MonetDB Reference Manual

Version 5.0

The MonetDB Development Team

This file documents the MonetDB Version 5.0 Reference Manual

Last updated: April 27, 2006

Table of Contents

1 General Introduction

The MonetDB 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 focussed on the features available in Version 5. Users of previous versions may benefit from the sections on e.g. SQL and the inner core of the system, because these components are shared with Version 4.

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. It can also be used to produce a Postscript version using *tex2dvi* followed by *dvips*. The PDF version can be produced from the Postscript using *ps2pdf*. Alternative formats, e.g. XML and DocBook format, can be readily obtained from the Texinfo file.

The copyright(2006) on the MonetDB software, documentation, and logo is owned by CWI. Other trademarks and copyrights referred to in this manual are the property of their respective owners.

Disclaimer The reference manual is still under development, much like the underlying source code base. This may lead to incomplete and inconsistencies descriptions, for which we apologize in advance. You can help improving the manual using the mailing list included in the website.

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.

The route towards becoming a database application expert is long and tedious. It requires building experiences from small toy examples towards large business critical application. The MonetDB reference manual is not designed for this purpose, but in combination with the abundance of tutorial articles on SQL-based systems, it should be possible to quickly create your first working MonetDB/SQL application.

The bulk of the MonetDB reference manual deals with the techniques deployed in the back-end for the expert user and researcher. Judicious use of the programming interfaces and database kernel modules for domain specific tasks lead to high-performance solutions. The grand challenge for the MonetDB development team is to assemble a sufficient and orthogonal set of partial solutions to accomodate a wide variety of front-ends.

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

1.2 How to Read This Manual

The reference manual covers a lot of ground, which at first reading may be slightly confusing. The material is presented in a top-down fashion. Starting at installing the system components, SQL and the application interface layer, it discusses the MAL software stack at length. Forward references are included frequently to point into the right direction for additional information.

First time users of MonetDB should read Section 1.8 [Download and Installation], page 7 and Chapter 3 [SQL], page 26. It prepares the ground to develop applications. Advanced topics for application builders are covered in Chapter 5 [User Interfaces], page 50.

The query language Chapter 4 [XQuery], page 49 is intended for users living at the edge of technology. It provides a functional complete implementation of the XQuery and Xupdate standard. Unfortunately, XQuery compiler is currently only available for MonetDB Version 4.

If you are interested in technical details of the MonetDB system, you should start reading Chapter 2 [MonetDB Overview], page 19. Two reading tracks are possible. The Chapter 7 [MonetDB Assembler Language], page 69 and subsequent sections describe the abstract machine and MAL optimizers to improve execution speed. It is relevant for a better understanding of the query processing behavior and provides an entry point to built new languages on top of the database kernel. The tutorial on SQL to MAL compilation provides a basis for developing your own language front-end.

The second track, The Inner Core describes the datastructures and operations exploited in the abstract machine layer. This part is essential for developers to aid in bug fixing and to extend the kernel with new functionality. Its information covers also the ground for MonetDB Version 4. For most readers, however, it can be skipped without causing problems to develop efficient applications.

1.3 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.

It is important to recognize that the MonetDB language interfaces are primarily aimed at experienced system programmers and administrators. End-users are advised to use any of the open-source graphical SQL workbenches to interact with the system. (Sec Chapter 5 [User Interfaces], page 50)

1.3.1 When to consider MonetDB ?

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 Assembler Language (MAL), it supports ANSI SQL-99 and W3C XQuery. 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 binary-relation database kernel. MonetDB is built on the canonical representation of database containers, namely binary relations. The datastructures are geared towards efficient representation when they mimic a n-ary relational scheme.

This led to an architecture where the traditional page-pool is replaced by one with a much larger granularity, called Binary Association Tables (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 science, 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 for software experts. It proofs valuable in those cases where an application specific and critical component makes all the difference between slow and fast implementation.

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

1.3.2 When not to consider MonetDB ?

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. More mature products 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/C++ based on a persistent object model, is a no-go area for MonetDB. Much like the other database engines, the overhead involved in 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. For one reason it was decided to factor out the ACID techniques and make them 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 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 Assembler 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.

In the multimedia applications we have exploited successfully the inherent data parallelism to speedup processing and reduce the synchronization cost. The underlying platforms were Linux-based cluster computers with sizeable main memories.

1.3.3 What are the MonetDB key features

The list below provides a glimpse on the technical characteristics and features of the MonetDB software packages. For the SQL front-end:

- It is based on the SQL'99 standard core.
- It supports nested queries.
- It supports views.
- It supports sequence types from the SQL'03 standard.

For the XQuery front-end:

- The W3C XQuery standard is fully implemented.
- The XUpdate draft standard is being implemented.

The software characteristics for the MonetDB packages are:

- The kernel source code is written in ANSI-C and POSIX compliant.
- The application interface libraries source code complies with in the latest language versions.
- The source code is written in a literate programming style, to stimulate proximity of code and its documentation.
- The source code is compiled and tests on many platforms with different compiler options to ensure portability.
- The source code is based on the GNU toolkit, e.g. Automake, Autoconf, and Libtool for portability.
- The source code is heavily tested on a daily basis, and scrutinized using the Valgrind toolkit.

The heart is the MonetDB server, which comes with the following innovative features.

- A fully decomposed storage scheme using memory mapped files.
- It supports scalable databases, 32- and 64-bit platforms.
- Connectivity is provided through TCP/IP sockets on many platforms.
- Index selection, creation, and maintenance is automatic.

- The relational operators materialize their results and are self-optimizing.
- The operations are cache- and memory-aware with supreme performance.
- The database backend is multi-threaded and guards a single physical database instance.

1.3.4 Size limitations for MonetDB

The maximal database size supported by MonetBD depends on the underlying processing platform, i.e. a 32- or 64-bit processor, and storage device, i.e. the file system and disk raids.

The number of columns per tables is practically unlimited. Unlike traditional database systems, the storage space limitation depend on the maximal size for an individual column. Each column is mapped to a file, whose limit is dictated by the operating system and hardware platform.

The number of concurrent user threads is a configuration parameter.

1.4 A Brief History of MonetDB

The Dark Ages [1979-2002] The development of the MonetDB software family goes back as far as the early eighties when the first relational kernel, called Troll, was delivered to a larger audience. It was spread over ca 1000 sites world-wide and became part of a software case-tool until the beginning of the nineties. None of the code of this system has survived, but several ideas and experiences on how to obtain a fast kernel by simplification and explicit materialization found their origin during this period.

The second part of the eighties was spent on building the first distributed main-memory database system in the context of the national project PRISMA. A fully functional system of 100 processors and a wealthy 1GB of main memory showed the road to develop database technology from a different perspective. A more detailed account on the experiences can be found in REF.

The Early Days [1993-1995] Immediately after the PRISMA project was termed dead, the basis for a new database kernel based on Binary Association Tables (BATs) was laid out. The original target was to aim for better support of scientific databases with their then archaic file structures.

The Data Distilleries Era [1996-2003] The datamining projects running as of 1993 called for better database support. It culminated in the spin-off Data Distilleries, which based their analytical customer relationship suite on the power provided by the early MonetDB implementations. In the years following, many technical innovations were paired with strong industrial maturing of the software base.

The Open-Source Challenge [2003-2006] Moving MonetDB Version 4 into the open-source field required a large number of extensions to the code base. It became utmost important to support a mature implementation of the SQL99 standard, and the bulk of application programming interfaces (PHP,JDBC,Perl,ODBC). The culprit of this activity was the first official release in 2004 and the release of the XQuery front-end in 2005.

The Road Ahead [2006- This manual marks the alpha-release of MonetDB Version 5, the result of a multi-year activity to clean up the software stack and to better support for simple database requests. (See Section 2.7 [Backward Compatibility], page 25)

The future: X100, Armada New versions in the MonetDB software family are under development. Extensions and renovation of the kernel are tackled in the X100 project, which provides high-volume access to columns stored in compressed form. Its volcano-style interpreter aims to provide performance in I/O-dominant and streaming settings using vectorized processing and Just-In-Time (de)compression.

The scene of distributed database is (again) addressed in the Armada project, but not using the traditional centralized administration focus. Instead the Armada project seeks the frontiers of autonomous database systems, which still provide a coherent functional view to its users. In its approach it challenges many dogmas in distributed database technology, such as the perspective on global consistency, the role of the client in managing the distributed world, and the way resources are spread.

The MonetDB software framework provides a rich setting to pursue these allees of database research. We hope that many may benefit from our investments, both research and business wise.

1.5 Manual Generation

The MonetDB code base is 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 also process the code.

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

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

The result, however, still includes the unexpanded tex info macros.

1.6 Conventions and Notation

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 verbatim font. The interaction with textual client interfaces uses the default prompt-setting of the underlying operating system.

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

1.7 Additional Resources

Although this reference manual aims to be complete for developing applications with MonetDB, it also depends on additional resources for a full understanding.

This reference manual relies on external documentation for the basics of its query languages SQL, XQuery, its application interfaces, PHP, Perl, Pyhton, and its middleware support, JDBC and ODBC. Examples are used to illustrate their behaviour in the context of MonetDB only. The resource locations identified below may at times proof valuable.

Perl DBI	http://www.perl.org/
PHP5	http://www.php.net/
Python	http://www.python.org/
XQuery	http://wwww.w3c.org/TR/xquery/

The primary source for additional information is the MonetDB website, http://monetdb.cwi.nl/, and the code base itself. Information on the background of its architecture can be found in the library of scientific publications.

1.8 Downloads and Installation

Using MonetDB for experimentation and application development requires installation of the back-end server, a front-end compiler for SQL or XQuery, the APIs, and a (graphical) user interface. These components are packaged conveniently for several platforms in the download section at SourceForge.

Most people prefer a standard distribution, 'officially' declared as stable. Stable means that special care has been taken to assure that errors reported during the nightly builds have been solved on the platforms of interest. Major bug fixes are also applied to the latest stable version, while functional enhancements are kept for the next release or the daily builds.

If you encounter errors during the installation, please have a look at the FAQ and MonetDB mailing list for common errors and some advice on how to proceed.

1.8.1 Software Versions

The MonetDB product family consists of several related packages. At the time of writing Version 4.10 is distributed through SourceForge. It comes with the SQL version 2.10 and PathFinder 0.10 front-ends packages.

This manual, however, is written in the context of preparing the launch of MonetDB Version 5. It works with the SQL version 2.10, but there is currently no XQuery front-end available.

From its inception, the MonetDB software has been designed to meet the highest level of standards regarding software portability. The interface to the operating system is based on the POSIX standard. The default C-compilation options are strict, considering all compiler warnings as fatal. String and memory manipulation are all guarded against overflows. This with multi-year deployment in a commercial setting makes the code base robust.

Thanks to the GNU tools `autoconf` and `automake` tools, the MonetDB software runs on a wide variety of hardware/software platforms. A summary of those operating systems and the compiler toolkits is shown below.

Operating System	Word size	Compilers
Microsoft	32 bits	XP, Server 2003
Cygwin (Windows)	32 bits	GNU
Gentoo (Linux 2.6.14)	32 bits	GNU
Fedora Core 4	64 bits	GNU, Intel
Fedora Core 3	64 bits	GNU, Intel
RedHat EL WS	64 bits	GNU, Intel
Debian 3.0	64 bits	GNU, PGI
Suse 9.3	64 bits	GNU, Intel
MacOS X 10.4 (Darwin 8.2.0)	32 bits	GNU
AIX 5.1	32 bits	GNU, IBM
IRIX 64 6.5	32,64 bits	GNU, SGI
Solaris 8 (SunOS 5.8)	32,64 bits	GNU, Sun
OpenZaurus	32 bits	GNU (cross)
Gumstix	32 bits	GNU (cross)
LinkStation	32 bits	GNU (cross)

The MonetDB development team uses many of these platforms to perform automated nightly regression testing. For more details see pxref(The Test Web).

1.8.2 Standard Distribution

The standard distribution is meant for users primarilly interesting in building SQL or XQuery applications. They should obtain the pre-packaged binary distribution from the download section at SourceForge. The system can be installed in a private directory for experimentation or in the Linux/Windows compliant default folder location.

The choice between Version 4.9 and 5.0 should be planned carefully. Both engines provide the same SQL functionality, but differ greatly under the hood. Version 4.9 is based on the MIL scripting language, which is known to be slow, and provides limited support for query optimization and program development (e.g. a debugger). The MIL language becomes depreciated as soon as the XQuery compiler has been ported. Contrary, Version 5.0 provides an assembler like scripting language, geared at supporting front-end apllication code generation. It is not a language for programmers to write their applications on a daily basis. This simplicity greatly enhances the parsing and interpretation performance. Especially, small SQL queries run more than twice as fast. For more details see Section 2.1 [Design considerations], page 19

The MonetDB code base evolves quickly with daily builds available for users preferring living at the edge. Application developers, however, may tune into the MonetDB mailing list to be warned when a major release has become available, or when detected errors require a patch.

Before you embark upon application development, take the quick tour from the tutorial section. It illustrates a small, but concrete application scenario geared at querying a historical database with trading trips of the world-famous East-Indian Trading Corp (1602-1795).

1.8.3 Source Distribution

The source distribution is needed if you intend to inspect or extend the code base. The source distribution comes with the complete test-bench to assure that changes do not affect (in as far as they get tested) its stability. A single stable release is maintained for external users while we concurrently work on the next release. Older versions are not actively maintained by the development team.

Set up of a fully functional system requires downloading a MonetDB server package and either/both the SQL and XQuery packages. The steps to be taken are described for installing the server only, because installing the other packages largely follow the same steps. The deviances are explained at the end of this section.

The development version can be obtained from the CVS repository at SourceForge. You have to login to the CVS server first:

```
cvs -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/monetdb login
```

Just type RETURN when asked for the password. Then get the MonetDB and SQL module sources by using the command:

```
cvs -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/monetdb co buildtools MonetDB5 sql
```

This will create the directories buildtools, MonetDB5 and sql in your current working directory, which hold the sources for all three project packages. See the readme file to install the buildtools. MonetDB developers should use `ext` instead of `pserver`, e.g.

```
cvs -d :ext:<username>@cvs.sf.net:/cvsroot/monetdb checkout MonetDB5 sql
```

Also see SourceForge documentation for more information on using cvs.

1.8.4 Linux Installation

This section helps you to install the MonetDB source and compile it on a Unix platform (e.g., Linux, MacOS X/Darwin, or CYGWIN).

1.8.4.1 Prerequisites

- *CVS* You only need this if you are building from CVS. If you start with the source distribution from SourceForge you don't need CVS. For instructions, see docman and look under the heading CVS Instructions.
- *Python* MonetDB uses Python (version 2.0.0 or better) during configuration of the software. See python.org for more information.
- *autoconf/automake/libtool* MonetDB uses GNU autoconf (>= 2.53), automake (>= 1.5), and libtool (>= 1.4) during configuration of the software. Libtool is also used during the build.
- *standard software development tools* To compile MonetDB, you also need to have a the following standard software development tools installed and ready for use on you system:
  - a C/C++ compiler (e.g., GNU's gcc/g++)</li>
  - a lexical analyzer generator (e.g., "lex" or "flex")</li>
  - a parser generator (e.g., "yacc" or "bison")</li>
  - GNU make ("gmake") (native make's on, e.g., IRIX and Solaris usually don't work).

- Mx, mel, and autogen These tools are only needed if you are building from CVS. If you start with the source distribution from SourceForge you don't need this.
- *disk space* After downloading, MonetDB takes up about 40 MB of disk space; compilation adds another 70 MB. Testing takes about 45 MB, and the installation about 20 MB (not including any databases).

1.8.4.2 Bootstrap, Configure and Make

Before executing the following steps, make sure that your shell-environment (especially variables like "PATH" and "LD_LIBRARY_PATH") is setup, thus that the tools listed above can be found.

In the top-level directory of MonetDB type the command:

```
./bootstrap
```

Then in any directory (preferably a *new empty* directory and *not* in the MonetDB top-level directory) give the command

```
<path-to>/configure [<options>]
```

where `<path-to>` is replaced with the (absolute or relative) path to the MonetDB top-level directory. The directory where you execute configure is the place where all intermediate source and object files are generated during compilation via "make".

By default, MonetDB is eventually (i.e., during "make install"; see below) installed to /usr/local/. To choose another target directory, you need to call

```
.../configure --prefix=<prefixdir> [<options>]
```

A sample of other useful configure options is:

```
--enable-debug   enable full debugging default=off
--enable-optimize   enable extra optimization default=off
--enable-warning   enable extended compiler warnings default=off
--enable-profile   enable profiling default=off
--enable-instrument   enable instrument default=off
--with-gcc=[compiler]   which C compiler to use
                         ("yes" = gcc/g++,
                          "no"  = system-specific C/C++ compiler,
                          &lt;other-compiler-name&gt;)
--with-gxx=[compiler]   which C++ compiler to use
--with-bits=[#bits]   specify number of bits (32 or 64)
```

Use the `--help` option of configure to find out more about configure options.

In the same directory (where you called configure) give the command

```
make
```

to compile the source code.

On a lightly loaded, 1.4 GHz Athlon Linux system, configure takes about 30 seconds, and make about 5 to 10 minutes, depending on the optimization level chosen during configure.

1.8.4.3 Testing the Build

If make went successfully, you can try

```
make check
```

This will do a lot of tests, some are unfortunately still expected to fail, but most should go successfully. At the end of the output there is a reference to an HTML file which is created by the test process and shows the test results.

Testing takes about 46 MB of disk space in the build directory. Running the tests takes about 13 minutes on the previously mentioned system.

1.8.4.4 Installing

Give the command

```
make install
```

By default (if no –prefix option was given to configure above), this will install in /usr/local/. Make sure you have appropriate privileges.

1.8.4.5 Testing the Installation

Make sure that /usr/local/bin (respectively <prefixdir>/bin, where <prefixdir> is the directory you specified with `--prefix` when calling configure) is in your PATH.

In the MonetDB top-level directory issue the command

```
Mtest.py -r
```

This should produce much the same output as make check above, but uses the installed version of MonetDB.

You need write permissions on part of the installation directory for this command: it will create subdirectories `var/MonetDB5/dbfarm` and Tests.

1.8.4.6 Documentation

The documentation recides in `share/MonetDB5/`. It consists of the pdf files of the current document and its web pages. The most recent version is best access from the MonetDB home page.

1.8.4.7 Troubleshooting

Bootstrap fails if any of the requisite programs can not be found or is an incompatible version. Bootstrap adds files to the source directory, so it must have write permissions.

During bootstrap, warnings like

```
Remember to add ‘AC_PROG_LIBTOOL’ to ‘configure.in’.
You should add the contents of ‘/usr/share/aclocal/libtool.m4’ to ‘aclocal.m4’.
configure.in:37: warning: do not use m4_patsubst: use patsubst or m4_bpatsubst
configure.in:104: warning: AC_PROG_LEX invoked multiple times
configure.in:334: warning: do not use m4_regexp: use regexp or m4_bregexp
automake/aclocal 1.6.3 is older than 1.7.
Patching aclocal.m4 for Intel compiler on Linux (icc/ecc).
patching file aclocal.m4
Hunk #1 FAILED at 2542.
1 out of 1 hunk FAILED -- saving rejects to file aclocal.m4.rej
patching file aclocal.m4
Hunk #1 FAILED at 1184.
Hunk #2 FAILED at 2444.
Hunk #3 FAILED at 2464.
3 out of 3 hunks FAILED -- saving rejects to file aclocal.m4.rej</pre>
might occur.
```

For some technical reasons, it's hard to completely avoid them. However, it is usually save to ignore them and simply proceed with the usual compilation procedure. Only in case the subsequent configure or make do fail, these warning might have to be take more serious; in any case, you should include the e bootstrap output in whenever you report (see below) compilation problems.

Configure will fail if certain essential programs can't be found or certain essential tasks (such as compiling a C program) can't be executed. The problem will usually be clear from the error message.

E.g., if configure cannot find package XYZ, it is either not installed on your machine, or it is not installed in places that configure usually searches (i.e., `/usr`, `/usr/local`). In the first case, you need to install package XYZ before you can configure, make, and install MonetDB. In the latter case, you need to tell configure via "–with-XYZ=<DIR>" where to find packages XYZ on your machine. configure then looks for the respective header files in `<DIR>/include`, and for the respective libraries in `<DIR>/lib`.

Bugs and other problems with compiling or running MonetDB should be reported using the respective bug-tracking system at SourceForge (preferred) or emailed to monet@cwi.nl.

1.8.5 Windows Installation

This section helps you to install MonetDB on a native Windows system (NT, 2000, XP).

1.8.5.1 Prerequisites

- *CVS* You need to have a working CVS. Several solutions are available. We use internally WinCVS and CVS under Cygwin. For general information about the SourceForge repository, see docman and look under the heading CVS Instructions. Pointers to CVS implementations for Windows can be found at e.g.: http://www.cvshome.org/cyclic/cvs/windows.html http://www.wincvs.org/ http://www.componentsoftware.com/products/CVS/
- *Python* MonetDB uses Python (version 2.0.0 or better) during configuration of the software. See python.org for more information.
- *Pthreads* Pthreads for Win32 should be installed to C:\Pthreads, otherwise you need to patch the "PTHREAD = C:\Pthreads" line in "NT\rules.msc" according to your setup. Newer versions of Pthreads seem to have the "include" &amp; "lib" directory not in C:\Pthreads, but in C:\Pthreads\prebuilt. In that case, you either have to move the "include" & "lib" directory from C:\Pthreads\prebuilt to C:\Pthreads, or set "PTHREAD = C:\Pthreads\prebuilt" in "NT\rules.msc".
- *UnxUtils* UnxUtils for Win32 must be installed in the root directory ("\", i.e., without the default "\UnxUtils\" prefix) of the drive where you want to compile MonetDB. The same drive is required, as the UnxUtils do not know about drive letters, and hence absolute paths start with "\" (i.e., without a leading "C:", "D:", ...). The default "\UnxUtils\" prefix has to be omitted, as otherwise the UnxUtils' "bison" does not find its own "bison.simple" file, which it expects to be in "/usr/share/bison/bison.simple" (aka. "\usr\share\bison\bison.simple"). Next to some tools used by Mtest.py for testing MonetDB (see below), UnxUtils provide the lexical analyzer generator "flex" and the parser generator "bison", which are required to compile MonetDB. Note: You have to install the UnxUtils using the original .zip file. With the third-party executable installer, choosing another than the default installation directory doesn't seem to work...
- *Microsoft Visual C++* You need Microsoft Visual C++ 5 or higher. Notice that if you do not own Microsoft Visual C++, you can still compile on Windows using the Cygnus Posix-emulation environment CYGWIN In that case, you should follow the standard instructions in the file 'HowToStart'. Alternatively, it might be possible to use the free WIN32-GNU compilers as a drop-in replacement for the Microsoft tools, but we have not tried this yet.
- *disk space* After downloading, MonetDB takes up about 40 MB of disk space; compilation adds another 70 MB. Testing takes about 45 MB, and the installation about 20 MB (not including any databases).

