MonetDB/XQuery Reference Manual

Version 4.22

The MonetDB Development Team

Last updated: Feb 2, 2008

Table of Contents

This is the reference manual of MonetDB/XQuery, and open-source XQuery database system built on:

- the open-source MonetDB column-store, developed at CWI.
- the open-source Pathfinder XQuery to relational algebra compiler, developed at Technical University Munich.
- the open-source PF/Tijah XML information retrieval system, developed at Technical University Twente.

This manual attempts at collecting all relevant information about the functionality of the system. For a quick hands-on introduction to MonetDB/XQuery, we refer to the Tutorial.

1 General Introduction

The MonetDB/XQuery reference manual serves as the primary entry point to locate information on its functionality, system architecture, services and best practices on using its components.

The manual is produced from a Texinfo framework file, which collects and organizes bits-and-pieces of information scattered around the many source components comprising the MonetDB software family. The Texinfo file is turned into a HTML browse-able version using *makeinfo* program. The PDF version can be produced using *pdflatex*. Alternative formats, e.g., XML and DocBook format, can be readily obtained from the Texinfo file.

1.1 Intended Audience

The MonetDB reference manual is aimed at application developers and researchers with an intermediate level exposure to database technology, its embedding in host environments, such as C, Perl, Python, PHP, or middleware solutions based on JDBC and ODBC.

Feedback on the functionality provided is highly appreciated, especially when you embark on a complex programming project. If the envisioned missing functionality is generally applicable it makes sense to contribute it to the community. Share your comments and thoughts through the MonetDB mailing list held at SourceForge.

1.2 Features and Limitations

In this section we give a short overview of the key features to (not) consider the MonetDB product family. In a nutshell, its origin in the area of data-mining and data-warehousing makes it an ideal choice for high volume, complex query dominant applications. MonetDB was not designed for high-volume secure OLTP settings initially.

1.2.1 MonetDB Pro's

A high-performance database management system. MonetDB is an easy accessible open-source DBMS for SQL-[XQuery-]based applications and database research projects. Its origin goes back over a decade, when we decided that the database hotset - the part used by the applications - can be largely held in main-memory or where a few columns of a broad relational table are sufficient to handle a request. Further exploitation of cache-conscious algorithms proved the validity of these design decisions.

A multi-model system. MonetDB supports multiple query language front-ends. Aside from its proprietary language, called the MonetDB Assembly Language (MAL), it aims at ANSI SQL-2003 and W3C XQuery with update facilities. Their underlying logical data model and computational scheme differs widely. The system is designed to provide a common ground for both languages and it is prepared to support languages based on yet another data model or processing paradigm.

A column-store based database kernel. MonetDB is built on the canonical representation of database containers, called Binary Association Tables (BATs). MonetDB is known as

one of the principal COLUMN-STORES, as typically, a separate BAT is used for each table column. The datastructures are geared towards efficient representation when they mimic an n-ary relational scheme.

This led to an architecture where the traditional page-pool is replaced by one with a much larger granularity based on BATs. They are sizeable entities -up to hundreds of megabytes- swapped into memory upon need. The benefit of this approach has been shown in numerous papers in the scientific literature.

A broad spectrum database system. MonetDB is continuously developed to support a broad application field. Although originally developed for Analytical CRM products, it is now being used at the low-end scale as an embedded relational kernel and projects are underway to tackle the huge database problems encountered in scientific databases, e.g. astronomy.

An extendable database system. MonetDB has been strongly influenced by the scientific experiments to understand the interplay between algorithms and hardware features. It has turned MonetDB into an extensible database system. It proves valuable in those cases where an application specific and critical component makes all the difference between slow and fast implementation.

An open-source software system. MonetDB has been developed over many years of research at CWI, whose charter ensures that results are easily accessible to others. Either through publications in the scientific domain or publication of the software components involved. The MonetDB mailing list is the access point to a larger audience for advice. A subscription to the mailing list helps the developer team to justify their hours put into MonetDB's development and maintenance.

1.2.2 MonetDB Con's

There are several areas where MonetDB has not yet built a reputation. They are the prime candidates for experimentation, but also areas where application construction may become risky. Mature products or commercial support may then provide a short-term solution, while MonetDB programmers team works on filling the functional gaps. The following areas should be considered with care:

Persistent object caches. The tendency to develop applications in Java and C based on a persistent object model, is a no-go area for MonetDB. Much like other database engines, the overhead of individual record access does not do justice to the data structures and algorithms in the kernel. They are chosen to optimize bulk processing, which always comes at a price for individual object access.

Nevertheless, MonetDB has been used from its early days in a commercial application, where the programmers took care in maintaining the Java object-cache. It is a route with great benefits, but also one where sufficient manpower should be devoted to perform a good job.

High-performance financial OLTP. MonetDB was originally not designed for highly concurrent transaction workloads. It was decided to make ACID hooks explicit in the query plans generated by the front-end compilers. Given the abundance of main memory nowadays and the slack CPU cycles to process database requests, it may be profitable to consider serial execution of all OLTP transactions.

The SQL implementation provides full transaction control and recovery.

Security. MonetDB has not been designed with a strong focus on security. The major precautions have been taken, but are incomplete when access to the hosting machine is granted or when direct access is granted to the Monet Assembly Language features. The system is preferably deployed in a sand-boxed environment where remote access is encapsulated in a dedicated application framework.

Scaling over multiple machines. MonetDB does not provide a centralized controlled, distributed database infrastructure yet. Instead, we move towards an architecture where multiple autonomous MonetDB instances are joining together to process a large and distributed workload.

1.3 Manual Generation

The MonetDB code base is a large collection of files, scattered over the system modules. Each source file is written in a literal programming style, which physically binds documentation with the relevant code sections. The utility program Mx processes the files marked *.mx to extract the code sections for system compilation or to prepare for a pretty printed listing.

The reference manual is based on *Texinfo* formatted documentation to simplify generation for different rendering platforms. The components for the reference manual are extracted by

```
Mx -i -B -H1 <filename>.mx
```

which generates the file <filename>.bdy.texi. These pieces are collected and glued together in a manual framework, running *makeinfo* to produce the desired output format. The *Texinfo* information is currently limited to the documentation, it could also be extended to process the code.

A printable version of an *.mx file can be produced using the commands:

```
Mx  <filename>.mx
pdflatex <filename>.tex
```

The typographical conventions used in this manual are straightforward. MONOSPACED text is used to designate names in the code base and examples. *Italics* is used in explanations to indicate where a user supplied value should be substituted.

Snippets of code are illustrated in SMALL CAPS font. The interaction with textual client interfaces uses the default prompt-setting of the underlying operating system.

Keywords in the MonetDB interface languages are case sensitive; SQL keywords are not case sensitive. No distinction is made in this manual.

1.4 Download and Installation

The MonetDB system is provided as open-source software and can be downloaded from http://monetdb.cwi.nl/Download/. The below information explains the different options to choose from.

MonetDB/XQuery can be installed in different ways, depending on:

- whether you want the stable release, or the latest development version.
- whether you want to compile yourself or you want a pre-compiled binary.
- what operating system you use.

MonetDB/XQuery consists of the software modules:

- MonetDB: the database kernel.
- MonetDB4: the query algebra interpreter and scheduler (still version 4).
- clients: the mclient utility, and MAPI libraries.
- XQuery: the Pathfinder compiler and its runtime support (including PF/Tijah)

Each module is identified by a major and minor version number, e.g. 4.20.

A Stable Release contains all relevant software modules that are tested together. Typically, the major numbers of the various modules are different, but the minor numbers match.

Stable Releases have **even** minor numbers, the Current development version numbers are **odd**.

Bug-fixes may get consolidated and tested into a bug-fix release, which adds another (even) modifier: e.g. 4.20.2.

1.4.1 Stable Release vs Stable Branch vs Current Branch

MonetDB/XQuery is released on a regular basis (multiple releases per year).

On a **major** release, the Current Branch in the CVS code repository becomes the Stable Branch. Typically, this is a crucial phase for the developer community where the Current Branch is tested and made stable, while a so-called "code-freeze" is enforced temporarily. After the moment that a new Stable has been created, the restrictions on the Current Branch are lifted.

Bugs are reported on the SourceForge Bug Tracker and are fixed by the community.

In the course of time, bug-fixes may be checked into the Stable Branch. Thus, the CVS code of the Stable Branch is not necessarily equal to the last Stable Release.

On a **minor** release, all bug-fixes made in the last Stable Branch are explicitly re-tested and get a new version number. For Windows, new installers are built that are put on the download page.

Developers should work on the Current Branch. The Current Branch is nightly tested (see the Test Web). Often recent modifications cause some tests to fail. Thus, the Current Branch may at times be unstable ("bleeding edge"). It is not suited for production use.

1.4.2 Linux RPMs

RPMs for the latest Stable release are available via SourceForge and also via YUM. You have to install an RPM for each software module. If you use YUM to get XQuery, it will install the other modules automatically (as XQuery depends on them).

1.4.3 Windows Installers

The Windows installer for the latest Stable release is provided in two flavors: 32-bits and 64-bits. If you happen to run a 64-bits version of Windows, we recommend the latter version, as the intensive use of memory mapping in MonetDB strongly favors a large 64-bit virtual addressing space.

The installer is a typical Windows "easy" install that installs all needed software using a click-able GUI.

1.4.4 Super Source Tarball

The quickest way to build MonetDB yourself is to download a super-source tarball and use the MONETDB-INSTALL.SH script to compile it. On many Linux and Mac OS X distributions, this script will work out-of-the-box.

```
usage: ./monetdb-install.sh < OPTS ... >

where OPTS are:
   --prefix=path     install into location path, defaults to /ufs/mk/MonetDB
   --build=path      use path as (temporary) build directory, defaults
                     to /var/tmp/MonetDB-XXXXXXXXX
   --enable-sql      build the MonetDB/SQL server
   --enable-xquery   build the MonetDB/XQuery server
   --nightly=target  download and install a nightly snapshot of the stable
                     or current branch, target must be 'stable' or 'current'
   --cvs             checkout a CVS snapshot of the current branch
   -j[X]             use parallel make with the optionally given limit
   --enable-debug    compile with debugging support via e.g. gdb
   --enable-optimise compile with high optimisation flags, enabling this
                     option increases compilation time considerably but
                     often yields in a faster MonetDB server
   --enable-optimize alias for --enable-optimise
   --quiet           suppress output going to stdout
   --help            this message
   --devhelp         special help for developers
   --version         show revision number and quit
```

The monetdb-install.sh script is a little helper, it does nothing more than executing some commands to help you get the complex process of getting a useful MonetDB instance running. The script itself is not a "distribution" at all, it is only a mere meta-file. Second, that you get confused in the jungle of branches, tags, releases, snapshots and more of that is quite understandable. However that seems to be the faith of MonetDB. Hence, the monetdb-install.sh script tries to help you again here.

Referring to the quote now, there are two options that in total select 3 out of the maze of branches, etc. To make it easier, in fact, MonetDB knows (short version) only two "branches", a "current" and a "stable" branch. Of course these branches remain in CVS, where our development is tracked. The –cvs option to monetdb-install.sh simply selects the "current" branch (you cannot choose), and makes a CVS checkout for that. With that option you get really up-to-date sources, but that comes at the price of needing more tools to build, which may be missing on your system. Exactly for that reason we do some preparatory steps on the CVS sources every night, removing most of the build tool requirements. We call those "prepared" sources "nightly snapshots". That is where the –nightly option comes in. If you don't require bleeding edge up-to-date sources, you best take a nightly snapshot. The only thing here, is that you have to choose: do you want the "current" or "stable" branch? Roughly, the "stable" branch means the latest release + bugfixes that we applied. The "current" branch contains new features and ground-breaking improvements. If you're looking for a solid and stable playground, you best take

the "stable" branch (–nightly=stable). If you are adventurous, and require new features, you best take the "current" branch. Note that when using –cvs you implicitly belong to this brave adventurous group of people

1.4.5 CVS Sources

Read-only access to CVS is available via SourceForge. If you want to become a developer, you can apply for a developer account there.

Please read the instructions on SourceForge how to use CVS.

MonetDB/XQuery consists of the CVS modules:

- buildtools
- MonetDB
- clients
- MonetDB4
- pathfinder

These modules roughly correspond to the released components, but "XQuery" is called "pathfinder", and there is also the "buildtools" module, necessary for compilation. The listed order is also the order of compilation and installation.

Compilation starts with bootstrapping (which invokes our buildtools), then configure (i.e. GNU autogen and automake), then "make install".

Note that MonetDB/XQuery requires certain prerequisite packages. The exact details of these and the compilation procedure on Unix-like systems are described in the **HowToStart** file

`../MonetDB/HowToStart.rst`

Windows compilation of Pathfinder does **not** work with the Microsoft Visual C++ compiler, due to its lack of support for the C-99 standards. The recommended compilation option (used in the distribution) is the Intel C++ Compiler. Specific Windows compilation information is found here:

`../buildtools/doc/windowsbuild.rst`

It is technically also possible on Windows to use the cygwin Unix emulation library, and even possible to build a native (non-cygwin-dependent) MonetDB/XQuery using the so-called MINGW version of the cygwin gcc compiler.

We have, however **abandoned** the cygwin approach to Windows compilation, so this compilation path is not well-tested anymore. We abandoned it because:

- 64-bits compilation is not supported in cygwin.
- cygwin compilation is very slow.
- the reliability and speed of the Intel Compiler built version is better.

1.5 Development Roadmap

The XQuery compiler is currently only available on MonetDB Version 4, and is still based on the early ("milprintsummer") prototype of the Pathfinder compiler.

There is a new, much better algebraic version of Pathfinder available now. Also, MonetDB/XQuery lacks cost-based optimization, and the update system is still rather unoptimized.

The next major release (Q1 2008) should:

- provide all functionality (inclusive all extensions) on top of the Pathfinder algebra compiler.

That done, the following major release has as main theme:

- port the runtime system to MonetDB5. This includes porting the shredder, serializer, XPath support (staircase joins), transactional (working set) administration, node construction and (optimized) update processing.

Other features that in parallel are being worked on:

- adding date/time support.
- XQTS (W3C XQuery Test Suite) compliance.
- XQUFTS (W3C XQuery Update Facility Test Suite) compliance.
- cost-based dynamic query optimization
- bandwidth savings in XRPC, and automatic query distribution.
- XRPC distributed update transactions with 2PC.
- update facility improvements.
- port PF/Tijah to support the XQuery Full-Text specification.

2 Server Management

This section explains you everything a database administrator (or power user) needs to know about managing MonetDB/XQuery installations.

2.1 Starting and Stopping

2.1.1 Linux, Mac OS X, and other Unix

A MonetDB/XQuery is built on the MonetDB engine (`Mserver`). The XQuery functionality is provided by the `pathfinder` extension module, which contains the Pathfinder XQuery-to-Relational compiler.

The server is thus started with the following command:

```
prompt> Mserver --dbinit="module(pathfinder);"
```

Other relevant options are:

- `--dbname=<DBNAME>` specify which database to open. If omitted, MonetDB uses the default database name `demo`. All database files are located in the `<gdk_dbfarm>/<DBNAME>` and `<xquery_logs>/<DBNAME>` directories. The values of `gdk_dbfarm` and `xquery_logs` are set in the `MonetDB.conf` file (see Section 2.6 [MonetDB.conf], page 12).
- `--set <OPTION>=<VALUE>` Any variable defined in `MonetDB.conf` can be overridden at server startup. If you have multiple database instances, apart from `--dbname` you will typically have to specify a `--set mapi_port=XXX` to avoid errors caused by two `Mservers` on the same machine trying to use the same TCP/IP port.

When the server starts up, you should see something like:

```
# MonetDB Server v4.20.0
# based on GDK   v1.20.0
# Copyright (c) 1993-2007, CWI. All rights reserved.
# Compiled for i686-pc-linux/64bit with 32bit OIDs; dynamically linked.
# Visit http://monetdb.cwi.nl/ for further information.
# PF/Tijah module v0.3.0 loaded. http://dbappl.cs.utwente.nl/pftijah
# MonetDB/XQuery module v0.20.0 loaded
# XRPC administrative console at http://localhost:50001/admin
MonetDB>
```

This tells that:

- the database software uses the 4.20 version of MonetDB, based on the 1.20 kernel, and the 0.20 XQuery front-end.
- the software was compiled for 64-bits Linux PCs using 32-bits OIDs
- the administrative GUI has started on `http://localhost:50001/admin`

Warnings like the following:

```
!WARNING: GDKlockHome: created directory /local/MonetDB/
```

are normal: the server creates an empty database if you run it for the first time.

Stopping the server The server is stopped by typing `exit();` in the `MonetDB` console prompt (or by killing the Mserver process using CTRL-C or `kill`).

However, the `exit();` method is to be preferred, as it checkpoints the database and clears the log. Thanks to the empty log, server startup will be much faster, as no recovery will be needed.

2.1.2 Windows

Simply click: 'Start' -> 'Programs' -> 'MonetDB XQuery' -> 'MonetDB XQuery Server'. This will start the MonetDB Server with XQuery support in a separate window. Although the window comes with an interactive prompt, you should (unless you know what you are doing) keep this window minimized.

Stopping the server To stop the MonetDB Server, you can close the `MonetDB XQuery Server` window, or type `exit();` in the server window, as described earlier.

Regrettably MonetDB is not yet available as a windows service. It has to be started explicitly by a logged in user.

2.2 Adding and Deleting Documents

MonetDB/XQuery comes with an Administrative GUI (see Section 3.2 [The Administrative GUI], page 26) that allows to add and delete single documents at a time by point-and-click with a mouse.

Adding and deleting documents also can be done using XQuery queries. As the XQuery standard does not specify anything about this, we added the following extension functions:

- `pf:add-doc($url as xs:string, $alias as xs:string)`
- `pf:add-doc($url as xs:string, $alias as xs:string, $coll as xs:string)`
- `pf:add-doc($url as xs:string, $alias as xs:string, $coll as xs:string, $perc as xs:integer)`
- `pf:del-doc($name as xs:string)`

For an easy walk-through of the built-in extension functions (see Section 4.2.1 [Document Management Functions], page 38), we refer to the Document Management Tutorial. There is also a Administrative GUI Tutorial.

If you must add many (tens, hundreds, thousands..) of documents in one go, please read the instructions in the Performance Tips section (see Section 2.9.3 [Bulk Loading a Collection], page 18).

The query the database contents (i.e. the catalog: which collections and documents exist?) is also done in XQuery:

- `pf:documents() as node()`
- `pf:collections($coll as xs:string) as node()`
- `pf:collections($name as xs:string) as node()`

These built-in extension functions are called Metadata Functions (see Section 4.2.2 [Metadata Functions], page 39).

2.3 Collections versus Documents

MonetDB/XQuery allows you to store multiple **XML collections**. Each XML collection in itself consists of at least one **XML document**, and is stored in a fixed set of tables (a main `100000XX_rid` table containing XML nodes, and supporting tables for text and attributes, QNames, etc). In principle, an XML collection can contain many (even millions) of documents. Thus, deciding to store many XML documents in a single collection, creates much less internal relational tables than storing each document in a separate collection (which is default).

On the other hand, storing documents together in the same collection means that access to these documents will create some interference (locking the tables, and shared index maintenance). So, the decision how to group your XML documents in collections is a **physical database design** problem that the database administrator should think about.

These trade-offs are described further in the Performance Tips section (see Section 2.9.1 [Separate Documents vs Document Collections], page 15).

A collection exists as long as it holds at least one document. When the last is deleted, it automatically disappears.

2.4 Read-only versus Updatable

Another decision is whether a collection should be **read-only or updatable** (i.e. updates are allowed). Read-only collections are a bit more compact, and have faster indices that preserve document order. Updatable collections, however, have much cheaper index maintenance in the case that new documents are added to a collections. Therefore, even in some read-only use cases, namely those with frequent document additions, it is beneficial to use updatable collections.

Whether a collection is updatable or read-only is decided when it is created (i.e. when the first document is added to it), and cannot be changed after that. However, the backup/restore procedure provides a workaround for this limitation.

2.5 Backup/Restore

The Administrative GUI provides backup/restore functionality (see Section 3.2 [The Administrative GUI], page 26).

Backups get stored in a sub-directory under `<gdk_dbfarm>/<dbname>/backup/<name>` (where <name> is the name of the backup). Inside each such directory, a new sub-directory is created for each XML collection in the database. Inside each collection sub-directory, various numbered directories are created, and inside these all XML documents are serialized using the `fn:put()` function (see Section 4.3.5 [The put() Function], page 44).

The `<gdk_dbfarm>/<dbname>/backup/<name>` directory contains all information to recreate the database, so you may archive/compress such a directory safely using standard file-based backup mechanisms.

2.6 MonetDB.conf

The file `MonetDB.conf` contains the configuration options of MonetDB. To change these options, you have to edit this file with your favorite text editor, save it and **restart the server**.

2.6.1 Where is MonetDB.conf located?

This file tends to be located in the `etc/` sub-directory of your MonetDB installation. You can find this value by opening the Administrative GUI in `http://127.0.0.1:50001/admin` you can click the **View DB environment** button.

If MonetDB/XQuery runs on some other `machine`, you can open the Administrative GUI on the URL `http://machine:50001/admin`, but by default the security settings do not allow that. You can type in the Mserver console window: `xrpc_trusted.delete();` to lift these restrictions (they will be re-imposed on server restart).

