whether "file mapping" is used to handle the table file.

Table Option	Type	Description
HUGE	Boolean	To specify that a table file can be larger than 2GB. For a MYSQL table, prevent the result set to be memory stored.
COMPRESS	Number	1 or 2 if the data file is g-zip compressed. Defaults to 0.
ZIPPED	Boolean	True if the table file(s) is/are zipped in one or several zip files.
SPLIT	Boolean	True for a VEC table when all columns are in separate files.
READONLY	Boolean	True if the data file must not be modified or erased.
SEPINDEX	Boolean	When true, indexes are saved in separate files.
BLOCK_SIZE	Number	The number of rows each block of a file based table contains. For an ODBC table this is the RowSet size option. For a JDBC table this is the fetch size. For a VIR table this is the table size in number of rows.
LRECL	Number	The file record size (often calculated by default).
AVG_ROW_LENGTH	Number	Can be specified to help CONNECT estimate the size of a variable record table length.
MULTIPLE	Number	Used to specify multiple file tables.
HEADER	Number	Applies to CSV, VEC and HTML files. Its meaning depends on the table type.
QUOTED	Number	The level of quoting used in CSV table files.
ENDING	Number	End of line length. Default to 1 for Unix/Linux and 2 for Windows.

For additional options specified in the OPTION_LIST option string, the syntax to use is:

```
… option_list='opname1=opvalue1,opname2=opvalue2…'
```

The option name is all that is between the start of the string or the last ‘,’ character and the next ‘=’ character, and the option value is all that is between this ‘=’ character and the next ‘,’ or end of string. For instance:

```
option_list='name=TABLE,coltype=HTML,attribute=border=1;cellpa
dding=5,headattr=bgcolor=yellow';
```

This defines four options, ‘name’, ‘coltype’, ‘attribute’, and ‘headattr’ with values ‘TABLE’, ‘HTML’, ‘border=1;cellpadding=5’, and ‘bgcolor=yellow’. The only restriction is that values cannot contain commas, but they can contain equal signs.

Column options:

Column Option	Type	Description
FLAG	Number	An integer value whose meaning depends on the table type.
FIELD_LENGTH	Number	Set the internal field length for DATE columns.
MAX_DIST*	Number	Maximum number of distinct values in this column.
DISTRIB*	Enum	“scattered”, “clustered”, “sorted” (ascending)
DATE_FORMAT	String	The format indicating how a date is stored in a file.
FIELD_FORMAT	String	The column format for some table types.
SPECIAL	String	The name of the SPECIAL column set to this column value.

*: These options are used for block indexing.

Index options:

Index Option	Type	Description
DYNAM	Boolean	Set the index as “dynamic”.
MAPPED	Boolean	Use index file mapping.

Note 1: Number, in the above option lists, is an unsigned big integer.

Note 2: Creating a CONNECT table based on file does not erase or create the file if the file name is specified in the CREATE TABLE statement (“outward” table). If the file does not exist, it will be populated by subsequent INSERT or LOAD commands or by the “AS select statement” of the CREATE TABLE command. Unlike the CSV engine, CONNECT easily permits to create tables based on already existing files, for instance files made by other applications. However, if the file name is not specified, a file with a name defaulting to *tabname.tabtype* will be created in the data directory (“inward” table).

Note 3: Dropping a CONNECT table is done with a standard DROP statement. For outward tables, this drops only the CONNECT table definition but does not erase the corresponding data file and index files. Use DELETE or TRUNCATE to do so. This is contrary to data and index files of inward tables that are erased on DROP like for other MariaDB engines.

Currently supported table types

CONNECT can handle very many table formats; it is indeed one of its main features. The TABLE_TYPE option specifies the type and format of the table. The available table types and their description are listed in the following table:

Type	Description
DOS	The table is contained in one or several files. The file format can be refined by some other options of the command or more often using a specific type as many of those described below. Otherwise, it is a flat text file where columns are placed at a fixed offset within each record, the last column being of variable length.
FIX	Text file arranged like DOS but with fixed length records.
BIN	Binary file with numeric values in platform representation, also with columns at fixed offset within records and fixed record length.
VEC	Binary file organized in vectors, in which column values are grouped consecutively, either split in separate files or in a unique file.
DBF*	File having the dBASE format.
CSV*	“Comma Separated Values” file in which each variable length record contains column values separated by a specific character (defaulting to the comma)
FMT	File in which each record contains the column values in a non-standard format (the same for each record) This format is specified in the column definition.
INI	File having the format of the initialization or configuration files used by many applications.
XML*	File having the XML or HTML format.
JSON*	File having the JSON format.
ZIP	Table giving information about the contain of a zip file.
ODBC*	Table extracted from an application accessible via ODBC or unixODBC. For example, from another DBMS or from an Excel spreadsheet.
JDBC*	Table accessed via a JDBC driver.
MONGO*	Table based on a MongoDB collection accessed via the MongoDB Java Driver or the MongoDB C Driver API[3].
MYSQL*	Table accessed using the MySQL API alike the FEDERATED engine.
PROXY*	A table based on another table existing on the current server.

[3] Using the MongoDB C Driver is available only when compiling MariaDB from source.

Type	Description
TBL*	Accessing a collection of tables as one table (like the MERGE engine does for MyISAM tables)
VIR*	Virtual table containing only special and virtual columns.
XCOL*	A table based on another table existing on the current server with one of its column containing comma separated values.
OCCUR*	A table based on another table existing on the current server, several columns of the object table containing values that can be grouped in only one column.
PIVOT*	Used to "pivot" the display of an existing table or view.
DIR	Virtual table that returns a file list like the Unix "ls" or DOS "dir" command.
WMI*	Windows Management Instrumentation table displaying information coming from a WMI provider. This type enables to get in tabular format all sort of information about the machine hardware and operating system (Windows only)
MAC	Virtual table returning information about the machine and network cards (Windows only)
OEM*	Table of any other formats not directly handled by CONNECT but whose access is implemented by an external FDW written in C++ (as a DLL or Shared Library).

Catalog Tables

For all table types marked with a '*', CONNECT is able to analyze the data source to retrieve the column definition. This can be used to define a "catalog" table that display the column description of the source, or to create a table without specifying the column definition that will be automatically constructed by CONNECT when creating the table.

These types and how to use them is described in the next chapter.

Data Types

Many data types make no or little sense when applied to plain files. This why CONNECT supports only a restricted set of data types. However, ODBC, JDBC or MYSQL source tables may contain data types not supported by CONNECT. In this case, CONNECT makes an automatic conversion to a similar supported type when it is possible.

The data types internally supported by CONNECT currently are:

Type name	Description	Used for
TYPE_STRING	Zero ended string	char, varchar, text
TYPE_INT	4 bytes integer	int, mediumint, integer
TYPE_SHORT	2 bytes integer	smallint
TYPE_TINY	1 byte integer	tinyint, Boolean
TYPE_BIGINT	8 bytes integer	bigint, longlong
TYPE_DOUBLE	8 bytes floating point	double, float, real
TYPE_DECIM	Numeric value	decimal, numeric, number
TYPE_DATE	4 bytes integer	date, datetime, time, timestamp, year

TYPE_STRING

This type corresponds to what is generally known as CHAR or VARCHAR by DB users, or as strings by programmers. Columns containing characters have a maximum length but the character string is of fixed or variable length depending on the file format.

The DATA_CHARSET option must be used to specify the character set used in the data source or file. Note that, unlike MariaDB, when a multi-byte character set is used, the column size represents the number of bytes the column value can contain, not the number of characters.

Starting with CONNECT version 1.6.4, the field length can be specified as 0. However, this make little sense with tables based on file because it corresponds to a zero-length field that cannot make the difference between a blank value or a null value. However, this can be used with an external type table that accesses a table fully supporting character fields of zero-length.

TYPE_INT

The INTEGER type contains integer numeric 4-byte values (the *int* of the C language) ranging from **–2,147,483,648** to **2,147,483,647** for signed type and **0** to **4,294,967,295** for unsigned type.

TYPE_SHORT

The SHORT data type contains integer numeric 2-byte values (the *short integer* of the C language) ranging from **–32,768** to **32,767** for signed type and **0** to **65,535** for unsigned type.

TYPE_TINY

The TINY data type contains signed integer numeric 1-byte values (the *char* of the C language) ranging from **–128** to **127** for signed type and **0** to **255** for unsigned type. For some table types, TYPE_TINY is used to represent Boolean values (0 is false, anything else is true).

TYPE_BIGINT

The BIGINT data type contains signed integer 8-byte values (the *long long* of the C language) ranging from **-9,223,372,036,854,775,808** to **9,223,372,036,854,775,807** for signed type and from **0** to **18,446,744,073,709,551,615** for unsigned type.

Inside tables, the coding of all numeric values depends on the table type. In tables represented by text files, the number is written in characters, while in tables represented by binary files (BIN or VEC) the number is directly stored in the binary representation corresponding to the platform.

The *length* (or *precision*) specification corresponds to the length of the table field in which the value is stored for text files only. It is used to set the output field length for all table types.

TYPE_DOUBLE

The DOUBLE data type corresponds to the C language *double* type, a floating-point double precision value coded with 8 bytes. Like for integers, the internal coding in tables depends on the table type, characters for text files, and platform binary representation for binary files.

The *length* specification corresponds to the length of the table field in which the value is stored for text files only. The *scale* (was *precision*) is the number of decimal digits written into text files. For binary table types (BIN and VEC) this does not apply. The *length* and *scale* specifications are used to set the output field length and number of decimals for all types of table.

TYPE_DECIM

The DECIMAL data type corresponds to what MySQL or ODBC data sources call NUMBER, NUMERIC or DECIMAL, a numeric value with a maximum number of digits (the precision) some of them eventually being decimal digits (the scale). The internal coding in CONNECT is a character representation of the number. For instance:

```
colname decimal(14,6)
```

This defines a column *colname* as numbers having a *precision* of 14 and a *scale* of 6. Supposing it is populated by:

```
insert into xxx values(-2658.74);
```

The internal representation of it will be the character string "-2658.740000". The way it is stored in a file table depends on the table type. The *length* field specification corresponds to the length of the table field in which the value is stored and is calculated by CONNECT from the *precision* and the *scale* values. This length is *precision* plus 1 if *scale* is not 0 (for the decimal point) plus 1 if this column is not unsigned

(for the eventual minus sign). In fix formatted tables the number is right justified in the field of width *length*, for variable formatted tables, such as CSV, the field is the representing character string.

Because this type in mainly used by CONNECT to handle numeric or decimal fields of ODBC, JDBC and MYSQL table types, CONNECT does not provide decimal calculations or comparison by itself. This is why decimal columns of CONNECT tables cannot be indexed.

DATE Data type

Internally, all temporal values are stored by CONNECT as a signed 4-bytes integer. The value 0 corresponds to January 01, 1970 12:00:00 am coordinated universal time (UTC). All other date/time values are represented by the number of seconds elapsed since or before midnight (00:00:00), January 1, 1970, to that date/time value. Date/time values before midnight January 1, 1970 are represented by a negative number of seconds.

CONNECT handles dates from **December 13, 20:45:52, 1901** to **January 18, 19:14:07, 2038**.

Although date and time information can be represented in both CHAR and INTEGER data types, the DATE data type has special associated properties. For each DATE value, CONNECT can store all or only some of the following information: century, year, month, day, hour, minute, and second.

Date Format in Text Tables

Internally, date/time values are handled as a signed 4-bytes integer. But in text tables (type DOS, FIX, CSV, FMT, and DBF) dates are most of the time stored as a formatted character string (although they also can be stored as a numeric string representing their internal value). Because there is infinity of ways to format a date, the format to use for decoding dates, as well as the field length in the file, must be associated to date columns (except when they are stored as the internal numeric value).

Note that this associated format is used only to describe the way the temporal value is stored internally. This format is used as well for output to decode the date in a SELECT statement as for input to encode the date in INSERT or UPDATE statements. However, what is kept in this value depends on the data type used in the column definition (all the MySQL temporal values can be specified.) When creating a table, the format is associated to a date column using the DATE_FORMAT option in the column definition, for instance:

```
create table birthday (
 Name varchar(17),
 Bday date field_length=10 date_format='MM/DD/YYYY',
 Btime time field_length=8 date_format='hh:mm tt')
engine=CONNECT table_type=CSV;
insert into birthday values('Charlie','2012-11-12','15:30:00');
select * from birthday;
```

The last query returns:

Name	Bday	Btime
Charlie	2012-11-12	15:30:00

The values of the INSERT statement must be specified using the standard MySQL syntax and these values are displayed as any MySQL temporal values. Indeed, the column formats apply only to the way these values are represented inside the data file. Here, the inserted record will be:

```
Charlie,11/12/2012,03:30 PM
```

Note: The FIELD_LENGTH option exists because the MySQL syntax does not allow specifying the field length between parentheses for temporal column types. If not specified, the field length is calculated from the date format (sometimes as a max value) or made equal to the default length value if there is no date format. In the above example, it could have been removed as the calculated values are the ones specified. However, if the table type would have been DOS or FIX, these values could be adjusted to fit the actual field length within the file.

A CONNECT format string consists of a series of elements that represent a particular piece of information and define its format. The elements will be recognized in the order they appear in the format string. Date and time format elements will be replaced by the actual date and time as they appear in the source string. They are defined by the following groups of characters:

Element	Description
YY	The last two digits of the year (that is, 1996 would be coded as "96").
YYYY	The full year (that is, 1996 could be entered as "96" but displayed as “1996”).
MM	The one or two-digit month number.
MMM	The three-character month abbreviation.
MMMM	The full month name.
DD	The one or two-digit month day.
DDD	The three-character weekday abbreviation.
DDDD	The full weekday name.
hh	The one or two-digit hour in 12-hour or 24-hour format.
mm	The one or two-digit minute.
ss	The one or two-digit second.
t	The one-letter AM/PM abbreviation (that is, AM is entered as "A").
tt	The two-letter AM/PM abbreviation (that is, AM is entered as "AM").

Usage Notes:
- To match the source string, you can add body text to the format string, enclosing it in single quotes or double quotes if it would be ambiguous. Punctuation marks do not need to be quoted.

- The hour information is regarded as 12-hour format if a “t” or “tt” element follows the “hh” element in the format or as 24-hour format otherwise.

- The "MM", "DD", "hh", "mm", "ss" elements can be specified with one or two letters (e.g. "MM" or "M") making no difference on input, but placing a leading zero to one-digit values on output[4] for two-letter elements.

- If the format contains elements DDD or DDDD, the day of week name is skipped on input and ignored to calculate the internal date value. On output, the correct day of week name is generated and displayed.

- Temporal values are always stored as numeric in BIN and VEC tables.

Handling dates that are out of the range of supported CONNECT dates
If you want to make a table containing, for instance, historical dates not being convertible into CONNECT dates, make your column CHAR or VARCHAR and store the dates in the MariaDB format. All date functions applied to these strings will convert them the MariaDB dates and will work as if they were real dates. Of course, they must be inserted and will be displayed using the MariaDB format.

NULL handling

CONNECT handles null values for data sources able to produce some. Currently this concerns mainly the ODBC, JDBC, MONGO, MYSQL, XML, JSON and INI table types. For INI, JSON, MONGO or XML types, null values are returned when the key is missing in the section (INI) or when the corresponding node does not exist in a row (XML, JSON, MONGO).

For other file tables, the issue is to define what a null value is. In a numeric column, 0 can sometimes be a valid value but, in some other cases, it can make no sense. The same for character columns; is a blank field a valid value or not?

[4] Here input and output are used to specify respectively decoding the date to get its numeric value from the data file and encoding a date to write it in the table file. Input is performed within Select queries; output is performed in Update or Insert queries.

A special case is DATE columns with a DATE_FORMAT specified. Any value not matching the format can be regarded as NULL.

CONNECT leaves the decision to you. When declaring a column in the CREATE TABLE statement, if it is declared NOT NULL, blank or zero values will be considered as valid values. Otherwise they will be considered as NULL values. In all cases, nulls are replaced on insert or update by pseudo null values, a zero-length character string for text types or a zero value for numeric types. Once converted to pseudo null values, they will be recognized as NULL only for columns declared as nullable.

For instance:

```
create table t1 (a int, b char(10)) engine=connect;
insert into t1 values (0,'zero'),(1,'one'),(2,'two'),(null,'???');
select * from t1 where a is null;
```

The select query replies:

a	b
NULL	zero
NULL	???

Indeed, the value 0 entered on the first row is regarded as NULL for a nullable column. However, if we execute the query:

```
select * from t1 where a = 0;
```

This will return no line because a NULL is not equal to 0 in an SQL where clause.

Now let us see what happens with not null columns:

```
create table t1 (a int not null, b char(10) not null) engine=connect;
insert into t1 values (0,'zero'),(1,'one'),(2,'two'),(null,'???');
```

The insert statement produces a warning saying:

Level	Code	Message
Warning	1048	Column 'a' cannot be null

It is replaced by a pseudo null 0 on the fourth row. Let us see the result:

```
select * from t1 where a is null;
select * from t1 where a = 0;
```

The first query returns no rows, 0 are valid values and not NULL. The second query replies:

a	b
0	zero
0	???

It shows that the NULL inserted value was replaced by a valid 0 value.

Unsigned numeric types

They are supported by CONNECT since version 1.01.0010 for fixed numeric types (TINY, SHORT, INTEGER and BIGINT)

Data type conversion

CONNECT can convert data from one type to another in many cases. These conversions are done without warning even when this leads to truncation or loss of precision. This is true, in particular, for tables of type ODBC, JDBC, MYSQL, and PROXY (via MySQL) because the source table may contain some data types not supported by CONNECT. They are converted when possible to CONNECT types.

MySQL types are converted as:

MySQL Types	CONNECT Type	Remark
integer, medium integer	TYPE_INT	4 bytes integer
small integer	TYPE_SHORT	2 bytes integer
tiny integer	TYPE_TINY	1 byte integer
char, varchar	TYPE_STRING	Same length. 0 length is now supported.
double, float, real	TYPE_DOUBLE	8 bytes floating point
decimal, numeric	TYPE_DECIM	Length depends on precision and scale
all date related types	TYPE_DATE	Date format can be set accordingly
bigint, longlong	TYPE_BIGINT	8 bytes integer
enum, set	TYPE_STRING	Numeric value not accessible
All text types	TYPE_STRING TYPE_ERROR	Depending on the *connect_type_conv* System variable value. See below.
Other types	TYPE_ERROR	Not supported, no conversion provided.

For ENUM, the length of the column is the length of the longest value of the enumeration. For SET the length is enough to contain all the set values concatenated with comma separator.

In the case of TEXT columns, the handling depends on the values given to the *connect_type_conv* and *connect_conv_size* system variables. If the value of *connect_type_conv* is:

NO No conversion. TYPE_ERROR is returned causing a "not supported" message.
YES (Default) The column is internally converted to TYPE_STRING corresponding to a column declared as VARCHAR(*n*), *n* being the value of *connect_conv_size*.
FORCE Also convert ODBC blob columns to TYPE_STRING.
SKIP No conversion. When column declaration is provided via Discovery (meaning the CONNECT table is created without column description) this column is not generated.

Note: *connect_type_conv* and *connect_conv_size* are session variables since version 1.3.6.

Note: BLOB is currently not converted by default until a TYPE_BIN type is added to CONNECT. However, the FORCE option can be specified for blob columns containing text and the SKIP option also applies to ODBC BLOB columns.

When converted, ODBC SQL types are converted as:

SQL Types	Connect Type	Remark
SQL_CHAR, SQL_VARCHAR	TYPE_STRING	
SQL_LONGVARCHAR	TYPE_STRING	len = min(abs(len), *connect_conv_size*)[5]
SQL_NUMERIC, SQL_DECIMAL	TYPE_DECIM	
SQL_INTEGER	TYPE_INT	
SQL_SMALLINT	TYPE_SHORT	
SQL_TINYINT, SQL_BIT	TYPE_TINY	
SQL_FLOAT, SQL_REAL, SQL_DOUBLE	TYPE_DOUBLE	

[5] If the column is generated by discovery (columns not specified) its length is *connect_conv_size*.

SQL Types	Connect Type	Remark
SQL_DATETIME	TYPE_DATE	len = 10.
SQL_INTERVAL	TYPE_STRING	len = 8 + ((scale) ? (scale+1) : 0)
SQL_TIMESTAMP	TYPE_DATE	len = 19 + ((scale) ? (scale +1) : 0)
SQL_BIGINT	TYPE_BIGINT	
SQL_GUID	TYPE_STRING	Len = 36.
SQL_BINARY, SQL_VARBINARY, SQL_LONGVARBINARY	TYPE_STRING	len = min(abs(len), *connect_conv_size*)[6]
Other types	TYPE_ERROR	Not supported.

JDBC types are converted as:

JDBC Types	Connect Type	Remark
(N)CHAR, (N)VARCHAR	TYPE_STRING	
LONG(N)VARCHAR	TYPE_STRING	len = min(abs(len), *connect_conv_size*)[7]
NUMERIC, DECIMAL, VARBINARY	TYPE_DECIM	
INTEGER	TYPE_INT	
SMALLINT	TYPE_SHORT	
TINYINT, BIT	TYPE_TINY	
FLOAT, REAL, DOUBLE	TYPE_DOUBLE	
DATE	TYPE_DATE	len = 10.
TIME	TYPE_DATE	len = 8 + ((scale) ? (scale+1) : 0)
TIMESTAMP	TYPE_DATE	len = 19 + ((scale) ? (scale +1) : 0)
BIGINT	TYPE_BIGINT	
UUID (specific to PortgreSQL)	TYPE_STRING TYPE_ERROR	len = 36 If *connect_type_conv* = NO
Other types	TYPE_ERROR	Not supported.

Note: The *connect_type_conv* SKIP option also applies to ODBC and JDBC tables.

[6] Only if the value of *connect_type_conv* is FORCE. The column should use the binary charset.
[7] If the column is generated by discovery (columns not specified) its length is *connect_conv_size*.

Inward and Outward Tables

There are two broad categories of file based CONNECT tables.

Outward Tables

Tables are “outward” when their file name is specified in the CREATE TABLE statement using the FILE_NAME option.

Firstly, remember that CONNECT implements MED (Management of External Data). This means that the "true" CONNECT tables – “outward tables” – are based on data that belong to files that can be produced by other applications or data imported from another DBMS.

Therefore, their data is "precious" and should not be modified except by specific commands such as INSERT, UPDATE, or DELETE. For other commands like for instance CREATE, DROP, or ALTER their data is never modified or erased.

Outward tables can be created on existing files or external tables. When they are dropped, only the local description is dropped, the file or external table is not dropped or erased.

Altering Outward tables

When an ALTER TABLE is issued, it just modifies the table definition accordingly without changing the data. ALTER can be used safely to, for instance, modify options such as MAPPED, HUGE or READONLY but with extreme care when modifying column definitions or order options because some column options such as FLAG should also be modified or may become wrong.

Changing the table type with ALTER often makes no sense. But many suspicious alterations can be acceptable if they are just meant to correct an existing wrong definition.

Translating a CONNECT table to another engine is alright but the opposite is forbidden when the target CONNECT table is not table based or when its data file exists (because the target table data cannot be changed and, the source table being dropped, the table data would be lost.) However, it can be done to create a new file-based tables when its file does not exist or is void.

Creating or dropping indexes is accepted because it does not modify the table data. However, it is often unsafe to do it with an ALTER TABLE statement that does other modifications.

Of course, all changes are acceptable for empty tables.

Note: Using outward tables requires the FILE privilege.

Inward Tables

A special type of file CONNECT tables are “inward” tables. They are file-based tables whose file name is not specified in the CREATE TABLE statement (no FILE_NAME option).

Their file will be located in the current database directory and their name will default to *tabn.typ* where *tabn* is the table name and *typ* is the table type folded to lower case. When they are created without using a CREATE TABLE … SELECT … statement, an empty file is made at create time and they can be populated by further inserts.

They behave like tables of other storage engines and, unlike outward CONNECT tables, are erased when the table is dropped. Of course, they should not be read-only to be usable. Even their utility is limited, they can be used for testing purpose or when the user has not the FILE privilege.

Altering Inward tables

One thing to know, because CONNECT builds indexes in a specific way, is that all index modifications are done using "in-place" algorithm – meaning not using a temporary table. This is why, when indexing is specified in an ALTER TABLE statement containing other changes that cannot be done “in-place”, the statement cannot be executed and raises an error.

Converting an inward table to an outward table, using an ALTER TABLE statement specifying a new file name and or a new table type, is restricted the same way it is when converting a table from another engine to an outward table. However, they are no restrictions to convert another engine table to a CONNECT inward table.

Relational Table Types

The main feature of CONNECT is to give MariaDB the ability to handle tables from many sources, native files, other DBMS's tables, or special "virtual" tables. Moreover, for all tables physically represented by data files, CONNECT recognizes many different file formats, described below but not limited in the future to this list, because more can be easily added to it on demand (OEM tables).

Most of these tables are based on files whose records represent one table row. Only the column representation within each record can differ

Data Files

Most of the tables processed by CONNECT are just plain DOS or UNIX data files, logically regarded as tables thanks to their description given when creating the table. This description comes from the CREATE TABLE statement as explained in this document. Depending on the application, these tables can already exist as data files, used as is by CONNECT, or can have been physically made by CONNECT as the result of a CREATE TABLE … SELECT … and/or INSERT statement(s).

The file *path/name* is given by the FILE_NAME option. If it is a relative path/name, it will be relative to the database directory, the one containing the table .FRM file.

Unless specified, the maturity of file table types is: STABLE.

Multiple File Tables

There are two types of multiple file tables. The first one is partitioned tables when each partition is stored in a separate file. CONNECT partition tables are described later in this document.

The second one is tables specified as "multiple". A **multiple** file table is one that is physically contained in several files of the same type instead of just one. These files are processed sequentially during the process of a query and the result is the same as if all the table files were merged into one. This is great to process files coming from different sources (such as cash register log files) or made at different time periods (such as bank monthly reports) regarded as one table. Note that the operations on such files are restricted to sequential Select and Update; and that VEC multiple tables are not supported by CONNECT. The file list depends on the setting of the **multiple** option of the CREATE TABLE statement for that table.

Multiple tables are specified by the option MULTIPLE= *n*, which can take four values:

0	Not a multiple table (the default). This can be used in an alter table statement.
1	The table is made from files located in the same directory. The FILE_NAME option is a pattern such as `'cash*.log'` that all the table file path/names verify.
2	The FILE_NAME gives the name of a file that contains de path/names of all the table files. This file can be made using a DIR table.
3	Like multiple=1 but also including eligible files from the directory sub-folders.

The **FILEID** special column, described later in this document, allows query pruning by filtering the file list or doing some grouping on the files that make a multiple table.

Note: Multiple was not implemented for XML tables. This restriction is removed since version 1.02.

Record Format

This characteristic applies to table files handled by the operating system input/output functions. It is **fixed** for table types FIX, BIN, DBF and VEC, and it is **variable** for DOS, VCT, FMT and some JSON tables.

For fixed tables, most I/O operations are done by block of BLOCK_SIZE rows. This diminishes the number of I/O's and enables block indexing.

Starting with this CONNECT version, the BLOCK_SIZE option can also be specified for variable tables. Then, a file similar to the block indexing file is created by CONNECT that gives the size in bytes of each block of BLOCK_SIZE rows. This enables to use block I/O's and block indexing to variable tables. It also enables CONNECT to return the exact row number for info commands.

File Mapping

For file-based tables of reasonable size, processing time can be greatly enhanced under Windows™ or some flavor of UNIX or Linux by using the technique of "file mapping", in which a file is processed as if it were entirely in memory. Mapping is specified when creating the table using the MAPPED=YES option. This does not apply to tables not handled by system I/O functions (XML and INI).

Big File tables

Because all files are handled by the standard input/output functions of the operating system, their size is limited to 2GB, the maximum size handled by standard functions. For some table types, CONNECT can deal with files that are larger than 2GB, or prone to become larger than this limit. These are the FIX, BIN and VEC types. To tell connect to use input/output functions dealing with big files, specify the option `huge=1` or `huge=YES` for that table. Note however that CONNECT cannot randomly access tables having more than 2G records.

Compressed file Tables

CONNECT can make and processed some tables whose data file is compressed. The only supported compression format is the gzlib format. Zip and zlib formats are supported differently. The table types that can be compressed are DOS, FIX, BIN, CSV and FMT. This can save some disk space at the cost of a somewhat longer processing time.

Some restrictions apply to compressed tables:

- Compressed tables are not indexable.
- Update and partial delete are not supported.

Use the numeric COMPRESS option to specify a compressed table:

0 Not compressed
1 Compressed in gzlib format.
2 Made of compressed blocks of BLOCK_SIZE records (enabling block indexing)

Zipped file Tables

Connect can work on the table file(s) that are compressed in one or several zip files.

The specific options used when creating tables based on zip files are:

Table Option	Type	Description
ZIPPED	Boolean	Required. To be set as true.
ENTRY*	String	The optional name or pattern of the zip entry or entries to be used with the table. If not specified, all entries or only the first one will be used depending on the *mulentries* option setting.
MULENTRIES*	Boolean	True if several entries are part of the table. If not specified, it defaults to *false* if the *entry* option is not specified. If the *entry* option is specified, it defaults to *true* is the entry name contains wildcard characters or *false* if it does not.
APPEND*	Boolean	Used when creating new zipped tables (see below)
LOAD*	String	Used when creating new zipped tables (see below)

Note: Options marked with a '*' must be specified in the option list.

Examples of use: Let's suppose you have a CSV file from which you would create a table by:

```
create table emp
... optional column definition
engine=connect table_type=CSV file_name='E:/Data/employee.csv'
sep_char=';' header=1;
```

If the CSV file is included in a ZIP file, the CREATE TABLE becomes:

```
create table empzip
... optional column definition
engine=connect table_type=CSV file_name='E:/Data/employee.zip'
sep_char=';' header=1 zipped=1 option_list='Entry=emp.csv';
```

The *file_name* option is the name of the zip file. The *entry* option is the name of the entry inside the zip file. If there is only one entry file inside the zip file, this option can be omitted.

If the table is made from several files such as emp01.csv, emp02.csv, etc., the standard create table would be:

```
create table empmul
... optional column definition
engine=connect table_type=CSV file_name='E:/Data/emp*.csv'
sep_char=';' header=1 multiple=1;
```

But if these files are all zipped inside a unique zip file, it becomes:

```
create table empzmul
... optional column definition
engine=connect table_type=CSV file_name='E:/Data/emp.zip'
sep_char=';' header=1 zipped=1 option_list='Entry=emp*.csv';
```

Here the *entry* option is the pattern that the files inside the zip file must match. If all entry files are ok, the *entry* option can be omitted but the Boolean option *mulentries* must be specified as true.

If the table is created on several zip files, it is specified as for all other multiple tables:

```
create table zempmul
... optional column definition
engine=connect table_type=CSV file_name='E:/Data/emp*.zip'
sep_char=';' header=1 multiple=1 zipped=yes
option_list='Entry=employee.csv';
```

Here again the *entry* option is used to restrict the entry file(s) to be used inside the zip files and can be omitted if all are Ok. All values of the *multiple* options are acceptable, for instance 3 to include also zip files located in sub-directories.

The column descriptions can be retrieved by the discovery process for table types allowing it. For multiple tables or multiple entries, it is supported only for CSV tables.

Catalog table can be created by adding **catfunc=columns**. This can be used to show the column definitions of multiple tables. Multiple must be set to false and the column definitions will be the ones of the first table or entry.

This first implementation has some restrictions:

1. Zipped tables are read only. UPDATE and DELETE are not supported. However, INSERT is supported in a specific way when making tables.
2. The inside files are decompressed into memory. Memory problems may arise with huge files.
3. Only file types that can be handled from memory are eligible for this. This includes DOS, FIX, BIN, CSV, FMT, JSON, and XML table types, as well as types based on these such as XCOL, OCCUR and PIVOT.

Optimization by indexing or block indexing is possible for table types supporting it. However, it applies to the uncompressed table. This means that the whole table is always uncompressed.

Partitioning is also supported. See how to do it in the chapter about partitioning.

Creating new zipped tables
Tables can be created to access already existing zip files. However, is it also possible to make the zip file from an existing file or table. Two ways are available to make the zip file:

Insert method:
INSERT can be used to make the table file for table types based on records (this excludes XML and JSON when PRETTY is not 0). However, the current implementation of the used package (minizip) does not support adding to an already existing zip entry. This means that when executing an INSERT statement, the inserted records would not be added but would replace the existing ones. CONNECT protects existing data by not allowing such inserts. Therefore, only three ways are available to do so:

1) Using only one INSERT statement to make the whole table. This is possible only for small tables and is principally useful when making tests.
2) Making the table from the data of another table. This can be done by executing an "`insert into table select * from another_table`" or by specifying "`as select * from another_table`" in the create table statement.
3) Making the table from a file whose format enables to use the "`load data infile`" statement.

To add a new entry in an existing zip file, specify "append=YES" in the option list. When inserting several entries, use ALTER to specify the required options, for instance:

```
create table znumul (
Chiffre int(3) not null,
Lettre char(16) not null)
engine=CONNECT table_type=CSV
file_name='C:/Data/FMT/mnum.zip' header=1 lrecl=20 zipped=1
option_list='Entry=Num1';
insert into znumul select * from num1;
alter table znumul option_list='Entry=Num2,Append=YES';
insert into znumul select * from num2;
alter table znumul option_list='Entry=Num3,Append=YES';
insert into znumul select * from num3;
alter table znumul option_list='Entry=Num*,Append=YES';
select * from znumul;
```

The last ALTER is needed to display all the entries.

File zipping method
This method enables to make the zip file from another file when creating the table. It applies to all table types including XML and JSON. It is specified in the CREATE TABLE statement with the LOAD option. For example:

```
create table XSERVZIP (
NUMERO varchar(4) not null,
LIEU varchar(15) not null,
CHEF varchar(5) not null,
FONCTION varchar(12) not null,
NOM varchar(21) not null)
engine=CONNECT table_type=XML file_name='E:/Xml/perso.zip' zipped=1
option_list='entry=services,load=E:/Xml/serv2.xml';
```

When executing this statement, the *serv2.xml* file will be zipped as *perso.zip*. The entry name must be specified as well as the column descriptions that cannot be retrieved from the zip entry file that does not exist yet.

It is also possible to create a multi-entries table from several files:

```
CREATE TABLE znewcities (
  _id char(5) NOT NULL,
  city char(16) NOT NULL,
  lat double(18,6) NOT NULL `FIELD_FORMAT`='loc:[0]',
  lng double(18,6) NOT NULL `FIELD_FORMAT`='loc:[1]',
  pop int(6) NOT NULL,
  state char(2) NOT NULL
) ENGINE=CONNECT TABLE_TYPE=JSON FILE_NAME='E:/Json/newcities.zip'
    ZIPPED=1 LRECL=1000
    OPTION_LIST='Load=E:/Json/city_*.json,mulentries=YES,pretty=0';
```

Here the files to load are specified with wildcard characters and the MULENTRIES options must be specified. However, the ENTRY option must not be specified, entry names will be made from the file names.

ZIP Table Type

A ZIP table type is also available. It is not meant to read the inside files but to display information about the zip file contain. For instance:

```
create table xzipinfo2 (
fn varchar(256)not null,
cmpsize bigint not null flag=1,
uncsize bigint not null flag=2,
method int not null flag=3,
date datetime not null flag=4)
engine=connect table_type=ZIP file_name='E:/Data/Json/cities.zip';
```

This will display the name, compressed size, uncompressed size, compress method, and compacting date of all entries inside the zip file. Column names are irrelevant; this is the *flag* value that means what information to retrieve.

DOS and FIX Table Types

Table of type DOS and FIX are based on text files. Within a record, column fields are positioned at a fixed offset from the beginning of the record. Except sometimes for the last field, column fields are also of fixed length. If the last field has varying length, the type of the table is DOS. For instance, having the file *dept.dat* formatted like:

```
0318 KINGSTON       70012 SALES       Bank/Insurance
0021 ARMONK         87777 CHQ         Corporate headquarter
0319 HARRISON       40567 SALES       Federal Administration
2452 POUGHKEEPSIE   31416 DEVELOPMENT Research & development
```

You can define a table based on it with:

```
create table department (
number char(4) not null,
location char(15) not null flag=5,
director char(5) not null flag=20,
function char(12) not null flag=26,
name char(22) not null flag=38)
engine=CONNECT table_type=DOS file_name='dept.dat';
```

Here the flag column option represents the offset of this column inside the records. If the offset of a column is not specified, it defaults to the end of the previous column and defaults to 0 for the first one. The *lrecl* parameter that represents the maximum size of a record is calculated by default as the end of the rightmost column and can be unspecified except when some trailing information exists after the rightmost column.

Note: A special case is files having an encoding such as UTF-8 (for instance specifying charset=UTF8) in which some characters may be represented with several bytes. Unlike the type size that MariaDB interprets as a number of characters, the lrecl value is the record size in bytes and the flag value represents

the offset of the field in the record in bytes. If the flag and/or the lrecl value are not specified, they will be calculated by the number of character in the fields multiplied by a value that is the maximum size in bytes of a character for the corresponding charset. For UTF-8 this value is 3 that is often far too much as they are very few characters requiring 3 bytes to be represented. When creating a new file, you are on the safe side by only doubling the maximum number of characters of a field to calculate the offset of the next field. Of course, for already existing files, the offset must be specified according to what exists within it.

Although the field representation is always text in the table file, you can freely choose the corresponding column type, characters, date, integer or floating point per its contents.

Sometimes, as in the *number* column of the above *department* table, you have the choice of the type, numeric or characters. This will modify how the column is internally handled -- in characters 0021 is different from 21 but not in numeric -- as well as how it is displayed.

If the last field has fixed length, the table should be referred as having the type FIX. For instance, to create a table on the file *boys.txt*:

```
John         Boston       25/01/1986  02/06/2010
Henry        Boston       07/06/1987  01/04/2008
George       San Jose     10/08/1981  02/06/2010
Sam          Chicago      22/11/1979  10/10/2007
James        Dallas       13/05/1992  14/12/2009
Bill         Boston       11/09/1986  10/02/2008
```

You can for instance use the command:

```
create table boys (
name char(12) not null,
city char(12) not null,
birth date not null date_format='DD/MM/YYYY',
hired date not null date_format='DD/MM/YYYY' flag=36)
engine=CONNECT table_type=FIX file_name='boys.txt' lrecl=48;
```

Here some *flag* options were not specified because the fields have no intermediate space between them except for the last column. The offsets are calculated by default adding the field length to the *offset* of the preceding field. However, for formatted date columns, the offset in the file depends on the format and cannot be calculated by default. For fixed files, the *lrecl* option is the physical length of the record including the line ending character(s). It is calculated by adding to the end of the last field 2 bytes under Windows (CRLF) or 1 byte under UNIX. If the file is imported from another operating system, the ENDING option will have to be specified with the proper value.

For this table, the last offset and the record length must be specified anyway because the date columns have field length coming from their format that is not known by CONNECT. Do not forget to add the line ending length to the total length of the fields.

This table is displayed as:

name	city	birth	hired
John	Boston	1986-01-25	2010-06-02
Henry	Boston	1987-06-07	2008-04-01
George	San Jose	1981-08-10	2010-06-02
Sam	Chicago	1979-11-22	2007-10-10
James	Dallas	1992-05-13	2009-12-14
Bill	Boston	1986-09-11	2008-02-10

Whenever possible, the fixed format should be preferred to the varying one because it is much faster to deal with fixed tables than with variable tables. Indeed, instead of being read or written record by record, FIX tables are processed by blocks of BLOCK_SIZE records, resulting in far less input/output operations to execute. The block size defaults to 100 if not specified in the Create Table statement. For tables of varying format, block read/write can be also obtained by specifying the BLOCK_SIZE option. CONNECT construct in that case a file containing the size of each block.

Note 1: It is not mandatory to declare in the table all the fields existing in the source file. However, if some fields are ignored, the *flag* option of the following field and/or the *lrecl* option will have to be specified.

Note 2: Some files have an EOF marker (CTRL+Z 0x1A) that can prevent the table to be recognized as fixed because the file length is not a multiple of the fixed record size. To indicate this, use in the option list the create option EOF. For instance, if after creating the FIX table *xtab* on the file *foo.dat* that you know have fixed record size, you get, when you try to use it, a message such as:

```
File foo.dat is not fixed length, len=302587 lrecl=141
```

After checking that the LRECL default or specified specification is correct, you can indicate to ignore that extra EOF character by:

```
alter table xtab option_list='eof=1';
```

Of course, you can specify this option directly in the Create statement. All this applies to some other table types, in particular to BIN tables.

Note 3: The width of the fields is the length specified in the column declaration. For instance, for a column declared as:

```
number int(3) not null,
```

The field width in the file is 3 bytes. This is the value used to calculate the offset of the next field if it is not specified. If this length is not specified, it defaults to the MySQL default type length.

Specifying the Field Format

Some files have specific format for their numeric fields. For instance, the decimal point is absent and/or the field should be filled with leading zeroes. To deal with such files, as well in reading as in writing, their format can be specified in the CREATE TABLE column definition. The syntax of the field format specification is:

```
Field_format='[Z][N|Dc][d]'
```

The optional parts of the format are:

Z The field has leading zeroes
N No decimal point exists in the file
D Specifies the decimal separator as the character *c*
d The number of decimals, defaults to the column precision

Let us see how it works in the following example. We define a table based on the file *xfmt.txt* having eight fields of 12 characters:

```
create table xfmt (
col1 double(12,3) not null,
col2 double(12,3) not null field_format='4',
col3 double(12,2) not null field_format='N3',
col4 double(12,3) not null field_format='ZD,',
col5 double(12,3) not null field_format='Z3',
```

```
col6 double(12,5) not null field_format='ZN5',
col7 int(12) not null field_format='N3',
col8 smallint(12) not null field_format='N3')
engine=CONNECT table_type=FIX file_name='xfmt.txt';
insert into xfmt values(4567.056,4567.056,4567.056,4567.056,-23456.8,
     3.14159,4567,4567);
select * from xfmt;
```

The first row is displayed as:

COL1	COL2	COL3	COL4	COL5	COL6	COL7	COL8
4567.056	4567.056	4567.06	4567.056	-23456.800	3.14159	4567	4567

The number of decimals displayed for all float columns is the column precision, the second argument of the column type option. Of course, integer columns have no decimals, although their formats specify some.

More interesting is the file layout. To see it let us define another table based on the same file but whose columns are all characters:

```
create table cfmt (
col1 char(12) not null,
col2 char(12) not null,
col3 char(12) not null,
col4 char(12) not null,
col5 char(12) not null,
col6 char(12) not null,
col7 char(12) not null,
col8 char(12) not null)
engine=CONNECT table_type=FIX file_name='xfmt.txt';
select * from cfmt;
```

The (transposed) display of the select command shows the file text layout for each field. Below a third column was added in this document to comment this result.

Column	Row 1	Comment (all numeric fields are written right justified)
COL1	4567.056	No format, the value was entered as is.
COL2	4567.0560	The format '4' forces to write 4 decimals.
COL3	4567060	N3 → No decimal point. The last 3 digits are decimals. However, the second decimal was rounded because of the column precision.
COL4	00004567,056	Z → Leading zeroes, 3 decimals (the column precision) The decimal separator is a comma (like in some European countries)
COL5	-0023456.800	Z3 → (Minus sign) leading zeroes, 3 decimals.
COL6	000000314159	ZN5 → Leading zeroes, no decimal point, 5 decimals.
COL7	4567000	N3 → No decimal point. The last 3 digits are decimals.
COL8	4567000	Same. Any decimals would be ignored.

Note: For columns internally using double precision floating-point numbers, MariaDB limits the decimal precision of any calculation to the column precision. The declared column precision should be at least the number of decimals of the format to avoid a loss of decimals as it happened for col3 of the above example.

DBF Table Type

A table of type DBF is physically a dBASE III or IV formatted file (used by many products like dBASE, Xbase, FoxPro etc.). This format is like the FIX type format with in addition a prefix giving the characteristics of the file and describing all the fields (columns) of the table.

Because DBF files have a header that contains Meta data about the file, in particular the column description, it is possible to create a table based on an existing DBF file without giving the column description, for instance:

```
create table cust engine=CONNECT table_type=DBF file_name='cust.dbf';
```

To see what CONNECT has done, you can use the DESCRIBE or SHOW CREATE TABLE commands, and eventually modify some options with the ALTER TABLE command.

The case of deleted lines is handled in a specific way for DBF tables. Deleted lines are not removed from the file but are "soft deleted" meaning they are marked as deleted. In particular, the number of lines contained in the file header does not take care of soft deleted lines. Therefore, if you execute these two commands applied to a DBF table named *tabdbf*:

```
select count(*) from tabdbf;
select count(*) from tabdbf where 1;
```

They can give a different result, the (fast) first one giving the number of physical lines in the file obtained from the header and the second one giving the number of line that are not (soft) deleted.

The commands UPDATE, INSERT, and DELETE can be used with DBF tables. The DELETE command marks the deleted lines as suppressed but keeps them in the file. The INSERT command, if it is used to populate a newly created table, constructs the file header before inserting new lines.

Note: For DBF tables, column name length is limited to 11 characters and field length to 256 bytes.

Conversion of dBASE Data Types
CONNECT handles only types that are stored as characters.

Symbol	DBF Type	CONNECT Type	Description
B	Binary (string)	TYPE_STRING	10 digits representing a .DBT block number.
C	Character	TYPE_STRING	All OEM code page characters - padded with blanks to the width of the field.
D	Date	TYPE_DATE	8 bytes - date stored as a string in the format YYYYMMDD.
N	Numeric	TYPE_INT TYPE_BIGINT TYPE_DOUBLE	Number stored as a string, right justified, and padded with blanks to the width of the field.
L	Logical	TYPE_STRING	1 byte - initialized to 0x20 otherwise T or F.
M	Memo (string)	TYPE_STRING	10 digits representing a .DBT block number.
@	Timestamp	Not supported	8 bytes - two longs, first for date, second for time. It is the number of days since 01/01/4713 BC.
I	Long	Not supported	4 bytes. Leftmost bit used to indicate sign, 0 negative.
+	Autoincrement	Not supported	Same as a Long
F	Float	TYPE_DOUBLE	Number stored as a string, right justified, and padded with blanks to the width of the field.
O	Double	Not supported	8 bytes - no conversions, stored as a double.
G	OLE	TYPE_STRING	10 digits representing a .DBT block number.