1.8.5.2 Compiling

Open a Windows command shell or a UnxUtils shell and go to the top-level directory of MonetDB. Make sure that the proper environment for MSVC++ is set, otherwise call the corresponding BAT file. (see C:\Program Files\Microsoft Visual Studio\VC98\Bin\VCVARS32.BAT). Make sure that Python, C:\Pthreads\lib (or wherever you installed Pthreads), and \usr\local\wbin are in the PATH of your command shell. When all prerequisites have been satisfied, you type

```
cd NT
nmake
```

to compile the source code. If nmake went successfully, you can try

```
nmake check
```

This will do a lot of tests, some are unfortunately still expected to fail, but most should go successfully. At the end of the output there is a reference to an HTML file which is created by the test process and shows the test results. An explanation of the test results can be found in XXX–to be filled in. Testing takes about 46 MB of disk space in the build directory.

1.8.5.3 Testing the Installation

Make sure that <sourcedir>\NT\bin, <sourcedir>\NT\lib, and <sourcedir>\NT\lib\MonetDB are in your PATH. In the MonetDB top-level directory issue the command

```
Mtest.py -r
```

(Make sure Mtest.py can be found, it should be in the bin directory that was filled by the nmake install command.)

This should produce much the same output as nmake check above, but uses the installed version of MonetDB.

1.8.5.4 Documentation

The documentation starts in <sourcedir>\NT\doc\MonetDB.html

NOTE: the current 'nmake install' does not yet generate this documentation to do this manually, execute *after* successful 'make install'. In the top-level directory of the MonetDB build tree, call

```
nmake html
```

1.8.6 Gentoo Installation

This section helps you to install the MonetDB source using Gentoo on all platforms, MacOs in particular. [To be documented by Fabian]

1.8.7 Daily Builds

Next to functionality and performance, stability and portability are first class goals of the MonetDB project. Pursuing these goals requires to constantly monitor stability and portability of the evolving MonetDB code base. For this purpose, we developed a test environment that automatically compiles and tests MonetDB (and its most prominent add-on packages) every night on a variety of system configurations.

Software patches and functional enhancements are checked into the repositories on a daily basis. A limited set of distribution packages is prepared to disseminate the latest to developers and application programmers as quickly as possible. Such builds may, however, contain bugs or sometimes even break old functionality, mostly due to incompatibility of compiler code.

The TestWeb provides access to the test web-site that summarizes the results of the Automated Testing activities on various platforms. It is a good starting point before picking up a daily build version.

Two versions of MonetDB are tested daily on all available platforms:

- the *cutting edge* development version ("Current"), i.e. the head of the main CVS branch; and
- the latest release version ("Stable"), i.e. the head of the most recent release branch.

The test reports consist of three overview pages ("cross-check-lists") revealing the results of

1. all compilation steps (bootstrap, configure, make, make install),
2. testing via "make check" (using debugmask 10, i.e., exhaustive monitoring and correction of physical BAT properties is enabled in the server), and
3. testing via "Mtest.py -d0 -r" (using debugmask 0, i.e., all debugging is switched off in the server).

1.8.7.1 Stability

With a (code-wise) complex system like MonetDB, modifying the source code — be it for fixing bugs or for adding new features — always bears the risk of breaking or at least altering some existing functionality. To facilitate the task of detecting such changes, small test scripts together with their respective correct/expected ("stable") output are collected within the CVS repository of MonetDB. Given the complexity of MonetDB, there is no way to do anything close to "exhaustive" testing, hence, the idea is to continuously extend the test collection. E.g., each developer should add some tests as soon as she/he adds new functionality. Likewise, a test script should be added for each bug report to monitor whether/when the bug is fixed, and to prevent (or at least detect) future occurrences of the same bug. The collection consists for hundreds of test scripts.

To run all the tests and compare their current output to their stable output, a tool called Mtest is included in the MonetDB code base. Mtest recursively walks through the source tree, runs tests, and checks for difference between the stable and the current output. As a result, Mtest creates a web interface that allows convenient access to the differences encountered during testing. Each developer is supposed to run "Mtest" (respectively "make check") on his/her favorite development platform and check the results before checking in her/his changes. During the automatic daily tests, "make check" and "Mtest" are run on all testing platforms and the TestWeb is generated to provide convenient access to the results.

1.8.7.2 Portability

Though Fedora Linux on AMD Athlon PC's is our main development platform at CWI, we do not limit our attention to this single platform. Supporting a broad range of hardware and software platform is an important concern.

Using standard configuration tools like automake, autoconf, and libtool, we have the same code base compiling not only on various flavors of Unix (e.g., Linux, Cygwin, AIX, IRIX, Solaris, MacOS X) but also on native Windows. Furthermore, the very code base compiles with a wide spectrum of (C-) compilers, ranging from GNU's gcc over several native Unix compilers (IBM, SGI, Sun, Intel, Portland Group) to Microsoft's Visual Studio and Visual Studio .NET on Windows.

On the hardware side, we have MonetDB running on "almost anything" from a Intel StrongARM-based Linux PDA with 64 MB of flash memory to an SGI Origin2000 with 32 MIPS R12k CPU's and a total of 64 GB of (shared) main memory.

1.9 Start and Stop the Server

Starting and stopping the server under Windows doesn't need an extensive description. Just locate the server in the program list and start it. After the server has been started, you can activate a textual client interface. Close the window of the server and/or client and it ceases to exist.

Once downloaded and installed MonetDB on your Linux system, it is adviceable to check its functionality with the following actions:

```
> monetdb --help
```

The `monetdb` script is a wrapper around the MonetDB server program `mserver5` and its protection program `mguardian`. The call triggers `mserver5 --help` and responds with something like:

```
Usage: monetdb [command] [options] [script]
Primary command:
    --status        Show the server status [default]
    --start         Server(s) is (are) started
    --stop          Server(s) is (are) stopped
    --databases     Show the known databases
    --checkpoint    Create the checkpoint for a database
    --recover       Restore a database to an earlier date
    --log           Show the server management log
```

```
Secondary options:
    --dbname=<database_name>
    --dbfarm=<directory>
    --dbinit=<stmt>           Server prepare statement
    --config=<config_file>    Configuration file
    --debug=<number>          Trace server actions[0]
    --daemon=yes|no           Run in background [no]
    --set <option>=<value>    Set environment value
    --ascii                   Use ascii dump for the database [default=no]
    --help                    This list of options
```

An error messages typically indicate a wrongly configured $PATH variable, making the `monetdb` program invisible to your shell. Retrace your installation steps.

An access mode violation indicates that either the directory path to the location of the MonetDB data store does not exists, or the user has lack of file system permissions to create new directories/files in the location specified.

To start a server for a database simpy type

```
> monetdb --start --dbname=voc
Do you want to create database 'voc' ? [yes/no]yes
Database xyz initialized
Starting database server 'voc'... ok
```

The question is asked only the first time you attempt to access the `voc` database.

It responds with a message that this action was successful, or an error message explains the problems encountered. The server is run as a daemon process and you need a MonetDB client program to connect to it. More details on the server configuration parameters are given in Section 2.2 [Architecture overview], page 20.

At any time you can inspect the status of the (running) servers using the default command option `--status`.

```
>monetdb --status
DBNAME  SERVER  GUARD   DELAY   STARTED
voc     21488   21502   12      Sun Dec 18 09:05:36 2005
```

It tells you when the server was started, the identity of the process looking after it, and the delay between successive checks. A server can be put to sleep using the command `monetdb --stop --dbname=<name>`. Ommission of a database name stops all running servers. The actions of monetdb are logged for post analysis, which can be inspected with `monetdb --log`

1.9.1 Database Configuration

The database environment is collected in a configuration file, which is used by server-side applications, such as `mserver5` and `mguardian`. A default version is installed in the database store upon its creation using the command `monetdb`. Below we illustrate its most important components.

- prefix=/ufs/myhome/monet5/Linux
- exec_prefix=${prefix}

- dbfarm=${prefix}/var/MonetDB5/dbfarm
- monet_mod_path=${exec_prefix}/lib/MonetDB5
- checkpoint_dir=${prefix}/var/MonetDB5/chkpnt
- dbname=demo
- version=5.0
- welcome=yes
- embedded=no
- gdk_debug=0 # to control level of debugging

The header consist of system wide information. The `prefix` and `exec_prefix` describe the location where MonetDB has been installed. `monet_mod_path` tells where to find the libraries. These arguments are critical for a proper working server.

The remainder consists of arguments used by functional modules, or related tools. Consult the documentation before changing them.

- delay=120
- mal_init=${prefix}/lib/MonetDB5/mal_init.mal
- mal_listing = 7
- sql_debug=0
- sql_logdir=${prefix}/var/MonetDB5/dblogs
- pf_httpd_port=8080
- xquery_output=dm
- xquery_cacheMB=100

1.9.2 Checkpoint and Recovery

Safeguarding the content of your database requires carefully planned steps. The easiest way is to shutdown the server first. Then the database directory holding the database can be compressed and stored away. You may want to create a tarball with compressed files and name them clearly for later recall.

The checkpoint is not complete without the corresponding log files produced by SQL. They are stored by default in a mirror directory of `dbfarm`, called `dblogs`. They too should be picked up and safeguarded for future use.

The checkpoint and recover operations are packaged as shell scripts, which are wrapped by the `monetdb` script. Taking a checkpoint and its recovery is as simple as:

```
shell>monetdb --checkpoint --dbname=demo
Checkpoint store .../var/MonetDB5/chkpnt
Preparing checkpoint file 'demo/2006-01-02-223704'
Preparing checkpoint log file 'demo/2006-01-02-223704-logs'

shell>monetdb --recover --dbname=demo
Checkpoint tags defined for 'demo'
2006-01-02-224335
2006-01-02-224233
...
```

```
Specify the checkpoint tag for 'demo' ? 2006-01-02-224335
Move existing database out of the way 'demo2006-01-02-224416'
Move existing database logs out of the way 'demo2006-01-02-224416'
Restore from store .../var/MonetDB5/chkpnt
Reload checkpoint file 'demo/2006-01-02-224335'
Reload checkpoint log file 'demo/2006-01-02-224335-logs'
Database recovery finished
```

Before the checkpoint operation starts the database server is brought down and restarted afterwards. The checkpoint files are stored away in a path available in the database configuration file.

1.9.3 Database Dumps

An alternative scheme to safeguard and transport a SQL database is to produce an `ascii`-based database dump. This option is standard available for the `mclient` and `mjClient`. However, be prepared that not all SQL schema features may be retained in the process. It also takes much more time to produce the dump file.

Consider you have already installed the SQL tutorial database `VOC` and wish to transport it to another machine. The following steps are required after you have started the database server.

```
shell> mclient -lsql --dump >/tmp/voc.sql
```

You can inspect the file `/tmp/voc.sql` to confirm that indeed a compact database dump is available. Move this file over to the new machine. The `monetdb` script can be used to create the database and start the server. Once done, it suffices to feed the dump file to a MonetDB client program to populate the database.

```
shell> monetdb --start --dbname=voc
Do you want to create database 'voc' ? [yes/no]yes
Database voc initialized
!WARNING: GDKlockHome: ignoring empty or invalid .gdk_lock.
!WARNING: BBPdir: initializing BBP.
# Monet Database Server V4.99.19
# Copyright (c) 1993-2006, CWI. All rights reserved.
# Compiled for x86_64-redhat-linux-gnu/64bit with 32bit OIDs; dynamically linked.
# config:/ufs/mk/monet5/Linux/var/MonetDB5/dbfarm/voc/.monetdb.conf
# dbfarm:/ufs/mk/monet5/Linux/var/MonetDB5/dbfarm
# dbname:voc
# Visit http://monetdb.cwi.nl/ for further information.
#include sql; mapi.listen();
shell> mclient -lsql /tmp/voc.sql
```

The dump facility is also avaible as an option to the `monetdb` script. It tags the dump and stores it away in the checkpoint directory.

2 Introduction to MonetDB Version 5

The MonetDB product family consists of a large number of components developed within our group over the last decade. Some components have already been shipped to happy customers, some are still in the making, and others have found a resting place in the attic.

The MonetDB architecture is designed to accommodate a wide-spectrum of standardized query language front-ends (SQL, XQuery), a variety of query transformation schemes, and different execution platforms (interpreted materialized or pipelined, dynamic compilation).

MonetDB Version 5 is a major release of our software infrastructure. The most notable differences are its greatly improved software stack and a new interface language, which turns the database server back-end into an abstract database machine with its associated assembler language (MAL). It supports backward compatibility of interfaces, tools, and source sharing where feasible within the limited scope of resources available.

In the remainder of this section we shortly introduce the MonetDB Version 5 design considerations and a quick overview of the its architecture.

2.1 Design considerations

Redesign of the MonetDB software stack was driven by the need to reduce the effort to extend the system into novel directions and to reduce the Total Execution Cost (TEC). The TEC is what an end-user or application program will notice. The TEC is composed on several cost factors:

- A) API message handling
- P) Parsing and semantic analysis
- O) Optimization and plan generation
- D) Data access to the persistent store
- E) Execution of the query terms
- R) Result delivery to the application

Choosing an architecture for processing database operations pre-supposes an intuition on where and how the cost will be distributed. In an OLTP setting you expect most of the cost to be in (P,O), while in OLAP it will be (D,E,R). In a distributed setting the components (O,D,E) are dominant. Web-applications would focus on (A,E,R).

Such a simple characterization ignores the wide-spread differences that can be experienced at each level. To illustrate, in D) and R) it makes a big difference whether the data is already in the cache or still on disk. With E) it makes a big difference whether you are comparing two integers, evaluation of a mathematical function, e.g. Gaussian, or a regular expression evaluation on a string. As a result, intense optimization in one area may become completely invisible due to being overshadowed by other cost factors.

The Version 5 infrastructure is designed to ease addressing each of these cost factors in a well-defined way, while retaining the flexibility to combine the components needed for a particular situation. This results in an architecture to assemble the components for a particular application domain and hardware platform.

The primary interface to the database kernel is still based on the exchange of text in the form of queries and simply formatted results. This interface is designed for ease of interpretation, versatility and flexible to accommodate system debugging and application tool development. Although a textual interface potentially leads to a performance degradation, our experience with earlier system versions showed that the overhead can be kept within acceptable bounds. Moreover, a textual interface reduces the programming effort otherwise needed to develop test and application programs.

2.2 Architecture overview

The architecture is built around three independent components: the MonetDB server, the MonetDB guardian, and the client application. The MonetDB server is the heart of the system, it manages a single physical database on one machine for all (concurrent) applications. The guardian program works along side a single server, keeping an eye on its behavior. If the server accidently crashes, it is this program that will attempt an automatic restart. Server and guardian are managed with the `monetdb` script, introduced in Section 1.9 [Start and Stop the Server], page 15.

The top layer consists of applications written in your favorite language. They provide both specific functionality for a particular product, e.g. Proximity, and generic functionality, e.g. the Aquabrowser. The applications communicate with the server using a de-facto standard interface packaged as a library e.g. JDBC, ODBC, Perl, PHP, etc..

The prime query language processors available in the MonetDB family are SQL and XQuery. The former supports the core functionality of SQL:1999 and extends into SQL:2003. The latter is based on the W3C standard and includes the XUpdate functionality. The query language processors each manage their own private catalog structure. Software bridges, e.g. import/export routines, are used to share data between language paradigms.

For each a language specific, strategic optimizer is included. It uses knowledge on the language semantics and general heuristics, such as pushing down selections through joins, to derive better programs.

2.3 MonetDB Assembler Language (MAL)

The target language for the query compilers is an assembler-like language, called the MonetDB Assembler Language (MAL). It is a simplified subset of the Monet Interface Language introduced previously. The language provides a direct the textual interface to the server back-end, i.e. MAL is a language available to applications as well.

Its design is geared to provide a controlled setting for the query processors. They can produce algebraic representations of query plans, which by the MAL optimizers are turned into physical execution plans. In this process an execution paradigm is chosen to best fit the engines available and application needs.

The *box* container model for MAL objects is a balance between hardwired decisions based on the typing scheme and hooks to implement those as user-defined extensions. For example, objects can be assembled in *box*es, which come with a simple management protocol. Their implementation can be geared towards any management style required, e.g. delivery of read-only objects for query processing, transparency to external resources through wrappers, and high-volume transaction support.

Much in the spirit of Version 4 MAL can be readily extended with user defined types and service modules. Furthermore, the language design is meant to support the full breath of computational paradigms deployed in a database setting. In particular, it should provide a clean infrastructure to implement triggers and stream-based database applications. Its design and implementation takes the functionality a significant step further. To name a few:

- All instructions are strongly typed before being executed.
- Polymorphic functions are supported. They act as templates that produce strongly typed instantiations when needed.
- Function style expressions where each assignment instruction can receive multiple target results; it forms a point in the dataflow graph.
- Co-routines (Factories) support building streaming applications.
- Properties are associated with the program code for ease of optimization and scheduling.

2.3.1 MonetDB bootstrap

Startup of the MonetDB server leads to loading the system kernel libraries as defined in a bootstrap script `mal_init.mx`. Its default location is decribed in the MonetDB configuration file. Failure to find the startup-file terminates the session. The global symbol table is initialized with function signatures, and the pre-compiled commands and pattern code blocks are loaded. The libraries may be dynamically loaded by default. Expect tens of modules and hundreds of operations to become readily available.

Modules once loaded can not be dropped without restarting the server. The rational behind this design decision is that a dynamic load/drop feature is often hardly used and severely complicates the code base. In particular, upon each access to the global symbol table we have to be prepaired that concurrent threads my be actively changing its structure. Especially, dropping modules may cause sever problems by not being able to detect all references kept around. This danger required all accesses to global information to be packaged in a critical section, which is known to be a severe performance hindrence.

2.4 Session Scenarios

In MonetDB multiple languages, optimizers, and execution engines can be combined at run time to satisfy a wide user-community. Such an assemblage of components is called a *scenario* and consists of a *reader*, *parser*, *optimizer*, *tactic scheduler* and *engine*. These hooks allow for both linked-in and external components.

The languages supported are SQL, XQuery, and the Monet Assembler Language (MAL). The default scenario handles MAL instructions, which is used to illustrate the behavior of the scenario steps.

The MAL reader component handles interaction with a front-end to obtain a string for subsequent compilation and execution. The reader uses the common stream package to read data in large chunks, if possible. In interactive mode the lines are processed one at a time.

The MAL parser component turns the string into an internal representation of the MAL program. During this phase semantic checks are performed, such that we end up with a type correct program.

The code block is subsequently sent to an MAL optimizer. In the default case the program is left untouched. For other languages, the optimizer deploys language specific code transformations, e.g. foreign-key optimizations in joins and volume reduction over intermediates. All optimization information is statically derived from the code blocks and possible catalogues maintained for the query language at hand. Optimizers leave advice and their findings in properties in the symbol table, see Chapter 10 [Property management], page 82.

Once the program has thus been refined, the MAL scheduler prepares for execution using tactical optimizations. For example, it may parallelize the code, generate an ad-hoc user-defined function, or prepare for efficient replication management. In the default case, the program is handed over to the MAL interpreter without any further modification.

The final stage is to choose an execution paradigm, i.e. interpretative (default), compilation of an ad-hoc user defined function, dataflow driven interpretation, or vectorized pipe-line execution by a dedicated engine.

A failure encountered in any of the steps terminates the scenario cycle. It returns to the user for a new command.

2.4.1 Scenario management

Scenarios are captured in modules; they can be dynamically loaded and remain active until the system is brought to a halt. The first time a scenario is used, the system looks for a scenario initialization routine `XYZinitSystem()` and executes it. It is typically used to prepare the server for language specific interactions. Thereafter its components are set to those required by the scenario (e.g. `"SQL"`) and the client initialization takes place.

When the last user interested in a particular scenario leaves the scene, we activate its finalization routine calling `XYZexitSystem()`. It typically perform cleanup, backup and monitoring functions.

A scenario is interpreted in a strictly linear fashion, i.e. performing a symbolic optimization after a parallelization is not permitted. The routines associated with each state in the scenario may patch the code so as to assure that subsequent execution can use a different scenario, e.g. to handle dynamic code fragments.

The state of execution is maintained in the scenario record for each individual client. Sharing this information between clients should be dealt with in the implementation of the scenario managers. Upon need, the client can postpone a session scenario by pushing a new one(language, optimize, tactic, processor). Propagation of the state information is encapsulated a scenario2scenario() call. Not all transformations may be legal.

The building blocks of scenarios are routines obeying a strict name signature. They require exclusive access to the client record. Any specific information should be accessible from there, e.g. access to a scenario specific state descriptor. The client scenario initialization and finalization brackets are `XYZinitClient()` and `XYZexitClient()`.

The `XYZparser(Client c)` contains the parser for language XYZ and should fill the mal program block associated with the client record. The latter may have been initialized with variables. Each language parser may require a catalog with information on the translation of language specific datastructures into their BAT equivalent.

The `XYZoptimizer(Client c)` contains language specific optimizations using the MAL intermediate code as a starting point.

The `XYZtactics(Client c)` synchronizes the program execution with the state of the machine, e.g. claiming resources, the history of the client or alignment of the request with concurrent actions (e.g. transaction coordination).

The `XYZengine(Client c)` contains the applicable back-end engine. The default is the MAL interpreter, which provides good balance between speed and ability to analysis its behavior.

2.5 MonetDB Client Interface

Clients gain access to the Monet server through a internet connection or through its server console. Access through the internet requires a client program at the source, which addresses the default port of a running server.

At the server side, each client is represented by a session record with the current status, such as name, file descriptors, namespace, and local stack. Each client session has a dedicated thread of control, which limits the number of concurrent users to the thread management facilities of the underlying operating system. A large client base should be supported using a single server-side client thread, geared at providing a particular service.

The number of clients permitted concurrent access is a compile time option. The console is the first and is always present. It reads from standard input and writes to standard output.

Client records are linked into a hierarchy, where the top record denotes the context of the Monet administrator. The next layer is formed by a database administrator and the third layer contains user sessions. This hierachy is used to share and constrain resources, such as global variables or references to catalogue information. During parallel execution additional layers may be constructed. [This feature needs more implementation support]

Client sessions remain in existence until the corresponding communication channels break or its retention timer expires The administrator and owner of a sesssion can manipulate the timeout with a system call.

Keeping track of instructions executed is a valueable tool for script processing and debugging. Its default value is defined in the MonetDB configuration file. It can be changed at runtime for individual clients using the operation `clients.listing`(*mask*). A listing bit controls the level of detail to be generated during program execution tracing. The lowest level (1) simply dumps the input, (2) also demonstrates the MAL internal structur (4) adds the hidden type information

The MAL debugger uses the client record to keep track of any pervasive debugger command. For detailed information on the debugger features see Chapter 14 [Program Debugging], page 108.

2.6 Development Roadmap

In this section we summarize the MonetDB development roadmap as foreseen early 2006. The information is organized around the major system components. A precise timeline can not be given. It depends too much on the available resources and urgency (= pressure) by our research needs and clients.