Alternatively, you can type in the **Mserver console window**:

```
monet_environment.find("config").print();
```

2.6.2 Version Information

- `gdk_arch`: compilation architecture, a concatenation of address space (e.g. `64bit`), instruction set (e.g. `i686-pc`) and operating system (e.g. `Win32`): `64biti686-pc-win32`
- `gdk_version`: MonetDB kernel version number, e.g. `1.20.0`
- `monet_version`: MonetDB MIL interpreter version, e.g. `4.20.0`

2.6.3 Database Directory Options

- `config`: the `MonetDB.conf` file, e.g. `C:\Program Files\CWI\MonetDB4\etc\MonetDB.conf`
- `datadir`: directory with static installation files, e.g. `C:\Program Files\CWI\MonetDB4\share`
- `gdk_dbfarm`: directory your data will be stored, e.g. `C:\Users\testuser\AppData\Roaming\MonetDB4\db`
  This one should point to the drive/file system where you have enough room and I/O bandwidth to store your data. On Windows, make sure that the log and data directories are not indexed by the Windows Indexing Service (this is default for `AppData` directories).
- `gdk_dbname`: the name of the database. The default name is `demo`. Sub-directories by this name will be created in the `gdk_dbfarm` directory, where the table data and the logs will be stored.

2.6.4 mclient Options

see Section 3.1 [The Mapi Client Utility], page 22.

- `mapi_open`: whether clients can connect from other machines, default `false`. This means that by default, other machines **cannot connect to Mserver**. This rigorous setting default is intended to protect unwitting users from the lack of user/password+SSL security in MonetDB4.
- `mapi_clients`: maximum number of concurrent query session, e.g. `2`, basically the threading level.

- `mapi_port`: TCP/IP port for `mclient` connections, e.g. `50000`. Note that the XRPC port `xrpc_port`, if unset, defaults to `mapi_port+1`.
- `xquery_output`: The `mclient -f` serialization mode (see "Output Modes" under Section 3.1 [The Mapi Client Utility], page 22). Basically, `dm` or `xml`; where the latter mode can be augmented with `-typed`, `-noheader`, `-noroot` or `-root-FOOBAR`

2.6.5 XML Document Cache Options

see Section 2.8 [XML Document Cache], page 15.

- `xquery_cacheMB`: maximum size in megabytes of XML cache, e.g. `100`
- `xquery_cacherules`: XML cache lifetime rules, consisting of semicolon separated `URLprefix=seconds` specifications, e.g. `http://monetdb.cwi.nl=1600;http://www.slashdot.org=80`
- `xquery_procMB`: maximum size for the module cache, e.g. `128`. The module cache contains query plans; it is an internal setting.

2.6.6 StandOff Options

see Section 5.7 [StandOff Extension], page 60.

- `standoff_ns`: Standoff attribute namespace, default empty.
- `standoff_start`: StandOff attribute name, default `start`
- `standoff_end`: StandOff attribute name, default `end`

2.6.7 XRPC Options

see Section 5.5 [XRPC Extension], page 49.

- `xrpc_admin`: semi-colon separated list of trusted hosts (prefix) to open the Administrative GUI from, e.g. `127.0.0.1; 192.168.2`; an empty list means all hosts are trusted.
- `xrpc_open`: whether XRPC accepts connections from other hosts, default `true`
- `xrpc_port`: port number for the XRPC built-in HTTP server, default `mapi_port+1`
- `xrpc_trusted`: semi-colon separated list of trusted module URI prefixes, e.g. `http://monetdb.cwi.nl;C:\Program Files\CWI\MonetDB4\share\MonetDB\xrpc\export`; empty means all modules are trusted.

2.6.8 Kernel Tuning Options

- `gdk_vm_minsize`: column size above which memory mapped files are used always, e.g. `1749291171`
- `gdk_mem_maxsize`: maximum memory load, e.g. `1749291171`
- `gdk_mem_pagebits`: number of bits used for XML page addressing (pagesize), e.g. `16` (=pages of 2**16=65536 nodes). The minimum setting on Windows is 16, because of page alignment restrictions. On Linux, the minimum is 12.
- `gdk_vmtrim`: whether virtual memory background save thread should run, e.g. `yes`

2.7 Security

When starting the MonetDB Server, it will open two network ports on your system (the `mapi_port` and the `xrpc_port` (by default 50000 and 50001), configurable in the `MonetDB.conf` file (see Section 2.6 [MonetDB.conf], page 12).

by default, however, the server is inaccessible from other machines.

If you want the database to be accessible from the outside, you must:

- modify your firewall configuration to open (one of) these ports.
- additionally, for `mclient -lx` access, set the `mapi_open` variable to `yes`.
- additionally, for XRPC access, set the `xrpc_open` variable to `yes`, **and** enter in `xrpc_trusted` the URI prefixes of XQuery modules from which functions are allowed to be executed (note that making it empty will allow any module to be executed!).
- finally, to allow access to the Administrative GUI from another machine, add those machine names to `xrpc_admin` variable (note that making it empty will allow connections from anywhere!).

2.7.1 Security Warning

Before opening up access, take note of the following issues:

- Currently, MonetDB/XQuery lacks proper authentication. There is a single user with administrative rights and fixed passwords, and SSL is not available yet. This will be fixed by the the upcoming **version 5** port of MonetDB/XQuery.
- Consequently, access to the database is all or nothing: either nobody has access or anybody. And, all users can make changes, delete documents, etc. That is, there is no concept of restricted user rights (everybody is an administrator)
- The fact that the `fn:doc()` function allows to read documents from a file path means that any XML file that can be read by the user that started `Mserver` will be accessible from the outside. For this reason, it is advisable to install MonetDB under a user account with restricted permissions (as MonetDB is not a Windows Service yet, this is not practical on Windows).
- Moreover, the MonetDB server was not designed with security as a first goal, so may be susceptible to e.g. buffer overrun attacks. Thus, while in principle users are just able to execute XQueries, an open port could in the worst case lead to your computer being hacked.

As a consequence, we strongly advise not to allow `mclient` access to MonetDB/XQuery from the internet. If `mclient` access is opened up for other machines inside your organization, your firewall must block access to it from the internet. The same goes for the Administrative GUI (`xrpc_admin`). For XRPC access, internet access might be given if the `xrpc_trusted` only lists those XQuery modules that you have verified to contain only innocuous functions. Still, you must be aware that you take the risk of running into an unknown buffer overrun issues, and might face denial-of-service attacks (by hackers that may send queries that slow down or crash your system).

Disclaimer: the CWI provides MonetDB/XQuery "as-is" for free, and does not accept any liability for its use (see the MonetDB License and Pathfinder License).

2.8 XML Document Cache

The **document cache** holds XML documents recently accessed with `fn:doc()` by their URI (i.e. documents that were not added explicitly to the database with `pf:add-doc()`).

Some observations on the XML cache:

- It is persistent across Mserver sessions.
- There is a **caching policy** that determines whether and until when a document that is loaded by an executing XQuery will stay in the cache.
- An important parameter is the size of the cache. It is controlled by the `xquery_cacheMB` variable in the `MonetDB.conf` file (see Section 2.6 [MonetDB.conf], page 12). This value is in megabytes (MB).
- A general rule is that MonetDB/XQuery **always** caches file URIs. The system uses the last-modified timestamp of the file system to guarantee that when a query is run, the cached document is actual (if it has changed on disk, the document is removed from the cache automatically, and is shredded anew).
- Other caching **lifetime-rules** may be specified to govern the caching of specific URLs. Such rules take the form `URLprefix:seconds` stating that all URLs starting with `URLprefix` should be cached for a certain number of seconds. The set of all rules should be concatenated with semi-colons, e.g. `http://monetdb.cwi.nl=1600;http://www.slashdot.org=80`
- the current size of the cache can be monitored through the Administrative GUI by clicking "View Database Statistics". The variables `xquery_cache_curdocs` and `xquery_cache_curMB` hold respectively the number of documents cached and the total size of the cache in megabytes.
- when the cache is full, the policy is currently to empty it completely. This is done automatically.

2.9 Performance Tips

In this section we explain some simple tricks that may enhance the performance of your MonetDB/XQuery application.

2.9.1 Separate Documents vs Document Collections

2.9.1.1 Storage Overhead

In MonetDB/XQuery all XML is stored in relational tables. Each document is stored in a separate table (and as MonetDB uses column-wise storage, each column in stored in a separate file and memory array). Each table (and column) however, even if it is empty, occupies some space on disk and in memory. In the case of the XML tables, the minimal size for en empty table is around 32KB.

Therefore, if the average size of the XML documents you store is much less than 32KB, and you have *many* (thousands, or millions) of them, storing each of them in a separate document in MonetDB/XQuery will result in a lot of memory and disk-space being wasted, and queries running slower.

For such usage scenarios, it is much better if MonetDB/XQuery can store many XML documents together in a single relational table.

This is made possible using the XQuery concept of a *collection*. When you add XML documents to the database with `pf:add-doc(url,name)` it gets stored in a separate new collection (that has the same name `name`).

However, if you pass an extra parameter `pf:add-doc(url,name,collection)` the document is added to the collection `collection`. If `collection` already existed, the document gets appended to it.

2.9.1.2 fn:collection() vs pf:collection()

XQuery supports the collection concept using the standard builtin function `fn:collection(name) as node()*`, which returns a *set* of document nodes that belong together. In MonetDB/XQuery it is perfectly feasible to have collections that contain millions of (small) documents.

XML documents are trees, and in MonetDB/XQuery, a collection is also made into a tree, by automatically adding a *super-root* node above all document nodes of the collection. MonetDB/XQuery also provides the built-in extension function `pf:collection(name) as node(` that returns this super-root. Thus, `fn:collection(name)` is roughly equivalent to `pf:collection(name)/child::*`. The extension function `pf:collection()` can be much faster than `fn:collection()`. on collections that have thousands of documents (or more). The reason is that the former returns just a single node, whereas the latter may return thousands.

2.9.1.3 Frequently Adding/Deleting Documents From Collections

If you have many small documents, store them together in a single (or a few) collection(s). Storing them physically together makes MonetDB/XQuery more efficient.

By default, collections are read-only. The fact that no updates occur on such collections is exploited by creating fully ordered inverted lists as index structures. However, such a fully sorted index needs to be rebuilt from the ground, each time a new document is added to the collection.

Note that updatable XML collections do not use the fully sorted inverted files, but rather use hash-tables. Hash tables can be maintained under updates cheaply and do not need to be rebuilt from scratch when a document is added to a collection.

Therefore, in situations where an existing collection is frequently extended with new documents, we recommend to make that collection updatable. This is done by passing yet another parameter `perc` to the first `pf:add-doc(url,name,collection,perc)` call, with which you create the collection. The `perc` indicates the per-page free-space that is left on pages to accommodate updates, and must be between 1 and 100 (a good value is 10).

2.9.2 Scalability

MonetDB/XQuery is quite scalable, when compared with other XQuery engines, being especially efficient in handling large (GBs) documents, by employing efficient **join** algorithms and advanced **self-tuning** indexing – both for structural (XPath traversals) and value-based queries (text and attribute values).

Still, there remain situations where scalability issues may appear. Here are a number of tips:

2.9.2.1 Making Sure Value Indices are used

MonetDB/XQuery automatically creates indices on all attribute and text node values, and these are used when expressions like:

```
(: accelerated by value index :)
<path1>[<path2>/text() = expr]

for $x in <path1>
where $x/<path2>/text() = expr
return $x

<path1>[<path2>/@attr = expr]

for $x in <path1>
where $x/<path2>/@attr = expr
return $x
```

This works regardless the type of `expr`; and `expr` may even be a loop-dependent expression (then we get a nested loop index join).

MonetDB/XQuery uses just-in-time query optimization based on sampling to determine whether the expression is selective enough to justify the use of an index.

warning: however, equality comparions on element nodes cannot be accelerated with these value indices:

```
(: not accelerated by value index :)
<path1>[<path2>/foo = expr]

for $x in <path1>
where $x/<path2>/foo = expr
return $x
```

The reason is that (barring a DTD or Schema knowledge – currently not exploited in MonetDB/XQuery) a comparison with the data value of an element, means that all descendant text node values have to be concatenated:

```
<foo>4<bar>2</bar></foo> = 42
```

evaluates to `true`! It is clear that this is hard to support with an index that stores the separate text values `4` and `2`.

For this reason, it is advisable to use `foo/text() = expr` comparisons rather than `foo = expr`.

2.9.2.2 Use Large Main Memories

MonetDB is a fast main-memory oriented database, that uses column-wise storage. The query engine, however, is known to consume quite a bit of RAM, especially on queries that generate large intermediate results. Therefore, having more RAM in your computer may strongly improve MonetDB performance. As a general principle, best performance is ensured if you have at least the amount of RAM roughly equal to the size of the XML documents that your queries are accessing.

2.9.2.3 Use 64-bits OS and MonetDB/XQuery

In 32-bits systems, the usable amount of RAM is limited to 4GB, and on most OSs even to 3GB (Linux) or 2GB (Windows). So, if you, after reading the previous tip, decided to put 4GB of RAM into your 32-bits machine, MonetDB/XQuery will *not* be able to use it all.

On 32-bits Windows, our binary distribution of MonetDB/XQuery *can* use the full 3GB because it is "large address aware" (Windows terminology). However, you must first configure windows to allow use of the full 3GB by large-address-aware applications, otherwise MonetDB/XQuery will be limited to using 2GB.

The better way to go with large data sizes, is to switch to a 64-bits operating system. MonetDB/XQuery is fully supported on 64-bits operating systems, and even comes with a binary distribution for 64-bits Windows. And even if you use the 32-bits MonetDB/XQuery binary on a 64-bits OS, it gets access to the 4GB instead of just 2 or 3GB.

The default 64-bits MonetDB/XQuery binaries are built with 32-bits object identifiers (OIDs). This is a compile-time option (the 64-bits versions are configured with `--enable-oid32`). If your XML documents have more than 2 billion elements (typically, we are then talking about XML in the size range of more than 40GB) you will hit storage limits inside MonetDB, if this XML is stored in a single XML collection. Also, with `--enable-oid32`) string columns in MonetDB are limited to 4GB (i.e. all *unique* text nodes in a collection are stored in a single column). To lift those restrictions, you should configure MonetDB with `--disable-oid32` and recompile.

2.9.3 Bulk Loading a Collection

To load many documents, the best approach is to use some shell language (shell-script, awk, perl, python) to generate an XML file that contains all file names (and if you wish document names). e.g. a file `/tmp/dir.xml`:

```
<dir>
<doc path="/foo/bar/" name="doc0000001.xml">
.....
<doc path="/foo/bar/" name="doc2300000.xml">
</dir>
```

you can then efficiently import all these documents using an XQuery over the temporary file `/tmp/dir.xml`:

```
for $d in doc("/tmp/dir.xml")//doc
 return
  pf:add-doc(fn:concat($d/@path,$d/@name), fn:string($d/@name), "my-coll", 0)
```

With the above, all documents will be loaded into a single collection `my-coll`, that is read-only (because the `pf:add-doc()` last parameter, `percentage=0`).

Note that when you have *many* documents, grouping them in one (or a few) XML document collections reduces storage and query processing overhead (see Section 2.9.1 [Separate Documents vs Document Collections], page 15).

2.9.4 XQuery Modules

MonetDB/XQuery has support for modules. It helps XQuery users to structure their query code, but are also the instrument for MonetDB/XQuery to implement **prepared queries** (see next section).

All XRPC requests benefit from the prepared query mechanism.

The below shows a simple example of an XQuery module `test.xq`, that just defines a single function `countDescendants("uri")`:

```
module namespace test = "http://monetdb.cwi.nl/XQuery/Documentation/Language/Modules/";

declare function test:countDescendants($doc as xs:string) as xs:integer
{
  count(doc($doc)//*)
};
```

You may type `import module` inside an XQuery query, after which you can use the functions (and variables) defined in it:

```
import module namespace test = "http://monetdb.cwi.nl/XQuery/Documentation/Language/Modul
                            at "http://monetdb.cwi.nl/XQuery/Documentation/Language/Modul
test:countDescendants("http://monetdb.cwi.nl/xmark/auctions.xml")
```

which basically does the same as the ad-hoc query, namely counting how many nodes the XMark document has:

```
count(doc("http://monetdb.cwi.nl/xmark/auctions.xml")//*)
```

Warning: while highly similar, the module feature as implemented by MonetDB/XQuery deviates in the following respects from the XQuery formal semantics:

- You *must* give a location hint in the "import module" statement. Each file hinted there will be loaded as a module. It has to match the namespace given in the "import module" statement, though.
- Modules cannot see variables declared in other modules, regardless if they imported the module themselves or not. A module is not allowed, though, to override variable declarations of other modules (conforming to the specs).
- Modules will see functions defined in other modules. They are not allowed to override them, though.
- All modules and the main query share the same type definitions. So modules will see XML Schema definitions imported by the main query. (see also below for XML Schema import)
- Pathfinder does allow cyclic importing of modules, regardless of their namespace.
- The XQuery specifications state that two module import statements that use the same target namespace should produce an error. This is not the case in MonetDB/XQuery: the module will be loaded once, but its functions and variables will be available under both namespace identifiers.

2.9.5 Expression Caching

Expression Caching is a powerful feature to create well-performing functionally rich applications. You can use it for query result caching (avoiding to compute the same query twice), as a mechanism to simulate cursors; allowing an expensive query that delivers large result to be evaluated once, allowing subsequent queries to show small parts of a result set, that eg fit on the screen.

The mechanism allows Caching of Arbitrary Subexpressions inside a so-called Multi-Query Session.

To profit from these, one should restructure the data access of applications to use the same database snapshot for multiple queries (with a long enough timeout to be sure the session stays alive in the cache).

As a second step, the XQuery queries should be analyzed and interesting expressions could be marked up. The marked up expressions are cached; if a query find them already cached, the result is available instantly.

One typical use of subexpression caching is skipping quickly paging back and forth through a large query result (what SQL users use "cursors" for), e.g. showing the elements in range `[LO,HI]`, without having to recompute the entire result for each query:

```
(# pf:session my-own-id:30000 #) {
  subsequence((# pf:cache my-male-persons #) { doc("auctions.xml)//person[gender = "male"] }
}
```

Note the entire query is wrapped in a `pf:session` XQuery pragma, which gives the session an ID (`my-own-id`). It also specifies a keep-alive time for the session of 30 seconds here. Then, inside the query body, the subexpression that retrieves all male persons is wrapped in a `pf:cache` pragma, identifying the expression by ID again (`my-male_persons`). The effect is that the first query will compute all male persons, but all subsequent invocations will have them already available; these queries will return instantly.

The in-session subexpression caching mechanism described in detail in see Section 5.3 [Session Expression Cache], page 46.

2.9.6 Prepared Queries

WARNING: the prepared function cache still makes use of the old compiler backend and does not use the optimizing algebraic query compiler. Therefore, its query performance can sometimes be inferior to other queries handled by MonetDB/XQuery. Also, as the old compiler backend is gradually phased out, it gets to be less well-maintained and tested in general. Use with caution. We hope to port XRPC to the algebra backend soon.

In SQL databases, interfaces like ODBC or JDBC allow to prepare **prepared queries** for faster execution of frequently used query patterns. For example, web-sites powered by a database usually generate a **fixed** set of queries to that database; each page fires off the same (set of) queries, possibly though with different parameters (that contain e.g. a customer identity or cookie). SQL systems can **prepare** for such prepared queries in advance, such that when a web-page is served out and the query result is needed quick, only the actual parameters need to be fed (**bound**) into a ready-to-run query (saving query parsing and optimization time, which is unavoidable for ad-hoc queries).

Such prepared queries can be seen as a queries whose result is a function of a number of parameters. MonetDB/XQuery takes this definition literally in its implementation of prepared queries:

prepared query = XQuery function

XQuery modules allow to define XQuery functions (and variables, but that's beside the point now). So:

- if a XQuery consists of only a function call, and
- that function is defined not inside the query itself, but inside a module (so there's an `import module` statement preceding the function call), and

- the query parameters are simple constants (atomic values of type `xs:integer`, `xs:decimal`, `xs:double`, or `xs:string`)

then MonetDB/XQuery will execute this query much faster, making use of a **prepared query plan**.

To put the caching mechanism to the test, first import the XMark document into the database:

```
mclient -lx
> pf:add-doc("http://monetdb.cwi.nl/xmark/auctions.xml",
             "http://monetdb.cwi.nl/xmark/auctions.xml")
```

then run the queries q.xq (standard), m.xq (function from module) while monitoring elapsed time:

```
shell> mclient -lx -t q.xq
1729

Trans      18.000 msec
Shred       0.000 msec
Query       5.000 msec
Print       0.000 msec
Timer      24.436 msec

shell> mclient -lx -t m.xq
1729

Trans       0.000 msec
Shred       0.000 msec
Query       2.000 msec
Print       0.000 msec
Timer       2.778 msec
```

We see that the latter "prepared" query (that uses a function defined in a module) performs almost **ten times** faster than the ad-hoc query! The translation time has completely disappeared for the prepared query `m.xq` and the query execution also improved by a factor of two.

This performance trend was confirmed on the XMark benchmark. We developed prepared queries for the benchmark, which produced a factor 10 (small 116KB documents) to 2 (larger 10MB documents) performance increase over the ad-hoc benchmark queries. Obviously, complex queries or queries on huge (>=GB) documents, which take many seconds to run anyway, do not benefit from prepared query mechanism (but neither are hindered by it).

As a general rule, the benefit of canned queries is larger for short-running than for long-running queries, because for short-running queries the XQuery translation and optimization time weighs in more heavily. We especially recommend the use of canned queries when MonetDB/XQuery is used to power web-sites.

3 Client Interfaces

Client interfaces allow end-users to interact directly (pose queries, perform updates) with MonetDB/XQuery.

The `mclient` is an command-line utility (based on Mapi) that can be used interactively, or in shell (.bat) scripts. It is the easiest way to start testing and working with MonetDB/XQuery.

The Administrative GUI is a web-based interface that allows you to browse your database contents, add and delete new documents, and perform backup/restore.

3.1 The Mapi Client Utility

The `mclient` program is the universal command-line tool that implements the MAPI protocol for client-server interaction with MonetDB.