For the **N** numeric type, CONNECT converts it to TYPE_DOUBLE if the decimals value is not 0, to TYPE_BIGINT if the length value is greater than 10, else to TYPE_INT.

For **M**, **B**, and **G** types, CONNECT just returns the DBT number.

Reading Soft Deleted Lines of a DBF table
It is possible to read these lines by changing the read mode of the table. This is specified by an option READMODE that can take the values:

0: Standard mode. This is the default option.
1: Read all lines including soft deleted ones.
2: Read only the soft deleted lines.

For example, to read all lines of the *tabdbf* table, you can do:

```
alter table tabdbf option_list='Readmode=1';
```

To come back to normal mode, specify READMODE=0.

BIN Table Type

A table of type BIN is physically a binary file in which each row is a logical record of fixed length[8]. Within a record, column fields are of a fixed offset and length as with FIX tables. Specific to BIN tables is that numerical values are internally encoded using native platform representation, so no conversion is needed to handle numerical values in expressions.

It is not required that the lines of a BIN file be separated by characters such as CR and/or LF but this is possible. In such an event, the *lrecl* option must be specified accordingly.

Note: Unlike for the DOS and FIX types, the width of the fields is the length of their internal representation in the file. For instance, for a column declared as:

```
number int(5) not null,
```

The field width in the file is 4 characters, the size of a binary integer. This is the value used to calculate the offset of the next field if it is not specified. Therefore, if the next field is placed 5 characters after this one, this declaration is not enough, and the flag option must be used on the next field.

Type Conversion in BIN Tables
Here are the correspondences between the column type and field format provided by default:

Column type	File default format
Char(*n*)	Text of *n* characters.
Date	Integer (4 bytes)
Int(*n*)	Integer (4 bytes)
Smallint(*n*)	Short integer (2 bytes)
Tinyint	Char (1 Byte)
Bigint(*n*)	Large integer (8 bytes)
Double(*n*,*d*)	Double floating point (8 bytes)

However, the column type need not necessarily match the field format within the table file. This occurs for field formats that correspond to numeric types that are not handled by CONNECT[9]. Indeed, BIN table files may internally contain float numbers or binary numbers of any byte length in big-endian or little-

[8] Sometimes it can be a physical record if LF or CRLF have been written in the file.
[9] Most of these are obsolete because CONNECT supports all column types except FLOAT.

endian representation[10]. Also, as in DOS or FIX tables, you may want to handle some character fields as numeric or vice versa.

This is why it is possible to specify the field format when it does not correspond to the column type default using the *field_format* column option in the CREATE TABLE statement. Here are the available field formats for BIN tables:

Field_format	Internal representation
[*n*]{L|B|H}[*n*]	*n* bytes binary number in little endian, big endian or host endian representation.
C	Characters string (*n* bytes)
I	Integer (4 bytes)
D	Double float (8 bytes)
S	Short integer (2 bytes)
T	Tiny integer (1 byte)
G	Big integer (8 bytes)
F or **R**	Real or float (Floating point number on 4 bytes)
X	Use the default format field for the column type

All field formats except the first one are a one-character specification[11]. 'X' is equivalent to not specifying the field format. For the 'C' character specification, *n* is the column width as specified with the column type. For one character formats the number of bytes of the numeric fields corresponds to what it is on most platforms. However, it could vary for some. The **G**, **I**, **S** and **T** formats are deprecated because they correspond to supported data types and may not be supported in future versions.

Here is an example of a BIN table. The file record layout is supposed to be:

```
NNNNCCCCCCCCCCIIIISSFFFFSS
```

Here N represents numeric characters, C any characters, I integer bytes, S short integer bytes, and F float number bytes. The `IIII` field contains a date in numeric format.

The table could be created by:

```
create table testbal (
fig int(4) not null field_format='C',
name char(10) not null,
birth date not null field_format='L',
id char(5) not null field_format='L2',
salary double(9,2) not null default 0.00 field_format='F',
dept int(4) not null field_format='L2')
engine=CONNECT table_type=BIN block_size=5 file_name='Testbal.dat';
```

Specifying the little-endian representation for binary values is not useful on most machines, but makes the create table statement portable on a machine using big endian, as well as the table file.

The field offsets and the file record length being calculated according to the column internal format, eventually modified by the field format, it is not necessary to specify them for a packed binary file without line ending. If a line ending is desired, specify the *ending* option or specify the *lrecl* option adding the ending width. The table can be filled by:

```
insert into testbal values
(5500,'ARCHIBALD','1980-01-25','3789',4380.50,318),
(123,'OLIVER','1953-08-10','23456',3400.68,2158),
(3123,'FOO','2002-07-23','888',default,318);
```

[10] The default endian representation used in the table file can be specified by setting the ENDIAN option as 'L' or 'B' in the option list.
[11] It can be specified with more than one character, but only the first one is significant.

Note that the types of the inserted values must match the column type, not the field format type.

The query:

```
select * from testbal;
```

Returns:

fig	name	birth	id	salary	dept
5500	ARCHIBALD	1980-01-25	3789	4380.50	318
123	OLIVER	1953-08-10	23456	3400.68	2158
3123	FOO	2002-07-23	888	0.00	318

Numeric fields alignment
In binary files, numeric fields and record length can be aligned on 4 or 8-byte boundaries to optimize performances on certain processors. This can be modified in the OPTION_LIST with an "align" option ("packed" meaning align=1 is the default).

VEC Table Type (Vertical Partitioning)

Warning: Avoid using this table type in production applications. This file format is specific to CONNECT and may not be supported in future versions.

Tables of type VEC are binary files that in some cases can provide good performance on read-intensive query workloads. CONNECT organizes their data on disk as columns of values from the same attribute, as opposed to storing it as rows of tabular records. This organization means that when a query needs to access only a few columns of a particular table, only those columns need to be read from disk. Conversely, in a row-oriented table, all values in a table are typically read from disk, wasting I/O bandwidth.

CONNECT provides two integral VEC formats, in which each column's data is adjacent.

Integral Vector Formats
In these true vertical formats, the VEC files are made of all the data of the first column, followed by all the data of the second column etc. All this can be in one physical file or each column data can be in a separate file. In the first case, the option MAX_ROWS=*m*, where *m* is the estimate of the maximum size (number of rows) of the table, must be specified to be able to insert some new records. This leaves an empty space after each column area in which new data can be inserted. In the second case, the "Split" option can be specified[12] at table creation and each column will be stored in a file named sequentially from the table file name followed by the rank of the column. Inserting new lines can freely augment such a table.

Differences between vector formats
These formats correspond to different needs. The integral vector format provides the best performance gain. It will be chosen when the speed of decisional queries must be optimized.

In the case of a unique file, inserting new data will be limited but there will be only one open and close to do. However, the size of the table cannot be calculated from the file size because of the eventual unused space in the file. It must be kept in a header containing the maximum number of rows and the current number of valid rows in the table. To achieve this, specify the option Header=*n* when creating the table. If n=1 the header will be placed at the beginning of the file, if n=2 it will be a separate file with the type '.blk', and if n=3 the header will be place at the end of the file. This last value is provided because batch inserting is sometimes slower when the header is at the beginning of the file. If not specified, the header option will default to 2 for this table type.

[12] The SPLIT option is true by default when MAX_ROW is not specified or 0.

On the other hand, the "Split" format with separate files have none of these issues, and is a much safer solution when the table must frequently inserted or shared among several users. For instance:

```
create table vtab (
a int not null,
b char(10) not null)
engine=CONNECT table_type=VEC file_name='vt.vec';
```

This table, split by default, will have the column values in files `vt1.vec` and `vt2.vec`.

For vector tables, the option BLOCK_SIZE=*n* is used for block reading and writing; however, to have a file made of blocks of equal size, the internal value of the MAX_ROWS=*m* option is eventually increased to become a multiple of *n*.

Like for BIN tables, numeric values are stored using platform internal layout, the correspondence between column types and internal format being the same than the default ones given above for BIN. However, field formats are not available for VEC tables.

Header Option
This applies to VEC tables that are not split. Because the file size depends on the MAX_ROWS value, CONNECT cannot know how many valid records exist in the file. Depending on the value of the HEADER option, this information is stored in a header that can be placed at the beginning of the file, at the end of the file or in a separate file called *fn*.blk. The valid values for the HEADER option are:

0	Defaults to 2 for standard tables and to 3 for inward tables.
1	The header is at the beginning of the file.
2	The header is in a separate file.
3	The header is at the end of the file.

The value 2 can be used when dealing with files created by another application with no header. The value 3 makes sometimes inserting in the file faster than when the header is at the beginning of the file.

Note: VEC being a file format specific to CONNECT, no big endian / little endian conversion is provided. These files are not portable between machines using a different byte order setting.

CSV and FMT Table Types

Many source data files are formatted with variable length fields and records. The simplest format, known as CSV (Comma Separated Variables), has column fields separated by a separator character. By default, the separator is a comma but can be specified by the SEP_CHAR option as any character, for instance a semi-colon.

If the CSV file first record is the list of column names, specifying the HEADER=1 option will skip the first record on reading. On writing, if the file is empty, the column names record is automatically written.

For instance, given the following *people.csv* file:

```
Name;birth;children
"Archibald";17/05/01;3
"Nabucho";12/08/03;2
```

You can create the corresponding table by:

```
create table people (
name char(12) not null,
birth date not null date_format='DD/MM/YY',
children smallint(2) not null)
engine=CONNECT table_type=CSV file_name='people.csv'
header=1 sep_char=';' quoted=1;
```

For CSV tables, the *flag* column option is the rank of the column into the file starting from 1 for the leftmost column. This is to enable having column displayed in a different order than in the file and/or to define the table specifying only some columns of the CSV file. For instance:

```
create table people (
name char(12) not null,
children smallint(2) not null flag=3,
birth date not null flag=2 date_format='DD/MM/YY')
engine=CONNECT table_type=CSV file_name='people.csv'
header=1 sep_char=';' quoted=1;
```

In this case the command:

```
select * from people;
```

will display the table as:

name	children	birth
Archibald	3	2001-05-17
Nabucho	2	2003-08-12

Many applications produce CSV files having some fields quoted, in particular because the field text contains the separator character. For such files, specify the `'QUOTED=n'` option to indicate the level of quoting and/or the `'QCHAR=c'` to specify what is this eventual quoting character, which is " by default. Quoting with single quotes must be specified as `QCHAR=''''`. On writing, fields will be quoted depending on the value of the quoting level, which is –1 by default meaning no quoting:

0 The fields between quotes are read and the quotes discarded. On writing, fields will be quoted only if they contain the separator character or begin with the quoting character. If they contain the quoting character, it will be doubled.
1 Only text fields will be written between quotes, except null fields. This includes also the column names of an eventual header.
2 All fields will be written between quotes, except null fields.
3 All fields will be written between quotes, including null fields.

Files written this way are successfully read by most applications including spreadsheets.

Note 1: If only the QCHAR option is specified, the QUOTED option will default to 1.

Note 2: For CSV tables whose separator is the tab character, specify `sep_char='\t'`.

Note 3: When creating a table on an existing CSV file, you can leave CONNECT analyze the file and make the column description. However, this is a not an elaborate analysis of the file and, for instance, DATE fields will not be recognized as such but will be regarded as string fields.

Note 4: The FIELD_FORMAT option can be used with CSV table columns and has the same meaning than for DOS or FIX tables. However, if the decimal separator is set to a comma and the field separator is also the comma (the default if not specified) this will be ambiguous and will lead to error unless the fields are all quoted (Quoting >= 2).

Note 5: For quoted columns, the field length must include eventual quotes.

Note 6: When only some columns are defined, CONNECT cannot calculate the record length. Therefore, the LRECL option must be specified. This also applies when column names are changed and the HEADER option is true.

Bad record error processing
CSV files often contain ill-formatted records. When this happens the process aborts with a message such as:

```
Bad format line 3 field 4 of funny.txt
```

When you know that your file contains records that are ill formatted and should be eliminated from normal processing, set the "maxerr" option of the CREATE TABLE statement, for instance:

```
Option_list='maxerr=100'
```

This will indicate that no error message be raised for the 100 first wrong lines. You can set Maxerr to a number greater than the number of wrong lines in your files to ignore them and get no errors.

Additionally, the "accept" option permit to keep those ill formatted lines with the bad field, and all succeeding fields of the record, nullified. If "accept" is specified without "maxerr", all ill formatted lines will be accepted.

FMT type

FMT tables handle files of various formats that are an extension of the concept of CSV files. CONNECT supports these files providing all lines have the same format and that all fields present in all records are recognizable (optional fields must have recognizable delimiters). These files are made by specific application and CONNECT handle them in read only mode.

FMT tables must be created as CSV tables, specifying their type as FMT. In addition, each column description must be added its format specification.

Column Format Specification of FMT tables
The input format for each column is specified as a FIELD_FORMAT option. A simple example is:

```
IP Char(15) not null field_format=' %n%s%n',
```

In the above example, the format for this (1st) field is `' %n%s%n'`. Note that the blank character at the beginning of this format **is** significant. No trailing blank should be specified in the column formats.

The syntax and meaning of the column input format is the one of the C **scanf** function.

However, CONNECT uses the input format in a specific way. Instead of using it to directly store the input value in the column buffer; it uses it to delimit the sub string of the input record that contains the corresponding column value. Retrieving this value is done later by the column functions as for standard CSV files.

Therefore, all column formats are made of five components:

1. An eventual description of what is met and ignored before the column value.
2. A marker of the beginning of the column value written as %n.
3. The format specification of the column value itself.
4. A marker of the end of the column value written as %n (or %m for optional fields).
5. An eventual description of what is met after the column value (not valid is %m was used).

For example, taking the file *funny.txt*:

```
12345,'BERTRAND',#200;5009.13
 56, 'POIROT-DELMOTTE' ,#4256 ;18009
345 ,'TRUCMUCHE' , #67; 19000.25
```

You can make a table *fmtsample* with 4 columns ID, NAME, DEPNO and SALARY, using the Create Table statement and column formats:

```
create table FMTSAMPLE (
ID Integer(5) not null field_format=' %n%d%n',
NAME Char(16) not null field_format=' , ''%n%[^'']%n''',
DEPNO Integer(4) not null field_format=' , #%n%d%n',
SALARY Double(12,2) not null field_format=' ; %n%f%n')
Engine=CONNECT table_type=FMT file_name='funny.txt';
```

Field 1 is an integer (%d) with eventual leading blanks.

Field 2 is separated from field 1 by optional blanks, a comma, and other optional blanks and is between single quotes. The leading quote is included in component 1 of the column format, followed by the %n marker. The column value is specified as `%[^']` meaning to keep any characters read until a quote is met. The ending marker (%n) is followed by the 5th component of the column format, the single quote that follows the column value.

Field 3, also separated by a comma, is a number preceded by a pound sign.

Field 4, separated by a semicolon eventually surrounded by blanks, is a number with an optional decimal point (%f).

This table will be displayed as:

ID	NAME	DEPNO	SALARY
12345	BERTRAND	200	5009.13
56	POIROT-DELMOTTE	4256	18009.00
345	TRUCMUCHE	67	19000.25

Optional Fields
To be recognized, a field normally must be at least one-character long. For instance, a numeric field must have at least one digit, or a character field cannot be void. However, many existing files do not follow this format.

Let us suppose for instance that the preceding example file could be:

```
12345,'BERTRAND',#200;5009.13
 56, 'POIROT-DELMOTTE' ,# ;18009
345 ,'' , #67; 19000.25
```

This will display an error message such as “Bad format line x field y of FMTSAMPLE”. To avoid this and accept these records, the corresponding fields must be specified as “optional”. In the above example, fields 2 and 3 can have null values (in lines 3 and 2 respectively). To specify them as optional, their format must be terminated by `%m` (instead of the second `%n`). A statement such as this can do the table creation:

```
create table FMTAMPLE (
ID Integer(5) not null field_format=' %n%d%n',
NAME Char(16) not null field_format=' , ''%n%[^'']%m',
DEPNO Integer(4) field_format=''' , #%n%d%m',
SALARY Double(12,2) field_format=' ; %n%f%n')
Engine=CONNECT table_type=FMT file_name='funny.txt';
```

Note that, because the statement must be terminated by `%m` with no additional characters, skipping the ending quote of field 2 was moved from the end of the second column format to the beginning of the third column format.

The table result is:

ID	NAME	DEPNO	SALARY
12345	BERTRAND	200	5,009.13
56	POIROT-DELMOTTE	NULL	18,009.00
345	NULL	67	19,000.25

Missing fields are replaced by null values if the column is nullable, blanks for character strings and 0 for numeric fields if it is not.

Note 1:
Because the formats are specified between quotes, quotes belonging to the formats must be doubled or escaped to avoid a CREATE TABLE statement syntax error.

Note 2:
Characters separating columns can be included as well in component 5 of the preceding column format or in component 1 of the succeeding column format but for blanks, which should be always included in component 1 of the succeeding column format because line trailing blanks can be sometimes lost. This is also mandatory for optional fields.

Note 3:
Because the format is mainly used to find the sub-string corresponding to a column value, the field specification does not necessarily match the column type. For instance supposing a table contains two integer columns, NBONE and NBTWO, the two lines describing these columns could be:

```
NBONE integer(5) not null field_format=' %n%d%n',
NBTWO integer(5) field_format=' %n%s%n',
```

The first one specifies a required integer field (%d), the second line describes a field that can be an integer, but can be replaced by a "-" (or any other) character. Specifying the format specification for this column as a character field (%s) enables to recognize it with no error in all cases. Later on, this field will be converted to integer by the column read function, and a null 0 value will be generated for field specified in their format as non-numeric.

Bad record error processing
When no match if found for a column field the process aborts with a message such as:

```
Bad format line 3 field 4 of funny.txt
```

This can mean as well that one line of the input line is ill formed or that the column format for this field has been wrongly specified. When you know that your file contains records that are ill formatted and should be eliminated from normal processing, set the “maxerr” option of the CREATE TABLE statement, for instance:

```
Option_list='maxerr=100'
```

This will indicate that no error message be raised for the 100 first wrong lines. You can set Maxerr to a number greater than the number of wrong lines in your files to ignore them and get no errors.

Additionally, the “accept” option permit to keep those ill formatted lines with the bad field, and all succeeding fields of the record, nullified. If “accept” is specified without “maxerr”, all ill formatted lines will be accepted.

Fields containing a formatted Date
A special case is one of columns containing a formatted date. In this case, two formats must be specified:

1. The field recognition format used to delimit the date in the input record.
2. The date format used to interpret the date.

3. The field length option if the date representation is different than the standard type size.

For example, let us suppose we have a web log source file containing records such a:

```
165.91.215.31 - - [17/Jul/2001:00:01:13 -0400] - "GET /usnews/home.htm HTTP/1.1" 302
```

The create table statement shall be like this:

```
create table WEBSAMP (
IP char(15) not null field_format='%n%s%n',
DATE datetime not null field_format=' - - [%n%s%n -0400]'
date_format='DD/MMM/YYYY:hh:mm:ss' field_length=20,
FILE char(128) not null field_format=' - "GET %n%s%n',
HTTP double(4,2) not null field_format=' HTTP/%n%f%n"',
NBONE int(5) not null field_format=' %n%d%n')
Engine=CONNECT table_type=FMT lrecl=400
file_name='e:\\data\\token\\Websamp.dat';
```

Note 1: Here, `field_length=20` was necessary because the default size for datetime columns is only 19. The `lrecl=400` was also specified because the actual file contains more information in each record making the record size calculated by default too small.

Note 2: The file name could have been specified as '`e:/data/token/Websamp.dat`'.

Note 3: FMT tables are currently read only.

NoSQL Table Types

They are based on files that do not match the relational format but often represent hierarchical data. CONNECT can handle JSON, INI-CFG, XML, and some HTML files.

The way it is done is different from what MySQL or PostgreSQL does. In addition to including in a table some column values of a specific data format (JSON, XML) to be handled by specific functions, CONNECT can directly uses JSON, XML or INI files that can be produced by other applications and this is the table definition that describes where and how the contained information must be retrieved.

This is also different from what MariaDB does with dynamic columns, which by the way is close from what MySQL and PostgreSQL do with the JSON column type.

XML Table Type

CONNECT supports tables represented by XML files. For these tables, the standard input/output functions of the operating system are not used but the parsing and processing of the file is delegated to a specialized library. Currently two such systems are supported: libxml2, a part of the GNOME framework, but which does not require GNOME and, on Windows, MS-DOM (DOMDOC) the Microsoft standard support of XML documents.

DOMDOC is the default for the Windows version of CONNECT and libxml2 is always used on other systems. On Windows, the choice can be specified using the XMLSUP create table list option, for instance specifying `option_list='xmlsup=libxml2'`.

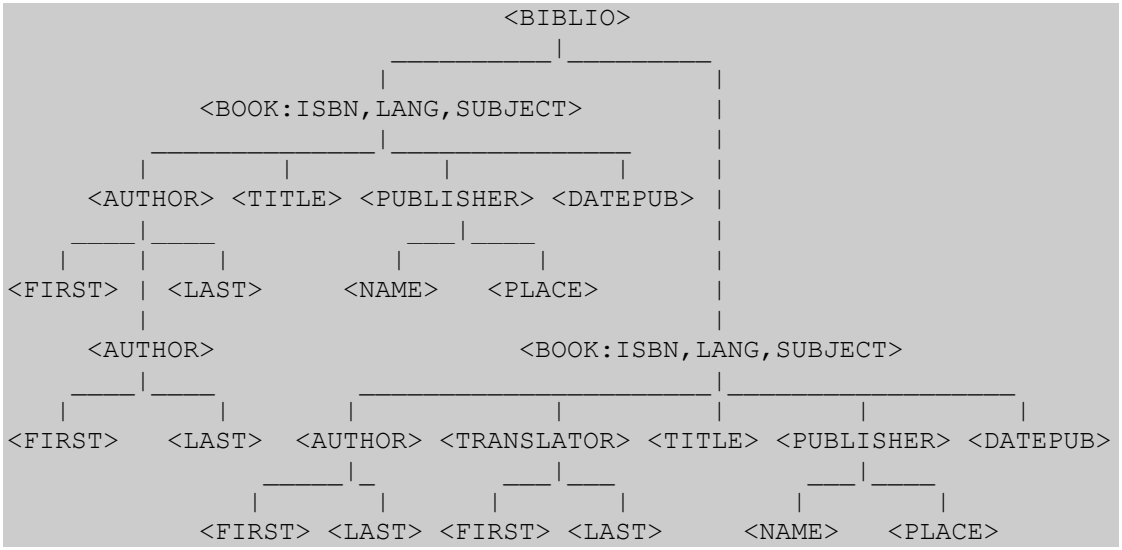
Creating XML tables

First, it must be understood that XML is a very general language used to encode data having any structure. In particular, the tag hierarchy in an XML file describes a tree structure of the data. For instance, consider the file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<BIBLIO SUBJECT="XML">
   <BOOK ISBN="9782212090819" LANG="fr" SUBJECT="applications">
      <AUTHOR>
         <FIRSTNAME>Jean-Christophe</FIRSTNAME>
         <LASTNAME>Bernadac</LASTNAME>
      </AUTHOR>
      <AUTHOR>
         <FIRSTNAME>François</FIRSTNAME>
         <LASTNAME>Knab</LASTNAME>
      </AUTHOR>
      <TITLE>Construire une application XML</TITLE>
      <PUBLISHER>
         <NAME>Eyrolles</NAME>
         <PLACE>Paris</PLACE>
      </PUBLISHER>
      <DATEPUB>1999</DATEPUB>
   </BOOK>
   <BOOK ISBN="9782840825685" LANG="fr" SUBJECT="applications">
      <AUTHOR>
         <FIRSTNAME>William J.</FIRSTNAME>
         <LASTNAME>Pardi</LASTNAME>
      </AUTHOR>
      <TRANSLATOR PREFIX="adapté de l'anglais par">
         <FIRSTNAME>James</FIRSTNAME>
         <LASTNAME>Guerin</LASTNAME>
      </TRANSLATOR>
      <TITLE>XML en Action</TITLE>
      <PUBLISHER>
         <NAME>Microsoft Press</NAME>
         <PLACE>Paris</PLACE>
```

```
      </PUBLISHER>
      <DATEPUB>1999</DATEPUB>
   </BOOK>
</BIBLIO>
```

It represents data having the structure:



This structure seems at first view far from being tabular. However, modern database management systems, including MariaDB, implement something close to the relational model and work on tables that are structurally not hierarchical but tabular with rows and columns.

Nevertheless, CONNECT can do it. Of course, it cannot guess what you want to extract from the XML ~~structure, but~~structure but gives you the possibility to specify it when you create the table[13].

Let us take a first example. Suppose you want to make a table from the above document, displaying the node contents.

For this, you can define a table *xsamptag* as:

```
create table xsamptag (
AUTHOR char(50),
TITLE char(32),
TRANSLATOR char(40),
PUBLISHER char(40),
DATEPUB int(4))
engine=CONNECT table_type=XML file_name='Xsample.xml';
```

It will be displayed as:

AUTHOR	TITLE	TRANSLATOR	PUBLISHER	DATEPUB
Jean-Christophe Bernadac	Construire une application XML	~~NULL~~<null>	Eyrolles Paris	1999
William J. Pardi	XML en Action	James Guerin	Microsoft Press Paris	1999

Let us try to understand what happened. By default, the columns names correspond to tag names. Because this file is rather simple, CONNECT could default the top tag of the table as the root node <BIBLIO> of the file, and the row tags as the <BOOK> children of the table tag. In a more complex file, this should have been specified, as we will see later. Note that we didn't have to worry about the sub-tags such as

[13] CONNECT does not claim to be able to deal with any XML document. Besides, those that can usefully be processed for data analysis are likely to have a structure that can easily be transformed into a table.

<FIRSTNAME> or <LASTNAME> because CONNECT automatically retrieves the entire text contained in a tag and its sub-tags[14].

Only the first author of the first book appears. This is because only the first occurrence of a column tag has been retrieved so the result has a proper tabular structure. We will see later what we can do about that.

How can we retrieve the values specified by attributes? By using a *Coltype* table option to specify the default column type. The value '@' means that column names match attribute names. Therefore, we can retrieve them by creating a table such as:

```
create table xsampattr (
ISBN char(15),
LANG char(2),
SUBJECT char(32))
engine=CONNECT table_type=XML file_name='Xsample.xml'
option_list='Coltype=@';
```

This table returns the following:

ISBN	LANG	SUBJECT
9782212090819	fr	applications
9782840825685	fr	applications

Now to define a table that will give us all the previous information, we must specify the column type for each column. Because in the next statement the column type defaults to Node, the *field_format* column parameter was used to indicate which columns are attributes:

```
create table xsamp (
ISBN char(15) field_format='@',
LANG char(2) field_format='@',
SUBJECT char(32) field_format='@',
AUTHOR char(50),
TITLE char(32),
TRANSLATOR char(40),
PUBLISHER char(40),
DATEPUB int(4))
engine=CONNECT table_type=XML file_name='Xsample.xml'
tabname='BIBLIO' option_list='rownode=BOOK';
```

Once done, we can enter the query:

```
select subject, lang, title, author from xsamp;
```

This will return the following result:

SUBJECT	LANG	TITLE	AUTHOR
applications	fr	Construire une application XML	Jean-Christophe Bernadac
applications	fr	XML en Action	William J. Pardi

Note that we have been lucky. Because unlike SQL, XML is case sensitive and the column names have matched the node names only because the column names were given in upper case. Note also that the order of the columns in the table could have been different from the order in which the nodes appear in the XML file.

[14] With libxml2, sub tags text can be separated by 0 or several blanks depending on the structure and indentation of the data file.

Using Xpath's with XML tables

Xpath is used by XML to locate and retrieve nodes. The table's main node Xpath is specified by the TABNAME option. If just the node name is given, CONNECT constructs an Xpath such as '//BIBLIO' in the example above that should retrieve the BIBLIO node wherever it is within the XML file.

The row nodes are by default the children of the table node. However, for instance to eliminate some children nodes that are not real row nodes, the row node name can be specified using the ROWNODE sub-option of the OPTION_LIST option.

The field_format options we used above can be specified to locate more precisely where and what information to retrieve using an Xpath-like syntax. For instance:

```
create table xsampall (
isbn char(15) field_format='@ISBN',
language char(2) field_format='@LANG',
subject char(32) field_format='@SUBJECT',
authorfn char(20) field_format='AUTHOR/FIRSTNAME',
authorln char(20) field_format='AUTHOR/LASTNAME',
title char(32) field_format='TITLE',
translated char(32) field_format='TRANSLATOR/@PREFIX',
tranfn char(20) field_format='TRANSLATOR/FIRSTNAME',
tranln char(20) field_format='TRANSLATOR/LASTNAME',
publisher char(20) field_format='PUBLISHER/NAME',
location char(20) field_format='PUBLISHER/PLACE',
year int(4) field_format='DATEPUB')
engine=CONNECT table_type=XML file_name='Xsample.xml'
tabname='BIBLIO' option_list='rownode=BOOK';
```

This very flexible column parameter serves several purposes:

- To specify the tag name, or the attribute name if different from the column name.
- To specify the type (tag or attribute) by a prefix of '@' for attributes.
- To specify the path for sub-tags using the '/' character.

This path is always relative to the current context (the column top node) and cannot be specified as an absolute path from the document root, therefore a leading '/' cannot be used. The path cannot be variable in node names or depth, therefore using '//' is not allowed.

The query:

```
select isbn, title, translated, tranfn, tranln, location from
    xsampall where translated is not null;
```

replies:

ISBN	TITLE	TRANSLATED	TRANFN	TRANLN	LOCATION
9782840825685	XML en Action	adapté de l'anglais par	James	Guerin	Paris

Libxml2 default name space issue

An issue with libxml2 is that some files can declare a default name space in their root node. Because Xpath only searches in that name space, the nodes will not be found if they are not prefixed. If this happens, specify the TABNAME option as an Xpath ignoring the current name space:

```
TABNAME="//*[local-name()='BIBLIO']"
```

This must also be done for the default of specified Xpath of the not attribute columns. For instance:

```
title char(32) field_format="*[local-name()='TITLE']",
```

Note: This raises an error (and is useless anyway) with DOMDOC.

Direct access on XML tables
Direct access is available on XML tables. This means that XML tables can be sorted and used in joins, even in the one-side of the join.

However, building a permanent index is not yet implemented. It is unclear whether this can be useful. Indeed, the DOM implementation that is used to access these tables firstly parses the whole file and constructs a node tree in memory. This may often be the longest part of the process, so the use of an index would not be of great value. Note also that this limits the XML files to a reasonable size. Anyway, when speed is important, this table type is not the best to use. Therefore, in these cases, it is probably better to convert the file to another type by inserting the XML table in another table of a more appropriate type for performance.

Accessing tags with namespaces
With the Windows DOMDOC support, this can be done using the prefix in the tabname column option and/or field_format column option. For instance, given the file gns.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<gpx xmlns:gns="http:dummy">
<gns:trkseg>
<trkpt lon="-121.9822235107421875" lat="37.3884925842285156">
<gns:ele>6.610851287841797</gns:ele>
<time>2014-04-01T14:54:05.000Z</time>
</trkpt>
<trkpt lon="-121.9821929931640625" lat="37.3885803222656250">
<ele>6.787827968597412</ele>
<time>2014-04-01T14:54:08.000Z</time>
</trkpt>
<trkpt lon="-121.9821624755859375" lat="37.3886299133300781">
<ele>6.771987438201904</ele>
<time>2014-04-01T14:54:10.000Z</time>
</trkpt>
</gns:trkseg>
</gpx>
```

and the defined CONNECT table:

```
CREATE TABLE xgns (
`lon` double(21,16) NOT NULL `field_format`='@',
`lat` double(20,16) NOT NULL `field_format`='@',
`ele` double(21,16) NOT NULL `field_format`='gns:ele',
`time` datetime date_format="YYYY-MM-DD 'T' hh:mm:ss '.000Z'"
) ENGINE=CONNECT DEFAULT CHARSET=latin1 `table_type`=XML
     `file_name`='gns.xml' tabname='gns:trkseg'
     option_list='xmlsup=domdoc';
```

```
select * from xgns;
```

Displays:

lon	lat	ele	time
-121,982223510742	37,3884925842285	6,6108512878418	01/04/2014 14:54:05
-121,982192993164	37,3885803222656	0	01/04/2014 14:54:08
-121,982162475586	37,3886299133301	0	01/04/2014 14:54:10

Only the prefixed 'ele' tag is recognized.

However, this does not work with the libxml2 support. The solution is then to use a function ignoring the name space:

```
CREATE TABLE xgns2 (
`lon` double(21,16) NOT NULL `field_format`='@',
`lat` double(20,16) NOT NULL `field_format`='@',
`ele` double(21,16) NOT NULL `field_format`="*[local-name()='ele']",
`time` datetime date_format="YYYY-MM-DD 'T' hh:mm:ss '.000Z'"
) ENGINE=CONNECT DEFAULT CHARSET=latin1 `table_type`=XML
     `file_name`='gns.xml' tabname="*[local-name()='trkseg']"
    option_list='xmlsup=libxml2';
```

Then :

```
select * from xgns2;
```

Displays:

lon	lat	ele	time
-121,982223510742	37,3884925842285	6,6108512878418	01/04/2014 14:54:05
-121,982192993164	37,3885803222656	6.7878279685974	01/04/2014 14:54:08
-121,982162475586	37,3886299133301	6.7719874382019	01/04/2014 14:54:10

This time, all 'ele` tags are recognized. This solution does not work with DOMDOC.

Having Columns defined by Discovery

It is possible to let the MariaDB discovery process do the job of column specification. When columns are not defined in the CREATE TABLE statement, CONNECT endeavors to analyze the XML file and to provide the column specifications. This is possible only for true XML tables, but not for HTML tables.

For instance, the *xsamp* table could have been created specifying:

```
create table xsamp
engine=CONNECT table_type=XML file_name='Xsample.xml'
tabname='BIBLIO' option_list='rownode=BOOK';
```

Let's check how it was actually specified using the SHOW CREATE TABLE statement:

```
CREATE TABLE `xsamp` (
  `ISBN` char(13) NOT NULL `FIELD_FORMAT`='@',
  `LANG` char(2) NOT NULL `FIELD_FORMAT`='@',
  `SUBJECT` char(12) NOT NULL `FIELD_FORMAT`='@',
  `AUTHOR` char(24) NOT NULL,
  `TRANSLATOR` char(12) DEFAULT NULL,
  `TITLE` char(30) NOT NULL,
  `PUBLISHER` char(21) NOT NULL,
  `DATEPUB` char(4) NOT NULL
```

```
) ENGINE=CONNECT DEFAULT CHARSET=latin1 `TABLE_TYPE`='XML'
`FILE_NAME`='E:/Data/Xml/Xsample.xml' `TABNAME`='BIBLIO'
`OPTION_LIST`='rownode=BOOK';
```

It is equivalent except for the column sizes that have been calculated from the file as the maximum length of the corresponding column when it was a normal value. Also, all columns are specified as type CHAR because XML does not provide information about the node content data type. Nullable is set to true if the column is missing in some rows.

If a more complex definition is desired, you can ask CONNECT to analyse the XPATH up to a given level using the LEVEL option in the option list. The level value is the number of nodes that are taken in the XPATH. For instance:

```
create table xsampall
engine=CONNECT table_type=XML file_name='Xsample.xml'
tabname='BIBLIO' option_list='rownode=BOOK,Level=1';
```

This will define the table as:

```
CREATE TABLE `xsampall` (
  `ISBN` char(13) NOT NULL `FIELD_FORMAT`='@',
  `LANG` char(2) NOT NULL `FIELD_FORMAT`='@',
  `SUBJECT` char(12) NOT NULL `FIELD_FORMAT`='@',
  `AUTHOR_FIRSTNAME` char(15) NOT NULL
`FIELD_FORMAT`='AUTHOR/FIRSTNAME',
  `AUTHOR_LASTNAME` char(8) NOT NULL
`FIELD_FORMAT`='AUTHOR/LASTNAME',
  `TRANSLATOR_PREFIX` char(24) DEFAULT NULL
`FIELD_FORMAT`='TRANSLATOR/@PREFIX',
  `TRANSLATOR_FIRSTNAME` char(7) DEFAULT NULL
`FIELD_FORMAT`='TRANSLATOR/FIRSTNAME',
  `TRANSLATOR_LASTNAME` char(6) DEFAULT NULL
`FIELD_FORMAT`='TRANSLATOR/LASTNAME',
  `TITLE` char(30) NOT NULL,
  `PUBLISHER_NAME` char(15) NOT NULL
`FIELD_FORMAT`='PUBLISHER/NAME',
  `PUBLISHER_PLACE` char(5) NOT NULL
`FIELD_FORMAT`='PUBLISHER/PLACE',
  `DATEPUB` char(4) NOT NULL
) ENGINE=CONNECT DEFAULT CHARSET=latin1 `TABLE_TYPE`='XML'
`FILE_NAME`='Xsample.xml' `TABNAME`='BIBLIO'
`OPTION_LIST`='rownode=BOOK,Level=1';
```

This method can be used as a quick way to make a “template” table definition that can later be edited to make the desired definition. In particular, column names are constructed from all the nodes of their path in order to have distinct column names. This can be manually edited to have the desired names, provided their XPATH is not modified.

To have a preview of how columns will be defined, you can use a catalog table like this:

```
create table xsacol
engine=CONNECT table_type=XML file_name='Xsample.xml'
tabname='BIBLIO' option_list='rownode=BOOK,Level=1' catfunc=col;
```

And when asking:

```
select column_name Name, type_name Type, column_size Size,
nullable, xpath from xsacol;
```

You get the description of what the table columns will be:

Name	Type	Size	Nullable	Xpath
ISBN	CHAR	13	0	@
LANG	CHAR	2	0	@
SUBJECT	CHAR	12	0	@
AUTHOR_FIRSTNAME	CHAR	15	0	AUTHOR/FIRSTNAME
AUTHOR_LASTNAME	CHAR	8	0	AUTHOR/LASTNAME
TRANSLATOR_PREFIX	CHAR	24	1	TRANSLATOR/@PREFIX
TRANSLATOR_FIRSTNAME	CHAR	7	1	TRANSLATOR/FIRSTNAME
TRANSLATOR_LASTNAME	CHAR	6	1	TRANSLATOR/LASTNAME
TITLE	CHAR	30	0	
PUBLISHER_NAME	CHAR	15	0	PUBLISHER/NAME
PUBLISHER_PLACE	CHAR	5	0	PUBLISHER/PLACE
DATEPUB	CHAR	4	0	

Write operations on XML tables

You can freely use the Update, Delete and Insert commands with XML tables. However, you must understand that the format of the updated or inserted data follows the specifications of the table you created, not the ones of the original source file. For instance, let us suppose we insert a new book using the *xsamp* table (not the *xsampall* table) with the command:

```
insert into xsamp
(isbn, lang, subject, author, title, publisher,datepub)
values('9782212090529','fr','général','Alain Michard',
'XML, Langage et Applications','Eyrolles Paris',1998);
```

Then if we ask:

```
select subject, author, title, translator, publisher from xsamp;
```

Everything seems correct when we get the result:

SUBJECT	AUTHOR	TITLE	TRANSLATOR	PUBLISHER
applications	Jean-Christophe Bernadac	Construire une application XML	NULL	Eyrolles Paris
applications	William J. Pardi	XML en Action	James Guerin	Microsoft Press Paris
général	Alain Michard	XML, Langage et Applications	NULL	Eyrolles Paris

However, if we enter the apparently equivalent query on the *xsampall* table, based on the same file:

```
select subject,
concat(authorfn, ' ', authorln) author , title,
concat(tranfn, ' ', tranln) translator,
concat(publisher, ' ', location) publisher from xsampall;
```

this returns an apparently wrong answer:

SUBJECT	AUTHOR	TITLE	TRANSLATOR	PUBLISHER
applications	Jean-Christophe Bernadac	Construire une application XML		Eyrolles Paris
applications	William J. Pardi	XML en Action	James Guerin	Microsoft Press Paris
général		XML, Langage et Applications		

What happened here? Simply, because we used the *xsamp* table to do the Insert, what has been inserted within the XML file had the structure described for *xsamp*:

```
   <BOOK ISBN="9782212090529" LANG="fr" SUBJECT="général">
      <AUTHOR>Alain Michard</AUTHOR>
      <TITLE>XML, Langage et Applications</TITLE>
      <TRANSLATOR></TRANSLATOR>
      <PUBLISHER>Eyrolles Paris</PUBLISHER>
      <DATEPUB>1998</DATEPUB>
   </BOOK>
```

CONNECT cannot "invent" sub-tags that are not part of the *xsamp* table. Because these sub-tags do not exist, the *xsampall* table cannot retrieve the information that should be attached to them. If we want to be able to query the XML file by all the defined tables, the correct way to insert a new book to the file is to use the *xsampall* table, the only one that addresses all the components of the original document:

```
delete from xsamp where isbn = '9782212090529';

insert into xsampall (isbn, language, subject, authorfn, authorln,
title, publisher, location, year)
values('9782212090529','fr','général','Alain','Michard',
'XML, Langage et Applications','Eyrolles','Paris',1998);
```

Now the added book, in the XML file, will have the required structure:

```
   <BOOK ISBN="9782212090529" LANG="fr" SUBJECT="général"
      <AUTHOR>
         <FIRSTNAME>Alain</FIRSTNAME>
         <LASTNAME>Michard</LASTNAME>
      </AUTHOR>
      <TITLE>XML, Langage et Applications</TITLE>
      <PUBLISHER>
         <NAME>Eyrolles</NAME>
         <PLACE>Paris</PLACE>
      </PUBLISHER>
      <DATEPUB>1998</DATEPUB>
   </BOOK>
```

Note: We used a column list in the Insert statements when creating the table, to avoid generating a <TRANSLATOR> node with sub-nodes, all containing null values (this works on Windows only)

Multiple Nodes in the XML Document

Let us come back to the above example XML file. We have seen that the author node can be "multiple" meaning that there can be more than one author of a book. What can we do to get the complete information fitting the relational model? CONNECT provides you with two ~~possibilities, but~~possibilities but restricted to only one such multiple node per table.

The first and most challenging one is to return as many rows than there are authors, the other columns being repeated as if we had make a join between the author column and the rest of the table. To achieve this, simply specify the "multiple" node name and the "expand" option when creating the table. For instance, we can create the *xsamp2* table like this:

```
create table xsamp2 (
ISBN char(15) field_format='@',
LANG char(2) field_format='@',
SUBJECT char(32) field_format='@',
AUTHOR char(40),
TITLE char(32),
TRANSLATOR char(32),
```

```
PUBLISHER char(32),
DATEPUB int(4))
engine=CONNECT table_type=XML file_name='Xsample.xml'
tabname='BIBLIO'
option_list='rownode=BOOK,Expand=1,Mulnode=AUTHOR,Limit=2';
```

In this statement, the Limit option specifies the maximum number of values that will be expanded. If not specified it defaults to 10. Any values above the limit will be ignored and a warning message issued[15]. Now you can enter a query such as:

```
select isbn, subject, author, title from xsamp2;
```

This will retrieve and display the following result:

ISBN	SUBJECT	AUTHOR	TITLE
9782212090819	applications	Jean-Christophe Bernadac	Construire une application XML
9782212090819	applications	François Knab	Construire une application XML
9782840825685	applications	William J. Pardi	XML en Action
9782212090529	général	Alain Michard	XML, Langage et Applications

In this case, this is as if the table had four rows. However if we enter the query:

```
select isbn, subject, title, publisher from xsamp2;
```

this time the result will be:

ISBN	SUBJECT	TITLE	PUBLISHER
9782212090819	applications	Construire une application XML	Eyrolles Paris
9782840825685	applications	XML en Action	Microsoft Press Paris
9782212090529	général	XML, Langage et Applications	Eyrolles Paris

Because the author column does not appear in the query, the corresponding row was not expanded. This is somewhat strange because this would have been different if we had been working on a table of a different type. However, it is closer to the relational model for which there should not be two identical rows (tuples) in a table. Nevertheless, you should be aware of this somewhat erratic behavior. For instance:

```
select count(*) from xsamp2;                    /* Replies 4—3 */
select count(author) from xsamp2;               /* Replies 4 */
select count(isbn) from xsamp2;                 /* Replies 3 */
select isbn, subject, title, publisher from xsamp2 where
    author <> '';
```

This last query replies:

ISBN	SUBJECT	TITLE	PUBLISHER
9782212090819	applications	Construire une application XML	Eyrolles Paris
9782212090819	applications	Construire une application XML	Eyrolles Paris
9782840825685	applications	XML en Action	Microsoft Press Paris
9782212090529	général	XML, Langage et Applications	Eyrolles Paris

[15] This may cause some rows to be lost because an eventual WHERE clause on the “multiple” column is applied only on the limited number of retrieved rows.

Even though the author column does not appear in the result, the corresponding row was expanded because the multiple column was used in the where clause.