2.6.1 SQL Roadmap

The long term objective for the SQL front-end is to provide all features available in SQL:2003. The priority for individual features is determined in an ad hoq way. The

SQL features scheduled for implementation and those that won;t be supported in the foreseeable future are shown below.

Our current assessment of the features planned for upcoming releases, in order of priority, are:

- *Full text retrieval* A full text retrieval support function consists of a special constructed index over text appearing in multiple columns of a relational table. This index is built using well-known Information Retrieval techniques, such as stemming, keyword recognition, and stop-word reduction. Several IR projects are underway, which enhance MonetDB with IR capabilities, e.g. see spiegle.
- *Stored SQL procedures* Stored procedures are a powerful scheme to offload operational abstractions to the server. The MonetDB server comes with its own programming language MAL to encode critical functions. Linking with the SQL compiler is readily available, leaving a MonetDB specific implementation of stored SQL procedures less important.
- *Support for multi-media objects*
- *General column and table constraint enforcement*
- *Internationalization of the character sets*
- *Full outer-join queries*
- *Triggers*

The database backend architecture prohibits easy implementation of several SQL-99 features. Those on the list below are not expected to be supported.

- Cursor based processing, because the execution engine is not based on the iterator model deployed in other engines. A simulation of the cursor based scheme would be utterly expensive from a performance point of view.
- Multi-level transaction isolation levels. Coarse grain isolation is provided using table level locks.

2.6.2 XQuery Roadmap

The XQuery compiler is currently only available on MonetDB Version 4. It is based on the Pathfinder compiler project.

A novel compiler based on a XQuery algebra is under development. We expect that an alpha release for MonetDB Version 5 becomes available in the fall of 2006.

2.6.3 Embedded MonetDB Roadmap

The embedded MonetDB software family provides support for both SQL and Xquery (Version 4 only). The software has been tuned to run on small scall hardware platforms.

A broader deployment of the embedded technology requires both extensions in the distributed MonetDB versions and its replication services. Continual attention is given to the memory footprint and cpu/io resource consumptions on embedded devices.

A separate project, called the Datacell, is underway and geared at providing a streaming environment for embedded applications. The supportive modules are scheduled for release in 2006.

2.6.4 Server Roadmap

The MonetDB server code base is continously being improved. Major areas under develoopement are:

- *Replication Service* A single-write multiple-read distributed replication service is prepared for release mid 2006. It will provide both the concept of merge tables and selective replication of tuples to different servers.
- *GIS support* Support for geographical application is underway. It consists of a concise library for managing geometric types.

2.7 Backward compatibility

The MonetDB software runs back to the early nineties and users have based their business critical applications on the platform since the mid-nineties. This long history creates a challenge in a research laboratory setting to find an evoluationary software developement track. Fortunately, the core development team is still at CWI and is dedicated to maintain the software base. Ofcourse, within the confines of limited resources available and in balance with the primary research activities.

The development of Version 5 was driven by the following compatibility requirements:

- The programming interface technology (ODBC, JDBC, Mapi) are all retained to reduce impact on external applications.
- SQL and XQuery should compile directly on both old and new versions.
- The MIL language is depreciated, but MIL programs can mostly be converted to the MAL language.[1]
- The kernel library GDK remains the primary execution engine, but it should also facilitate emerging alternatives.
- The key extension modules to the GDK kernel are mostly carried over without change.
- Auxiliary search paths are not supported, due to lack of interest in Version 4 user community and alternatives based on the BAT algebra.

[1] Only those features are compiled that do not require runtime knowledge on the typing structure.

3 SQL

The de facto language for database applications is SQL. It evolved through several phases of standardization to the version currently known as SQL-2003. The SQL standard provides an ideal language framework, in terms of standardization committee viewpoints. It is, however, hardly met by any of the existing (commercial) implementations. This is largely due to legacy of old software and backward compatibility requirements from their client base. See for instance this on-line article on SQL standards.

The MonetDB database system was originally developed as a database back-end kernel with its own, low-level algebraic interface and scripting language. The development of a SQL front end has been purposely postponed to the point where the kernel code base was sufficiently mature and field tested in large, mission critical applications in the financial sector.

In 2002 the first version of the SQL front end emerged. This late development made it possible to immediately start from the SQL'99 definition. As soon as the SQL'03 specifications became available, its content was taken as the primary frame of reference. The SQL development strategy is driven by immediate needs of the user base, such that less-frequently used features end up low on the development stack.

The purpose of this chapter is to give a quick introduction on the SQL front end, its limitations, and the way to use it. Throughout this document proficiency in elementary use of SQL is assumed. If you are new to this world then pick up any of the introduction books and study it carefully. The SQL Implementation front-end is based on the SQL-99 standard, which is covered in many text books e.g. *J. Melton and A.R. Simon*, SQL:1999 Understanding Relational Language Components, ISBN 1558604561.

The architecture is based on a separate compiler module, which translates SQL statements into MAL. In this process common optimization heuristics, specific to relational algebra are performed. There are bindings for SQL with JDBC, ODBC, PHP and C, (see Chapter 5 [User Interfaces], page 50) to integrate seamlessly in existing developments environments. Client utilities like Section 5.4.1 [Aqua Data Studio], page 52, Section 5.4.2 [DbVisualizer], page 53 and Section 5.4.3 [iSQL-Viewer], page 54 work flawlessly with the MonetDB SQL implementation and make your experience with even better!

The remainder of this chapter provides a short tutorial to get going with MonetDB/SQL. A synopsis of the language features provides a quick intro on the scope of the current implementation and the short list for functional enhancements planned. The programming support sections illustrate some of the advanced features to analyse and your SQL code.

3.1 Getting Started with MonetDB/SQL

Working with SQL requires installation of the MonetDB server and SQL compiler module. It can be downloaded from the download section at SourceForge. It is already included in the Windows installers.

The first step is to start the database backend following the steps in Section 1.9 [Start and Stop the Server], page 15 or illustrated below. Once it runs, you can choose between a textual interface or one of the graphical user interfaces (See Chapter 5 [User Interfaces], page 50). The easiest to start with is a textual SQL client. Under Windows, this client is already configured to directly contact the running server. For Linux users we illustrate the actions taken behind the scene.

The first action on Linux is to start the database server as described in Section 1.9 [Start and Stop the Server], page 15.

```
shell> monetdb --start --dbname=voc
Starting database server 'voc'... ok
shell>
```

Failures to start the server are reported in the database log, which you can view using the commands

```
monetdb --log
monetdb --log --dbname=voc
```

The next step is to connect to the server with a textual client. Two such clients are included in the distribution: `mclient` and `mjclient`.

The former is the native, C-implementation of the MonetDB client interface. It is a no-frills and fast interface geared at developers. The `mjclient` is a Java implementation and it uses a standardized database interaction protocol. If you are familiar with JDBC-based applications, or intend to build those, this interface may be your prime choice. The `mjclient` utility is illustrated:

```
shell> mjclient --database=voc --user=monetdb
password: *******
Welcome to the MonetDB interactive JDBC terminal!
Database: MonetDB 4.99.19
Driver: MonetDB Native Driver 1.3 (Spur_pre1 20060112)
Type \q to quit, \h for a list of available commands
auto commit mode: on
monetdb->
```

The default password for the user *monetdb* is *monetdb*. The location of the jar file is taken from the MonetDB installation directory. `mjclient` reads settings from the `~/.monetdb` file (in property=value format) for ease of use.

The world of SQL is now available to the user:

```
monetdb-> SELECT 'Hello SQL, here I come' AS "message";
+------------------------+
| message                |
+========================+
| Hello SQL, here I come |
+------------------------+
1 row
monetdb->
```

The alternative interface is `mclient` with its Spartan rendering scheme:

```
shell> mclient -lsql
sql>SELECT 'Hello SQL, here I come' AS "message";
#  # table_name
# message # name
# char # type
# 22 # length
[ "Hello SQL, here I come"      ]
sql>
```

In a clean setup, either client program should run without problems. If you haven't managed to contact the database server either way, backtrack your steps. Is the server running? (use `monetdb --status`) More arguments may be needed if the server lives on a different machine or the client does not have access to the MonetDB configuration file to access the defaults. See Section 5.1 [Mapi Client], page 50 or Section 5.2 [Jdbc Client], page 50 for details.

3.2 The VOC tutorial

Exploring the wealth of functionality offered by MonetDB is best started using a toy database. An example of such database is the VOC data set that provides a peephole view into the administrative system of a multi-national company, the *Vereenigde geoctrooieerde Oostindische Compagnie* (VOC for short - The (Dutch) East Indian Company).

The VOC was granted a monopoly on the trade in the East Indies on March 20, 1602 by the representatives of the provinces of the Dutch republic. Attached to this monopoly was the duty to fight the enemies of the Republic and prevent other European nations to enter the East India trade. During its history of over 200 years, the VOC became the largest company of its kind, trading spices like nutmeg, cloves, cinnamon, pepper, and other consumer products like tea, silk and Chinese porcelain. Her factories or trade centers were world famous: Desjima in Japan, Mokha in Yemen, Surat in Persia and of course Batavia, the Company's headquarters on Java.

The history of the VOC is an active area of research and a focal point for multi-country heritage projects, e.g. TANAP, which includes a short historic overview of the VOC written by world expert on the topic F. Gaastra. The archives of the VOC are spread around the world, but a large contingent still resides in the National Archive<, The Hague. The archives comprise over 25 million historical records. Much of which has not (yet) been digitized.

The MonetDB/SQL tutorial is based on the material published in the book J.R. Bruijn, F.S. Gaastra and I. Schaar, *Dutch-Asiatic Shipping in the 17th and 18th Centuries*, which gives an account of the trips made to the East and ships returned safely (or wrecked on the way). A total of 8000 records are provided. They include information about ship name and type, captain, the arrival/departure of harbors along the route, personnel accounts, and anecdotal information.

3.2.1 Acquiring and loading the VOC data set

The VOC data set can be downloaded from the MonetDB Assets site as a gzipped file with SQL statements. After the zipfile has been extracted, the file should be loaded into MonetDB via either the Java based Section 5.2 [Jdbc Client], page 50 utility, or the Section 5.1 [Mapi Client], page 50 C-program. Alternatively, the URL to the VOC data file can be supplied to the `mjclient` utility, which then directly reads from the URL.

Before you load the VOC data set, it is advised to first add a different user with its own schema to the MonetDB database. We illustrate this process using the textual SQL client. Make sure the MonetDB server has been started, then start the SQL client. Under Linux you will see something like this:

```
shell> mjclient -uvoc
password: ***
Welcome to the MonetDB interactive JDBC terminal!
Database: MonetDB 4.99.19
Driver: MonetDB Native Driver 1.3 (Spur_pre1 20060112)
Type \q to quit, \h for a list of available commands
auto commit mode: on
voc> CREATE USER "voc" WITH PASSWORD 'voc' NAME 'VOC Explorer' SCHEMA "sys";
Operation successful
voc> CREATE SCHEMA "voc" AUTHORIZATION "voc";
Operation successful
voc> ALTER USER "voc" SET SCHEMA "voc";
voc>\q
```

In the remainder of the tutorial you can no log onto the server as user `voc`.

The tutorial database can be initialized using either `mjclient` and `mclient` as follows, provided the sql-dump file has already been downloaded:

```
shell> mjclient -uvoc -f voc_dump.sql -Xbatching
password: ***
```

Or using a remotely living dump.

```
shell> mjclient -uvoc -f http://monetdb.cwi.nl/Assets/VOC/voc_dump.sql.gz -Xbatching
password:***
shell>mclient -lsql -uvoc <voc_dump.sql
```

The argument `Xbatching` instructs the JDBC client to batch instructions before shipping them to the server. Loading the database takes a few seconds on a state-of-the-art machine.

3.2.2 Exploring the VOC data set

The `mjclient` contains a *describe* operator, denoted by `\d` to inspect the definition of the database schema and its tables. This functionality is not available in `MapiClient`.

```
voc-> \d
TABLE   sys.craftsmen
TABLE   sys.impotenten
TABLE   sys.invoices
TABLE   sys.passengers
TABLE   sys.seafarers
TABLE   sys.soldiers
TABLE   sys.total
TABLE   sys.voyages
```

The set consists of 8 tables, which are all bound to each other using FOREIGN KEY relationships. The voyages table is considered to be the main table, which all others reference to. Every table, except invoices has a PRIMARY KEY defined over the columns number and number_sup. Since the invoices table holds zero or more invoices per voyage (identified by number, number_sub) a PRIMARY KEY constraint is not possible. Details of the tables are readily available.

```
voc-> \d sys.soldiers
CREATE TABLE "sys"."soldiers" (
        "number"               int      NOT NULL,
        "number_sup"           char(1)  NOT NULL,
        "trip"                 int ,
        "trip_sup"             char(1),
        "onboard_at_departure" int ,
        "death_at_cape"        int ,
        "left_at_cape"         int ,
        "onboard_at_cape"      int ,
        "death_during_voyage"  int ,
        "onboard_at_arrival"   int ,
        CONSTRAINT "soldiers_number_number_sup_pkey" PRIMARY KEY ("number", "number_su
        CONSTRAINT "soldiers_number_number_sup_fkey" FOREIGN KEY ("number", "number_su
);
voc->
```

The tables craftsmen, importenten, passengers, seafarers, and soldiers all share the same columns. We can define a VIEW that combines them all into one big table, to make them easier to access.

```
voc-> CREATE VIEW onboard_people AS
voc=> SELECT * FROM (
voc=(    SELECT ’craftsmen’ AS type, craftsmen.* FROM craftsmen
voc=(    UNION ALL
voc=(    SELECT ’impotenten’ AS type, impotenten.* FROM impotenten
voc=(    UNION ALL
voc=(    SELECT ’passengers’ AS type, passengers.* FROM passengers
voc=(    UNION ALL
voc=(    SELECT ’seafarers’ AS type, seafarers.* FROM seafarers
voc=(    UNION ALL
voc=(    SELECT ’soldiers’ AS type, soldiers.* FROM soldiers
voc=(    UNION ALL
voc=(    SELECT ’total’ AS type, total.* FROM total
voc=( ) AS onboard_people_table;
Operation successful
```

The new view will show up and we can just use it as a normal table, to for instance calculate the number of records for each group of people:

```
voc-> SELECT type, COUNT(*) AS total FROM onboard_people GROUP BY type ORDER BY type;
+------------+-------+
| type       | total |
+============+=======+
| craftsmen  |  2349 |
| impotenten |   938 |
| passengers |  2813 |
| seafarers  |  4468 |
| soldiers   |  4177 |
| total      |  2454 |
+------------+-------+
6 rows
```

```
voc-> select count(*) from impotenten;
+--------------+
| count_number |
+==============+
|          938 |
+--------------+
1 row
```

It is possible to play with the set in many ways, to find out several things that took place during the voyages of the ships, or the money that was earned. A few examples are shown below.

```
voc-> SELECT COUNT(*) FROM voyages WHERE particulars LIKE ’%_recked%’;
+--------------+
| count_number |
+==============+
|          354 |
+--------------+
1 row

voc-> SELECT chamber, CAST(AVG(invoice) AS integer) AS average
voc=> FROM invoices
voc=> WHERE invoice IS NOT NULL
voc=> GROUP BY chamber
voc=> ORDER BY average DESC;
+---------+----------+
| chamber | average  |
+=========+==========+
| A       |  282996 |
| Z       |  259300 |
| H       |  150182 |
| R       |  149628 |
| D       |  149522 |
| E       |  149518 |
| <NULL>  |   83309 |
+---------+----------+
7 rows

voc-> CREATE VIEW extended_onboard AS SELECT number, number_sup, trip, trip_sup, onboa
Operation successful
```

3.2.3 Historical background

Please take the time to experiment with the VOC data. For your convenience, we give here a short histororical background and interpretation of what to find in this database. The introduction given below is an OCR version from the book: J.R. Bruijn, F.S. Gaastra and I. Schaar, Dutch-Asiatic Shipping in the 17th and 18th Centuries

This book presents tables which give a virtually complete survey of the direct shipping between the Netherlands and Asia between 1595-1795. This period contains, first, the voyages of the so-called Voorcompagnieand, hence, those for and under control of the Vereenigde Oostindische Compagnie (VOC). The survey ends in 1795. That year saw an end of the regular sailings of the VOC between the Netherlands and Asia, since, following the Batavian revolution in January, the Netherlands became involved in war with England. The last outward voyage left on 26 December 1794. After news of the changed situation in the Netherlands was received in Asia, the last homeward voyage took place in the spring of 1795. The VOC itself was disbanded in 1798.

In total 66 voyages of the voorcompagnieare listed, one more than the traditionally accepted number. The reconnaissance ship, POSTILJON, from the fleet of Mahu and De Cordes, that was collected en route is given its own number (0022). Since the attempt of the Australische Compagnie to circumvent the monopoly of the VOC can be considered as a continuation of the voorcompagniethe voyage of Schouten and Le Maire is also listed (0196-0197). For the rest, exclusively the outward and homeward voyages of the VOC are mentioned in the tables. Of those there were in total 4722 outward and 3359 homeward. The administration of the company was strictly followed, so that, for example, the voyage of Hudson in 1609 (0133) is listed, but not that of Roggeveen in 1721-1722. Voyages of East Indiamen that were driven off course, and arrived for instance in Surinam, or those which went no further than the Cape are mentioned, as opposed to those of warships of the five Admiralties which, from 1783, were sent to Asia to protect the fleets and possessions of the VOC.

The sources of the journeys consist primarily of the archives of the VOC in the Algemeen Rijksarchief in The Hague. They are, on the one hand, the so-called 'Uitloopboeken' and ship registers, and, on the other, the 'Overgekomen Brieven en Papieren' (OBP's). The latter contain the regular reports on the arrival and departure of ships in Batavia and other Asiatic harbors. In addition, the 'Overgekomen Brieven van de Kaap de Goede Hoop' and some other, more dispersed sources must be mentioned. The data on the voyages of the voorcompagniederive above all from ources published by the Linschoten Vereeniging.

In volume I, the principal sources are described extensively and the origin of the information on each voyage is given. In addition, that volume contains an introduction on the organization of the VOC's shipping, which also includes an analysis and summary of the data presented in the tables. Various other supplementary information, such as the value of the export from the Netherlands, only available by year, is also published there. The tables follow closely the material presented in the major sources ('Uitloopboeken' en OBP's). Since these sources are not uniform over a period of almost two centuries, the level of completeness of the information given for each voyage also varies.

Homeward voyage During the compilation of the tables it became necessary in a few cases to add an A to some numbers. This occurred 5 times, in the following places: 5022, 5980, 5987, 6246 and 6649. Similarly in three cases a number had to be left open. The following numbers have not been used: 4605, 5027 and 8215. The voyage number is followed by a figure which shows whether the ship is making its first, second or subsequent voyage. The outward and homeward voyages are counted separately. The first voyage from the Netherlands and the subsequent homeward voyages are both shown by a '1'. Occasionally a ship was built or acquired in Asia. The first outward voyage of such a ship is considered as its second voyage.

Ship's name A uniform spelling has been chosen for the numerous variants given in the sources. In alphabetical ordering and in the index, the most relevant word was chosen. Thus the WAPEN VAN, HOORN (0243) is given under HOORN, the HOF NIET ALTIJD ZOMER (2380) under ZOMER and the VROUWE REBECCA JACOBA (3668) under REBECCA. It should be noted that especially in the seventeenth century ships' names were frequently provided with additions which were not used in a consistent fashion. The AMSTERDAM (0431) was sometimes called NIEUWAMSTERDAM, the WITTE OLIFANT (0533), the OLIFANT. The most frequent name is given in the tables. In the eighteenth century, especially, ships' names were frequently changed, or they used each other's names. This is always mentioned under the heading Particulars and in the index.

Master's name Similarly, a uniform spelling has been chosen for the name of the master, generally schipper, but in the eighteenth century also a kapitein or kapiteinluitenant. The index is arranged by surname or patronymic.

Tonnage The volume of the ships is given in metric tons. The sources give the figures in lasten (1 last = 2 tons). After 1636, however, information in lasten is no longer of any value, as, for fiscal reasons, the VOC's figures were kept artificially low. From then on the volume has been calculated on the basis of the measurements of the ships, according to a simple formula (volume in lasten = length x breadth x depth in Amsterdam feet, divided by 200; 1 ft. = 28,3 cm). The results of this calculation have been doubled and are given in the tables. This method and the problems regarding the assessment of the ships' volume is described in Volume I. In a number of cases where inconsistent information was found, both calculations are given, thus e.g. 600/850.

Type of ship Occasionally, in those cases where this is mentioned in the sources, the type of the ship is given in the same column as the tonnage. In general, the most frequent type of ship, the retourschip (East Indiaman) is not mentioned in the sources. Therefore, where the type of ship is not mentioned, it may often be assumed that an East Indiaman is meant. The various other types - hoeker, kat, pinas, jacht, fluit, paketboot - are given in Dutch.

Built The year given in this column refers to the year in which the ship was built. If the ship was hired or bought by the VOC, then this is mentioned in the column, together with the year in which the transaction occurred.

Yard The place is given where the ship was built. The chambers of the VOC had their own yard. 'A' refers to Amsterdam, 'Z' to Zeeland, 'D' to Delft, 'R' to Rotterdam, 'H' to Hoorn, and 'E' to Enkhuizen. When a ship was hired or bought by the VOC, the letter indicates the chamber that was responsible for the transaction. The ships of the voorcompagniedid not belong to a chamber. In these cases, A' indicates that a ship was built at an Amsterdam yard. The chambers also had no part in the buying or building of ships by the Hoge Regering in Batavia. In these cases the place of building or purchase in Asia is given.

Chamber With the outward voyages, this column gives the chamber which equipped the ship; with the homeward, the chamber to which the ship was addressed. There is no entry in this column for ships organized by the voorcompagnie

Departure

Under this heading is given the date and place of departure from Europe, Asia or the Cape of Good Hope. A date like 03-02-1645 refers to 3 February 1645. Where sailings from the Republic are concerned, the date given refers to the departure from the roads.

Amsterdam, Hoorn and Enkhuizen ships generally left from Texel roads, Zeeland ships from the Wielingen or the roads of Rammekens, and Rotterdam and Delft ships from Goeree. Sometimes, ships were forced by storms or damage to return to the roads for a time or they sought shelter in one of the estuaries on the coast of Holland or Zeeland. Where possible, this is mentioned under the heading Particulars. In general the first date of departure is given in the tables, but in some cases, a later date has been chosen, in deference to the sources.

As for leaving Batavia, departure from the roads of the town was decisive, and not, as is frequently described in the Company papers, the reaching of the 'open sea' after passing the Sunda Strait. Where departures from other Asian ports are concerned in general only the Company establishment from which the ship sailed is given. Thus Ceylon is mentioned in the columns, but it can be assumed that most ships left from the Bay of Galle, at the southern point of the island. China is given for ships which left from Canton, and the date refers to departure from the roads at Whampoa. Bengal is given for ships which left from the anchorage in the Ganges close to the VOC-establishment at Hughly.