On a Windows platform it can be started using start->MonetDB->MonetDB SQL Client. Alternatively, you can use the command window to start `mclient.exe`. Be aware that your environment variables are properly set to find the libraries of interest.

On a Linux platform it provides readline functionality, which greatly improves user interaction. A history can be maintained to ease interaction over multiple sessions.

A `mclient` requires minimally a language and host or port argument. The default setting is geared at establishing a guest connection to a SQL or XQuery database at a default server running on the localhost. The `-h hostname` specifies on which machine the MonetDB server is running. If you communicate with a MonetDB server on the same machine, it can be omitted.

The timer switch reports on the round-about time for queries sent to the server. It provides a first impression on the execution cost.

```
Usage: mclient --language=(sql|xquery|mal|mil) [ options ]

Options are:
 -d database | --database=database  database to connect to
 -e          | --echo              echo the query
 -f kind     | --format=kind       specify output format {dm,xml} for XQuery, or {csv,t
 -H          | --history          load/save cmdline history (default off)
 -h hostname | --host=hostname     host to connect to
 -i          | --interactive      read stdin after command line args
 -l language | --language=lang     {sql,xquery,mal,mil}
 -L logfile  | --log=logfile       save client/server interaction
 -P passwd   | --passwd=passwd     password
 -p portnr   | --port=portnr       port to connect to
 -s stmt     | --statement=stmt    run single statement
 -t          | --time              time commands
 -X          | --Xdebug            trace mapi network interaction
 -u user     | --user=user         user id
 -?          | --help              show this usage message
 -| cmd      | --pager=cmd         for pagination
```

```
SQL specific opions
 -r nr       | --rows=nr          for pagination
 -w nr       | --width=nr         for pagination
 -D          | --dump             create an SQL dump

XQuery specific options
 -C colname  | --collection=name  collection name
 -I docname  | --input=docname    document name, XML document on standard input
```

The default `mapi_port` TCP port used is 50000. If this port happens to be in use on the server machine (which generally is only the case if you run two MonetDB servers on it), you will have to use the `-p port` do define the port in which the `mserver` is listening. Otherwise, it may also be omitted. If there are more than one mserver running you must also specify the database name `-d database`. In this case, if your port is set to the wrong database, the connection will be always redirect to the correct one. Note that the default port (and other default options) can be set in the server configuration file.

Within the context of each query language there are more options. They can be shown usin the command \? or using the commandline.

```
shell> mclient -lx --help
mclient interactive MonetDB/XQuery session: type an XQuery or XQUF update.

Supported document-management XQuery extensions:
 pf:collections() as node()
 pf:documents($collectionName as xs:string) as node()
 pf:del-doc($documentName as xs:string)
 pf:add-doc($uri as xs:string, $documentName as xs:string
     [,$collectionName as xs:string [,$freePercentage as xs:integer]])

Session commands:
<>      - send query to server (or CTRL-D)
\?      - show this message
\<file  - read input from file
\>file  - save response in file, or stdout if no file is given
\|cmd   - pipe result to process, or stop when no command is given
\h      - show the readline history
\t      - toggle timer
\f      - result format: dm or xml[-noheader][-typed|-noroot|-root-FOOBAR]
\L file - save client/server interaction
\X      - trace mclient code
\q      - terminate session
xquery>
```

3.1.1 Adding Documents

It is possible with `mclient` to add a local XML document (e.g. `/tmp/newdoc.xml`) to the database:

```
shell> mclient -I newdoc.xml -C my-coll < /tmp/newdoc.xml
shell>
```

The `-I docname` switch is used to indicate the document name, and the optional `-C collname` to indicate the collection in which it should be stored. The document is expected on standard input.

TODO: it is not yet possible to specify a free percentage (see Section 2.2 [Adding and Deleting Documents], page 10), so if the collection does not exist already, `mclient` will create it as a read-only collection always.

3.1.2 Timing

The -t option causes timing info to be printed, after the query has been executed:

```
shell> mclient -t -lx test.xq
"Hello World"
Trans      82.000 msec
Fetch       0.000 msec
Shred       0.000 msec
Query      37.000 msec
Print       6.000 msec
Total     131.000 msec
```

- `Trans` time it took to translate and optimize the XQuery to MIL algebra.
- `Fetch` time it took to retrieve/read XML documents that were used by `fn:doc(url)` statements in the query. Note that MonetDB has a document cache, so a document may already be cached in the database (in this case, no time is spent fetching nor shredding it).
- `Shred` time it took to shred (import) those XML documents into the database (includes `Fetch` time).
- `Query` time it took to actually execute the query (includes `Shred` time).
- `Print` time it took to print the query result.
- `Total` time everything took on the client side (includes `Trans+Query+Print`; any difference thus is communication time between client and server and the time to print the result on the command-line at the client.

3.1.3 Output Modes

The result of an XQuery is a value sequence; where the values may be nodes (i.e. snippets of XML) or simple values (such as integers or strings). Such sequences may be printed in different ways. This can be controlled with the `-f mode` switch.

- `none` print no output at all. Useful for performance monitoring, e.g. to exclude any effects of printing (large) results.
- `dm` print easily readable text. Atomic values appear as-is, and sequences result in commas.
- `xml` print in XML. sequences of atomic values appears as text nodes.

Let us consider the example XQuery `test.xq`:

```
(1, 42.0, "Hello World", <node attr="value">test</node>)
```

When executing the query in default mode, we get:

```
shell> mclient -lx test.xq
1,
42.000000,
"Hello World",
<node attr="value,test</node>
$
```

however, in `xml` mode we get:

```
shell> mclient -lx -f xml test.xq
<?xml version="1.0" encoding="utf-8"?>
<XQueryResult>
1 42.000000 "Hello World" <node attr="value,test</node>
</XQueryResult>
```

3.1.4 `xml` submodes

The output mode `xml` can be further configured as follows:

- `-noheader` results are printed in UTF-8, as specified by the first line of XML (the *header*. This header may be omitted if desired, by appending `-noheader` to the mode (thus: `-s xml-noheader`).
- `-noroot` By default MonetDB/XQuery generates an artificial XML root element with tag `XQueryResult`. This is desirable to ensure the output to be valid XML (in case of multiple result values). If you know that your query yields a single XML node, or you do not want the output to be valid XML, you may omit the artificial root by adding the sub-mode `-noroot` (thus: `-f xml-noroot`, or more useful: `-f xml-noheader-noroot`).
- `-root-FOOBAR` you may also want to change the name of the artificial root tag to e.g. `xyz`. This is done by adding `-root-xyz` as **last** submode (i.e. `-f xml-root-xyz`).
- `-typed` Finally, rigorous support for typed XML output is provided in yet another output mode just called `xml-typed`. The goal of this mode is to serialize XQuery results in such a way that all type information is retained. Atomic values and sequences are enclosed in resp. `<atomic-value>` `<sequence>` elements of the `http://monetdb.cwi.nl/XQuery/results` namespace. Type information is annotated with `xsi:type` attributes.

Below we demonstrate the `xml-typed` mode. It produces the following result (indenting and line breaks added by hand, for readability):

```
shell> mclient -lx -f xml-typed test.xq
<?xml version="1.0" encoding="utf-8"?>
<result:sequence
 xmlns:result="http://monetdb.cwi.nl/XQuery/results"
 xmlns:xs="http://www.w3.org/2001/XMLSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
     <result:atomic-value xsi:type="xs:integer">1</result:atomic-value>
     <result:atomic-value xsi:type="xs:decimal">42.000000</result:atomic-value>
     <result:atomic-value xsi:type="xs:string">Hello World</result:atomic-value>
     <result:element><node attr="value">test</node></result:element>
</result:sequence>
```

The default mode for MonetDB/XQuery is `dm` but can be set using the `xquery_output` variable in the `MonetDB.conf` file (see Section 2.6 [MonetDB.conf], page 12).

3.2 The Administrative GUI

The Administrative GUI uses the built-in XRPC HTTP server (see Section 5.5.3 [XRPC Server], page 53) to present a web-based database administration GUI. In fact, the GUI is a "glueless" pure-HTML application that uses the JavaScript XRPC API to interact with MonetDB/XQuery without any server-side code. As such, it is also a demonstration of its Javascript API for XRPC (see Section 6.1 [Using XRPC from JavaScript], page 65).

If you have MonetDB/XQuery running on your local machine, just point your browser to `http://127.0.0.1:50001/admin`. If your `Mserver` runs on a different `<machine>`, then point it to `http://<machine>:50001/admin`.

However, the Administrative GUI usually only allows clients from the local machine (for security reasons). For it to work from another machine, you have to add the name of that machine in to the semicolon list value of the `xrpc_admin` variable in the `MonetDB.conf` file (see Section 2.6 [MonetDB.conf], page 12). Giving `xrpc_admin` the value of the empty string, has the special meaning that you can connect from anywhere.

You can look at the Tutorial Page for a walkthrough.

The Administrative GUI provides the following functionality:

- **List Collections**: list all document collections, even those for which shredding is in progress. By clicking on a collection, you can see a list of all documents it contains.
- **List All Documents**: list all documents in the entire database. You can click "view" to see the entire XML file in your web browser (**warning:** web browsers typically can only handle a few MB of XML documents; do not try this with large XML files!!).
- **Add Document**: add a document (to an XML collection). There is a dialog window for selecting an XML file on the hard drive on the machine where your browser runs, but this route only works if you run the `Mserver` on the same machine as your browser. Otherwise you have to type a filepath that is valid on the `Mserver` machine (or a URL) to identify the document that you want to add.

  If you want to add **many** documents, we refer to the Performance Tips (see Section 2.9.3 [Bulk Loading a Collection], page 18).
- **View Database Statistics** displays statistics on the current size of the XQuery log (see Section 4.3 [XQuery Updates], page 41), the size of the XML Document Cache (see Section 2.8 [XML Document Cache], page 15) and current memory usage.
- **View DB Environment** shows the `MonetDB.conf` file (see Section 2.6 [MonetDB.conf], page 12, which is used to set all important configuration defaults for MonetDB/XQuery.
- **Backup/Restore Database** allows to perform Backup/Restore of the entire database system (see Section 2.5 [Backup/Restore], page 11).
- **XRPC Test Page** provides a click-able demo where you can see our XRPC at (see Section 5.5 [XRPC Extension], page 49).

4 XQuery Reference

MonetDB/XQuery provides a full-fledged XQuery implementation, including support for updates (the XQuery Update Facility).

More details about the exact feature set that is supported in MonetDB/XQuery can be found here:

This documentation makes no attempt to teach you the XQuery language. For this we refer to tutorials on the web, and to the reference information provided by the W3C:

- W3C XQuery 1.0 now full standard recommendation.
- W3C XPath 2.0 the XML navigation sub-language, common to both XQuery and XSLT.
- W3C XQuery Functions and Operators provides an overview of all built-in functions.
- W3C XQuery Update Facility is a proposal in-the-works for updating XML documents (we still use the July 2006 version).

Additional W3C reference material:

- W3C XML Schema 2.0 note that schema support in MonetDB/XQuery is incomplete at this point.
- W3C XQuery Serialization defines how XML documents should be converted to text.
- W3C XQuery Formal Semantics formally defines the behavior of all XQuery language constructs.

4.1 Supported Functions

MonetDB/XQuery supports a wide range of the built-in functions described in the W3C specifications. This section provides an overview is of functions that MonetDB/XQuery currently supports.

We encourage new Developers to volunteer and help fill in these blanks. Work has started on adding date/time functions but help is surely welcome.

4.1.1 Aggregation Functions

name	parameters	supported
`fn:count`	($srcval as item*) as xs:integer	yes
`fn:avg`	($srcval as xdt:anyAtomicType*) as xdt:anyAtomicType?	yes
`fn:max`		yes
`fn:max`	($srcval as xdt:anyAtomicType*, $collationLiteral as string) as xdt:anyAtomicType?	no
`fn:min`	($srcval as xdt:anyAtomicType*) as xdt:anyAtomicType?	yes
`fn:min`	($srcval as xdt:anyAtomicType*, $collationLiteral as string) as xdt:anyAtomicType?	no
`fn:sum`		yes

fn:sum	($arg as xdt:anyAtomicType*, $zero as xdt:anyAtomicType?) as xdt:anyAtomicType?	yes

4.1.2 Numeric Functions

name	parameters	supported
fn:number	() as xs:double	yes
fn:number	($srcval as item?) as xs:double	yes
fn:abs	($srcval as numeric?) as numeric?	yes
fn:ceiling	($srcval as numeric?) as numeric?	yes
fn:floor	($srcval as numeric?) as numeric?	yes
op:numeric- add	($operand1 as numeric, $operand2 as numeric) as numeric	yes
op:numeric- divide	($operand1 as numeric, $operand2 as numeric) as numeric	yes
op:numeric- equal	($operand1 as numeric, $operand2 as numeric) as xs:boolean	yes
op:numeric- greater-than	($operand1 as numeric, $operand2 as numeric) as xs:boolean	yes
op:numeric- integer-divide	($operand1 as xs:integer, $operand2 as xs:integer) as xs:integer	yes
op:numeric- less-than	($operand1 as numeric, $operand2 as numeric) as xs:boolean	yes
op:numeric- mod	($operand1 as numeric, $operand2 as numeric) as numeric	yes
op:numeric- multiply	($operand1 as numeric, $operand2 as numeric) as numeric	yes
op:numeric- subtract	($operand1 as numeric, $operand2 as numeric) as numeric	yes
op:numeric- unary-minus	($operand as numeric) as numeric	yes
op:numeric- unary-plus	($operand as numeric) as numeric	yes
fn:round	($srcval as numeric?) as numeric?	yes
fn:round- half-to-even	($srcval as numeric?) as numeric?	no
fn:round- half-to-even	($srcval as numeric?, $precision as integer) as numeric?	no
op:to	($firstval as xs:integer, $lastval as xs:integer) as xs:integer+	yes

4.1.3 Boolean Functions

name	parameters	supported
fn:boolean	($srcval as item*) as xs:boolean	yes

fn:false	() as xs:boolean	yes
fn:not	($srcval as item*) as xs:boolean	yes
fn:true	() as xs:boolean	yes
op:base64Binary-equal	($value1 as xs:base64Binary, $value2 as xs:base64Binary) as xs:boolean	no
fn:deep- equal	($parameter1 as item*, $parameter2 as item*) as xs:boolean	will
fn:deep- equal	($parameter1 as item*, $parameter2 as item*, $collationLiteral as string) as xs:boolean	will
fn:compare	($comparand1 as xs:string?, $comparand2 as xs:string?) as xs:integer?	yes
fn:compare	($comparand1 as xs:string?, $comparand2 as xs:string?, $collationLiteral as xs:string) as xs:integer?	yes
op:boolean- equal	($value1 as xs:boolean, $value2 as xs:boolean) as xs:boolean	yes
op:boolean-greater-than	($srcval1 as xs:boolean, $srcval2 as xs:boolean) as xs:boolean	yes
op:boolean-less-than	($srcval1 as xs:boolean, $srcval2 as xs:boolean) as xs:boolean	yes
op:hexBinary- equal	($value1 as xs:hexBinary, $value2 as xs:hexBinary) as xs:boolean	no

4.1.4 String Functions

name	parameters	supported
fn:concat	() as xs:string	yes
fn:concat	($op1 as xs:string?) as xs:string	yes
fn:concat	($op1 as xs:string?, $op2 as xs:string?, ...) as xs:string	yes
fn:contains	($operand1 as xs:string?, $operand2 as xs:string?) as xs:boolean?	yes
fn:contains	($operand1 as xs:string?, $operand2 as xs:string?, $collationLiteral as xs:string) as xs:boolean?	no
fn:default-collation	() as xs:anyURI?	no
fn:ends-with	($operand1 as xs:string?, $operand2 as xs:string?) as xs:boolean?	yes
fn:ends-with	($operand1 as xs:string?, $operand2 as xs:string?, $collationLiteral as xs:string) as xs:boolean?	no

fn:lower-case	($srcval as xs:string?) as xs:string?	yes
fn:matches	($input as xs:string?, $pattern as xs:string) as xs:boolean?	yes
fn:matches	($input as xs:string?, $pattern as xs:string, $flags as xs:string) as xs:boolean?	yes
fn:normalize- space	() as xs:string?	yes
fn:normalize- space	($srcval as xs:string?) as xs:string?	yes
fn:normalize-unicode	($srcval as xs:string?) as xs:string?	no
fn:normalize-unicode	($srcval as xs:string?, $normalizationForm as xs:string) as xs:string?	no
fn:starts-with	($operand1 as xs:string?, $operand2 as xs:string?) as xs:boolean?	yes
fn:starts-with	($operand1 as xs:string?, $operand2 as xs:string?, $collation-Literal as xs:string) as xs:boolean?	no
fn:string	() as xs:string	yes
fn:string	($srcval as item?) as xs:string	yes
fn:string-join	($operand1 as xs:string*, $operand2 as xs:string) as xs:string	yes
fn:string-length	() as xs:integer?	yes
fn:string-length	($srcval as xs:string?) as xs:integer?	yes
fn:string-pad	($padString as xs:string?, $padCount as xs:integer) as xs:string?	will
fn:replace	($input as xs:string?, $pattern as xs:string, $replacement as xs:string) as xs:string?	yes
fn:replace	($input as xs:string?, $pattern as xs:string, $replacement as xs:string, $flags as xs:string) as xs:string?	yes
fn:substring	($sourceString as xs:string?, $startingLoc as xs:double) as xs:string?	yes
fn:substring	($sourceString as xs:string?, $startingLoc as xs:double, $length as xs:double) as xs:string?	yes
fn:substring- after	($operand1 as xs:string?, $operand2 as xs:string?) as xs:string?	yes
fn:substring- after	($operand1 as xs:string?, $operand2 as xs:string?, $collation-Literal as xs:string) as xs:string?	no
fn:substring-before	($operand1 as xs:string?, $operand2 as xs:string?) as xs:string?	yes
fn:substring-before	($operand1 as xs:string?, $operand2 as xs:string?, $collation-Literal as xs:string) as xs:string?	no

fn:tokenize	($input as xs:string?, $pattern as xs:string) as xs:string*	no
fn:tokenize	($input as xs:string?, $pattern as xs:string, $flags as xs:string) as xs:string*	no
fn:upper-case	($srcval as xs:string?) as xs:string?	yes
fn:translate	($srcval as xs:string?, $mapString as xs:string?, $transString as xs:string?) as xs:string?	yes

4.1.5 Node Functions

name	parameters	supported
fn:data	($srcval as item*) as xdt:anyAtomicType*	yes
fn:nilled	($srcval as node?) as xs:boolean	no
fn:distinct- nodes	($srcval as node*) as node*	yes
fn:doc	($uri as xs:string?) as document?	yes
fn:collection	($srcval as xs:string) as node*	yes
fn:put	($node as node(), $uri as xs:string?) as empty-sequence()	yes
op:except	($parameter1 as node*, $parameter2 as node*) as node*	yes
fn:id	($srcval as xs:string*) as element*	yes
fn:idref	($srcval as xs:string*) as node*	yes
fn:input	() as node*	no
op:intersect	($parameter1 as node*, $parameter2 as node*) as node*	yes
fn:local-name	() as xs:string	yes
fn:local-name	($srcval as node?) as xs:string	yes
fn:name	() as xs:string	yes
fn:name	($srcval as node?) as xs:string	yes
op:node-after	($parameter1 as node, $parameter2 as node) as xs:boolean	yes
op:node-before	($parameter1 as node, $parameter2 as node) as xs:boolean	yes

op:is-same-node	($parameter1 as node, $parameter2 as node) as xs:boolean	yes
op:NOTATION-equal	($srcval1 as xs:NOTATION, $srcval2 as xs:NOTATION) as xs:boolean	no
fn:root	() as node	yes
fn:root	($srcval as node) as node	yes
op:union	($parameter1 as node*, $parameter2 as node*) as node*	yes
fn:lang	($testlang as xs:string) as xs:boolean	will

4.1.6 Sequence Functions

name	parameters	supported
op:concatenate	($seq1 as item*, $seq2 as item*) as item*	will
fn:distinct- values	($srcval as xs:anyAtomicType*) as xs:anyAtomicType*	yes
fn:distinct- values	($srcval as xs:anyAtomicType*, $collationLiteral as xs:string) as xs:anyAtomicType*	no
fn:empty	($srcval as item*) as xs:boolean	yes
fn:exactly-one	($srcval as item*) as item	yes
fn:exists	($srcval as item*) as xs:boolean	yes
fn:index-of	($seqParam as xs:anyAtomicType*, $srchParam as xs:anyAtomicType) as xs:integer*	will
fn:index-of	($seqParam as xs:anyAtomicType*, $srchParam as xs:anyAtomicType, $collationLiteral as xs:string) as xs:integer*	will
fn:insert- before	($target as item*, $position as xs:integer, $inserts as item*) as item*	no
fn:item-at	($seqParam as item*, $posParam as integer) as item?	will
fn:last	() as xs:integer?	yes
fn:one-or-more	($srcval as item*) as item+	yes
fn:position	() as xs:integer?	yes
fn:subsequence	($sourceSeq as item*, $startingLoc as xs:double) as item*	yes

fn:subsequence	($sourceSeq as item*, $startingLoc as xs:double, $length as xs:double) as item*	yes
fn:remove	($target as item*, $position as xs:integer) as item*	no
fn:zero-or-one	($srcval as item*) as item?	yes
fn:unordered	($sourceSeq as item*) as item*	yes

4.1.7 QName Functions

name	parameters	supported
fn:get-local-name-from-QName	($srcval as xs:QName?) as xs:string?	no
fn:get-namespace-from-QName	($srcval as xs:QName?) as xs:string?	no
fn:get-in-scope-namespaces	($element as element) as xs:string*	no
fn:expanded-QName	($paramURI as xs:string, $paramLocal as xs:string) as xs:QName	no
fn:node-name	($srcval as node?) as xs:QName?	no
op:QName-equal	($srcval1 as xs:QName, $srcval2 as xs:QName) as xs:boolean	no
fn:resolve-QName	($qname as xs:string, $element as element) as xs:QName	no