Intermediate Multiple Node

The "multiple" node can be an intermediate node. If we want to do the same expanding with the *xsampall* table, there will be nothing more to do. The *xsampall2* table can be created with:

```
create table xsampall2 (
isbn char(15) field_format='@ISBN',
language char(2) field_format='@LANG',
subject char(32) field_format='@SUBJECT',
authorfn char(20) field_format='AUTHOR/FIRSTNAME',
authorln char(20) field_format='AUTHOR/LASTNAME',
title char(32) field_format='TITLE',
translated char(32) field_format='TRANSLATOR/@PREFIX',
tranfn char(20) field_format='TRANSLATOR/FIRSTNAME',
tranln char(20) field_format='TRANSLATOR/LASTNAME',
publisher char(20) field_format='PUBLISHER/NAME',
location char(20) field_format='PUBLISHER/PLACE',
year int(4) field_format='DATEPUB')
engine=CONNECT table_type=XML file_name='Xsample.xml'
tabname='BIBLIO'
option_list='rownode=BOOK,Expand=1,Mulnode=AUTHOR,Limit=2';
```

The only difference is that the "multiple" node is an intermediate node in the path. The resulting table can be seen with a query such as:

```
select subject, language lang, title, authorfn first, authorln
    last, year from xsampall2;
```

This query displays:

SUBJECT	LANG	TITLE	FIRST	LAST	YEAR
applications	fr	Construire une application XML	Jean-Christophe	Bernadac	1999
applications	fr	Construire une application XML	François	Knab	1999
applications	fr	XML en Action	William J.	Pardi	1999
général	fr	XML, Langage et Applications	Alain	Michard	1998

These composite tables, half array half tree, reserve some surprises for us when updating, deleting from or inserting into them. Insert just cannot generate this structure; if two rows are inserted with just a different author, two book nodes will be generated in the XML file. Delete always deletes one book node and all its children nodes even if specified against only one author. Update is more complicated:

```
update xsampall2 set authorfn = 'Simon' where authorln = 'Knab';
update xsampall2 set year = 2002 where authorln = 'Bernadac';
update xsampall2 set authorln = 'Mercier' where year = 2002;
```

After these three updates, the first ~~one~~ two responding "Affected rows: ~~0~~1" and the ~~two others~~last one responding "Affected rows: ~~1~~2", the last query answers:

subjectSUBJECT	langLANG	titleTITLE	firstFIRST	lastLAST	yearYEAR
applicationsapplications	frfr	Construire une application XMLConstruire une application XML	Jean-ChristopheJean-Christophe	MercierMercier	20022002
applicationsapplications	frfr	Construire une application XMLConstruire une application XML	SimonFrançois	MercierKnab	20022002
applicationsapplications	frfr	XML en ActionXML en Action	William J.William J.	PardiPardi	19991999
généralgénéral	frfr	XML, Langage et ApplicationsXML, Langage et Applications	AlainAlain	MichardMichard	19981998

What must be understood here is that the Update modifies node values in the XML file, not cell values in the relational table. The first update did not worked as expected, unable to retrieve the first name node of the second author and changing it to a new value.normally The second update changed the year value of the book and this shows for the two expanded rows because there is only one DATEPUB node for that book. Because the third update applies to all rows having a certain date value, this row was retrieved but not expanded because no author data appeared in the Where clause; consequently only the firstboth author names was were updated.

Making a List of Multiple Values

Another way to see multiple values is to ask CONNECT to make a comma separated list of the multiple node values. This time, it can only be done if the "multiple" node is not intermediate. For example, we can modify the *xsamp2* table definition by:

```
alter table xsamp2 option_list='rownode=BOOK,Mulnode=AUTHOR,Limit=3';
```

This time 'Expand' is not specified, and Limit gives the maximum number of items in the list. Now if we enter the query:

```
select isbn, subject, author "AUTHOR(S)", title from xsamp2;
```

we will get the following result:

ISBN	SUBJECT	AUTHOR(S)	TITLE
9782212090819	applications	Jean-Christophe Bernadac, François Knab	Construire une application XML
9782840825685	applications	William J. Pardi	XML en Action
9782212090529	général	Alain Michard	XML, Langage et Applications

Note that updating the "multiple" column is not possible because CONNECT does not know which of the nodes to update.

This could not have been done with the *xsampall2* table because the author node is intermediate in the path, and making two lists, one of first names and another one of last names would not make sense anyway.

What if a table contains several multiple nodes
This can be handled by creating several tables on the same file, each containing only one multiple node and constructing the desired result using joins.

Support of HTML Tables

Most tables included in HTML documents cannot be processed by CONNECT because the HTML language is often not compatible with the syntax of XML. In particular, XML requires all open tags to

be matched by a closing tag while it is sometimes optional in HTML. This is often the case concerning column tags.

However, you can meet tables that respect the XML syntax but have some of the features of HTML tables. For instance:

```
<?xml version="1.0"?>
<Beers>
  <table>
    <th><td>Name</td><td>Origin</td><td>Description</td></th>
    <tr>
      <td><brandName>Huntsman</brandName></td>
      <td><origin>Bath, UK</origin></td>
      <td><details>Wonderful hop, light alcohol</details></td>
    </tr>
    <tr>
      <td><brandName>Tuborg</brandName></td>
      <td><origin>Danmark</origin></td>
      <td><details>In small bottles</details></td>
    </tr>
  </table>
</Beers>
```

Here the different column tags are included in `<td></td>` tags as for HTML tables. You cannot just add this tag in the Xpath of the columns, because the search is done on the first occurrence of each tag, and this would cause this search to fail for all columns except the first one. This case is handled by specifying the *Colnode* table option that gives the name of these column tags, for example:

```
create table beers (
`Name` char(16) field_format='brandName',
`Origin` char(16) field_format='origin',
`Description` char(32) field_format='details')
engine=CONNECT table_type=XML file_name='beers.xml'
tabname='table' option_list='rownode=tr,colnode=td';
```

The table will be displayed as:

Name	Origin	Description
Huntsman	Bath, UK	Wonderful hop, light alcohol
Tuborg	Danmark	In small bottles

However, you can deal with tables even closer to the HTML model. For example the *coffee.htm* file:

```
<TABLE summary="This table charts the number of cups of coffe
                consumed by each senator, the type of coffee (decaf
                or regular), and whether taken with sugar.">
  <CAPTION>Cups of coffee consumed by each senator</CAPTION>
  <TR>
    <TH>Name</TH>
    <TH>Cups</TH>
    <TH>Type of Coffee</TH>
    <TH>Sugar?</TH>
  </TR>
  <TR>
    <TD>T. Sexton</TD>
    <TD>10</TD>
    <TD>Espresso</TD>
    <TD>No</TD>
  </TR>
```

```
  <TR>
    <TD>J. Dinnen</TD>
    <TD>5</TD>
    <TD>Decaf</TD>
    <TD>Yes</TD>
  </TR>
</TABLE>
```

Here, column values are directly represented by the TD tag text. You cannot declare them as tags nor as attributes. In addition, they are not located using their name but by their position within the row. Here is how to declare such a table to CONNECT:

```
create table coffee (
`Name` char(16),
`Cups` int(8),
`Type` char(16),
`Sugar` char(4))
engine=connect table_type=XML file_name='coffee.htm'
tabname='TABLE' header=1 option_list='Coltype=HTML';
```

You specify the fact that columns are located by position by setting the *Coltype* option to 'HTML'. Each column position (0 based) will be the value of the *flag* column parameter that is set by default in sequence. Now we are able to display the table:

Name	Cups	Type	Sugar
T. Sexton	10	Espresso	No
J. Dinnen	5	Decaf	Yes

Note 1: We specified '`header=n`' in the create statement to indicate that the first *n* rows of the table are not data rows and should be skipped.

Note 2: In this last example, we did not specify the node names using the *Rownode* and *Colnode* options because when *Coltype* is set to 'HTML' they default to `'Rownode=TR' and 'Colnode=TD'`.

Note 3: The *Coltype* option is a word only the first character of which is significant. Recognized values are:

T(ag) or N(ode)	Column names match a tag name (the default).
A(ttribute) or @	Column names match an attribute name.
H(tml) or C(ol) or P(os)	Column are retrieved by their position.

New file setting

Some create options are used only when creating a table on a new file, i. e. when inserting into a file that does not exist yet. When specified, the 'Header' option will create a header row with the name of the table columns. This is chiefly useful for HTML tables to be displayed on a web browser.

Some new list-options are used in this context:

Option	Description
Encoding	The encoding of the new document, defaulting to UTF-8.
Attribute	A list of 'attname=attvalue' separated by ';' to add to the table node.
HeadAttr	An attribute list to be added to the header row node.

Let us see for instance, the following create statement:

```
create table handlers (
handler char(64),
```

```
version char(20),
author char(64),
description char(255),
maturity char(12))
engine=CONNECT      table_type=XML      file_name='handlers.htm'
tabname='TABLE' header=yes
option_list='coltype=HTML,encoding=ISO-8859-1,
attribute=border=1;cellpadding=5,headattr=bgcolor=yellow';
```

Supposing the table file does not exist yet, the first insert into that table, for instance by the following statement:

```
insert into handlers select plugin_name, plugin_version,
plugin_author, plugin_description, plugin_maturity from
information_schema.plugins where plugin_type = 'DAEMON';
```

will generate the following file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- Created by CONNECT Version 1.01.0008 August 18, 2013 -->
<TABLE border="1" cellpadding="5">
  <TR bgcolor="yellow">
    <TH>handler</TH>
    <TH>version</TH>
    <TH>author</TH>
    <TH>description</TH>
    <TH>maturity</TH>
  </TR>
  <TR>
    <TD>Maria</TD>
    <TD>1.5</TD>
    <TD>Monty Program Ab</TD>
    <TD>Compatibility aliases for the Aria engine</TD>
    <TD>Gamma</TD>
  </TR>
</TABLE>
```

This file can be used to display the table on a web browser (encoding should be ISO-8859-x)

handler	version	author	description	maturity
Maria	1.5	Monty Program Ab	Compatibility aliases for the Aria engine	Gamma

Note: The XML document encoding is generally specified in the XML header node, and can be different from the DATA_CHARSET, which is always UTF-8 for XML tables. Therefore the table DATA_CHARSET character set should be unspecified, or specified as UTF8. The Encoding specification is useful only for new XML files and ignored for existing files having their encoding already specified in the header node.

JSON Table Type

JSON (JavaScript Object Notation) is a lightweight data-interchange format widely used on the Internet. Many applications, generally written in JavaScript or PHP use and produce JSON data, which are exchanged as files of different physical format.

It is also possible to query, create or update such information in a database like manner. MongoDB does it using a JavaScript like language. PostgreSQL includes this facility by using a specific data type and related functions alike dynamic columns.

The CONNECT engine adds this facility to MariaDB by supporting tables based on JSON data files. This is done like for XML tables by creating tables describing what should be retrieved from the file and how it should be processed.

Let us start from the file "*biblio3.json*" that is the JSON equivalent of the XML *Xsample* file we have described in the XML table chapter:

```
[
  {
    "ISBN": "9782212090819",
    "LANG": "fr",
    "SUBJECT": "applications",
    "AUTHOR": [
      {
        "FIRSTNAME": "Jean-Christophe",
        "LASTNAME": "Bernadac"
      },
      {
        "FIRSTNAME": "François",
        "LASTNAME": "Knab"
      }
    ],
    "TITLE": "Construire une application XML",
    "PUBLISHER": {
      "NAME": "Eyrolles",
      "PLACE": "Paris"
    },
    "DATEPUB": 1999
  },
  {
    "ISBN": "9782840825685",
    "LANG": "fr",
    "SUBJECT": "applications",
    "AUTHOR": [
      {
        "FIRSTNAME": "William J.",
        "LASTNAME": "Pardi"
      }
    ],
    "TITLE": "XML en Action",
    "TRANSLATED": {
      "PREFIX": "adapté de l'anglais par",
      "TRANSLATOR": {
        "FIRSTNAME": "James",
        "LASTNAME": "Guerin"
        }
    },
    "PUBLISHER": {
      "NAME": "Microsoft Press",
      "PLACE": "Paris"
    },
    "DATEPUB": 1999
```

```
  }
]
```

This file contains the different items existing in JSON.

Arrays: They are enclosed in square brackets and contain a list of comma separated values.
Objects: They are enclosed in curly brackets. They contain a comma separated list of pairs, each pair composed of a key name between double quotes, followed by a ':' character and followed by a value.
Values: Values can be an array or an object. They also can be a string between double quote, an integer or float number, a Boolean value or a null value.

The simplest way for CONNECT to locate a table in such a file is by an array containing a list of objects. Each array value will be a table row and each pair of the row objects will represent a column, the key being the column name and the value the column value.

A first try to create a table on this file will be to take the outer array as the table:

```
create table jsample (
ISBN char(15),
LANG char(2),
SUBJECT char(32),
AUTHOR char(128),
TITLE char(32),
TRANSLATED char(80),
PUBLISHER char(20),
DATEPUB int(4))
engine=CONNECT table_type=JSON
File_name='biblio3.json';
```

If we execute the query:

```
select isbn, author, title, publisher from jsample;
```

We get the result:

isbn	author	title	publisher
9782212090819	Jean-Christophe Bernadac	Construire une application XML	Eyrolles Paris
9782840825685	William J. Pardi	XML en Action	Microsoft Press Pari

Note that by default, column values that are objects have been set to the concatenation of all the string values of the object separated by a blank. When a column value is an array, only the first item of the array is retrieved.

However, things are generally more complicated. If JSON files do not contain attributes (although object pairs are like attributes) they contain a new item, ARRAYS. We have seen that they can be used like XML multiple nodes, here to specify several authors, but they are more general because they can contain objects of different types, even it may not be advisable to do so.

This is why CONNECT enables to specify in the column FIELD_FORMAT option a "JPATH" that is used to described exactly where are the items to display and how to handles arrays.

Here is an example of a new table that can be created on the same file, allowing choosing the column names, to get some sub-objects and to specify how to handle the author array

```
create table jsampall (
ISBN char(15),
```

```
Language char(2) field_format='LANG',
Subject char(32) field_format='SUBJECT',
Author char(128) field_format='AUTHOR.[" and "]',
Title char(32) field_format='TITLE',
Translation char(32) field_format='TRANSLATOR.PREFIX',
Translator char(80) field_format='TRANSLATOR',
Publisher char(20) field_format='PUBLISHER.NAME',
Location char(16) field_format='PUBLISHER.PLACE',
Year int(4) field_format='DATEPUB')
engine=CONNECT table_type=JSON File_name='biblio3.json';
```

Given the query:

```
select title, author, publisher, location from jsampall;
```

The result is:

title	author	publisher	location
Construire une application XML	Jean-Christophe Bernadac and François Knab	Eyrolles	Paris
XML en Action	William J. Pardi	Microsoft Press	Paris

Here is another example showing that one can choose what to extract from the file and how to "expand" an array, meaning to generate one row for each array value:

```
create table jsampex (
ISBN char(15),
Title char(32) field_format='TITLE',
AuthorFN char(128) field_format='AUTHOR[*].FIRSTNAME',
AuthorLN char(128) field_format='AUTHOR[*].LASTNAME',
Year int(4) field_format='DATEPUB')
engine=CONNECT table_type=JSON File_name='biblio3.json';
```

It is displayed as:

ISBN	Title	AuthorFN	AuthorLN	Year
9782212090819	Construire une application XML	Jean-Christophe	Bernadac	1999
9782212090819	Construire une application XML	François	Knab	1999
9782840825685	XML en Action	William J.	Pardi	1999

The Jpath Specification

Caution: In this version of CONNECT, the Jpath specification has changed to be the one of the native JSON functions and more compatible with what is generally used. It is close to the standard definition and compatible to what MongoDB and other products do. The ':' separator is replaced by '.'. Position in array is accepted MongoDB style with no square brackets. Array specification specific to CONNECT are still accepted but [*] is used for expanding and [x] for multiply. However, tables created with the previous syntax can still be used by adding SEP_CHAR=':' (can be done with ALTER TABLE).

It is the description of the path to follow to reach the required item. Each step is the key name (case sensitive) of the pair when crossing an object, and the position number of the value when crossing an array. Key specifications are separated by a '.' character.

For instance, in the above file, the last name of the second author of a book is reached by:

```
$.AUTHOR[1].LASTNAME  // standard style
$AUTHOR.1.LASTNAME    // MongoDB style
```

```
AUTHOR:[1]:LASTNAME   // old style when SEP_CHAR=':'
```

The '$' or "$." prefix specifies the root of the path and can be omitted with CONNECT.

The array specification can also indicate how it must be processed:

Specification	Array Type	Limit	Description
[*n*] or *n*[16]	All	N.A.	Take the *n*th value of the array.
[*]	All		Expand. Generate one row for each array value.
["*string*"]	String		Concatenate all values separated by the specified *string*.
[+]	Numeric		Make the sum of all the array non-null values.
[x]	Numeric		Make the product of all array non-null values.
[!]	Numeric		Make the average of all the array non-null values.
[>] or [<]	All		Return the greatest or least non-null value of the array.
[#]	All	N.A.	Return the number of values in the array.
[]	All		Expand if under an expanded object. Otherwise Sum if numeric, else concatenation separated by ", ".
	All	N.A.	If an array, expand it if under an expanded object or take the first value of it.

Note 1: When the LIMIT restriction is applicable, only the first *m* array items are used, *m* being the value of the LIMIT option (to be specified in OPTION_LIST). The LIMIT default value is 10.

Note 2: An alternative way to indicate what is to be expanded – useful in particular with discovery -- is to use the EXPAND option in the option list, for instance:

```
OPTION_LIST='Expand=AUTHOR'
```

AUTHOR is here the key of the pair that has the array as value (case sensitive). Expand is limited to only one branch (all expanded arrays must be under the same object)

Let us take as an example the file *expense.json* shown in Appendix A.
The table *jexpall* expands all under and including the week array:

```
create table jexpall (
WHO char(12),
WEEK int(2) field_format='$.WEEK[*].NUMBER',
WHAT char(32) field_format='$.WEEK[*].EXPENSE[*].WHAT',
AMOUNT double(8,2) field_format='$.WEEK[*].EXPENSE[*].AMOUNT')
engine=CONNECT table_type=JSON File_name='expense.json';
```

WHO	WEEK	WHAT	AMOUNT
Joe	3	Beer	18.00
Joe	3	Food	12.00
Joe	3	Food	19.00
Joe	3	Car	20.00
Joe	4	Beer	19.00
Joe	4	Beer	16.00
Joe	4	Food	17.00
Joe	4	Food	17.00

[16] The value *n* can be 0 based or 1 based depending on the BASE table option. The default is 0 to match what is the current usage in the Json world but it can be set to 1 for tables created in old versions.

Joe	4	Beer	14.00
Joe	5	Beer	14.00
Joe	5	Food	12.00
Beth	3	Beer	16.00
Beth	4	Food	17.00
Beth	4	Beer	15.00
Beth	5	Food	12.00
Beth	5	Beer	20.00
Janet	3	Car	19.00
Janet	3	Food	18.00
Janet	3	Beer	18.00
Janet	4	Car	17.00
Janet	5	Beer	14.00
Janet	5	Car	12.00
Janet	5	Beer	19.00
Janet	5	Food	12.00

The table *jexpw* shows what was bought and the sum and average of amounts for each person and week:

```
create table jexpw (
WHO char(12) not null,
WEEK int(2) not null field_format='$.WEEK[*].NUMBER',
WHAT char(32) not null field_format='$.WEEK[].EXPENSE[", "].WHAT',
SUM double(8,2) not null field_format='$.WEEK[].EXPENSE[+].AMOUNT',
AVERAGE double(8,2) not null
     field_format='$.WEEK[].EXPENSE[!].AMOUNT')
engine=CONNECT table_type=JSON File_name='expense.json';
```

WHO	WEEK	WHAT	SUM	AVERAGE
Joe	3	Beer, Food, Food, Car	69.00	17.25
Joe	4	Beer, Beer, Food, Food, Beer	83.00	16.60
Joe	5	Beer, Food	26.00	13.00
Beth	3	Beer	16.00	16.00
Beth	4	Food, Beer	32.00	16.00
Beth	5	Food, Beer	32.00	16.00
Janet	3	Car, Food, Beer	55.00	18.33
Janet	4	Car	17.00	17.00
Janet	5	Beer, Car, Beer, Food	57.00	14.25

Let us see what does the table *jexpz*:

```
create table jexpz (
WHO char(12) not null,
WEEKS char(12) not null field_format='WEEK[", "].NUMBER',
SUMS char(64) not null field_format='WEEK["+"].EXPENSE[+].AMOUNT',
SUM double(8,2) not null field_format='WEEK[+].EXPENSE[+].AMOUNT',
AVGS char(64) not null field_format='WEEK["+"].EXPENSE[!].AMOUNT',
SUMAVG double(8,2) not null field_format='WEEK[+].EXPENSE[!].AMOUNT',
AVGSUM double(8,2) not null field_format='WEEK[!].EXPENSE[+].AMOUNT',
AVERAGE double(8,2) not null
     field_format='WEEK[!].EXPENSE[*].AMOUNT')
engine=CONNECT table_type=JSON File_name='expense.json';
```

WHO	WEEKS	SUMS	SUM	AVGS	SUMAVG	AVGSUM	AVERAGE
Joe	3, 4, 5	69.00+83.00+26.00	178.00	17.25+16.60+13.00	46.85	59.33	16.18
Beth	3, 4, 5	16.00+32.00+32.00	80.00	16.00+16.00+16.00	48.00	26.67	16.00
Janet	3, 4, 5	55.00+17.00+57.00	129.00	18.33+17.00+14.25	49.58	43.00	16.12

For all persons:

Column 1 shows the person name.
Column 2 shows the weeks for which values are calculated.
Column 3 lists the sums of expenses for each week.
Column 4 calculates the sum of all expenses by person.
Column 5 shows the week's expense averages.
Column 6 calculates the sum of these averages.
Column 7 calculates the average of the week's sum of expenses.
Column 8 calculates the average expense by person.

It would be very difficult, if even possible, to obtain this result from table jexpall using an SQL query.

Handling of NULL Values

Json has a **null** explicit value that can be met in arrays or object key values. When regarding json as a relational table, a column value can be null because the corresponding json item is explicitly null or implicitly because the corresponding item is missing in an array or object. CONNECT does not make any difference between explicit or implicit nulls.

However, it is possible to specify how nulls are handled and represented. This is done by setting the string session variable *connect_json_null*. The default value of connect_json_null is "<null>"; it can be changed, for instance, by:

```
SET connect_json_null='NULL';
```

This changes its representation when a column displays the text of an object or the concatenation of the values of an array.

It is also possible to tell CONNECT to ignore nulls by:

```
SET connect_json_null=NULL;
```

When doing so, nulls do not appear in object text or array lists. However, this does not change the behavior of array calculation not the result of array count.

Having Columns defined by Discovery

It is possible to let the MariaDB discovery process do the job of column specification. When columns are not defined in the CREATE TABLE statement, CONNECT endeavors to analyze the JSON file and to provide the column specifications. This is possible only for tables represented by an array of objects because CONNECT retrieves the column names from the object pair keys and their definition from the object pair values. For instance, the *jsample* table could be created saying:

```
create table jsample engine=connect table_type=JSON
file_name='biblio3.json';
```

Let's check how it was specified using the SHOW CREATE TABLE statement:

```
CREATE TABLE `jsample` (
  `ISBN` char(13) NOT NULL,
  `LANG` char(2) NOT NULL,
  `SUBJECT` char(12) NOT NULL,
  `AUTHOR` varchar(256) DEFAULT NULL,
```

```
  `TITLE` char(30) NOT NULL,
  `TRANSLATED` varchar(256) DEFAULT NULL,
  `PUBLISHER` varchar(256) DEFAULT NULL,
  `DATEPUB` int(4) NOT NULL
) ENGINE=CONNECT DEFAULT CHARSET=latin1 `TABLE_TYPE`='JSON'
`FILE_NAME`='biblio3.json';
```

It is equivalent except for the column sizes that have been calculated from the file as the maximum length of the corresponding column when it was a normal value. For columns that are Json arrays or objects, the column is specified as a VARCHAR string of length 256, supposedly big enough to contain the sub-object's concatenated values. Nullable is set to true if the column is null or missing in some rows or if its JPATH contains arrays.

If a more complex definition is desired, you can ask CONNECT to analyse the JPATH up to a given level using the LEVEL option in the option list. The level value is the number of sub-objects that are taken in the JPATH. For instance:

```
create table jsampall2 engine=connect table_type=JSON
file_name='biblio3.json' option_list='level=1';
```

This will define the table as:

```
CREATE TABLE `jsampall2` (
  `ISBN` char(13) NOT NULL,
  `LANG` char(2) NOT NULL,
  `SUBJECT` char(12) NOT NULL,
  `AUTHOR_FIRSTNAME` char(15) NOT NULL `FIELD_FORMAT`='AUTHOR..FIRSTNAME',
  `AUTHOR_LASTNAME` char(8) NOT NULL `FIELD_FORMAT`='AUTHOR..LASTNAME',
  `TITLE` char(30) NOT NULL,
  `TRANSLATED_PREFIX` char(23) DEFAULT NULL `FIELD_FORMAT`='TRANSLATED.PREFIX',
  `TRANSLATED_TRANSLATOR` varchar(256) DEFAULT NULL
`FIELD_FORMAT`='TRANSLATED.TRANSLATOR',
  `PUBLISHER_NAME` char(15) NOT NULL `FIELD_FORMAT`='PUBLISHER.NAME',
  `PUBLISHER_PLACE` char(5) NOT NULL `FIELD_FORMAT`='PUBLISHER.PLACE',
  `DATEPUB` int(4) NOT NULL
) ENGINE=CONNECT DEFAULT CHARSET=latin1 `TABLE_TYPE`='JSON' `FILE_NAME`='biblio3.json'
`OPTION_LIST`='level=1';
```

The problem is that CONNECT cannot guess what you want to do with arrays. Here the AUTHOR array is left undefined, which means that only its first value will be retrieved unless you also had specified "Expand=AUTHOR" in the option list. But of course, you can replace it by anything else.

This method can be used as a quick way to make a "template" table definition that can later be edited to make the desired definition. In particular, column names are constructed from all the object keys of their path in order to have distinct column names. This can be manually edited to have the desired names, provided their JPATH key names are not modified.

Level can also be given the value -1 to create only columns that are simple values (no array or object).

Note: Since version 1.6.4, CONNECT eliminates columns that are "void" or whose type cannot be determined. For instance given the file *sresto.json*:

```
{"_id":1,"name":"Corner Social","cuisine":"American","grades":[{"grade":"A","score":6}]}
{"_id":2,"name":"La Nueva Clasica Antillana","cuisine":"Spanish","grades":[]}
```

Previously, when using discovery, creating the table by:

```
create table sjr0
engine=connect table_type=JSON file_name='sresto.json'
option_list='Pretty=0,Level=1' lrecl=128;
```

The table was previously created as:

```
CREATE TABLE `sjr0` (
  `_id` bigint(1) NOT NULL,
  `name` char(26) NOT NULL,
  `cuisine` char(8) NOT NULL,
  `grades` char(1) DEFAULT NULL,
  `grades_grade` char(1) DEFAULT NULL `FIELD_FORMAT`='$.grades[0].grade',
  `grades_score` bigint(1) DEFAULT NULL `FIELD_FORMAT`='$.grades[0].score'
) ENGINE=CONNECT DEFAULT CHARSET=latin1 `TABLE_TYPE`='JSON'
     `FILE_NAME`='sresto.json' `OPTION_LIST`='Pretty=0,Level=1,Accept=1'
     `LRECL`=128;
```

The column “grades” was added because of the void array in line 2. Now this column is skipped and does not appear anymore (unless the option Accept=1 is added in the option list).

JSON Catalogue Tables

Another way to see JSON table column specifications is to use a catalogue table. For instance:

```
create table bibcol engine=connect table_type=JSON
     file_name='biblio3.json' option_list='level=2' catfunc=columns;
select column_name, type_name type, column_size size, jpath from
     bibcol;
```

This reply:

column_name	type	size	jpath
ISBN	CHAR	13	
LANG	CHAR	2	
SUBJECT	CHAR	12	
AUTHOR_FIRSTNAME	CHAR	15	AUTHOR..FIRSTNAME
AUTHOR_LASTNAME	CHAR	8	AUTHOR..LASTNAME
TITLE	CHAR	30	
TRANSLATED_PREFIX	CHAR	23	TRANSLATED.PREFIX
TRANSLATED_TRANSLATOR_FIRSTNAME	CHAR	5	TRANSLATED.TRANSLATOR.FIRSTNAME
TRANSLATED_TRANSLATOR_LASTNAME	CHAR	6	TRANSLATED.TRANSLATOR.LASTNAME
PUBLISHER_NAME	CHAR	15	PUBLISHER.NAME
PUBLISHER_PLACE	CHAR	5	PUBLISHER.PLACE
DATEPUB	INTEGER	4	

All this is mostly useful when creating a table on a remote file that you cannot easily see.

Finding the table within a JSON file

Given the file “facebook.json”:

```
{
   "data": [
      {
         "id": "X999_Y999",
         "from": {
            "name": "Tom Brady", "id": "X12"
         },
         "message": "Looking forward to 2010!",
         "actions": [
            {
               "name": "Comment",
               "link": "http://www.facebook.com/X999/posts/Y999"
```

```
            },
            {
               "name": "Like",
               "link": "http://www.facebook.com/X999/posts/Y999"
            }
         ],
         "type": "status",
         "created_time": "2010-08-02T21:27:44+0000",
         "updated_time": "2010-08-02T21:27:44+0000"
      },
      {
         "id": "X998_Y998",
         "from": {
            "name": "Peyton Manning", "id": "X18"
         },
         "message": "Where's my contract?",
         "actions": [
            {
               "name": "Comment",
               "link": "http://www.facebook.com/X998/posts/Y998"
            },
            {
               "name": "Like",
               "link": "http://www.facebook.com/X998/posts/Y998"
            }
         ],
         "type": "status",
         "created_time": "2010-08-02T21:27:44+0000",
         "updated_time": "2010-08-02T21:27:44+0000"
      }
   ]
}
```

The table we want to analyze is represented by the array value of the "data" object. Here is how this is specified in the CREATE TABLE statement:

```
create table jfacebook (
`ID` char(10) field_format='id',
`Name` char(32) field_format='from.name',
`MyID` char(16) field_format='from.id',
`Message` varchar(256) field_format='message',
`Action` char(16) field_format='actions..name',
`Link` varchar(256) field_format='actions..link',
`Type` char(16) field_format='type',
`Created` datetime date_format='YYYY-MM-DD\'T\'hh:mm:ss'
field_format='created_time',
`Updated` datetime date_format='YYYY-MM-DD\'T\'hh:mm:ss'
field_format='updated_time')
engine=connect table_type=JSON file_name='facebook.json'
option_list='Object=data,Expand=actions';
```

This is the OBJECT option that gives the Jpath of the table. Note also an alternate way to declare the array to be expanded by the EXPAND option of the option_list.

Because some string values contain a date representation, the corresponding columns are declared as datetime and the date format is specified for them.

The Jpath of the object option has the same syntax than the column Jpath but of course all array steps must be specified using the *n* format.

Note: All this applies only to tables having PRETTY = 2 (see below).

JSON File Formats

The examples we have seen so far are files that, even they can be formatted in different ways (blanks, tabs, carriage return and line feed are ignored when parsing them), respect the JSON syntax and are made of only one item (Object or Array). Like for XML files, they are entirely parsed and a memory representation is made used to process them. This implies that they are of reasonable size to avoid an out of memory condition. Tables based on such files are recognized by the option Pretty=2 that we did not specify above because this is the default.

An alternate format, which is the format of exported MongoDB files, is a file where each row is physically stored in one file record. For instance:

```
{ "_id" : "01001", "city" : "AGAWAM", "loc" : [ -72.622739, 42.070206 ], "pop" : 15338, "state" : "MA" }
{ "_id" : "01002", "city" : "CUSHMAN", "loc" : [ -72.51564999999999, 42.377017 ], "pop" : 36963, "state" : "MA" }
{ "_id" : "01005", "city" : "BARRE", "loc" : [ -72.1083540000001, 42.409698 ], "pop" : 4546, "state" : "MA" }
{ "_id" : "01007", "city" : "BELCHERTOWN", "loc" : [ -72.4109530000001, 42.275103 ], "pop" : 10579, "state" : "MA" }
...
{ "_id" : "99929", "city" : "WRANGELL", "loc" : [ -132.352918, 56.433524 ], "pop" : 2573, "state" : "AK" }
{ "_id" : "99950", "city" : "KETCHIKAN", "loc" : [ -133.18479, 55.942471 ], "pop" : 422, "state" : "AK" }
```

The original file, “cities.json”, has 29352 records. To base a table on this file we must specify the option Pretty=0 in the option list. For instance:

```
create table cities (
`_id` char(5) key,
`city` char(32),
`lat` double(12,6) field_format='loc.0',
`long` double(12,6) field_format='loc.1',
`pop` int(8),
`state` char(2) distrib='clustered')
engine=CONNECT table_type=JSON file_name='cities.json'
lrecl=128 option_list='pretty=0';
```

Note the use of *n* array specifications for the latitude and longitude columns.

When using this format, the table is processed by CONNECT like a DOS, CSV or FMT table. Rows are retrieved and parsed by records and the table can be very large. Another advantage is that such a table can be indexed, which can be of great value for very large tables. The “distrib” option of the “state” column tells CONNECT to use block indexing when possible.

For such tables – as well as for pretty=1 ones – the record size must be specified using the LRECL option. Be sure you don’t specify it too small as it is used to allocate the read/write buffers and the memory used for parsing the rows. In doubt, be generous as it does not cost much in memory allocation.

Another format exists, noted by Pretty=1, which is similar to this one but has some additions to represent a JSON array. A header and a trailer records are added containing the opening and closing square bracket, and all records but the last are followed by a comma. It has the same advantages for reading and updating, but inserting and deleting are executed in the PRETTY=2 other way.

Alternate Table Arrangement

We have seen that the most natural way to represent a table in a JSON file is to make it on an array of objects. However, other possibilities exist. A table can be an array of arrays, a one column table can be an array of values, or a one row table can be just one object or one value. One row tables are internally handled by adding a one value array around them.

Let us see how to handle, for instance, a table that is an array of arrays. The file:

```
[
```

```
 [56, "Coucou", 500.00],
 [[2,0,1,4], "Hello World", 2.0316],
 ["1784", "John Doo", 32.4500],
 [1914, ["Nabucho","donosor"], 5.12],
 [7, "sept", [0.77,1.22,2.01]],
 [8, "huit", 13.0]
]
```

A table can be created on this file as:

```
create table xjson (
`a` int(6) field_format='1',
`b` char(32) field_format='2',
`c` double(10,4) field_format='3')
engine=connect table_type=JSON file_name='test.json'
option_list='Pretty=1,Jmode=1,Base=1' lrecl=128;
```

Columns are specified by their position in the row arrays. By default, this is 0 based but for this table the base was set to 1 by the Base option of the option list. Another new option in the option list is Jmode=1. It indicates what type of table this is. The Jmode values are:

0 An array of objects. This is the default.
1 An array of Array. Like this one.
2 An array of values.

When reading, this is not required as the type of the array items is specified for the columns; however, it is required when inserting new rows so CONNECT knows what to insert. For instance:

```
insert into xjson values(25, 'Breakfast', 1.414);
```

After this, it is displayed as:

a	b	c
56	Coucou	500.0000
2	Hello World	2.0316
1784	John Doo	32.4500
1914	Nabucho	5.1200
7	sept	0.7700
8	huit	13.0000
25	Breakfast	1.4140

Unspecified array values are represented by their first element.

Getting and Setting JSON Representation of a Column

It is possible to retrieve and display a column contain as the full JSON string corresponding to it in the JSON file. This is specified in the JPATH by a “*” where the object or array would be specified. For instance:

```
create table jsample2 (
ISBN char(15),
Lng char(2) field_format='LANG',
json_Author char(255) field_format='AUTHOR.*',
Title char(32) field_format='TITLE',
Year int(4) field_format='DATEPUB')
engine=CONNECT table_type=JSON file_name='biblio3.json';
```

Now the query:

```
select json_Author from jsample2;
```

will return and display :

json_Author
[{"FIRSTNAME":"Jean-Christophe","LASTNAME":"Bernadac"},{"FIRSTNAME":"François","LASTNAME":"Knab"}]
[{"FIRSTNAME":"William J.","LASTNAME":"Pardi"}]

This also works on input, a column specified as so can be directly set to a JSON valid string.

This feature is of great value as we will see below.

CRUD Operations on JSON Tables

The SQL commands INSERT, UPDATE and DELETE are fully supported for JSON tables. For INSERT and UPDATE, if the target values are simple values, there are no problems.

However, there are some issues when the added or modified values are objects or arrays.

Concerning objects, the same problems exist that we have already seen with the XML type. The added or modified object will have the format described in the table definition, which can be different from the one of the JSON file. Modifications should be done using a file specifying the full path of modified objects.

New problems are raised when trying to modify the values of an array. Only updates can be done on the original table. First of all, for the values of the array to be distinct values, all update operations concerning array values must be done using a table expanding this array.

For instance, to modify the authors of the *biblio.json* based table, the *jsampex* table must be used. Doing so, updating and deleting authors is possible using standard SQL commands. For example, to change the first name of Knab from François to John:

```
update jsampex set authorfn = 'John' where authorln = 'Knab';
```

However, it would be wrong to do:

```
update jsampex set authorfn = 'John' where isbn = '9782212090819';
```

Because this would change the first name of both authors as they share the same ISBN.

Where things become more difficult is when trying to delete or insert an author of a book. Indeed, a DELETE command will delete the whole book and an INSERT command will add a new complete row instead of adding a new author in the same array. Here we are penalized by the SQL language that cannot give us a way to specify this. Something like:

```
update jsampex add authorfn = 'Charles', authorln = 'Dickens'
where title = 'XML en Action';
```

However, this does not exist in SQL. Does this mean that it is impossible to do it? No, but it requires us to use a table specified on the same file but adapted to this task.

One way to do it is to specify a table for which the authors are no more an expanded array. Supposing we want to add an author to the “XML en Action” book, we will do it on a table containing just the author(s) of that book, which is the second book of the table.

```
create table jauthor (
FIRSTNAME char(64),
LASTNAME char(64))
engine=CONNECT table_type=JSON File_name='biblio3.json'
option_list='Object=21.AUTHOR';
```

The command:

```
select * from jauthor;
```

replies:

FIRSTNAME	LASTNAME
William J.	Pardi

It is a standard JSON table that is an array of objects in which we can freely insert or delete rows.

```
insert into jauthor values('Charles','Dickens');
```

We can check that this was done correctly by:

```
select * from jsampex;
```

This will display:

ISBN	Title	AuthorFN	AuthorLN	Year
9782212090819	Construire une application XML	Jean-Christophe	Bernadac	1999
9782212090819	Construire une application XML	John	Knab	1999
9782840825685	XML en Action	William J.	Pardi	1999
9782840825685	XML en Action	Charles	Dickens	1999

Note: If this table were a big table with many books, it would be difficult to know what the order of a specific book is in the table. This can be found by adding a special ROWID column in the table.

However, an alternate way to do it is by using direct JSON column representation as in the JSAMPLE2 table. This can be done by:

```
update jsample2 set json_Author =
'[{"FIRSTNAME":"William J.","LASTNAME":"Pardi"},
  {"FIRSTNAME":"Charles","LASTNAME":"Dickens"}]'
where isbn = '9782840825685';
```

Here, we didn't have to find the index of the sub array to modify. However, this is not quite satisfying because we had to manually write the whole JSON value to set to the *json_Author* column.

Therefore, we need specific functions to do so. They are introduced now.

JSON User Defined Functions

Although such functions written by other parties do exist[17], CONNECT provides its own UDF's that are specifically adapted to the JSON table type and easily available because, being inside the CONNECT library or DLL, they require no additional module to be loaded[18].

In particular, MariaDB 10.2 and 10.3 feature native JSON functions. In some cases, it is possible that these native functions can be used. However, mixing native and UDF JSON functions in the same query often does not work because the way they recognize their arguments is different and might even cause a server crash.

Here is the list of the CONNECT functions; more can be added if required.

Name	Type	Return	Description
jsonvalue	Function	STRING	Make a JSON value from its unique argument.
json_make_array	Function	STRING	Make a JSON array containing its arguments.
json_array_add_values	Functions	STRING	Adds to its first array argument all following arguments.
json_array_add	Function	STRING	Adds to its first array argument its second arguments.
json_array_delete	Function	STRING	Deletes the *n*th element of its first array argument.
json_make_object	Function	STRING	Make a JSON object containing its arguments.
json_object_nonull	Function	STRING	Make a JSON object containing its not null arguments.
json_object_key	Function	STRING	Make a JSON object for key/value pairs.
json_object_add	Function	STRING	Adds to its first object argument its second argument.
json_object_delete	Function	STRING	Deletes the *n*th element of its first object argument.
json_object_list	Function	STRING	Returns the list of object keys as an array.
json_object_values	Function	STRING	Returns the list of object values as an array.
jsonset_grp_size	Function	INTEGER	Sets the JsonGrpSize value and returns it.
jsonget_grp_size	Function	INTEGER	Returns the JsonGrpSize value.
json_array_grp	Aggregate	STRING	Makes JSON arrays from coming argument.
json_object_grp	Aggregate	STRING	Makes JSON objects from coming arguments.
jsonlocate	Function	STRING	Returns the JPATH to access one element.
json_locate_all	Function	STRING	Returns the JPATH's of all occurrences of an element.
jsoncontains	Function	INTEGER	Returns 0 or 1 if an element is contained in the document.
Jsoncontains_path	Function	INTEGER	Returns 0 or 1 if a JPATH is contained in the document.
json_item_merge	Function	STRING	Merges two arrays or two objects.
json_get_item	Function	STRING	Access and returns a Json item by a JPATH key.
jsonget_string	Function	STRING	Access and returns a string element by a JPATH key.
jsonget_int	Function	INTEGER	Access and returns an integer element by a JPATH key.
jsonget_real	Function	REAL	Access and returns a real element by a JPATH key.
json_set_item	Function	STRING	Set item values located to paths.
json_insert_item	Function	STRING	Insert item values located to paths.
json_update_item	Function	STRING	Update item values located to paths.
json_file	Function	STRING	Returns the contains of (Json) file.
jfile_make	Function	STRING	Make a Json file from its Json item first argument.
json_serialize	Function	STRING	Serializes the return of a "Jbin" function.
jbin_array	Function	STRING*	Make a JSON array containing its arguments.
jbin_array_add_values	Function	STRING*	Adds to its first array argument all following arguments.

[17] See for instance:
https://mariadb.com/kb/en/mariadb/json-functions/
https://github.com/mysqludf/lib_mysqludf_json#readme
https://blogs.oracle.com/svetasmirnova/entry/json_udf_functions_0_4

[18] See Appendix C to make these functions in a separate library module.

Name	Type	Return	Description
jbin_array_add	Function	STRING*	Adds to its first array argument its second arguments.
jbin_array_delete	Function	STRING*	Deletes the *n*th element of its first array argument.
jbin_object	Function	STRING*	Make a JSON object containing its arguments.
jbin_object_nonull	Function	STRING*	Make a JSON object containing its not null arguments.
jbin_object_key	Function	STRING*	Make a JSON object for key/value pairs.
jbin_object_add	Function	STRING*	Adds to its first object argument its second argument.
jbin_object_delete	Function	STRING*	Deletes the *n*th element of its first object argument.
jbin_object_list	Function	STRING*	Returns the list of object keys as an array.
jbin_item_merge	Function	STRING*	Merges two arrays or two objects.
jbin_get_item	Function	STRING*	Access and returns a Json item by a JPATH key.
jbin_set_item	Function	STRING	Set item values located to paths.
jbin_insert_item	Function	STRING	Insert item values located to paths.
jbin_update_item	Function	STRING	Update item values located to paths.
jbin_file	Function	STRING*	Returns of a (Json) file contain.

String values are mapped to JSON strings. These strings are automatically escaped to conform to the JSON syntax. The automatic escaping is bypassed when the value has an alias beginning with 'json_'. This is automatically the case when a JSON UDF argument is another JSON UDF whose name begins with "json_" (not case sensitive). This is why all functions that do not return a Json item are not prefixed by "json_".

Numeric values are (big) integers, double floating-point values or decimal values. Decimal values are character strings containing a numeric representation and are treated as strings. Floating point values contain a decimal point and/or an exponent. Integers are written without decimal points.

To install these functions, execute the following commands[19]:

```
create function jsonvalue returns string soname 'ha_connect';
create function json_make_array returns string soname 'ha_connect';
create function json_array_add_values returns string soname 'ha_connect';
create function json_array_add returns string soname 'ha_connect';
create function json_array_delete returns string soname 'ha_connect';
create function json_make_object returns string soname 'ha_connect';
create function json_object_nonull returns string soname 'ha_connect';
create function json_object_key returns string soname 'ha_connect';
create function json_object_add returns string soname 'ha_connect';
create function json_object_delete returns string soname 'ha_connect';
create function json_object_list returns string soname 'ha_connect';
create function json_object_values returns string soname 'ha_connect';
create function jsonset_grp_size returns integer soname 'ha_connect';
create function jsonget_grp_size returns integer soname 'ha_connect';
create aggregate function json_array_grp returns string soname 'ha_connect';
create aggregate function json_object_grp returns string soname
     'ha_connect';
create function jsonlocate returns string soname 'ha_connect';
create function json_locate_all returns string soname 'ha_connect';
create function jsoncontains returns integer soname 'ha_connect';
create function jsoncontains_path returns integer soname 'ha_connect';
create function json_item_merge returns string soname 'ha_connect';
create function json_get_item returns string soname 'ha_connect';
create function jsonget_string returns string soname 'ha_connect';
create function jsonget_int returns integer soname 'ha_connect';
create function jsonget_real returns real soname 'ha_connect';
create function json_set_item returns string soname 'ha_connect';
create function json_insert_item returns string soname 'ha_connect';
create function json_update_item returns string soname 'ha_connect';
create function json_file returns string soname 'ha_connect';
```

[19] This works on Windows only. On Linux, the module name must be specified as 'ha_connect.so'.