Call at Cape The data in this column give the arrival at (above) and the departure (below) from the Cape of Good Hope. In general no distinction is made between Table Bay and False Bay. Mention is made, when given in the sources, of ships which put in to the more northerly Saldanha and St. Helena Bays. When a ship sailed past the Cape, this is denoted by 'no call'. When it is not known whether the ship stopped at the Cape at all - especially frequent before the foundation of the refreshment station there in 1652 - the column is left blank.

Arrival The third column contains the date and place of arrival in Asia, Europe or, when that was the destination of the journey, at the Cape of Good Hope. The place of arrival is given in the same manner as that of departure, though, in addition to the estuaries mentioned above, ships sometimes arrived in the Netherlands via the Vlie or at Delfzijl. The place of arrival in Asia refers to the establishment reached, unless the sources specify the actual port.

On board It is possible to differentiate the number of those on board into various categories. For the outward journey, these are seafarers, soldiers, craftsmen, and passengers. The craftsmen are those who were employed to perform some particular service in Asia, and are thus not part of the crew as such. 'Passengers' is in fact a residual category, including high officials of the Company, including ministers of religion with their wives and servants, but also slaves and stowaways. Whenever such a differentiation is not possible, which is especially the case in the early years, a figure for the total is given. Italics are used for this, or when the figures refer to more than one category. Only those categories are mentioned which were on board. Therefore, when one category is mentioned, this implies that the others were not represented on board.

The sources for the return voyages are of a different kind and normally far less complete. They are totally absent for the journey between the Cape and the Netherlands. However, another category must be mentioned, namely the impotenten, who for various reasons were released from active service for the VOC and sent back to Europe. With regard to many voyages the sources only give the number of passengers and impotenten, and not the number of sailors and soldiers. Obviously, the absence of figures under these headings does not imply that there were none on board.

Information on the outward voyages is divided into six columns:

1. onboard_at_departure The number on board at departure
2. death_at_cape The number dying between the Netherlands and the Cape. Frequently this figure refers to all the categories together, even when the other information is available per category. In such cases this figure is printed in italics.
3. left_at_cape The number who leave the ship at the Cape.
4. onboard_at_cape The number who come on board at the Cape.
5. death_during_voyage The number dying on the whole voyage. Subtraction of III from V gives the number dying between the Cape and Asia.
6. onboard_at_arrival The number on board on arrival in Asia.

Three columns are given for the homeward voyages:

1. onboard_at_departure The number on board at departure
2. death_at_cape The number dying en route to the Cape
3. left_at_cape The number who went from board at the Cape

The figures in the various columns are taken from different sources which are not always consistent with each other. Therefore the figures on changes in the number of those on board during the voyages do not always tally with those on the size of the crew at departure and arrival.

Invoice value For the return voyages, the total value of the ship's cargo, according to the invoice made up in Batavia or some other establishment, is given, as is the chamber for which the cargo was destined. Generally, this was for the chamber under whose jurisdiction the ship sailed, but occasionally a proportion of the cargo was for one or more of the other chambers.

Particulars Under the last heading details deriving from the basic sources are given. They are generally incidental and as such not to be placed in one of the preceding columns. Because the sources are not the same across the whole period, and at times less complete, the extent and sort of material under this heading could not be consistent.

In so far as it is available, information deals with the ports of call on the journey, with the details of changes in the composition of the crew and with the eventual fate of the ship. For the return voyage, the name of the fleet-commander is generally given, and, after his name, the number of the ship he was on. Finally, where necessary, differences in data between various sources are indicated. Occasionally, particulars from a published source are added.

Corresponding number This number, placed at the far right of the tables, denotes the next homeward voyage of the ship in volume II (naturally absent when the ship remains in Asia), or, in volume III, for homeward voyages, the number of the ship's previous outward voyage. In those cases where the ship was acquired in Asia, no corresponding number is given for the first homeward voyage from Asia.

Due to the long duration of the preparation of these two volumes there are some inconsistencies in the text of the particulars and in the use of language.

3.3 SQL Language

The MonetDB/SQL language follows the SQL-2003 specification. This section introduces the language components supported in the current and upcoming releases.

3.3.1 SQL Preface

We use the following notation to explain the supported variation SQL. The SQL syntax is summarized in extended BNF. Alternative constructs are seperated by `|` and grouped by parenthesis. Optional parts are marked with square brackets. A repetition is marked with either ’+’ or ’*’ to indicate at least once and many times, respectively. Lexical tokens are illustrated in capitals.

Sometimes MonetDB/SQL supports extra syntax this is identified by a ’+’ before the clause. Unfortunately, we do not support the full SQL 2003 jet and this is specified by a ’-’ before the clause. If next to the ’-’ a ’<’ and version number are given the feature is supported in the older releases.

Currently we have partial support for SQL-2003. Features are added when (enough) users express their interest in these.

- Cursors, because the underlying engine is not based on record iterators.
- Triggers They will be supported in a future release.
- Asserts They will be supported in a future release.
- Domains
- Collate
- Character sets
- SQL User Defined Types

3.3.2 SQL Data Definition

3.3.2.1 Create TABLE

The parser currently supports the full <table scope> specifier, but the implementation is limited to LOCAL TEMPORARY tables (ie the tables are only visable in the clients session) and on COMMIT we ’DROP’ these temporaries.

```
The CREATE table statement follows the 2003 syntax, ie

        CREATE [ <table scope> ] TABLE
                <table name>
                <table contents source>
                [ ON COMMIT <table commit action> ROWS ]
                +[ ON COMMIT DROP ]

        <table scope> ::= <global or local> TEMPORARY

        <global or local> ::= GLOBAL | LOCAL

        <table commit action> ::= PRESERVE | DELETE

<table contents source> ::=
                <table element list>
        -|      OF <path-resolved user-defined type name> [ <subtable clause> ] [ <table el
        -<3.0|  <as subquery clause>
```

```
<table element list> ::= ( <table element> [ { , <table element> }... ] )

<table element> ::=
                <column definition>
        |       <table constraint definition>
        |       <like clause>
        -|      <self-referencing column specification>
        |       <column options>

-<self-referencing column specification> ::= REF IS <self-referencing column name> <referen

-<reference generation> ::= SYSTEM GENERATED | USER GENERATED | DERIVED

-<self-referencing column name> ::= <column name>

<column options> ::= <column name> WITH OPTIONS <column option list>

<column option list> ::= [ -<scope clause> ] [ <default clause> ] [ <column constraint defi

-<subtable clause> ::= UNDER <supertable clause>

-<supertable clause> ::= <supertable name>

-<supertable name> ::= <table name>
```

3.3.2.2 create table like

It is posible to create a table which looks like an existing table. This can be done using the create table like statement. Currently there is no support for additional options. A work around is to use the alter statement to change options.

```
<like clause> ::= LIKE <table name> [ <like options> ]

-<like options> ::= <identity option> | <column default option>

-<identity option> ::= INCLUDING IDENTITY | EXCLUDING IDENTITY

-<column default option> ::= INCLUDING DEFAULTS | EXCLUDING DEFAULTS
```

3.3.2.3 create table AS subquery

As of version 3.0 support is added for table construction based on subqueries.

```
-<3.0   <as subquery clause> ::= [ ( <column name list> ) ] AS <subquery> <with or without 

-<3.0   <with or without data> ::= WITH NO DATA | WITH DATA
```

3.3.2.4 columns

```
<column definition> ::=
                <column name> [ <data type> | -<domain name> ] [ -<reference scope check> ]|
                [ <default clause> | <identity column specification> | -<generation clause>
                [ <column constraint definition>... ] [ -<collate clause> ]
```

3.3.2.5 Identity column

SQL 2003 added identity columns, which are columns for which the values are coming from a sequenc generator. Besides the SQL 2003 syntax also the syntax from mysql (auto_increment) and postgress (serial data type) are supported.

```
<identity column specification> ::=
                GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY
                [ ( <common sequence generator options> ) ]
        +|      auto_increment

-<generation clause> ::= <generation rule> AS <generation expression>

-<generation rule> ::= GENERATED ALWAYS

-<generation expression> ::= ( <value expression> )
```

3.3.2.6 Default values

To make insert statements easier a default value can associated with each column. Besides literal values, temporal and sequence functions can be used as default value. Value of these functions at insert time will be used.

```
<default clause> ::= DEFAULT <default option>

<default option> ::=
                <literal>
        |       <datetime value function>
        |       USER
        |       CURRENT_USER
        |       CURRENT_ROLE
        |       SESSION_USER
        |       NULL
        +|      NEXT VALUE FOR <sequence name>
```

For example an auto increment column can be created using the following column specification: col_name integer default NEXT VALUE FOR sequence_name

3.3.2.7 Column and Table Constraints

Column and Table constraints are supported. Besides the simple NOT NULL check also UNIQUE, PRIMARY and FOREIGN keys are supported. The limitation stems from the missing triggers, ie we currently check constraints directly on insert, update and delete. The NULL matching on foreign keys is limited to the SIMPLE MATCH type (NULL values statisfy the constraint). The FULL and PARTIAL MATCH types are not supported. The referential action is currently limited to RESTRICT, ie an update fails if a other columns have references to it.

```
<column constraint definition> ::= [ <constraint name definition> ] <column constraint> [ <
```

```
<column constraint> ::=
                NOT NULL
        |       <unique specification>
        |       <references specification>
        -|       <check constraint definition>

@c  we need to update to the new 2003 syntax soon...

<reference scope check> ::= REFERENCES ARE [ NOT ] CHECKED [ ON DELETE <reference scope che

<reference scope check action> ::= <referential action>

--h3 11.6 <table constraint definition> (p543)
--/h3

--p
Specify an integrity constraint.
--/p

<table constraint definition> ::= [ <constraint name definition> ] <table constraint> [ <co

<table constraint> ::=
                <unique constraint definition>
        |       <referential constraint definition>
        |       <check constraint definition>


<unique constraint definition> ::=
                <unique specification> ( <unique column list> )
        -|      UNIQUE ( VALUE )

<unique specification> ::= UNIQUE | PRIMARY KEY

<unique column list> ::= <column name list>

--h3 11.8 <referential constraint definition> (p547)
--/h3

--p
Specify a referential constraint.
--/p

<referential constraint definition> ::= FOREIGN KEY ( <referencing columns> ) <references s

<references specification> ::= REFERENCES <referenced table and columns> [ MATCH <match typ
```

```
<match type> ::= FULL | PARTIAL | SIMPLE

<referencing columns> ::= <reference column list>

<referenced table and columns> ::= <table name> [ ( <reference column list> ) ]

<reference column list> ::= <column name list>

<referential triggered action> ::= <update rule> [ <delete rule> ] | <delete rule> [ <updat

<update rule> ::= ON UPDATE <referential action>

<delete rule> ::= ON DELETE <referential action>

<referential action> ::= CASCADE | SET NULL | SET DEFAULT | RESTRICT | NO ACTION

<check constraint definition> ::= CHECK ( <search condition> )

@subsubsection ALTER TABLE

<alter table statement> ::= ALTER TABLE <table name> <alter table action>

<alter table action> ::=
                <add column definition>
        |       <alter column definition>
        |       <drop column definition>
        |       <add table constraint definition>
        |       <drop table constraint definition>

@subsubsection ADD column

<add column definition> ::= ADD [ COLUMN ] <column definition>

@subsubsection ALTER column

<alter column definition> ::= ALTER [ COLUMN ] <column name> <alter column action>
<alter column action> ::=
                <set column default clause>
        |       <drop column default clause>
        |       <add column scope clause>
        |       <drop column scope clause>
        |       <alter identity column specification>

<set column default clause> ::= SET <default clause>
<drop column default clause> ::= DROP DEFAULT
-<add column scope clause> ::= ADD <scope clause>
-<drop column scope clause> ::= DROP SCOPE <drop behavior>
```

```
<alter identity column specification> ::= <alter identity column option>...

<alter identity column option> ::=
                <alter sequence generator restart option>
        |       SET <basic sequence generator option>

<drop column definition> ::= DROP [ COLUMN ] <column name> <drop behavior>

@subsubsection ADD constraint
<add table constraint definition> ::= ADD <table constraint definition>
<drop table constraint definition> ::= DROP CONSTRAINT <constraint name> <drop behavior>

@subsubsection DROP table
<drop table statement> ::= DROP TABLE <table name> <drop behavior>
```

3.3.2.8 CREATE VIEW

Regular view specifications are supported. However, recursive views and referenceable views are not supported. Next to this 2003 feature we support creating a view on top of a set of bats. In this case the query expression is replaced by the keyword BATS, and the bats are found based on the view specification. As this feature requires indepth knowledge of the system it is only open to the ADMIN_ROLE.

```
<view definition> ::=
                CREATE -[ RECURSIVE ] VIEW <table name> <view specification> AS <view query
                [ WITH -[ <levels clause> ] CHECK OPTION ]

<view query expression> ::=
        <query expression>
 +|     BATS

<view specification> ::= <regular view specification> | <referenceable view specification>

<regular view specification> ::= [ ( <view column list> ) ]

-<referenceable view specification> ::= OF <path-resolved user-defined type name> [ <subvie

-<subview clause> ::= UNDER <table name>

<view element list> ::= ( <view element> [ { , <view element> }... ] )

<view element> ::= <self-referencing column specification> | <view column option>

<view column option> ::= <column name> WITH OPTIONS -<scope clause>
```

```
<levels clause> ::= CASCADED | LOCAL

<view column list> ::= <column name list>
```

3.3.3 SQL Data Types

MonetDB/SQL supports the following list of types.

CHAR[ACTER] (L)	character string with length L
VARCHAR (L) | CHARACTER VARYING (L)	string with atmost length L
CLOB | CHARACTER LARGE OBJECT	
BLOB | BINARY LARGE OBJECT	
DECIMAL(P,S) | NUMERIC(P,S)	
SMALLINT	16 bit integer
INT	32 bit integer
BIGINT	64 bit integer
serial	special 64 bit integer (sequence generator)
REAL	32 bit floating point
DOUBLE [PRECISION]	64 bit floating point
BOO[LEAN]	
DATE	
TIME(T)	
TIMESTAMP(T)	
INTERVAL(Q)	

3.3.4 SQL Data Manipulation

3.3.5 SQL Schema Definition

3.3.6 SQL Users

3.3.7 SQL Transactions

3.4 MonetDB/SQL Features

The SQL Implementation implementation is based on the SQL-99 standard. It is built incrementally over time to cope with the effort required to realize all features. Priority is given to those features relevant to our research and upon request from our valued partners.

The plethora of SQL implementations also show that the standard alone is hardly sufficient. All database systems have extended or twisted the language definition to satisfy the needs of their customer, to support legacy, and to better tap into the functionality offered by their kernel implementations. MonetDB/SQL is bound to follow this path too. The kernel imposes limitations on the features that can and cannot be realized. The are summarized in Section 2.6 [Development Roadmap], page 23.

In the context of programming support, we provide a few primitives to simplify debugging of SQL programs. They are described separately. The list below provides a synopsis of the SQL features supported in the released version.

3.4.1 SQL-99 Feature list

In this section we give an itemized list of the SQL-99 features supported in the current release.

- Primary and foreign key integrity enforcement.
- Subqueries are fully supported.
- Table expression `union` and `intersection`.
- Table views for query processing only.
- Binding SQL functions with to externally supplied routines.
- Auto-increment keys (SQL 2003).

3.4.2 SET Statement

MonetDB/SQL comes with a limited variable scheme. Globel, session based variables can be introduced using the construct:

```
SET <variable>=<string>
SET <variable>=<boolean>
SET <variable>=<int>
```

Their type is inherited from the literal value supplied. The SQL engine comes with a limited set of environment variables to control its behavior.

- The `debug` variable takes an integer and sets the server global debug flag. (See MonetDB documentation) It also activates the debugger when the query is being executed.
- The `explain` variable currently takes the values 'plan' or 'performance'. The 'plan' qualifier produces a relational table with the plan derived for execution using the applicable back end and optimizer schemes. The 'performance' qualifier produces a performance trace of the SQL queries for post analysis. It is available in the system table called `history`. Omission of either qualifier merely results in a straight execution of the query.
-

  The `auto_commit` variable takes a boolean and controls automatic commit after a successful SQL execution, or conversely an automatic rollback.
- The `reply_size` limits the number of tuples sent to the front end. [It is unclear how the remainder can be obtained within the SQL language framework. The reply_size should be replaced by the `limit` language construct]
- The `history` variable is a boolean flag which leads to capturing execution information of any SQL query in a table for post-analysis.

By default all remaining variables are stored as strings and any type analysis is up to the user. The can be freely used by the SQL programmer for inclusion in his queries. [TODO, a little more flexibilty would mean a lot]

The variables and their type definition are assembled in a system table called 'sessions'. Its default content is shown below:

```
>select * from sessions;
#---------------------------------------------------------------------#
# t                     t                   t               t         # name
# str                    str                 str              int       # type
#-----------------------------------------------------------------------#
[ "debug",                 "0",                 "int",           0       ]
[ "reply_size",            "-1",                "int",           0       ]
[ "explain",               "",                  "varchar",       0       ]
[ "auto_commit",           "false",             "boolean",       0       ]
[ "current_schema",        "sys",               "varchar",       1       ]
[ "current_user",          "monetdb",           "varchar",       1       ]
[ "current_role",          "monetdb",           "varchar",       0       ]
[ "current_timezone",      "0",                 "sec_interval",  0       ]
[ "optimizer",             "yes",               "varchar",       0       ]
```

3.4.3 EXPLAIN Statement

The intermediate code produced by the SQL Implementation compiler can be made visible using the `explain` statement modifier. It gives a detailed description of the actions taken to produce the answer. The example below illustrates what you can expect when a simple query is pre-pended by the `explain` modifier. Although the details of this program are better understood when you have read the Chapter on MAL Chapter 7 [MonetDB Assembler Language], page 69 the global structure is easy to explain.

```
>select count(*) from tables;
[ 27 ]
>explain select count(*) from tables;
#factory sql_cache.s1_0():bit;
#    _2:bat[:void,:int]  := sql.bind("sys","ptables","id",0);
#    _8:bat[:void,:int]  := sql.bind("sys","ptables","id",1);
#    _11 := bat.setWriteMode(_8);
#    _15:bat[:oid,:int]  := sql.bind("sys","ptables","id",3);
#    _18 := bat.setWriteMode(_15);
#    _24:bat[:void,:oid]  := sql.bind_dbat("sys","ptables",0);
#    _39:bat[:void,:int]  := sql.bind("sys","ttables","id",0);
#    _45:bat[:void,:oid]  := sql.bind_dbat("sys","ttables",0);
#barrier _90 := true;
#    _13 := algebra.kunion(_2,_11);
#    _20 := algebra.kdifference(_13,_18);
#    _22 := algebra.kunion(_20,_18);
#    _26 := bat.reverse(_24);
#    _28 := algebra.kdifference(_22,_26);
#    _33 := algebra.markT(_28,0@0);
#    _35 := bat.reverse(_33);
#    _37 := algebra.join(_35,_22);
#    _47 := bat.reverse(_45);
#    _49 := algebra.kdifference(_39,_47);
#    _53 := algebra.markT(_49,0@0);
#    _55 := bat.reverse(_53);
#    _57 := algebra.join(_55,_39);
#    _59 := bat.setWriteMode(_37);
```

```
#    bat.append(_59,_57);
#    _65 := algebra.markT(_59,0@0);
#    _67 := bat.reverse(_65);
#    _69 := algebra.join(_67,_59);
#    _74 := algebra.markT(_69,0@0);
#    _76 := bat.reverse(_74);
#    _78 := algebra.join(_76,_69);
#    _80 := aggr.count(_78);
#    sql.exportValue(1,"sys.tables","count_id","int",32,0,6,_80);
#    yield _90;
#    redo _90;
#exit _90;
#end s1_0;
```

The SQL compiler keeps a limited cache of queries. Each query is looked up in this cache based on an expression pattern match where the constants may take on different values. If it doesn;t exist, the query is converted into a *factory* code block and stored in the module `sqlcache`. It consists of a prelude section, which locates the tables of interest in the SQL catalogs. The block between `barrier` and `yield` is the actual code executed upon each call of this function. It is a large collection of relational algebra operators, whose execution semantics depend on the actual MAL engine. The `factory` ensures that only this part is called when the query is executed repetitively.

The call to the cached function is included in the function `main`, which is the only piece of code produced if the query is used more than once. The query cache disappears when the server is brought to a halt.

When/how is the cache cleared? Can you list elements in the cache directly, e.g `explain;` or `explain sqlcache` or `explain sqlcache.sql0;`

3.4.4 DEBUG Statement

The SQL statements are translated into MAL programs, which are optimized and stored away in an `sql_cache` module. The generated code can be debugged with the MAL debugger. It provides a simple mechanism to trace the execution, hunting for possible errors and detect performance bottlenecks (Section 14.3 [Runtime Inspection], page 111).

The example below, illustrates how you can easily obtain a quick overview of the cost components of the query using the debugger timer flag and `continuation` command for the debugger.