4.1.8 URI Functions

name	parameters	supported
op:anyURI-equal	($srcval1 as xs:anyURI, $srcval2 as xs:anyURI) as xs:boolean	no
fn:base-uri	($srcval as node) as xs:string?	no
fn:base-uri	() as xs:string?	no
fn:document-uri	($srcval as node) as xs:string?	no
fn:escape-uri	($uri-part as string, $escape-reserved as xs:boolean) as xs:string	no
fn:get-namespace-uri-for-prefix	($element as element, $prefix as xs:string) as xs:string?	no
fn:resolve-uri	($relative as xs:string) as xs:string	no
fn:resolve-uri	($relative as xs:string, $base as anyURI) as xs:string	no

fn:namespace-uri	() as xs:string	yes
fn:namespace-uri	($srcval as node?) as xs:string	yes

4.1.9 Runtime Functions

name	parameters	supported
fn:error	() as none	yes
fn:error	($srcval as item?) as none	yes
fn:trace	($value as item*, $label as xs:string) as item*	no
fn:codepoints-to-string	($srcval as xs:integer*) as xs:string	no
fn:string-to-codepoints	($srcval as xs:string) as xs:integer*	no

4.1.10 Date/Time Functions

name	parameters	supported
op:add-dayTimeDuration-to-date	($srcval1 as xs:date, $srcval2 as xdt:dayTimeDuration) as xs:date	no
op:add-dayTimeDuration-to-dateTime	($srcval1 as xs:dateTime, $srcval2 as xdt:dayTimeDuration) as xs:dateTime	no
op:add-dayTimeDuration-to-time	($srcval1 as xs:time, $srcval2 as xdt:dayTimeDuration) as xs:time	no
op:add-dayTimeDurations	($srcval1 as xdt:dayTimeDuration, $srcval2 as xdt:dayTimeDuration) as xdt:dayTimeDuration	no
op:add-yearMonthDuration-to-date	($srcval1 as xs:date, $srcval2 as xdt:yearMonthDuration) as xs:date	no
op:add-yearMonthDuration-to-dateTime	($srcval1 as xs:dateTime, $srcval2 as xdt:yearMonthDuration) as xs:dateTime	no
op:add-yearMonthDurations	($srcval1 as xdt:yearMonthDuration, $srcval2 as xdt:yearMonthDuration) as xdt:yearMonthDuration	no
fn:adjust-date-to-timezone	($srcval as xs:date?) as xs:date?	no
fn:adjust-date-to-timezone	($srcval as xs:date?, $timezone as xdt:dayTimeDuration?) as xs:date?	no

fn:adjust-dateTime-to-timezone	($srcval as xs:dateTime?) as xs:dateTime?	no
fn:adjust-dateTime-to-timezone	($srcval as xs:dateTime?, $timezone as xdt:dayTimeDuration?) as xs:dateTime?	no
fn:adjust-time-to-timezone	($srcval as xs:time?) as xs:dateTime?	no
fn:current-date	() as date	no
fn:current-dateTime	() as dateTime	no
fn:current-time	() as time	no
fn:adjust-time-to-timezone	($srcval as xs:time?, $timezone as xdt:dayTimeDuration?) as xs:time?	no
op:date-equal	($operand1 as xs:date, $operand2 as xs:date) as xs:boolean	no
op:date-greater-than	($operand1 as xs:date, $operand2 as xs:date) as xs:boolean	no
op:date-less-than	($operand1 as xs:date, $operand2 as xs:date) as xs:boolean	no
op:dateTime-equal	($operand1 as xs:dateTime, $operand2 as xs:dateTime) as xs:boolean	no
op:dateTime-greater-than	($operand1 as xs:dateTime, $operand2 as xs:dateTime) as xs:boolean	no
op:dateTime-less-than	($operand1 as xs:dateTime, $operand2 as xs:dateTime) as xs:boolean	no
op:dayTimeDuration-equal	($operand1 as xdt:dayTimeDuration, $operand2 as xdt:dayTimeDuration) as xs:boolean	no
op:dayTimeDuration-greater-than	($operand1 as xdt:dayTimeDuration, $operand2 as xdt:dayTimeDuration) as xs:boolean	no
op:dayTimeDuration-less-than	($operand1 as xdt:dayTimeDuration, $operand2 as xdt:dayTimeDuration) as xs:boolean	no
op:divide-dayTimeDuration	($srcval1 as xdt:dayTimeDuration, $srcval2 as xs:decimal) as xdt:dayTimeDuration	no
op:divide-yearMonthDuration	($srcval1 as xdt:yearMonthDuration, $srcval2 as xs:decimal) as xdt:yearMonthDuration	no
op:gDay-equal	($operand1 as xs:gDay, $operand2 as xs:gDay) as xs:boolean	no
fn:get-day-from-date	($srcval as xs:date?) as xs:integer?	no
fn:get-day-from-dateTime	($srcval as xs:dateTime?) as xs:integer?	no

fn:get-days-from-dayTimeDuration	($srcval as xdt:dayTimeDuration?) as xs:integer?	no
fn:get-hours-from-dateTime	($srcval as xs:dateTime?) as xs:integer?	no
fn:get-hours-from-dayTimeDuration	($srcval as xdt:dayTimeDuration?) as xs:integer?	no
fn:get-hours-from-time	($srcval as xs:time?) as xs:integer?	no
fn:get-minutes-from-dateTime	($srcval as xs:dateTime?) as xs:integer?	no
fn:get-minutes-from-dayTimeDuration	($srcval as xdt:dayTimeDuration?) as xs:integer?	no
fn:get-minutes-from-time	($srcval as xs:time?) as xs:integer?	no
fn:get-month-from-date	($srcval as xs:date?) as xs:integer?	no
fn:get-month-from-dateTime	($srcval as xs:dateTime?) as xs:integer?	no
fn:get-months-from-yearMonthDuration	($srcval as xdt:yearMonthDuration?) as xs:integer?	no
fn:get-seconds-from-dateTime	($srcval as xs:dateTime?) as xs:decimal?	no
fn:get-seconds-from-dayTimeDuration	($srcval as xdt:dayTimeDuration?) as xs:decimal?	no
fn:get-seconds-from-time	($srcval as xs:time?) as xs:decimal?	no
fn:get-timezone-from-date	($srcval as xs:date?) as xdt:dayTimeDuration?	no
fn:get-timezone-from-dateTime	($srcval as xs:dateTime?) as xdt:dayTimeDuration?	no
fn:get-timezone-from-time	($srcval as xs:time?) as xdt:dayTimeDuration?	no
fn:get-year-from-date	($srcval as xs:date?) as xs:integer?	no
fn:get-year-from-dateTime	($srcval as xs:dateTime?) as xs:integer?	no
fn:get-years-from-yearMonthDuration	($srcval as xdt:yearMonthDuration?) as xs:integer?	no
op:gMonth-equal	($operand1 as xs:gMonth, $operand2 as xs:gMonth) as xs:boolean	no
op:gMonthDay-equal	($operand1 as xs:gMonthDay, $operand2 as xs:gMonthDay) as xs:boolean	no
op:gYear-equal	($operand1 as xs:gYear, $operand2 as xs:gYear) as xs:boolean	no

op:gYearMonth-equal	($operand1 as xs:gYearMonth, $operand2 as xs:gYearMonth) as xs:boolean	no
fn:implicit-timezone	() as xs:dayTimeDuration?	no
op:multiply-dayTimeDuration	($srcval1 as xdt:dayTimeDuration, $srcval2 as xs:decimal) as xdt:dayTimeDuration	no
op:multiply-yearMonthDuration	($srcval1 as xdt:yearMonthDuration, $srcval2 as xs:decimal) as xdt:yearMonthDuration	no
op:subtract-dates	($srcval1 as xs:date, $srcval2 as xs:date) as xdt:dayTimeDuration	no
fn:subtract-dateTimes-yielding-dayTimeDuration	($srcval1 as xs:dateTime, $srcval2 as xs:dateTime) as xdt:dayTimeDuration	no
fn:subtract-dateTimes-yielding-yearMonthDuration	($srcval1 as xs:dateTime, $srcval2 as xs:dateTime) as xdt:yearMonthDuration	no
op:subtract-dayTimeDuration-from-date	($srcval1 as xs:date, $srcval2 as xdt:dayTimeDuration) as xs:date	no
op:subtract-dayTimeDuration-from-dateTime	($srcval1 as xs:dateTime, $srcval2 as xs:dayTimeDuration) as xs:dateTime	no
op:subtract-dayTimeDuration-from-time	($srcval1 as xs:time, $srcval2 as xs:dayTimeDuration) as xs:time	no
op:subtract-dayTimeDurations	($srcval1 as xdt:dayTimeDuration, $srcval2 as xdt:dayTimeDuration) as xdt:dayTimeDuration	no
op:subtract-times	($srcval1 as xs:time, $srcval2 as xs:time) as xdt:dayTimeDuration	no
op:subtract-yearMonthDuration-from-date	($srcval1 as xs:date, $srcval2 as xdt:yearMonthDuration) as xs:date	no
op:subtract-yearMonthDuration-from-dateTime	($srcval1 as xs:dateTime, $srcval2 as xdt:yearMonthDuration) as xs:dateTime	no
op:subtract-yearMonthDurations	($srcval1 as xdt:yearMonthDuration, $srcval2 as xdt:yearMonthDuration) as xdt:yearMonthDuration	no
op:time-equal	($operand1 as xs:time, $operand2 as xs:time) as xs:boolean	no
op:time-greater-than	($operand1 as xs:time, $operand2 as xs:time) as xs:boolean	no
op:time-less-than	($operand1 as xs:time, $operand2 as xs:time) as xs:boolean	no
op:yearMonthDuration-equal	($operand1 as xdt:yearMonthDuration, $operand2 as xdt:yearMonthDuration) as xs:boolean	no

op:yearMonthDuration-greater-than	($operand1 as xdt:yearMonthDuration, $operand2 as no xdt:yearMonthDuration) as xs:boolean
op:yearMonthDuration-less-than	($operand1 as xdt:yearMonthDuration, $operand2 as no xdt:yearMonthDuration) as xs:boolean

4.2 Extension Functions

MonetDB/XQuery offers various non-W3c recommended builtin functions, organized in the following categories:

4.2.1 Document Management Functions

The `pf:add-doc()` function adds a new XML document available at some URI to the database, under a logical name (second parameter). It is also possible to provide as third parameter a collection name. This makes it possible to add a document to an existing document collection. All documents in a collection store all their data together in the same MonetDB tables. Especially in cases where you have may (thousands or more) of (presumably small) XML documents, it is advisable to store these together in one or a few collections, because storing a small document in a single collection (by the same name, which is the default behavior if only two parameters are provided to `pf:add-doc()`) will cause a lot of table-header and MonetDB meta data overhead, because each single document will lead to the creation of a couple of relational tables, such that a large XML collection may cause millions of them.

Normally, collections are created read-only, meaning that updates to them are prohibited and cause runtime errors. To allow updates, documents have to be shredded explicitly as updatable, by passing a fourth parameter to `pf:add-doc()`. This parameter must have a value between 1 and 99, that indicates the percentage of unallocated space that should be left per page, to accommodate future updates. All documents inside the same collection are either all updatable, or all read-only. Note that after a collection has been created by the first `pf:add-doc()`, its status cannot be changed anymore. There is a workaround, based on the backup/restore mechanism.

`pf:add-doc`	($uri as xs:string, $name as xs:string)
`pf:add-doc`	($uri as xs:string, $name as xs:string, $coll as xs:string)
`pf:add-doc`	($uri as xs:string, $name as xs:string, $coll as xs:string, $perc as xs:integer)
`pf:del-doc`	($name as xs:string)

A query that calls any of these functions, does not return a result, highly similar to the XQuery Update Facility. However, this family of MonetDB/XQuery extension functions is not considered the same as XQUF update queries. In fact, it is specifically **forbidden** to mix XQUF updates and document management commands in the same transaction.

We should note that MonetDB/XQuery, apart from atomicity with respect to document management (i.e. a document management query either fully succeeds or fully fails), also provides durability and some form of isolation. Isolation, however is not fully perfect.

It may happen that a read-only or update query that started before a document management query committed, ends up seeing its effects. That is, if execution of this concurrent query reaches execution of `fn:doc()`, it is evaluated with respect of the actual state of the

database at that time. This is an aberration of snapshot isolation, which demands that fn:doc() be evaluated with respect to the database state at the *start* of the query.

On the other hand, once a query has gained access to a document, the query caches it in its database snapshot such that subsequent calls to `fn:doc()` will continue to find it, regardless whether it has been deleted since.

4.2.2 Metadata Functions

The below functions provide meta-data information about the XML database stored in MonetDB/XQuery. We have tried to maintain ACID properties with respect to the document management functions described above. This is achieved currently using a simple locking approach, which means that it will block on conflicting document management functions (the shredding function `pf:add-doc()`, may take considerable time on large XML instances).

If you rather trade consistency for not locking, you may use the `collections-unsafe()` and `documents-unsafe()` functions. If a concurrent document management query adds two collections to the database, the latter functions may list documents added by the first `pf:add-doc()` yet omit documents still being added by the second `pf:add-doc()`.

`pf:collection`	($name as xs:string) as xs:node
`pf:collections`	() as node()*
`pf:collections-unsafe`	() as node()*
`pf:documents`	() as node()*
`pf:documents-unsafe`	() as node()*
`pf:documents`	($name as xs:string) as xs:node
`pf:documents-unsafe`	($name as xs:string) as xs:node
`pf:docname`	($n as node() as xs:string

The pf:collections() function returns an XML node for each existing collection in the format:

```
<collection updatable="true" size="64 KiB" numDocs="1">hello.xml</collection>
```

and pf:documents() does likewise for all documents in all collections in the database:

```
<document updatable="true" url="c:\HelloWorld.xml" collection="hello.xml">hello</document>
```

The second variant of pf:documents() restricts the list to only documents from a certain collection.

The pf:docname() function is a convenience function that given a node, returns the name of the document it stems from.

4.2.3 NID Functions

MonetDB/XQuery internally assigns integer Node Identifiers (NIDs) to all XML nodes. These NIDs are tied to node identity and do not change under updates. NIDs are interesting because they provide an uniform way to identify XML nodes, with a very efficient O(1) lookup mechanism (that works in constant time).

`pf:nid`	(element()) as xs:string

The pf:nid() function returns the NID of an element. Though a NID is an integer, it is returned as an xs:string. The NID can be passed as a parameter to the built-in fn:id() function. This standard function allows to lookup nodes by their ID/IDREF values. Since ID/IDREF values cannot be numbers, the fn:id() function can recognize NIDs from normal ID/IDREF attribute values.

4.2.4 PF/Tijah Functions

The PF/Tijah project has added flexible structured XML ranking (i.e. keyword search with XPath predicates) to MonetDB/XQuery.

You can find detailed documentation on the PF/Tijah website.

`tijah:ft-index-info`	() as element*
`tijah:ft-index-info`	(string) as element*
`tijah:create-ft-index`	() as docmgmt
`tijah:create-ft-index`	(string*) as docmgmt
`tijah:create-ft-index`	(node) as docmgmt
`tijah:create-ft-index`	(string*,node) as docmgmt
`tijah:extend-ft-index`	(string*) as docmgmt
`tijah:extend-ft-index`	(string*,node) as docmgmt
`tijah:delete-ft-index`	() as docmgmt
`tijah:delete-ft-index`	(node) as docmgmt
`tijah:queryall-id`	(string) as integer
`tijah:queryall-id`	(string, node) as integer
`tijah:query-id`	(node*, string) as integer
`tijah:query-id`	(node*, string, node) as integer
`tijah:queryall`	(string) as node*
`tijah:queryall`	(string, node) as node*
`tijah:query`	(node*, string) as node*
`tijah:query`	(node*, string, node) as node*
`tijah:nodes`	(integer) as node*
`tijah:score`	(integer, node) as double
`tijah:tokenize`	(string?) as string
`tijah:resultsize`	(integer) as integer

4.2.5 Arithmetic Functions

The XQuery language lacks some basic arithmetic functions, so we added them as extensions.

`pf:log`	($v as xs:decimal) as xs:decimal
`pf:log`	($v as xs:double) as xs:double
`pf:log`	($v as xs:decimal?) as xs:decimal?
`pf:log`	($v as xs:double?) as xs:double?
`pf:sqrt`	($v as xs:decimal) as xs:decimal
`pf:sqrt`	($v as xs:double) as xs:double
`pf:sqrt`	($v as xs:decimal?) as xs:decimal?
`pf:sqrt`	($v as xs:double?) as xs:double?
`pf:pow`	($v as xs:decimal, $w as xs:decimal) as xs:decimal
`pf:pow`	($v as xs:double, $w as xs:double) as xs:double
`pf:pow`	($v as xs:decimal?, $w as xs:decimal?) as xs:decimal?
`pf:pow`	($v as xs:double?, $w as xs:double?) as xs:double?
`pf:product`	($v as xs:double*) as xs:double

4.2.6 Probabilistic XML

documentation needed!

`pxmlsup:val_except`	(xs:string*, xs:string*) as xs:string*
`pxmlsup:val_except`	(xs:integer*, xs:integer*) as xs:integer*
`pxmlsup:deep-equal`	(node(), node()) as xs:boolean
`pxmlsup:edit-distance`	(xs:string, xs:string) as xs:integer

4.3 XQuery Updates

MonetDB/XQuery supports the W3C XQuery Update Facility with the following remarks:

- there is no support for the `transform` feature!
- rename uses a `do rename .. into` syntax instead of `do rename .. as`
- currently, each updating query executes in a single transaction (a la auto-commit). MonetDB/XQuery has no support yet for multi-query transactions. Note that in the XQUF, there is no proposed syntax (start transaction, abort, commit) for this yet either.
- the XQUF specifications leave many details of the `fn:put()` function to the implementation. Below we discuss how we handled those in MonetDB/XQuery.
- August 2007, W3C published a new and final call for the XQUF that makes some changes to the syntax (e.g. `do insert` became `insert node`). These changes are not supported yet in this version of MonetDB/XQuery.

In the remainder, we provide some background information of how transactions are implemented in MonetDB/XQuery.

4.3.1 Transactions and Performance

Generally speaking, the updating mechanism in MonetDB/XQuery uses `snapshot isolation` with page-level concurrency control. We will shortly discuss how it handles ACID properties (Atomicity, Consistency, Isolation, Durability):

- **Atomicity**: a transaction commits either all its changes or none. This is implemented using a Write-Ahead-Log (WAL). The WAL contains all data modified by a transaction, and writing the commit record in the WAL is the atomic action that commits the transaction.
- **Consistency**: is ignored at this point. Note that the XQuery language does not yet define query-based constraints, therefore consistency of the data is mostly a non-issue. Of course, XML data with a declared XML Schema or DTD should continue to validate against that schema. However, at this stage MonetDB/XQuery does not support validation. One could say that as a bare minimum, the updates should produce valid XML. However, even regarding this criterion, the MonetDB/XQuery implementation of the XQUF is not safe. Known problems are: you can create XML with adjacent text nodes, duplicate attributes, and non-matching ID/IDREF data.
- **Isolation**: transactions should not observe any effect of non-committed concurrent transactions. With snapshot isolation, MonetDB/XQuery runs queries against a consistent snapshot of each XML collection at the time the query first accessed it (e.g. with fn:collection() or fn:doc()), which never changes. Note that this definition is slightly

different from running a transaction against the global database state at the query started (because there may elapse some processing time between query execution start and the opening of a document, and some queries may open multiple collections, at different times).

- **Durability**: the mentioned WAL-based commit ensures that once the commit record is written, the database recovery procedure (always performed before database startup) ensures that committed transactions are never lost.

The log directory for the WAL is called `xquery_logs` and is located in `<dbfarm>/<dbname>`.

The performance of MonetDB/XQuery transactions is not yet optimal. It is possible to do simple inserts that run in less than 50ms. For this to happen, the query part of your update must be fast (e.g. identify the data to be modified with equi-comparisons on a text() or attribute value, which uses value indices), the query must be coded in an XQuery `updating function`, so MonetDB/XQuery can cache the query plan, and it should only modify the database in one spot. A current weak-point are queries that identify many different update locations; update time tends to be super-linearly correlated with the amount of update spots.

You will find that the performance of MonetDB/XQuery tends to be dominated by CPU time, rather than I/O. Until now, under ideal circumstances, observed maximum update throughput have not surpassed 50 per second.

In future releases, we expect to improve update performance significantly.

4.3.2 Check-pointing

After an update, the changed parts of an XML document (while already present in the WAL), must also be flushed to disk at some point to reduce RAM usage. This process is called `check-pointing`, and happens in the background.

We have chosen to base document addition (`pf:add-doc()`) on check-pointing rather than on the WAL. Otherwise, shredding a large document into the database would produce huge volumes of log, causing very expensive recovery. Instead, MonetDB/XQuery directly flushes new document data to disk.

Check-pointing and WAL are two different I/O mechanisms in MonetDB, which makes it impossible to provide an atomic commit that uses both. For this reason, MonetDB/XQuery currently does not allow to mix the check-pointing-based document management functions (e.g. `pf:add-doc()`) with the WAL-based XQuery updates.

The check-pointing mechanism, while being efficient at large data volumes, carries a considerable minimum cost (latency). For this reason, shredding small documents into MonetDB/XQuery can be a bit slow; you are advised to try shredding multiple small documents in a single query to improve throughput. The Tips Section see Section 2.9.3 [Bulk Loading a Collection], page 18 explains how you can add many documents in a single query efficiently.