```
create function jfile_make returns string soname 'ha_connect';
create function json_serialize returns string soname 'ha_connect';
create function jbin_array returns string soname 'ha_connect';
create function jbin_array_add_values returns string soname 'ha_connect';
create function jbin_array_add returns string soname 'ha_connect';
create function jbin_array_delete returns string soname 'ha_connect';
create function jbin_object returns string soname 'ha_connect';
create function jbin_object_nonull returns string soname 'ha_connect';
create function jbin_object_key returns string soname 'ha_connect';
create function jbin_object_add returns string soname 'ha_connect';
create function jbin_object_delete returns string soname 'ha_connect';
create function jbin_object_list returns string soname 'ha_connect';
create function jbin_item_merge returns string soname 'ha_connect';
create function jbin_get_item returns string soname 'ha_connect';
create function jbin_set_item returns string soname 'ha_connect';
create function jbin_insert_item returns string soname 'ha_connect';
create function jbin_update_item returns string soname 'ha_connect';
create function jbin_file returns string soname 'ha_connect';
```

Note: In this document, json function names are often written with leading upper-case letters for clarity. It is possible to do so in SQL queries because function names are case insensitive. However, when creating or dropping them, their names must be in lower case like they are in the library module.

JsonValue(val):

Returns a JSON value as a string, for instance:

```
select JsonValue(3.1416);
```

JsonValue(3.1416)
3.141600

Json_Make_Array([val*1*[, …, val*n*]]):

This function was named “Json_Array” in previous versions of CONNECT. It was renamed because MariaDB 10.2 features native JSON functions including a “Json_Array” function. The native function does almost the same than the UDF one but does not accept CONNECT specific arguments such as the result from JBIN functions.

Json_Make_Array returns a string denoting a JSON array with all its arguments as members. For example:

```
select Json_Make_Array(56, 3.1416, 'My name is "Foo"', NULL);
```

Json_Make_Array(56, 3.1416, 'My name is "Foo"', NULL)
[56,3.141600,"My name is \"Foo\"",null]

Note: The argument list can be void. If so a void array is returned.

Json_Array_Add_Values(json_doc[, val_list]):

The first argument must be a JSON array string. Then all other arguments are added as members of this array. For example:

```
select Json_Array_Add_Values(Json_Make_Array(56, 3.1416, 'machin',
    NULL), 'One more', 'Two more') Array;
```

Array
[56,3.141600,"machin",null,"One more","Two more"]

Json_Array_Add(json_doc, val_list, [arg3], [arg4] …):

The first argument must be a JSON array. The second argument is added as member of this array. For example:

```
select
Json_Array_Add(Json_Make_Array(56,3.1416,'machin',NULL),
'One more') Array;
```

Array
[56,3.141600,"machin",null,"One more"]

Note: The first array is not escaped, its (alias) name beginning with 'json_'.

Now we can see how adding an author to the JSAMPLE2 table can alternatively be done:

```
update jsample2 set json_author = json_array_add(json_author,
json_make_object('Charles' FIRSTNAME, 'Dickens' LASTNAME))
where isbn = '9782840825685';
```

Note: Calling a column returning JSON a name prefixed by json_ (like *json_author* here) is good practice and remove the need to give it an alias to prevent escaping when used as an argument.

Additional arguments:
If a third integer argument is given, it specifies the position (zero based) of the added value:

```
select Json_Array_Add('[5,3,8,7,9]' json_, 4, 2) Array;
```

Array
[5,3,4,8,7,9]

If a string argument is added, it specifies the Json path to the array to be modified. For instance:

```
select Json_Array_Add('{"a":1,"b":2,"c":[3,4]}' json_, 5, 1, 'c');
```

Json_Array_Add('{"a":1,"b":2,"c":[3, 4]}' json_, 5, 1, 'c')
{"a":1,"b":2,"c":[3,5,4]}

Json_Array_Delete(json_doc, index, [arg3] …):

The first argument should be a JSON array. The second argument is an integer indicating the rank (0 based conforming to general json usage) of the element to delete. For example:

```
select Json_Array_Delete(Json_Make_Array(56,3.1416,'foo',NULL),1)
    Array;
```

Array
[56,"foo",null]

Now we can see how to delete the second author from the JSAMPLE2 table:

```
update jsample2 set json_author =
json_array_delete(json_author, 1) where isbn =
'9782840825685';
```

A Json path can be specified as a third string argument. It enables to specify to which item of the json document the deleting is applied.

Json_Make_Object([val*1*[, …, val*n*]]):

This function was named "Json_Object" in previous versions of CONNECT. It was renamed because MariaDB 10.2 features native JSON functions including a "Json_Object" function. The native function does what the UDF Json_Object_Key does.

Json_Make_Object returns a string denoting a JSON object. For instance:

```
select Json_Make_Object(56, 3.1416, 'machin', NULL);
```

The object is filled with pairs corresponding to the given arguments. The key of each pair is made from the argument (default or specified) alias.

Json_Make_Object(56, 3.1416, 'machin', NULL)
{"56":56,"3.1416":3.141600,"machin":"machin","NULL":null}

When needed, specify the keys by giving an alias to the arguments:

```
select Json_Make_Object(56 qty,3.1416 price,'machin' truc, NULL
    garanty);
```

Json_Make_Object(56 qty,3.1416 price,'machin' truc, NULL garanty)
{"qty":56,"price":3.141600,"truc":"machin","garanty":null}

If the alias is prefixed by 'json_' (to prevent escaping) the key name is stripped from that prefix.

This function is chiefly useful when entering values retrieved from a table, the key being by default the column name:

```
select Json_Make_Object(matricule, nom, titre, salaire) from
connect.employe where nom = 'PANTIER';
```

Json_Make_Object(matricule, nom, titre, salaire)
{"matricule":40567,"nom":"PANTIER","titre":"DIRECTEUR","salaire":14000.000000}

Json_Object_Nonull([val*1* [, …, val*n*]])

This function works like Json_Make_Object but "null" arguments are ignored and not inserted in the object.
Arguments are regarded as "null" if they are JSON null values, void arrays or objects, or arrays or objects containing only null members.

It is mainly used to avoid constructing useless null items when converting tables (see later).

Json_Object_Key([key*1*, val*1* [, …, key*n*, val*n*]])

Return a string denoting a JSON object. For instance:

```
select Json_Object_Key('qty', 56, 'price', 3.1416, 'truc', 'machin',
    'garanty', NULL);
```

The object is filled with pairs made from each key/value arguments.

Json_Object_Key('qty', 56, 'price', 3.1416, 'truc', 'machin', 'garanty', NULL)
{"qty":56,"price":3.141600,"truc":"machin","garanty":null}

Json_Object_Add(json_doc, pair, [path] …):

The first argument must be a JSON object. The second argument is added as a pair to this object. For example:

```
select Json_Object_Add('{"item":"T-shirt","qty":27,"price":24.99}'
     json_old,'blue' color) newobj;
```

newobj
{"item":"T-shirt","qty":27,"price":24.990000,"color":"blue"}

Note: If the specified key already exists in the object, its value is replaced by the new one.

Third string argument is a Json path to the target object within the document.

Json_Object_Delete(json_doc, key, [path] …):

The first argument must be a JSON object. The second argument is the key of the pair to delete. For example:

```
select Json_Object_Delete('{"item":"T-shirt","qty":27,"price":24.99}'
     json_old, 'qty') newobj;
```

newobj
{"item":"T-shirt","price":24.99}

Third string argument can be a Json path to the object to be the target of deletion.

Json_Object_List(json_object):

The first argument must be a JSON object. This function returns an array containing the list of all keys existing in the object. For example:

```
select Json_Object_List(Json_Make_Object(56 qty,3.1416 price,'machin'
     truc, NULL garanty)) "Key List";
```

Key List
["qty","price","truc","garanty"]

Json_Object_Values(json_object):

The first argument must be a JSON object. This function returns an array containing the list of all values existing in the object. For example:

```
select Json_Object_Values('{"One":1,"Two":2,"Three":3}')
"Value List";
```

Value List
[1,2,3]

Note: This function is new and may not exists in older versions.

JsonSet_Grp_Size(*val*)

This function is used to set the JsonGrpSize value. This value is used by the following aggregate functions as a ceiling value of the number of items in each group. It returns the JsonGrpSize value that can be its default value when passed 0 as argument.

JsonGet_Grp_Size(*val*)

This function returns the JsonGrpSize value.

Json_Array_Grp(arg)

This is an aggregate function that makes an array filled from values coming from the rows retrieved by a query. Let us suppose we have the *pet* table:

name	race	number
John	dog	2
Bill	cat	1
Mary	dog	1
Mary	cat	1
Lisbeth	rabbit	2
Kevin	cat	2
Kevin	bird	6
Donald	dog	1
Donald	fish	3

The query:

```
select name, json_array_grp(race) from pet group by name;
```

will return:

name	json_array_grp(race)
Bill	["cat"]
Donald	["dog","fish"]
John	["dog"]
Kevin	["cat","bird"]
Lisbeth	["rabbit"]
Mary	["dog","cat"]

One problem with the JSON aggregate functions is that they construct their result in memory and cannot know the needed amount of storage, not knowing the number of rows of the used table.

Therefore, the number of values for each group is limited. This limit is the value of *JsonGrpSize* whose default value is 10 but can be set using the JsonSet_Grp_Size function[20]. Nevertheless, working on a larger table is possible, but only after setting *JsonGrpSize* to the ceiling of the number of rows per group for the table. Try not to set it to a very large value to avoid memory exhaustion.

Json_Object_Grp(arg1,arg2)

This function works like Json_Array_Grp. It makes a JSON object filled with values passed from its first argument. Values passed from the second argument will be the keys associated with the values.

This can be seen with the query:

```
select name, json_object_grp(number,race) from pet group by name;
```

This query returns:

name	json_object_grp(number,race)
Bill	{"cat":1}
Donald	{"dog":1,"fish":3}
John	{"dog":2}
Kevin	{"cat":2,"bird":6}

[20] When *JsonGrpSize* is set to 0, CONNECT uses the deprecated *connect_json_grp_size* session variable instead. This can be temporarily used for existing applications written for older versions but works only is the CONNECT engine is installed.

Lisbeth	{"rabbit":2}
Mary	{"dog":1,"cat":1}

JsonLocate(json_doc, item, [*int*], …):

The first argument must be a JSON tree. The second argument is the item to be located. The item to be located can be a constant or a json item. Constant values must be equal in type and value to be found. This is "shallow equality" – strings, integers and doubles won't match.

This function returns the json path to the located item or null if it is not found. For example:

```
select JsonLocate('{"AUTHORS":[{"FN":"Jules", "LN":"Verne"},
{"FN":"Jack", "LN":"London"}]}' json_, 'Jack') Path;
```

This query returns:

Path
$.AUTHORS[1].FN

The path syntax is the new one, the same used in JSON CONNECT tables.

By default, the path of the first occurrence of the item is returned. The third parameter can be used to specify the occurrence whose path is to be returned. For instance:

```
select
JsonLocate('[45,28,[36,45],89]',45) first,
JsonLocate('[45,28,[36,45],89]',45,2) second,
JsonLocate('[45,28,[36,45],89]',45.0) `wrong type`,
JsonLocate('[45,28,[36,45],89]','[36,45]' json_) json;
```

first	second	wrong type	json
$[0]	$[2][1]	<null>	$[2]

For string items, the comparison is case sensitive by default. However, it is possible to specify a string to be compared case insensitively by giving it an alias beginning by “ci”:

```
select JsonLocate('{"AUTHORS":[{"FN":"Jules", "LN":"Verne"},
{"FN":"Jack", "LN":"London"}]}' json_, 'VERNE' ci) Path;
```

Path
$.AUTHORS[0].LN

Json_Locate_All(json_doc, item, [depth]):

The first argument must be a JSON item. The second argument is the item to be located. This function returns the paths to all locations of the item as an array of strings. For example:

```
select Json_Locate_All('[[45,28],[[36,45],89]]',45);
```

This query returns:

All paths
["$[0][0]","$[1][0][1]"]

The returned array can be applied other functions. For instance, to get the number of occurrences of an item in a json tree, you can do:

```
select
JsonGet_Int(Json_Locate_All('[[45,28],[[36,45],89]]',45),
'$[#]') "Nb of occurs";
```

The displayed result:

Nb of occurs
2

If specified, the third integer argument set the depth to search in the document. This means the maximum items in the paths. This value defaults to 10 but can be increased for complex documents or reduced to set the maximum wanted depth of the returned paths.

JsonContains(json_doc, item [, *int*])

This function can be used to check whether an item is contained in a document. Its arguments are the same than the ones of the *JsonLocate* function; only the return value changes. The integer returned value is 1 is the item is contained in the document or 0 otherwise.

JsonContains_Path(json_doc, path)

This function can be used to check whether a Json path is contained in the document. The integer returned value is 1 is the path is contained in the document or 0 otherwise.

Json_Item_Merge(json_doc1, json_doc2)

This function merges two arrays or two objects. For arrays, this is done by adding to the first array all the values of the second array. For instance:

```
select Json_Item_Merge(Json_Make_Array('a','b','c'),
Json_Make_Array('d','e','f')) as "Result";
```

The function returns:

Result
["a","b","c","d","e","f"]

For objects, the pairs of the second object are added to the first object if the key does not exist yet in it; otherwise the pair of the first object is set with the value of the matching pair of the second object. For instance:

```
select Json_Item_Merge(Json_Make_Object(1 "a", 2 "b", 3 "c"),
Json_Make_Object(4 "d",5 "b",6 "f")) as "Result";
```

The function returns:

Result
{"a":1,"b":5,"c":3,"d":4,"f":6}

Json_Get_Item(json_doc, path)

This function returns a subset of the json document passed as first argument. The second argument is the json path of the item to be returned and should be one returning a json item (terminated by a '*'). If not, the function will try to make it right but this is not fool proof. For instance:

```
select Json_Get_Item(Json_Make_Object('foo' as "first",
Json_Make_Array('a', 33) as "json_second"), 'second') as
"item";
```

The correct path should have been 'second.*' but in this simple case the function was able to make it right. The returned item:

item
["a",33]

Note: The array is aliased "json_second" to indicate it is a json item and avoid escaping it. However, the "json_" prefix is skipped when making the object and must not be added to the path.

{JsonGet_String | JsonGet_Int | JsonGet_Real}(json_doc, path, [prec])

The first argument should be a JSON item. If it is a string with no alias, it will be converted as a json item. The second argument is the path of the item to be located in the first argument and returned, eventually converted according to the used function. For example:

```
select
JsonGet_String('{"qty":7,"price":29.50,"garanty":null}','price') "String",
JsonGet_Int('{"qty":7,"price":29.50,"garanty":null}','price') "Int",
JsonGet_Real('{"qty":7,"price":29.50,"garanty":null}','price') "Real";
```

This query returns:

String	Int	Real
29.50	29	29.500000000000000

The function JsonGet_Real can be given a third argument to specify the number of decimal digits of the returned value. For instance:

```
select
JsonGet_Real('{"qty":7,"price":29.50,"garanty":null}','price',4) "Real";
```

This will return:

Real
29.5000

The given path can specify all operators for arrays except the "expand" [*] operator. For instance:

```
select
JsonGet_Int(Json_Make_Array(45,28,36,45,89), '[4]') "Rank",
JsonGet_Int(Json_Make_Array(45,28,36,45,89), '[#]') "Number",
JsonGet_String(Json_Make_Array(45,28,36,45,89), '[","]') "Concat",
JsonGet_Int(Json_Make_Array(45,28,36,45,89), '[+]') "Sum",
JsonGet_Real(Json_Make_Array(45,28,36,45,89), '[!]', 2) "Avg";
```

The result:

Rank	Number	Concat	Sum	Avg
89	5	45,28,36,45,89	243	48.60

Json_{Set | Insert | Update}_Item(json_doc, [item, path [, val, path …]])

These functions insert or update data in a JSON document and return the result. The value/path pairs are evaluated left to right. The document produced by evaluating one pair becomes the new value against which the next pair is evaluated.

`Json_Set_Item`	replaces existing values and adds non-existing values.
`Json_Insert_Item`	inserts values without replacing existing values.
`Json_Update_Item`	replaces *only* existing values.

Example:

```
set @j = Json_Make_Array(1, 2, 3, Json_Object_Key('quatre', 4));
select Json_Set_Item(@j, 'foo', '$[1]', 5, '$[3].cinq') as "Set",
Json_Insert_Item(@j, 'foo', '$[1]', 5, '$[3].cinq') as "Insert",
Json_Update_Item(@j, 'foo', '$[1]', 5, '$[3].cinq') as "Update";
```

This query returns:

Set	Insert	Update
[1,"foo",3,{"quatre":4,"cinq":5}]	[1,2,3,{"quatre":4,"cinq":5}]	[1,"foo",3,{"quatre":4}]

Json_File(file_name, [arg2, [arg3]], …):

The first argument must be a file name. This function returns the text of the file that is supposed to be a json file. If only one argument is specified, the file text is returned without being parsed. Up to two additional arguments can be specified:

- A string argument is the path to the sub-item to be returned.
- An integer argument specifies the *pretty* format value of the file.

This function is chiefly used to get the json item argument of other json functions from a json file. For instance, supposing the file *tb.json* is:

```
{ "_id" : 5, "type" : "food", "ratings" : [ 5, 8, 9 ] }
{ "_id" : 6, "type" : "car", "ratings" : [ 5, 9 ] }
```

Extracting a value from it can be done with a query such as:

```
select JsonGet_String(Json_File('tb.json', 0), '$[1].type')
"Type";
```

This query returns:

Type
car

However, we'll see that, most of the time, it is better to use Jbin_File or to directly specify the file name in queries. In particular this function should not be used for queries that must modify the json item because, even the modified json is returned the file itself would be unchanged.

Jfile_Make(json_doc, arg2, [arg3], …):

The first argument must be a json item[21]. Following arguments are a string file name and an integer pretty value (defaulting to 2) in any order. This function makes a json file containing the first argument item.

The returned string value is the made file name. If not specified as argument, the file name can in some cases be retrieved from the first argument; in such case the file itself is modified.

This function can be used to create or format of a json file. For instance supposing we want to format the file tb.json, this can be done with the query:

```
select Jfile_Make('tb.json' jfile_, 2);
```

The *tb.json* file will be changed to:

[21] If it is just a string, *Jfile_Make* will try its best to see if it is a json item or an input file name.

```
[
  {
    "_id": 5,
    "type": "food",
    "ratings": [
      5,
      8,
      9
    ]
  },
  {
    "_id": 6,
    "type": "car",
    "ratings": [
      5,
      9
    ]
  }
]
```

The "JBIN" return type

Almost all the functions returning a json string – whose name begins with *Json_* -- have a counterpart whose name begins with *Jbin_*. This is as well for performance (speed and memory) as for a better control of what the functions should do.

This is due to the way CONNECT UDF's work internally. The *Json* functions, when receiving json strings as parameters, parse them and construct a binary tree in memory. They work on this tree and before returning; serialize this tree to return a new json string.

If the json document is big, this can be quite consuming in time and storage. It is all right when one simple json function is called – it must be done anyway – but is a waste of time and memory when json functions are used as parameters to other json functions.

To avoid multiple serializing and parsing, the *Jbin* functions should be used as parameters to other functions. Indeed, they do not serialize the memory document tree, but return a structure allowing the receiving function to have direct access to the memory tree. This saves the serialize-parse steps otherwise needed to pass the argument and remove the need to reallocate the memory of the binary tree, which by the way is 6 to 7 times the size of the json string. For instance:

```
select
Json_Make_Object(Jbin_Array_Add(Jbin_Array('a','b','c'), 'd')
as "Jbin_foo") as "Result";
```

This query returns:

Result
{"foo":["a","b","c","d"]}

Here the binary json tree allocated by Jbin_Array is completed by Jbin_Array_Add and Json_Make_Object and serialized only once to make the final result string. It would be serialized and parsed two more times if using "Json" functions.

Note that Jbin results are recognized as such because aliased beginning by "Jbin_". This is why in Json_Make_Object function the alias is specified a "Jbin_foo".

What happens if not recognized as such? These functions are declared as returning a string and to take care of this, the returned structure begins with a zero-terminated string. For instance,[22]:

```
select Jbin_Array('a','b','c');
```

This query replies:

Jbin_Array('a','b','c')
Binary Json array

Note: When testing, the tree returned by a "Jbin" function can be seen using the *Json_Serialize* function whose unique parameter must be a "Jbin" result. For instance:

```
select Json_Serialize(Jbin_Array('a','b','c'));
```

This query returns:

Json_Serialize(Jbin_Array('a','b','c'))
["a","b","c"]

Note: For this simple example, this is equivalent to using the Json_Make_Array function.

Using a file as json UDF first argument

We have seen that many json UDFs can have an additional argument not yet described. This is in the case where the json item argument was referring to a file. Then the additional integer argument is the pretty value of the json file. It matters only when the first argument is just a file name (to make the UDF understand this argument is a file name, it should be aliased with a name beginning with jfile_) or if the function modifies the file, in which case it will be rewritten with this pretty format.

The json item is made by extracting from the file the required part. This can be the whole file but more often only some of it. There are two ways to specify the sub-item of the file to be used:

1. Specifying it if the Json_File or Jbin_File arguments.
2. Specifying it in the receiving function (not possible for all functions)

It doesn't make any difference when the Jbin_File is used but it does with Json_File. For instance:

```
select Jfile_Make('{"a":1, "b":[44, 55]}' json_, 'test.json');
select Json_Array_Add(Json_File('test.json', 'b'), 66);
```

The second query returns:

Json_Array_Add(Json_File('test.json', 'b'), 66)
[44,55,66]

It just returns the – modified -- subset returned by the Json_File function, while the query:

```
select Json_Array_Add(Json_File('test.json'), 66, 'b');
```

returns what was received from Json_File with the modification made on the subset.

[22] Not all client programs are able to recognize zero terminated string. In particular the default MariaDB client is prone to display some rubbish behind the returned string.

Json_Array_Add(Json_File('test.json'), 66, 'b')
{"a":1,"b":[44,55,66]}

Note that in both case the *test.json* file is not modified. This is because the *Json_File* function returns a string representing all or part of the file text but no information about the file name. This is all right to check what would be the effect of the modification to the file.

However, to have the file modified, use the Jbin_File function or directly give the file name. Jbin_File returns a structure containing all these information, the file name, a pointer to the file parsed tree and eventually a pointer to the subset when a path is given as a second argument:

```
select Json_Array_Add(Jbin_File('test.json', 'b'), 66);
```

This query returns:

Json_Array_Add(Jbin_File('test.json', 'b'), 66)
test.json

This time the file is modified. This can be checked with:

```
select Json_File('test.json', 3);
```

Json_File('test.json', 3)
{"a":1,"b":[44,55,66]}

The reason why the first argument is returned by such a query is because of tables such as:

```
create table tb (
n int key,
jfile_cols char(10) not null);
insert into tb values(1,'test.json');
```

In this table, the *jfile_cols* column just contains a file name. If we update it by:

```
update tb set jfile_cols = select
Json_Array_Add(Jbin_File('test.json', 'b'), 66)
where n = 1;
```

This is the *test.json* file that must be modified, not the *jfile_cols* column. This can be checked by:

```
select JsonGet_String(jfile_cols, '[1]:*') from tb;
```

JsonGet_String(jfile_cols, '[1]:*')
{"a":1,"b":[44,55,66]}

Note: It was an important facility to name the second column of the table beginning by “jfile_” so the json functions knew it was a file name without obliging to specify an alias in the queries.

Using “Jbin” to control what the query execution does

This is applying, in particular, when acting on json files. We have seen that a file was not modified when using the *Json_File* function as an argument to a modifying function because the modifying function just received a copy of the json file. This is not true when using the *Jbin_File* function that does not serialize the binary document and make it directly accessible. Also, as we have seen earlier, json functions that modify their first file parameter modify the file and return the file name. This is done by directly serializing the internal binary document as a file.

However, the “Jbin” counterpart of these functions does not serialize the binary document and thus does not modify the json file. For example, let us compare these two queries:

```
/* First query */
select Json_Make_Object(Jbin_Object_Add(Jbin_File('bt2.json'),
4 as "d") as "Jbin_bt1") as "Result";

/* Second query */
select Json_Make_Object(Json_Object_Add(Jbin_File('bt2.json'),
4 as "d") as "Jfile_bt1") as "Result";
```

Both queries return:

Result
{"bt1":{"a":1,"b":2,"c":3,"d":4}}

In the first query *Jbin_Object_Add* does not serialize the document (no “Jbin” functions do) and *Json_Make_Object* just returns a serialized modified tree. Consequently, the file *bt2.json* is not modified. This query is all right to copy a modified version of the json file without modifying it.

However, in the second query *Json_Object_Add* does modify the json file and returns the file name. The *Json_Make_Object* function receives this file name, read and parses the file, makes an object from it and returns the serialized result. This modification can be done willingly but can be an unwanted side effect of the query.

Therefore, using “Jbin” argument functions, in addition to being faster and using less memory, is also safer when dealing with json files that should not be modified.

Using JSON as Dynamic Columns

The JSON NOSQL language has all the features to be used as an alternative to dynamic columns. For instance, the MariaDB documentation gives as an example of dynamic columns:

```
create table assets (
    ->   item_name varchar(32) primary key, /* A common attribute for all items */
    ->   dynamic_cols  blob  /* Dynamic columns will be stored here */
    -> );
Query OK, 0 rows affected (0.05 sec)

INSERT INTO assets VALUES
    ->   ('MariaDB T-shirt', COLUMN_CREATE('color', 'blue', 'size', 'XL'));
Query OK, 1 row affected (0.04 sec)

INSERT INTO assets VALUES
    ->   ('Thinkpad Laptop', COLUMN_CREATE('color', 'black', 'price', 500));
Query OK, 1 row affected (0.00 sec)

SELECT item_name, COLUMN_GET(dynamic_cols, 'color' as char) AS color FROM assets;
+-----------------+-------+
| item_name       | color |
+-----------------+-------+
| MariaDB T-shirt | blue  |
| Thinkpad Laptop | black |
+-----------------+-------+
2 rows in set (0.09 sec)

/* Remove a column: */
UPDATE assets SET dynamic_cols=COLUMN_DELETE(dynamic_cols, "price")
    -> WHERE COLUMN_GET(dynamic_cols, 'color' as char)='black';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

/* Add a column: */
UPDATE assets SET dynamic_cols=COLUMN_ADD(dynamic_cols, 'warranty', '3 years')
    -> WHERE item_name='Thinkpad Laptop';
Query OK, 1 row affected (0.00 sec)
```

```
Rows matched: 1  Changed: 1  Warnings: 0

/* You can also list all columns, or (starting from MariaDB 10.0.1)
   get them together with their values in JSON format: */
SELECT item_name, column_list(dynamic_cols) FROM assets;
+-----------------+---------------------------+
| item_name       | column_list(dynamic_cols) |
+-----------------+---------------------------+
| MariaDB T-shirt | `size`,`color`            |
| Thinkpad Laptop | `color`,`warranty`        |
+-----------------+---------------------------+
2 rows in set (0.00 sec)

SELECT item_name, COLUMN_JSON(dynamic_cols) FROM assets;
+-----------------+---------------------------------------+
| item_name       | COLUMN_JSON(dynamic_cols)             |
+-----------------+---------------------------------------+
| MariaDB T-shirt | {"size":"XL","color":"blue"}          |
| Thinkpad Laptop | {"color":"black","warranty":"3 years"} |
+-----------------+---------------------------------------+
2 rows in set (0.00 sec)
```

The same result can be obtained with json columns using the json UDF's:

```
/* JSON equivalent */
create table jassets (
    ->   item_name varchar(32) primary key, /* A common attribute for all items */
    ->   json_cols varchar(512)  /* Jason columns will be stored here */
    -> );
Query OK, 0 rows affected (0.04 sec)

INSERT INTO jassets VALUES
    ->   ('MariaDB T-shirt', Json_Make_Object('blue' color, 'XL' size));
Query OK, 1 row affected (0.00 sec)

INSERT INTO jassets VALUES
    ->   ('Thinkpad Laptop', Json_Make_Object('black' color, 500 price));
Query OK, 1 row affected (0.00 sec)

SELECT item_name, JsonGet_String(json_cols, 'color') AS color FROM jassets;
+-----------------+-------+
| item_name       | color |
+-----------------+-------+
| MariaDB T-shirt | blue  |
| Thinkpad Laptop | black |
+-----------------+-------+
2 rows in set (0.00 sec)

/* Remove a column: */
UPDATE jassets SET json_cols=Json_Object_Delete(json_cols, 'price')
    -> WHERE JsonGet_String(json_cols, 'color')='black';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

/* Add a column */
UPDATE jassets SET json_cols=Json_Object_Add(json_cols, '3 years' warranty)
    -> WHERE item name='Thinkpad Laptop';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

/* You can also list all columns, or get them together with their values in JSON
format: */
SELECT item_name, Json_Object_List(json_cols) FROM jassets;
+-----------------+-----------------------------+
| item_name       | Json_Object_List(json_cols) |
+-----------------+-----------------------------+
| MariaDB T-shirt | ["color","size"]            |
| Thinkpad Laptop | ["color","warranty"]        |
+-----------------+-----------------------------+
2 rows in set (0.00 sec)

SELECT item_name, json_cols FROM jassets;
+-----------------+---------------------------------------+
| item_name       | json_cols                             |
+-----------------+---------------------------------------+
| MariaDB T-shirt | {"color":"blue","size":"XL"}          |
```

```
| Thinkpad Laptop | {"color":"black","warranty":"3 years"} |
+-----------------+----------------------------------------+
2 rows in set (0.00 sec)
```

However, using JSON brings features not exiting in dynamic columns:

- Use of a language used by many implementation and developers.
- Full support of arrays, currently missing from dynamic columns.
- Access of subpart of json by JPATH that can include calculations on arrays.
- Possible references to json files.

With more experience, additional UDF's can be easily written to support new needs.

Converting Tables to JSON

The JSON UDF's and the direct Jpath "*" facility are powerful tools to convert table and files to the JSON format. For instance, the file *biblio3.json* we used previously can be obtained by converting the *xsample.xml* file. This can be done like this:

```
create table xj1 (row varchar(500) field_format='*')
engine=connect table_type=JSON file_name='biblio3.json'
option_list='jmode=2';
```

And then:

```
insert into xj1
select json_object_nonull(ISBN, language LANG, SUBJECT,
json_array_grp(json_make_object(authorfn FIRSTNAME, authorln
LASTNAME)) json_AUTHOR, TITLE,
json_make_object(translated PREFIX, json_make_object(tranfn
FIRSTNAME, tranln LASTNAME) json_TRANSLATOR) json_TRANSLATED,
json_make_object(publisher NAME, location PLACE)
json_PUBLISHER, date DATEPUB) from xsampall2 group by isbn;
```

The xj1 table rows will directly receive the Json object made by the SELECT statement used in the INSERT statement and the table file will be made as shown (xj1 is pretty=2 by default). Its mode is Jmode=2 because the values inserted are strings even if they denote json objects.

Another way to do this is to create a table describing the file format we want before the *biblio3.json* file existed:

```
create table jsampall3 (
ISBN char(15),
LANGUAGE char(2) field_format='LANG',
SUBJECT char(32),
AUTHORFN char(128) field_format='AUTHOR:[X]:FIRSTNAME',
AUTHORLN char(128) field_format='AUTHOR:[X]:LASTNAME',
TITLE char(32),
TRANSLATED char(32) field_format='TRANSLATOR:PREFIX',
TRANSLATORFN char(128) field_format='TRANSLATOR:FIRSTNAME',
TRANSLATORLN char(128) field_format='TRANSLATOR:LASTNAME',
PUBLISHER char(20) field_format='PUBLISHER:NAME',
LOCATION char(20) field_format='PUBLISHER:PLACE',
DATE int(4) field_format='DATEPUB')
engine=CONNECT table_type=JSON file_name='biblio3.json';
```

and to populate it by:

```
insert into jsampall3 select * from xsampall;
```

This is a simpler method. However, the issue is that this method cannot handle the multiple column values. This is why we inserted from *xsampall* not from *xsampall2*. How can we add the missing multiple authors in this table? Here again we must create a utility table able to handle JSON strings.

```
create table xj2 (ISBN char(15), author varchar(150)
field_format='AUTHOR:*') engine=connect table_type=JSON
file_name='biblio3.json' option_list='jmode=1';

update xj2 set author =
(select json_array_grp(json_make_object(authorfn FIRSTNAME,
authorln LASTNAME)) from xsampall2 where isbn = xj2.isbn);
```

Voilà !

Converting json files

We have seen that json files can be formatted differently depending on the *pretty* option. In particular, big data files should be formatted with *pretty* equal to 0 when used by a CONNECT json table. The best and simplest way to convert a file from one format to another is to use the *Jfile_Make* function. Indeed, this function makes a file of specified format using the syntax:

```
Jfile_Make(json_document, [file_name], [pretty]);
```

The file name is optional when the json document comes from a *Jbin_File* function because the returned structure makes it available. For instance, to convert back the json file *tb.json* to pretty= 0, this can be simply done by:

```
select Jfile_Make(Jbin_File('tb.json'), 0);
```

Performance Consideration

MySQL and PostgreSQL have a JSON data type that is not just text but an internal encoding of JSON data. This is to save parsing time when executing JSON functions. Of course, the parse must be done anyway when creating the data and serializing must be done to output the result.

CONNECT directly works on character strings impersonating JSON values with the need of parsing them all the time but with the advantage of working easily on external data. Generally, this is not too penalizing because JSON data are often of some or reasonable size. The only case where it can be a serious problem is when working on a big JSON file.

Then, the file should be formatted or converted to pretty=0. Also, it should not be used directly by JSON UDFs because they do parse the whole file even when only a subset is used. Instead it should be use by a JSON table created on it. Indeed, JSON tables do not parse the whole document but just the item corresponding to the row they are working on. In addition, indexing can be used by the table as explained previously in this document.

Generally speaking, the maximum flexibility offered by CONNECT is by using JSON tables and JSON UDFs together. Some things are better handled by tables, other by UDFs. The tools are there but it is up to you to discover the best way to resolve your problems.

Specifying a JSON table Encoding

An important feature of JSON is that strings should be coded in UNICODE. As a matter of facts, all examples we have found on the Internet seemed to be just ASCII. This because UNICODE is generally encoded in JSON files using UTF8 or UTF16 or UTF32.

To specify the required encoding, just use the DATA_CHARSET CONNECT option.

Retrieving JSON data from MongoDB

Classified as a NoSQL database program, MongoDB uses JSON-like documents (BSON) grouped in collections. The simplest, and only one in previous versions of CONNECT, way to access MongoDB data was to export a collection to a JSON file. This produces a file having the pretty=0 format. View as SQL, a collection is a table and documents are table rows.

Since CONNECT version 1.6, it is now possible to directly access MongoDB collections. This is the purpose of the MONGO table type described later. However, JSON tables can also do it in a somewhat different way[23].

It is achieved by specifying the MongoDB connection URI while creating the table. For instance:

```
create or replace table jinvent (
_id char(24) not null,
item char(12) not null,
instock varchar(300) not null field_format='instock.*')
engine=connect table_type=JSON tabname='inventory' lrecl=512
connection='mongodb://localhost:27017';
```

In this statement, the *file_name* option was replaced by the *connection* option. It is the URI enabling to retrieve data from a local or remote MongoDB server. The *tabname* option is the name of the MongoDB collection that will be used and the *dbname* option could have been used to indicate the database containing the collection (it defaults to the current database).

The way it works is that the documents retrieved from MongoDB are serialized and CONNECT use them as if they were read from a file. This implies serializing by MongoDB and parsing by CONNECT and is not the best performance wise. CONNECT tries its best to reduce the data transfer when a query contains a reduced column list and/or a where clause. This way makes all the possibilities of the JSON table type available, such as calculated arrays.

However, to work on big JSON collations, using the MONGO table type is the preferred way.

Note: JSON tables using the MongoDB access accept the specific MONGO options *colist*, *filter*, *driver* and *pipe*. They are described in the MONGO table chapter.

INI Table Type

The INI type is the one of “configure” or “initializing” files often met on Windows machines. For instance, let us suppose you have a contact file *contact.ini* such as:

```
[BER]
name=Bertrand
forename=Olivier
address=21 rue Ferdinand Buisson
city=Issy-les-Mlx
zipcode=92130
tel=09.54.36.29.60
cell=06.70.06.04.16

[WEL]
name=Schmitt
forename=Bernard
hired=19/02/1985
address=64 tiergarten strasse
city=Berlin
zipcode=95013
tel=03.43.377.360
```

[23] Providing the MONGO support is installed as described for MONGO tables.

```
[UK1]
name=Smith
forename=Henry
hired=08/11/2003
address=143 Blum Rd.
city=London
zipcode=NW1 2BP
```

CONNECT let you view it as a table in two different ways.

Column layout

The first way is to regard it as a table having one line per section, the columns being the keys you want to display. In this case, the create statement could be:

```
create table contact (
contact char(16) flag=1,
name char(20),
forename char(32),
hired date date_format='DD/MM/YYYY',
address char(64),
city char(20),
zipcode char(8),
tel char(16))
engine=CONNECT table_type=INI file_name='contact.ini';
```

The column that will contain the section name can have any name but must specify `flag=1`. All other column must have the names of the keys we want to display (case insensitive). The type can be character or numeric depending on the key value type, and the length is the maximum expected length for the key value. Once done, for instance:

```
select contact, name, hired, city, tel from contact;
```

This statement will display the file in tabular format.

contact	name	hired	city	tel
BER	Bertrand	NULL	Issy-les-Mlx	09.54.36.29.60
WEL	Schmitt	1985-02-19	Berlin	03.43.377.360
UK1	Smith	2003-11-08	London	NULL

Only the keys defined in the create statements are visible; keys that do not exist in a section are displayed as null if the column was declared as nullable, or pseudo null (blank for character, 1/1/70 for dates, and 0 for numeric) for columns declared NOT NULL.

All relational operations can be applied to this table. The table (and the file) can be updated, inserted and conditionally deleted. The only constraint is that when inserting values, the section name must be the first in the list of values.

Note 1: When inserting, if a section already exists, no new section will be created but the new values will be added or replace those of the existing section. Thus, the following two commands are equivalent:

```
update contact set forename = 'Harry' where contact = 'UK1';
insert into contact (contact,forename) values('UK1','Harry');
```

Note 2: Because sections represent one line, a DELETE statement on a section key will delete the whole section.

Row layout

To be a good candidate for tabular representation, an INI file should have often the same keys in all sections. In practice, many files commonly found on computers, such as the *win.ini* file of the Windows directory or the *my.ini* file cannot be viewed that way because each section have different keys. In this case, a second way is to regard the file as a table having one row per section key and whose columns can be the section name, the key name and the key value.

For instance, let us define the table:

```
create table xcont (
section char(16) flag=1,
keyname char(16) flag=2,
value char(32))
engine=CONNECT table_type=INI file_name='contact.ini'
option_list='Layout=Row';
```

In this statement, the "Layout" option sets the display format, Column by default or anything else not beginning by 'C' for row layout display. The names of the three columns can be freely chosen. The Flag option gives the meaning of the column. Specify `flag=1` for the section name and `flag=2` for the key name. Otherwise, the column will contain the key value. Once done, the command:

```
select * from xcont;
```

Will display the following result:

section	keyname	value
BER	name	Bertrand
BER	forename	Olivier
BER	address	21 rue Ferdinand Buisson
BER	city	Issy-les-Mlx
BER	zipcode	92130
BER	tel	09.54.36.29.60
BER	cell	06.70.06.04.16
WEL	name	Schmitt
WEL	forename	Bernard
WEL	hired	19/02/1985
WEL	address	64 tiergarten strasse
WEL	city	Berlin
WEL	zipcode	95013
WEL	tel	03.43.377.360
UK1	name	Smith
UK1	forename	Henry
UK1	hired	08/11/2003
UK1	address	143 Blum Rd.
UK1	city	London
UK1	zipcode	NW1 2BP

Note: When processing an INI table, all section names are retrieved in a buffer of 8K bytes. For a big file having many sections, this size can be increased using for example:

```
option_list='secsize=16K';
```

External Table Types

Because so many ODBC and JDBC drivers exist and only the main ones have been heavily tested, these table types cannot be ranked as STABLE. Use them with care in production applications.

These types can be used to access tables belonging to the current or another data base server. Five types are currently provided:

ODBC To be used to access tables from a database management system providing an ODBC connector. ODBC is a standard of Microsoft and is currently available on Windows. On Linux, it can also be used provided a specific application emulating ODBC is installed. Currently only unixODBC is supported.

JDBC To be used to access tables from a database management system providing a JDBC connector. JDBC is an Oracle standard implemented in Java and principally meant to be used by Java applications. Using it directly from C or C++ application seems to be almost impossible due to an Oracle bug still not fixed. However, this can be achieved using a Java wrapper class used as an interface between C++ and JDBC. On another hand, JDBC is available on all platforms and operating systems.

Mongo To access MongoDB collections as tables via their MongoDB Java Driver or MongoDB C Driver. Mongo is available with all MariaDB distributions supporting Java (JDBC). However, because this requires both MongoDB and the C Driver to be installed and operational, this table type is not currently available in binary distributions but only when compiling MariaDB from source.

MySQL This type is the preferred way to access tables belonging to another MySQL or MariaDB server. It uses the MySQL API to access the external table. Even though this can be obtained using the FEDERATED(X) plugin, this specific type is used internally by CONNECT because it also makes it possible to access tables belonging to the current server.

PROXY Internally used by some table types to access other tables from one table.

External Table Specification

The four main external table types – ODBC, JDBC, MONGO and MYSQL – are specified giving the following information about:

1. The data source. This is specified in the CONNECTION option.
2. The remote table or view to access. This can be specified within the connection string or using specific CONNECT options.
3. The column definitions. This can be also left to CONNECT to find them using the discovery MariaDB feature.

The way this works is by establishing a connection to the external data source and by sending it an SQL statement[24] enabling to execute the original query. To enhance performance, it is necessary to have the remote data source do the maximum processing. This is needed in particular to reduce the amount of data returned by the data source.

This is why, for SELECT queries, CONNECT uses the *cond_push* MariaDB feature to retrieve the maximum of the WHERE clause of the original query that can be added to the query sent to the data source. This is automatic and does not require anything to be done by the user.

However, more can be done. In addition to access a remote table, CONNECT offers the possibility to specify what the remote server must do. This is done by specifying it as a view in the SRCDEF option. For example:

```
CREATE TABLE custnum ENGINE=CONNECT TABLE_TYPE=XXX
CONNECTION='connecton string'
SRCDEF='select pays as country, count(*) as customers from custnum
     group by pays';
```

[24] Or its equivalent using API functions for MONGO.

Doing so, the GROUP BY clause will be done by the remote server reducing considerably the amount of data sent back on the connection.

This may even be increased by adding to the SRCDEF part of the "compatible" part of the query WHERE clauses like this is done for table-based tables. Note that for MariaDB, this table has two columns, *country* and *customers*. Supposing the original query is:

```
SELECT * FROM custnum WHERE (country = 'UK' OR country = 'USA') AND
     customers > 5;
```

How can we make the WHERE clause be added to the sent SRCDEF? There are many problems:

1. Where to include the additional information.
2. What about the use of alias.
3. How to know what will be a WHERE clause or a HAVING clause.

The first problem is solved by preparing the srcdef view to receive clauses. The above example SRCDEF becomes:

```
SRCDEF='select pays as country, count(*) as customers from custnum
     where %s group by pays having %s';
```

The **%s** in the SRCDEF are place holders for eventual compatible parts of the original query WHERE clause. If the SELECT query does not specify a WHERE clause or a not acceptable WHERE clause, place holders will be filled by dummy clauses (1=1).

The other problems must be solved by adding to the create table a list of columns that must be translated because they are aliases or/and aliases on aggregate functions that must become a HAVING clause. For example, in this case:

```
CREATE TABLE custnum ENGINE=CONNECT TABLE_TYPE=XXX
CONNECTION='connecton string'
SRCDEF='select pays as country, count(*) as customers from custnum
     where %s group by pays having %s'
OPTION_LIST='Alias=customers=*count(*);country=pays';
```

This is specified by the ALIAS option, to be used in the option list. It is made of a semi-colon separated list of items containing:

1. The local column name (alias in the remote server)
2. An equal sign.
3. An eventual '*' indicating this is column correspond to an aggregate function.
4. The remote column name.

With this information, CONNECT will be able to make the query sent to the remote data source:

```
select pays as country, count(*) as customers from custnum
where (pays = 'UK' OR pays = 'USA') group by country having
count(*) > 5
```

Note: Some data source, including MySQL and MariaDB, accept aliases in the HAVING clause. In that case, the alias option could have been specified as:

```
OPTION_LIST='Alias=customers=*;country=pays';
```

Note: Another option exists, PHPOS, enabling to specify what place holders are present and in what order. To be specified as "W", "WH", "H", or "HW". It is rarely used because by default CONNECT can set it from the SRCDEF content. The only cases it is needed is when the SRCDEF contains only a HAVING place holder or when the HAVING place holder occurs before the WHERE place holder, which can occur on queries containing joins. CONNECT cannot handle more than one place holder of each type.

Note: SRCDEF is not available for MONGO tables, but other way of achieving this exist and are describes in the MONGO table type chapter.

ODBC Table Type: Accessing Tables from another DBMS

ODBC (Open Database Connectivity) is a standard API for accessing database management systems (DBMS). CONNECT uses this API to access data contained in other DBMS without having to implement a specific application for each one. An exception is the access to MySQL that should be done using the MYSQL table type.

Note: On Linux, unixODBC must be installed.

CONNECT ODBC Tables

These tables are given the type ODBC. For example, if a "Customers" table is contained in an Access™ database you can define it with a command such as:

```
create table Customer (
CustomerID varchar(5),
CompanyName varchar(40),
ContactName varchar(30),
ContactTitle varchar(30),
Address varchar(60),
City varchar(15),
Region varchar(15),
PostalCode varchar(10),
Country varchar(15),
Phone varchar(24),
Fax varchar(24))
engine=connect table_type=ODBC block_size=10
tabname='Customers'
Connection='DSN=MS Access Database;DBQ=C:/Program
Files/Microsoft Office/Office/1033/FPNWIND.MDB;';
```

Tabname option defaults to the table name. It is required if the source table name is different from the name of the CONNECT table. Note also that for some data sources this name is case sensitive.

Often, because CONNECT can retrieve the table description using ODBC catalog functions, the column definitions can be unspecified. For instance, this table can be simply created as:

```
create table Customer engine=connect table_type=ODBC
block_size=10 tabname='Customers'
Connection='DSN=MS Access Database;DBQ=C:/Program
Files/Microsoft Office/Office/1033/FPNWIND.MDB;';
```

Note however that this process defines the columns according to what is returned by the data source for the *SQLColumns* function. This can be column types not supported by connect that are translated per what is said in Data type conversion page16~~19~~. Also, some data sources fail to return all details such as the number of decimals. This why you may want to use discovery to generate a table create statement that you can retrieve by the *show create table* statement and further edit to meet your specific needs.

The BLOCK_SIZE specification will be used later to set the RowsetSize when retrieving rows from the ODBC table. A reasonably large RowsetSize can greatly accelerate the fetching process.

If you specify the column description, the column names of your table must exist in the data source table. However, you are not obliged to define all the data source columns and you can change the order of the

columns. Some type conversion can also be done if appropriate. For instance, to access the FireBird sample table EMPLOYEE, you could define your table as:

```
create table empodbc (
EMP_NO smallint(5) not null,
FULL_NAME varchar(37) not null),
PHONE_EXT varchar(4) not null,
HIRE_DATE date,
DEPT_NO smallint(3) not null,
JOB_COUNTRY varchar(15),
SALARY double(12,2) not null)
engine=CONNECT table_type=ODBC tabname='EMPLOYEE'
connection='DSN=firebird';
```

This definition ignores the FIRST_NAME, LAST_NAME, JOB_CODE, and JOB_GRADE columns. It places the FULL_NAME last column of the original table in second position. The type of the HIRE_DATE column was changed from *timestamp* to *date* and the type of the DEPT_NO column was changed from *char* to *integer*.

Currently, some restrictions apply to ODBC tables:

1. Cursor type is forward only (sequential reading) by default.
2. Prior to version 1.04 no indexing of ODBC tables (do not specify any columns as key) However, because CONNECT can often add a where clause to the query sent to the data source, indexing will be used by the data source if it supports it. (Remote indexing is available with version 1.04)
3. This version of CONNECT ODBC supports SELECT and INSERT. UPDATE and DELETE are also supported in a somewhat restricted way (see below). For other operations, use an ODBC table with the EXECSRC option (see below) to directly send proper command to the data source.

Random Access of ODBC Tables
In CONNECT version 1.03 ODBC tables are not indexable. Version 1.04 adds remote indexing facility to the ODBC table type.

However, some queries require random access to an ODBC table; for instance, when it is joined to another table or used in an order by queries applied to a long column or large tables.

There are several ways to enable random (position) access to a CONNECT ODBC table. They are depending on the following table options:

Option	Type	Used For
Block_Size	Integer	Specifying the rowset size.
Memory*	Integer	Storing the result set in memory.
Scrollable*	Boolean	Using a scrollable cursor.

(*): To be specified in the option_list.

When dealing with small tables, the simpler way to enable random access is to specify a rowset size equal of larger than the table size (or the result set size if a push down where clause is used). This means that the whole result is in memory on the first FETCH and CONNECT will use it for further positional accesses.

Another way to have the result set in memory is to use the MEMORY option. This option can be set to the following values:

0 No memory used.
1 Memory size required is calculated during the first sequential table read. The allocated memory is filled during the second sequential read. Then the table rows are retrieved from the memory.

2 A first query is executed to get the result set size and the needed memory is allocated. It is filled on the first sequential reading. Then random access of the table is possible.

In the case of an ORDER BY query, MariaDB firstly retrieves the sequentially the result set and the position of each records. Often the sort can be done from the result set if it is not too big. But if too big, or if it implies some "long" columns, only the positions are sorted and MariaDB retrieves the final result from the table read in random order. To be able to retrieve it from memory after the first sequential read, the MEMORY option must be set to 2.

For tables too large to be stored in memory remains the possibility to use a scrollable cursor. However, scrollable cursors are not supported by all data sources.

With CONNECT version 1.04, another way to provide random access is to specify some columns to be indexed. This should be done only when the corresponding column of the source table is also indexed. This should be used for tables too large to be stored in memory and is similar to the remote indexing used by the MYSQL table type and by the FEDERATED engine.

There remains the possibility to extract requested data from the external table and to construct another table of any file format from the data source. For instance, to construct a fixed formatted DOS table containing the CUSTOMER table data, create the table as:

```
create table Custfix engine=connect File_name='customer.txt'
 table_type=fix block_size=20 as select * from customer;
```

Now you can use *custfix* for fast database operations on the copied *customer* table data.

Retrieving Data from a Spread Sheet
ODBC can also be used to create tables based on tabular data belonging to an Excel spread sheet:

```
create table XLCONT
engine=CONNECT table_type=ODBC tabname='CONTACT'
Connection='DSN=Excel Files;DBQ=D:/Ber/Doc/Contact_BP.xls;';
```

This supposes that a tabular zone of the sheet including column headers is defined as a table named CONTACT or using a "named reference". Refer to the Excel documentation for how to specify tables inside sheets. Once done, you can ask:

```
select * from xlcont;
```

This will extract the data from Excel and display:

Nom	Fonction	Societe
Boisseau Frederic		9 Telecom
Martelliere Nicolas		Vidal SA (Groupe UBM)
Remy Agathe		Price Minister
Du Halgouet Tanguy		Danone
Vandamme Anna		GDF
Thomas Willy		Europ Assistance France
Thomas Dominique		Acoss (DG des URSSAF)
Thomas Berengere	Responsable SI Decisionnel	DEXIA Credit Local
Husy Frederic	Responsable Decisionnel	Neuf Cegetel
Lemonnier Nathalie	Directeur Marketing Client	Louis Vuitton
Louis Loic	Reporting International Decisionnel	Accor
Menseau Eric		Orange France

Here again, the columns description was left to CONNECT when creating the table.

Multiple ODBC tables
The concept of multiple table can be extended to ODBC tables when they are physically represented by files, for instance to Excel or Access tables. The condition is that the connect string for the table must contain a field DBQ=*filename*, in which wildcard characters can be included as for multiple=1 tables in their filename. For instance, a table contained in several Excel files CA200401.xls, CA200402.xls, …CA200412.xls can be created by a command such as:

```
create table ca04mul (Date char(19), Operation varchar(64),
Debit double(15,2), Credit double(15,2))
engine=CONNECT table_type=ODBC multiple=1
qchar= '"' tabname='bank account'
connection='DSN=Excel Files;DBQ=D:/Ber/CA/CA2004*.xls;';
```

Providing that in each file the applying information is internally set for Excel as a table named “bank account”. This extension to ODBC does not support *multiple*=2. The *qchar* option was specified to make the identifiers quoted in the select statement sent to ODBC, in particular when the table or column names contain blanks, to avoid SQL syntax errors.

Caution: Avoid accessing tables belonging to the currently running MariaDB server via the MySQL ODBC connector. This may not work and cause the server to be restarted.

Performance consideration
To avoid extracting entire tables from an ODBC source, which can be a lengthy process, CONNECT extracts the “compatible” part of query WHERE clauses and add it to the ODBC query. Compatible means that it must be understood by the data source. In particular, clauses involving scalar functions are not kept because the data source may have different functions than MariaDB or use a different syntax. Of course, clauses involving sub-select are also skipped. This will transfer eventual indexing to the data source.

Take care with clauses involving string items because you may not know whether they are treated by the data source as case sensitive or case insensitive. In doubt, make your queries as if the data source was processing strings as case sensitive to avoid incomplete result.

Using ODBC Tables inside correlated sub-queries
Unlike not correlated subqueries that are executed only once, correlated subqueries are executed many times. It is what ODBC calls a "requery". Several methods can be used by CONNECT to deal with this depending on the setting of the MEMORY or SCROLLABLE Boolean options:

Option	Description
Default	Implementing "requery" by discarding the current result set and re-submitting the query (as MFC does)
Memory=1 or 2	Storing the result set in memory as MYSQL tables do.
Scrollable=Yes	Using a scrollable cursor.

Note: the MEMORY and SCROLLABLE options must be specified in the OPTION_LIST.

Because the table is accessed several times, this can make queries last very long except for small tables and is almost unacceptable for big tables. However, if it cannot be avoided, using the memory method is the best choice and can be more than four times faster than the default method. If it is supported by the driver, using a scrollable cursor is slightly slower than using memory but can be an alternative to avoid memory problems when the sub-query returns a huge result set.

If the result set is of reasonable size, it is also possible to specify the BLOCK_SIZE option equal or slightly larger than the result set. The whole result set being read on the first fetch, can be accessed many times without having to do anything else.

Another good workaround is to replace within the correlated sub-query the ODBC table by a local copy of it because MariaDB is often able to optimize the query and to provide a very fast execution.

Accessing specified views

Instead of specifying a source table name via the TABNAME option, it is possible to retrieve data from a "view" whose definition is given in a new option SRCDEF. For instance:

```
CREATE TABLE custnum (
country varchar(15) NOT NULL,
customers int(6) NOT NULL)
ENGINE=CONNECT  TABLE_TYPE=ODBC BLOCK_SIZE=10
CONNECTION='DSN=MS Access Database;DBQ=C:/Program Files/Microsoft
    Office/Office/1033/FPNWIND.MDB;'
SRCDEF='select country, count(*) as customers from customers group by
    country';
```

Or simply, because CONNECT can retrieve the returned column definition:

```
CREATE TABLE custnum ENGINE=CONNECT TABLE_TYPE=ODBC BLOCK_SIZE=10
CONNECTION='DSN=MS Access Database;DBQ=C:/Program Files/Microsoft
    Office/Office/1033/FPNWIND.MDB;'
SRCDEF='select country, count(*) as customers from customers group by
    country';
```

Then, when executing for instance:

```
select * from custnum where customers > 3;
```

The processing of the group by is done by the data source, which returns only the generated result set on which only the where clause is performed locally. The result:

country	customers
Brazil	9
France	11
Germany	11
Mexico	5
Spain	5
UK	7
USA	13
Venezuela	4

This makes possible to let the data source do complicated operations, such as joining several tables or executing procedures returning a result set. This minimizes the data transfer through ODBC.

CRUD Operations

The only data modifying operations are the INSERT, UPDATE and DELETE commands. They can be executed successfully only if the data source database or tables are not read/only.

INSERT Command

When inserting values to an ODBC table, local values are used and sent to the ODBC table. This does not make any difference when the values are constant but in a query such as:

```
insert into t1 select * from t2;
```

Where t1 is an ODBC table, t2 is a locally defined table that must exist on the local server. Besides, it is a good way to create a distant ODBC table from local data.

CONNECT does not directly support INSERT commands such as:

```
insert into t1 values(2,'Deux') on duplicate key update msg = 'Two';
```

Indeed, the "on duplicate key update" part of it is ignored, and will result in error if the key value is duplicated.

UPDATE and DELETE Commands

Unlike the INSERT command, UPDATE and DELETE are supported in a simplified way. They are just rephrased to correspond to the data source syntax and sent to the data source for execution. Let us suppose we created the table:

```
create table tolite (
id int(9) not null,
nom varchar(12) not null,
nais date default null,
rem varchar(32) default null)
ENGINE=CONNECT TABLE_TYPE=ODBC tabname='lite'
CONNECTION='DSN=SQLite3 Datasource;Database=test.sqlite3'
CHARSET=utf8 DATA_CHARSET=utf8;
```

We can populate it by:

```
insert into tolite values(1,'Toto',now(),'First'),
(2,'Foo','2012-07-14','Second'),(4,'Machin','1968-05-30','Third');
```

The function **now()** will be executed by MariaDB and it returned value sent to the ODBC table.

Let us see what happens when updating the table. If we use the query:

```
Update tolite set nom = 'Gillespie' where id = 10;
```

CONNECT will rephrase the command as:

```
UPDATE lite SET nom = 'Gillespie' WHERE id = 10;
```

What it did is just to replace the local table name by the remote table name and change all the back ticks to blanks or the data source identifier quoting characters if QUOTED is specified. Then this command will be sent to the data source to be executed by it.

This is simpler and can be faster than doing a positional update using a cursor and commands such as "select … for update of …" that are not supported by all data sources. However, there are some restrictions that must be understood due to the way it is handled by MariaDB.

1. MariaDB does not know about all the above. The command will be parsed as if it were to be executed locally. Therefore, it must respect the MySQL syntax.
2. Being executed by the data source, the (rephrased) command must also respect the data source syntax.
3. All data referenced in the SET and WHERE clause belongs to the data source.

This is possible because both MariaDB and the data source are using the SQL language. But you must use only the basic features that are part of the core SQL language. For instance, keywords like IGNORE or LOW_PRIORITY will cause syntax error with many data source.

Scalar function names also can be different, which severely restrict the use of them. For instance:

```
update tolite set nais = now() where id = 2;
```

This will not work with SQLite3, the data source returning an "unknown scalar function" error message. Note that in this particular case, you can rephrase it to:

```
update tolite set nais = date('now') where id = 2;
```

This understood by both parsers, and even if this function would return NULL executed by MariaDB, it does return the current date when executed by SQLite3. But this begins to become too trickery so to overcome all these restrictions, and permit to have all types of commands executed by the data source, CONNECT provides a specific ODBC table subtype described now.

Sending commands to a Data Source

This can be done using a special subtype of ODBC tables. Let us see this on an example:

```
create table crlite (
command varchar(128) not null,
number int(5) not null flag=1,
message varchar(255) flag=2)
engine=connect table_type=odbc
connection='Driver=SQLite3 ODBC
     Driver;Database=test.sqlite3;NoWCHAR=yes'
option_list='Execsrc=1';
```

The key points in this create statement are the EXECSRC option and the column definition.

The EXECSRC option tells that this table will be used to send a command to the data source. Most of the sent commands do not return result set. Therefore, the table columns are used to specify the command to be executed and to get the result of the execution. The name of these columns can be chosen arbitrarily, their function coming from the FLAG value:

Flag=0: The command to execute.
Flag=1: The affected rows, or -1 in case of error, or the result number of column if the command returns a result set.
Flag=2: The returned (eventually error) message.

How to use this table and specify the command to send? By executing a command such as:

```
select * from crlite where command = 'a command';
```

This will send the command specified in the WHERE clause to the data source and return the result of its execution. The syntax of the WHERE clause must be exactly as shown above. For instance:

```
select * from crlite where command =
'CREATE TABLE lite (
ID integer primary key autoincrement,
name char(12) not null,
birth date,
rem varchar(32))';
```

This command returns:

command	number	message
CREATE TABLE lite (ID integer primary key autoincrement, name…	0	Affected rows

Now we can create a standard ODBC table on the newly created table:

```
CREATE TABLE tlite
ENGINE=CONNECT TABLE_TYPE=ODBC tabname='lite'
CONNECTION='Driver=SQLite3 ODBC
     Driver;Database=test.sqlite3;NoWCHAR=yes'
CHARSET=utf8 DATA_CHARSET=utf8;
```

We can populate it directly using the supported INSERT statement:

```
insert into tlite(name,birth) values('Toto','2005-06-12');
insert into tlite(name,birth,rem) values('Foo',NULL,'No ID');
insert into tlite(name,birth) values('Truc','1998-10-27');
insert into tlite(name,birth,rem) values('John','1968-05-30','Last');
```

And see the result:

```
select * from tlite;
```

ID	name	birth	rem
1	Toto	2005-06-12	NULL
2	Foo	NULL	No ID
3	Truc	1998-10-27	NULL
4	John	1968-05-30	Last

Any command, for instance UPDATE, can be executed from the *crlite* table:

```
select * from crlite where command =
'update lite set birth = ''2012-07-14'' where ID = 2';
```

This command returns:

command	number	message
update lite set birth = '2012-07-15' where ID = 2	1	Affected rows

Let us verify it:

```
select * from tlite where ID = 2;
```

ID	name	birth	rem
2	Foo	2012-07-15	No ID

The syntax to send a command is rather strange and may seem unnatural. It is possible to use an easier syntax by defining a stored procedure such as:

```
create procedure send_cmd(cmd varchar(255))
MODIFIES SQL DATA
select * from crlite where command = cmd;
```

Now you can send commands like this:

```
call send_cmd('drop tlite');
```

This is possible only when sending one single command.

Sending several commands together
Grouping commands uses an easier syntax and is faster because only one connection is made for the all of them. To send several commands in one call, use the following syntax:

```
select * from crlite where command in (
'update lite set birth = ''2012-07-14'' where ID = 2',
'update lite set birth = ''2009-08-10'' where ID = 3');
```

When several commands are sent, the execution stops at the end of them or after a command that is in error. To continue after *n* errors, set the option maxerr = *n* (0 by default) in the option list.

Note 1: It is possible to specify the SRCDEF option when creating an EXECSRC table. It will be the command sent by default when a WHERE clause is not specified.

Note 2: Most data sources do not allow sending several commands separated by semi-colons.

Note 3: Quotes inside commands must be escaped. This can be avoided by using a different quoting character than the one used in the command

Note 4: The sent command must obey the data source syntax.

Note 5: Sent commands apply in the specified database. However, they can address any table within this database, or belonging to another database, using the name syntax *schema.tabname*.

Connecting to a Data Source

There are two ways to establish a connection to a data source:

1. Using SQLDriverConnect and a Connection String
2. Using SQLConnect and a Data Source Name (DSN)

The first way uses a Connection String whose components describe what is needed to establish the connection. It is the most complete way to do it and by default CONNECT uses it.

The second way is a simplified way in which ODBC is just given the name of a DSN that must have been defined to ODBC or UnixODBC and that contains the necessary information to establish the connection. Only the user name and password can be specified out of the DSN specification.

(1) Defining the Connection String

Using the first way, the connection string must be specified. This is sometimes the most difficult task when creating ODBC tables because, depending on the operating system and the data source, this string can widely differ.

The format of the ODBC Connection String is:

```
connection-string::= empty-string[;] | attribute[;] | attribute; connection-string
empty-string ::=
attribute ::= attribute-keyword=attribute-value | DRIVER=[{]attribute-value[}]
attribute-keyword ::= DSN | UID | PWD | driver-defined-attribute-keyword
attribute-value ::= character-string
driver-defined-attribute-keyword = identifier
```

Where character-string has zero or more characters; identifier has one or more characters; attribute-keyword is not case-sensitive; attribute-value may be case-sensitive; and the value of the DSN keyword does not consist solely of blanks. Due to the connection string grammar, keywords and attribute values that contain the characters []{}(),;?*=!@ should be avoided. The value of the DSN keyword cannot consist only of blanks, and should not contain leading blanks. Because of the grammar of the system information, keywords and data source names cannot contain the backslash (\) character. Applications do not have to add braces around the attribute value after the DRIVER keyword unless the attribute contains a semicolon (;), in which case the braces are required. If the attribute value that the driver receives includes the braces, the driver should not remove them, but they should be part of the returned connection string.

ODBC Defined Connection Attributes

The ODBC defined attributes are:

- **DSN** - the name of the data source to connect to[25].
- **DRIVER** - the name of the driver to connect to. You can use this in DSN-less connections.

- [25] You must create the DSN before attempting to refer to it. You create new DSNs through the ODBC Administrator (Windows), ODBCAdmin (unixODBC's GUI manager) or by including its definition in the odbc.ini file.

- **FILEDSN** - the name of a file containing the connection attributes.
- **UID/PWD** - any username and password the database requires for authentication.
- **SAVEFILE** - request the DSN attributes are saved in this file.

Other attributes are DSN dependent attributes. The connection string can give the name of the driver in the DRIVER field or the data source in the DSN field (attention! meet the spelling and case) and has other fields that depend on the data source. When specifying a file, the DBQ field must give the **full** path and name of the file containing the table. Refer to the specific ODBC connector documentation for the exact syntax of the connection string.

(2) Using a Predefined DSN
This is done by specifying in the option list the Boolean option “UseDSN” as yes or 1. In addition, string options “user” and “password” can be optionally specified in the option list

When doing so, the connection string just contains the name of the predefined Data Source. For instance:

```
CREATE TABLE tlite ENGINE=CONNECT TABLE_TYPE=ODBC tabname='lite'
CONNECTION='SQLite3 Datasource'
OPTION_LIST='UseDSN=Yes,User=me,Password=mypass';
```

Note: the connection data source name (limited to 32 characters) should not be preceded by “DSN=”.

ODBC Tables on Linux/Unix: In order to use ODBC tables, you will need to have unixODBC installed. Additionally, you will need the ODBC driver for your foreign server's protocol. For example, for MS SQL Server or Sybase, you will need to have FreeTDS installed.

Make sure the user running mysqld (usually the mysql user) has permission to the ODBC data source configuration and the ODBC drivers.

If you get an error on Linux/Unix when using TABLE_TYPE=ODBC:

```
Error Code: 1105 [unixODBC][Driver Manager]Can't open lib
     '/usr/cachesys/bin/libcacheodbc.so' : file not found
```

You must make sure that the user running mysqld (usually "mysql") has enough permission to load the ODBC driver library. It can happen that the driver file does not have enough read privileges (use chmod to fix this), or loading is prevented by SELinux configuration.

Try this command in a shell to check if the driver had enough permission:

```
sudo -u mysql ldd /usr/cachesys/bin/libcacheodbc.so
```

SELinux

SELinux can cause various problems. If you think SELinux is causing problems, check the system log (e.g. /var/log/messages) or the audit log (e.g. /var/log/audit/audit.log).

mysqld can't load some executable code, so it can't use the ODBC driver.

Example error:

```
Error Code: 1105 [unixODBC][Driver Manager]Can't open lib
'/usr/cachesys/bin/libcacheodbc.so' : file not found
```

Audit log:

```
type=AVC msg=audit(1384890085.406:76): avc: denied { execute }
for pid=1433 comm="mysqld"
path="/usr/cachesys/bin/libcacheodbc.so" dev=dm-0 ino=3279212
scontext=unconfined_u:system_r:mysqld_t:s0
tcontext=unconfined_u:object_r:usr_t:s0 tclass=file
```

mysqld can't open TCP sockets on some ports, so it can't connect to the foreign server.

Example error:

```
ERROR 1296 (HY000): Got error 174 '[unixODBC][FreeTDS][SQL
Server]Unable to connect to data source' from CONNECT
```

Audit log:

```
type=AVC msg=audit(1423094175.109:433): avc:  denied  {
name_connect } for  pid=3193 comm="mysqld" dest=1433
scontext=system_u:system_r:mysqld_t:s0
tcontext=system_u:object_r:mssql_port_t:s0 tclass=tcp_socket
```

ODBC Catalog Information

First of all, it must be understood that depending on the version of the used ODBC driver, some additional information on the tables are existing, such as table QUALIFIER or OWNER for old versions, now named CATALOG or SCHEMA since version 3.

CATALOG is apparently rarely used by most data sources, but SCHEMA (formerly OWNER) is and corresponds to the DATABASE information of MySQL.

The issue is that if no schema name is specified, some data sources return information for all schemas while some others only return the information of the "default" schema. In addition, the used "schema" or "database" is sometimes implied by the connection string and sometimes is not. Sometimes, it also can be included in a data source definition.

CONNECT offers two ways to specify this information:

1. When specified, the DBNAME create table option is regarded by ODBC tables as the SCHEMA name.
2. Table names can be specified as "*sch.tab*" or "*cat.sch.tab*" allowing setting the schema and eventual catalog information.

When both are used, the qualified table name has precedence over DBNAME. For instance:

Tabname	DBname	Description
test.t1		The t1 table of the test schema.
test.t1	mydb	The t1 table of the test schema (test has precedence)
t1	mydb	The t1 table of the mydb schema
%.%.%		All tables in all catalogs and all schemas
t1		The t1 table in the default or all schema depending on the DSN
%.t1		The t1 table in all schemas for all DSN
test.%		All tables in the test schema

When creating a standard ODBC table, you should make sure only one source table is specified. Specifying more than one source table must be done only for CONNECT catalog tables (with CATFUNC=tables or columns)

In particular, when column definition is left to the Discovery feature, if tables with the same name are present in several schemas and the schema name is not specified, several columns with the same name will be generated. This will make the creation to fail with a not very explicit error message.

Note: With some ODBC drivers, the DBNAME option or qualified table name is useless because the schema implied by the connection string or the definition of the data source has priority over the specified DBNAME.

Table name case
Another issue when dealing with ODBC tables is the way table and column names are handled regarding of the case.

For instance, Oracle follows to the SQL standard here. It converts non-quoted identifiers to upper case. This is correct and expected. PostgreSQL is not standard. It converts identifiers to lower case. MySQL/MariaDB is not standard. They preserve identifiers on Linux, and convert to lower case on Windows. Think about that if you fail to see a table or a column on an ODBC data source.

JDBC Table Type: Accessing Tables from another DBMS

The JDBC table type is a newly implemented table type and this first version should be regarded as a beta release. However, if the automatic compilation of it is possible after the java JDK was installed, the complete distribution of it is not fully implemented[26] in older versions.

This will require that:

1. The Java SDK is installed on your system.
2. The java wrapper class files are available on your system.
3. And of course, some JDBC drivers exist to be used with the matching DBMS.

Point 2 was made automatic in the newest versions of MariaDB.

Compiling from Source Distribution

Even when the Java JDK has been installed, CMake sometimes cannot find the location where it stands. For instance, on Linux the Oracle Java JDK package might be installed in a path not known by the CMake lookup functions causing error message such as:

```
CMake Error at /usr/share/CMake/Modules/FindPackageHandleStandardArgs.CMake:148 (message):
  Could NOT find Java (missing: Java_JAR_EXECUTABLE Java_JAVAC_EXECUTABLE
  Java_JAVAH_EXECUTABLE Java_JAVADOC_EXECUTABLE)
```

When this happen, provide a Java prefix as a hint on where the package was loaded. For instance, on Ubuntu I was obliged to enter:

```
export JAVA_HOME=/usr/lib/jvm/java-8-oracle
```

After that, the compilation of the CONNECT JDBC type was made successfully.

Compiling the Java source files
They are the source of the java wrapper classes used to access JDBC drivers. In source distribution, they are in the CONNECT source directory.

The default wrapper, JdbcInterface, uses the standard way to get a connection to the drivers via the DriverManager.getConnection method. Other wrappers enable connection to a Data Source, eventually implementing pooling. However, they must be compiled and installed manually.

The available wrappers are:

Wrapper	Description
JdbcInterface	Used to make the connection with available drivers the standard way.
ApacheInterface	Based on the Apache common-dbcp2 package this interface enables making connections to DBCP data sources with any JDBC drivers.
MariadbInterface	Makes connection to a MariaDB data source.

[26] The distributed JdbcInterface.jar file contains the JdbcInterface wrapper only. New versions distribute a JavaWrappers.jar that contains all currently existing wrappers.

Wrapper	Description
MysqlInterface	Makes connection to a Mysql data source. Must be used with a MySQL driver that implements data sources.
OracleInterface	Makes connection to an Oracle data source.
PostgresqlInterface	Makes connection to a Postgresql data source.

The wrapper used by default is specified by the *connect_java_wrapper* session variable and is initially set to **wrappers/JdbcInterface**. The wrapper to use for a table can also be specified in the option list as a *wrapper* option of the "create table" statements.

Note: Conforming java naming usage, class names are preceded by the java package name with a slash separator. However, this is not mandatory for CONNECT that adds the package name if it is missing.

The JdbcInterface wrapper is always usable when Java is present on your machine. Binary distributions have this wrapper already compiled as a JdbcInterface.jar file installed in the plugin directory whose path is automatically included in the class path of the JVM. Recent versions also add a JavaWrappers.jar that contains all wrappers, including those used by the MONGO table type. Therefore, there is no need to worry about their path.

Compiling the ApacheInterface wrapper requires that the Apache common-DBCP2 package be installed. Other wrappers are to be used only with the matching JDBC drivers that must be available when compiling them.

It is a good idea to export all the compiled wrappers in a unique jar file (like the JavaWrappers.jar of the binary distribution).

Installing the jar file in the plugin directory is the best place because it is part of the class path.
Depending on what is installed on your system, the source files can be reduced accordingly.

Setting the required information

Before any operation with a JDBC driver can be made, CONNECT must initialize the environment that will make working with Java possible. This will consist of:

1. Loading dynamically the JVM library module.
2. Creating the Java Virtual Machine.
3. Establishing contact with the java wrapper class.
4. Connecting to the used JDBC driver.

Indeed, the JVM library module is not statically linked to the CONNECT plugin. This is to make it possible to use a CONNECT plugin that has been compiled with the JDBC table type on a machine where the Java SDK is not installed. Otherwise, users not interested in the JDBC table type would be obliged to install the Java SDK on their machine to be able to load the CONNECT storage engine.

JVM Library Location
If the JVM library (jvm.dll on Windows, libjvm.so on Linux) was not placed in the standard library load path, CONNECT cannot find it and must be told where to search for it. This happens in particular on Linux when the Oracle Java package was installed in a private location.

If the JAVA_HOME variable was exported as explained above, CONNECT can sometimes find it using this information. Otherwise, its search path can be added to the LD_LIBRARY_PATH environment variable. But all this is complicated because making environment variables permanent on Linux is painful (many different methods must be used depending on the Linux version and the used shell).

Therefore, CONNECT introduced a new global variable *connect_jvm_path* to store this information. It can be set when starting the server as a command line option or even afterwards **before the first use of the JDBC table type**. For example:

```
set global connect_jvm_path="/usr/lib/jvm/java-8-oracle/jre/lib/i386/client"
```

or[27]:

```
set global connect_jvm_path="/usr/lib/jvm/java-8-oracle/jre/lib/i386/server"
```

Note that this may not be required on Windows because the path to the JVM library can sometimes be found in the registry.

Once this library is loaded, CONNECT can create the required Java Virtual Machine.

Java Class Path
This is the list of paths Java searches when loading classes. With CONNECT, the classes to load will be the java wrapper classes used to communicate with the drivers, and the used JDBC driver classes that are grouped inside jar files. If the ApacheInterface wrapper must be used, the class path must also include all three jars used by the Apache package. If MONGO tables using the MongoDB Java Driver will be used, add also its path to the JDBC paths.

Caution: This class path is passed as a parameter to the Java Virtual Machine (JVM) when creating it and cannot be modified as it is a read only property. In addition, because MariaDB is a multi-threading application, this JVM cannot be destroyed and will be used throughout the entire life of the MariaDB server. Therefore, be sure it is correctly set before you use the JDBC table type for the first time. Otherwise there will be practically no alternative than to shut down the server and restart it.

The path to the wrapper classes must point to the directory containing the *wrappers* sub-directory. If a JdbcInterface.jar file was made, its path is the directory where it is located followed by the jar file name. It is unclear where because this will depend on the installation process. If you start from a source distribution, it can be in the storage/connect directory where the CONNECT source files are or where you moved them or compiled the JdbcInterface.jar file.

For binary distributions, there is nothing to do because the jar files have been installed in the plugin directory whose path is always automatically included in the class path available to the JVM.

Remaining are the paths of all the installed JDBC drivers that you intend to use. Remember that their path must include the jar file itself. Some applications use an environment variable CLASSPATH to contain them. Paths are separated by ':' on Linux and by ';' on Windows.

If the CLASSPATH variable exists and if it is available inside MariaDB, so far so good. You can check this using an UDF function provided by CONNECT that returns environment variable values:

```
create function envar returns string soname 'ha_connect.so';
select envar('CLASSPATH');
```

Most of the time, this will return null or some required files are missing. Therefore, CONNECT introduced a global variable to store this information. The paths specified in this variable will be added and have precedence to the ones, if any, of the CLASSPATH environment variable. As for the jvm path, this variable *connect_class_path* should be specified when starting the server but can also be set before using the JDBC table type for the first time.

The current directory (sql/data) is also placed by CONNECT at the beginning of the class path.

As an example, here is how I start MariaDB when doing tests on Linux:

```
olivier@olivier-Aspire-8920:~$ sudo /usr/local/mysql/bin/mysqld -u root --console --default-storage-engine=myisam --skip-innodb --connect_jvm_path="/usr/lib/jvm/java-8-oracle/jre/lib/i386/server" --connect_class_path="/home/olivier/mariadb/10.1/storage/connect:/media/olivier/SOURCE/mysql-connector-java-6.0.2/mysql-connector-java-6.0.2-bin.jar"
```

[27] The client library is smaller and faster for connection. The server library is more optimized and can be used in case of heavy load usage.

CONNECT JDBC Tables

These tables are given the type JDBC. For instance, supposing you want to access the *boys* table located on an external local or remote database management system providing a JDBC connector:

```
create table boys (
name char(12),
city char(12),
birth date,
hired date);
```

To access this table via JDBC you can create a table such as:

```
create table jboys engine=connect table_type=JDBC tabname=boys
connection='jdbc:mysql://localhost/dbname?user=root';
```

The CONNECTION option is the URL used to establish the connection with the remote server. Its syntax depends on the external DBMS and in this example, is the one used to connect as root to a MySQL or MariaDB local database using the MySQL JDBC connector.

As for ODBC, the columns definition can be omitted and will be retrieved by the discovery process. The restrictions concerning column definitions are the same as for ODBC.

Note: The dbname indicated in the URL corresponds for many DBMS to the catalog information. For MySQL and MariaDB it is the schema (often called database) of the connection.

Using a Federated Server

Alternatively, a JDBC table can specify its connection options via a Federated server. For instance, supposing you have a table accessing an external Postgresql table defined as:

```
create table juuid engine=connect table type=JDBC
tabname=testuuid
connection='jdbc:postgresql:test?user=postgres&password=pwd';c
reate table jt1 engine=connect table_type=JDBC
connection='jdbc:postgresql:mtr' dbname=public tabname=t1
option_list='User=mtr,Password=mtr';
```

You can create a Federated server:

```
create server 'post1' foreign data wrapper 'postgresql' options (
HOST 'localhost',
DATABASE 'mtrtest',
USER 'mtrpostgres',
PASSWORD 'mtrpwd',
PORT 0,
SOCKET '',
OWNER 'rootpostgres');
```

Now the JDBC table can be created by:

```
create table juuid engine=connect table type=JDBC create table
jt1 engine=connect table_type=JDBC connection='post1'
dbname=public tabname=t1estuuid;
```

or by:

```
create table juuid engine=connect table type=JDBC
connection='post1/testuuid'; create table jt1 engine=connect
table_type=JDBC connection='post1/t1' dbname=public;
```

or even by:

```
create table jt1 engine=connect table_type=JDBC
connection='post1/public.t1';
```

In any case, the location of the remote table can be changed in the Federated server without having to alter all the tables using this server.

JDBC needs a URL to establish a connection. CONNECT can construct that URL from the information contained in such Federated server definition when the URL syntax is similar to the one of MySQL, MariaDB or Postgresql. However, some other DBMS's such as Oracle use a different URL syntax. In this case, simply replace the HOST information by the required URL in the Federated server definition. For instance:

```
create server 'oracle' foreign data wrapper 'oracle' options (
HOST 'jdbc:oracle:thin:@localhost:1521:xe',
DATABASE 'SYSTEM',
USER 'system',
PASSWORD 'manager',
PORT 0,
SOCKET '',
OWNER 'SYSTEM');
```

Now you can create an Oracle table with something like this:

```
create table empor engine=connect table_type=JDBC
connection='oracle/HR.EMPLOYEES';
```

Note: Oracle, as Postgresql, does not seem to understand the DATABASE setting as the table schema that must can be specified in the Create Table statement if not the default one.

Connecting to a JDBC driver

When the connection to the driver is established by the JdbcInterface wrapper class, it uses the options that are provided when creating the CONNECT JDBC tables. Inside the default Java wrapper, the driver's main class is loaded by the *DriverManager.getConnection* function that takes three arguments:

URL That is the URL that you specified in the CONNECTION option.
User As specified in the OPTION_LIST or NULL if not specified.
Password As specified in the OPTION_LIST or NULL if not specified.

The URL varies depending on the connected DBMS. Refer to the documentation of the specific JDBC driver for a description of the syntax to use. User and password can also be specified in the option list.

Beware that the database name in the URL can be interpreted differently depending on the DBMS. For MySQL, this is the schema in which the tables are found. However, for Postgresql, this is the catalog and the schema ~~must~~ can be specified using the CONNECT *dbname* option (seems to be 'public' by default).

For instance, a table accessing a Postgresql table via JDBC can be created with a create statement such as:

```
create table jt1 engine=connect table_type=JDBC
connection='jdbc:postgresql://localhost/mtr' dbname=public tabname=t1
option_list='User=mtr,Password=mtr';
```

Often, more parameters are available in the URL, such as the user name and password. Assuming the default host and schema are 'localhost' and 'public' this table can be alternatively created by:

```
create table jt1 engine=connect table_type=JDBC tabname=t1
connection='jdbc:postgresql:mtr?user=mtr&password=mtr';
```

Note: In previous versions of JDBC, to obtain a connection, java first had to initialize the JDBC driver by calling the method `Class.forName`. In this case, see the documentation of your DBMS driver to obtain the name of the class that implements the interface `java.sql.Driver`. This name can be specified as an option DRIVER to be put in the option list. However, most modern JDBC drivers since version 4 are self-loading and do not require this option to be specified. Giving the driver class name is also required to retrieve a driver inside an executable jar file.

The wrapper class also creates some required items and a statement class. Some characteristics of this statement will depend on the options specified when creating the table:

Scrollable[28] Determines the cursor type: no= *forward_only* or yes=*scroll_insensitive*.
Block_size Will be used to set the statement fetch size.

Fetch Size
The fetch size determines the number of rows that are internally retrieved by the driver on each interaction with the DBMS. Its default value depends on the JDBC driver. It is equal to 10 for some drivers but not for the MySQL or MariaDB connectors.

The MySQL/MariaDB connectors retrieve all the rows returned by one query and keep them in a memory cache. This is generally fine in most cases but not when retrieving a large result set that can make the query fail with a memory exhausted exception.

To avoid this, when accessing a big table and expecting large result sets, you should specify the BLOCK_SIZE option to 1 (the only acceptable value). However, a problem remains:

Suppose you execute a query such as:

```
select id, name, phone from jbig limit 10;
```

Not knowing the limit clause, CONNECT sends to the remote DBMS the query:

```
SELECT id, name, phone FROM big;
```

In this query, *big* can be a huge table having million rows. Having correctly specified the block size as 1 when creating the table, the wrapper just reads the 10 first rows and stops. However, when closing the statement, these MySQL/MariaDB drivers must still retrieve all the rows returned by the query. Therefore, the wrapper class when closing the statement also cancels the query to stop that extra reading.

The bad news is that if it works all right for some previous versions of the MySQL driver, it does not work for new versions as well as for the MariaDB driver that apparently ignores the cancel command.

[28] To be specified in the option list.

The good news is that you can use old MySQL driver to access MariaDB databases. It is possible also that this bug will be fixed in future versions of the drivers.

Connection to a Data Source
This is the java preferred way to establish a connection because a data source can keep a pool of connections that can be re-used when necessary. This makes establishing connections much faster once it was done for the first time.

CONNECT provide additional wrappers that are included in the JavaWrappers.jar file. The source of these files ~~are~~is located in the CONNECT source directory. The wrapper to use can be specified in the global variable *connect_java_wrapper*, which defaults to "JdbcInterface".

It can also be specified for a table in the option list by setting the option *wrapper* to its name. For instance:

```
create table jboys
engine=CONNECT table_type=JDBC tabname='boys'
connection='jdbc:mariadb://localhost/connect?user=root&useSSL=false'
option_list='Wrapper=MariadbInterface,Scrollable=1';
```

They can be used instead of the standard JdbcInterface and are using created data sources.

The **Apache** one uses data sources implemented by the Apache-commons-dbcp2 package and can be used with all drivers including those not implementing data sources. However, the Apache package must be installed and its three required jar files accessible via the class path:

1. Commons-dbcp2-2.1.1.jar
2. Commons-pool2-2.4.2.jar
3. Commons-logging-1.2.jar

~~1. commons-dbcp2-2.1.1.jar~~
~~2. commons-pool2-2.4.2.jar~~

Note: the versions numbers can be different on your installation.

The other ones use data sources provided by the matching JDBC driver. There are currently four wrappers to be used with mysql-6.0.2, MariaDB, Oracle and PostgreSQL.

Unlike the class path, the used wrapper can be changed even after the JVM machine was created, providing the required jar files are existing and specified in the class path.

Random Access to JDBC Tables

The same methods described for ODBC tables can be used with JDBC tables.

Note that in the case of the MySQL or MariaDB connectors, because they internally read the whole result set in memory, using the MEMORY option would be a waste of memory. It is much better to specify the use of a scrollable cursor when needed.

Other Operations with JDBC Tables

Except for how the connection string is specified and the table type set to JDBC, all operations with ODBC tables are done for JDBC tables the same way. Refer to the ODBC chapter to know about:

- Accessing specified views (SRCDEF)
- CRUD operations.
- Sending commands to a data source.
- JDBC catalog information.

Note: Some JDBC drivers fail when the global *time_zone* variable is ambiguous, which sometimes happens when it is set to SYSTEM. If so, reset it to a not ambiguous value, for instance:

```
set global time_zone = '+2:00';
```

JDBC specific restrictions

Connecting via data sources created externally (for instance using Tomcat) is not supported yet.

Other restrictions are the same as for the ODBC table type.

Handling the UUID Data Type

PostgreSQL has a native UUID data type, internally stored as BIN(16). This is neither a SQL nor a MariaDB data type. The best we can do is to handle it by its character representation.

UUID will be translated to CHAR(36) when column definitions are set using discovery. Locally a PostgreSQL UUID column will be handled like a CHAR or VARCHAR column. Example:

Using the PostgreSQL table *testuuid* in the text database:

```
 Table « public.testuuid »
 Column | Type | Default
--------+------+--------------------
 id     | uuid | uuid_generate_v4()
 msg    | text |
```

Its column definitions can by queried by:

```
create or replace table juuidcol engine=connect
table_type=JDBC tabname=testuuid catfunc=columns
connection='jdbc:postgresql:test?user=postgres&password=pwd';
```

```
select table_name "Table", column_name "Column", data_type
"Type", type_name "Name", column_size "Size" from juuidcol;
```

This query returns:

Table	Column	Type	Name	Size
testuuid	id	1111	uuid	2147483647
testuuid	msg	12	text	2147483647

Note: PostgreSQL, when a column size is undefined, returns 2147483647, which is not acceptable for MariaDB. CONNECT change it to the value of the *connect_conv_size* session variable. Also, for TEXT columns the data type returned is 12 (SQL_VARCHAR) instead of -1 the SQL_TEXT value.

Accessing this table via JDBC by:

```
CREATE TABLE juuid ENGINE=connect TABLE_TYPE=JDBC TABNAME=testuuid
CONNECTION='jdbc:postgresql:test?user=postgres&password=pwd';
```

it will be created by discovery as:

```
CREATE TABLE `juuid` (
  `id` char(36) DEFAULT NULL,
  `msg` varchar(8192) DEFAULT NULL
) ENGINE=CONNECT DEFAULT CHARSET=latin1
CONNECTION='jdbc:postgresql:test?user=postgres&password=pwd'
`TABLE_TYPE`='JDBC' `TABNAME`='testuuid';
```

Note: 8192 being here the _connect_conv_size_ value.

Let's populate it:

```
insert into juuid(msg) values('First');
insert into juuid(msg) values('Second');
select * from juuid;
```

Result:

id	msg
4b173ee1-1488-4355-a7ed-62ba59c2b3e7	First
6859f850-94a7-4903-8d3c-fc3c874fc274	Second

Here the *id* column values come from the DEFAULT of the PostgreSQL column that was specified as *uuid_generate_v4()*.

It can be set from MariaDB. For instance:

```
insert into juuid
  values('2f835fb8-73b0-42f3-a1d3-8a532b38feca','inserted');
insert into juuid values(NULL,'null');
insert into juuid values('','random');
select * from juuid;
```

Result:

id	msg
4b173ee1-1488-4355-a7ed-62ba59c2b3e7	First
6859f850-94a7-4903-8d3c-fc3c874fc274	Second
2f835fb8-73b0-42f3-a1d3-8a532b38feca	inserted
<null>	null
8fc0a30e-dc66-4b95-ba57-497a161f4180	random

The first insert specifies a valid UUID character representation. The second one set it to NULL. The third one (a void string) generates a Java random UUID. UPDATE commands obey the same specification.

These commands both work:

```
select * from juuid where id = '2f835fb8-73b0-42f3-a1d3-8a532b38feca';
delete from juuid where id = '2f835fb8-73b0-42f3-a1d3-8a532b38feca';
```

However, this one fails:

```
select * from juuid where id like '%42f3%';
```

Saying:

```
1296: Got error 174 'ExecuteQuery:
org.postgresql.util.PSQLException:
ERROR: operator does not exist: uuid ~~ unknown
hint: no operator corresponds to the data name and to the
argument types.
```

because CONNECT cond_push feature added the WHERE clause to the query sent to PostgreSQL:

```
SELECT id, msg FROM testuuid WHERE id LIKE '%42f3%'
```

and the LIKE operator does not apply to UUID in PostgreSQL.

To handle this, a new session variable was added to CONNECT: *connect_cond_push*. It permits to specify if cond_push is enable or not for CONNECT and defaults to 1 (enabled). In this case, you can execute:

```
set connect_cond_push=0;
```

Doing so, the where clause will be executed by MariaDB only and the query will not fail anymore.

Executing the JDBC tests

Four tests exist but they are disabled because requiring some work to localized them according to the operating system and available java package and JDBC drivers and DBMS.

Two of them, **jdbc.test** and **jdbc_new.test**, are accessing MariaDB via JDBC drivers that are contained in a fat jar file that is part of the test. They should be executable without anything to do on Windows; simply adding the option *–enable-disabled* when running the tests.

However, on Linux these tests can fail to locate the JVM library. Before executing them, you should export the JAVA_HOME environment variable set to the prefix of the java installation or export the LD_LIBRARY_PATH containing the path to the JVM lib.

MONGO Table Type: Accessing Collections from MongoDB

Note: The source files of this new type were currently distributed only with MariaDB version 10.1, 10.2 and 10.3. This MONGO type was available only when compiling MariaDB from source with: `cmake -DCONNECT_WITH_MONGO=ON`

Such a version will not be rated GA anymore.

Starting with CONNECT version 1.06.005 that will be distributed with all MariaDB versions released after October 1st 2017, the MONGO table type will be also available with binary distributions supporting JDBC. However, this version of CONNECT being rated as GA (stable), the MONGO table type will be disabled for MariaDB versions 10.0 and 10.1, not being thoroughly tested yet. It is still possible to enable it, for testing or use at user's risk, when compiling MariaDB from source with the option : `CONNECT_WITH_MONGO ON.`

Classified as a NoSQL database program, MongoDB uses JSON-like documents (BSON) grouped in collections. The MONGO type is used to directly access MongoDB collections as tables.
Accessing MongoDB from CONNECT can be done in different ways:

1. As a MONGO table via the MongoDB C Driver.
2. As a MONGO table via the MongoDB Java Driver.
3. As a JDBC table using some commercially available MongoDB JDBC drivers.
4. As a JSON table via the MongoDB C or Java Driver.

Using the MongoDB C Driver
This is currently not available from binary distributions but only for versions compiled from source. The preferred versions of the MongoDB C Driver start from 1.7 because they provide package recognition. What must be done is:

1. Install libbson and the MongoDB C Driver.
2. Configure, compile and install MariaDB.

With earlier versions of the Mongo C Driver, the additional include directories and libraries will have to be specified manually when compiling.

When possible, this is the preferred access way because it does not require all the java path settings et because it is faster than using the java driver.

Using the Mongo Java Driver
This is available with all distributions including JDBC support when compiling from source or from a 1.2 or 10.3 binary distribution[29]. The only additional things to do are:

1. Install the MongoDB Java Driver by downloading its *jar* file. Several versions are available. If possible use the latest version 3 one.
2. Add the path to it in the CLASSPATH environment variable or in the *connect_class_path* variable. This is like what is done to declare JDBC drivers.

Connection is established by new Java wrappers *Mongo3Interface* and *Mongo2Interface*. They are available in a JDBC distribution in the JavaWrappers.jar file. If version 2 of the Java Driver is used, specify "Version=2" in the option list when creating tables.

Using JDBC
See the documentation of the existing commercial JDBC MongoDB drivers.

Using JSON
See the specific chapter of the JSON Table Type.

The following describes the MONGO table type.

CONNECT MONGO Tables

Creating and running MONGO tables requires a connection to a running local or remote MongoDB server.

A MONGO table is defined to access a MongoDB collection. The table rows will be the collection documents. For instance, to create a table based on the MongoDB sample collection *restaurants*, you can do something such as the following:

```
create table resto (
_id varchar(24) not null,
name varchar(64) not null,
cuisine char(200) not null,
borough char(16) not null,
restaurant_id varchar(12) not null)
engine=connect table_type=MONGO tabname='restaurants'
data_charset=utf8 connection='mongodb://localhost:27017';
```

Note: The used driver is by default the C driver if only the MongoDB C Driver is installed and the Java driver if only the MongoDB Java Driver is installed. If both are available, it can be specified by the DRIVER option to be specified in the option list and defaults to C.

Here we did not define all the items of the collection documents but only those that are JSON values. The database is *test* by default. The connection value is the URI used to establish a connection to a local or remote MongoDB server. The value shown in this example corresponds to a local server started with its default port. It is the default connection value for MONGO tables so we could have omit specifying it.

Using discovery is available. This table could have been created by:

```
create table resto
engine=connect table_type=MONGO tabname='restaurants'
data_charset=utf8 option_list='level=-1';
```

Here "level=-1" is used to create only columns that are simple values (no array or object). Without this, with the default value "level=0" the table had been created as:

[29] With a binary distribution that does not enable the MONGO table type, it is possible to access MongoDB using an OEM module. See Appendix B for details.