[ERROR: the 'start transaction' is needed now]

```
>start transaction;
&4 f
>debug select count(*) from tables;
#mdb #    mdb.start();
mdb>next
#mdb #    sql_cache.s0_0();
mdb>timer
mdb>continue
#    13 usec#    _2:bat[:void,:int]  := sql.bind(_3="sys", _4="ptables", _5="id", _6=0)
#     2 usec#    _8:bat[:void,:int]  := sql.bind(_3="sys", _4="ptables", _5="id", _9=1)
#     3 usec#    _11 := bat.setWriteMode(_8=<tmp_515>)
#     2 usec#    _15:bat[:oid,:int]  := sql.bind(_3="sys", _4="ptables", _5="id", _16=3)
#     1 usec#    _18 := bat.setWriteMode(_15=<tmp_516>)
#     2 usec#    _24:bat[:void,:oid]  := sql.bind_dbat(_3="sys", _4="ptables", _6=0)
#     4 usec#    _39:bat[:void,:int]  := sql.bind(_40="sys", _41="ttables", _42="id", _43=0
#     1 usec#    _45:bat[:void,:oid]  := sql.bind_dbat(_40="sys", _41="ttables", _43=0)
#    17 usec#    _13 := algebra.kunion(_2=<tmp_26>, _11=<tmp_515>)
#     7 usec#    _20 := algebra.kdifference(_13=<tmp_1372>, _18=<tmp_516>)
#     3 usec#    _22 := algebra.kunion(_20=<tmp_1374>, _18=<tmp_516>)
#     1 usec#    _26 := bat.reverse(_24=<tmp_514>)
#     3 usec#    _28 := algebra.kdifference(_22=<tmp_1375>, _26=<~tmp_514>)
#     4 usec#    _33 := algebra.markT(_28=<tmp_1376>, _31=0@0)
#     0 usec#    _35 := bat.reverse(_33=<tmp_1377>)
#    22 usec#    _37 := algebra.join(_35=<~tmp_1377>, _22=<tmp_1375>)
#     1 usec#    _47 := bat.reverse(_45=<tmp_731>)
#     4 usec#    _49 := algebra.kdifference(_39=<tmp_732>, _47=<~tmp_731>)
#     2 usec#    _53 := algebra.markT(_49=<tmp_1400>, _31=0@0)
#     0 usec#    _55 := bat.reverse(_53=<tmp_1402>)
#     4 usec#    _57 := algebra.join(_55=<~tmp_1402>, _39=<tmp_732>)
#    12 usec#    _59 := bat.setWriteMode(_37=<tmp_1401>)
#     2 usec#    bat.append(_59=<tmp_1401>, _57=<tmp_1403>)
#     8 usec#    _65 := algebra.markT(_59=<tmp_1401>, _31=0@0)
#     0 usec#    _67 := bat.reverse(_65=<tmp_1405>)
#     9 usec#    _69 := algebra.join(_67=<~tmp_1405>, _59=<tmp_1401>)
#     2 usec#    _74 := algebra.markT(_69=<tmp_1406>, _72=0@0)
#     1 usec#    _76 := bat.reverse(_74=<tmp_1404>)
#     7 usec#    _78 := algebra.join(_76=<~tmp_1404>, _69=<tmp_1406>)
#     1 usec#    _80 := aggr.count(_78=<tmp_1410>)
&1 0 1 1 1
# sys.tables # table_name
# count_id # name
# int # type
# 2 # length
[ 27    ]
#    20 usec#    sql.exportValue(_83=1, _84="sys.tables", _85="count_id", _86="int", _87=32
#     2 usec#    sql_cache.s3_0()
>
```

3.5 Optimizer Control

The code produced by MonetDB/SQL is massaged by several code optimizers to arrive at the best possible plan for evaluation. However, for development purposes and the rare case that more control is needed, the SQL session variable `optimizer` can be set to a MAL instruction list to identify the optimizer steps needed. [todo]

```
>select optimizer;
#--------------------#
```

```
# t                  # name
# str                # type
#--------------------#
[ "optimizer.sql();" ]
>set optimizer="optimizer.factorize();";
>explain select 1;
```

The final result the optimizer steps become visible using the `explain` statement modifier. Alternatively, the optimizer script line may include a call to `optimizer.showPlan();`.

3.6 Overlaying the BAT storage

The SQL implementation exploits many facilities of the underlying MonetDB kernel. Most notably, its efficient column-wise storage scheme with associated algebraic primitives. However, this kernel can also be programmed directly using the MonetDB Assembler Language (MAL), which provides a much richer set of operators then strictly necessary for a correct implementation and execution of SQL

Although the BATs used for SQL storage can be manipulated directly at the MAL layer, it is strongly discouraged. For, SQL uses a rather complex administration to realise transaction safety. Working at the MAL layer directly could easily compromise the integrity of the system as a whole.

However, there are cases where access to a collection of BATs from the SQL environment can be handy. For example, to inspect some of the system tables maintained in the kernel. This functionality is realised using an variation of the `view statement`. It is best illustrated with a short example.

First, create in MonetDB the BATs of interest. Only void-BATs are allowed and the user should guarantee that they are aligned.

```
a:=bat.new(:void,:int);
b:=bat.new(:void,:str);
bat.setSequenceBase(a,0@0);
bat.setSequenceBase(b,0@0);
bat.setPersistent(a);
bat.setPersistent(b);
bat.setName(a,"age");
bat.setName(b,"name");
```

These BATs can be filled with attribute values, but make sure that all BATs are given an equal number of tuples. To conclude, commit the database to disk, whereafter the BATs become available to SQL, once you restart Mserver/SQL.

```
bat.append(a, 32);
bat.append(b, "John");
bat.append(a, 31);
bat.append(b, "Mary");
transaction.commit(a,b);
```

The BATs are made visible for read-only access using the a column specification. As it normally expects identifiers, we have to use double quotes instead.

```
CREATE VIEW friends ("age","name") AS BATS;
```

At this point the content of the BATs can be queried using ordinary SQL queries. Updates are prohibited.

4 XQuery

Give a 3 page introduction on how to use XQuery

5 User Interfaces

5.1 Mapi Client Calling Arguments

The `mclient` program provides a textual interface to a MonetDB server. 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 establing a guest connection to a SQL database at a default server running on the localhost.

A blocked mode interaction permits assemblage of multiple instructions in a buffer before it is shipped to the server for execution.

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

```
mclient [ options ]
```

Options are:

```
-b t/f      | --blocked=true/false /* blocked mode */
-D          | --dump           /* dump sql database */
-e          | --error          /* exit on error */
-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  /* {mal,sql,mil,xquery} */
-P passwd   | --passwd=passwd  /* password */
-p portnr   | --port=portnr    /* port to connect to */
-q          | --quiet          /* don't print welcome message [default=on]*/
-s stmt     | --statement=stmt /* run single statement */
-T          | --time           /* time commands */
-t          | --trace          /* trace Monet interaction */
-u user     | --user=user      /* user id [default=monetdb] */
-?          | --help           /* show this usage message */
```

5.2 Jdbc Client Calling Arguments

The textual client using the JDBC protocol comes with several options to fine-tune the interaction with the database server. A synopsis of the calling arguments is given below

```
mjclient [-h host[:port]] [-p port] \
[-f file] [-u user] [-l language] [-b [database]] \
[-d [table]] [-e] [-X<opt>]
```

but also using long option equivalents –host –port –file –user –language –dump –echo –database are allowed. Arguments may be written directly after the option like -p45123.

If no host and port are given, localhost and 45123 (=sql_port default) are assumed. An `.monetdb` file may exist in the user's home directory. This file can contain preferences to use each time mjclient is started. Options given on the command line override the preferences file. The `.monetdb` file syntax is `<option>=<value>` where option is one of the options host, port, file, mode debug, or password. Note that the last one is perilous and therefore not available as command line option. If no input file is given using the -f flag, an interactive session is started on the terminal.

OPTIONS

`-h --host` The hostname of the host that runs the MonetDB database. A port number can be supplied by use of a colon, i.e. -h somehost:12345.

`-p --port` The port number to connect to.

`-f --file` A file name to use either for reading or writing. The file will be used for writing when dump mode is used (-d –dump). In read mode, the file can also be an URL pointing to a plain text file that is optionally gzip compressed.

`-u --user` The username to use when connecting to the database.

`-l --language`
Use the given language, for example 'xquery'.

`-d --dump` Dumps the given table(s), or the complete database if none given.

`--help` This screen.

`-e --echo` Also outputs the contents of the input file, if any.

`-b --database`
Try to connect to the given database (only makes sense if connecting to a DatabasePool or equivalent process).

EXTRA OPTIONS

`-Xdebug` Writes a transmission log to disk for debugging purposes. If a file name is given, it is used, otherwise a file called monet<timestamp>.log is created. A given file will never be overwritten; instead a unique variation of the file is used.

`-Xblksize`
Specifies the blocksize when using block mode, given in bytes.

`-Xoutput` The output mode when dumping. Default is sql, xml may be used for an experimental XML output.

`-Xbatching`
Indicates that a batch should be used instead of direct communication with the server for each statement. If a number is given, it is used as batch size. I.e. 8000 would execute the contents on the batch after each 8000 read rows. Batching can greatly speedup the process of restoring a database dump.

5.3 Console and Mapi Client

5.3.1 Online Help

Online help on the modules and functions can be obtained using manual commands and the readline key-binding '`<TAB><TAB>`'. The argument is a (partial) operator call, which is looked up in the symbol table. If the pattern includes a '(' it also displays the signature for each match. The `address` and `address` attributes are also shown if the call contains the closing bracket ')'.

```
>manual.completion("bat.is");
bat.isSynced
bat.isCached
bat.isPersistent
bat.isTransient
bat.isSortedReverse
bat.isSorted
bat.isaSet
bat.isaKey
>manual.help("bat.isSorted(");
command bat.isSorted(b:bat[:any_1,:any_2]):bit
>manual.help("bat.isSorted()");
command bat.isSorted(b:bat[:any_1,:any_2]):bit address BKCisSorted;
Returns whether a BAT is ordered on head or not.
>manual.search("isSorted()");
command bat.isSortedReverse(b:bat[:any_1,:any_2]):bit address BKCisSortedReverse;
Returns whether a BAT is ordered on head or not.
command bat.isSorted(b:bat[:any_1,:any_2]):bit address BKCisSorted;
Returns whether a BAT is ordered on head or not.
```

Keyword based lookup is supported by the operation `manual.search`; Additional routines are available in the `inspect` module to built reflexive code.

The module and function names can be replaced by the wildcard character '*'. General regulat pattern matching is not supported.

```
> *.print(<TAB><TAB>
command color.print(c:color):void
pattern array.print(a:bat[:any_1,:any_2],b:bat[:any_1,:int]...):void
pattern io.print(b1:bat[:any_1,:any]...):int
pattern io.print(order:int,b:bat[:any_1,:any],b2:bat[:any_1,:any]...):int
pattern io.print(val:any_1):int
pattern io.print(val:any_1,lst:any...):int
pattern io.print(val:bat[:any_1,:any_2]):int
```

5.4 Data Management Tools

There are many user-friendly tools to interact with a SQL database server. Although some tools are targetted to a single backend server, the systems described below have been successfully linked with MonetDB/SQL. For all systems considered, make sure you have Mserver installed on your system and that you can access the server using the ref(JDBC client).

5.4.1 Aqua Data Studio

Aqua Data Studio is a graphical user interface to interact with MonetDB/SQL. It is available on Windows, Linux, and MacOS platforms from the distribution site. Download the Version 4.5 executable and install the software.

The first step to make Aqua Data Studio aware of MonetDB is to register the database server . Go to the `Server->Register Server` panel and select the Generic- JDBC RDBMS theme. It requires the following additional field settings:

Name:	MonetDB SQL
Type:	*Whatever you want*
Login Name:	monetdb
Password:	monetdb
URL	jdbc:monetdb://localhost/database
Driver:	nl.cwi.monetdb.jdbc.MonetDriver
Driver Location:	C:\Program Files\CWI\MonetDB\share\MonetDB\lib\monetdb-1.3-jdbc.jar

The location of the JDBC driver under Linux and OSX is by default /usr/share/MonetDB/lib/monetdb-1.3-jdbc.jar.

Once the settings has been completed, start the MonetDB server and try to connect. If necessary extend the heap size of your java engine, e.g. use -Xmx1024M .

Consult your system administrator if other MonetDB user creditials and locality settings are required.

5.4.2 DbVisualizer

DbVisualizer is a platform independent tool aimed to simplify database development and management for database administrators and developers. It's a very cool tool (can even draw dependency graphs based on the schema and foreign keys.

Free, personal use versions are available from their website. Download and install the software. The following scheme works for version 4.3.6 and version 5.0.

After starting DbVisualizer for the first time, it will load its default welcome screen. Before doing anything, open the Driver Manager, using the menu Tools->Driver Manager... In the Driver Manager add a new Driver using Driver->Create Driver... Type the name of the driver in the Name field, e.g. MonetDB. Type the URL format for the driver: jdbc:monetdb://hostname/database In the Driver File Paths box, follow the directions to load a JAR file that contains the JDBC driver. Use the MonetDB JDBC driver that came with your distribution, typically installed in ${prefix }/share/MonetDB/lib/monetdb-X.Y-jdbc.jar.

After adding, the window should list the driver class nl.cwi.monetdb.jdbc.MonetDriver and automatically fills it in in the Driver Class field. Close the Driver Manager window. Back in the main window, in the Getting Started box, click the Tools->Connection Wizard link. A connection Wizard will pop up, and you will think you are working on a Microsoft Powered computer, no matter which operating system you use. Type the name for the database, e.g. MonetDB and click "Next". Select the just added monetdb Database Driver from the drop down list and again press "Next". Go grab a cup of coffee, when you found out the Wizard not only looks like a Microsoft Product(tm), but also acts like one. Try to hide your frustration and cancel the Wizard and continue the manual way.

From the menu select Database->Create Database Connection. A popup dialog will try to persue you to use the wizard. You know better so, click "No". Fill in the name for the connection in the Alias field, e.g. MonetDB. Select the monetdb driver from the list. Copy the default URL by clicking on the "URL Format: ..." text field and change it to reflect the right hostname (usually localhost will do). Fill in the default userid and password (monetdb). Press the "Connect" button. It will report the database being used and the JDBC driver in use. In the left pane the monetdb database now becomes available from browsing.

Explore the application and have fun!

A caveat of the free-version system is its performance on SQL scripts. They are sent as a single string to the server for execution. This is not the most optimal situation for MonetDB. Running a batch script is better started from the MonetDB SQL client.

5.4.3 iSQL-Viewer

Another open-source graphical user interface is the iSQL-Viewer. It runs on any Java-enabled platform.

Once installed and started you have the option define a service (or let the system find one itself). [Ignore the autodetect of services option]

The Tools->Service Manager choice brings up a form to provide the detail for the MonetDB connections. Click the third button in toolbar and select 'Local Service' to define a service Immediately select the tab named 'Resource' and add the location of the MonetDB JDBC driver, installed by default in C:\Program Files\MonetDB\share\MonetDB\lib\MonetDB_JDBC.jar. under Windows and and in /usr/share/MonetDB/lib/MonetDB_JDBC.jar under Linux and OSX

Go back to tab 'General' and enter the following items:

Connection Name	MonetDB SQL
JDBC Driver	nl.cwi.monetdb.jdbc.MonetDriver
JDBC URL	jdbc:monetdb://localhost/database
User name	monetdb
Password	monetdb

Warning, disable the authentication prompt if you can not edit the user and password fields.

In the `Tools>Service Manager>Resource` panel enter the location of the MonetDB JDBC driver `C:\Program Files\MonetDB\share\MonetDB\lib\MonetDB_JDBC.jar`.

Documentation and tutorial on iSQL-Viewer are available on their website.

5.5 Application Programming Interfaces

MonetDB comes with a complete set of programming libraries. Their basis is the Mapi library, which describes the protocol understood by the server. The Perl,PHP, and Python library are mostly wrappers around the Mapi routines.

The programming interface is based on a client-server architecture, where the client program connects to a server using a TCP-IP connection to exchange commands and receives ansers. The underlying protocol uses plain UTF-8 data for ease of use and debugging. This leads to publically visible information exchange over a network, which may be undesirable. Therefore, a private and secure channel can be set up with the Secure Socket Layer functionality being offered.

A more tightly connection between application logic and database server is described in Section 6.7 [Embedded MonetDB], page 65.

6 The MonetDB Programming Interface

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,...

6.1 Sample MAPI Application

The database driver implementation given in this document focuses on developing applications in C/C++. The command collection has been chosen to align with common practice, i.e. queries follow a prepare, execute, and fetch_row paradigm. The output is considered a regular table. An example of a mini application below illustrates the main operations.

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

#define die(dbh,hdl) (hdl?mapi_explain_query(hdl,stderr):              \
                      dbh?mapi_explain(dbh,stderr):                    \
                          fprintf(stderr,"command failed\n"),          \
                  exit(-1))

int main(int argc, char **argv)
{
    Mapi dbh;
    MapiHdl hdl = NULL;

    dbh = mapi_connect("localhost", 50000, "monetdb", "monetdb", "sql");
    if (mapi_error(dbh))
        die(dbh, hdl);

    if ((hdl = mapi_query(dbh, "create table emp(name varchar(20), age int)")) == NULL
        || mapi_error(dbh) != MOK)
        die(dbh, hdl);
    if (mapi_close_handle(hdl) != MOK)
        die(dbh, hdl);
    if ((hdl = mapi_query(dbh, "insert into emp values('John', 23)")) == NULL
        || mapi_error(dbh) != MOK)
        die(dbh, hdl);
    mapi_close_handle(hdl);
    if (mapi_error(dbh) != MOK)
        die(dbh, hdl);
    if ((hdl = mapi_query(dbh, "insert into emp values('Mary', 22)")) == NULL
        || mapi_error(dbh) != MOK)
        die(dbh, hdl);
    mapi_close_handle(hdl);
    if (mapi_error(dbh) != MOK)
        die(dbh, hdl);
```

```
    if ((hdl = mapi_query(dbh, "select * from emp")) == NULL
        || mapi_error(dbh) != MOK)
        die(dbh, hdl);

    while (mapi_fetch_row(hdl)) {
        char *nme = mapi_fetch_field(hdl, 0);
        char *age = mapi_fetch_field(hdl, 1);
        printf("%s is %s\n", nme, age);
    }
    if (mapi_error(dbh) != MOK)
        die(dbh, hdl);
    mapi_close_handle(hdl);
    if (mapi_error(dbh) != MOK)
        die(dbh, hdl);
    mapi_disconnect(dbh);

    return 0;
}
```

The `mapi_connect()` operation establishes a communication channel with a running server on the local machine. The user name is "monetdb" with password "monetdb". The query language interface is either "sql", "mil" or "xquery". Future versions are expected to recognize also "ram" and "mal".

Errors on the interaction can be captured using `mapi_error()`, possibly followed by a request to dump a short error message explanation on a standard file location. It has been abstracted away in a macro.

Provided we can establish a connection, the interaction proceeds as in many similar application development packages. Queries are shipped for execution using `mapi_query()` and an answer table can be consumed one row at a time. In many cases these functions suffice.

The Mapi interface provides caching of rows at the client side. `mapi_query()` will load tuples into the cache, after which they can be read repeatedly using `mapi_fetch_row()` or directly accessed (`mapi_seek_row()`). This facility is particularly handy when small, but stable query results are repeatedly used in the client program.

To ease communication between application code and the cache entries, the user can bind the C-variables both for input and output to the query parameters, and output columns, respectively. The query parameters are indicated by '?' and may appear anywhere in the query template.

6.2 Caveats

The Mapi library expects complete lines from the server as answers to query actions. Incomplete lines leads to Mapi waiting forever on the server. Thus formatted printing is discouraged in favor of tabular printing as offered by the `table.print()` commands.

6.3 Compilation

The Mapi application uses include files found in the MonetDB distribution directory or at a central location on your machine. Assuming the location of the distribution is marked in the environment variable $MONETDB_PREFIX, the following actions are needed to get a working program.

```
cc sample.c -I$MONETDB_PREFIX \
        -I$MONETDB_PREFIX/include/common \
        -I$MONETDB_PREFIX/include/mapi  \
        -L$MONETDB_PREFIX/lib \
        -lMapi -lutils -lstream \
        -Wl,--rpath -Wl,$MONETDB_PREFIX/lib/MonetDB:$MONETDB_PREFIX/lib \
        -o sample
```

6.4 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_connect_ssl()	Connect to a Mserver using Secure Socket Layer (SSL)
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_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_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_trace()	Get trace flag
mapi_get_user()	Current user name
mapi_next_result()	Go to next result set
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_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_trace()	Set trace flag
mapi_trace_log()	Keep log of interaction
mapi_virtual_result()	Submit a virtual result set
mapi_unquote()	remove escaped characters

6.5 Mapi Library

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.

6.5.1 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);
```

6.6 Mapi Function Reference

6.6.1 Connecting and Disconnecting

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

  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.
- Mapi mapi_connect_ssl(const char *host, int port, const char *username, const char *password, const char *lang)

  Setup a connection with a Mserver at a *host*:*port* and login with *username* and *password*. The connection is made using the Secure Socket Layer (SSL) and hence all data transfers to and from the server are encrypted. The parameters are the same as in mapi_connect().

- 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.

6.6.2 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.

6.6.3 Getting Results

- int mapi_get_field_count(Mapi mid)

  Return the number of fields in the current row.
- int mapi_get_row_count(Mapi mid)

  If possible, return the number of rows in the last select call. A -1 is returned if this information is not available.
- int 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()`.
- int 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, int rownr, int whence)

  Reset the row pointer to the requested row number. If whence is `MAPI_SEEK_SET` (0), rownr is the absolute row number (0 being the first row); if whence is `MAPI_SEEK_CUR` (1), rownr is relative to the current row; if whence is MAPI_SEEK_END (2), 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().
- MapiMsg mapi_next_result(MapiHdl hdl)

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

6.6.4 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.

6.6.5 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.

6.6.6 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_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_trace(Mapi mid, int flag)

  Set the trace flag to monitor interaction with the server.
- int mapi_get_trace(Mapi mid)

  Return the current value of the trace flag.
- MapiMsg mapi_trace_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. It is not intended for recovery.

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_version(Mapi mid)
- int mapi_get_versionId(Mapi mid)
- char *mapi_get_motd(Mapi mid)
- char **mapi_tables(Mapi mid)

  Return a list of accessible database tables.
- char **mapi_fields(Mapi mid)

  Return a list of accessible tables fields. This can also be obtained by inspecting the field descriptor returned by `mapi_fetch_field()`.

6.7 Embedded MonetDB

The Embedded MonetDB version is optimized for running on small board computers as a database back-end for a single client. It is of particular interest if you need database functionality within a limited application setting, e.g a self-contained database distributed as part of the application. Within this context, much of the code to facilitate and protect concurrent use of the kernel can be disabled. For example, the communication overhead of client-server TCP-IP interaction is removed. Moreover, locking of critical resources in the kernel is not needed anymore, which results in significant performance improvements.

The approach taken is to wrap a server such that the interaction between client code and server can still follow the Mapi protocol. It leads to a C-program with calls to the Mapi library routines, which provides some protection against havoq behaviour. From a programming view, it differs from a client-server application in the startup and (implicit) termination.

You normally only have to change the call `mapi_connect()` into `embedded_sql()` (or `embedded_mal()`). It requires an optional argument list to refine the environment variables used by the server. In combination with the header file `embeddedclient.h` it provides the basis to compile and link the program.

The behavior of an embedded SQL program can be simulated with a server started as follows:

```
mserver5 --set embedded=yes --dbinit="include sql;" &
```

As a result, the server starts in 'daemon' mode, loads the SQL support library, and waits for a connection. Only one connection is permitted.

6.7.1 Mbedded Example

A minimalistic embedded application is shown below. It creates a temporary table in the database, fills it, and retrieves the records for some statistics gathering.

The key operation is `embedded_sql()` which takes an optional environment argument list. Upon success of this call, there will be a separate server thread running in the same user space to handle the database requests. A short-circuit interaction is established between the application and the kernel using in memory buffers.

The body of the program consists of the Mapi calls you have already seen (see Chapter 6 [The Mapi Library], page 56). It terminates with a call to `mapi_disconnect()` which lets the MonetDB thread gracefully die.