4.3.3 Snapshot Isolation Anomalies

You must be aware that snapshot isolation is not exactly the same as serializability. That is, while the isolation level provides repeatable reads, the `write skew` anomaly may occur. The problem stems from the fact that under snapshot isolation, two transactions are only

considered in conflict (leading to one being aborted) only if they write the same data. That is, read/write conflicts are ignored.

Write skew is a situation, where there is a constraint on two different database locations, X,Y; say X+Y <= c. It may happen that some transaction does Y=Y+1, and another does X=X+1. If we assume that before their updates X+Y=c-1, both updates seem to produce a consistent database with X+Y=c. However, after committing both, the constraint is violated; because X+Y=c+1. Because the write skew effect affects only constraint-based consistency, which is not supported anyway in XQuery there is no direct impact.

However, your application implicitly also maintains constraints (i.e. data sanity) and in this context the problem still may hit you. A workaround is to dummy-update the value of the non-modified variables tied by constraints explicitly: just assign them the value they already had (i.e. update Y=Y+1 and X=X). MonetDB/XQuery does not check whether updated values have actually changed (it always assume they have), so that will force the write/write conflict and thus consistency.

Note that major relational database systems, such as Oracle and PostgreSQL only offer snapshot isolation, and most users seem content with it (Microsoft SQLserver also offers it optionally for additional performance). The bright side of snapshot isolation is that reads do not require locks – which is why read-only query performance of MonetDB/XQuery has not suffered much when we added suppport for XQUF updates.

4.3.4 Locking and Page Fragmentation

Snapshot isolation detects concurrency conflicts in terms of transactions that update the same place in a document. This notion is determined with the granularity of a `page`. That is, documents are stored in tables, and tables are stored in pages (a page is a fixed number of XML nodes).

On Windows systems, the pageSize is 64K (*100/(100+freePerc) XML nodes), and on Linux it is 16K. The `freePerc` is the percentage free space you leave during shredding on the pages.

The pageSize can be changed with the `gdk_mem_pagebits` setting in the `MonetDB.conf` file (see Section 2.6 [MonetDB.conf], page 12). It contains the number of bits to use for page addressing (e.g. a setting of 16 leads to pages of 2**16 = 64K nodes). Reducing the page size, may reduce locking conflicts. Note however, that on Windows 64K is the minimum size (and on Linux it is 4K). This restriction comes from the use that MonetDB makes of memory mapping; XML page boundaries must therefore correspond to virtual memory page boundaries.

If you insert considerable amounts of data (more than your freePerc * pageSize) in many places of your documents, new pages are inserted. Because these pages stem from different locations on the disk, the physical representation of your document will become fragmented. If many such insertion points exist, the linear memory layout that MonetDB prefers, may change into a series of many short fragments. This affects both the speed in which such a document can be opened (memory mapped), as well as the efficiency with which the staircase join XPath operators of MonetDB/XQuery may traverse it.

Therefore, it is good practice to regularly `defragment` your updatable collections. An easy way to do this is to backup and restore your database with the Administrative GUI.

4.3.5 The put() Function

The `fn:put()` function stores a node as a new document at some URI. MonetDB/XQuery only supports `file://` URIs here (though the `file://` prefix itself may be omitted, since a filename is assumed as default) with the security restriction that the file path must be `local`. That is, it may not start with a slash, or drive letter, nor may it contain backwards paths (..). This means that the XML files produced by `fn:put()` will be located in the `<gdk_dbfarm>/<dbname>/` directory, or one of its sub-directories. If the filepath provided as a parameter to `fn:put()` contains a sub-directory path, this sub-directory is created automatically.

The restriction on local filepaths is a securiy feature: otherwise it would be possible to overwrite any writable file in the server filesystem using XQuery queries that contain `fn:put()`.

One of the interesting uses of `fn:put()` is to `cache` intermediate results produced by a costly query. A handly place to put these is the local `tmp/` directory, as MonetDB/XQuery automatically removes all temporary documents that are older than an hour (see Section 5.10 [Temporary Documents], page 64). This way, your application does not need to worry about garbage collection.

5 XQuery Extensions

The database research efforts that drive MonetDB/XQuery have led us to include a number of non-standard XQuery features in the system, that may be useful to you.

5.1 Document Management

The contents of the database can be inspected with the extension function `pf:collections()`; it returns all XML collections. For each collection, or for the entire database, a list of all XML documents contained in it can be obtained with `pf:documents()` (see see Section 4.2.2 [Metadata Functions], page 39).

Similarly, the `pf:add-doc()` and `pf:del-doc()` functions allow for adding documents to and deleting documents from an XML database using an XQuery query (see Section 4.2.1 [Document Management Functions], page 38). Alternatively, you can also add and delete documents with the Administrative GUI (see Section 3.2 [The Administrative GUI], page 26). It provides a simple, yet effective GUI to view all documents in a collection, providing buttons to add and delete documents.

Any query containing an `pf:add-doc()` or `pf:del-doc()` is called a **document management query**. A document management query does not return any value, highly similar to the XQuery Update Facility. However, this family of MonetDB/XQuery extension functions is not considered the same as XQUF update queries. In fact, it is specifically **forbidden** to mix XQUF updates and document management commands in the same transaction.

This has a technical reason: for MonetDB/XQuery it is difficult to provide atomicity based on two quite different principles, namely write ahead logging (used for updates) and check-pointing (used for document management). This design decision was made to allow efficient bulk import. This means that an `pf:add-doc()` call directly creates new table images on disk (check-pointing), rather than writing the added XML document(s) first to a log.

Similar to the XQUF syntax of `updating function`, used to declare a function that performs updates and hence has no return value; we introduce the syntax of `document management function`, used to declare a user-defined function that performs an `pf:add-doc()` or `pf:del-doc()`.

```
declare document management function addFoo($name as xs:string)
{ pf:add-doc($name,$name) };

addFoo("http://monetdb.cwi.nl/XQuery/files/bib.xml")
```

Note that the above function declaration uses the special keywords `document management` and does not specify a return type.

5.2 PF/Tijah Text Indexing

The database group of University Twente has created the Tijah XML retrieval system. In PF/Tijah, it has become fully integrated and distributed with MonetDB/XQuery.

You can find documentation at the PF/Tijah website. This site also provides information on how to create text indices on XML documents stored in MonetDB/XQuery and how to perform keyword search on them.

NOTE: it is currently not possible to update XML documents that have a text index on them. It is possible though, to add and delete entire XML documents from text-indexed document collections. If you have an application that queries a large XML document collection, while at the same time new documents are added frequently (or deleted), it is still a good idea to store your data in an updatable collection (even while it is not allowed to actually use in XQuery updates). The reason is that **index maintenance** is cheaper on updatable collections than on read-only collections (see Section 2.9 [Performance Tips], page 15).

PF/Tijah is highly configurable in terms of the ranking functions supported. *Additional info:* the OSIR paper provides a quick technical background on PF/Tijah.

5.3 Session Expression Cache

Expression Caching is a powerful feature to create well-performing functionally rich applications. You can use it for query result caching (avoiding to compute the same query twice), as a mechanism to simulate query result cursors; allowing an expensive query that delivers large result to be evaluated once, allowing subsequent queries to show small parts of a result set, that e.g. fit on the screen.

The mechanism allows Caching of Arbitrary Subexpressions inside a so-called Multi-Query Session.

5.3.1 Multi-Query Sessions

MonetDB/XQuery allows you to interact with the database server using a single session in which you see the same snapshot of the database all the time. That is, a multi-query session that may last for a long time.

You get such a session by prefixing queries using XQuery options:

```
declare option pf:session-id "ID";
declare option pf:session-timeout "MSECS";

QUERY
```

Here you should substitute `QUERY` by your query, `ID` by an identifier (letters, numbers, underscore, -), and `MSECS` by an integer number that indicates a duration in milliseconds.

XQuery options are part of the XQuery standard and systems implementing it are free to define their meaning. Non-meaningful options are simply ignored, such that adding such options will not affect the interoperability of your queries.

The function of the `pf:session-*` options is that all queries that are wrapped as such with the same `ID` use the same database snapshot.

An example query is one that display male persons:

```
doc("auctions.xml)//person[gender = "male"]/name
```

which could be wrapped in the `pf:session` pragma as follows:

```
declare option pf:session-id "my-own-id";
declare option pf:session-timeout "10000";

doc("auctions.xml)//person[gender = "male"] }
```

this says that the session is called `my-own-id` (a name the user is free to make up), and that it should be kept alive for 10 seconds (10000ms). After 10 seconds of inactivity in the session, the session is silently terminated, which means that the database snapshot is released at the server.

5.3.2 Caching of Arbitrary Subexpressions

Now consider in our example that we have a web interface that displays a table of person names. However, only 20 names fit on a screen, and the application provides a scroll bar and "next" and "previous" buttons to navigate through the list of persons. Each time the user clicks on those buttons, a new query will be executed like:

```
subsequence(doc("auctions.xml)//person[gender = "male"]/name, LO, HI)
```

with different values for `LO` and `HI`. This means that the entire query gets re-evaluated, which may take a long time, resulting in a poor user experience.

The sub-expression caching infrastructure allows users to mark up any subexpression for caching, using a pragma:

```
(# pf:cache EXPRID #) { EXPR }
```

Pragma's `(# xx #)` are not an extension themselves, they are part of the XQuery standard and are normally ignored, semantically they do not change the query, so the presence of pragmas does not affect the interoperability of your queries.

Again, `EXPRID` is an identifier made up by the user and `EXPR` can be anything. It should be noted, however, that the `pf:cache` pragma cannot be used inside `for`-loops.

For example, we could rewrite our previous example query into this one, which displays the first 10 male persons:

```
declare option pf:session-id "my-own-id";
declare option pf:session-timeout "30000";

subsequence((# pf:cache my-male-persons #) { doc("auctions.xml)//person[gender = "male"] },
```

which says that within session `my-own-id`, the subquery for male persons should be cached under name `my-male-persons`. This has as effect that on the first time this query is executed in the session, the result of the expression is cached inside the session. Any subsequent request enclosed by a `pf:cache` pragma with the `my-male-persons` identifier will not take computational effort, as the result is already cached.

For example, if a user hits the "next" button, the next 10 male persons can be produced in no time as follows:

```
declare option pf:session-id "my-own-id";
declare option pf:session-timeout "30000";

subsequence((# pf:cache my-male-persons #) { doc("auctions.xml)//person[gender = "male"] },
```

A side effect of the query with caching pragmas is that the session timeout is set to the current time plus the timeout (here `30000`, hence 30 seconds). In other words, each query that uses a cached session causes that session to be kept alive for the amount of time it specifies.

Note that one can terminate a session by sending a (dummy) query with a timeout of 0.

5.3.3 Consistency

We should note that the current implementation of Session Subexpression Caching in MonetDB/XQuery is rather simple, as it requires the user to annotate the interesting subexpressions with pragmas (rather than doing this automatically).

Even more, it is the responsibility of the user to be consistent in the use of `pf:cache` pragma identifiers: if the same identifier is used in the same session for different subexpressions, incorrect results will be returned (MonetDB/XQuery does not test itself that the subexpression syntax which produced a cached result is identical to the syntax given in the prior query that computed the sub-result).

5.3.4 Concurrent Access to a Session

The session reuse mechanism in MonetDB/XQuery will cache sessions, yet allows only a single query to access it at the same time (it locks the session). This limits parallelism on multi-core machines. For this reason, we support the option `pf:session-nocache`:

```
declare option pf:session-id "ID";
declare option pf:session-timeout "MSECS";
declare option pf:session-nocache "true";

QUERY
```

The idea behind a `session-nocache` session is that it only re-uses a session (with potentially pre-created cached results attached to it), but it is not allowed to store any new cached subexpression values. While this means that particular activities of this queries will not be available for re-use in subsequent queries, the fact that the state of the session is left unchanged means that multiple of such `nocache` session queries (in the same session!) can run in parallel on a multi-core server.

Thus, queries whose results are likely not to be reused, but whose computation relies on precomputed expressions, are a target for running with `session-use`, with the benefit that increased parallel performance can be obtained. This is only relevant if you have multiple queries that could be executed concurrently.

This option is also useful for avoiding to pollute the cached session with many constructed nodes, if you query constructs many nodes, as explained below.

5.3.5 Memory Consumption

A final issue is the size of the cache. For each session, a default limit of 128MB of results is maintained. This quantity can be changed by modifying the value of the `xquery_procMB` MonetDB environment variable, followed by a server restart.

The item sequences in the session cache are management automatically by the system using an LRU scheme.

Special attention should be paid to caching subexpressions that perform node construction. The MonetDB/XQuery of node construction causes temporary tables to be populated with tuples that represent the new nodes. Therefore, such queries cause extra memory consumption (in addition to the XML document in the database that remains open, and the cached sequence of items, there is extra data being kept that represents the new nodes).

To avoid polluting memory with many constructed nodes, you should consider using `pf:session-nocache` to avoid caching them. Of course, if the constructed nodes are what you want to cache, you should do so, but beware of the size.

The complication with constructed node space is that the system cannot garbage collect it, hence this memory space only grows. There is a hard limit imposed on the amount of constructed nodes (1M), after which the session gets terminated! This draconian measure is currently the only way to keep resource consumption under control.

5.3.6 Updates

Due to the snapshot semantics, users will see the same database state throughout the entire session.

Updating queries are allowed in a session, however these always trigger the termination of the session.

5.4 HTTP Access

MonetDB/XQuery comes with a built-in HTTP server (see Section 5.5 [XRPC Extension], page 49), that serves out the directory `<datadir>/MonetDB/xrpc/`. Here `datadir>` is defined in the `MonetDB.conf` configuration file (see see Section 2.6 [MonetDB.conf], page 12). This allows you to build a simple website right on top of MonetDB/XQuery.

An important feature of the HTTP server is to serve out all documents in the database. Any document `FOOBAR` stored in your database, can be accessed on the URL:

```
http://<machine>:<xrpc_port>/xrpc/doc/FOOBAR
```

where `<machine>` should be substituted by the hostname or IP address of the machine where MonetDB/XQuery runs, and `<xrpc_port>` is the TCP/IP port (see Section 2.6 [MonetDB.conf], page 12).

Inside XQuery queries you can also use the synonym URI:

```
xrpc://<machine>/doc/FOOBAR
```

The use of the `xrpc://` URI naming scheme tells MonetDB/XQuery that the remote hosts implements XRPC (see Section 5.5 [XRPC Extension], page 49), which may in the future enable distributed query optimizations.

5.5 XRPC Extension

XRPC is a simple XQuery extension that allows *efficient* and *interoperable* distributed queries. You can use it to

- query MonetDB/XQuery from any application (or web page) using SOAP requests,
- pose queries that involve multiple MonetDB/XQuery servers, or
- pose queries that even involve *other XQuery engines* such as Galax and Saxon.

The latter option, querying other engines, is made possible by the *XRPC Wrapper*, a simple HTTP request server that is distributed with MonetDB/XQuery.

XRPC is the result of research of the CWI group in distributed and P2P XQuery evaluation. Background information can be found in this VLDB paper.

WARNING: XRPC still makes use of the old compiler backend and does not use the optimizing algebraic query compiler. Therefore, its query performance can sometimes be inferior to other queries handled by MonetDB/XQuery. Also, as the old compiler backend is gradually phased out, it gets to be less well-maintained and tested in general. Use with caution. We hope to port XRPC to the algebra backend soon.

The XQuery 1.0 language only provides a *data shipping* model for querying XML documents distributed on the Internet. The built-in function `fn:doc()` fetches an XML document from a remote peer to the local server, where it subsequently can be queried. The recently published W3C working draft of XQuery Update Facility (XQUF) introduces the built-in function `fn:put()` for remote storage of an updated document, which again implies data shipping.

To equip XQuery with *function shipping* style distributed querying abilities, we introduce **XRPC**. XRPC is a minimal yet powerful XQuery extension that enables efficient distributed querying of heterogeneous XQuery data sources. XRPC enhances the existing concept of XQuery functions with the Remote Procedure Call (RPC) paradigm. By calling out of an XQuery `for`-loop to multiple destinations, and by calling functions that themselves perform XRPC calls, complex P2P communication patterns can be achieved.

The XRPC extension is orthogonal to all XQuery features, including XQUF. Hence, in all places where a function application is allowed by the XQuery 1.0 language, an XRPC function call can be placed. All functions defined in an XQuery module can be called remotely, provided that both the caller and the callee of the function have access to the same module definition file. All XQUF updating expressions[1] can be included in the definition of an updating XQuery module function, which then can be called with XRPC.

The XRPC extension is enabled by default in MonetDB/XQuery. It is compiled together with the `pathfinder` module. XRPC has two major components, a request handler (module `xrpc_server`) and a message sender (module `xrpc_client`). Both module are loaded when the module `pathfinder` is loaded in `Mserver`.

5.5.1 XRPC Syntax

Remote function applications take the XQuery syntax:

`execute at { `*Expr*` } { `*FunApp(ParamList)*` }`

where *Expr* is an XQuery expression that specified the URI of the peer on which the function *FunApp* is to be executed.

For a precise syntax definition, we show the rules of the XQuery 1.0 grammar that were changed:

```
PrimaryExpr  ::= ... | FunctionCall | XRPCCall | ...
XRPCCall     ::= "execute at" "{" ExprSing "}" "{" FunctionCall "}"
FunctionCall ::= QName "(" (ExprSingle("," ExprSingle)*)? ")"
```

We restrict the function application *FunApp* to user-defined functions that are defined in a module. Thus, the defining parameters of an XRPC call are: *(i)* a module URI, *(ii)* a function name, and *(iii)* the actual parameters (passed by value). The module URI is the one bound to the namespace identifier in the function application. The module URI *must* be supplemented by a so-called *at*-hint, which also is a URI.

[1] The `transform` expression is not supported yet.

The current choice to allow functions defined in XQuery modules is due to efficiency and security reasons. MonetDB/XQuery has the mechanism of caching the query plan of a module. For all subsequent use of the functions in a cached module, only the function parameters need to be extracted to be feed directly into the query plan. For security reason, by allowing only modules, it is trivial to specify which modules are allowed to be executed or not.

It is **important** to know that actual parameters of the called function are passed **by value** (in contrary to *by reference*), which implies that if an XML node is passed as a parameter of an XRPC call, only its sub-tree is serialized in the request message and sent to the remote site. At the other side, the node will have a different identity; namely the one tied to the XRPC SOAP message (which also can be considered a document).

5.5.2 XRPC Examples

As a running example, we assume a set of XQuery database systems (peers) that each store a film database document "*filmDB.xml*" with contents similar to:

```
<films>
  <film>
    <name>The Rock</name>
    <actor>Sean Connery</actor>
  </film>
  <film>
    <name>Goldfinger</name>
    <actor>Sean Connery</actor>
  </film>
  <film>
    <name>Green Card</name>
    <actor>Gerard Depardieu</actor>
  </film>
</films>
```

We assume an XQuery module "*film.xq*" stored at "*example.org*", that defines a function `filmsByActor()`:

```
module namespace film="films";

declare function film:filmsByActor($actor as xs:string) as node()*
{ doc("filmDB.xml")//name[../actor=$actor] };
```

We can execute this function on remote peer "*x.example.org*" to get a sequence of films in which Sean Connery plays in the remote movie database:

```
import module namespace f="films" at "http://example.org/film.xq";

<films> {                                                          (Q1)
  execute at {"x.example.org"} {f:filmsByActor("Sean Connery")}
} </films>
```

Above example yields (white spaces have been added for readability):

```
<films>
  <name>The Rock</name>
```

```
  <name>Goldfinger</name>
</films>
```

All functions defined in an XQuery module can be called remotely, *provided* that both the XRPC client and the XPRC server can access the *same* module definition file. **Beware** that the XRPC server does not check if it is indeed accessing the *same* module definition file as meant by the caller. Hence, if the XRPC client uses a local file, and the XRPC server happens to have a file on the server's local system with the same name but different contents, the query can produce unexpected results. It is up to the query writer to prevent this problem from happening.

5.5.2.1 More Examples

A more elaborate example demonstrates the possibility of multiple remote function calls to a peer:

```
import module namespace f="films" at "http://example.org/film.xq";

<films> {
  for $actor in ("Julie Andrews", "Sean Connery")                    (Q2)
  return
    execute at {"x.example.org"} {f:filmsByActor($actor)}
} </films>
```

To make it a bit more complex, we could do multiple function calls to multiple remote peers:

```
import module namespace f="films" at "http://example.org/film.xq";

<films> {
  for $actor in ("Julie Andrews", "Sean Connery")                    (Q3)
  for $dst in ("x.example.org", "y.example.org")
  return execute at {$dst} {f:filmsByActor($actor)}
} </films>
```

Complex communication patterns may be programmed with XRPC, especially if recursive functions are used:

```
module namespace film="filmdb";

declare function
film:recursiveActor($dsts as xs:string*, $actor as xs:string) as node()*
{
    let $cnt := fn:count($destinations)
    let $pos := ($cnt / 2) cast as xs:integer
    let $dsts1 := fn:subsequence($destinations, 1, $pos)
    let $dsts2 := fn:subsequence($destinations, $pos+1)
    let $peer1 := $destinations[1]                                   (Q4)
    let $peer2 := $destinations[$pos]
    return (
        if ($cnt > 1) then
            execute at {$peer1} {film:recursiveActor($dsts1, $actor)}
```

```
        else (),
        doc("filmDB.xml")//name[../actor=$actor],
        if ($cnt > 2) then
            execute at {$peer2} {film:recursiveActor($dsts2, $actor)}
        else ())
};
```

The above function executes the function `recursiveActor` on a set of destination peers, uniting all results, and does so by constructing an binary spanning tree of recursive XRPC calls.

5.5.3 XRPC Server

On database startup, the the HTTP server built-in for XRPC is started up automatically:

```
$ Mserver --dbinit="module(pathfinder);"
...
# XRPC administrative GUI at http://localhost:50001/admin
MonetDB>
```

The Administrative GUI (see Section 3.2 [The Administrative GUI], page 26) is a pure-HTML application that allows to administer MonetDB/XQuery. It is built on top of XRPC and the HTTP port number used is visible in the start-up message above.