```
CREATE TABLE `resto` (
  `_id` char(24) NOT NULL,
  `address` varchar(136) NOT NULL,
  `borough` char(13) NOT NULL,
  `cuisine` char(64) NOT NULL,
  `grades` varchar(638) NOT NULL,
  `name` char(98) NOT NULL,
  `restaurant_id` char(8) NOT NULL
) ENGINE=CONNECT DEFAULT CHARSET=latin1 `TABLE_TYPE`='MONGO'
`TABNAME`='restaurants' `DATA_CHARSET`='utf8';
```

In this collection, the *address* column is a JSON object and the column *grades* is a JSON array. Unlike the JSON table type, just specifying the column name with no Jpath result in displaying the JSON representation of them. For instance:

```
select name, address from resto limit 3;
```

name	address
Morris Park Bake Shop	{"building":"1007","coord":[-73.8561,40.8484], "street":"Morris ParkAve", "zipcode":"10462"}
Wendy'S	{"building":"469","coord":[-73.9617,40.6629], "street":"Flatbush Avenue", "zipcode":"11225"}
Reynolds Restaurant	{"building":"351","coord":[-73.9851,40.7677], "street":"West 57Street", "zipcode":"10019"}

MongoDB Dot Notation

To address the items inside object or arrays, specify the Jpath in MongoDB syntax[30]:

```
create table newresto (
_id varchar(24) not null,
name varchar(64) not null,
cuisine char(200) not null,
borough char(16) not null,
street varchar(65) field_format='address.street',
building char(16) field_format='address.building',
zipcode char(5) field_format='address.zipcode',
grade char(1) field_format='grades.0.grade',
score int(4) not null field_format='grades.0.score',
`date` date field_format='grades.0.date',
restaurant_id varchar(255) not null)
engine=connect table_type=MONGO tabname='restaurants'
data_charset=utf8 connection='mongodb://localhost:27017';
```

```
select name, street, score, date from newresto limit 5;
```

name	street	score	date
Morris Park Bake Shop	Morris Park Ave	2	03/03/2014
Wendy'S	Flatbush Avenue	8	30/12/2014
Dj Reynolds Pub And Restaurant	West 57 Street	2	06/09/2014
Riviera Caterer	Stillwell Avenue	5	10/06/2014
Tov Kosher Kitchen	63 Road	20	24/11/2014

[30] If using Discovery, specify the Level option accordingly.

MONGO Specific Options

The MongoDB syntax for Jpath does not allow the CONNECT specific items on arrays. The same effect can still be obtained by a different way. For this, additional options are used when creating MONGO tables.

Option	Type	Description
Colist	String	Options to pass to the MongoDB cursor.
Filter	String	Query used by the MongoDB cursor.
Pipeline*	Boolean	If True, Colist is a pipeline.
Fullarray*Error! Bookmark not defined.[9]	Boolean	Used when creating with Discovery.
Driver*	String	C or Java.
Version*	Integer	The Java Driver version (defaults to 3)

*: To be specified in the option list.

Note: For the content of the three first options, refer to the MongoDB documentation.

Colist Option

Used to pass different options when making the MongoDB cursor used to retrieve the collation documents. One of them is the *projection*, allowing to limit the items retrieved in documents. It is hardly useful because this limitation is made automatically by CONNECT. However, it can be used when using discovery to eliminate the *_id* (or another) column when you are not willing to keep it:

```
create table restest
engine=connect table_type=MONGO tabname='restaurants'
data_charset=utf8 option_list='level=-1'
colist='{"projection":{"_id":0,"limit":5}}';
```

In this example, we added another cursor option, the *limit* option that works like the limit SQL clause. This additional option works only with the C driver. When using the Java driver, colist should be:

```
colist='{"_id":0}';
```

And limit would be specified with select statements.

Filter Option

This option is used to specify a "filter" that works as a where clause on the table. Supposing we want to create a table restricted to the restaurant making English cuisine that are not located in the Manhattan borough, we can do it by:

```
create table english
engine=connect table_type=MONGO tabname='restaurants'
data_charset=utf8
colist='{"projection":{"cuisine":0}}'
filter='{"cuisine":"English","borough":{"$ne":"Manhattan"}}'
option_list='Level=-1';
```

And if we ask:

```
select * from english;
```

This query will return:

_id	borough	name	restaurant_id

58ada47de5a51ddfcd5ee1f3	Brooklyn	The Park Slope Chipshop	40816202
58ada47de5a51ddfcd5ee999	Brooklyn	Chip Shop	41076583
58ada47ee5a51ddfcd5f13d5	Brooklyn	The Monro	41660253
58ada47ee5a51ddfcd5f176e	Brooklyn	Dear Bushwick	41690534
58ada47ee5a51ddfcd5f1e91	Queens	Snowdonia Pub	50000290

Pipeline Option

When this option is specified as true (by YES or 1) the *Colist* option contains a MongoDB pipeline applying to the table collation. This is a powerful mean for doing things such as expanding arrays like we do with JSON tables. For instance:

```
create table resto2 (
name varchar(64) not null,
borough char(16) not null,
date datetime not null,
grade char(1) not null,
score int(4) not null)
engine=connect table_type=MONGO tabname='restaurants'
data_charset=utf8
colist='{"pipeline":[{"$match":{"cuisine":"French"}},{"$unwind
":"$grades"},{"$project":{"_id":0,"name":1,"borough":1,"date":
"$grades.date","grade":"$grades.grade","score":"$grades.score"
}}]}'
option_list='Pipeline=1';
```

In this pipeline "$match" is an early filter, "$unwind" means that the grades array will be expanded (one Document for each array values) and "$project" eliminates the *_id* and *cuisine* columns and gives the Jpath for the date, grade and score columns.

```
select name, grade, score, date from resto2
where borough = 'Bronx';
```

This query replies:

name	grade	score	date
Bistro Sk	A	10	21/11/2014 01:00:00
Bistro Sk	A	12	19/02/2014 01:00:00
Bistro Sk	B	18	12/06/2013 02:00:00

This make possible to get things like we do with JSON tables:

```
select name, avg(score) average from resto2
group by name having average >= 25;
```

Can be used to get the average score inside the *grades* array.

name	average
Bouley Botanical	25,0000
Cheri	46,0000
Graine De Paris	30,0000
Le Pescadeux	29,7500

Fullarray Option
This option, like the Level option, is only interpreted when creating a table with Discovery (meaning not specifying the columns). It tells CONNECT to generate a column for all existing values in the array. For instance, let us see the MongoDB collection *tar* by:

```
create table seetar (
Collection varchar(300) not null field_format='*')
engine=CONNECT table_type=MONGO tabname=tar;
```

The format '*' indicates we want to see the Json documents. This small collection is:

Collection
{"_id":{"$oid":"58f63a5099b37d9c930f9f3b"},"item":"journal","prices":[87.0,45.0,63.0,12.0,78.0]}
{"_id":{"$oid":"58f63a5099b37d9c930f9f3c"},"item":"notebook","prices":[123.0,456.0,789.0]}

The *Fullarray* option can be used here to generate enough columns to see all the prices of the document *prices* array.

```
create table tar
engine=connect table_type=MONGO
colist='{"projection":{"_id":0}}'
option_list='level=1,Fullarray=YES';
```

The table has been created as:

```
CREATE TABLE `tar` (
  `item` char(8) NOT NULL,
  `prices_0` double(12,6) NOT NULL `FIELD_FORMAT`='prices.0',
  `prices_1` double(12,6) NOT NULL `FIELD_FORMAT`='prices.1',
  `prices_2` double(12,6) NOT NULL `FIELD_FORMAT`='prices.2',
  `prices_3` double(12,6) DEFAULT NULL `FIELD_FORMAT`='prices.3',
  `prices_4` double(12,6) DEFAULT NULL `FIELD_FORMAT`='prices.4'
) ENGINE=CONNECT DEFAULT CHARSET=latin1 `TABLE_TYPE`='MONGO'
    `COLIST`='{"projection":{"_id":0}}'
    `OPTION_LIST`='level=1,Fullarray=YES';
```

And is displayed as:

item	prices_0	prices_1	prices_2	prices_3	prices_4
journal	87.00	45.00	63.00	12.00	78.00
notebook	123.00	456.00	789.00	NULL	NULL

CRUD Operations

All modifying operations are supported. However, updating or inserting into arrays must be done in a specific way. Like with the *Fullarray* option, we must have enough columns to specify the array values. For instance, we can create a new table by:

```
create table testin (
n int not null,
m char(12) not null,
surname char(16) not null field_format='person.name.first',
name char(16) not null field_format='person.name.last',
age int(3) not null field_format='person.age',
price_1 double(8,2) field_format='d.0',
price_2 double(8,2) field_format='d.1',
```

```
price_3 double(8,2) field_format='d.2')
engine=connect table_type=MONGO tabname='tin'
connection='mongodb://localhost:27017';
```

Now it is possible to populate it by:

```
insert into testin values
(1789, 'Welcome', 'Olivier','Bertrand',56, 3.14, 2.36, 8.45),
(1515, 'Hello', 'John','Smith',32, 65.17, 98.12, NULL),
(2014, 'Coucou', 'Foo','Bar',20, NULL, 74, 81356);
```

The result will be:

n	m	surname	name	age	price_1	price_2	price_3
1789	Welcome	Olivier	Bertrand	56	3,14	2,36	8,45
1515	Hello	John	Smith	32	65,17	98,12	NULL
2014	Coucou	Foo	Bar	20	NULL	74	81356

Note: If the collection does not exist yet when creating the table and inserting in it, MongoDB creates it automatically.

It can be updated by queries such as:

```
update testin set price_3 = 83.36 where n = 2014;
```

To look how the array is generated, let us create another table:

```
create table tintin (
n int not null,
name char(16) not null field_format='person.name.first',
prices varchar(255) field_format='d')
engine=connect table_type=MONGO tabname='tin';
```

This table is displayed as:

_id	name	prices
1789	Olivier	[3.14, 2.36, 8.45]
1515	John	[65.17, 98.12]
2014	Foo	[<null>, 74.0, 83.36]

Note: This last table can be used to make array calculations like with JSON tables using the JSON UDF functions. For instance:

```
select name, jsonget_real(prices,'[+]') sum_prices,
jsonget_real(prices,'[!]') avg_prices from tintin;
```

This query returns:

name	sum_prices	avg_prices
Olivier	13.95	4.65
John	163.29	81.64
Foo	157,36	78.68

Note: When calculating on arrays, null values are ignored.

Status of MONGO Table Type

This table type is still under development. It has significant advantages over the JSON type to access MongoDB collections. Firstly, the access being direct, tables are always up to date if the collection has been modified by another application. Performance wise, it is much faster than JSON, because most processing is done by MongoDB on BSON, its internal representation of JSON data, which is designed to optimize all operations. Note that using the MongoDB C Driver is about twice as fast as using the MongoDB Java Driver.

Current Restrictions
Option "CATFUNC=tables" is not implemented yet.
Options SRCDEF and EXECSRC do not apply to MONGO tables.

MYSQL Table Type: Accessing MySQL/MariaDB Tables

This table type uses the libmysql API to access a MySQL or MariaDB table or view. This table must be created on the current server or on another local or remote server. This is like what the FEDERATED storage engine provides with some differences.

Currently the FEDERATED like syntax can be used to create such a table, for instance:

```
create table essai (
num integer(4) not null,
line char(15) not null)
engine=CONNECT table_type=MYSQL
connection='mysql://root@localhost/test/people';
```

The connection string can have the same syntax than the one used by FEDERATED[31]:

```
scheme://username:password@hostname:port/database/tablename
scheme://username@hostname/database/tablename
scheme://username:password@hostname/database/tablename
```

However, it can also be mixed with CONNECT standard options. For instance:

```
create table essai (
num integer(4) not null,
line char(15) not null)
engine=CONNECT table_type=MYSQL dbname=test tabname=people
connection='mysql://root@localhost';
```

The pure (deprecated) CONNECT syntax is still accepted:

```
create table essai (
num integer(4) not null,
line char(15) not null)
engine=CONNECT table_type=MYSQL dbname=test tabname=people
option_list='user=root,host=localhost';
```

The specific connection options are:

Option	Default value	Description
Tabname	The table name	The name of the table to access.
Dbname	The current DB name	The database where the table is located.

[31] It can also be specified as a reference to a federated server:

```
connection="connection_one"
connection="connection_one/table_foo"
```

Option	Default value	Description
Host	localhost*	The host of the server, a name or an IP address.
User	The current user	The connection user name.
Password	No password	An optional user password.
Port	The currently used port	The port of the server.
Quoted	0	1 if remote Tabname must be quoted.

*: When the host is specified as "localhost", the connection is established on Linux using Linux sockets. On Windows, the connection is established by default using share memory if it is enabled. If not, the TCP protocol is used. An alternative is to specify the host as "." to use a named pipe connection (if it is enabled). This makes possible to use these table types with server skipping networking.

Caution: Take care not to refer to the MYSQL table itself to avoid an infinite loop!

MYSQL table can refer to the current server as well as to another server. Views can be referred by name or directly giving a source definition, for instance:

```
create table grp engine=connect table_type=mysql
CONNECTION='mysql://root@localhost/test/people'
SRCDEF='select title, count(*) as cnt from employees group by title';
```

When specified, the columns of the MYSQL table must exist in the accessed table with the same name, but can be only a subset of them and specified in a different order. Their type must be a type supported by CONNECT and, if it is not identical to the type of the accessed table matching column, a conversion can be done using the rules given in Data type conversion.

Note: For columns prone to be targeted by a WHERE clause, keep the column type compatible with the source table column type (numeric or character) to have a correct rephrasing of the WHERE clause.

If you do not want to restrict or change the column definition, do not provide it and leave CONNECT get the column definition from the remote server. For instance:

```
create table essai engine=CONNECT table_type=MYSQL
connection='mysql://root@localhost/test/people';
```

This will create the *essai* table with the same columns than the *people* table. If the target table contains CONNECT incompatible type columns, see Data type conversion to know how these columns can be converted or skipped.

Charset Specification

When accessing the remote table, CONNECT sets the connection charset set to the default local table charset as the FEDERATED engine does.

Do not specify a column character set if it is different from the table default character set even when it is the case on the remote table. This is because the remote column is translated to the local table character set when reading it[32]. If it must keep its setting, for instance to UTF8 when containing Unicode characters, specify the local default charset to its character set.

This means that it is not possible to correctly retrieve a remote table if it contains columns having different character sets. A solution is to retrieve it by several local tables, each accessing only columns with the same character set.

[32]This is the default but it can be modified by the setting of the variable 'character_set_results' of the target server.

Indexing of MYSQL Tables

Indexes are rarely useful with MYSQL tables. This is because CONNECT tries to access only the requested rows. For instance, if you ask:

```
select * from essai where num = 23;
```

CONNECT will construct and send to the server the query:

```
SELECT num, line FROM people WHERE num = 23
```

If the *people* table is indexed on *num*, indexing will be used on the remote server. This, in all cases, will limit the amount of data to retrieve on the network.

However, an index can be specified for columns that are prone to be used to join another table to the MYSQL table because there are no where clauses permitting to reduce the fetched rows. For instance:

```
select d.id, d.name, f.dept, f.salary
from loc_tab d straight_join cnc_tab f on d.id = f.id
where f.salary > 10000;
```

If the *id* column of the remote table addressed by the *cnc_tab* MYSQL table is indexed (which is likely if it is a key) you should also index the id column of the MYSQL *cnc_tab* table. If so, using "remote" indexing as does FEDERATED, only the useful rows of the remote table will be retrieved during the join process. However, because these rows are retrieved by separate SELECT statements, this will be useful only when retrieving a few rows of a big table.

In particular, you should not specify an index for columns not used for joining and above all DO NOT index a joined column if it is not indexed in the remote table. This would cause multiple scans of the remote table to retrieve the joined rows one by one.

CRUD Operations

The CONNECT MYSQL type supports SELECT and INSERT and a somewhat limited form of UPDATE and DELETE. The MYSQL type uses similar methods than the ODBC type to implement the INSERT, UPDATE and DELETE commands. Refer to the ODBC chapter for the restrictions concerning them.

For the UPDATE and DELETE commands, there are fewer restrictions because the remote server being a MySQL server, the syntax of the command will be always acceptable by both servers.

For instance, you can freely use keywords like IGNORE or LOW_PRIORITY as well as scalar functions in the SET and WHERE clauses.

However, there is still an issue on multi-table statements. Let us suppose you have a *t1* table on the remote server and want to execute a query such as:

```
update essai as x set line = (select msg from t1 where id = x.num)
     where num = 2;
```

When parsed locally, you will have errors if no *t1* table exists or if it does not have the referenced columns. When *t1* does not exist, you can overcome this issue by creating a local dummy *t1* table:

```
create table t1 (id int, msg char(1)) engine=BLACKHOLE;
```

This will make the local parser happy and permit to execute the command on the remote server. Note however that having a local MYSQL table defined on the remote *t1* table does not solve the problem unless it is also names *t1* locally.

Therefore, to permit to have all types of commands executed by the data source without any restriction, CONNECT provides a specific MYSQL table subtype described now.

Sending commands to a MySQL Server

This can be done like for ODBC or JDBC tables by defining a specific table that will be used to send commands and get the result of their execution.

```
create table send (
command varchar(128) not null,
warnings int(4) not null flag=3,
number int(5) not null flag=1,
message varchar(255) flag=2)
engine=connect table_type=mysql
connection='mysql://user@host/database'
option_list='Execsrc=1,Maxerr=2';
```

The key points in this create statement are the EXECSRC option and the column definition.

The EXECSRC option tells that this table will be used to send commands to the MySQL server. Most of the sent commands do not return result set. Therefore, the table columns are used to specify the command to be executed and to get the result of the execution. The name of these columns can be chosen arbitrarily, their function coming from the FLAG value:

Flag=0: The command to execute (the default)
Flag=1: The number of affected rows, or the result number of columns if the command would return a result set.
Flag=2: The returned (eventually error) message.
Flag=3: The number of warnings.

How to use this table and specify the command to send? By executing a command such as:

```
select * from send where command = 'a command';
```

This will send the command specified in the WHERE clause to the data source and return the result of its execution. The syntax of the WHERE clause must be exactly as shown above. For instance:

```
select * from send where command =
'CREATE TABLE people (
num integer(4) primary key autoincrement,
line char(15) not null';
```

This command returns:

command	warnings	number	message
CREATE TABLE people (num integer(4) primary key aut…	0	0	Affected rows

Sending several commands in one call
It can be faster to execute because there will be only one connection for all of them. To send several commands in one call, use the following syntax:

```
select * from send where command in (
"update people set line = 'Two' where id = 2",
"update people set line = 'Three' where id = 3");
```

When several commands are sent, the execution stops at the end of them or after a command that is in error. To continue after *n* errors, set the option maxerr = *n* (0 by default) in the option list.

Note 1: It is possible to specify the SRCDEF option when creating an EXECSRC table. It will be the command sent by default when a WHERE clause is not specified.

Note 2: Backslashes inside commands must be escaped. Simple quotes must be escaped if the command is specified between simple quotes, and double quotes if it is specified between double quotes.

Note 3: Sent commands apply in the specified database. However, they can address any table within this database.

Note 4: Currently, all commands are executed in mode AUTOCOMMIT.

Retrieving Warnings and Notes
If a sent command causes warnings to be issued, it is useless to resend a "show warnings" command because the MySQL server is opened and closed when sending commands. Therefore, getting warnings requires a specific (and tricky) way.

To indicate that warning text must be added to the returned result, you must send a multi-command query containing "pseudo" commands that are not sent to the server but directly interpreted by the EXECSRC table. These "pseudo" commands are:

Warning	To get warnings
Note	To get notes
Error	To get errors returned as warnings (?)

Note that they must be spelled (case insensitive) exactly as above, no final "s". For instance:

```
select * from send where command in ('Warning','Note',
'drop table if exists try',
'create table try (id int key auto_increment, msg varchar(32) not
    null) engine=aria',
"insert into try(msg) values('One'),(NULL),('Three') ",
"insert into try values(2,'Deux') on duplicate key update msg =
    'Two'",
"insert into try(message) values('Four'),('Five'),('Six')",
'insert into try(id) values(NULL)',
"update try set msg = 'Four' where id = 4",
'select * from try');
```

This can return something like this:

command	warnings	number	message
drop table if exists try	1	0	Affected rows
Note	0	1051	Unknown table 'try'
create table try (id int key auto_increment, msg…	0	0	Affected rows
insert into try(msg) values('One'),(NULL),('Three')	1	3	Affected rows
Warning	0	1048	Column 'msg' cannot be null
insert into try values(2,'Deux') on duplicate key…	0	2	Affected rows
insert into try(msge) values('Four'),('Five'),('Six')	0	1054	Unknown column 'msge' in 'field list'
insert into try(id) values(NULL)	1	1	Affected rows
Warning	0	1364	Field 'msg' doesn't have a default value
update try set msg = 'Four' where id = 4	0	1	Affected rows
select * from try	0	2	Result set columns

The execution continued after the command in error because of the MAXERR option. Normally this would have stopped the execution.

Of course, the last "select" command is useless here because it cannot return the table contain. Another MYSQL table without the EXECSRC option and with proper column definition should be used instead.

Connection Engine Limitations

Data types

There is a maximum key.index length of 255 bytes. You may be able to declare the table without an index and rely on the engine condition pushdown and remote schema.

The following types can't be used:

- BIT
- BINARY
- TINYBLOB, BLOB, MEDIUMBLOB, LONGBLOB
- TINYTEXT, MEDIUMTEXT, LONGTEXT
- Geometry types

Note 1: TEXT is allowed. However, the handling depends on the values given to the connect_type_conv and connect_conv_size system variables, and by default no conversion of TEXT columns is permitted.

Note 2: ENUM and SET types are retrieved as char or varchar types. Unlike with the original table, they cannot be used in clauses involving their numeric value. However, comma separated values from a SET column are a good candidate for XCOL tables.

SQL Limitations

The following SQL queries are not supported

- REPLACE INTO
- INSERT ... ON DUPLICATE KEY UPDATE

CONNECT MYSQL versus FEDERATED

The CONNECT MYSQL table type should not be regarded as a replacement for the FEDERATED(X) engine. The main use of the MYSQL type is to access other engine local tables as if they were CONNECT tables. This was necessary when accessing tables from some CONNECT table types such as TBL, XCOL, OCCUR, or PIVOT that are designed to access CONNECT tables only. When their target table is not a CONNECT table, these types are silently using internally an intermediate MYSQL table.

However, there are cases where you can use MYSQL CONNECT tables yourself, for instance:

1. When the table will be used by a TBL table. This enables you to specify the connection parameters for each sub-table and is more efficient than using a local FEDERATED sub-table.
2. When the desired returned data is directly specified by the SRCDEF option. This is great to let the remote server do most of the job, such as grouping and/or joining tables. This cannot be done with the FEDERATED engine.
3. To take advantage of the *push_cond* facility that adds a where clause to the command sent to the remote table. This restricts the size of the result set and can be crucial for big tables. See the details of this in the ODBC table type.
4. For tables with the EXECSRC option on.
5. When doing tests. For instance, to check a connection string.

If you need multi-table updating, deleting, or bulk inserting on a remote table, you can alternatively use the FEDERATED engine or a “send” table specifying the EXECSRC option on.

PROXY Table Type

A PROXY table is a table that access and read the data of another table or view. For instance, to create a table based on the boys FIX table:

```
create table xboy engine=connect table_type=PROXY tabname=boys;
```

Simply, PROXY being the default type when TABNAME is specified:

```
create table xboy engine=connect tabname=boys;
```

Because the boys table can be directly used, what can be the use of a proxy table? Well, its main use is to be internally used by other table types such as TBL, XCOL, OCCUR, or PIVOT. Indeed, PROXY table are CONNECT tables, meaning that they can be based on tables of any engines and accessed by table types that can to access only CONNECT tables.

Proxy on not CONNECT Tables
When the sub-table is a view or a not CONNECT table, CONNECT internally creates a temporary CONNECT table of MYSQL type to access it. This connection is using the same default parameters than for a MYSQL table. It is also possible to specify them to the PROXY table using in the PROXY declaration the same OPTION_LIST options than for a MYSQL table. Of course, it is simpler and more natural to use directly the MYSQL type in this case.

Normally, the default parameters should enable the PROXY table to reconnect the server. However, an issue is when the current user was logged using a password. The security protocol prevents CONNECT to retrieve this password and requires it to be given in the PROXY table create statement. For instance, adding to it:

```
… option_list='Password=mypass';
```

However, it is often not advisable to write in clear a password that can be seen by all user able to see the table declaration by show create table; in particular, if the table is used when the current user is root. To avoid this, a specific user should be created on the local host that will be used by proxy tables to retrieve local tables. This user can have minimum grant options, for instance SELECT on desired directories, and needs no password. Supposing 'proxy' is such a user, the option list to add will be:

```
… option_list='user=proxy';
```

Using a PROXY Table as a View
A PROXY table can also be used by itself to modify the way a table is viewed. For instance, a proxy table does not use the indexes of the object table. It is also possible to define its columns with different names or type, to use only some of them or to changes their order. For instance:

```
create table city (
city varchar(11),
boy char(12) flag=1,
birth date)
engine=CONNECT tabname=boys;
select * from city;
```

This will display:

city	boy	birth
Boston	John	1986-01-25
Boston	Henry	1987-06-07
San Jose	George	1981-08-10
Chicago	Sam	1979-11-22
Dallas	James	1992-05-13
Boston	Bill	1986-09-11

Here we did not have to specify column format or offset because data are retrieved from the boys table, not directly from the boys.txt file. The flag option of the *boy* column indicates that it corresponds to the first column of the boys table, the *name* column.

Avoiding PROXY table loop
CONNECT can test whether a PROXY, or PROXY based, table refers directly or indirectly to itself. If direct reference can be tested at the table creation, indirect reference can only be tested when executing a query on the table. However, this is possible only for local tables. When using remote tables or views, the problem can occur if the remote table or the view refers back to one of the local tables of the chain. The same caution should be used than when using FEDERATED tables.

Modifying Operations
All INSERT/UPDATE/DELETE operations can be used with proxy tables. However, the same restrictions applying to the source table also apply to the proxy table.

Note: All PROXY and PROXY based table types are not indexable.

Virtual Table Types

These tables have no proprietary data. Some work as "views" to other tables, others retrieve data from the operating system.

XCOL Table Type

XCOL tables are based on another table or view, like PROXY tables. This type can be used when the object table has a column that contains a list of values. In particular, this is the case when the object table has a column of the SET type.

Suppose we have a '*chlist'* table that can be displayed as:

mother	children
Sophia	Vivian, Antony
Lisbeth	Lucy,Charles,Diana
Corinne	
Claude	Marc
Janet	Arthur,Sandra,Peter,John

We can have a different view on these data, where each child will be associated with his/her mother by creating an XCOL table by:

```
CREATE TABLE xchild (
mother char(12) NOT NULL,
child char(12) DEFAULT NULL flag=2
) ENGINE=CONNECT table_type=XCOL tabname='chlist'
option_list='colname=child';
```

The COLNAME option specifies the name of the column receiving the list items. This will return from:

```
select * from xchild;
```

The requested view:

mother	child
Sophia	Vivian
Sophia	Antony
Lisbeth	Lucy
Lisbeth	Charles
Lisbeth	Diana
Corinne	NULL
Claude	Marc
Janet	Arthur
Janet	Sandra
Janet	Peter
Janet	John

Several things should be noted here:

- When the original *children* field is void, what happens depends on the NULL specification of the "multiple" column. If it is nullable, like here, a void string will generate a NULL value. However, if the column is not nullable, no row will be generated at all.
- Blanks after the separator are ignored.

- No copy of the original data was done. Both tables use the same source data.
- Specifying the column definitions in the CREATE TABLE statement is optional. When doing so, the columns are defined with the same name and type than in the target table.
- The flag column option is the base 1 position of the matching column in the target table, to be specified if the column name is changed.

The "multiple" column *child* can be used as any other column. For instance:

```
select * from xchild where substr(child,1,1) = 'A';
```

This will return:

mother	child
Sophia	Antony
Janet	Arthur

If a query does not involve the "multiple" column, no row multiplication will be done. For instance:

```
select mother from xchild;
```

This will just return all the mothers:

mother
Sophia
Lisbeth
Corinne
Claude
Janet

The same occurs with other type of select statements, for instance:

```
select count(*) from xchild;          returns 5;
select count(child) from xchild;      returns 10
select count(mother) from xchild;     returns 5
```

Grouping also gives different result:

```
select mother, count(*) from xchild group by mother;
```

Replies:

mother	COUNT(*)
Claude	1
Corinne	1
Janet	1
Lisbeth	1
Sophia	1

While the query:

```
select mother, count(child) from xchild group by mother;
```

Gives the more interesting result:

mother	COUNT(child)
Claude	1
Corinne	0
Janet	4
Lisbeth	3
Sophia	2

Some more options are available for this table type:

Option	Description
Sep_char	The separator character used in the "multiple" column, defaults to the comma.
Mult	Indicates the max number of multiple items. It is used to internally calculate the max size of the table and defaults to 10. (To be specified in OPTION_LIST).

Using Special Columns with XCOL

Special columns can be used in XCOL tables. The mostly useful one is ROWNUM that gives the rank of the value in the list of values. For instance:

```
CREATE TABLE xchild2 (
rank int NOT NULL SPECIAL=ROWID,
mother char(12) NOT NULL,
child char(12) NOT NULL flag=2
) ENGINE=CONNECT table_type=XCOL tabname='chlist'
option_list='colname=child';
```

This table will be displayed as:

rank	mother	child
1	Sophia	Vivian
2	Sophia	Antony
1	Lisbeth	Lucy
2	Lisbeth	Charles
3	Lisbeth	Diana
1	Claude	Marc
1	Janet	Arthur
2	Janet	Sandra
3	Janet	Peter
4	Janet	John

To list only the first child of each mother you can do:

```
SELECT mother, child FROM xchild2 where rank = 1 ;
```

Returning:

mother	child
Sophia	Vivian
Lisbeth	Lucy
Claude	Marc
Janet	Arthur

However, note the following pitfall: trying to get the names of all mothers having more than 2 children cannot be done by:

```
SELECT mother FROM xchild2 where rank > 2;
```

This is because no row multiplication being done, the rank value is always 1. The correct way to obtain this result is longer but cannot use the ROWNUM column:

```
SELECT mother FROM xchild2 group by mother having count(child) > 2;
```

XCOL tables based on specified views

Instead of specifying a source table name via the TABNAME option, it is possible to retrieve data from a "view" whose definition is given in a new option SRCDEF. For instance:

```
create table xsvars engine=connect table_type=XCOL
srcdef='show variables like "optimizer_switch"'
option_list='Colname=Value';
```

Then, for instance:

```
select value from xsvars limit 10;
```

This will display something like:

value
index_merge=on
index_merge_union=on
index_merge_sort_union=on
index_merge_intersection=on
index_merge_sort_intersection=off
engine_condition_pushdown=off
index_condition_pushdown=on
derived_merge=on
derived_with_keys=on
firstmatch=on

Note: All XCOL tables are read only.

OCCUR Table Type

Like the XCOL table type, OCCUR is an extension to the PROXY type when referring to a table or view having several columns containing the same kind of data. It enables having a different view of the table where the data from these columns are put in a single column, eventually causing several rows to be generated from one row of the object table. For example, supposing we have the *pets* table:

name	dog	cat	rabbit	bird	fish
John	2	0	0	0	0
Bill	0	1	0	0	0
Mary	1	1	0	0	0
Lisbeth	0	0	2	0	0
Kevin	0	2	0	6	0
Donald	1	0	0	0	3

We can create an occur table by:

```
create table xpet (
name varchar(12) not null,
race char(6) not null,
number int not null)
engine=connect table_type=occur tabname=pets
option_list='OccurCol=number,RankCol=race'
Colist='dog,cat,rabbit,bird,fish';
```

When displaying it by

```
select * from xpet;
```

We will get the result:

name	race	number
John	dog	2
Bill	cat	1
Mary	dog	1
Mary	cat	1
Lisbeth	rabbit	2
Kevin	cat	2
Kevin	bird	6
Donald	dog	1
Donald	fish	3

First, the values of the column listed in the Colist option have been put in a unique column whose name is given by the OccurCol option. When several columns have non-null (or pseudo-null) values, several rows are generated, with the other normal columns values repeated.

In addition, an optional special column was added whose name is given by the RankCol option. This column contains the name of the source column from which the value of the OccurCol column comes from. It permits here to know the race of the pets whose number is given in *number*.

This table type permit to make queries that would be more complicated to make on the original tables. For instance, to know who as more than 1 pet of a kind, you can simply ask:

```
select * from xpet where number > 1;
```

You will get the result:

name	race	number
John	dog	2
Lisbeth	rabbit	2
Kevin	cat	2
Kevin	bird	6
Donald	fish	3

Note 1: Like for XCOL tables, no row multiplication for queries not implying the Occur column. It is also possible to use the ROWNUM special column with similar result.

Note 2: Because the OccurCol was declared “not null” no rows were generated for null or pseudo-null values of the column list. If the OccurCol is declared as nullable, rows are also generated for columns containing null or pseudo-null values.

Occur tables can be also defined from views or source definition. CONNECT is also able to generate the column definitions if not specified. For example:

```
create table ocsrc engine=connect table_type=occur
colist='january,february,march,april,may,june,july,august,september,
october,november,december' option_list='rankcol=month,occurcol=day'
srcdef='select ''Foo'' name, 8 january, 7 february, 2 march, 1 april,
 8 may, 14 june, 25 july, 10 august, 13 september, 22 october, 28
 november, 14 december';
```

This table is displayed as:

name	month	day
Foo	january	8
Foo	february	7
Foo	march	2
Foo	april	1
Foo	may	8
Foo	june	14
Foo	july	25
Foo	august	10
Foo	september	13
Foo	october	22
Foo	november	28
Foo	december	14

Note: All OCCUR tables are read only.

PIVOT Table Type

This table type can be used to transform the result of another table or view (called the source table) into a pivoted table along “pivot” and “facts” columns. A pivot table is a great reporting tool that sorts and sums (by default) independent of the original data layout in the source table.

For example, let us suppose you have the following “Expenses” table:

Who	Week	What	Amount
Joe	3	Beer	18.00
Beth	4	Food	17.00
Janet	5	Beer	14.00
Joe	3	Food	12.00
Joe	4	Beer	19.00
Janet	5	Car	12.00
Joe	3	Food	19.00
Beth	4	Beer	15.00
Janet	5	Beer	19.00
Joe	3	Car	20.00
Joe	4	Beer	16.00

Beth	5	Food	12.00
Beth	3	Beer	16.00
Joe	4	Food	17.00
Joe	5	Beer	14.00
Janet	3	Car	19.00
Joe	4	Food	17.00
Beth	5	Beer	20.00
Janet	3	Food	18.00
Joe	4	Beer	14.00
Joe	5	Food	12.00
Janet	3	Beer	18.00
Janet	4	Car	17.00
Janet	5	Food	12.00

Pivoting the table contents using the 'Who' and 'Week' fields for the left columns, and the 'What' field for the top heading and summing the 'Amount' fields for each cell in the new table, gives the following desired result:

Who	Week	Beer	Car	Food
Beth	3	16.00	0.00	0.00
Beth	4	15.00	0.00	17.00
Beth	5	20.00	0.00	12.00
Janet	3	18.00	19.00	18.00
Janet	4	0.00	17.00	0.00
Janet	5	33.00	12.00	12.00
Joe	3	18.00	20.00	31.00
Joe	4	49.00	0.00	34.00
Joe	5	14.00	0.00	12.00

Note that SQL enables you to get the same result presented differently by using the “group by” clause, namely:

```
select who, week, what, sum(amount) from expenses
       group by who, week, what;
```

However, there is no way to get the pivoted layout shown above just using SQL. Even using imbedded SQL programming for some DBMS is not quite simple and automatic.

The Pivot table type of CONNECT makes doing this much simpler.

Using the PIVOT Tables Type

To get the result shown in the example above, just define it as a new table with the statement:

```
create table pivex
engine=connect table_type=pivot tabname=expenses;
```

Now you can use it as any other table, for instance to display the result shown above, just say:

```
select * from pivex;
```

The CONNECT implementation of the PIVOT table type does much of the work required to transform the source table:

1. Finding the "Facts" column, by default the last column of the source table[33].
2. Finding the "Pivot" column, by default the last remaining column.
3. Choosing the aggregate function to use, "SUM" by default.
4. Constructing and executing the "Group By" on the "Facts" column, getting its result in memory.
5. Getting all the distinct values in the "Pivot" column and defining a "Data" column for each.
6. Spreading the result of the intermediate memory table into the final table.

Note: The source table "Pivot" column must not be nullable (there are no such things as a "null" column) The creation will be refused even is this nullable column does not contain null values.

If a different result is desired, Create Table options are available to change the defaults used by Pivot. For instance, if we want to display the average expense for each person and product, spread in columns for each week, use the following statement:

```
create table pivex2
engine=connect table_type=pivot tabname=expenses
option_list='PivotCol=Week,Function=AVG';
```

Now saying:

```
select * from pivex2;
```

Will display the resulting table:

Who	What	3	4	5
Beth	Beer	16.00	15.00	20.00
Beth	Food	0.00	17.00	12.00
Janet	Beer	18.00	0.00	16.50
Janet	Car	19.00	17.00	12.00
Janet	Food	18.00	0.00	12.00
Joe	Beer	18.00	16.33	14.00
Joe	Car	20.00	0.00	0.00
Joe	Food	15.50	17.00	12.00

Restricting the columns in a Pivot Table

Let us suppose that we want a Pivot table from *expenses* summing the expenses for all people and products whatever week it was bought. We can do this just by removing from the *pivex* table the *week* column from the column list.