The tight coupling of application and kernel code also carries some dangers. Many of the MonetDB data structures can be directly accessed, or calls to the kernel routines are possible. It is highly advised to stick to the Mapi interaction protocol. It gives a little more protection against malicious behavior or unintended side-effects.

```
#include <embeddedclient.h>

#define die(dbh,hdl) (hdl?mapi_explain_result(hdl,stderr):      \
              dbh?mapi_explain(dbh,stderr):         \
                  fprintf(stderr,"command failed\n"),   \
              exit(-1))

#define close_handle(X,Y) if (mapi_close_handle(X) != MOK) die(X, Y);

int
main()
{
    Mapi dbh;
    MapiHdl hdl = NULL;
    int i;
```

```
    dbh= embedded_sql(NULL,0);
    if (dbh == NULL || mapi_error(dbh))
        die(dbh, hdl);

    /* switch off autocommit */
    if (mapi_setAutocommit(dbh, 0) != MOK || mapi_error(dbh))
        die(dbh,NULL);

    if ((hdl = mapi_query(dbh, "create table emp"
                                " (name varchar(20),age int)")) == NULL || mapi_error(
        die(dbh, hdl);
    close_handle(dbh,hdl);

    for(i=0; i< 1000; i++) {
        char query[100];
        snprintf(query, 100, "insert into emp values('user%d', %d)", i, i % 82);
        if ((hdl = mapi_query(dbh, query)) == NULL || mapi_error(dbh))
            die(dbh, hdl);
        close_handle(dbh,hdl);
    }

    if ((hdl = mapi_query(dbh, "select * from emp")) == NULL || mapi_error(dbh))
        die(dbh, hdl);

    i=0;
    while (mapi_fetch_row(hdl))  {
        char *age = mapi_fetch_field(hdl, 1);
        i= i+ atoi(age);
    }
    if (mapi_error(dbh))
        die(dbh, hdl);
    close_handle(dbh,hdl);
    printf("The footprint is %d Mb \n",i);

    mapi_disconnect(dbh);
    return 0;
}
```

The embedded MonetDB engine is available as the library `libembedded_sql.a` (and `libembedded_mal.a`) to be linked with a C-program. Provided the programming environment have been initialized properly, it suffices to prepare the embedded application using

```
gcc myprog.c -o myprog `monetdb5-config --cflags --libs`
```

You might also write a Makefile to build the program as follows.[todo]

```
CC= gcc
INCLUDE=`monetdb5-config --cflags`
LIBS=`monetdb5-config --libs`
myprog: myprog.o
```

```
        ${CC} myprog.o -o myprog ${LIBS}
myprog.o : myprog.c
        ${CC} -c ${INCLUDE} myprog.c

clean: myprog.o
        rm -f myprog myprog.o
```

The configuration parameters for the server are read from its default location in the file system. In an embedded setting this location may not be accessible. It requires calls to `embedded_option()` before you asks for the instantiation of the server code itself. The code snippet below illustrate how our example is given hardwired knowledge on the desired settings:

```
main(){
...
        Mbedded_option("dbfarm",".");
        Mbedded_option("dbname","demo");
        Mbedded_option("user","guest");
        Mbedded_option("password","anonymous");
...
        Mbedded_sql(NULL,0);
```

For an overview of the system configuration parameters see XYZ.

6.7.2 Limitations for Embedded MonetDB

In embedded applications the memory footprint is a factor of concern. The raw footprint as delivered by the Unix `size` command is often used. It is, however, also easily misleading, because the footprint depends on both the code segments and buffered database partitions in use. Therefore it makes sense to experiment with a minimal, but functionally complete application to decide if the resources limitations are obeyed.

The minimal static footprint of MonetDB is about 16 Mb (+ ca 4Mb for SQL). After module loading the space quickly grows to about 60Mb. *This footprint should be reduced.*

A better frame of reference for embedded applications is our sample program, which is a simple, yet complete embedded application inspired by an MP3 player. The table below illustrates some basic properties on different embedded SQL platforms.

	Mbedded	SQLite	MySQL	PostgreSQL
Prepare time				
Max memory				
CPU time				

The `Prepare time` denotes the compilation and link time on a state-of-the-art PC.

The embedded application world calls for many, highly specialized enhancements. It is often well worth the effort to carve out the functionality needed from the MonetDB software packages. The easiest solution to limit the functionality and reduce resource consumption is to reduce the modules loaded. This requires patches to the startup scripts.

The benefit of an embedded database application also comes with limitations. The one and foremost limitation of embedded MonetDB is that the first application accessing the database effectively locks out any other concurrent use. Even in those situations where concurrent applications merely read the database, or create privately held tables.

7 Monetdb Assembler Language

The primary textual interface to the Monetdb kernel is a simple, assembler-like language, called MAL. The language reflects the virtual machine architecture around the kernel libraries and has been designed for speed of parsing, ease of analysis, and ease of target compilation by query compilers. The language is not meant as a primary programming language, or scripting language. Such use is even discouraged.

Furthermore, a MAL program is considered a specification of intended computation and data flow behavior. It should be understood that its actual evaluation depends on the execution paradigm chosen in the scenario. The program blocks can both be interpreted as ordered sequences of assembler instructions, or as a representation of a data-flow graph that should be resolved in a dataflow driven manner. The language syntax uses a functional style definition of actions and mark those that affect the flow explicitly. Flow of control keywords identify a point to chance the interpretation and denote a synchronization point.

MAL is the target language for query compilers, such as the SQL and XQuery front-ends. Even simple SQL queries generate a long sequence of MAL instructions. They represent both the administrative actions to ensure binding and transaction control, the flow dependencies to produce the query result, and the steps needed to prepare the result set for delivery to the front-end.

Only when the algebraic structure is too limited (e.g. updates), or the database back-end lacks feasible builtin bulk operators, one has to rely on more detailed flow of control primitives. But even in that case, the basic blocks to be processed by a MAL back-end are considered large, e.g. tens of simple bulk assignment instructions.

The remainder of this chapter provide a concise overview of the language features and illustrative examples.

7.1 MAL instructions

MAL instructions have purposely a simple format. Each instruction denotes a data-flow statement. It is syntactically represented by an assignment, where an expression (function call) delivers results to multiple target variables. The assignment patterns recognized are illustrated below.

```
(t1,..,t32) := module.fcn(a1,..,a32);
t1 := module.fcn(a1,..,a32);
t1 := v1 operator v2;
t1 := constant;
(t1,..,tn) := (a1,..,an);
```

Variables are implicitly defined upon first use and take on a type through a type classifier or inherit it from the context in which they are used, see Chapter 8 [Type resolution], page 77. Operators are grouped into user defined modules Section 10.2 [MAL Modules], page 83. Binary arithmetic operations are merely provided as a short-hand, e.g. the expression `t:=2+2` is converted directly into `t:= calc.+(2,2)`.

Target variables are optional. The compiler introduces temporary variables to hold the result of the expression upon need. They act as data sinks in the flow graph.

For parsing simplicity, each instruction fits on a single line. Comments start with a sharp '#' and continues to the end of the line. They are retained in the internal code representation to ease debugging of compiler generated MAL programs.

The data structure to represent a MAL block is kept simple. It carries a sequence of MAL statements and a symbol table. The MAL instruction is a code byte string overlaid with the instruction pattern, which contains references into the symbol tables and administrative data for the interpreter.

This method leads to a large allocated block, which can be easily freed, and pattern makes it possible to accommodate a variable argument list. Variable- and statement- block together describe the static part of a MAL procedure. It carries carry enough information to produce a listing and to aid symbolic debugging.

7.2 Flow of control

The flow of control within a MAL program block can be changed by tagging an assignment with either `return`, `yield`, `barrier`, `catch`, `leave`, `redo`, or `exit`.

The flow modifiers `return` and `yield` mark the end of a call and return one or more results to the calling environment. The `return` and `yield` are followed by a target list or an assignment, which is executed first.

The `barrier` (`catch`) and `exit` pair mark a guarded statement block. They may be nested to form a proper hierarchy identified by their target variables.

The `leave` and `redo` are conditional flow modifiers. Their first target variable is also denoted as the control variable. It is used after the assignment statement has been evaluated to decide on the flow-of-control action to be taken. Built-in controls exists for booleans and numeric values. The barrier block is opened when the control variable holds true, when its numeric value >= 0, or when it is a non-empty string. The `nil` value blocks entry in all cases.

Once inside the barrier you have an option to prematurely `leave` it at the exit statement or to `redo` interpretation just after the corresponding barrier statement. Much like 'break' and 'continue' statements in the programming language C. The action is taken when the condition is met.

The `exit` marks the exit for a block. Its optional assignment can be used to re-initialize the barrier control variables or wrap-up any related administration.

A control block is recognized by its target variable set, which implies that nesting on the same target set is prohibited.

The barrier blocks can be properly nested to form a hierarchy of basic blocks. The control flow within and between blocks is simple enough to deal with during an optimizer stage. The `redo` and `leave` statements mark the partial end of a block. Statements within these blocks can be re-arranged according to the data-flow dependencies. The order of partial blocks can not be changed that easily. It depends on the mutual exclusion of the data flows within each partial block.

Common guarded blocks in imperative languages are the for-loop and if-then-else constructs. They can be simulated as follows.

Consider the statement `for(i=1;i<10;i++) print(i)`. The (optimized) MAL block to implement this becomes:

```
       i:= 1;
barrier B:= i<10;
       print(i);
       i:= i+1;
redo B:= i<10;
exit B;
```

Translation of the statement `if(i<1) print("ok"); else print("wrong");` becomes:

```
      i:=1;
barrier ifpart:= i<1;
      print("ok");
exit ifpart;
barrier elsepart:= i>=1;
      print("wrong");
exit elsepart;
```

Note that both guarded blocks can be interchanged without affecting the outcome. Moreover, neither block would have been entered if the variable happens to be assigned `nil`.

The primitives are sufficient to model a wide variety of iterators, whose pattern look like:

```
barrier i:= newIterator(T);
      elm:= getElement(T,i);
      ...
      leave i:= noMoreElements(T);
      ...
      redo i:= hasMoreElements(T);
exit i:= exitIterator(T);
```

The semantics obeyed by the iterator implementations is as follows. The redo expression updates the target variable *i* and control proceeds at the first statement after the barrier when the barrier is opened by *i*. If the barrier could not be re-opened, execution proceeds with the first statement after the redo. Likewise, the leave control statement skips to the exit when the control variable *i* shows a closed barrier block. Otherwise, it continues with the next instruction. Note, in both failed cases the control variable is possibly changed.

A recurring situation is to iterate over the elements in a BAT. This is supported by an iterator implementation in the bbp module as follows:

```
barrier (idx,hd,tl):= newIterator(B);
      ...
      redo (idx,hd,tl):= hasMoreElements(B);
exit i;
```

Where idx is an integer to denote the row in the BAT, hd and tl denote values of the current element. This scheme is more efficient

7.3 MAL functions

MAL comes with a standard functional abstraction scheme. MAL functions are represented by MAL instruction lists, enclosed by a `function` and `end` statement. The `function` designates its signature, i.e. it lists the arguments and their types. The `end` statement marks the end of this sequence. Its argument is the function name.

An illustrative example is:

```
function helloWorld(msg:str):str;
    io.print(msg);
    return msg;
end hellowWorld;
```

The functional abstraction scheme comes with several variations: commands, patterns, and factories. They are discussed shortly.

7.3.1 Polymorphic Functions

MAL supports polymorphic functions using type variables. The type variable is denoted by `any` and an optional index. Each time a polymorphic MAL function is called, the symbol table is first inspected for the matching strongly typed version. If non exists, a copy of the polymorphic routine is generated, whereafter the type variables are replaced with their concrete types. The function body is immediately type checked and, if no errors occured, added to the symbol table.

The generic type variable `any` designates an unknown type, which may be filled at type resolution time. Unlike indexed polymorphic type arguments, `any` type arguments match possibly with different concrete types.

An example of a parameterised function is shown below:

```
function helloWorld(msg:any_1):any_1;
io.print(msg);
return msg;
```

The type variables ensure that the return type equals the argument type. Type variables can be used at any place where a type name is permitted. Beware that polymorphic type variables are propagated throughout the function body. This may invalidate type resolutions taken Chapter 8 [Type resolution], page 77.

This version of `helloWorld` can also be used for other arguments types, i.e. `bit,sht,lng,flt,dbl,...`. For example, calling `helloWorld(3.14:flt)` echoes a float value.

7.3.2 Commands and patterns

The MAL function body can also be implemented with a C-function. They are introduced to the MAL type checker by providing their signature and an `address` qualifier for linkage.

We distinguish both `command` and `pattern` C-function blocks. They differ in the information accessible at run time. The `command` variant calls the underlying C-function, passing pointers to the arguments on the MAL runtime stack. The `pattern` command is passed pointers to the MAL definition block, the runtime stack, and the instruction itself. It can be used to analyse the types of the arguments directly.

For example, the definitions below link the kernel routine `BKCinsert_bun` with the function `bat.insert()`. It does not fully specify the result type. The `io.print()` pattern applies to any BAT argument list, provided they match on the head column type.

```
command bat.insert(b:bat[:any_1,:any_2], ht:any_1, tt:any_2) :bat[:any_1,:any_2]
address BKCinsert_bun;
```

```
pattern io.print(b1:bat[:any_1,:any]...):int
address IOtable;
```

7.3.3 Lifespan analysis

Optimizers may be interested in the characteristic of the barrier blocks for making a decision. The variables have a lifespan in the code blocks, denoted by properties beginLifespan,endLifespan. The beginLifespan denotes the intruction where it receives its first value, the endLifespan the last instruction in which it was used as operand or target.

If, however, the last use lies within a BARRIER block, we can not be sure about its end of life status, because a block redo may implictly revive it. For these situations we associate the endLifespan with the block exit.

In many cases, we have to determine if the lifespan interferes with a optimization decision being prepared. The lifespan is calculated once at the beginning of the optimizer sequence. It should either be maintained to reflect the most accurate situation while optimizing the code base. In particular it means that any move/remove/addition of an instruction calls for either a recalculation or delta propagation. Unclear what will be the best strategy. For the time being we just recalc.

7.4 Factories

A convenient programming construct is the co-routine, which is specified as an ordinary function, but maintains its own state between calls, and permits re-entry other than by the first statement.

The random generator example is used to illustrate its definition and use.

```
factory random(seed:int,limit:int):int;
    rnd:=seed;
    lim:= limit;
barrier lim;
    leave lim:= lim-1;
    rnd:= rnd*125;
    yield rnd:= rnd % 32676;
    redo lim;
exit lim;
end random;
```

The first time this factory is called, a factory plant is created in the local system to handle the requests. The plant carries the stack frame and synchronizes access.

In this case it initializes the generator. The random number is generated and `yield` as a result of the call. The factory process is then put to sleep. The second call received by the factory wakes it up at the point where it went to sleep. In this case it will find a `redo` statement and produces the next random number. Note that also in this case a seed and limit value are expected, but they are ignored in the body. This factory can be called upon to generate at most 'limit' random numbers using the 'seed' to initialize the generator. Thereafter it is being removed, i.e. reset to the original state.

A cooperative group of factories can be readily constructed. For example, assume we would like the random factories to respond to both `random(seed,limit)` and `random()`. This can be defined as follows:

```
factory random(seed:int,limit:int):int;
    rnd:=seed;
    lim:= limit;
barrier lim;
    LEAVE lim:= lim-1;
    rnd:= rnd*125;
    YIELD rnd:= rnd % 32676;
    REDO lim;
exit lim;
end random;

factory random():int;
barrier forever:=true;
    yield random(0,0);
    redo forever;
exit forever;
end random;
```

7.4.1 Client support

For simple cases, e.g. implementation of a random function, it suffices to ensure that the state is secured between calls. But, in a database context there are multiple clients active. This means we have to be more precise on the relationship between a co-routine and the client for which it works.

The co-routine concept researched in Monet 5 is the notion of a 'factory', which consists of 'factory plants' at possibly different locations and with different policies to handle client requests. Factory management is limited to its owner, which is derived from the module in which it is placed. By default Admin is the owner of all modules.

The factory produces elements for multiple clients. Sharing the factory state or even remote processing is up to the factory owner. They are set through properties for the factory plant.

The default policy is to instantiate one shared plant for each factory. If necessary, the factory can keep track of a client list to differentiate the states. A possible implementation would be:

```
factory random(seed:int,clientid:int):int;
    clt:= bat.new(:int,:int);
    bat.insert(clt,clientid,seed);
barrier always:=true;
    rnd:= algebra.find(clt,clientid);
catch   rnd; #failed to find client
    insert(clt,clientid,seed);
    rnd:= algebra.find(clt,clientid);
exit    rnd;
    rnd:= rnd * 125;
    rnd:= rnd % 32676;
    algebra.replace(clt,clientid,rnd);
    yield rnd;
```

```
    redo always;
exit always;
end random;
```

The operators to built client aware factories are, `factory.getCaller()`, which returns a client index, `factory.getModule()` and `factory.getFunction()`, which returns the identity of scope enclosed.

To illustrate, the client specific random generator can be shielded using the factory:

```
factory random(seed:int):int;
barrier always:=true;
    clientid:= factory.getCaller();
    yield user.random(seed, clientid);
    redo always;
exit always;
end random;
```

7.4.2 Complex factory examples

One interesting use of the factory scheme is to model a volcano-style query processor. Each node in the query tree is an iterator that calls upon the operands to produce a chunk, which are combined into a new chunk for concumption of the parent. The prototypical join(R,S) query illustrates it. The plan does not test for all boundary conditions, it merely implements a nested loop. The end of a sequence is identified by a NIL chunk.

```
factory query();
    Left:= sql.bind("relationA");
    Right:= sql.bind("relationB");
    rc:= sql.joinStep(Left,Right);
barrier rc!= nil;
    io.print(rc);
    rc:= sql.joinStep(Left,Right);
    redo rc!= nil;
exit rc;
end query;

#nested loop join
factory sql.joinStep(Left:bat[:any,:any],Right:bat[:any,:any]):bat[:any,:any];
    lc:= chopper.chunkStep(Left);
barrier outer:= lc != nil;
    rc:= chopper.chunkStep(Right);
    barrier inner:= rc != nil;
        chunk:= algebra.join(lc,rc);
        yield chunk;
        rc:= chopper.chunkStep(Right);
        redo inner:= rc != nil;
    exit inner;
    lc:= chopper.chunkStep(Left);
    redo outer:= lc != nil;
exit outer;
```

```
        # we have seen everything
        return nil;
    end joinStep;

    #factory for left branch
    factory chunkStepL(L:bat[:any,:any]):bat[:any,:any];
        i:= 0;
        j:= 20;
        cnt:= algebra.count(L);
    barrier outer:= j<cnt;
        chunk:= algebra.slice(L,i,j);
        i:= j;
        j:= i+ 20;
        yield chunk;
        redo loop:= j<cnt;
    exit outer;
        # send last portion
        chunk:= algebra.slice(L,i,cnt);
        yielD chunk;
        return nil;
    end chunkStep;

    #factory for right leg
    factory chunkStepR(L:bat[:any,:any]):bat[:any,:any];
```

So far we haven;t re-used the pattern that both legs are identical. This could be modeled by a generic chunk factory. Choosing a new factory for each query steps reduces the administrative overhead.

The Factory concept is still rather experimental and many questions should be considered, e.g. What is the lifetime of a factory? Does it persists after all clients has disappeared? What additional control do you need? Can you throw an exception to a Factory?

8 MAL Type Resolution

Given the interpretative nature of many of the MAL instructions, when and where type resolution takes place is a critical design issue. Performing it too late, i.e. at each instruction call, leads to performance problems if we derive the same information over and over again. However, many built-in operators have polymorphic typed signatures, so we cannot escape it altogether.

Consider the small illustrative MAL program:

```
function sample(nme:str, val:any_1):bit;
   c := 2 * 3;
   b := bbp.bind(nme);  #find a BAT
   h := algebra.select(b,val,val);
   t := aggr.count(h);
   x := io.print(t);
   y := io.print(val);
end sample;
```

The function definition is polymorphic typed on the 2nd argument, it becomes a concrete type upon invocation. The system could attempt a type check, but quickly runs into assumptions that generally do not hold. The first assignment can be type checked during parsing and a symbolic optimizer could even evaluate the expression once. Looking up a BAT in the buffer pool leads to an element `:bat[`*ht,tt*`]` where *ht* and *tt* are runtime dependent types, which means that the selection operation can not be type-checked immediately. It is an example of an embedded polypmorphic statement, which requires intervention of the user/optimizer to make the type explicit before the type resolver becomes active. The operation `count` can be checked, if it is given a BAT argument. This assumes that we can infer that 'h' is indeed a BAT, which requires assurance that `algebra.select` produces one. However, there are no rules to avoid addition of new operators, or to differentiate among different implementations based on the argument types. Since `print(t)` contains an undetermined typed argument we should postpone typechecking as well. The last print statement can be checked upon function invocation.

Life becomes really complex if the body contains a loop with variable types. For then we also have to keep track of the original state of the function. Or alternatively, type checking should consider the runtime stack rather than the function definition itself.

These examples give little room to achieve our prime objective, i.e. a fast and early type resolution scheme. Any non-polymorphic function can be type checked and marked type-safe upon completion. Type checking polymorphic functions are post-poned until a concrete type instance is known. It leads to a clone, which can be type checked and is entered into the symbol table.

8.1 Atomary types

MonetDB supports an extensible type system to accomodate a wide spectrum of database kernels and application needs. The type administration keeps track of their properties and provides access to the underlying implementations.

MAL recognizes the definition of a new atom type by replacing the `module` keyword with `atom`. Atoms definitions require special care, because their definition and properties should be communicated with the GDK kernel library. The commands defined in an `atom` block are screened as of interest to the gdk_atom library.

MonetDB comes with the hardwired types `bit,chr,sht,int,lng,oid,flt, dbl,str` and `bat`, the representation of a bat identifier. The kernel code has been optimized to deal with these types efficiently, i.e. without unnecessary function call overheads.

A small collection of user-defined `atom` types is shipped with the sysem. They implement types considered essential for end-user applications, such as `color, date, time, blob`, and `url`. They are implemented using the type extension mechanism described below. As such, they provide examples for future extensions. A concrete example is the 'blob' datatype in the MonetDB atom module library(see ../modules/atoms/blob.mx)

8.1.1 Defining your own types

For the courageous at heart, you may enter the difficult world of extending the kernel library. The easiest way is to derive the atom modules from one shipped in the source distributie. More involved atomary types require a study of the documentation associated with the atom structures (gdk_atoms), because you have to develop a handful routines complying with the signatures required in the GDK library. They are registered upon loading the `atom` module.

9 Boxed variables

Clients sessions often come with a global scope of variable settings. Access to these global variables should be easy, but they should also provide protection against concurrent update when the client wishes to perform parallel processing. Likewise, databases, query languages, etc. may define constants and variables accessible, e.g. relational schemas, to a selected user group.

The approach taken is to rely on persistent object spaces as pioniered in Lynda and -later- JavaSpaces. They are called boxes in MonetDB and act as managed containers for persistent variables.

Before a client program can interact with a box, it should open it, passing qualifying authorization information and parameters to instruct the box-manager of the intended use. A built-in box is implicitly opened when you request for its service.

At the end of a session, the box should be closed. Some box-managers may implement a lease-scheme to automatically close interaction with a client when the lease runs out. Likewise, the box can be notified when the last reference to a leased object ceases to exist.

A box can be extended with a new object using the function deposit(name) with name a local variable. The default implementation silently accepts any new definition of the box. If the variable was known already in the box, its value is overwritten.