By default, the XRPC server listens to the port number `(mapi_port + 1)`. As the default `mapi_port` number is `50000`, the default `xrpc_port` number is thus `50001` (please note the port number of the XRPC administrative GUI above).

The `xrpc_port` variable can be set by editing the `MonetDB.conf` file (see Section 2.6 [MonetDB.conf], page 12) file, or at `Mserver` startup using the `--set var=value` command line switch.

Changing `mapi_port` will change de XRPC server port as well. The `xrpc_port` can be also set explicitly and this will not affect the value of `mapi_port`.

MonetDB/XQuery provides a standard place for putting XQuery Modules that are callable from outside: `<datadir>/MonetDB/xrpc/export`. Here `datadir>` is defined in the `MonetDB.conf` configuration file (see see Section 2.6 [MonetDB.conf], page 12).

Note that `<datadir>/MonetDB/xrpc/` is the root of the directory served out by the HTTP server, so all XQuery module files in the `export/` directory are served out as well (and can be referred to as URIs in XQuery qeries).

Trusted Modules.

For security reasons, the XRPC server will not execute an arbitrary module, instead, it will only execute those modules which location (given by the `at`-hint) has the same prefix as one of the values listed in the MIL variable `xrpc_trusted`. This `xrpc_trusted` variable contains semi-colon separated list of URI prefixes that are to be trusted, and can be set by by editing the `MonetDB.conf` file (see Section 2.6 [MonetDB.conf], page 12).

By default, only modules stored in the MonetDB domain and in the `export` directory can be called. Calls to functions in un-trusted modules will be rejected with an HTTP response code `403`.

Setting the value of `xrpc_trsuted` to be empty means that all module URIs will be trusted.

File Serving. The XRPC server is a simple HTTP server as well. It serves all files stored in:

`${prefix}`/share/MonetDB/xrpc

For example, the dummy XQuery module "*export.xq*" that is standard installed in:

`${prefix}`/share/MonetDB/xrpc/export

can be retrieved using the URL:

http://`<yourhost>`:`<xrpc_port>`/export/export.xq

Directory listing is turned off and it can only be turned on by changing the XRPC source code.

5.5.4 SOAP Message Format

The design goal of XRPC is to create a distributed XQuery mechanism with which different XQuery processors at different sites can jointly execute queries. This implies that our XRPC extension also encompasses a *network protocol*.

Network communicating in XRPC uses the Simple Object Access Protocol (SOAP), i.e. XML messages over HTTP. The SOAP XRPC message format is defined in **XRPC.xsd**. According to the classification in the article "Discover SOAP encoding's impact on Web service performance", the SOAP XRPC protocol belongs to the family of "*document/literal*". Note that SOAP XRPC should not be confused with SOAP RPC, a sub-protocol defined by the SOAP 1.2 standard[2].

XRPC Request Message. SOAP messages consist of an envelope, with an optional `Header` element and a `Body` element. Inside the body, we define a `request` element with several attributes:

- required attributes
  - `module`: namespace URI of the XQuery module (NB: do not use the user defined prefix for the module!)
  - `method`: name of the called function
  - `arity`: the number of parameters the called method has
  - `location`: the *at*-hint, i.e. the location where the module file is stored.
- optional attributes
  - `iter-cnt`: number of iterations included in this request
  - `updCall`: is the called function an updating function (as defined by XQUF) or not. Note that the pathfinder document management functions (e.g. `pf:add-doc()`) are also considered to be updating functions by XRPC.

[2] SOAP RPC is oriented towards binding with programming languages such as C++ and Java, and specifies parameter marshaling of a certain number of simple (atomic) data type. However, its supported atomic data types do not match directly those of the XQuery Data Model (XDM), and the support for arrays and structs is not relevant in XRPC, where there rather is a need for supporting arbitrary-shaped XML nodes as parameters as well as sequences of heterogeneously typed items. This is the reason why SOAP XRPC message format, while supporting the general SOAP standard over HTTP with the purpose of RPC, implements a new parameter passing sub-format, hence *SOAP XRPC != SOAP RPC*.

The actual parameter values of a single function call are enclosed by a `call` element. Each individual parameter consists of a `sequence` element, that contains zero or more values.

Below we show the SOAP XRPC request message generated for the first example query (Q2) that looks for films played by Sean Connery:

```
<?xml version="1.0" encoding="utf-8"?>
<env:Envelope
    xmlns:xrpc="http://monetdb.cwi.nl/XQuery"
    xmlns:env="http://www.w3.org/2003/05/soap-envelope"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://monetdb.cwi.nl/XQuery http://monetdb.cwi.nl/XQuery/XRPC.xsd"
  <env:Body>
    <xrpc:request module="filmdb"
                  method="filmsByActor"
                  arity="1"
                  location="http://example.org/film.xq"
                  iter-cnt="1"
                  updCall="false">
      <xrpc:call>
        <xrpc:sequence>
          <xrpc:atomic-value xsi:type="xs:string">Sean Connery</xrpc:atomic-value>
        </xrpc:sequence>
      </xrpc:call>
    </xrpc:request>
  </env:Body>
</env:Envelope>
```

- *Atomic values* are represented with `atomic-value`, and are annotated with their (simple) XML Schema Type in the `xsi:type` attribute. Thus, the heterogeneously typed sequence consisting on a string `"abc"` and a double `3.1` would become:

  ```
  <xrpc:sequence>
    <xrpc:atomic-value xsi:type="xs:string">abc</xrpc:atomic-value>
    <xrpc:atomic-value xsi:type="xs:double">3.1</xrpc:atomic-value>
  </xrpc:sequence>
  ```

- *XML nodes* are passed by value, enclosed by an `element` element:

  ```
  <xrpc:sequence>
    <xrpc:element>
      <filmName>The Rock</filmName>
    </xrpc:element>
    <xrpc:element>
      <filmName>Goldfinger</filmName>
    </xrpc:element>
  </xrpc:sequence>
  ```

  Similarly, the XML Schema XRPC.xsd defines enclosing elements for document, attribute, text, processing instruction, and comment nodes. Document nodes are repre-

sented in the SOAP message as a `document` element that contains the serialized document root. Text, comment and processing instruction nodes are serialized textually inside the respective elements `text`, `comment` and `processing-instruction`. Attribute nodes are serialized *inside* the `attribute` element: `<xrpc:attribute x="y">`.

- *User-defined types:* XRPC fully supports the XQuery Data Model, a requirement for making it an orthogonal language feature. This implies XRPC also supports passing of values of user-defined XML Schema types, including the ability to validate SOAP messages. XQuery already allows importing XML Schema files that contain such definitions. Values of user-defined types are enclosed in SOAP messages by `element` elements, with a `xsi:type` attribute annotating their type. The XQuery system implementing XRPC should include a `xmlns` namespace definition as well as a `xsi:schemaLocation` declaration inside the `Envelope` element when values of such imported element types occur in the SOAP message.
- *Multi-parameter functions:* for functions with more than one parameters, the value of each parameter is enclosed in a separate `sequence` element. For example, to call the function

  ```
  declare function add ($v1 as xs:integer, $v2 as xs:integer) as xs:integer
  ```

  with the parameters 10 and 20, the values are serialized as the following:

  ```
  <xrpc:sequence>
    <xrpc:atomic-value xsi:type="xs:integer">10</xrpc:atomic-value>
  <xrpc:sequence>
  <xrpc:sequence>
    <xrpc:atomic-value xsi:type="xs:integer">20</xrpc:atomic-value>
  <xrpc:sequence>
  ```

- *Loop-lifting:* on of the main feature of the SOAP XRPC protocol is the support for loop-lifting, that is, *all* iterations in a `for`-loop that containing the applications of the *same* function (but usually with different parameter values) on the *same* remote peer, are serialized in *one* XRPC request message. The parameter values of each iteration is enclosed in a separate `call` element. The execution results of all those iterations will also be serialized into *one* XRPC response message. For example, the example query (Q2) above contains two iterations that call the same function on the same remote peer. For this query, the following request message (only the main part is shown) will be generated:

  ```
  <xrpc:request module="filmdb"
                method="filmsByActor"
                arity="1"
                location="http://example.org/film.xq"
                iter-cnt="2"
                updCall="false">
    <xrpc:call>
      <xrpc:sequence>
        <xrpc:atomic-value xsi:type="xs:string">Julie Andrews</xrpc:atomic-value>
      </xrpc:sequence>
    </xrpc:call>
    <xrpc:call>
  ```

```
        <xrpc:sequence>
          <xrpc:atomic-value xsi:type="xs:string">Sean Connery</xrpc:atomic-value>
        </xrpc:sequence>
      </xrpc:call>
    </xrpc:request>
```

XRPC Response Messages follow the same principles, e.g.:

```
<?xml version="1.0" encoding="utf-8"?>
<env:Envelope
  xmlns:xrpc="http://monetdb.cwi.nl/XQuery"
  xmlns:env="http://www.w3.org/2003/05/soap-envelope"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://monetdb.cwi.nl/XQuery
                      http://monetdb.cwi.nl/XQuery/XRPC.xsd">
  <env:Body>
    <xrpc:response module="filmdb" method="filmsByActor">
      <xrpc:sequence>
        <xrpc:element><filmName>The Rock</filmName></xrpc:element>
        <xrpc:element><filmName>Goldfinger</filmName></xrpc:element>
      </xrpc:sequence>
    </xrpc:response>
  </env:Body>
</env:Envelope>
```

Inside the body is now a `xrpc:response` element that contains the result `sequence` of the remote function call.

XRPC Error Message. Whenever an XRPC server discovers an error during the processing of an XRPC request, it immediately stops execution and sends back an XRPC error message, using the format of the SOAP Fault message (see SOAP Version 1.2 Part 0: Primer and Part 1: Messaging Framework). For example, the following SOAP Fault message indicates that a required module could not be loaded:

```
<?xml version="1.0" encoding="utf-8"?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope"
              xmlns:xml="http://www.w3.org/XML/1998/namespace">
  <env:Body>
    <env:Fault>
      <env:Code>
        <env:Value>env:Receiver</env:Value>
      </env:Code>
      <env:Reason>
        <env:Text xml:lang="en">could not load module!</env:Text>
      </env:Reason>
    </env:Fault>
  </env:Body>
</env:Envelope>
```

5.5.5 XRPC Wrapper

WARNING: XRPC still makes use of the old compiler backend and does not use the optimizing algebraic query compiler. Therefore, its query performance can sometimes be inferior to other queries handled by MonetDB/XQuery. Also, as the old compiler backend is gradually phased out, it gets to be less well-maintained and tested in general. Use with caution. We hope to port XRPC to the algebra backend soon.

The XRPC Wrapper quickly enables third party XQuery engines to participate in distributed XQuery querying using XRPC, without the engine having integrated support for XRPC.

The XRPC Wrapper is a SOAP service handler that listens to HTTP connections to receive and handle incoming SOAP XRPC requests.

The XRPC Wrapper consists of two JAR packages in `$prefix/share/MonetDB/lib`: the XRPC Wrapper itself (`xrpcwrapper.jar`) and an XRPC Java client (`xrpcwrapper-test.jar`) that can be used to test the XRPC Wrapper automatically.

Using the XRPC Wrapper. A synopsis of the calling arguments is given below:

```
java -jar $prefix/share/MonetDB/lib/xrpcwrapper.jar -c <command \
          [-d debug] [-h help] [-p port] [-q quiet] [-r rootdir] \
          [-R --remove] [-v version]
```

The `-c <command>` option is a mandatory option, because the XRPC Wrapper needs to know how to start the XQuery engine. The `command` string *must* contain enough information for the XRPC Wrapper to be able to execute the engine by just passing the `command` string to Java's `Runtime.getRuntime().exe()`, hence, it is recommended to include the complete path of the binary in the `command` string.

For example, if the Saxon XQuery processor should be use, the `-c` option would look like the following:

```
--command "java -cp <pathto>/saxon8.jar net.sf.saxon.Query"
```

The following is the detailed information of all options:

`-c --command`
: This option is MANDATORY! This option specifies the command for executing the XQuery engine and all options that should be passed to the XQuery engine. The command and all options MUST be specified in ONE string.

`-d --debug`
: Turn on the DEBUG mode to get more information (e.g. the messages exchanged) printed.

`-h --help` Print this help message.

`-p --port` The port number to which the XRPC wrapper listens to (default: 50002)

`-q --quiet`
: Suppress printing the welcome header.

`-r --rootdir`
: The root directory to store temporary files (default: `System.getProperty("java.io.tmpdir")`).

`-R --remove`
: Remove the temporary files (`<request | query | all>`) that contain the XRPC request message (`--remove request`) and/or the generated XQuery query (`--remove query`, or `--remove all`) after a request has been handled.

`-v --version`
: Print version number and exit.

Using the Test Client. A synopsis of the calling arguments is given below:

```
java -jar $prefix/share/MonetDB/lib/xrpcwrapper-test.jar \
          [-f function] [-i iterations] [-k keep] [-l location] \
          [-h help] [-r rootdir] [-s host[:port]] [-v verbose]
```

The following is the detailed information of all options:

`-f --function`
: This option is MANDATORY! This option specifies which one of the XQuery functions declared in the test module file `xrpcwrapper_testfunctions.xq` should be called. Currently, the following functions are declared: `echoVoid`, `echoInteger`, `echoDouble`, `echoString`, `echoParam`, `getPerson`, `getDoc`, `firstClosedAuction`, `buyerAndAuction`, `auctionOfBuyer`. With the special option `--function all`, all functions in `xrpcwrapper_testfunctions.xq` will be called, one at a time.

`-i --iterations`
: Number of iterations the function should be called (default: `1`).

`-l --location`
: Location where the XQuery test module file is stored. This option can be used in case the XRPC Wrapper is running on a remote host. (default: `System.getProperty("java.io.tmpdir")/xrpcwrapper_testfunctions.xq`).

`-s --server`
: The host URL (<host>[:port]) of the XRPC handler (default: `http://localhost:50002`).

`-r --rootdir`
: The root directory to store temporary files. (default: `System.getProperty("java.io.tmpdir")`).

`-k --keep` Do not remove the temporary files that were generated by the test client before exit.

`-v --verbose`
: Print additional information, such as the XRPC request/response message.

`-h --help` Print this help message.

Known problem: differences among XQuery engines

The XRPC Wrapper has been tested with the Saxon (Saxon-B 8.9) and the Galax (v0.7.2) XQuery processors. Due to the varieties in the XQuery implementations (e.g. partial support of the XQuery types, element construction expressions), it is possible that the generated queries can not be handled by another XQuery engine. In such case, changes in

the sources are necessary. One might first try to edit the XQuery module file `wrapper_functions.xq` in `xrpcwrapper.jar` in such a way that only those XQuery language features, which are supported by the XQuery engine, are used in the function definitions. If this does not solve all problems, the source code of the function `generateQuery` in `XRPCWrapperWorker.java` needs to be changed as well.

5.6 Transitive Closure Extension

There is some interest in the research community for a dedicated *transitive closure* operator (contrasted to XQuery's means to provide for recursion, user-defined functions). MonetDB/XQuery, hence, provides the syntax extension

```
"with" $variable ["as" Type] "seeded by" SeedExpr "recurse" Expr
```

The semantics of this expression is

1. Evaluate `SeedExpr`. It serves as a seed to the recursion process and is bound to variable `$variable` in the first recursion step.
2. For each recursion step, evaluate the expression's body `Expr`. This body may refer to variable `$variable`, which is bound to the outcome of the previous recursion step (or to the seed expression if we are in the first step).
3. All evaluations of the body are collected by means of the XQuery `union` operator to form the expression result. Recursion stops as soon as we reach a fix point.

A few remarks:

- An optional type declaration may be used to restrict the type of the recursion variable. If it is omitted, `Type` defaults to `node*`. In any case, the static types of both expressions, `seedExpr` and `Expr` must be subtypes of `Type`.
- XQuery's `union` operator is only defined on nodes. Hence, Type must be a subtype of `node*`. There are some more restrictions on `Type` to make the entire expression sensible (e.g., its quantifier must be greater than 1).
- It is possible to write recursive expressions that do not reach a fix point. Evaluation won't terminate in that case.

The transitive closure operator is only supported when using Pathfinder's algebraic back-end. On the other hand, the algebraic back-end does not support any other means of recursion yet.

5.7 StandOff Extension

XML is often used to store annotations (i.e. meta-data, data describing other data). In particular, XML **StandOff Annotation**, concern annotations that annotate some object that itself is **not** included in the XML document. Such StandOff annotation often refers to *regions* in this object. We support a form of XML annotations that denotes these regions as XML node attributes called `start` and `end`.

Consider, for example, a video file (documentary) annotated as follows:

```
<sample>
 <video>
  <scene id="Intro" start="0" end="800"/>
  <scene id="Interview" start="801" end="10400"/>
```

```
  <scene id="Outro" start="10401" end="13400"/>
 </video>
 <music>
  <song artist="Beatles" start="0" end="4500"/>
  <song artist="Bach" start="10000" end="13000"/>
 </music>
</sample>
```

On the above example XML file (multimedia case), one may want to ask which music was played during the interview. In that case, we want `song` elements whose regions *overlap* with the Interview `shot`. Without StandOff extensions, such queries are tedious to express in XQuery, and perform very slowly.

with StandOff extensions, the query can be posed as follows:

```
doc("example.xml")//scene[@id="Interview"]/select-wide::song
```

Note the `select-wide` is an extension of the XPath (and thus XQuery) syntax.

Inside the server, the StandOff steps are implemented efficiently using sophisticated *interval-join* algorithms, as well as a *temporal index*. Both are employed automatically by MonetDB/XQuery, without need of user or DBA intervention. The XIME-P 2006 paper from our scientific library gives technical background on these StandOff extensions.

5.7.1 New XPath Steps

The StandOff axis steps, similar in behavior to the standard XPath steps (e.g. `child::*`, `descendant::*`) have been added to MonetDB/XQuery to make querying concurrent such region really easy.

- `/select-narrow::`
- `/select-wide::`
- `/reject-narrow::`
- `/reject-wide::`
- Axis steps will always be 'local' e.g. will only yield matches from the same document (fragment).
- Each node is only returned once (no duplicates) and in document order.

5.7.2 context/select-narrow::nodename

From the set of nodes with nodename 'nodename', say: {n1, n2...}, return only the nodes contained in the context nodes (e.g. return n if there is a context node for which holds: context_start <= n_start and n_end <= context_end)

5.7.2.1 context/select-wide::nodename

From the set of nodes with nodename 'nodename', say: {n1, n2...}, return only the nodes overlapping with the context nodes (e.g. return n if there is a context node for which holds: context_start <= n_end and n_start <= context_end)

5.7.2.2 context/reject-narrow::nodename

From the set of nodes with nodename 'nodename', say: {n1, n2...}, return all BUT the nodes contained in the context nodes (e.g. return n if there is NO context node for which holds: context_start <= n_start and n_end <= context_end)

5.7.2.3 context/reject-wide::nodename

From the set of nodes with nodename 'nodename', say: {n1, n2...}, return all BUT the nodes overlapping with the context nodes (e.g. return n if there is NO context node for which holds: context_start <= n_end and n_start <= context_end)

5.7.3 Enabling StandOff

The steps have been made available in MonetDB/XQuery next to the regular XPath axis. The StandOff steps have been **turned off by default** as they do not follow the XQuery recommendation as set by the W3C. To *enable* the steps you need to start the database server (Mserver) with the option `--set standoff=enable`.

5.7.4 Motivation and Examples

We have found a surprising wide variety of XML data owners to have region annotations:

- StandOff In Multimedia: XML that holds the output of video scene detection or speech recognition tools (etc.). Used in various kinds of content-based multimedia search/browsing systems.
- StandOff In Forensic: XML describing interesting features discovered on confiscated hard drives (e.g. person names, addresses, emails, recovered file hierarchies, etc..). The regions refer to the positions on disk where the features where found. Used in computer-assisted crime scene investigations (CSI).
- StandOff In NLP: XML describing the grammatical structure of natural texts. Inline annotation cannot be used because natural language is ambiguous, and multiple parses are often possible. Thus structure is separated from content, and refers to it by word position. Used in automatic question answering systems.
- StandOff In Bio-Informatics: XML storing DNA sequences annotated by genome research groups. The regions refer by position in the DNA strands. The annotations may contain clinical characteristics of patients or hold additional bio-molecular data on those genes. Used in collaborative genome research efforts.

If you have similar XML data and use MonetDB/XQuery to manage this, please contact us on the mailing list.

For XQueries with such region overlap/containment conditions, other XML database systems resort to query plans that have to compare all pairs of regions ("quadratic complexity"). On XML data sizes above a few hundred KB, this quickly systems become unusably slow. In contrast, MonetDB/XQuery with StandOff extensions runs bio-informatics queries on gigabytes of XML annotations within a few seconds.

5.8 Persistent Node Identifiers (NIDs)

If an XML document has an XML Schema or DTD, certain attributes can be marked to be ID or IDREF attributes. Such attributes can be used in the `fn:id()` function to look up nodes by their ID. This way, the XML Data Model provides some support for graph data structures.

Internally, each node in MonetDB/XQuery (not only element nodes, but nodes of all kinds, except attribute nodes) have a unique **node identifier** (NID). This NID can be compared with ROW-IDs in relational database system. Moreover, the NID is tightly coupled

to the physical location of the node's information in the database storage system. This also implies that given a NID, the database system knows exactly where it is located, so it can read the node's information without even having to use an index structure.

Such high-performance lookups may be useful to applications. For this reason we exposed an built-in extension function:

```
pf:nid($n as node()) as xs:string
```

It returns a string that contains a number. Note that in the XML Data Model, ID attributes must be non-numerical strings.