```
alter table pivex drop column week;
```

Alternatively, this can be done when doing the create table for the table. This is obvious if the columns are specified but when they are not, skipping the unwanted columns must be specified with the SKIPCOL option. For instance:

```
create table pivex
engine=connect table_type=pivot tabname=expenses
option_list='SkipCol=week';
```

The result we get from the new table is:

WHO	Beer	Car	Food

[33] Finding "Facts" or "Pivot" columns works only for table based pivot tables. They do not for view or srcdef based pivot tables, for which they must be explicitly specified.

Beth	51.00	0.00	29.00
Janet	51.00	48.00	30.00
Joe	81.00	20.00	77.00

Note: Restricting columns is also needed when the source tables contains extra columns that should not be part of the pivot table. This is true in particular for key columns that prevent a proper grouping.

PIVOT Create Table Syntax

The Create Table statement for PIVOT tables uses the following syntax:

```
CREATE TABLE pivot_table_name
[(column_definition)]
ENGINE=CONNECT TABLE_TYPE=PIVOT
{TABNAME='source_table_name' | SRCDEF='source_table_def'}
[OPTION_LIST='pivot_table_option_list'];
```

The column definition has two sets of columns:

1. A set of columns belonging to the source table, not including the “facts” and “pivot” columns.
2. “Data” columns receiving the values of the aggregated “facts” columns named from the values of the “pivot” column. They are indicated by the “flag” option.

The **options** and **sub-options** available for Pivot tables are:

Option	Type	Description
Tabname	*[DB.]Name*	The name of the table to “pivot”. If not set **SrcDef** must be specified.
SrcDef	*SQL_statement*	The statement used to generate the intermediate mysql table.
DBname	*name*	The name of the database containing the source table. Defaults to the current database.
Function*	*name*	The name of the aggregate function used for the data columns, SUM by default.
PivotCol*	*name*	Specifies the name of the Pivot column whose values are used to fill the “data” columns having the flag option.
FncCol*	[*func*(]*name*[)]	Specifies the name of the data “Facts” column. If the form *func(name)* is used, the aggregate function name is set to *func*.
SkipCol*	*name*[;*name*…]	Specifies the name of the source table unwanted columns to skip separated by semi-colons.
Groupby*	*Boolean*	Set it to True (1 or Yes) if the table already has a GROUP BY format.
Accept*	*Boolean*	To accept non-matching Pivot column values.

*: These options must be specified in the OPTION_LIST.

Additional access options

There are four cases where PIVOT must call the server containing the source table or on which the SrcDef statement must be executed:

1. The source table is not a CONNECT table.
2. The SrcDef option is specified.
3. The source table is on another server.
4. The columns are not specified.

By default, PIVOT tries to call the currently used server using host=localhost, user=root not using password, and port=3306. However, this may not be what is needed, in particular if the local root user has a password in which case you can get an “access denied” error message when creating or using the pivot table.

Specify the HOST, USER, PASSWORD and/or PORT options in the OPTION_LIST to override the default connection options used to access the source table, get column specifications, execute the generated GROUP BY or SrcDef query.

Defining a Pivot table

There are principally two ways to define a PIVOT table:

1. From an existing table or view.
2. Directly giving the SQL statement returning the result to pivot.

Defining a Pivot Table from a source Table

The **tabname** standard table option is used to give the name of the source table or view.

For tables or views, the internal Group By will be internally generated, except when the GROUPBY option is specified as true. Do it only when the table or view already has a valid GROUP BY format.

Directly defining the Source of a Pivot Table in SQL

Alternatively, the internal source can be directly defined using the **SrcDef** option that must have the proper group by format.

As we have seen above, a proper Pivot Table is made from an internal intermediate table resulting from the execution of a GROUP BY statement. In many cases, it is simpler or desirable to directly specify this when creating the pivot table. This may be because the source is the result of a complex process including filtering and/or joining tables.

To do this, use the **SrcDef** option, often replacing all other options. For instance, suppose that in the first example we are only interested in weeks 4 and 5. We could of course display it by:

```
select * from pivex where week in (4,5);
```

However, what if this table is a huge table? In this case, the correct way to do it is to define the pivot table as this:

```
create table pivex4
engine=connect table_type=pivot
option_list='PivotCol=what,FncCol=amount'
SrcDef='select who, week, what, sum(amount) from expenses
where week in (4,5) group by who, week, what';
```

If your source table has millions of records and you plan to pivot only a small subset of it, doing so will make a lot of a difference performance wise. In addition, you have entire liberty to use expressions, scalar functions, aliases, join, where and having clauses in your SQL statement. The only constraint is that you are responsible for the result of this statement to have the correct format for the pivot processing.

Using SrcDef also permits to use expressions and/or scalar functions. For instance:

```
create table xpivot (
Who char(10) not null,
What char(12) not null,
First double(8,2) flag=1,
Middle double(8,2) flag=1,
Last double(8,2) flag=1)
engine=connect table_type=PIVOT
option_list='PivotCol=wk,FncCol=amnt'
```

```
Srcdef='select who, what, case when week=3 then ''First'' when
week=5 then ''Last'' else ''Middle'' end as wk, sum(amount) *
6.56 as amnt from expenses group by who, what, wk';
```

Now the statement:

```
select * from xpivot;
```

Will display the result:

Who	What	First	Middle	Last
Beth	Beer	104.96	98.40	131.20
Beth	Food	0.00	111.52	78.72
Janet	Beer	118.08	0.00	216.48
Janet	Car	124.64	111.52	78.72
Janet	Food	118.08	0.00	78.72
Joe	Beer	118.08	321.44	91.84
Joe	Car	131.20	0.00	0.00
Joe	Food	203.36	223.04	78.72

Note 1: to avoid multiple lines having the same fixed column values, it is mandatory in **SrcDef** to place the pivot column at the end of the group by list.

Note 2: in the create statement **SrcDef,** it is mandatory to give aliases to the columns containing expressions so they are recognized by the other options.

Note 3: in the **SrcDef** select statement, quotes must be escaped because the entire statement is passed to MariaDB between quotes. Alternatively, specify it between double quotes.

Note 4: We could have left CONNECT do the column definitions. However, because they are defined from the sorted names, the Middle column would have been placed at the end of them.

Specifying the columns corresponding to the Pivot column
These columns must be named from the values existing in the "pivot" column. For instance, supposing we have the following *pet* table:

name	race	number
John	dog	2
Bill	cat	1
Mary	dog	1
Mary	cat	1
Lisbeth	rabbit	2
Kevin	cat	2
Kevin	bird	6
Donald	dog	1
Donald	fish	3

Pivoting it using *race* as the pivot column is done with:

```
create table pivet
engine=connect table_type=pivot tabname=pet
option_list='PivotCol=race,groupby=1';
```

This gives the result:

name	dog	cat	rabbit	bird	fish
John	2	0	0	0	0
Bill	0	1	0	0	0
Mary	1	1	0	0	0
Lisbeth	0	0	2	0	0
Kevin	0	2	0	6	0
Donald	1	0	0	0	3

By the way, does this ring a bell? It shows that in a way PIVOT tables are doing the opposite of what OCCUR tables do.

We can alternatively define specifically the table columns but what happens if the Pivot column contains values that is not matching a "data" column? There are three cases depending on the specified options and flags.

First case: If no specific options are specified, this is an error and, when trying to display the table, the query will abort with an error message stating that a non-matching value was met. Note that because the column list is established when creating the table, this is prone to occur if some rows containing new values for the pivot column are inserted in the source table. If this happens, you should re-create the table or manually add the new columns to the pivot table.

Second case: The accept option was specified. For instance:

```
create table xpivet2 (
name varchar(12) not null,
dog int not null default 0 flag=1,
cat int not null default 0 flag=1)
engine=connect table_type=pivot tabname=pet
option_list='PivotCol=race,groupby=1,Accept=1';
```

No error will be raised and the non-matching values will be ignored. This table will be displayed as:

name	dog	cat
John	2	0
Bill	0	1
Mary	1	1
Lisbeth	0	0
Kevin	0	2
Donald	1	0

Third case: A "dump" column was specified with the flag value equal to 2. All non-matching values will be added in this column. For instance:

```
create table xpivet (
name varchar(12) not null,
dog int not null default 0 flag=1,
cat int not null default 0 flag=1,
other int not null default 0 flag=2)
engine=connect table_type=pivot tabname=pet
option_list='PivotCol=race,groupby=1';
```

This table will be displayed as:

name	dog	cat	other
John	2	0	0
Bill	0	1	0
Mary	1	1	0
Lisbeth	0	0	2
Kevin	0	2	6
Donald	1	0	3

Note: It is a good idea to provide such a "dump" column if the source table is prone to be inserted new rows that can have a value for the pivot column that did not exist when the pivot table was created.

Pivoting big source tables
This may sometimes be risky. If the pivot column contains too many distinct values, the resulting table may have too many columns. In all cases the process involved, finding distinct values when creating the table or doing the group by when using it, can be very long and sometimes can fail because of exhausted memory.

Restrictions by a where clause should be applied to the source table when creating the pivot table rather than to the pivot table itself. This can be done by creating an intermediate table or using as source a view or a srcdef option.

Note: All PIVOT tables are read only.

TBL Table Type: Table List

This type allows defining a table as a list of tables of any engine and type. This is more flexible than multiple tables that must be all of the same file type. This type does, but is more flexible than, what is done with the MERGE engine.

The list of the columns of the TBL table may not necessarily include all the columns of the tables of the list. If the name of some columns is different in the sub-tables, the column to use can be specified by its position given by the FLAG option of the column. If the ACCEPT option is set to true (Y or 1) columns that do not exist in some of the sub-tables are accepted and their value will be null or pseudo-null[34] for the tables not having this column. The column types can also be different and an automatic conversion will be done if necessary.

Note: If not specified, the column definitions are retrieved from the first table of the table list.

The default database of the sub-tables is the current database or if not, can be specified in the DBNAME option. For the tables that are not in the default database, this can be specified in the table list. For instance, to create a table based on the French table *employe* in the current database and on the English table *employee* of the *db2* database, the syntax of the create statement can be:

```
CREATE TABLE allemp (
SERIALNO char(5) NOT NULL flag=1,
NAME varchar(12) NOT NULL flag=2,
SEX smallint(1),
TITLE varchar(15) NOT NULL flag=3,
MANAGER char(5) DEFAULT NULL flag=4,
DEPARTMENT char(4) NOT NULL flag=5,
SECRETARY char(5) DEFAULT NULL flag=6,
SALARY double(8,2) NOT NULL flag=7)
ENGINE=CONNECT table_type=TBL
table_list='employe,db2.employee' option_list='Accept=1';
```

[34] This depends on the nullability of the column.

The search for columns in sub tables is done by name and, if they exist with a different name, by their position given by a not null FLAG option. Column *sex* exists only in the English table (FLAG is 0). Its values will null value for the French table.

For instance, the query:

```
select name, sex, title, salary from allemp where department = 318;
```

Can reply:

NAME	SEX	TITLE	SALARY
BARBOUD	NULL	VENDEUR	9700.00
MARCHANT	NULL	VENDEUR	8800.00
MINIARD	NULL	ADMINISTRATIF	7500.00
POUPIN	NULL	INGENIEUR	7450.00
ANTERPE	NULL	INGENIEUR	6850.00
LOULOUTE	NULL	SECRETAIRE	4900.00
TARTINE	NULL	OPERATRICE	2800.00
WERTHER	NULL	DIRECTEUR	14500.00
VOITURIN	NULL	VENDEUR	10130.00
BANCROFT	2	SALESMAN	9600.00
MERCHANT	1	SALESMAN	8700.00
SHRINKY	2	ADMINISTRATOR	7500.00
WALTER	1	ENGINEER	7400.00
TONGHO	1	ENGINEER	6800.00
HONEY	2	SECRETARY	4900.00
PLUMHEAD	2	TYPIST	2800.00
WERTHER	1	DIRECTOR	14500.00
WHEELFOR	1	SALESMAN	10030.00

The first 9 rows, coming from the French table, have a null for the *sex* value. They would have 0 if the sex column had been created NOT NULL.

Sub-tables of not CONNECT engines
Sub-tables are accessed as PROXY tables. For not CONNECT sub-tables that are accessed via the MySQL API, it is possible like with PROXY to change the MYSQL default options. Of course, this will apply to all not CONNECT tables of the list.

Using the TABID special column
The TABID special column can be used to see from which table the rows come from and to restrict the access to only some of sub-tables.

Let us see the following example where t1 and t2 are MyISAM tables similar to the ones given in the MERGE description:

```
create table xt1 (
a int(11) not null,
message char(20))
engine=CONNECT table_type=MYSQL tabname='t1'
option_list='database=test,user=root';

create table xt2 (
```

```
a int(11) not null,
message char(20))
engine=CONNECT table_type=MYSQL tabname='t2'
option_list='database=test,user=root';

create table total (
tabname char(8) not null special='TABID',
a int(11) not null,
message char(20))
engine=CONNECT table_type=TBL table_list='xt1,xt2';

select * from total;
```

The result returned by the SELECT statement is:

tabname	a	message
xt1	1	Testing
xt1	2	table
xt1	3	t1
xt2	1	Testing
xt2	2	table
xt2	3	t2

Now if you send the query:

```
select * from total where tabname = 'xt2';
```

CONNECT will analyze the where clause and only read the *xt2* table. This can save time if you want to retrieve only a few sub-tables from a TBL table containing many sub-tables.

Parallel Execution

This is currently unavailable until some bugs are fixed.

~~When the sub-tables are located on different servers[35], it is possible to execute the remote queries simultaneously instead of sequentially. To enable this, set the THREAD option to yes.~~

~~Additional options available for this table type:~~

~~Option~~	~~Description~~
~~Maxerr~~	~~The max number of missing tables in the table list before an error is raised. Defaults to 0.~~
~~Accept~~	~~If true, missing columns are accepted and return null values. Defaults to false.~~
~~Thread~~	~~If true, enables parallel execution of remote sub-tables.~~

~~These options can be specified in the OPTION_LIST.~~

Note 1: All TBL tables are read only.

Note 2: An alternative to using the TBL type is to create a partition table specifying each partition to be a sub-table. This will be described later in this document.

[35] There is a bug that is not fixed yet. ~~Until it is, all remote tables must be executed on different servers. Do not do it if more than one table are executed on the same remote server.~~

Using the TBL and MYSQL types together

Used together, these types raise all the limitations of the FEDERATED and MERGE engines.

MERGE: Its limitation is obvious, the merged tables must be identical MyISAM tables, and MyISAM is even not the default engine for MariaDB. However, TBL accesses a collection of CONNECT tables, but because these tables can be user specified or internally created MYSQL tables, there is no limitation to the type of the tables that can be merged.

TBL is also much more flexible. The merged tables must not be "identical", they just should have the columns defined in the TBL table. If the type of one column in a merged table is not the one of the corresponding column of the TBL table, the column value will be converted. As we have seen, if one column of the TBL table of the TBL column does not exist in one of the merged table, the corresponding value will be set to null. If columns in a sub-table have a different name, they can be accessed by position using the FLAG column option of CONNECT.

However, one limitation of the TBL type regarding MERGE is that TBL tables are currently read-only; INSERT is not supported by TBL. Therefore, rather use MERGE to access a list of identical MyISAM tables because it will be faster, not passing by the MySQL API.

FEDERATED(X): The main limitation of FEDERATED is to access only MySQL/MariaDB tables. The MYSQL table type of CONNECT has the same limitation but CONNECT provides the ODBC and JDBC table types that can access tables of any RDBS providing an ODBC or JDBC driver (including MySQL even it is not really useful!)

Another major limitation of FEDERATED is to access only one table. By combining TBL and MYSQL tables, CONNECT enables to access a collection of local or remote tables as one table. Of course, the sub-tables can be on different servers. With one SELECT statement, a company manager will be able to interrogate results coming from all his subsidiary computers. This is great for distribution, banking, and many other industries.

Remotely executing complex queries

Many companies or administrations must deal with distributed information. CONNECT enables to deal with it efficiently without having to copy it to a centralized database. Let us suppose we have on some remote network machines *m1, m2, ... mn* some information contained in two tables *t1* and *t2*.

Suppose we want to execute on all servers a query such as:

```
select c1, sum(c2) from t1 a, t2 b where a.id = b.id group by c1;
```

This raises many problems. Returning the column values of the *t1* and *t2* tables from all servers can be a lot of network traffic. The GROUP BY on the possibly huge resulting tables can be a long process. In addition, the join on the *t1* and *t2* tables may be relevant only if the joined tuples belong to the same machine, obliging to add a condition on an additional TABID or SERVID special column.

All this can be avoided and optimized by forcing the query to be locally executed on each server and retrieving only the small results of the GROUP BY queries. Here is how to do it. For each remote machine, create a table that will retrieve the locally executed query. For instance, for *m1*:

```
create table rt1 engine=connect option_list='host=m1'
srcdef='select c1, sum(c2) as sc2 from t1 a, t2 b where a.id = b.id
     group by c1';
```

Note the alias for the functional column. An alias would be required for the c1 column if its name was different on some machines. The t1 and t2 table names can also be eventually different on the remote

machines. The true names must be used in the SRCDEF parameter. This will create a set of tables with two columns named *c1* and *sc2*[36].

Then create the table that will retrieve the result of all these tables:

```
create table rtall engine=connect table_type=tbl
table_list='rt1,rt2,…,rtn' option_list='thread=yes';
```

Now you can retrieve the desired result by:

```
select c1, sum(sc2) from rtall;
```

Almost all the work will be done on the remote machines, simultaneously thanks to the thread option[37], making this query super-fast even on big tables scattered on many remote machines.

Providing a list of servers

An interesting case is when the query to run on remote machines is the same for all of them. It is then possible to avoid declaring all sub-tables. In this case, the table list option will be used to specify the list of servers the SRCDEF query must be sent. This will be a list of URL's and/or Federated server names.

For instance, supposing that federated servers srv1, srv2, … srv*n* were created for all remote servers, it will be possible to create a TBL table allowing getting the result of a query executed on all of them by:

```
create table qall [column definition]
engine=connect table_type=TBL srcdef='a query'
table_list='srv1,srv2,…,srvn' [option_list='tread=yes'];
```

For instance:

```
create table verall engine=connect table_type=TBL
srcdef='select @@version' table_list=',server_one';
select * from verall;
```

This reply:

@@version
10.0.3-MariaDB-debug
10.0.2-MariaDB

Here the server list specifies a void server corresponding to the local running MariaDB and a federated server named *server_one*.

Special "Virtual" Tables

The special table types supported by CONNECT are the Virtual table type (VIR), Directory Listing table type (DIR), the Windows Management Instrumentation Table Type (WMI), and the "Mac Address" type (MAC).

These tables are "virtual tables", meaning they have no physical data but rather produce result data using specific algorithms. Note that this is close to what Views are, so they could be regarded as special views.

---

[36] To generate the columns from the SRCDEF query, CONNECT must execute it. This will make sure it is ok. However, if the remote server is not connected yet, or the remote table not existing yet, you can alternatively specify the columns in the CREATE TABLE statement.

[37] Thread is currently experimental. Use it only for test and report any malfunction on JIRA.

Virtual table type “VIR”

A VIR table is a virtual table having only Special or Virtual columns. Its only property is its “size”, or cardinality, meaning the number of virtual rows it contains. It is created using the syntax:

```
CREATE TABLE name [coldef] ENGINE=CONNECT TABLE_TYPE=VIR
     [BLOCK_SIZE=n];
```

The optional BLOCK_SIZE option gives the size of the table, defaulting to 1 if not specified. When its columns are not specified, it is almost equivalent to a SEQUENCE table “seq_1_to_*Size*”.

Displaying constants or expressions

Many DBMS use a no-column one-line table to do this, often call “dual”. MySQL and MariaDB use syntax where no table is specified. With CONNECT, you can achieve the same purpose with a virtual table, with the noticeable advantage of being able to display several lines. For example:

```
create table virt engine=connect table_type=VIR block_size=10;
select concat('The square root of  ', n, ' is') what,
     round(sqrt(n),16) value from virt;
```

This will return:

what	value
The square root of 1 is	1.0000000000000000
The square root of 2 is	1.4142135623730951
The square root of 3 is	1.7320508075688772
The square root of 4 is	2.0000000000000000
The square root of 5 is	2.2360679774997898
The square root of 6 is	2.4494897427831779
The square root of 7 is	2.6457513110645907
The square root of 8 is	2.8284271247461903
The square root of 9 is	3.0000000000000000
The square root of 10 is	3.1622776601683795

What happened here? First, unlike Oracle “dual” table that have no columns, a MariaDB table must have at least one column. By default, CONNECT creates VIR tables with one special column. This can be seen with the SHOW CREATE TABLE statement:

```
CREATE TABLE `virt` (
  `n` int(11) NOT NULL `SPECIAL`=ROWID,
  PRIMARY KEY (`n`)
) ENGINE=CONNECT DEFAULT CHARSET=latin1 `TABLE_TYPE`='VIR'
`BLOCK_SIZE`=10
```

This special column is called “n” and its value is the row number starting from 1. It is purely a virtual table and no data file exists corresponding to it and to its index.

It is possible to specify the columns of a VIR table but they must be CONNECT special columns or virtual columns. For instance:

```
create table virt2 (
 n int key not null special=rowid,
 sig1 bigint as ((n*(n+1))/2) virtual,
 sig2 bigint as(((2*n+1)*(n+1)*n)/6) virtual)
engine=connect table_type=VIR block_size=10000000;
select * from virt2 limit 995, 5;
```

This table shows the sum and the sum of the square of the *n* first integers:

n	sig1	sig2
996	496506	329845486
997	497503	330839495
998	498501	331835499
999	499500	332833500
1000	500500	333833500

Note that the size of the table can be made very big as there no physical data. However, the result should be limited in the queries. For instance:

```
select * from virt2 where n = 1664510;
```

Such a query could last very long if the *rowid* column were not indexed. Note that by default, CONNECT declares the "n" column as a primary key. Actually, VIR tables can be indexed but only on the ROWID (or ROWNUM) columns of the table. This is a virtual index for which no data is stored.

Generating a Table filled with constant values
An interesting use of virtual tables, which often cannot be achieved with a table of any other type, is to generate a table containing constant values.

This is easily done with a virtual table. Let us define the table FILLER as:

```
create table filler engine=connect table_type=VIR block_size=5000000;
```

Here we choose a size larger than the biggest table we want to generate. Later if we need a table pre-filled with default and/or null values, we can do for example:

```
create table tp (
id int(6) key not null,
name char(16) not null,
salary float(8,2));
insert into tp select n, 'unknown', NULL from filler where n
<= 10000;
```

This will generate a table having 10000 rows that can be updated later when needed. Note that a SEQUENCE table could have been used here instead of FILLING.

VIR tables vs. SEQUENCE tables
With just its default column, a VIR table is almost equivalent to a SEQUENCE table. The syntax used is the main difference, for instance:

```
select * from seq_100_to_150_step_10;
```

can be obtained with a VIR table (of size >= 15) by:

```
select n*10 from vir where n between 10 and 15;
```

Therefore, the main difference is to be able to define the columns of VIR tables. Unfortunately, there are currently many limitations to virtual columns that hopefully should be removed in the future.

DIR Type

A table of type DIR returns a list of file name and description as a result set. To create a DIR table, use a Create Table statement such as:

```
create table source (
DRIVE char(2),
PATH varchar(256),
FNAME varchar(256),
FTYPE char(4),
SIZE double(12,0) flag=5,
MODIFIED datetime)
engine=CONNECT table_type=DIR file_name='..\\*.cc';
```

When used in a query, the table returns the same file information listing than the system "DIR *.cc" statement would return if executed in the same current directory (here supposedly ..\)

For instance, the query:

```
select fname, size, modified from source
where fname like '%handler%';
```

Displays:

fname	size	modified
handler	152177	2011-06-13 18:08:29
sql_handler	25321	2011-06-13 18:08:31

Note: the important item in this table is the flag option value (set sequentially from 0 by default on Windows and from 1 on linux) because it determines which information item is returned in the column:

Flag value	Information
0	The disk drive (Windows)
1	The file path
2	The file name
3	The file type
4	The file attribute
5	The file size
6	The last write access date
7	The last read access date
8	The file creation date

The Subdir option

When specified in the create table statement, the Boolean *subdir* option indicates to list, in addition to the files contained in the specified directory, all the files verifying the filename pattern that are contained in sub-directories of the specified directory. For instance, using:

```
create table data (
PATH varchar(256) flag=1,
FNAME varchar(256),
FTYPE char(4),
SIZE double(12,0) flag=5)
engine=CONNECT table_type=DIR file_name='*.frm'
option_list='subdir=1';
select path, count(*), sum(size) from data group by path;
```

You will get the following result set showing how many tables are created in the MariaDB databases and what is the total length of the FRM files:

path	count(*)	sum(size)
\CommonSource\mariadb-5.2.7\sql\data\connect\	30	264469
\CommonSource\mariadb-5.2.7\sql\data\mysql\	23	207168
\CommonSource\mariadb-5.2.7\sql\data\test\	22	196882

The Nodir option (Windows)
The Boolean *Nodir* option can be set to false (0 or no) to add directories that match the file name pattern from the listed files (it is true by default). This is an addition to CONNECT version 1.6. Previously, directory names matching pattern where listed on Windows. Directories were and are never listed on Linux.

Note: The way file names are retrieved make positional access to them impossible. Therefore, DIR tables cannot be indexed nor sorted when it is done using positions.

Be aware, in particular when using the *subdir* option, that queries on DIR tables are slow and can last almost forever if made on a directory that contains a great number of files in it and its sub-directories.

DIR tables can be used to populate a list of files used to create a multiple=2 table. However, this is not as useful as it was when the multiple 3 did not exist.

Windows Management Instrumentation Table Type "WMI"

Note: This table type is available on Windows only.

WMI provides an operating system interface through which instrumented components provide information. Some Microsoft tools to retrieve information through WMI are the WMIC console command and the WMI CMI Studio application.

The CONNECT WMI table type enables administrators and operators not capable of scripting or programming on top of WMI to enjoy the benefit of WMI without even learning about it. It permits to present this information as tables that can be queried, transformed, copied in documents or other tables.

To create a WMI table displaying information coming from a WMI provider, you must provide the namespace and the class name that characterize the information you want to retrieve. The best way to find them is to use the WMI CIM Studio that have tools to browse namespaces and classes and that can display the names of the properties of that class.

The column names of the tables must be the names (case insensitive) of the properties you want to retrieve. For instance:

```
create table alias (
friendlyname char(32) not null,
target char(50) not null)
engine=CONNECT table_type='WMI'
     option_list='Namespace=root\\cli,Class=Msft_CliAlias';
```

WMI tables return one row for each instance of the related information. The above example is handy to get the class equivalent of the alias of the WMIC command and to have a list of many classes commonly used.

Because most of the useful classes belong to the 'root\cimv2' namespace, this is the default value for WMI tables when the namespace is not specified. Some classes have many properties whose name and type may not be known when creating the table. To find them, you can use the WMI CMI Studio application but his will be rarely required because CONNECT can retrieve them.

Actually, the class specification also has default values for some namespaces. For the 'root\cli' namespace the class name defaults to 'Msft_CliAlias' and for the 'root_cimv2' namespace the class default value is 'Win32_ComputerSystemProduct'. Because many class names begin with 'Win32_' it is not necessary to say it and specifying the class as 'Product' will effectively use class 'Win32_Product'.

For example if you define a table as:

```
create table CSPROD engine=CONNECT table_type='WMI';
```

It will return the information on the current machine, using the class ComputerSystemProduct of the CIMV2 namespace. For instance:

```
select * from csprod;
```

Will return a result such as:

Column	Row 1
Caption	Computer system product
Description	Computer system product
IdentifyingNumber	LXAP50X32982327A922300
Name	Aspire 8920
SKUNumber	
UUID	00FC523D-B8F7-DC12-A70E-00B0D1A46136
Vendor	Acer
Version	Aspire 8920

Note: This is a transposed display that can be obtained with some GUI.

Getting column information
An issue, when creating a WMI table, is to make its column definition. Indeed, even when you know the namespace and the class for the wanted information, it is not easy to find what are the names and types of its properties. However, because CONNECT can retrieve this information from the WMI provider, you can simply omit defining columns and CONNECT will do the job.

Alternatively, you can get this information using a catalog table (see below).

Performance Consideration
Some WMI providers can be very slow to answer. This is not an issue for those that return few object instances, such as the ones returning computer, motherboard, or Bios information. They generally return only one row (instance). However, some can return many rows, in particular the "CIM_DataFile" class. This is why care must be taken about them.

Firstly, it is possible to limit the allocated result size by using the 'Estimate' create table option. To avoid result truncation, CONNECT allocates a result of 100 rows that is enough for almost all tables. The 'Estimate' option permits to reduce this size for all classes that return only a few rows, and in some rare case to increase it to avoid truncation.

However, it is not possible to limit the time taken by some WMI providers to answer, in particular the CIM_DATAFILE class. Indeed, the Microsoft documentation says about it:

"Avoid enumerating or querying for all instances of **CIM_DataFile** on a computer because the volume of data is likely to either affect performance or cause the computer to stop responding."

Indeed, even a simple query such as:

```
select count(*) from cim where drive = 'D:' and path like
     '\\MariaDB\\%';
```

is prone to last almost forever (probably due to the LIKE clause). Therefore, when not asking for some specific items, you should consider using the DIR table type instead.

Syntax of WMI queries
Queries to WMI providers are done using the WQL language, not the SQL language. CONNECT does the job of making the WQL query. However, because of the restriction of the WQL syntax, the WHERE clause will be generated only when respecting the following restrictions:

1. No function.
2. No comparison between two columns.
3. No expression (currently a CONNECT restriction)
4. No BETWEEN and IN predicates.

Filtering with WHERE clauses not respecting these conditions will still be done by MariaDB only, except in the case of CIM_Datafile class for the reason given above.

However, there is one point that is not covered yet, the syntax used to specify dates in queries. WQL does not recognize dates as number items but translates them to its internal format dates specified as text. Many formats are recognized as described in the Microsoft documentation but only one is useful because common to WQL and MariaDB SQL. Here is an example of a query on a table named "cim" created by:

```
create table cim (
Name varchar(255) not null,
LastModified datetime not null)
engine=CONNECT table_type='WMI'
     option_list='class=CIM_DataFile,estimate=5000';
```

The date must be specified with the format in which CIM DATETIME values are stored (WMI uses the date and time formats defined by the Distributed Management Task Force)

```
select * from cim where drive = 'D:' and path = '\\PlugDB\\Bin\\'
     and lastmodified > '20120415000000.000000+120';
```

This syntax must be strictly respected. The text has the format:

```
yyyymmddHHMMSS.mmmmmmsUUU
```

It is: year, month, day, hour, minute, second, millisecond, and signed minute deviation from UTC. This format is locale-independent so you can write a query that runs on any machine.

Note 1: The WMI table type is available only in Windows versions of CONNECT.

Note 2: WMI tables are read only.

Note 3: WMI tables are not indexable.

Note 4: WMI consider all strings as case insensitive.

MAC Address Table Type "MAC"

Note: This table type is available on Windows only.

This type is used to display general information about the computer and, in particular, about its network cards. To create such a table, the syntax to use is:

```
create table tabname (column definition)
engine=CONNECT table_type=MAC;
```

Column names can be freely chosen because their signification, i.e. the values they will display, comes from the specified Flag option. The valid values for Flag are:

Flag	Valeur	Type
1	Host name	varchar(132)
2	Domain	varchar(132)
3	DNS address	varchar(24)
4	Node type	int(1)
5	Scope ID	varchar(256)
6	Routing	int(1)
7	Proxy	int(1)
8	DNS	int(1)
10	Name	varchar(260)
11	Description	varchar(132)
12	MAC address	char(24)
13	Type	int(3)
14	DHCP	int(1)
15	IP address	char(16)
16	SUBNET mask	char(16)
17	GATEWAY	char(16)
18	DHCP server	char(16)
19	Have WINS	int(1)
20	Primary WINS	char(16)
21	Secondary WINS	char(16)
22	Lease obtained	datetime
23	Lease expires	datetime

Note: The information of columns having a Flag value less than 10 are unique for the computer, the other ones are specific to the network cards of the computer.

For instance, you can define the table *macaddr* as:

```
create table macaddr (
Host varchar(132) flag=1,
Card varchar(132) flag=11,
Address char(24) flag=12,
IP char(16) flag=15,
Gateway char(16) flag=17,
Lease datetime flag=23)
engine=CONNECT table_type=MAC;
```

If you execute the query:

```
select host, address, ip, gateway, lease from macaddr;
```

It will return, for example:

Host	Address	IP	Gateway	Lease
OLIVIER	00-A0-D1-A4-61-36	0.0.0.0	0.0.0.0	1970-01-01 00:00:00
OLIVIER	00-1D-E0-9B-90-0B	192.168.0.10	192.168.0.254	2011-09-18 10:28:58

OEM Type: Implemented in an External LIB

Although CONNECT provides a rich set of table types, specific applications may need to access data organized in a way that is not handled by its existing foreign data wrappers (FDW). To handle these cases, CONNECT features an interface that enables developers to implement in C++ the required table wrapper and use it as if it were part of the standard CONNECT table type list. CONNECT can use these additional wrappers providing the corresponding external module (dll or shared lib) be available.

To create such a table on an external wrapper, use a Create Table statement as shown below.

```
create table xtab [(column definitions)]
engine=CONNECT table_type=OEM module='libname'
subtype='MYTYPE' [standard table options]
Option_list='Myopt=foo';
```

The option **module** gives the name of the DLL or shared library implementing the OEM wrapper for the table type. This library must be located in the plugin directory like all other plugins or UDF's.

This library must export a function GetMYTYPE. The option **subtype** enables CONNECT to have the name of the exported function and to use the new table type. Other options are interpreted by the OEM type and can also be specified within the **option_list** option.

Column definitions can be unspecified only if the external wrapper can return this information. For this it must export a function ColMYTYPE returning these definitions in a format acceptable by the CONNECT discovery function.

Which and how options must be specified and the way columns must be defined may vary depending on the OEM type used and should be documented by the OEM type implementer(s).

An OEM table example

The OEM table MONGO whose source is shown on appendix B permits to use MONGO like tables with MariaDB binary distributions containing but not enabling the MONGO table type.

Of course, the mongo (dll or so) exporting the GetMONGO and ColMONGO functions must be available in the plugin directory for all this to work.

Some currently developed OEM table modules and subtypes:

Module	Subtype	Description
libhello	HELLO	A sample OEM wrapper displaying a one line table saying "Hello world"
mongo	MONGO	Enables using tables based on MongoDB collections.
Tabfic	FIC	Handles files having the Windev HyperFile format.
Tabofx	OFC	Handles Open Financial Connectivity files.
Tabofx	QIF	Handles Quicken Interchange Format files.
Cirpack	CRPK	Handles CDR's from Cirpack UTP's.
Tabplg	PLG	Access tables from the PlugDB DBMS (supports Discovery).

How to implement an OEM wrapper is out of the scope of this document.

Catalog Tables

A catalog table is one that returns information about another table, or data source. It is similar to what MySQL commands such as DESCRIBE or SHOW do. Applied to local tables, this just duplicates what these commands do, with the noticeable difference that they are tables and can be used inside queries as joined tables or inside sub-selects.

But their main interest is to enable querying the structure of external tables that cannot be directly queried with description commands. Let's see an example:

Suppose we want to access the tables from a Microsoft Access database as an ODBC type table. The first information we must obtain is the list of tables existing in this data source. To get it, we will create a catalog table that will return it extracted from the result set of the SQLTables ODBC function:

```
create table tabinfo (
table_name varchar(128) not null,
table_type varchar(16) not null)
engine=connect table_type=ODBC catfunc=tables
Connection='DSN=MS Access Database;DBQ=C:/Program
Files/Microsoft Office/Office/1033/FPNWIND.MDB;';
```

The SQLTables function returns a result set having the following columns:

Field	Data Type	Null	Info Type	Flag Value
Table_Cat	char(128)	NO	FLD_CAT	17
Table_Schema	char(128)	NO	FLD_SCHEM	18
Table_Name	char(128)	NO	FLD_NAME	1
Table_Type	char(16)	NO	FLD_TYPE	2
Remark	char(255)	NO	FLD_REM	5

Note: The Info Type and Flag Value are CONNECT interpretations of this result.

Here we could have omitted the column definitions of the catalog table or, as in the above example, chose the columns returning the name and type of the tables. If specified, the columns must have the exact name of the corresponding SQLTables result set, or be given a different name with the matching flag value specification.

(The Table_Type can be TABLE, SYSTEM TABLE, VIEW, etc.)

For instance, to get the tables we want to use we can ask:

```
select table_name from tabinfo where table_type = 'TABLE';
```

This will return:

table_name
Categories
Customers
Employees
Products
Shippers
Suppliers

Now we want to create the table to access the CUSTOMERS table. Because CONNECT can retrieve the column description of ODBC tables, it not necessary to specify them in the create table statement:

```
create table Customers engine=connect table_type=ODBC
Connection='DSN=MS Access Database;DBQ=C:/Program
Files/Microsoft Office/Office/1033/FPNWIND.MDB;';
```

However, if we prefer to specify them (to eventually modify them) we must know what the column definitions of that table are. We can get this information with a catalog table. This is how to do it:

```
create table custinfo engine=connect table_type=ODBC
tabname=customers catfunc=columns
Connection='DSN=MS Access Database;DBQ=C:/Program
Files/Microsoft Office/Office/1033/FPNWIND.MDB;';
```

Alternatively, it is possible to specify what column of the catalog table we want:

```
create table custinfo (
column_name char(128) not null,
type_name char(20) not null,
length int(10) not null flag=7,
prec smallint(6) not null flag=9)
nullable smallint(6) not null)
engine=connect table_type=ODBC tabname=customers
catfunc=columns
Connection='DSN=MS Access Database;DBQ=C:/Program
Files/Microsoft Office/Office/1033/FPNWIND.MDB;';
```

To get the column info:

```
select * from custinfo;
```

which results in this table:

column_name	type_name	length	prec	nullable
CustomerID	VARCHAR	5	0	1
CompanyName	VARCHAR	40	0	1
ContactName	VARCHAR	30	0	1
ContactTitle	VARCHAR	30	0	1
Address	VARCHAR	60	0	1
City	VARCHAR	15	0	1
Region	VARCHAR	15	0	1
PostalCode	VARCHAR	10	0	1
Country	VARCHAR	15	0	1
Phone	VARCHAR	24	0	1
Fax	VARCHAR	24	0	1

Now you can create the CUSTOMERS table as:

```
create table Customers (
CustomerID varchar(5),
CompanyName varchar(40),
ContactName varchar(30),
ContactTitle varchar(30),
Address varchar(60),
City varchar(15),
Region varchar(15),
```

```
PostalCode varchar(10),
Country varchar(15),
Phone varchar(24),
Fax varchar(24))
engine=connect table_type=ODBC block_size=10
Connection='DSN=MS Access Database;DBQ=C:/Program
Files/Microsoft Office/Office/1033/FPNWIND.MDB;';
```

Let us explain what we did here: First of all, the creation of the catalog table. This table returns the result set of an ODBC SQLColumns function sent to the ODBC data source. Columns functions always return a data set having some of the following columns, depending on the table type:

Field	Data Type	Null	Info Type	Flag Value	Returned by
Table_Cat*	char(128)	NO	FLD_CAT	17	ODBC, JDBC
Table_Schema*	char(128)	NO	FLD_SCHEM	18	ODBC, JDBC
Table_Name	char(128)	NO	FLD_TABNAME	19	ODBC, JDBC
Column_Name	char(128)	NO	FLD_NAME	1	ALL
Data_Type	smallint(6)	NO	FLD_TYPE	2	ALL
Type_Name	char(30)	NO	FLD_TYPENAME	3	ALL
Column_Size*	int(10)	NO	FLD_PREC	4	ALL
Buffer_Length*	int(10)	NO	FLD_LENGTH	5	ALL
Decimal_Digits*	smallint(6)	NO	FLD_SCALE	6	ALL
Radix	smallint(6)	NO	FLD_RADIX	7	ODBC, JDBC, MYSQL
Nullable	smallint(6)	NO	FLD_NULL	8	ODBC, JDBC, MYSQL
Remarks	char(255)	NO	FLD_REM	9	ODBC, JDBC, MYSQL
Collation	char(32)	NO	FLD_CHARSET	10	MYSQL
Key	char(4)	NO	FLD_KEY	11	MYSQL
Default_value	N.A.		FLD_DEFAULT	12	
Privilege	N.A.		FLD_PRIV	13	
Date_fmt	char(32)	NO	FLD_DATEFMT	15	MYSQL
Xpath/Jpath	Varchar(256)	NO	FLD_FORMAT	16	XML/JSON

*: These names have changed since earlier versions of CONNECT. For tables created earlier, if you get an error message saying something such as "Invalid flag 0 for column *precision*", alter or re-create the table changing the name or adding the proper flag (precision: 4, length: 5, scale: 6)

Note: ALL includes ODBC, JDBC, MYSQL, DBF, CSV, PROXY, TBL, XML, JSON, XCOL, and WMI table types. More could be added later.

We chose among these columns the ones that were useful for our create statement, using the flag value when we gave them a different name (case insensitive).

The options used in this definition are the same as the one used later for the actual CUSTOMERS data tables except that:

1. The TABNAME option is mandatory here to specify what the queried table name is.
2. The CATFUNC option was added both to indicate that this is a catalog table, and to specify that we want column information.

Note: If the TABNAME option had not been specified, this table would have returned the columns of all the tables defined in the connected data source.

Currently the available CATFUNC are:

Function	Specified as:	Applies to table types:
FNC_TAB	**tab**les	ODBC, JDBC, MYSQL
FNC_COL	**col**umns	ODBC, JDBC, MYSQL, DBF, CSV, XML, JSON, PROXY, XCOL, TBL, WMI, ZIP
FNC_DSN	**datasource**s **dsn** **sqldatasource**s	ODBC
FNC_DRIVER	**driver**s **sqldriver**s	ODBC, JDBC

Note: Only the bold part of the function name specification is required.

The DATASOURCE and DRIVERS functions respectively return the list of available data sources and ODBC drivers available on the system.

The SQLDataSources function returns a result set having the following columns:

Field	Data Type	Null	Info Type	Flag value
Name	varchar(256)	NO	FLD_NAME	1
Description	varchar(256)	NO	FLD_REM	9

To get the data source, you can do for instance:

```
create table datasources
engine=CONNECT table_type=ODBC catfunc=DSN;
```

The SQLDrivers function returns a result set having the following columns:

Field	Type	Null	Info Type	Flag value
Description	varchar(128)	YES	FLD_NAME	1
Attributes	varchar(256)	YES	FLD_REM	9

You can get the driver list with:

```
create table drivers
engine=CONNECT table_type=ODBC catfunc=drivers;
```

Another example, WMI table
To create a catalog table returning the attribute names of a WMI class, use the same table options as the ones used with the normal WMI table plus the additional option 'catfunc=columns'. If specified, the columns of such a catalog table can be chosen among the following:

Name	Type	Flag	Description
Column_Name	CHAR	1	The name of the property
Data_Type	INT	2	The SQL data type
Type_Name	CHAR	3	The SQL type name
Column_Size	INT	4	The field length in character
Buffer_Length	INT	5	Depends on the coding
Scale	INT	6	Depends on the type

If you wish to use a different name for a column, set the Flag column option.

For example, before creating the “csprod” table, you could have created the info table:

```
create table CSPRODCOL (
Column_name char(64) not null,
Data_Type int(3) not null,
Type_name char(16) not null,
Length int(6) not null flag=5,
Prec int(2) not null flag=6)
engine=CONNECT table_type='WMI' catfunc=col;
```

Now the query:

```
select * from csprodcol;
```

will display the result:

Column_name	Data_Type	Type_name	Length	Prec
Caption	1	CHAR	255	1
Description	1	CHAR	255	1
IdentifyingNumber	1	CHAR	255	1
Name	1	CHAR	255	1
SKUNumber	1	CHAR	255	1
UUID	1	CHAR	255	1
Vendor	1	CHAR	255	1
Version	1	CHAR	255	1

This can help to define the columns of the matching normal table.

Note 1: The column length, for the Info table as well as for the normal table, can be chosen arbitrarily, it just must be enough to contain the returned information.

Note 2: The Scale column returns 1 for text columns (meaning case insensitive); 2 for float and double columns; and 0 for other numeric columns.

Catalog Table result size limit

Because catalog tables are processed like the information retrieved by “Discovery” when table columns are not specified in a Create Table statement, their result set is entirely retrieved and memory allocated.

By default, this allocation is done for a maximum return line number of:

Catfunc	Max lines
Drivers	256
Data Sources	512
Columns	20,000
Tables	10,000

When the number of lines retrieved for a table is more than this maximum, a warning is issued by CONNECT. This is mainly prone to occur with columns (and tables) with some data sources having many tables when the table name is not specified.

If this happens, it is possible to increase the default limit using the MAXRES option, for instance:

```
create table allcols engine=connect table_type=odbc
connection='DSN=ORACLE_TEST;UID=system;PWD=manager'
option_list='Maxres=110000' catfunc=columns;
```

Indeed, because the entire table result is memorized before the query is executed; the returned value would be limited even on a query such as:

```
select count(*) from allcols;
```

Virtual and Special Columns

CONNECT supports MariaDB virtual and persistent columns. It is also possible to declare a column as being a CONNECT special column. Let us see on an example how this can be done.

The *boys* table we have seen previously can be recreated as:

```
create table boys (
linenum int(6) not null default 0 special=rowid,
name char(12) not null,
city char(12) not null,
birth date not null date_format='DD/MM/YYYY',
hired date not null date_format='DD/MM/YYYY' flag=36,
agehired int(3) as (timestampdiff(year,hired,birth)) virtual,
fn char(100) not null default '' special=FILEID)
engine=CONNECT table_type=FIX file_name='boys.txt' lrecl=48;
```

We have defined two CONNECT special columns. You can give them any name; it is the field SPECIAL option that specifies the special column functional name.

Note: the default values specified for the special columns are ignored by CONNECT. They are specified just to prevent getting warning messages when inserting new rows.

For the definition of the *agehired* virtual column, no CONNECT options can be specified as it has no offset or length, not being stored in the file.

The command:

```
select * from boys where city = 'boston';
```

Will display the following result set as:

linenum	name	city	birth	hired	agehired	fn
1	John	Boston	1986-01-25	2010-06-02	24	d:\mariadb\sql\data\boys.txt
2	Henry	Boston	1987-06-07	2008-04-01	20	d:\mariadb\sql\data\boys.txt
6	Bill	Boston	1986-09-11	2008-02-10	21	d:\mariadb\sql\data\boys.txt

Existing special columns are listed in the following table:

Special Name	Type	Description of the column value
ROWID	Integer	The row ordinal number in the table or partition. This not quite equivalent to a virtual column with an auto increment of 1 because rows are renumbered when deleting rows.
ROWNUM	Integer	The row ordinal number in the file or group of rows. This is different from ROWID for multiple tables, TBL/XCOL/OCCUR/PIVOT tables, XML tables with a multiple column, and for DBF tables where ROWNUM includes soft deleted rows.
FILEID **FDISK** **FPATH** **FNAME** **FTYPE**	String	FILEID returns the full name of the file this row belongs to. Useful in particular for multiple tables represented by several files. The other special columns can be used to retrieve only one part of the full name.
TABID	String	The name of the table this row belongs to. Useful for TBL tables.

Special Name	Type	Description of the column value
PARTID	String	The name of the partition this row belongs to. Specific to partitioned tables.
SERVID	String	The name of the federated server or server host used by a MYSQL table. “ODBC” for an ODBC table, “JDBC” for a JDBC table and “Current” for all other table types.

Note: CONNECT does not currently support auto incremented columns. However, ROWID special columns provide in some cases functionalities like auto incremented columns.

Note: CONNECT cannot support dynamic columns because they need Blob support.

Indexing

Indexing is one of the main ways to optimize queries. Key columns, in particular when they are used to join tables, should be indexed. But what should be done for columns that have only few distinct values? If they are randomly placed in the table, they should not be indexed because reading many rows in random order can be slower than reading the entire table sequentially. However, if the values are sorted or clustered, block indexing can be a good alternative because CONNECT use it while still reading the table sequentially.

CONNECT provides five indexing types:

1. Standard Indexing
2. Block Indexing
3. Remote Indexing
4. Dynamic Indexing
5. Virtual Indexing

Standard Indexing

CONNECT standard indexes are created and used as the ones of other storage engines although they have a specific internal format. The CONNECT handler supports the use of standard indexes for most of the file based table types.

You can define them in the CREATE TABLE statement, or either using the CREATE INDEX statement or the ALTER TABLE statement. In all cases, the index files are automatically made. They can be dropped either using the DROP INDEX statement or the ALTER TABLE statement, and this erases the index files.

Indexes are automatically reconstructed when the table is created, modified by INSERT, UPDATE or DELETE commands, or when the SEPINDEX option is changed. If you have a lot of changes to do on a table at one moment, you can use table locking to prevent indexes to be reconstructed after each statement. The indexes will be reconstructed when unlocking the table. For instance:

```
lock table t1 write;
insert into t1 values(...);
insert into t1 values(...);
...
unlock tables;
```

If a table was modified by an external application that does not handle indexing, the indexes must be reconstructed to prevent returning false or incomplete results. To do this, use the OPTIMIZE TABLE command.

For outward tables, index files are not erased when dropping the table. This is the same as for the data file and preserves the possibility of several users using the same data file via different tables.

Unlike other handlers, CONNECT constructs the indexes as files that are named by default from the data file name, not from the table name, and located in the data file directory. Depending on the SEPINDEX table option, indexes are saved in a unique file or in separate files (if SEPINDEX is true). For instance, if indexes are in separate files, the primary index of the table dept.dat of type DOS is a file named dept_PRIMARY.dnx. This makes possible to define several tables on the same data file, with eventual different options such as mapped or not mapped, and to share the index files as well.

If the index file should have a different name, for instance because several tables are created on the same data file with different indexes, specify the base index file name with the XFILE_NAME option.

Note 1: Indexed column must be declared NOT NULL, CONNECT not supporting indexes containing null values.

Note 2: MRR is used by standard indexing if it is enabled.

Note 3: Prefix indexing is not supported. If specified, CONNECT engine ignores the prefix and builds a whole index.

Handling index errors

The way CONNECT handles indexing is very specific. All table modifications are done regardless of indexing. Only after a table has been modified, or when an OPTIMIZE TABLE command is send, the indexes are made. If an error occurs, the corresponding index is not made. However, CONNECT being a non-transactional engine; it is unable to roll back the changes made to the table. The main causes of indexing errors are:

- Trying to index a nullable column. In this case, you can alter the table to declare the column as not nullable or, if the column is nullable indeed, make it not indexed.
- Entering duplicate values in a column indexed by a unique index. In this case, if the index was wrongly declared as unique, alter is declaration to reflect this. If the column should really contain unique values, you must manually remove or update the duplicate values.

In both cases, after correcting the error, remake the indexes with the OPTIMIZE TABLE command.

Index file mapping

To accelerate the indexing process, CONNECT makes an index structure in memory from the index file. This can be done by reading the index file or using it as if it was in memory by "file mapping". On enabled version, file mapping is used according to the Boolean *connect_indx_map* system variable. Set it to 0 (file read) or 1 (file mapping)

Block Indexing

To accelerate input/output, CONNECT uses when possible a read/write mode by blocks of *n* rows, *n* being the value given in the BLOCK_SIZE option of the Create Table, or a default value depending on the table type. This is automatic for fixed files (FIX, BIN, DBF or VEC), but must be specified for variable files (DOS, CSV or FMT).

For blocked tables, further optimization can be achieved if the data values for some columns are "clustered" meaning that they are not evenly scattered in the table but grouped in some consecutive rows. Block indexing permits to skip blocks in which no rows fulfill a conditional predicate without having even to read the block. This is true in particular for sorted columns.

You indicate this when creating the table by using the DISTRIB=*d* column option. The enum value *d* can be *scattered*, *clustered*, or *sorted*. In general, only one column can be sorted. Block indexing is used only for clustered and sorted columns.

Difference between standard indexing and block indexing

- Block indexing is internally handled by CONNECT while reading sequentially a table data. This means that when standard indexing is used on a table, block indexing is not used.
- In a query, only one standard index can be used. However, block indexing can combine the restrictions coming from a where clause implying several clustered/sorted columns.
- The block index files are faster to make and much smaller than standard index files.

Notes for this Release:

- On all operations that create or modify a table, CONNECT automatically calculates or recalculates and saves the mini/maxi or bitmap values for each block, enabling it to skip block containing no acceptable values. In the case where the optimize file does not correspond anymore to the table, because it has been accidentally destroyed, or because some column definitions have been altered, you can use the OPTIMIZE TABLE command to reconstruct the optimization file.
- Sorted column special processing is currently restricted to ascending sort. Column sorted in descending order must be flagged as clustered. Improper sorting is not checked in Update or Insert operations but is flagged when optimizing the table.
- Block indexing can be done in two ways. Keeping the min/max values existing for each block, or keeping a bitmap allowing knowing what column distinct values are met in each block. This second

ways often gives a better optimization, except for sorted columns for which both are equivalent. The bitmap approach can be done only on columns having not too many distinct values. This is estimated by the MAX_DIST option value associated to the column when creating the table. Bitmap block indexing will be used if this number is not greater than the MAXBMP setting (currently 32).

- CONNECT cannot perform block indexing on case insensitive character columns. To force block indexing on a character column, specify its charset as not case insensitive, for instance:

```
sitmat char(1) not null collate 'latin1_bin'
distrib=clustered max_dist=8,
```

Remote Indexing

Remote indexing is specific to the MYSQL table type. It is equivalent to what the FEDERATED storage does. A MYSQL table does not support indexes per se. Because access to the table is handled remotely, it is the remote table that supports the indexes. What the MYSQL table does is just to add to the SELECT command sent to the remote server a where clause allowing the remote server to use indexing when applicable.

Note however that because CONNECT adds when possible all or part of the where clause of the original query, this happens often even if the remote indexed column is not declared locally indexed. The only, but very important, case a column should be locally declared indexed is when it is used to join tables. Otherwise, the required where clause would not be added to the sent SELECT query.

Dynamic Indexing

An indexed created as "dynamic" is a standard index which, in some cases, can be reconstructed for a specific query. This happens for some queries where two tables are joined by an indexed key column. If the "*from"* table is big and the "*to"* big table reduced in size because of a where clause, it can be worthwhile to reconstruct the index on this reduced table.

Because of the time added by reconstructing the index, this will be valuable only if the time gained by reducing the index size if more than this reconstruction time. Therefore, this should not be done if the "*from*" table is small because there will not be enough row joining to compensate the additional time. Otherwise, the gain of using a dynamic index is:

- Indexing time is a little faster if the index is smaller.
- The join process will return only the rows fulfilling the where clause.
- Because the table is read sequentially when reconstructing the index there no need for MRR.
- Constructing the index can be faster if the table is reduced by block indexing.
- While constructing the index, CONNECT also stores in memory the values of other used columns.

This last point is particularly important. It means that after the index is reconstructed, the join is done on a temporary memory table.

Unfortunately, storage engines being called independently by MariaDB for each table, CONNECT has no global information to decide when it is good to use dynamic indexing. Therefore, you should use it only on cases where you see that some important join queries take a very long time and only on columns used for joining the table. How to declare an index to be dynamic is by using the Boolean DYNAM index option. For instance, the query:

```
select d.diag, count(*) cnt from diag d, patients p where d.pnb =
    p.pnb and ageyears < 17 and county = 30 and drg <> 11 and d.diag
    between 4296 and 9434 group by d.diag order by cnt desc;
```

Such a query joining the *diag* table to the *patients* table may last a very long time if the tables are big. To declare the primary key on the *pnb* column of the *patients* table to be dynamic:

```
alter table patients drop primary key;
alter table patients add primary key (pnb) comment 'DYNAMIC' dynam=1;
```

Note 1: The comment is not mandatory here but useful to see that the index is dynamic if you use the SHOW INDEX command.

Note 2: There is currently no way to just change the DYNAM option without dropping and adding the index. This is unfortunate because it takes time.

Virtual Indexing

It applies only to the virtual tables of type VIR and must be made on a column specifying SPECIAL=ROWID or SPECIAL=ROWNUM.

Partitioning and Sharding

CONNECT supports the MySQL/MariaDB partition specification. It is done is a way similar than MyISAM or InnoDB do by using the PARTITION engine that must be enabled for this to work. This type of partitioning is sometimes referred as "horizontal partitioning".

Partitioning enables you to distribute portions of individual tables across a file system according to rules which you can set largely as needed. In effect, different portions of a table are stored as separate tables in different locations. The user-selected rule by which the division of data is accomplished is known as a partitioning function, which in MariaDB can be the modulus, simple matching against a set of ranges or value lists, an internal hashing function, or a linear hashing function.

CONNECT takes this notion a step further, by providing two types of partitioning:

1. File partitioning. Each partition is stored in a separate file like in multiple tables.
2. Table partitioning. Each partition is stored in a separate table like in TBL tables.

Partition engine issues

Using partitions sometimes requires creating the tables in an unnatural way to avoid some error due to several partition engine bugs:

1. Engine specific column and index options are not recognized and cause a syntax error when the table is created. The workaround is to create the table in two steps, a CREATE TABLE statement followed by an ALTER TABLE statement.
2. The connection string, when specified for the table, is lost by the partition engine. The workaround is to specify the connection string in the OPTION_LIST.
3. MySQL upstream bug #71095. In case of list columns partitioning it sometimes causes a false "impossible where" clause to be raised. This makes a wrong void result returned when it should not be void. There is no workaround but this bug should be hopefully fixed.

The following examples are using the above workaround syntax to address these issues.

File Partitioning

File partitioning applies to file based CONNECT table types. Like for multiple tables, physical data is stored in several files instead of just one. The differences from multiple tables are:

1. Data is distributed amongst the different files following the partition rule.
2. Unlike multiple tables, partitioned tables are not read only.
3. Unlike multiple tables, partitioned tables can be indexable.
4. The file names are generated from the partition names.
5. Query pruning is automatically made by the partition engine.

The table file names are generated differently depending on whether the table is an inward or outward table. For inward tables, for which the file name is not specified, the partition file names are:

```
Data file name: table_name#P#partition_name.table_file_type
Index file name: table_name#P#partition_name.index_file_type
```

For instance, for the table:

```
CREATE TABLE t1 (
id INT KEY NOT NULL,
msg VARCHAR(32))
ENGINE=CONNECT TABLE_TYPE=FIX
partition by range(id) (
partition first values less than(10),
partition middle values less than(50),
partition last values less than(MAXVALUE));
```

CONNECT will generate in the current data directory the files:

```
t1#P#first.fix
t1#P#first.fnx
t1#P#middle.fix
t1#P#middle.fnx
t1#P#last.fix
t1#P#last.fnx
```

This is similar than what the partition engine does for other engines. As a matter of facts, CONNECT partitioned inward tables behave like other engines partition tables do. Just the data format is different.

Note: If sub-partitioning is used, inward table files and index files are named:

```
table_name#P#partition_name#SP#subpartition_name.type
table_name#P#partition_name#SP#subpartition_name.index_type
```

Outward Tables

The real problems occur with outward tables, in particular when they are created from already existing files. The first issue is to make the partition table use the correct existing file names. The second one, only for already existing not void tables, is to be sure the partitioning function match the distribution of the data already existing in the files.

The first issue is addressed by the way data file names are constructed. For instance, let us suppose we want to make a table from the fixed formatted files:

```
E:\Data\part1.txt
E:\Data\part2.txt
E:\Data\part3.txt
```

This can be done by creating a table such as:

```
create table t2 (
id int not null,
msg varchar(32),
index XID(id))
engine=connect table_type=FIX file_name='E:/Data/part%s.txt'
partition by range(id) (
partition `1` values less than(10),
partition `2` values less than(50),
partition `3` values less than(MAXVALUE));
```

The rule is that for each partition the matching file name is internally generated by replacing in the given FILE_NAME option value the "*%s*" part by the partition name.

If the table was initially void, further inserts will populate it according to the partition function. However, if the files did exist and contained data, this is your responsibility to determine what partition function matches the data distribution in them. This means in particular that partitioning by key or by hash cannot be used (except in exceptional cases) because you have almost no control to what the used algorithm does.

In the example above, there is no problem if the table is initially void, but if it is not, serious problems can be met if the initial distribution does not match the table distribution. Supposing a row in which "id" as the value 12 was initially contained in the part1.txt file, it will be seen when selecting the whole table but if you ask:

```
select * from t2 where id = 12;
```

The result will have 0 rows. This is because, according to the partition function query, pruning will only look inside the second partition and will miss the row that is in the wrong partition.

One way to check for wrong distribution if for instance to compare the results from queries such as:

```
SELECT partition_name, table_rows FROM
information_schema.partitions WHERE table_name = 't2';
```

And

```
SELECT CASE WHEN id < 10 THEN 1 WHEN id < 50 THEN 2 ELSE 3 END
AS pn, COUNT(*) FROM part3 GROUP BY pn;
```

If they match, the distribution can be correct although this does not prove it. However, if they do not match, the distribution is surely wrong.

Partitioning on a Special Column
There are some cases where the files of a multiple table do not contain columns that can be used for range or list partitioning. For instance, let's suppose we have a multiple table based on the following files:

```
tmp/boston.txt
tmp/chicago.txt
tmp/atlanta.txt
```

Each of them containing the same kind of data:

```
ID: int
First_name: varchar(16)
Last_name: varchar(30)
Birth: date
Hired: date
Job: char(10)
Salary: double(8,2)
```

A multiple table can be created on them, for instance by:

```
create table mulemp (
id int NOT NULL,
first_name varchar(16) NOT NULL,
last_name varchar(30) NOT NULL,
birth date NOT NULL date_format='DD/MM/YYYY',
hired date NOT NULL date_format='DD/MM/YYYY',
job char(10) NOT NULL,
salary double(8,2) NOT NULL
) engine=CONNECT table_type=FIX file_name='tmp/*.txt' multiple=1;
```

The issue is that if we want to create a partitioned table on these files, there are no columns to use for defining a partition function. Each city file can have the same kind of column values and there is no way to distinguish them.

However, there is a solution. It is to add to the table a special column that will be used by the partition function. For instance, the new table creation can be done by:

```
create table partemp (
id int NOT NULL,
first_name varchar(16) NOT NULL,
last_name varchar(30) NOT NULL,
birth date NOT NULL date_format='DD/MM/YYYY',
hired date NOT NULL date_format='DD/MM/YYYY',
```

```
job char(16) NOT NULL,
salary double(10,2) NOT NULL,
city char(12) default 'boston' special=PARTID,
index XID(id)
) engine=CONNECT table_type=FIX file_name='E:/Data/Test/%s.txt';
alter table partemp
partition by list columns(city) (
partition `atlanta` values in('atlanta'),
partition `boston` values in('boston'),
partition `chicago` values in('chicago'));
```

Note 1: we had to do it in two steps because of the column CONNECT options.

Note 2: the special column PARTID returns the name of the partition in which the row is located.

Note 3: here we could have used the FNAME special column instead because the file name is specified as being the partition name.

This may seem rather stupid because it means for instance that a row will be in partition *boston* if it belongs to the partition *boston*! However, it works because the partition engine doesn't know about special columns and behaves as if the *city* column was a real column.

What happens if we populate it by?

```
insert into partemp(id,first_name,last_name,birth,hired,job,salary) values
(1205,'Harry','Cover','1982-10-07','2010-09-21','MANAGEMENT',125000.00);
insert into partemp values
(1524,'Jim','Beams','1985-06-18','2012-07-25','SALES',52000.00,'chicago'),
(1431,'Johnny','Walker','1988-03-12','2012-08-09','RESEARCH',46521.87,'boston'),
(1864,'Jack','Daniels','1991-12-01','2013-02-16','DEVELOPMENT',63540.50,'atlanta');
```

The value given for the *city* column (explicitly or by default) will be used by the partition engine to decide in which partition to insert the rows. It will be ignored by CONNECT (a special column cannot be given a value) but later will return the matching value. For instance:

```
select city, first_name, job from partemp where id in (1524,1431);
```

This query returns:

city	first_name	job
boston	Johnny	RESEARCH
chicago	Jim	SALES

Everything works as if the *city* column was a real column contained in the table data files.

Partitioning of zipped tables
Two cases are currently supported:

If a table is based on several zipped files, portioning is done the standard way as above. This is the *file_name* option specifying the name of the zip files that shall contain the '%s' part used to generate the file names.

If a table is based on only one zip file containing several entries, this will be indicated by placing the '%s' part in the *entry* option value.

Note: If a table is based on several zipped files each containing several entries, only the first case is possible. Using sub-partitioning to make partitions on each entry is not supported yet.

Table Partitioning

With table partitioning, each partition is physically represented by a sub-table. Compared to standard partitioning, this brings the following features:

1. The partitions can be tables driven by different engines. This relieves the current existing limitation of the partition engine.
2. The partitions can be tables driven by engines not currently supporting partitioning.
3. Partition tables can be located on remote servers, enabling table sharding.
4. Like for TBL tables, the columns of the partition table do not necessarily match the columns of the sub-tables.

The way it is done is to create the partition table with a table type referring to other tables, PROXY, MYSQL, ODBC or JDBC. Let us see how this is done on a simple example. Supposing we have created the following tables:

```
create table xt1 (
id int not null,
msg varchar(32))
engine=myisam;

create table xt2 (
id int not null,
msg varchar(32));  /* engine=innoDB */

create table xt3 (
id int not null,
msg varchar(32))
engine=connect table_type=CSV;
```

We can for instance create a partition table using these tables as physical partitions by:

```
create table t3 (
id int not null,
msg varchar(32))
engine=connect table_type=PROXY tabname='xt%s'
partition by range columns(id) (
partition `1` values less than(10),
partition `2` values less than(50),
partition `3` values less than(MAXVALUE));
```

Here the name of each partition sub-table will be made by replacing the '%s' part of the tabname option value by the partition name. Now if we do:

```
insert into t3 values
(4, 'four'),(7,'seven'),(10,'ten'),(40,'forty'),
(60,'sixty'),(81,'eighty one'),(72,'seventy two'),
(11,'eleven'),(1,'one'),(35,'thirty five'),(8,'eight');
```

The rows will be distributed in the different sub-tables according to the partition function. This can be seen by executing the query:

```
select partition_name, table_rows from
information_schema.partitions where table_name = 't3';
```

This query replies:

partition_name	table_rows
1	4
2	4
3	3

Query pruning is of course automatic, for instance:

```
explain partitions select * from t3 where id = 81;
```

This query replies:

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	part5	3	ALL	<null>	<null>	<null>	<null>	22	Using where

When executing this select query, only sub-table xt3 will be used.

Indexing with Table Partitioning

Using the PROXY table type seems natural. However, in this current version, the issue is that PROXY (and ODBC) tables are not indexable. Therefore, if you want the table to be indexed, you must use the MYSQL table type. The CREATE TABLE statement will be almost the same:

```
create table t4 (
id int key not null,
msg varchar(32))
engine=connect table_type=MYSQL tabname='xt%s'
partition by range columns(id) (
partition `1` values less than(10),
partition `2` values less than(50),
partition `3` values less than(MAXVALUE));
```

The column *id* is declared as a key, and the table type is now MYSQL. This makes Sub-tables to be accessed by calling a MySQL server as MYSQL tables do. Note that this modifies only the way CONNECT sub-tables are accessed because other engine tables where silently accessed this way.

However, indexing just make the partitioned table to use “remote indexing” the way FEDERATED tables do. This means that when sending the query to retrieve the table data, a where clause will be added to the query. For instance, let’s suppose you ask:

```
select * from t4 where id = 7;
```

The query sent to the server will be:

```
SELECT `id`, `msg` FROM `xt1` WHERE `id` = 7
```

On a query like this one, it does not change much because the where clause could have been added anyway by the cond_push function, but it does make a difference in case of join. The main thing to understand is that real indexing is done by the called table and therefore that it should be indexed.

This also means that the xt1, xt2, and xt3 table indexes should be made separately because creating the t2 table as indexed does **not** make the indexes on the sub-tables.

Sharding with Table Partitioning

Using table partitioning can have one more advantage. Because the sub-tables can address a table located on another server, it is possible to shard a table on separate servers and hardware machines. This may be required to access as one table data already located on several remote machines, such as servers of a company branches. Or it can be just used to split a huge table for performance reason.

For instance, supposing we have created the following tables:

```
create table rt1 (id int key not null, msg varchar(32))
engine=federated connection='mysql://root@host1/test/sales';

create table rt2 (id int key not null, msg varchar(32))
engine=federated connection='mysql://root@host2/test/sales';

create table rt3 (id int key not null, msg varchar(32))
engine=federated connection='mysql://root@host3/test/sales';
```

Creating the partition table accessing all these will be almost like what we did with the *t4* table:

```
create table t5 (
id int key not null,
msg varchar(32))
engine=connect table_type=MYSQL tabname='rt%s'
partition by range columns(id) (
partition `1` values less than(10),
partition `2` values less than(50),
partition `3` values less than(MAXVALUE));
```

The only difference is the tabname option now referring to the *rt1*, *rt2*, and *rt3* tables. However, even if it works, this is not the best way to do it. This is because accessing a table via the MySQL API is done twice per table. Once by CONNECT to access the FEDERATED table on the local server, then a second time by FEDERATED engine to access the remote table.

The CONNECT MYSQL table type being used anyway, you'd rather use it to directly access the remote tables. Indeed, the partition names can also be used to modify the connection URL's. For instance, in the case shown above, the partition table can be created as:

```
create table t6 (
id int key not null,
msg varchar(32))
engine=connect table_type=MYSQL
option_list='connect=mysql://root@host%s/test/sales'
partition by range columns(id) (
partition `1` values less than(10),
partition `2` values less than(50),
partition `3` values less than(MAXVALUE));
```

Several things can be noted here:

1. As we have seen before, the partition engine currently loses the connection string. This is why it was specified as "connect" in the option list.
2. For each partition sub-tables, the "%s" part of the connection string has been replaced by the partition name.
3. It is not needed anymore to define the rt1, rt2, and rt3 tables (even it does not harm) and the FEDERATED engine is no more used to access the remote tables.

This is a simple case where the connection string is almost the same for all the sub-tables. But what if the sub-tables are accessed by very different connection strings? For instance:

```
For rt1: connection='mysql://root:tinono@127.0.0.1:3307/test/xt1'
For rt2: connection='mysql://foo:foopass@denver/dbemp/xt2'
For rt3: connection='mysql://root@huston :5505/test/tabx'
```

There are two solutions. The first one is to use the parts of the connection string to differentiate as partition names:

```
create table t7 (
id int key not null,
msg varchar(32))
engine=connect table_type=MYSQL
option_list='connect=mysql://%s'
partition by range columns(id) (
partition `root:tinono@127.0.0.1:3307/test/xt1` values less than(10),
partition `foo:foopass@denver/dbemp/xt2` values less than(50),
partition `root@huston :5505/test/tabx` values less than(MAXVALUE));
```

The second one, allowing avoiding too long partition names, is to create federated servers to access the remote tables (if they do not already exist, else just use them). For instance, the first one could be:

```
create server `server_one` foreign data wrapper 'mysql'
options
  (host '127.0.0.1',
  database 'test',
  user 'root',
  password 'tinono',
  port 3307);
```

Similarly, "server_two" and "server_three" would be created and the final partition table would be created as:

```
create table t8 (
id int key not null,
msg varchar(32))
engine=connect table_type=MYSQL
option_list='connect=server_%s'
partition by range columns(id) (
partition `one/xt1` values less than(10),
partition `two/xt2` values less than(50),
partition `three/tabx` values less than(MAXVALUE));
```

It would be even simpler if all remote tables had the same name on the remote databases, for instance if they all were named *xt1*, the connection string could be set as "server_%s/xt1" and the partition names would be just "one", "two", and "three".

Sharding on a Special Column

The technique we have seen above with file partitioning is also available with table partitioning. Companies willing to use as one table data sharded on the company branch servers can, as we have seen, add to the table create definition a special column. For instance:

```
create table t9 (
id int not null,
msg varchar(32),
branch char(16) default 'main' special=PARTID,
index XID(id))
engine=connect table_type=MYSQL
option_list='connect=server_%s/sales'
partition by range columns(id) (
partition `main` values in('main'),
partition `east` values in('east'),
partition `west` values in('west'));
```

This example assumes that federated servers had been created named “server_main”, “server_east” and “server_west” and that all remote tables are named “sales”. Note also that in this example, the column *id* is no more a key.

Current Partition Limitations

Partition names are limited to 64 characters.

Because the partition engine was written before some other engines were added to MariaDB, the way it works is sometime incompatible with these engines, in particular with CONNECT.

Update statement
With the sample tables above, you can do update statements such as:

```
update t2 set msg = 'quatre' where id = 4;
```

It works perfectly and is accepted by CONNECT. However, let us consider the statement:

```
update t2 set id = 41 where msg = 'four';
```

This statement is not accepted by CONNECT. The reason is that the column *id* being part of the partition function, changing its value may require the modified row to be moved to another partition. The way it is done by the partition engine is to delete the old row and to re-insert the new modified one. However, this is done in a way that is not currently compatible with CONNECT (remember that CONNECT supports UPDATE in a specific way for the table type MYSQL)

This limitation could be temporary. Meanwhile the workaround is to manually do what is done above, Deleting the row to modify and inserting the modified row:

```
delete from t2 where id = 4;
insert into t2 values(41, 'four');
```

Alter Table statement
For all CONNECT outward tables, the ALTER TABLE statement does not make any change in the table data. This is why ALTER TABLE should not be used to modify the partition definition, except of course to correct a wrong definition. Note that using ALTER TABLE to create a partition table in two steps because column options would be lost is valid as it applies to a table that is not yet partitioned.

As we have seen, it is also safe to use it to create or drop indexes. Otherwise, a simple rule of thumb is to avoid altering a table definition and better drop and re-create a table whose definition must be modified. Just remember that for outward CONNECT tables, dropping a table does not erase the data and that creating it does not modify existing data.

Rowid special column
Each partition being handled separately as one table, the ROWID special column returns the rank of the row in its partition, not in the whole table. This means that for partition tables ROWID and ROWNUM are equivalent.

Using CONNECT

The main characteristic of CONNECT is to enable accessing data scattered on a machine as if it was a centralized database. This, and the fact that file locking is not used by connect (data files are open and closed for each query) makes CONNECT very useful for importing or exporting data into or from a MariaDB database and also for all types of Business Intelligence applications. However, it is not suited for transactional applications.

For instance, the index type used by CONNECT is closer to bitmap indexing than to B-trees. It is very fast for retrieving result but not when updating is done. In fact, even if only one indexed value is modified in a big table, the index is entirely remade (yet this being four to five times faster than for a b-tree index). But normally in Business Intelligence applications, files are not modified so often.

If you are using CONNECT to analyze files that can be modified by an external process, the indexes are of course not modified by it and become outdated. Use the OPTIMIZE TABLE command to update them before using the tables based on them.

This means also that CONNECT is not designed to be used by centralized servers, which are mostly used for transactions and often must run a long time without human intervening.

Performance

Performances vary a great deal depending on the table type. For instance, ODBC tables are only retrieved as fast as the other DBMS can do. If you have a lot of queries to execute, the best way to optimize your work can be sometime to translate the data from one type to another. Fortunately, this is very simple with CONNECT. Fixed format like FIX, BIN or VEC tables can be created from slower ones by commands such as:

```
Create table fastable table_specs select * from slowtable;
```

FIX and BIN are often the better choice because the I/O functions are done on blocks of BLOCK_SIZE rows. VEC tables can be very efficient for tables having many columns only a few being used in each query. Furthermore, for tables of reasonable size, the MAPPED option can very often speed up many queries.

Create Table statement

Be aware of the two broad kinds of CONNECT tables:

Inward They are table whose file name is not specified at create. An empty file will be given a default name (*tabname.tabtype*) and will be populated like for other engines. They do not require the FILE privilege and can be used for testing purpose.

Outward They are all other CONNECT tables and access external data sources or files. They are the true useful tables but require the FILE privilege.

Drop Table statement

For outward tables, the Drop Table statement just removes the table definition but does not erase the table data. However, dropping an inward table also erase the table data as well.

AlterTable statement

Be careful using the ALTER TABLE statement with outward tables. Currently the data compatibility is not tested and the modified definition can become incompatible with the data. In particular, Alter modifies the table definition only but does not modify the table data. Consequently, the table type should not be modified this way, except to correct an incorrect definition. Also, adding, dropping or modifying columns may be wrong because the default offset values (when not explicitly given by the FLAG option) may be wrong when recompiled with missing columns.

Safe use of ALTER is for indexing, as we have seen earlier, and to change options such as MAPPED or HUGE those do not impact the data format but just the way the data file is accessed. Modifying the

BLOCK_SIZE option is all right with FIX, BIN, DBF, split VEC tables; however it is unsafe for VEC tables that are not split (only one data file) because at their creation the estimate size has been made a multiple of the block size. This can cause errors if this estimate is not a multiple of the new value of the block size.

In all cases, it is safer to drop and re-create the table (outward tables) or to make another one from the table that must be modified. This is as fast as altering the table because table data is not modified.

Update and Delete for file tables

CONNECT can execute these commands following two different algorithms:

1. It can do it in place, directly modifying rows (update) or moving rows (delete) within the table file. This is a fast way to do it when indexing is used.
2. It can do it using a temporary file to make the changes. This is required when updating variable record length tables and is more secure in all cases.

The choice between these algorithms depends on the session variable **connect_use_tempfile**. This is an enum session variable that can be given the values:

Value	Description
NO	The first algorithm is always used. Because it can cause errors when updating variable record length tables, this value should be set only for testing.
AUTO	This is the default value. It leaves CONNECT choose the algorithm to use. Currently it is equivalent to NO except when updating variable record length tables (DOS, CSV or FMT) with file mapping forced to OFF.
YES	Using a temporary file is chosen with some exceptions that are: when file mapping is ON, for VEC tables and when deleting from DBF tables (soft delete) For variable record length tables, file mapping is forced to OFF.
FORCE	Like YES but forcing file mapping to be OFF for all table types.
TEST	Reserved for CONNECT development.

The default AUTO value favors the best response time. Using a temporary file is longer but leaves the table unchanged when the process is interrupted manually or by errors.

Importing file data into MariaDB tables

Directly using external (file) data has many advantages, such as to work on “fresh” data produced for instance by cash registers, telephone switches, or scientific apparatus. However, you may want in some case to import external data into your MariaDB database. This is extremely simple and flexible using the CONNECT handler. For instance, let us suppose you want to import the data of the *xsample.xml* XML file previously given in example into a MyISAM table called *biblio* belonging to the *connect* database. All you have to do is to create it by:

```
create table biblio engine=myisam select * from xsampall2;
```

This last statement creates the MyISAM table and inserts the original XML data, translated to tabular format by the *xsampall2* CONNECT table, into the MariaDB *biblio* table. Note that further transformation on the data could have been achieved by using a more elaborate Select statement in the Create statement, for instance using filters, alias or applying functions to the data. However, because the Create Table process copies table data, later modifications of the *xsample.xml* file will not change the *biblio* table and changes to the *biblio* table will not modify the *xsample.xml* file.

All these can be combined or transformed by further SQL operations. This makes working with CONNECT much more flexible (but not so fast) than just using the LOAD statement.

Exporting data from MariaDB

This is obviously possible with CONNECT, in particular for all formats not supported by the standard 'Into File' feature of the Select statement. Let us consider the query:

```
select plugin_name handler, plugin_version version, plugin_author
author, plugin_description description, plugin_maturity maturity
from information_schema.plugins where plugin_type = 'STORAGE ENGINE';
```

Supposing you want to get the result of this query into a file *handlers.htm* in XML/HTML format, allowing displaying it on an Internet browser, this is how you can do it:

Just create the CONNECT table that will be used to make the file:

```
create table handout
engine=CONNECT table_type=XML file_name='handout.htm' header=yes
option_list='name=TABLE,coltype=HTML,attribute=border=1;cellpadding=5
     ,headattr=bgcolor=yellow'
select plugin_name handler, plugin_version version, plugin_author
author, plugin_description description, plugin_maturity maturity
from information_schema.plugins where plugin_type = 'STORAGE ENGINE';
```

Here the column definition is not given and will come from the Select statement following the Create. The CONNECT options are the same we have seen previously. This will do both actions, creating the matching *handlers* CONNECT table and 'filling' it with the query result.

Note 1: This could not be done in only one statement if the table type had required using explicit CONNECT column options. In this case, firstly create the table, and then populate it with an Insert statement.

Note 2: The source "plugins" table column "description" is a long text column, data type not supported for CONNECT tables. It has been silently internally replaced by varchar(*n*), *n* being the value of the *connect_conv_size* global variable.

Condition Pushdown

The ODBC, JDBC, MYSQL, TBL and WMI table types use condition pushdown in order to restrict the number of rows returned by the RDBS source or the WMI component. Since MariaDB 5.5 the engine condition pushdown is OFF by default. It is therefore necessary to set it ON, for instance by:

```
set optimizer_switch='engine_condition_pushdown=on';
```

Or starting mysqld with this parameter set to ON, for instance:

```
mysqld --console --engine_condition_pushdown=on
```

Note 1: specifying –console is important to have some error messages from CONNECT printed because MariaDB does not always retrieve them.

Note 2: since MariaDB 10.0.4, the CONDITION_PUSHDOWN argument is no more accepted. However, it is no more needed because CONNECT uses condition pushdown unconditionally.

Current Status of the CONNECT Handler

The current CONNECT handler is a GA version. It was written starting both from an aborted project written for MySQL in 2004 and from the "DBCONNECT" program. It was tested on all the examples described in this document, and is distributed with a set of 53 test cases. Here is a not limited list of future developments:

1. Adding more table types.
2. Make more tests files (53 are already made)
3. ~~Supporting parallel execution of TBL sub-tables when executed on different servers~~.

4. Adding more data types, in particular unsigned ones (done for unsigned).
5. Supporting indexing on nullable and decimal columns.
6. Adding more optimize tools (block indexing, dynamic indexing, etc.) (done)
7. Supporting MRR (done)
8. Supporting partitioning (done)

No programs are bug free, especially new ones. Please report all bugs or documentation errors using the means provided by MariaDB.

Security

The use of the CONNECT engine requires the FILE privilege, except for “inward” tables. This should not be an important restriction. The use of CONNECT “outward” tables on a remote server seems of limited interest without knowing the files existing on it and must be protected anyway. On the other hand, using it on the local client machine is not an issue because it is always possible to create locally a user with the FILE privilege.

Appendix A

Some JSON sample files.

Expense.json


```
[
  {
    "WHO": "Joe",
    "WEEK": [
      {
        "NUMBER": 3,
        "EXPENSE": [
          {
            "WHAT": "Beer",
            "AMOUNT": 18.00
          },
          {
            "WHAT": "Food",
            "AMOUNT": 12.00
          },
          {
            "WHAT": "Food",
            "AMOUNT": 19.00
          },
          {
            "WHAT": "Car",
            "AMOUNT": 20.00
          }
        ]
      },
      {
        "NUMBER": 4,
        "EXPENSE": [
          {
            "WHAT": "Beer",
            "AMOUNT": 19.00
          },
          {
            "WHAT": "Beer",
            "AMOUNT": 16.00
          },
          {
            "WHAT": "Food",
            "AMOUNT": 17.00
          },
          {
            "WHAT": "Food",
            "AMOUNT": 17.00
          },
          {
            "WHAT": "Beer",
            "AMOUNT": 14.00
          }
        ]
      },
      {
```

```
      "NUMBER": 5,
      "EXPENSE": [
        {
          "WHAT": "Beer",
          "AMOUNT": 14.00
        },
        {
          "WHAT": "Food",
          "AMOUNT": 12.00
        }
      ]
    }
  ]
},
{
  "WHO": "Beth",
  "WEEK": [
    {
      "NUMBER": 3,
      "EXPENSE": [
        {
          "WHAT": "Beer",
          "AMOUNT": 16.00
        }
      ]
    },
    {
      "NUMBER": 4,
      "EXPENSE": [
        {
          "WHAT": "Food",
          "AMOUNT": 17.00
        },
        {
          "WHAT": "Beer",
          "AMOUNT": 15.00
        }
      ]
    },
    {
      "NUMBER": 5,
      "EXPENSE": [
        {
          "WHAT": "Food",
          "AMOUNT": 12.00
        },
        {
          "WHAT": "Beer",
          "AMOUNT": 20.00
        }
      ]
    }
  ]
},
{
  "WHO": "Janet",
```

```
    "WEEK": [
      {
        "NUMBER": 3,
        "EXPENSE": [
          {
            "WHAT": "Car",
            "AMOUNT": 19.00
          },
          {
            "WHAT": "Food",
            "AMOUNT": 18.00
          },
          {
            "WHAT": "Beer",
            "AMOUNT": 18.00
          }
        ]
      },
      {
        "NUMBER": 4,
        "EXPENSE": [
          {
            "WHAT": "Car",
            "AMOUNT": 17.00
          }
        ]
      },
      {
        "NUMBER": 5,
        "EXPENSE": [
          {
            "WHAT": "Beer",
            "AMOUNT": 14.00
          },
          {
            "WHAT": "Car",
            "AMOUNT": 12.00
          },
          {
            "WHAT": "Beer",
            "AMOUNT": 19.00
          },
          {
            "WHAT": "Food",
            "AMOUNT": 12.00
          }
        ]
      }
    ]
  }
]
```

Appendix B

This is an example showing how an OEM table can be implemented.

The header File my_global.h:

```
/**************************************************************************/
/*  Definitions needed by the included files.                             */
/**************************************************************************/
#if !defined(MY_GLOBAL_H)
#define MY_GLOBAL_H
typedef unsigned int uint;
typedef unsigned int uint32;
typedef unsigned short ushort;
typedef unsigned long ulong;
typedef unsigned long DWORD;
typedef char *LPSTR;
typedef const char *LPCSTR;
typedef int BOOL;
#if defined(__WIN__)
typedef void *HANDLE;
#else
typedef int HANDLE;
#endif
typedef char *PSZ;
typedef const char *PCSZ;
typedef unsigned char BYTE;
typedef unsigned char uchar;
typedef long long longlong;
typedef unsigned long long ulonglong;
typedef char my_bool;
struct charset_info_st {};
typedef const charset_info_st CHARSET_INFO;
#define FALSE 0
#define TRUE  1
#define Item char
#define MY_MAX(a,b) ((a>b)?(a):(b))
#define MY_MIN(a,b) ((a<b)?(a):(b))
#endif // MY_GLOBAL_H
```

Note: This a fake *my_global.h* that just contains what is useful for the *jmgoem.cpp* source file.

The source File jmgoem.cpp:

```
/************* jmgoem C++ Program Source Code File (.CPP) *************/
/* PROGRAM NAME: jmgoem    Version 1.0                                */
/*  (C) Copyright to the author Olivier BERTRAND          2017        */
/*  This program is the Java MONGO OEM module definition.             */
/**************************************************************************/

/**************************************************************************/
/*  Definitions needed by the included files.                             */
/**************************************************************************/
#include "my_global.h"

/**************************************************************************/
/*  Include application header files:                                     */
/*  global.h    is header containing all global declarations.         */
/*  plgdbsem.h  is header containing the DB application declarations.  */
/*  (x)table.h  is header containing the TDBASE declarations.         */
/*  tabext.h    is header containing the TDBEXT declarations.         */
/*  mongo.h     is header containing the MONGO declarations.          */
```

```
/***********************************************************************/
#include "global.h"
#include "plgdbsem.h"
#if defined(HAVE_JMGO)
#include "csort.h"
#include "javaconn.h"
#endif   // HAVE_JMGO
#include "xtable.h"
#include "tabext.h"
#include "mongo.h"

/***********************************************************************/
/*  These functions are exported from the MONGO library.               */
/***********************************************************************/
extern "C" {
  PTABDEF __stdcall GetMONGO(PGLOBAL, void*);
  PQRYRES __stdcall ColMONGO(PGLOBAL, PTOS, void*, char*, char*, bool);
} // extern "C"

/***********************************************************************/
/*  DB static variables.                                               */
/***********************************************************************/
int TDB::Tnum;
int DTVAL::Shift;
#if defined(HAVE_JMGO)
int    CSORT::Limit = 0;
double CSORT::Lg2 = log(2.0);
size_t CSORT::Cpn[1000] = {0};          /* Precalculated cmpnum values */
#if defined(HAVE_JAVACONN)
char *JvmPath = NULL;
char *ClassPath = NULL;
char *GetPluginDir(void)
{return "C:/mongo-java-driver/mongo-java-driver-3.4.2.jar;"
        "C:/MariaDB-10.1/MariaDB/storage/connect/";}
char *GetJavaWrapper(void) {return (char*)"wrappers/Mongo3Interface";}
#else   // !HAVE_JAVACONN
HANDLE JAVAConn::LibJvm;              // Handle to the jvm DLL
CRTJVM JAVAConn::CreateJavaVM;
GETJVM JAVAConn::GetCreatedJavaVMs;
#if defined(_DEBUG)
GETDEF JAVAConn::GetDefaultJavaVMInitArgs;
#endif  //  _DEBUG
#endif // !HAVE_JAVACONN
#endif   // HAVE_JMGO

/***********************************************************************/
/*  This function returns a Mongo definition class.                    */
/***********************************************************************/
PTABDEF __stdcall GetMONGO(PGLOBAL g, void *memp)
{
  return new(g, memp) MGODEF;
} // end of GetMONGO

#ifdef NOEXP
/***********************************************************************/
/* Functions to be defined if not exported by the CONNECT version.     */
/***********************************************************************/
bool IsNum(PSZ s)
{
  for (char *p = s; *p; p++)
    if (*p == ']')
      break;
```

```
    else if (!isdigit(*p) || *p == '-')
      return false;

  return true;
}      // end of IsNum

void *PlugSubAlloc(PGLOBAL g, void *memp, size_t size)
{
  PPOOLHEADER pph;                          /* Points on area header. */

  if (!memp)
    memp = g->Sarea;

  size = ((size + 7) / 8) * 8;        /* Round up size to multiple of 8 */
  pph = (PPOOLHEADER)memp;

  if ((uint)size > pph->FreeBlk) {    /* Not enough memory left in pool */
    PCSZ pname = "Work";

    sprintf(g->Message,
      "Not enough memory in %s area for request of %u (used=%d free=%d)",
                          pname, (uint)size, pph->To_Free, pph->FreeBlk);
    throw 1234;
  } /* endif size OS32 code */

  //  Do the suballocation the simplest way
  memp = MakePtr(memp, pph->To_Free); /* Points to suballocated block  */
  pph->To_Free += (OFFSET)size;       /* New offset of pool free block */
  pph->FreeBlk -= (uint)size;         /* New size   of pool free block */
  return (memp);
} /* end of PlugSubAlloc */
#endif

/***********************************************************************/
/*  Return the columns definition to MariaDB.                          */
/***********************************************************************/
PQRYRES __stdcall ColMONGO(PGLOBAL g, PTOS tp, char *tab,
                                               char *db, bool info)
{
#ifdef NOMGOCOL
  // Cannot use discovery
  strcpy(g->Message, "No discovery, MGOColumns is not accessible");
  return NULL;
#else
  return MGOColumns(g, db, NULL, tp, info);
#endif
} // end of ColMONGO
```

The file mongo.def: (required only on Windows)

```
LIBRARY      MONGO
EXPORTS
   GetMONGO      @1
   ColMONGO      @2
```

Compiling this OEM

To compile this OEM module, firstly make the two or three required files by a copy/past from the above listings.

Even if this module is to be used with a binary distribution, you need some source files in order to successfully compile it. At least the CONNECT header files that are included in *jmgoem.cpp* and the

ones they can include. This can be obtained by downloading the MariaDB source file tar.gz and extracting from it the CONNECT sources files in a directory that will be added to the additional source directories if it is not the directory containing the above files.

The module must be linked to the *ha_connect.lib* of the binary version it will used with. Recent distributions add this lib in the plugin directory.

The resulting module, for instance mongo.so or mongo.dll, must be placed in the plugin directory of the MariaDB server. Then, you will be able to use MONGO like tables simply replacing in the CREATE TABLE statement the option TABLE_TYPE=MONGO by TABLE_TYPE=OEM SUBTYPE=MONGO MODULE='mongo.(so|dll)'. Actually, the module name, here supposedly 'mongo', can be anything you like.

This will work with the last (not yet) distributed versions of MariaDB 10.0 and 10.1 because, even it is not enabled, the MONGO type is included in them. This is also the case of MariaDB 10.2.9 but then, on Windows, you will have to define NOEXP and NOMGOCOL because these functions are not exported by this version.

To implement it for older versions that do not contain the MONGO type, you can do it by adding the corresponding source files, namely *javaconn.cpp, jmgfam.cpp, jmgoconn.cpp, mongo.cpp* and *tabjmg.cpp* that you should find in the CONNECT extracted source files if you downloaded a recent version. As they include my_global.h, this is the reason why the included file was named this way. In addition, your compiling should define HAVE_JMGO and HAVE_JAVACONN. Of course, this is possible only if the *ha_connect.lib* is available.

Appendix C

Compiling Json UDFs in a Separate library

Although the JSON UDF's can be nicely included in the CONNECT library module, there are cases when you may need to have them in a separate library.

This is when CONNECT is compiled embedded, or if you want to test or use these UDF's with other MariaDB versions not including them.

To make it, you need to have access to the last MariaDB source code. Then, make a project containing these files:

1. jsonudf.cpp
2. json.cpp
3. value.cpp
4. osutil.c
5. plugutil.cpp
6. maputil.cpp
7. jsonutil.cpp

jsonutil.cpp is not distributed with the source code, you will have to make it from the following:

```
#include "my_global.h"
#include "mysqld.h"
#include "plugin.h"
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <errno.h>

#include "global.h"

extern "C" int GetTraceValue(void) { return 0; }
uint GetJsonGrpSize(void) { return 100; }

/*****************************************************************************/
/*  These replace missing function of the (not used) DTVAL class.      */
/*****************************************************************************/
typedef struct _datpar   *PDTP;
PDTP MakeDateFormat(PGLOBAL, PSZ, bool, bool, int) { return NULL; }
int ExtractDate(char*, PDTP, int, int val[6]) { return 0; }


#ifdef __WIN__
my_bool CloseFileHandle(HANDLE h)
{
       return !CloseHandle(h);
} /* end of CloseFileHandle */

#else  /* UNIX */
my_bool CloseFileHandle(HANDLE h)
{
       return (close(h)) ? TRUE : FALSE;
}  /* end of CloseFileHandle */

int GetLastError()
{
       return errno;
}  /* end of GetLastError */
```

```
#endif  // UNIX

/***********************************************************************/
/*  Program for sub-allocating one item in a storage area.             */
/*  Note: This function is equivalent to PlugSubAlloc except that in   */
/*  case of insufficient memory, it returns NULL instead of doing a    */
/*  long jump. The caller must test the return value for error.        */
/***********************************************************************/
void *PlgDBSubAlloc(PGLOBAL g, void *memp, size_t size)
{
  PPOOLHEADER pph;                          // Points on area header.

  if (!memp)         //  Allocation is to be done in the Sarea
    memp = g->Sarea;

  size = ((size + 7) / 8) * 8;  /* Round up size to multiple of 8 */
  pph = (PPOOLHEADER)memp;

  if ((uint)size > pph->FreeBlk) { /* Not enough memory left in pool */
    sprintf(g->Message,
     "Not enough memory in Work area for request of %d (used=%d free=%d)",
                    (int)size, pph->To_Free, pph->FreeBlk);
    return NULL;
  } // endif size

  // Do the suballocation the simplest way
  memp = MakePtr(memp, pph->To_Free);   // Points to sub_allocated block
  pph->To_Free += size;                 // New offset of pool free block
  pph->FreeBlk -= size;                 // New size   of pool free block

  return (memp);
} // end of PlgDBSubAlloc
```

You can create the file by copy/paste from the above.

Set all the additional include directories to the MariaDB include directories used in plugin compiling plus the reference of the storage/connect directories, and compile like any other UDF giving any name to the made library module (I used jsonudf.dll on Windows)

Then you can create the functions using this name as the *soname* parameter.

There are some restrictions when using the UDF's this way:

- The *connect_json_grp_size* variable cannot be accessed. The group size is set to 100.
- In case of error, warnings are replaced by messages sent to *stderr*.
- No trace.

Index