A local copy of an object can be obtained using the pattern 'take(name,[param])', where name denotes the variable of interest. The type of the receiving variable should match the one known for the object. Whether an actual copy is produced or a reference to a shared object is returned is defined by the box manager.

The object is given back to the box manager calling 'release(name)'. It may update the content of the repository accordingly, release locks, and move the value to persistent store. Whatever the semantics of the box requires. [The default implementation is a no-op]

Finally, the object manager can be requested to 'discard(name)' a variable completely. The default implementation is to reclaim the space in the box.

Concurrency control, replication services, as well as access to remote stores may be delegated to a box manager. Depending on the intended semantics, the box manager may keep track of the clients holding links to this members, provide a traditional 2-phase locking scheme, optimistic control, or check-out/check-in scheme. In all cases, these management issues are transparant to the main thread (=client) of control, which operates on a temporary snapshot. For the time being we realize the managers as critical code sections, i.e. one client is permitted access to the box space at a time.

Example: consider the client function:

```
function myfcn():void;
b:bat[:oid,:int] := bbp.take("mytable");
c:bat[:int,:str] := sql.take("person","age");
d:= intersect(b,c);
io.print(d);
u:str:= client.take(user);
io.print(u);
client.release(user);
```

```
    end function;
```

The function binds to a copy from the local persistent BAT space, much like bat-names are resolved in earlier versions. The second statement uses an implementation of take that searches a variable of interest using two string properties. It illustrates that a box manager is free to extend/overload the predefined scheme, which is geared towards storing MAL variables.

The result bat 'c' is temporary and disappears upon garbage collection. The variable 'u' is looked up as the string object user.

Note that BATs b and c need be released at some point. In general this point in time does not coincide with a computational boundary like a function return. During a session, several bats may be taken out of the box, being processed, and only at the end of a session being released. In this example, it means that the reference to b and c is lost at the end of the function (due to garbarge collection) and that subsequent use requires another take() call. The box manager bbp is notified of the implicit release and can take garbage collection actions.

The box may be inspected at several times during a scenario run. The first time is when the MAL program is type-checked for the box operations. Typechecking a take() function is tricky. If the argument is a string literal, the box can be queried directly for the objects' type. If found, its type is matched against the lhs variable. This strategy fails in the situation when at runtime the object is subsequently replaced by another typed-instance in the box. We assume this not to happen and the exceptions it raises a valuable advice to reconsider the programming style.

The type indicator for the destination variable should be provided to proceed with proper type checking. It can resolve overloaded function selection.

Inspection of the Box can be encoded using an iterator at the MAL layer and relying on the functionality of the box. However, to improve introspection, we assume that all box implementations provide a few rudimentary functions, called objects(arglist) and dir(arglist). The function objects() produces a BAT with the object names, possibly limited to those identified by the arglist.

9.1 Future

The world of boxes has not been explored deeply yet. It is envisioned that it could play a role to import/export different objects, e.g. introduce xml.take() which converts an XML document to a BAT, jpeg.take() similer for an image.

Nesting boxes (like Russian dolls) is possible. It provides a simple containment scheme between boxes, but in general will interfere with the semantics of each box.

Each box has [should] have an access control list, which names the users having permission to read/write its content. The first one to create the box becomes the owner. He may grant/revoke access to the box to users on a selective basis.

9.1.1 Session box

Aside from box associated with the modules, a session box is created dynamically on behalf of each client. Such boxes are considered private and require access by the user name (and password). At the end of a session they are closed, which means that they are saved in persistent store until the next session starts. For example:

```
    function m():void;
    box.open("client_name");
    box.deposit("client_name","pi",3.417:flt);
    f:flt := box.take("client_name","pi");
    io.print(t);
    box.close("client_name");
    end function;
```

In the namespace it is placed subordinate to any space introduced by the system administrator. It will contain global client data, e.g. user, language, database, port, and any other session parameter. The boxes are all collected in the context of the database directory, i.e. the directory <dbfarm>/box

9.1.2 Garbage collection

The key objects managed by MonetDB are the persistent BATs, which call for an efficient scheme to make them accessible for manipulation in the MAL procedures taking into account a possibly hostile parallel access.

Most kernel routines produce BATs as a result, which will be referenced from the runtime stack. They should be garbage collected as soon as deemed possible to free-up space. By default, temporary results are garbage collected before returning from a MAL function.

9.1.3 Polymorphic globals

The top level interaction keeps a 'box' with global variables, i.e. each MAL statement is interpreted in an already initialized stack frame. This causes the following problems: 1) how to get rid of global variables and 2) how to deal with variables that can take 'any' type. It is illustrated as follows:

```
    f:= mil.take("GT");
    io.print(f);
```

When executed in the context of a function, the answer will be simple [1]. The variable type need not be defined, because the print operation can handle any type using the concrete type associated with f on the stack.

However, when executed interactively, statement by statement, the answer printed will be [0]. The reason is that both statements are converted into a function and executed independently. Between function calls the variables are reset, i.e. temporary variables are removed and others are re-initialized. [check]

10 Property Management

Properties come in several classes, those linked with the symbol table and those linked with the runtime environment. The former are determined once upon parsing or catalogue lookup. The runtime properties have two major subclasses, i.e. reflective and prescriptive. The reflective properties merely provide a fast cache to information aggregated from the target. Prescriptive properties communicate desirable states, leaving it to other system components to reach this state at the cheapest cost possible. This multifacetted world makes it difficult to come up with a concise model for dealing with properties. The approach taken here is an experimental step into this direction.

This `mal_properties` module provides a generic scheme to administer property sets and a concise API to manage them. Its design is geared towards support of MAL optimizers, which typically make multiple passes over a program to derive an alternative, better version. Such code-transformations are aided by keeping track of derived information, e.g. the expected size of a temporary result or the alignment property between BATs.

Properties capture part of the state of the system in the form of an simple term expression `(name, operator, constant)`. The property model assumes a namespace built around Identifiers. The operator satisfy the syntax rules for MAL operators. Conditional operators are quite common, e.g. the triple (count, `<`, 1000) can be used to denote a small table.

10.1 Property Associations

The property bearing objects in the MAL setting are variables (symbol table entries) and MAL function blocks. The direct relationship between instructions and a target variable, make it possible to keep the instruction properties in the corresponding target variable.

Variables properties. The variables can be extended at any time with a property set. Properties have a scope identical to the scope of the corresponding variable. Ommision of the operator and value turns it into a boolean valued property, whose default value is `true`.

```
b{count=1000,sorted}:= mymodule.action("table");
name{aligngroup=312} := bbp.take("person_name");
age{aligngroup=312} := bbp.take("person_age");
```

The example illustrates a mechanism to maintain alignment information. Such a property is helpful for optimizers to pick an efficient algorithm.

MAL function signatures. A function signature contains a description of the objects it is willing to accept and an indication of the expected result. The arguments can be tagged with properties that 'should be obeyed, or implied' by the actual arguments. It extends the typing scheme used during compilation/optimization. Likewise, the return values can be tagged with properties that 'at least' exist upon function return.

```
function test(b:bat[:void,:int]{count<1000}):bat[:void,:int]{sorted}
   #code block
end test
```

These properties are informative to optimizers. They can be enforced at runtime using the operation `optimizer.enforceRules()` which injects calls into the program to check them. An assertion error is raised if the property does not hold. The code snippet

```
z:= user.test(b);
```

is translated into the following code block;

```
mal.assert(b,"count","<",1000);
z:= user.test(b);
mal.assert(z,"sorted");
```

MAL instructions. Properties associated with an instruction are aimed at the a tactical optimizer and execution engine. It says something about the context in which the instruction should take place, e.g. a timeout property.

The second layer involves tagging structures known to front-ends, such as SQL and XML. Given that they produce a MAL block, setting properties becomes an explicit action. This can be handled as a first step in the optimization phase, which 'executes' all property setting actions first. However, 'preferred' properties may have to be dealt with later. For example, an operation should produce a sorted BAT. This calls for a runtime test and possible coercion. Alternatively, the desired property leads to a different selection of an operator implementation.

How to propagate properties? Property inspection and manipulation is strongly linked with the operators of interest. Optimizers continuously inspect and update the properties, while kernel operators should not be bothered with their existence.

10.2 Module import

The import statement simple switches the parser to a new input file, which takes precedence. The context for which the file should be interpreted is determined by the module name supplied. Typically this involves a module, whose definitions are stored at a known location.

11 The MAL Interpreter

The MAL interpreter always works in the context of a single user session, which provides for storage access to global variables and modules.

Runtime storage for variables are allocated on the stack of the interpreter thread. The physical stack is often limited in size, which calls for safeguarding their value and garbage collection before returning. A malicious procedure or implementation will lead to memory leakage.

A system command (linked C-routine) may be interested in extending the stack. This is precluded, because it could interfere with the recursive calling sequence of procedures. To accommodate the (rare) case, the routine should issue an exception to be handled by the interpreter before retrying. All other errors are turned into an exception, followed by continuing at the exception handling block of the MAL procedure.

11.1 MAL API

The linkage between MAL interpreter and compiled C-routines is kept as simple as possible. Basically we distinguish four kinds of calling conventions: CMDcall, FCNcall, THRDcall, and PATcall. The FCNcall indicates calling a MAL procedure, which leads to a recursive call to the interpreter.

CMDcall initiates calling a linked function, passing pointers to the parameters and result variable, i.e. f(ptr a0,..., ptr aN) The function returns a MAL-SUCCEED upon success and a pointer to an exception string upon failure. Failure leads to raise-ing an exception in the interpreter loop, by either looking up the relevant exception message in the module administration or construction of a standard string. Upon successfull return, we update the ValRecord with length indications.

The PATcall initiates a call which contains the MAL context. i.e. f(MalBlkPtr mb, MalStkPtr stk, InstrPtr pci) The blk provides access to the code definitions. It is primarilly used by routines intended to manipulate the code base itself, such as the optimizers. The Mal stack frame pointer provides access to the values maintained. The arguments passed are offsets into the stack frame rather than pointers to the actual value.

BAT parameters require some care. Ideally, a BAT should not be kept around long. This would mean that each time we access a BAT it has to be pinned in memory and upon leaving the function, it is unpinned. This degrades performance significantly. After the parameters are fixed, we can safely free the destination variable and re-initialize it to nil;

11.2 Exception handling

Calling a built-in or user-defined routine may lead to an error, a cached status message to be dealt with in MAL, or as an error status in Mapi.

Exceptions raised within a linked-in function requires some care. First, the called procedure does not know anything about the MAL interpreter context. Thus, we need to return all relevant information upon leaving the linked library routine.

Second, exceptional cases can be handled deeply in the recursion, where they may also be handled, i.e. by issueing an GDKerror message. The upper layers merely receive a negative integer value to indicate occurrence of an error somewhere in the calling sequence. We have to also look into GDKerrbuf to see if there was an error raised deeply inside the system.

The policy is to require all C-functions to return a string-pointer. Upon successfull call, this string function is NULL. Otherwise it contains an encoding of the exceptional state encountered. This message starts with the exception identifer, followed by contextual details.

11.3 BAT reference counting

A key issue is to deal with temporary BATs in an efficient way. References to bats in the buffer pool may cause dangling references at the language level. This appears as soons as your share a reference and delete the BAT from one angle. If not carefull, the dangling pointer may subsequently be associated with another BAT

Dangling references can be dealt with in several ways. First, we could increase the reference count each time a BAT becomes shared and rely on the garbage collector to properly decrement the count before abanding a execution frame. Second, we could try to avoid dangling pointers by permitting just one reference per BAT. This doesn't work, because we may keep BAT-views around, which depend on the existence of the underlying BAT.

11.4 Garbage collection

Garbage collection is relatively straightforward, because most values are retained on the stackframe of an interpreter call. However, two storage types and possibly user-defined type garbage collector definitions require attention. [TAKE CARE OF fixable atoms !!]

All string values are private to the VALrecord, which means they have to be freed explicitly before a MAL function returns. The first step is to always safe the destination variable before a function call is made.

All operations are responsible to properly set the reference count of the BATs being produced or destroyed. The exception are the target BAT variables. Their reference is saved until the operation is finished. They the reference can be reduced (due to overwrite of the target variable) or retained(the operation failed).

Actually, the libraries should not leave the physical reference count being set. This is only allowed during a GDK operation. All references should be logical.

11.5 Performance section

The interpreter has a built-in performance monitor hook, which is activated using the compile option MALprofiler. Activation can lead to a significant performance degradation, because for all traced functions we have to keep track of essential system counter information.

11.6 Bootstrap and module load

The server is bootstrapped by processing a MAL script with module definitions. For each module encountered, the object library lib_<modulename>.so is searched for in . . . /lib/MonetDB. The corresponding signature are defined in . . . /lib/<modulename>.mal.

The default bootstrap script is called . . . /lib/MonetDB/mal_init.mal and it is designated in the configuration file as the mal_init property. The rationale for this set-up is that database administrators can extend/overload the bootstrap procedure without affecting the distributed software package. It merely requires a different direction for the mal_init property.

The scheme also isolates the functionality embedded in modules from inadvertise use on non-compliant databases. [access control issue, how to limit what a user can do on a database?]

Unlike previous versions of MonetDB, modules can not be unloaded. Dynamic libraries are always global and, therefore, it is best to load them as part of the server initialization phase.

The MAL program should be compiled with -rdynamic and -ldl. This enables loading the routines and finding out the address of a particular routine

The mapping from MAL module.function() identifier to an address is resolved in the function getAddress. Since all modules libraries are loaded completely with GLOBAL visibility, it suffices to provide the internal function name. In case an attempt to link to an address fails, a final attempt is made to locate the *.o file in the current directory.

11.7 Module loading

The default location to search for the module is in monet_mod_path unless an absolute path is given.

Loading further relies on the Linux policy to search for the module location in the following order: 1) the colon-separated list of directories in the user's LD_LIBRARY, 2) the libraries specified in /etc/ld.so.cache, and 3) /usr/lib followed by /lib If the module contains a routine _init, then that code is executed before the loader returns. Likewise the routine _fini is called just before the module is unloaded.

A module loading conflict emerges if a function is redefined. A duplicate load is simply ignored by keeping track of modules already loaded.

To speedup restart and to simplify debugging, the Monet server can be statically linked with some (or all) of the modules. A complicating factor is then to avoid users to initiate another load of the module file, because it would lead to a dlopen error.

The partial way out of this dilema is to administer somewhere the statically bound modules, or to enforce that each module comes with a known routine for which we can search. In the current version we use the former approach.

11.8 MAL runtime stack

The runtime context of a MAL procedure is allocated on the runtime stack of the corresponding interpreter. Access to the elements in the stack are through index offsets, determined during MAL procedure parsing. This method has been proven highy efficient compared to using individual GDKmalloci calls.

Unlike Monet Version 4, the scope administration for MAL procedures is decoupled from their actual runtime behavior. This means we are more relaxed on space allocation, because the size is determined by the number of MAL procedure definitions instead of the runtime calling behavior. (See mal_interpreter for details on value stack management)

The variable names and types are kept in the stack to ease debugging. The underlying string value need not be garbage collected.

The interpreter should be protected against physical stack overflow. The solution chosen is to maintain an incremental depth size. Once it exceeds a threshold, we call upon the kernel to ensure we are still within safe bounds.

12 The Optimizer Landscape

One of the prime reasons to design the MAL intermediate language is to have a high-level description for database queries, which is easy to generate by a front-end compiler and easy to decode, optimize and interpret.

An optimizer needs several mechanisms to be effective. It should be able to perform a symbolic evaluation of a code fragment and collect the result in properties for further decision making. The prototypical case is where an optimizer estimates the result size of a selection.

Another major issue is to be able to generate and explore a space of alternative evaluation plans. This exploration may take place up front, but can also be ran at runtime for query fragments.

12.1 Optimizers

A query optimizer is often a large and complex piece of code, which enumerates alternative evaluation plans from which 'the best' plan is selected for evaluation. Limited progress has been made sofar to decompose the optimizer into (orthogonal) components, because it is a common believe in research that a holistic view on the problem is a prerequisite to find the best plan. Conversely, commercial optimizers use a cost-model driven approach, which explores part of the space using a limited (up to 300) rewriting rules.

Our hypothesis is that query optimization should be realized with a collection of query optimizer transformers (QOT), each dedicated to a specific task. Furthermore, they are assembled in scenarios to support specific application domains or achieve a desired behavior. Such scenarios are selected on a session basis, a query basis, or dynamically at runtime; they are part of the query plan.

The query transformer list below is under consideration for development. For each we consider its goal, approach, and expected impact. Moreover, the minimal prerequisites identify the essential optimizers that should have done their work already. For example, it doesn;t make sense to perform a static evaluation unless you have already propagated the constants using Alias Removal.

Scalar expressions (SXoptimizer) Goal: to remove scalar expressions which need be evaluated once during the query lifetime. Rationale: static expressions appear when variables used denote literal constants (e.g. 1+1), when catalog information can be merged with the plan (e.g. max(B.salary)), when session variables are used which are initialized once (e.g. user()). Early evaluation aids subsequent optimization. Approach: inspect all instructions to locate static expressions. Whether they should be removed depends on the expected reuse, which in most cases call for an explicit request upon query registration to do so. The result of a static evaluation provides a ground for AR. Impact: relevant for stored queries (MAL functions) Prereq: AR, CX

Relational Expression Optimizer (RXoptimizer) Goal: to evaluate a relational plan using properties of BATs, such as being empty or forming an aligned group. These optimizations assume that the code generator can detect properties while compiling e.g. an SQL query. Impact: high Prereq:

Alias Removal (ARoptimizer) Goal: to reduce the number of variables referenceing the same value, thereby reducing the analysis complexity. Rationale: query transformations often result in replacing the right-hand side expression with a result variable. This pollutes the code block with simple assignments e.g. V:=T. Within the descendant flow the occurrence of V could be replaced by T, provided V is never assigned a new value. Approach: literal constants within a MAL block are already recognized and replaced by a single variable. Impact: medium

Common Expression Optimizer (CXoptimizer) Goal: to reduce the amount of work by avoiding calculation of the same operation twice. Rationale: to simplify code generation for front-ends, they do not have to remember the subexpressions already evaluated. It is much easier to detect at the MAL level. Approach: simply walk through the instruction sequence and locate identical patterns. (Enhance is with semantic equivalent instructions) Impact: High Prereq: AR

Access Mode Optimizer (AMoptimizer) Goal: the default access mode for BATs is read-only. When updates are needed, it is switched to WriteMode, which needs to be done only once. Approach: remove duplicate calls Impact: low

Dead Code Removal (DCoptimizer) Goal: to remove all instructions whose result is not used Rationale: due to sloppy coding or alternative execution paths dead code may appear. Als XML Pathfinder is expected to produce a large number of simple assignments. Approach: Every instruction should produce a value used somewhere else. Impact: low

Heuristic Rule Rewrites (HRoptimizer) Goal: to reduce the volume as quick as possible. Rationale: most queries are focussed on a small part of the database. To avoid carrying too many intermediates, the selection should be performed as early as possible in the process. This assumes that selectivity factors are known upfront, which in turn depends on histogram of the value distribution. Approach: locate selections and push them back/forth through the flow graph. Impact: high

Join Path Optimizer (JPoptimizer) Goal: to reduce the volume produced by a join sequence Rationale: join paths are potentially expensive operations. Ideally the join path is evaluated starting at the smallest component, so as to reduce the size of the intermediate results. Approach: to successfully reduce the volume we need to estimate their processing cost. This calls for statistics over the value distribution, in particular, correlation histograms. If statistics are not available upfront, we have to restore to an incremental algorithm, which decides on the steps using the size of the relations. Impact: high

Operator Sort (OSoptimizer) Goal: to sort the dataflow graph in such a way as to reduce the cost, or to assure locality of access for operands. Rationale: A simple optimizer is to order the instructions for execution by permutation of the query components Approach: Impact:

Singleton Set (SSoptimizer) Goal: to replace sets that are known to produce precisely one tuple. Rationale: Singleton sets can be represented by value pairs in the MAL program, which reduces to a scalar expression. Approach: Identify a set variable for replacement. Impact:

Result Cacher (RCoptimizer) Goal: to reduce the processing cost by keeping track of expensive to compute intermediate results Rationale: Approach: result caching becomes active after an instruction has been evaluated. The result can be cached as long as its underlying operands remain unchanged. Result caching can be made transparent to the user, but affects the other QOTs Impact: high

Vector Execution (VEoptimizer) Goal: to rewrite a query to use a cache-optimal vector implementation Rationale: processing in the cache is by far the best you can get. However, the operands may far exceed the cache size and should be broken into pieces followed by a staged execution of the fragments involved. Approach: replace the query plan with fragment streamers Impact:

Staged Execution (SEoptimizer) Goal: to split a query plan into a number of steps, such that the first response set is delivered as quickly as possible. The remainder is only produced upon request. Rationale: interactive queries call for quick response and an indication of the processing time involved to run it too completion. Approach: staged execution can be realized using a fragmentation scheme over the database, e.g. each table is replaced by a union of fragments. This fragmentation could be determined upfront by the user or is derived from the query and database statistics. impact: high

Code Parallizer (CPoptimizer) Goal: to exploit parallel IO and cpu processing in both SMP and MPP settings. Rationale: throwing more resources to solve a complex query helps, provided it is easy to determine that parallel processing recovers the administrative overhead Approach: every flow path segment can be handled by an independent process thread. Impact: high

Query Evaluation Maps (QEMoptimizer) Goal: to avoid touching any tuple that is not relevant for answering a query. Rationale: the majority of work in solving a query is to disgard tuples of no interest and to find correlated tuples through join conditions. Ideally, the database learns these properties over time and re-organizes (or builts a map) to replace disgarding by map lookup. Approach: piggyback selection and joins as database fragmentation instructions Impact: high

MAL Compiler (MCcompiler) (tactics) Goal: to avoid interpretation of functional expressions Rationale: interpretation of arithmetic expressions with an interpreter is always expensive. Replacing a complex arithmetic expressin with a simple dynamically compiled C-functions often pays off. Especially for cached (MAL) queries Approach: Impact: high

Dynamic Query Scheduler (DQscheduler) (tactics) Goal: to organize the work in a way so as to optimize resource usage Rationale: straight interpretation of a query plan may not lead to the best use of the underlying resources. For example, the content of the runtime cache may provide an opportunity to safe time by accessing a cached source Approach: query scheduling is the last step before a relation algebra interpreter takes over control. The scheduling step involves a re-ordering of the instructions within the boundaries imposed by the flow graph. impact: medium

Aggregate Groups (AGoptimizer) Goal: to reduce the cost of computing aggregate expressions over times Rationale: many of our applications call for calculation of aggregates over dynamically defined groupings. They call for lengtly scans and it pays to piggyback all aggregate calculates, leaving their result in the cache for later consumption (eg the optimizers) Approach: Impact: High

Data Cube optimizer (DCoptimizer) Goal: to recognize data cube operations Rationale: Approach: Impact:

Demand Driven Interpreter (DDoptimizer) (tactics) Goal: to use the best interpreter and libraries geared at the task at hand Rationale: Interpretation of a query plan can be based on different computational models. A demand driven interpretation starts at the intended output and 'walks' backward through the flow graph to collect the pieces, possibly in a pipelined fashion. (Vulcano model) Approach: merely calls for a different implementation of the core operators Impact: high

Iterator Strength Reduction (SRoptimizer) Goal: to reduce the cost of iterator execution by moving instructions out of the loop. Rationale: although iteration at the MAL level should be avoided due to the inherent low performance compared to built-in operators, it is not forbidden. In that case we should confine the iterator block to the minimal work needed. Approach: inspect the flowgraph for each iterator and move instructions around. Impact: low

Accumulator Evaliations (AEoptimizer) Goal: to replace operators with cheaper ones. Rationale: based on the actual state of the computation and the richness of the supporting libraries there may exists alternative routes to solve a query. Approach: Operator rewriting depends on properties. No general technique. The first implementation looks at calculator expressions such as they appear frequently in the RAM compiler. Impact: high Prerequisite: should be called after CXoptimizer to avoid clashes.