So the second extension was to modify the behavior of `fn:id()` to accept numeric identifiers, and to interpret them as NIDs. Thus you can use the normal `fn:id()` function to lookup your NIDs.

Warning: while NIDs are stable under updates, a Database Restore (see Section 2.5 [Backup/Restore], page 11) of an **updatable** XML Collection may change the NID values of your XML. So, in updatable XML data that should live across Restore points, you cannot reliably use NIDs. Note that this is not an issue for read-only XML Collections (see Section 2.4 [Read-only versus Updatable], page 11).

5.9 The Collection Node

MonetDB/XQuery provides efficient support for large collections of small XML documents by storing these together in a **XML Collection** (see Section 2.9.1 [Separate Documents vs Document Collections], page 15).

An XML collection, as identified by the standard function `fn:collection`, is considered to be a sequence of document nodes.

However, in MonetDB/XQuery, a collection is actually stored as an XML Tree, by putting a special **collection node** on top.

This is a non-standard implementation of XML Data Model, that in most aspects is just a hidden feature of the MonetDB/XQuery storage system, because in normal use the collection node is invisible.

That is, XPath steps will **never** return a collection node. However, when the downward axes `child::node()` and `descendant::*` (or shorter `/node()` and `//*`) are given the collection node **as input**, they do work. That is, all document nodes of the XML documents in the collection are returned by the `child::*` step and `descendant::*` reaches all nodes in the entire collection.

The built-in extension function:

```
pf:collection($coll as xs:string) as node()
```

provides you this special collection node. The reason for its introduction is performance-related: when the default `fn:collection` would yield thousands or millions of results, the use of `pf:` rather than `fn:` can make a difference; especially in combination with the use of element or value indices.

5.10 Temporary Documents

see Section 2.4 [Read-only versus Updatable], page 11

WARNING: the use case of serving out small parts of a large results is now better covered using the see Section 5.3 [Session Expression Cache], page 46.

One of the interesting uses of `fn:put()` (see Section 4.3.5 [The put() Function], page 44) is to `cache` intermediate results produced by a costly query. For instance, a GUI that shows some result table, may be able to show a limited amount of results on the screen and provide a scroll-bar to go up or down in the result list. Instead of recomputing the query result every time the scroll-bar is moved, an application can use `fn:put()` to serialize the query result in some temporary location TMP, and then use `fn:subsequence(doc('TMP'))` to show slices of it.

The question then arises where temporary files can be stored conveniently, and how they will be garbage collected. For this purpose, MonetDB/XQuery offers the `tmp/` directory (that is, the `<gdk_dbfarm>/<dbname>/tmp` directory). The MonetDB/XQuery server monitors this directory continuously, making sure that files that are older than `one hour` are deleted automatically. This simple mechanism makes it easy for e.g. web applications to use temporary XML, without having to resort to a stateful client-server protocol (e.g. a session object that would perform temporary XML cleanup at session end).

Of course, your disk should have enough space for one hour of produced temporary results, otherwise disk-full errors may occur. Use this feature with care.

6 Programming Interfaces

There are various ways to make your application programs access data in MonetDB/XQuery. The basic client-server programmers interface for MonetDB is **Mapi**; you can use it from C/C++. This is a fastest but also most low-level and MonetDB-specific interface that works with XQuery. While in principle Mapi bindings for other languages (perl, python, php) exist, these are currently focused on relational (MIL, MAL, SQL) access; some work is needed to make it possible to use them for XML (see contributing).

For high-performance (low-latency) applications such as web environments, we recommend using **XRPC**. MonetDB/XQuery comes with a built-in HTTP server that can service SOAP calls that execute an XQuery via the XRPC mechanism. Such SOAP requests invoke a XQuery Function that must be predefined in an XQuery Module. Such pre-defined functions are executed as *canned queries*, therefore can be very efficient (no query optimization time). We provide convenience APIs for posing XRPC SOAP queries from Java and Javascript (i.e. directly from web pages). The Administrative GUI of MonetDB/XQuery an example of an application entirely based on such AJAX-style web pages (see Section 3.2 [The Administrative GUI], page 26).

MonetDB/SQL also supports **JDBC** and this interface can also be used to pose XQuery queries to MonetDB/XQuery (passing a special `language=xquery` option at connection initialization).

For reference, we also provide the instructions for setting up `mclient` using **CGI** under the Apache web-server. However, we consider CGI superseded by XRPC for web-based applications.

6.1 Using XRPC from JavaScript

WARNING: XRPC still makes use of the old compiler backend and does not use the optimizing algebraic query compiler. Therefore, its query performance can sometimes be inferior to other queries handled by MonetDB/XQuery. Also, as the old compiler backend is gradually phased out, it gets to be less well-maintained and tested in general. Use with caution. We hope to port XRPC to the algebra backend soon.

XRPC allows to make SOAP calls to a MonetDB/XQuery server. With XRPC you can invoke predefined XQuery Functions. The XQuery function must be defined in an XQuery Module file that should be accessible by the server via an URL.

Because XRPC makes use of such predefined modules, MonetDB/XQuery can preprocess the module and perform query optimization beforehand. This makes XRPC a highly efficient API, allowing in simple queries for response times of for less than 10 milliseconds.

By default when MonetDB/XQuery starts, its HTTP server is started on port 50001. It serves out the directory is in `share/xrpc/`. This HTTP server interprets POST requests that have a local URI starting with `/xrpc` as XRPC requests. Inside the POST request body is a SOAP request, and the returned answer is a SOAP message again. This API creates a valid SOAP request, sends it, and calls a callback function when the response comes in.

WARNING: XRPC still makes use of the old compiler backend and does not use the optimizing algebraic query compiler. Therefore, its query performance can sometimes be

inferior to other queries handled by MonetDB/XQuery. Also, as the old compiler backend is gradually phased out, it gets to be less well-maintained and tested in general. Use with caution. We hope to port XRPC to the algebra backend soon.

6.1.1 API

In the file `share/xrpc/admin/xrpcwebclient.js` you find a JavaScript library that works with both FireFox and Internet Explorer.

For safety reasons, Internet Explorer and FireFox usually only allow SOAP requests to be sent from Javascript to the same host as the web-server!!. This restriction also affects this JavaScript XRPC API.

The main function is `XRPC()`, it takes the URL of a MonetDB/XQuery server.

```
function XRPC(posturl,    /* Your XRPC server. Usually: "http://yourhost:yourport/xrpc" */
              module,     /* (logical) module namespace URI (NB: do not use the user define
              moduleurl,  /* module (physical) at-hint URL. Module file must be here! */
              method,     /* method name (matches function name in module) */
              arity,      /* arity of the method, i.e. number of parameters. */
              call,       /* one or more XRPC_CALL() parameter specs (concatenated strings)
              callback);  /* a JavaScript callback function that should be called when the
```

What you get back is the full SOAP result message described in TODO. You can process it in Javascript as you like.

The function parameters is a string that can be constructed using the following helper functions:

```
function XRPC_CALL(parameters);
function XRPC_SEQ(sequence);
function XRPC_ATOM(type, value);
function XRPC_ELEMENT(value);
```

Each set of parameters is enclosed in an XRPC_CALL. It is in fact possible to pass multiple such XRPC_CALLS. This means the function will be invoked multiple times and you will get back multiple result sequences (one for each call).

For each parameter in a function call, you specify a sequences, i.e. an XRPC_SEQ. Inside a sequence, you find zero or more XRPC_ATOMs and/or XRPC_ELEMENTs. For atomic types, the type is assumed to be from the `xs` namespace, e.g. passing `'integer'` gives you an `xs:integer`.

6.1.2 Example

You must include the file `share/xrpc/admin/xrpcwebclient.js` in your HTML as follows:

```
<html>
    <head>
        <script type="text/javascript" src="xrpcwebclient.js">
            var clnt = new XRPCWebClient();
        </script>
    </head>

    ..
```

Then you fire off from Javascript an XRPC call, e.g. calling the `add(100,200)` function defined in an XQuery Module.

```
XRPC('http://localhost:50001',
     'xrpc-test-function',
     'http://www.monetdb.nl/XQuery/files/xrpc-mod.xq",
     'add',
     '2',
     XRPC_CALL(XRPC_SEQ(XRPC_ATOM('integer', '100') + XRPC_ATOM('integer', '200'))),
     exampleCallback);

exampleCallback(result) {
    alert(result);
}
```

The MonetDB/XQuery administrative GUI (`http://localhost:5001/admin/`) is fully programmed using Javascript and XRPC. You can study its code in `share/xrpc/admin/admin.js`.

6.2 Using XRPC from Java

WARNING: XRPC still makes use of the old compiler backend and does not use the optimizing algebraic query compiler. Therefore, its query performance can sometimes be inferior to other queries handled by MonetDB/XQuery. Also, as the old compiler backend is gradually phased out, it gets to be less well-maintained and tested in general. Use with caution. We hope to port XRPC to the algebra backend soon.

XRPC allows to make SOAP calls to a MonetDB/XQuery server. With XRPC you can invoke predefined XQuery Functions. The XQuery function must be defined in an XQuery Module file that should be accessible by the server via an URL.

Because XRPC makes use of such predefined modules, MonetDB/XQuery can preprocess the module and perform query optimization beforehand. This makes XRPC a highly efficient API, allowing in simple queries for response times of for less than 10 milliseconds.

By default when MonetDB/XQuery starts, its HTTP server is started on port 50001. It serves out the directory is in `share/xrpc/`. This HTTP server interprets POST requests that have a local URI starting with `/xrpc` as XRPC requests. Inside the POST request body is a SOAP request, and the returned answer is a SOAP message again. This API creates a valid SOAP request, sends it, and calls a callback function when the response comes in.

WARNING: XRPC still makes use of the old compiler backend and does not use the optimizing algebraic query compiler. Therefore, its query performance can sometimes be inferior to other queries handled by MonetDB/XQuery. Also, as the old compiler backend is gradually phased out, it gets to be less well-maintained and tested in general. Use with caution. We hope to port XRPC to the algebra backend soon.

6.2.1 API

The Java client API is defined in package `nl.cwi.monetdb.xquery.util`, and is included in both the XRPC wrapper (`$prefix/share/MoneDB/lib/xrpcwrapper.jar`) and the test

client of the XRPC wrapper (`$prefix/share/MoneDB/lib/xrpcwrapper-test.jar`)[1]. The test client program (`$prefix/share/MoneDB/lib/xrpcwrapper-test.jar`) of the XRPC Wrapper is a complete example for using the Java client API.

The JavaDoc can be found here.

6.3 The JDBC Library

NOTE: to use JDBC for XQuery querying, one should add `language=xquery` to the connect string:

```
DriverManager.getConnection(
    "jdbc:monetdb://localhost:50000/database?language=xquery",
     "monetdb", "monetdb");
```

The result comes back as a single-row single-column multi-line string result. That is, JDBC makes no attempt at interpreting the XQuery output.

Note that by default the XQuery server is only reachable from the localhost. If you need the server to be connected from other hosts using JDBC, make sure you uncomment `mapi_open=yes` in the `MonetDB.conf` file.

6.3.1 MonetDB JDBC Driver

The most obvious way to connect to a data source using the Java programming language is by making use of the in Java defined JDBC framework. MonetDB has a native Java JDBC driver type 4 which allows use of the MonetDB database in a Java alike way.

It is quite difficult to have a fully complete JDBC implementation. Also this driver isn't complete in that sense. However, it is believed that the most prominent parts of the JDBC interface are implemented, and in such a way that they adhere to the specifications. If you make extensive use of JDBC semantics and rely on many of its features, please read the release notes which are to be found in the SRC/JDBC directory of the sql CVS tree.

This document gives a short description how to use the MonetDB JDBC driver in Java applications. A familiarity with the Java JDBC API is required to fully understand this document. Please note that you can find the complete JDBC API on Sun's web site HTTP://JAVA.SUN.COM/.

In order to use the MonetDB JDBC driver in Java applications you need (of course) a running MonetDB/SQL instance, preferably via merovingian.

6.3.1.1 Getting the driver Jar

The easiest way to acquire the driver is to download it from our download page. You will find a file called called MONETDB-X.Y-JDBC.JAR where X and Y are major and minor version numbers. The current release as of this writing is 1.11.

6.3.1.2 Compiling the driver (using ant, optional)

If you prefer to build the driver yourself, make sure you acquire the MonetDB Java repository, e.g. as part of the Super Source Tarball. The Java sources are built using Apache's Ant tool. Simply issuing the command ANT DISTJDBC should be sufficient to build the

[1] (Note: the Java client API will be released in a separate JAR package in the next release of MonetDB/XQuery.)

driver jar-archive in the subdirectory JARS. See the ANT web site for more documentation on the ant build-tool: HTTP://ANT.APACHE.ORG/. The Java sources require at least a Java 2 platform 1.4 compatible compiler. The JDBC driver, however, currently cannot be compiled with a Java 1.6 or up compiler.

6.3.1.3 Testing the driver using the JdbcClient utility

Before you start developing your programs which use the MonetDB JDBC driver it is generally a good idea to check if the driver actually works in your environment. JdbcClient is a no longer distributed, but when compling from sources, it is still built and put in the JARS directory. Follow the steps below to assure your setup is complete:

1. start merovingian
2. create a database using MONETDB CREATE MYTEST
3. run the JdbcClient utility using JAVA -JAR {PATH/TO/JDBCCLIENT.JAR} -DMYTEST -UMONETDB (with password monetdb)

The last step should give you something like this:

```
% java -jar jars/jdbcclient.jar -umonetdb
password:

Welcome to the MonetDB interactive JDBC terminal!
Database: MonetDB 5.0.0
Driver: MonetDB Native Driver 1.5 (Steadfast_pre4 20061124)
Type \q to quit, \h for a list of available commands
auto commit mode: on
monetdb->
```

From here you can execute a simple query to assure yourself everything is setup to work correctly. If the connection fails, observe the error messages from JdbcClient and the merovingian logs for clues.

6.3.1.4 Using the driver in your Java programs

To use the MonetDB JDBC driver, the MONETDB-X.Y-JDBC.JAR jar-archive has to be in the Java classpath. Make sure this is actually the case.

Loading the driver in your Java program requires two lines of code:

```
// make sure the ClassLoader has the MonetDB JDBC driver loaded
Class.forName("nl.cwi.monetdb.jdbc.MonetDriver");
// request a Connection to a MonetDB server running on 'localhost'
Connection con = DriverManager.getConnection("jdbc:monetdb://localhost/database", "monetdb"
```

The first line makes sure the Java ClassLoader has initialised (and loaded) the Driver class of the MonetDB JDBC package, so it is registered with the DriverManager. The second line requests a Connection object from the DriverManager which is suitable for MonetDB.

The string passed to the "GETCONNECTION()"method is defined as "JDBC:MONETDB://<HOST>[:<PORT>]/<DATABASE>" where elements between "<" and ">" are required and elements between "[" and "]" are optional.

6.3.1.5 A sample Java program

```
import java.sql.*;

/**
 * This example assumes there exist tables a and b filled with some data.
 * On these tables some queries are executed and the JDBC driver is tested
 * on it's accuracy and robustness against 'users'.
 *
 * @author Fabian Groffen
 */
public class MJDBCTest {
    public static void main(String[] args) throws Exception {
        // make sure the driver is loaded
        Class.forName("nl.cwi.monetdb.jdbc.MonetDriver");
        Connection con = DriverManager.getConnection("jdbc:monetdb://localhost/database", "
        Statement st = con.createStatement();
        ResultSet rs;

        rs = st.executeQuery("SELECT a.var1, COUNT(b.id) as total FROM a, b WHERE a.var1 = 
        // get meta data and print columns with their type
        ResultSetMetaData md = rs.getMetaData();
        for (int i = 1; i <= md.getColumnCount(); i++) {
            System.out.print(md.getColumnName(i) + ":" +
                md.getColumnTypeName(i) + "\t");
        }
        System.out.println("");
        // print the data: only the first 5 rows, while there probably are
        // a lot more. This shouldn't cause any problems afterwards since the
        // result should get properly discarded on the next query
        for (int i = 0; rs.next() && i < 5; i++) {
            for (int j = 1; j <= md.getColumnCount(); j++) {
                System.out.print(rs.getString(j) + "\t");
            }
            System.out.println("");
        }

        // tell the driver to only return 5 rows, it can optimize on this
        // value, and will not fetch any more than 5 rows.
        st.setMaxRows(5);
        // we ask the database for 22 rows, while we set the JDBC driver to
        // 5 rows, this shouldn't be a problem at all...
        rs = st.executeQuery("select * from a limit 22");
        // read till the driver says there are no rows left
        for (int i = 0; rs.next(); i++) {
            System.out.print("[" + rs.getString("var1") + "]");
            System.out.print("[" + rs.getString("var2") + "]");
```

```
            System.out.print("[" + rs.getInt("var3") + "]");
            System.out.println("[" + rs.getString("var4") + "]");
        }

        // this close is not needed, should be done by next execute(Query) call
        // however if there can be some time between this point and the next
        // execute call, it is from a resource perspective better to close it.
        //rs.close();

        // unset the row limit; 0 means as much as the database sends us
        st.setMaxRows(0);
        // we only ask 10 rows
        rs = st.executeQuery("select * from b limit 10;");
        // and simply print them
        while (rs.next()) {
            System.out.print(rs.getInt("rowid") + ", ");
            System.out.print(rs.getString("id") + ", ");
            System.out.print(rs.getInt("var1") + ", ");
            System.out.print(rs.getInt("var2") + ", ");
            System.out.print(rs.getString("var3") + ", ");
            System.out.println(rs.getString("var4"));
        }

        // this close is not needed, as the Statement will close the last
        // ResultSet around when it's closed
        // again, if that can take some time, it's nicer to close immediately
        // the reason why these closes are commented out here, is to test if
        // the driver really cleans up it's mess like it should
        //rs.close();

        // perform a ResultSet-less query (with no trailing ; since that should
        // be possible as well and is JDBC standard)
        // Note that this method should return the number of updated rows. This
        // method however always returns -1, since Monet currently doesn't
        // support returning the affected rows.
        st.executeUpdate("delete from a where var1 = 'zzzz'");

        // closing the connection should take care of closing all generated
        // statements from it...
        // don't forget to do it yourself if the connection is reused or much
        // longer alive, since the Statement object contains a lot of things
        // you probably want to reclaim if you don't need them anymore.
        //st.close();
        con.close();
    }
}
```

6.4 The Mapi Library

The easiest way to extend the functionality of MonetDB is to construct an independent application, which communicates with a running server using a database driver with a simple API and a textual protocol. The effectiveness of such an approach has been demonstrated by the wide use of database API implementations, such as Perl DBI, PHP, ODBC,...

6.4.1 An Example

C and C++ programs can use the MAPI library to execute queries on MonetDB/XQuery.

We give a short example with a minimal Mapi program:

- `mapi_connect()` and `mapi_disconnect()`: make a connection to a database server (`Mapi mid;`). **note:** pass the value `"xquery"` in the `language` parameter, when connecting.
- `mapi_profile()` and `mapi_output()`: configure query timing and XML output format.
- `mapi_error()` and `mapi_error_str()`: check for and print connection errors (on `Mapi mid`).
- `mapi_query()` and `mapi_close_handle()` do a query and get a handle to it (`MapiHdl hdl`).
- `mapi_result_error()`: check for query evaluation errors (on `MapiHdl hdl`).
- `mapi_fetch_line()`: get a line of (result or error) output from the server (on `MapiHdl hdl`). **note:** output lines are prefixed with a `'='` character that must be escaped.

```
#include <stdio.h>
#include <Mapi.h>
#include <stdlib.h>

int
main(int argc, char** argv) {
        const char *prog  = argv[0];
        const char *host  = argv[1]; /* where Mserver is started, e.g. localhost */
        const char *db    = argv[2]; /* database name e.g. demo */
        int  port         = atoi(argv[3]); /* mapi_port e.g. 50000 */
        char *mode        = argv[4]; /* output format e.g. xml  */
        const char *query = argv[5]; /* single-line query e.g. '1+1' (use quotes) */
        FILE *fp          = stderr;
        char *line;

        if (argc != 6) {
                fprintf(fp, "usage: %s <host>    <db> <port> <mode> <query>\n", prog);
                fprintf(fp, "  e.g. %s localhost demo 50000  xml    '1+1'\n",   prog);
        } else {
                /* CONNECT TO SERVER, default unsecure user/password, language="xquery
                Mapi    mid = mapi_connect(host, port, "monetdb", "monetdb", "xquery",
                MapiHdl hdl;
                if (mid == NULL) {
                        fprintf(fp, "%s: failed to connect.\n", prog);
                } else {
```

```
                        mapi_profile(mid, 1);    /* SWITCH ON SERVER-SIDE QUERY TIMING
                        mapi_output(mid, mode);  /* SET XML OUTPUT FORMAT */
                        hdl = mapi_query(mid, query); /* FIRE OFF A QUERY */

                        if (hdl == NULL || mapi_error(mid) != MOK) /* CHECK CONNECTION
                                fprintf(fp, "%s: connection error: %s\n", prog, mapi_e
                        if (hdl) {
                                if (mapi_result_error(hdl) != MOK) /* CHECK QUERY ERRO
                                        fprintf(fp, "%s: query error\n", prog);
                                else
                                        fp = stdout; /* success: connection&query went

                                /* FETCH SERVER QUERY ANSWER LINE-BY-LINE */
                                while((line = mapi_fetch_line(hdl)) != NULL) {
                                        if (*line == '=') line++; // XML result lines
                                        fprintf(fp, "%s\n", line);
                                }
                        }
                        mapi_close_handle(hdl); /* CLOSE QUERY HANDLE */
                }
                mapi_disconnect(mid); /* CLOSE CONNECTION */
        }
        return (fp == stdout)? 0 : -1;
}
```

The following action is needed to get a working program. Compilation of the application relies on the *monetdb-config* program shipped with the distribution. It localizes the include files and library directories. Once properly installed, the application can be compiled and linked as follows:

```
cc sample.c `monetdb-clients-config --cflags --libs` -lMapi -o sample
./sample
```

It assumes that the dynamic loadable libraries are in public places. If, however, the system is installed in your private environment then the following option can be used on most ELF platforms.

```
cc sample.c `monetdb-clients-config --cflags --libs` -lMapi -o sample \
`monetdb-clients-config --libs | sed -e's:-L:-R:g'`
./sample
```

The compilation on Windows is slightly more complicated. It requires more attention towards the location of the include files and libraries.

6.4.2 Command Summary

The quick reference guide to the Mapi library is given below. More details on their constraints and defaults are given in the next section.

mapi_bind()	Bind string C-variable to a field
mapi_bind_numeric()	Bind numeric C-variable to field
mapi_bind_var()	Bind typed C-variable to a field

mapi_cache_freeup()	Forcefully shuffle fraction for cache refreshment
mapi_cache_limit()	Set the tuple cache limit
mapi_cache_shuffle()	Set shuffle fraction for cache refreshment
mapi_clear_bindings()	Clear all field bindings
mapi_clear_params()	Clear all parameter bindings
mapi_close_handle()	Close query handle and free resources
mapi_connect()	Connect to a Mserver
mapi_destroy()	Free handle resources
mapi_disconnect()	Disconnect from server
mapi_error()	Test for error occurrence
mapi_execute()	Execute a query
mapi_execute_array()	Execute a query using string arguments
mapi_explain()	Display error message and context on stream
mapi_explain_query()	Display error message and context on stream
mapi_fetch_all_rows()	Fetch all answers from server into cache
mapi_fetch_field()	Fetch a field from the current row
mapi_fetch_field_len()	Fetch the length of a field from the current row
mapi_fetch_field_array()	Fetch all fields from the current row
mapi_fetch_line()	Retrieve the next line
mapi_fetch_reset()	Set the cache reader to the beginning
mapi_fetch_row()	Fetch row of values
mapi_finish()	Terminate the current query
mapi_get_dbname()	Database being served
mapi_get_field_count()	Number of fields in current row
mapi_get_host()	Host name of server
mapi_get_query()	Query being executed
mapi_get_language()	Query language name
mapi_get_mapi_version()	Mapi version name
mapi_get_monet_versionId()	MonetDB version identifier
mapi_get_monet_version()	MonetDB version name
mapi_get_motd()	Get server welcome message
mapi_get_row_count()	Number of rows in cache or -1
mapi_get_last_id()	last inserted id of an auto_increment (or alike) column
mapi_get_from()	Get the stream 'from'
mapi_get_to()	Get the stream 'to'
mapi_get_trace()	Get trace flag
mapi_get_user()	Current user name
mapi_log()	Keep log of client/server interaction
mapi_next_result()	Go to next result set
mapi_needmore()	Return whether more data is needed
mapi_ping()	Test server for accessibility
mapi_prepare()	Prepare a query for execution
mapi_prepare_array()	Prepare a query for execution using arguments
mapi_query()	Send a query for execution
mapi_query_array()	Send a query for execution with arguments
mapi_query_handle()	Send a query for execution
mapi_quick_query_array()	Send a query for execution with arguments

mapi_quick_query()	Send a query for execution
mapi_quick_response()	Quick pass response to stream
mapi_quote()	Escape characters
mapi_reconnect()	Reconnect with a clean session context
mapi_rows_affected()	Obtain number of rows changed
mapi_seek_row()	Move row reader to specific location in cache
mapi_setAutocommit()	Set auto-commit flag
mapi_setAlgebra()	Use algebra backend
mapi_stream_query()	Send query and prepare for reading tuple stream
mapi_table()	Get current table name
mapi_timeout()	Set timeout for long-running queries[TODO]
mapi_output()	Set output format
mapi_stream_into()	Stream document into server
mapi_profile()	Set profile flag
mapi_trace()	Set trace flag
mapi_virtual_result()	Submit a virtual result set
mapi_unquote()	remove escaped characters

6.4.3 Library Synopsis

The routines to build a MonetDB application are grouped in the library MonetDB Programming Interface, or shorthand Mapi.

The protocol information is stored in a Mapi interface descriptor (mid). This descriptor can be used to ship queries, which return a MapiHdl to represent the query answer. The application can set up several channels with the same or a different `mserver`. It is the programmer's responsibility not to mix the descriptors in retrieving the results.

The application may be multi-threaded as long as the user respects the individual connections represented by the database handlers.

The interface assumes a cautious user, who understands and has experience with the query or programming language model. It should also be clear that references returned by the API point directly into the administrative structures of Mapi. This means that they are valid only for a short period, mostly between successive `mapi_fetch_row()` commands. It also means that it the values are to retained, they have to be copied. A defensive programming style is advised.

Upon an error, the routines `mapi_explain()` and `mapi_explain_query()` give information about the context of the failed call, including the expression shipped and any response received. The side-effect is clearing the error status.

6.4.4 Error Message

Almost every call can fail since the connection with the database server can fail at any time. Functions that return a handle (either `Mapi` or `MapiHdl`) may return NULL on failure, or they may return the handle with the error flag set. If the function returns a non-NULL handle, always check for errors with mapi_error.

Functions that return MapiMsg indicate success and failure with the following codes.

MOK	No error
MERROR	Mapi internal error.

MTIMEOUT Error communicating with the server.

When these functions return MERROR or MTIMEOUT, an explanation of the error can be had by calling one of the functions `mapi_error_str()`, `mapi_explain()`, or `mapi_explain_query()`.

To check for error messages from the server, call `mapi_result_error()`. This function returns NULL if there was no error, or the error message if there was. A user-friendly message can be printed using `map_explain_result()`. Typical usage is:

```
do {
    if ((error = mapi_result_error(hdl)) != NULL)
        mapi_explain_result(hdl, stderr);
    while ((line = mapi_fetch_line(hdl)) != NULL)
        /* use output */;
} while (mapi_next_result(hdl) == 1);
```

6.4.5 Mapi Function Reference

6.4.6 Connecting and Disconnecting

- Mapi mapi_connect(const char *host, int port, const char *username, const char *password, const char *lang, const char *dbname)

  Setup a connection with a Mserver at a *host*:*port* and login with *username* and *password*. If host == NULL, the local host is accessed. If host starts with a '/' and the system supports it, host is actually the name of a UNIX domain socket, and port is ignored. If port == 0, a default port is used. If username == NULL, the username of the owner of the client application containing the Mapi code is used. If password == NULL, the password is omitted. The preferred query language is any of {sql,mil,mal,xquery }. On success, the function returns a pointer to a structure with administration about the connection.

- MapiMsg mapi_disconnect(Mapi mid)

  Terminate the session described by *mid*. The only possible uses of the handle after this call is *mapi_destroy()* and `mapi_reconnect()`. Other uses lead to failure.

- MapiMsg mapi_destroy(Mapi mid)

  Terminate the session described by *mid* if not already done so, and free all resources. The handle cannot be used anymore.

- MapiMsg mapi_reconnect(Mapi mid)

  Close the current channel (if still open) and re-establish a fresh connection. This will remove all global session variables.

- MapiMsg mapi_ping(Mapi mid)

  Test availability of the server. Returns zero upon success.

6.4.7 Sending Queries

- MapiHdl mapi_query(Mapi mid, const char *Command)

  Send the Command to the database server represented by mid. This function returns a query handle with which the results of the query can be retrieved. The handle

should be closed with `mapi_close_handle()`. The command response is buffered for consumption, c.f. mapi_fetch_row().

- MapiMsg mapi_query_handle(MapiHdl hdl, const char *Command)

  Send the Command to the database server represented by hdl, reusing the handle from a previous query. If Command is zero it takes the last query string kept around. The command response is buffered for consumption, e.g. `mapi_fetch_row()`.

- MapiHdl mapi_query_array(Mapi mid, const char *Command, char **argv)

  Send the Command to the database server replacing the placeholders (?) by the string arguments presented.

- MapiHdl mapi_quick_query(Mapi mid, const char *Command, FILE *fd)

  Similar to `mapi_query()`, except that the response of the server is copied immediately to the file indicated.

- MapiHdl mapi_quick_query_array(Mapi mid, const char *Command, char **argv, FILE *fd)

  Similar to `mapi_query_array()`, except that the response of the server is not analyzed, but shipped immediately to the file indicated.

- MapiHdl mapi_stream_query(Mapi mid, const char *Command, int windowsize)

  Send the request for processing and fetch a limited number of tuples (determined by the window size) to assess any erroneous situation. Thereafter, prepare for continual reading of tuples from the stream, until an error occurs. Each time a tuple arrives, the cache is shifted one.

- MapiHdl mapi_prepare(Mapi mid, const char *Command)

  Move the query to a newly allocated query handle (which is returned). Possibly interact with the back-end to prepare the query for execution.

- MapiMsg mapi_execute(MapiHdl hdl)

  Ship a previously prepared command to the backend for execution. A single answer is pre-fetched to detect any runtime error. MOK is returned upon success.

- MapiMsg mapi_execute_array(MapiHdl hdl, char **argv)

  Similar to `mapi\_execute` but replacing the placeholders for the string values provided.

- MapiMsg mapi_finish(MapiHdl hdl)

  Terminate a query. This routine is used in the rare cases that consumption of the tuple stream produced should be prematurely terminated. It is automatically called when a new query using the same query handle is shipped to the database and when the query handle is closed with `mapi_close_handle()`.

- MapiMsg mapi_virtual_result(MapiHdl hdl, int columns, const char **columnnames, const char **columntypes, const int *columnlengths, int tuplecount, const char ***tuples)

  Submit a table of results to the library that can then subsequently be accessed as if it came from the server. columns is the number of columns of the result set and must be greater than zero. columnnames is a list of pointers to strings giving the names of the individual columns. Each pointer may be NULL and columnnames may be NULL if there are no names. tuplecount is the length (number of rows) of the result set. If

tuplecount is less than zero, the number of rows is determined by a NULL pointer in the list of tuples pointers. tuples is a list of pointers to row values. Each row value is a list of pointers to strings giving the individual results. If one of these pointers is NULL it indicates a NULL/nil value.

6.4.8 Getting Results

- int mapi_get_field_count(MapiHdl mid)

  Return the number of fields in the current row.

- mapi_int64 mapi_get_row_count(MapiHdl mid)

  If possible, return the number of rows in the last select call. A -1 is returned if this information is not available.

- mapi_int64 mapi_get_last_id(MapiHdl mid)

  If possible, return the last inserted id of auto_increment (or alike) column. A -1 is returned if this information is not available. We restrict this to single row inserts and one auto_increment column per table. If the restrictions do not hold, the result is unspecified.

- mapi_int64 mapi_rows_affected(MapiHdl hdl)

  Return the number of rows affected by a database update command such as SQL's INSERT/DELETE/UPDATE statements.

- int mapi_fetch_row(MapiHdl hdl)

  Retrieve a row from the server. The text retrieved is kept around in a buffer linked with the query handle from which selective fields can be extracted. It returns the number of fields recognized. A zero is returned upon encountering end of sequence or error. This can be analyzed in using `mapi_error()`.

- mapi_int64 mapi_fetch_all_rows(MapiHdl hdl)

  All rows are cached at the client side first. Subsequent calls to `mapi_fetch_row()` will take the row from the cache. The number or rows cached is returned.

- int mapi_quick_response(MapiHdl hdl, FILE *fd)

  Read the answer to a query and pass the results verbatim to a stream. The result is not analyzed or cached.

- MapiMsg mapi_seek_row(MapiHdl hdl, mapi_int64 rownr, int whence)

  Reset the row pointer to the requested row number. If whence is `MAPI_SEEK_SET`, rownr is the absolute row number (0 being the first row); if whence is `MAPI_SEEK_CUR`, rownr is relative to the current row; if whence is `MAPI_SEEK_END`, rownr is relative to the last row.

- MapiMsg mapi_fetch_reset(MapiHdl hdl)

  Reset the row pointer to the first line in the cache. This need not be a tuple. This is mostly used in combination with fetching all tuples at once.

- char **mapi_fetch_field_array(MapiHdl hdl)

  Return an array of string pointers to the individual fields. A zero is returned upon encountering end of sequence or error. This can be analyzed in using `mapi\_error()`.

- char *mapi_fetch_field(MapiHdl hdl, int fnr)

  Return a pointer a C-string representation of the value returned. A zero is returned upon encountering an error or when the database value is NULL; this can be analyzed in using `mapi\_error()`.

- size_t mapi_fetch_fiels_len(MapiHdl hdl, int fnr)

  Return the length of the C-string representation excluding trailing NULL byte of the value. Zero is returned upon encountering an error, when the database value is NULL, of when the string is the empty string. This can be analyzed by using `mapi\_error()` and `mapi\_fetch\_field()`.

- MapiMsg mapi_next_result(MapiHdl hdl)

  Go to the next result set, discarding the rest of the output of the current result set.

6.4.9 Errors

- MapiMsg mapi_error(Mapi mid)

  Return the last error code or 0 if there is no error.

- char *mapi_error_str(Mapi mid)

  Return a pointer to the last error message.

- char *mapi_result_error(MapiHdl hdl)

  Return a pointer to the last error message from the server.

- MapiMsg mapi_explain(Mapi mid, FILE *fd)

  Write the error message obtained from `mserver` to a file.

- MapiMsg mapi_explain_query(MapiHdl hdl, FILE *fd)

  Write the error message obtained from `mserver` to a file.

- MapiMsg mapi_explain_result(MapiHdl hdl, FILE *fd)

  Write the error message obtained from `mserver` to a file.

6.4.10 Parameters

- MapiMsg mapi_bind(MapiHdl hdl, int fldnr, char **val)

  Bind a string variable with a field in the return table. Upon a successful subsequent `mapi\_fetch\_row()` the indicated field is stored in the space pointed to by val. Returns an error if the field identified does not exist.

- MapiMsg mapi_bind_var(MapiHdl hdl, int fldnr, int type, void *val)

  Bind a variable to a field in the return table. Upon a successful subsequent `mapi\_fetch\_row()`, the indicated field is converted to the given type and stored in the space pointed to by val. The types recognized are `{ MAPI\_TINY, MAPI\_UTINY, MAPI\_SHORT, MAPI\_USHORT, MAPI_INT, MAPI_UINT, MAPI_LONG, MAPI_ULONG, MAPI_LONGLONG, MAPI_ULONGLONG, MAPI_CHAR, MAPI_VARCHAR, MAPI_FLOAT, MAPI_DOUBLE, MAPI_DATE, MAPI_TIME, MAPI_DATETIME }`. The binding operations should be performed after the mapi_execute command. Subsequently all rows being fetched also involve delivery of the field values in the C-variables using proper conversion. For variable length strings a pointer is set into the cache.

- MapiMsg mapi_bind_numeric(MapiHdl hdl, int fldnr, int scale, int precision, void *val)

  Bind to a numeric variable, internally represented by MAPI_INT Describe the location of a numeric parameter in a query template.
- MapiMsg mapi_clear_bindings(MapiHdl hdl)

  Clear all field bindings.
- MapiMsg mapi_param(MapiHdl hdl, int fldnr, char **val)

  Bind a string variable with the n-th placeholder in the query template. No conversion takes place.
- MapiMsg mapi_param_type(MapiHdl hdl, int fldnr, int ctype, int sqltype, void *val)

  Bind a variable whose type is described by ctype to a parameter whose type is described by sqltype.
- MapiMsg mapi_param_numeric(MapiHdl hdl, int fldnr, int scale, int precision, void *val)

  Bind to a numeric variable, internally represented by MAPI_INT.
- MapiMsg mapi_param_string(MapiHdl hdl, int fldnr, int sqltype, char *val, int *sizeptr)

  Bind a string variable, internally represented by MAPI_VARCHAR, to a parameter. The sizeptr parameter points to the length of the string pointed to by val. If sizeptr == NULL or *sizeptr == -1, the string is NULL-terminated.
- MapiMsg mapi_clear_params(MapiHdl hdl)

  Clear all parameter bindings.

6.4.11 Miscellaneous

- MapiMsg mapi_setAutocommit(Mapi mid, int autocommit)

  Set the autocommit flag (default is on). This only has an effect when the language is SQL. In that case, the server commits after each statement sent to the server.
- MapiMsg mapi_setAlgebra(Mapi mid, int algebra)

  Tell the backend to use or stop using the algebra-based compiler.
- MapiMsg mapi_cache_limit(Mapi mid, int maxrows)

  A limited number of tuples are pre-fetched after each `execute()`. If maxrows is negative, all rows will be fetched before the application is permitted to continue. Once the cache is filled, a number of tuples are shuffled to make room for new ones, but taking into account non-read elements. Filling the cache quicker than reading leads to an error.
- MapiMsg mapi_cache_shuffle(MapiHdl hdl, int percentage)

  Make room in the cache by shuffling percentage tuples out of the cache. It is sometimes handy to do so, for example, when your application is stream-based and you process each tuple as it arrives and still need a limited look-back. This percentage can be set between 0 to 100. Making shuffle= 100% (default) leads to paging behavior, while shuffle==1 leads to a sliding window over a tuple stream with 1% refreshing.
- MapiMsg mapi_cache_freeup(MapiHdl hdl, int percentage)

  Forcefully shuffle the cache making room for new rows. It ignores the read counter, so rows may be lost.

- char * mapi_quote(const char *str, int size)

  Escape special characters such as `\n`, `\t` in str with backslashes. The returned value is a newly allocated string which should be freed by the caller.
- char * mapi_unquote(const char *name)

  The reverse action of `mapi_quote()`, turning the database representation into a C-representation. The storage space is dynamically created and should be freed after use.
- MapiMsg mapi_output(Mapi mid, char *output)

  Set the output format for results send by the server.
- MapiMsg mapi_stream_into(Mapi mid, char *docname, char *colname, FILE *fp)

  Stream a document into the server. The name of the document is specified in docname, the collection is optionally specified in colname (if NULL, it defaults to docname), and the content of the document comes from fp.
- MapiMsg mapi_profile(Mapi mid, int flag)

  Set the profile flag to time commands send to the server.
- MapiMsg mapi_trace(Mapi mid, int flag)

  Set the trace flag to monitor interaction of the client with the library. It is primarilly used for debugging Mapi applications.
- int mapi_get_trace(Mapi mid)

  Return the current value of the trace flag.
- MapiMsg mapi_log(Mapi mid, const char *fname)

  Log the interaction between the client and server for offline inspection. Beware that the log file overwrites any previous log. For detailed interaction trace with the Mapi library itself use mapi_trace().

The remaining operations are wrappers around the data structures maintained. Note that column properties are derived from the table output returned from the server.

- char *mapi_get_name(MapiHdl hdl, int fnr)
- char *mapi_get_type(MapiHdl hdl, int fnr)
- char *mapi_get_table(MapiHdl hdl, int fnr)
- int mapi_get_len(Mapi mid, int fnr)
- char *mapi_get_dbname(Mapi mid)
- char *mapi_get_host(Mapi mid)
- char *mapi_get_user(Mapi mid)
- char *mapi_get_lang(Mapi mid)
- char *mapi_get_motd(Mapi mid)

6.5 CGI binding for .xq files

Here are the instructions to make `.xq` files directly executable from a Apache web server.

Note: The code provided below is still in an experimental stage and is not intended to be used in a production environment.

6.5.1 httpd.conf

First you must adapt the httpd.conf configuration file of the Apache web server, in order to a) turn on cgi scripts (if not already the case), and b) add a "handler" for <code>.xq</code> files. To do so, I made the following changes:

```
$ diff -w httpd.conf httpd.conf.default
 858c858
 &lt;     AddHandler cgi-script .cgi
 ---
 &gt;     #AddHandler cgi-script .cgi
 885,890d884
 &lt; # redirect xquery files to our cgi script
 &lt; AddType    text/xml .xq
 &lt; AddHandler xquery-type .xq
 &lt; Action xquery-type /cgi-bin/xquery.cgi
```

Don't forget to restart Apache; so it reads httpd.conf

6.5.2 xquery.cgi

In the cgi-bin/ directory, you must also place the script (xquery_cgi) under the name xquery.cgi with executable file permissions:

```
chmod 755 xquery.cgi
```

Beware! You must potentially adapt:

1. the location of bash
2. WWWDIR(the htdocs directory where Apache stores its content)
3. MONETDIR (the installation dir of MonetDB/XQuery)

```
#!/bin/bash
echo 'Content-type: text/xml'
echo

MONETDIR=/path_to_monetdb/
WWWDIR=/var/www/htdocs

XQFILE=$WWWDIR/$REDIRECT_URL

if [ x$QUERY_STRING == x ]
then
  $MONETDIR/bin/mclient --set prefix=$MONETDIR --set exec_prefix=$MONETDIR -fxml -lx $XQFI
else
   `echo "sed -e s/%$QUERY_STRING/g $XQFILE" | sed -e "s/&amp;/\/g -e s\/%/g" -e "s/=/%\//
  $MONETDIR/bin/mclient --set prefix=$MONETDIR --set exec_prefix=$MONETDIR -fxml -lx /tmp/
fi
```

6.5.3 passing parameters

The <code>xquery.cgi</code> script allows parameter passing. Suppose we have the following XQuery file (save it as example.xq in the htdocs>-folder):

```
for $i in 1 to %max%
return element { "mult" } {
  $i, " times ", %table%, " is ", $i * %table%
}
```

The table and max have been parametrized. Parameters take the form: %name% The parameters can be used in the usual URL convention:

```
<a href="http://localhost/example.xq?table=3&max=10"
class='external'>http://localhost/example.xq?table=3&max=10</a>
```

Beware: parameter substitution in this script is **very simple**; it won't work with special characters in it (that get escaped by the web server) or even spaces. For optimal web server performance, we may want to do all parameter substitution inside mclient; so that we don't need to fork a bash shell process on each web request.