Code Inliner (CIoptimizer) Goal: to reduce the calling depth of the interpreter and to obtain a better starting point for code squeezing Rationale: substitution of code blocks (or macro expansion) leads to longer linear code sequences. This provides opportunities for squeezing. Moreover, at runtime building and managing a stackframe is rather expensive. This should be avoided for functions called repeatedly. Approach: called explicity to inline a module (or symbol) Impact: medium

Code Outliner (COoptimizer) Goal: to reduce the program size by replacing a group with a single instruction Rationale: inverse macro expansion leads to shorter linear code sequences. This provides opportunities for less interpreter overhead, and to optimize complex, but repetative instruction sequences with a single hardwired call Approach: called explicitly to outline a module (or symbol) Impact: medium

Garbage Collector (GCoptimizer) Goal: to release resources as quickly as possible Rationale: BATs referenced from a MAL program keep resources locked. Approach: In cooperation with a resource scheduler we should identify those that can be released quickly. It requires a forced gargabe collection call at the end of the BAT's lifespan. Impact: large

Foreign Key replacements (FKoptimizer) Goal: to improve multi-attribute joins over foreign key constraints Rationale: the code produced by the SQL frontend involves foreign key constraints, which provides many opportunities for speedy code. Impact: large

12.2 Optimizer Dependencies

The optimizers are highly targeted to a particular problem. Aside from the resources available to invest in plan optimization, optimizers are partly dependent and may interfere.

To aid selection of the components of interest, we have grouped them in a preferred order of deployment.

Group A: Code Inliner (CIoptimizer)
Static expression evaluator. (SXoptimizer)
Relational Expression Evaluator. (RXoptimizer)
Strength Reduction (SRoptimizer)

Group B: Common Expression Optimizer (CXoptimizer)

	Query Evaluation Maps (QMoptimizer)
Group C:	Join Path Optimizer (JPoptimizer)
	Operator Cost Reduction (OCoptimizer)
	Operator Sort (OSoptimizer)
	Foreign Key handling (FKoptimizer)
	Aggregate Groups (AGoptimizer)
	Data Cube optimizer (DCoptimizer)
	Heuristic Rule Rewrite (HRoptimizer)
group D:	Code Parallizer (CPoptimizer)
	Accumulator Evaliations (AEoptimizer)
	Result Cacher (RCoptimizer)
	Replication Manager (RManager)
group E:	MAL Compiler (MCcompiler)
	Dynamic Query Scheduler (DQscheduler)
	Vector Execution (VEoptimizer)
	Staged Execution (SEoptimizer)
group F:	Alias Removal (ARoptimizer)
	Access Mode Optimizer (AMoptimizer)
	Dead Code Removal (DCoptimizer)
	Garbage Collector (GCoptimizer)

Alias removal can be applied after each other optimization step.

12.3 Optimizer building blocks

Some instructions are independent of the execution context. In particular, expressions over side-effect free functions with constant parameters could be evaluated before the program block is considered further.

A major task for an optimizer is to select instruction (sequences) which can and should be replaced with cheaper ones. The cost model underlying this decision depends on the processing stage and the overall objective. For example, based on a symbolic analysis their may exist better implementations within the interpreter to perform the job (e.g. hashjoin vs mergejoin). Alternative, expensive intermediates may be cached for later use.

Plan enumeration is often implemented as a Memo structure, which designates alternative sub-plans based on a cost metric. Perhaps we can combine these memo structures into a large table for all possible combinations encountered for a user.

The MAL language does not imply a specific optimizer to be used. Its programs are merely a sequence of specifications, which is interpreted by an engine specific to a given task. Activation of the engine is controlled by a scenario, which currently includes two hooks for optimization; a strategic optimizer and a tactical optimizer. Both engines take a MAL program and produce a (new/modified) MAL program for execution by the lower layers.

MAL programs end-up in the symbol table linked to a user session. An optimizer has the freedom to change the code, provided it is known that the plan derived is invariant to changes in the environment. All others lead to alternative plans, which should be collected as a trail of MAL program blocks. These trails can be inspected for a posteriori analysis, at least in terms of some statistics on the properties of the MAL program structures automatically. Alternatively, the trail may be pruned and re-optimized when appropriate from changes in the environment.

Breaking up the optimizer into different components and grouping them together in arbitrary sequences calls for careful programming.

The rule applied for all optimizers is to not-return before checking the state of the MAL program, and to assure the dataflow and variable scopes are properly set. It costs some performance, but the difficulties that arise from optimizer interference are very hard to debug. One of the easiest pitfalls is to derive an optimized version of a MAL function while it is already referenced by or when polymorphic typechecking is required afterwards. For example,

12.4 Optimizer framework

The large number of query transformers calls for a flexible scheme for the deploy them. The approach taken is to make all optimizers visible at the language level as a MAL pattern. Then (semantic) optimizer merely inspects a MAL block for their occurrences and activitates it.

Furthermore, the default optimizer scheme can be associated with a client record. The strategic optimizer merely prepends each query with this scheme before it searches/activates the optimizer routines.

The optimizer routines have access to the client context, the MAL block, and the program counter where optimizer call was found. Each query transformer should remove itself from the MAL block;

The optimizer terminates when no optimizer transformer call remains. [Some of the optimizers above should be moved to the tactic level]

Note, all optimizer instructions are executed only once. This means that the instruction can be removed from further consideration. However, in the case that a designated function is selected for optimization (e.g. CXoptimizer(user,qry)) the pc is assumed 0. The first instruction always denotes the signature and can not be removed.

To safeguard against incomplete optimizer implementations it is advisable to perform an optimizerCheck at the end. It takes as arguments the number of optimizer actions taken and the total cpu time spent. The body performs a full flow and type check and re-initializes the lifespan administration. In debugging mode also a copy of the new block is retained for inspection.

For each function it should be relatively easy to determine its safety property. This calls for accessing the function MAL block and to inspect the arguments of the signature.

Any instruction may block identification of a common subexpression. It suffices to stumble upon an unsafe function whose parameter lists has a non-empty intersection with the targeted instruction. To illustrate, consider the sequence

```
L1 := f(A,B,C);
...
G1 := g(D,E,F);
...
l2:= f(A,B,C);
...
L2:= h()
```

The instruction G1:=g(D,E,F) is blocking if G1 is an alias for {A,B,C}. Alternatively, function g() may be unsafe and {D,E,F} has a non-empty intersection with {A,B,C}. An alias can only be used later on for readonly (and not be used for a function with sideeffects)

12.4.1 Flow analysis

In many optimization rules, the data flow dependency between statements is of crucial importance. The MAL language encodes a multi-source, multi-sink dataflow network. Optimizers typically extract part of the workflow and use the language properties to enumerate semantic equivalent solutions, which under a given cost model turns out to result in better performance.

The flow graph plays a crucial role in many optimization steps. It is unclear as yet what primitives and what storage structure is most adequate. For the time being we introduce the operations needed and evaluate them directly against the program

The routine flowStep(pca,pcb) checks whether the output of instruction pca flows directly into pcb, without its parameters being changed inbetween. This calls for checking assignments as well as operators that use any of the targets of pca, but which also change them (e.g. insert/delete bat) TODO, now more restrictive then needed.

For each variable we should determine its scope of stability. End-points in the flow graph are illustrative as dead-code, that do not produce persistent data. It can be removed when you know there are no side-effect.

Side-effect free evaluation is a property that should be known upfront. For the time being, we assume it for all operations known to the system. The property ‘unsafe‘ is reserved to identify cases where this does not hold. Typically, a bun-insert operation is unsafe, as it changes one of the parameters.

12.4.2 Basic Algebraic Blocks

Many code snippets produced by e.g. the SQL compiler is just a linear representation of an algebra tree/graph. Its detection makes a number of optimization decisions more easy, because the operations are known to be side-effect free within the tree/graph. This can be used to re-order the plan without concern on impact of the outcome. It suffice to respect the flow graph. [unclear as what we need]

13 The Optimizer Toolkit

In this section we introduce the collection of MAL optimizers included in the code base. The tool kit is incrementally built, triggered by experimentation and curiousity. Several optimizers require further development to cope with the many features making up the MonetDB system. Such limitations on the implementation are indicated where appropriate.

13.1 Alias Removal

The routine `optimizer.aliasRemoval()` walks through the program looking for simple assignment statements, e.g. V:=W. It replaces all subsequent occurrences of V by W, provided V is assigned a value once and W does not change in the remainder of the code. Special care should be taken for iterator blocks as illustrated in the case below:

```
        i:=0;
        b:= "done";
barrier go:= true;
        c:=i+1;
        d:="step";
        v:=d;
        io.print(v);
        i:=c;
redo go:= i<2;
exit go;
        io.print(b);
        optimizer.aliasRemoval();
```

The constant strings are propagated to the `print()` routine, while the initial assigment `i:=0` should be retained. The code block becomes:

```
        i:=0;
barrier go:= true;
        c:=i+1;
        io.print("step");
        i:=c;
redo go:= i<2;
exit go;
        io.print("done");
```

13.2 Dead Code Removal

Dead code fragments are recognized by assignments to variables whose value is not consumed any more. It can be detected by marking all variables used as arguments as being relevant. In parallel, we built a list of instructions that should appear in the final result. The new code block is than built in one scan, discarding the superflous instructions.

Instructions that produce side effects to the environment, e.g. printing and BAT updates, should be taken into account. Such (possibly recursive) functions should be marked with a property (`unsafe`) For now we recognize a few important ones Likewise instructions marked as control flow instructions should be retained.

An illustrative example is the following MAL snippet:

```
V7 := bat.new(:void,:int);
V10 := bat.new(:int,:void);
V16 := algebra.markH(V7);
V17 := algebra.join(V16,V7);
V19 := bat.new(:void,:int);
V22 := bat.new(:void,:int);
V23 := algebra.join(V16,V22);
io.print("done");
optimizer.deadCodeRemoval();
```

The dead code removal trims this program to the following short block:

```
io.print("done");
```

13.3 Accumulator Evaluations

Bulk arithmetic calculations are pretty expensive, because new BATs are created for each expression. This memory hunger can be reduced by detecting opportunities for accummulator processing, i.e. where a (temporary) variable is overwritten. For example, consider the program snippet

```
t3:= batcalc.*(64,t2);
t4:= batcalc,+(t1,t3);
optimizer.expressionAccumulation();
```

If variable t2 is not used any further and retains its type throughout the program block, we can re-use its storage space and propagate its alias through the remainder of the code.

```
batcalc.*(t2,64,t2);
t4:= batcalc.+(t2,t1,t2);
```

The implementation is straight forward. It only deals with the arithmetic operations available in batcalc right now. This set will be gradually be extended. The key decision is to determine whether we may overwrite any of the arguments. We assume this is permissible if the argument was produced by an operator from the algebra, algebra2, group, aggrX3 modules.

13.4 Heuristic rewrites rules

One of the oldest optimizer tricks in relational query processing is to apply heuristic rules to reduce the processing cost. For example, a selection predicate is pushed through another operator to reduce the number of tuples to consider. Heuristic rewrites are relatively easy to apply in a context where the expression is still close to a relational algebra tree. Therefore, many of the standard rewrite rules are already applied by the SQL front-end as part of its strategic optimization decisions.

Finding rewrite opportunities within a linear MAL program may be more difficult. For, the pattern should respect the flow of control that may already be introduced. The last resort for the optimizer builder is to write a C-function to look for a pattern of interest and transform it. The code base contains an example how to built such user specific optimizer routines. It translates the pattern:

```
y:= reverse(R);
z:= select(y,l,h);
```

into the statement:

```
z:= selectHead(x,R,l,h)
```

13.5 Common Subexpression Elimination

Common subexpression elimination merely involves a single scan through the program block to detect re-curring statements. The key problem to be addressed is to make sure that the parameters involved in the repeatative instruction are invariant.

The analysis of optimizer.commonExpressionRemoval() is rather crude. All functions with possible side-effects on their arguments should have been marked as 'unsafe'. Their use within a MAL block breaks the dataflow graph for all objects involved (BATs, everything kept in boxes).

The common subexpression optimizer locates backwards the identical instructions. It stops as soon as it has found an identical one. Before we can replace the expression with the variable(s) of the previous one, we should assure that we haven;t passed a safety barrier.

```
b:= bat.new(:int,:int);
c:= bat.new(:int,:int);
d:= algebra.select(b,0,100);
e:= algebra.select(b,0,100);
k1:= 24;
k2:= 27;
l:= k1+k2;
l2:= k1+k2;
l3:= l2+k1;
optimizer.commonExpressionRemoval();
```

is translated into the code block where the first two instructions are not common, because they have side effects.

```
b := bat.new(:int,:int);
c := bat.new(:int,:int);
d := algebra.select(b,0,100);
e := d;
k1 := 24;
k2 := 27;
l := calc.+(k1,k2);
l3 := calc.+(l,k1);
```

The current implementation is rather expensive nested-loop algorithm, which does not perform well for large MAL blocks. Furthermore, after each hit we apply alias removal. This potentially generates more common expressions. The search can be improved significantly when a dataflow graph is maintained for all variables. For, equality of instruction implies that all variables have been used before in a similar context.

13.6 Emptyset Reduction

One of the key decisions during MAL optimization is to estimate the size of the BATs produced and consumed. Two cases are of interest for symbolic processing. Namely when a BAT is known to contain no tuples and those that have precisely one element. Such information may come from application domain knowledge or as a side effect from symbolic evaluation. It is associated with the program under inspection as properties.

The empty set property is used by the reduction algorithm presented here. Any empty set is propagated through the program to arrive at a smaller and therefore faster evaluation.

For example, consider the following MAL test

```
V1 := bat.new(:void,:int);
V7 := bat.new(:void,:int);
V10{empty} := bat.new(:int,:void);
V11 := bat.reverse(V10);
V12 := algebra.kdifference(V7,V11);
V16 := algebra.markH(V12);
V17 := algebra.join(V16,V7);
bat.append(V1,V17);
    optimizer.emptySet();
```

Calling the optimizer optimize.emptySet("user","test") replaces this program by the following code snippet.

```
 V1 := bat.new(:void,:int);
 V7 := bat.new(:void,:int);
 V16 := algebra.markH(V7);
 V17 := algebra.join(V16,V7);
 bat.append(V1,V17);
```

During empty set propagation, new candidates may appear. For example, taking the intersection with an empty set creates a target variable that is empty too. It becomes an immediate target for optimization. The current implementation is conservative. A limited set of instructions is considered. Any addition to the MonetDB instruction set would call for assessment on their effect.

13.7 Singleton Set Reduction

Application semantics and precise cost analysis may identify the result of an operation to produce a BAT with a single element. Such variables can be tagged with the property `singleton`, whereafter the operation `optimizer.singleton()` derives an MAL program using a symbolic evaluation as far as possible.

During its evaluation, more singleton sets can be created, leading to a ripple effect through the code. A non-optimizable instruction leads to a construction of a new table with the single instance.

```
b:= bat.new(:int,:int);
bat.insert(b,1,2);
c{singleton}:= algebra.select(b,0,4);
d:= algebra.markH(c);
io.print(d);
```

```
optimizer.singleton();
```

is translated by into the code block

```
b := bat.new(:int,:int);
bat.insert(b,1,2);
c{singleton} := algebra.select(b,0,4);
($15,$16):= bat.unpack(c{singleton});
d := bat.pack(nil,$16);
io.print(d);
```

13.8 Peephole optimization

Recursive descend query compilers easily miss opportunities for better code generation, because limited context is retained or lookahead available. The peephole optimizer is built around such recurring patterns and compensates for the compilers 'mistakes'. The collection of peephole patterns should grow over time and front-end specific variations are foreseen.

The SQL frontend heavily relies on a pivot table, which is a generated oid sequence. Unfortunately, this is not seen and the pattern '$i := calc.oid(0@0); $j:= algebra.markT($k,$i);' occurs often. This can be replaced with '$j:= algebra.markT($k)';

Another example of a 2-way instruction sequence produced is then '$j:= algebra.markT($k); $l:= bat.reverse($j);', which can be replaced by '$l:= algebra.markH($k);'.

The reverse-reverse operation also falls into this category. Reversal pairs may result from the processing scheme of a front-end compiler or from a side-effect from other optimization steps. Such reversal pairs should be removed as quickly as possible, so as to reduce the complexity of finding alternative optimization opportunities. As in all cases we should ensure that the intermediates dropped are not used for other purposes as well.

```
r:bat[:int,:int]:= bat.new(:int,:int);
o:= calc.oid(0@0);
z:= algebra.markT(r,o);
rr:= bat.reverse(z);
s := bat.reverse(r);
t := bat.reverse(s);
io.print(t);
optimizer.peephole();
```

which is translated by the peephole optimizer into:

```
r:bat[:int,:int] := bat.new(:int,:int);
rr := algebra.markH(r);
io.print(r);
```

13.9 Multiplex Compilation

The MonetDB operator multiplex concept has been pivotal to easily apply any scalar function to elements in a containers. Any operator `cmd` came with its multiplex variant [`cmd`]. Given the signature `cmd(T1,..,Tn) : Tr`, it could be applied also as `[CMD](bat[:any_1,:T1],...,bat[any_1,Tn]) :bat[any_1,Tr]`.

The semantics of the multiplex is to perform the natural join on all bat-valued parameters, and to execute the CMD for each combination of matching tuples. All results are collected in a result BAT. All but one argument may be replaced by a scalar value.

The generic solution to the multiplex operators is to translate them to a MAL loop. A snippet of its behaviour:

```
    b:= bat.new(:int,:int);
    bat.insert(b,1,1);
    c:bat[:int,:int]:= optimizer.multiplex("calc.+",b,1);
```

The current implementation requires the target type to be mentioned explicitly. The result of the optimizer is:

```
    b := bat.new(:int,:int);
    bat.insert(b,1,1);
    _8 := bat.new(:int,:int);
barrier (_11,_12,_13):= chopper.newIterator(b);
    _15 := calc.+(_13,1);
    bat.insert(_8,_12,_15);
catch MALexception;
exit MALexception;
    redo (_11,_12,_13):= chopper.hasMoreElements(b);
exit (_11,_12,_13);
    c := _8;
```

[WARNING] the semantics does not align precisely with M4. For, in search for efficient operators we assume aligned BATs and the script version does not deal with duplicates in the head of the bats. [/WARNING]

13.10 Garbage Collection

Garbage collection of temporary variables, such as strings and BATs, takes place upon returning from a function call. Especially for BATs this may keep sizable resources locked longer than strictly necessary. Although the programmer can influence their lifespan by assignment of the `nil`, thereby triggering the garbage collector, it is more appropriate to rely on an optimizer to inject these statements. For, it keeps the program smaller and a better target for code-optimizations.

The operation `optimizer.garbageCollector()` removes all BATs that are at their end of life to make room for new ones. It is typically called as one of the last optimizer steps. A snippet of a the effect of the garbage collector:

```
    t1 := bat.new(:void,:int);
    t2 := array.grid(132000,8,1,0);
    t3 := array.grid(1,100,10560,0);
    t4 := array.grid(1,100,10560,0,8);
    t5 := batcalc.+(t2,t4);
    t6 := batcalc.oid(t5);
    t7 := algebra.join(t6,t1);
    optimizer.setGarbageCollector();
```

is translated into the following code block:

```
    t1 := bat.new(:void,:int);
    t2 := array.grid(132000,8,1,0);
    t3 := array.grid(1,100,10560,0);
    t4 := array.grid(1,100,10560,0,8);
    t5 := batcalc.+(t2,t4);
    bat.setGarbage(t2);
    bat.setGarbage(t4);
    t6 := batcalc.oid(t5);
    bat.setGarbage(t5);
    t7 := algebra.join(t6,t1);
    bat.setGarbage(t6);
    bat.setGarbage(t1);
```

The current algorithm is straight forward. After each instruction we check whether its BAT arguments are needed in the future. If not, we inject a garbage collection statement to release them, provided there are no other reasons to retain it. This should be done carefully, because the instruction may be part of a loop. If the variable is defined inside the loop, we can safely remove it.

13.11 Code Factorization

In most real-life situations queries are repeatedly called with only slight changes in their parameters. This situation can be captured by the query compilers by keeping a cache of recent query plans. In MonetDB context such queries are represented as parameterized MAL programs.

To further optimize the cached functions it might help to split the query plan into two sections. One section with those actions that do not depend on the arguments given and another section that contains the heart of the query using all information. Such a program can be represented by a MAL factory, which is a re-entrend query plan.

An example of how factorize changes the code is shown below:

```
function test(s:str):lng;
    b:= bat.new(:int,:str);
    bat.insert(b,1,"hello");
    z:= algebra.select(b,s,s);
    i:= aggr.count(z);
    return i;
end test;
optimizer.factorize("user","test");
```

which translates into the following block:

```
factory user.test(s:str):lng;
    b := bat.new(:int,:str);
    bat.insert(b,1,"hello");
barrier always := true;
    z := algebra.select(b,s,s);
    i := aggr.count(z);
    yield i;
    redo always;
```

```
exit always;
end test;
```

The factorizer included is a prototype implementation of MAL factorization. The approach taken is to split the program into two pieces and wrap it as a MAL factory. The optimization assumes that the database is not changed on tables accessed only once during the factory lifetime. Such changes should be detected from the outside and followed by re-starting the factory.

A refined scheme where the user can identify the 'frozen' parameters is left for the future. As the mapping of a query to any of the possible available factories to deal with the request. For the time being we simple reorganize the plan for all parameters

Caveats. The factorize operation interferes with `optimizer.expressionAccumulation()` because that may overwrite the arguments. For the time being this is captured in a local routine.

13.12 Multiple Association Tables

A Multi Association Table descriptor (MAT) defines an ordered collection of type compatible BATs, whose union represents a single (virtual) BAT. The MAT may relate to a partitioned BAT, but could also be an arbitrary collection of temporary BATs within a program fragment.

The MAT definition lives within the scope of a single block. The MAT optimizer simply expands the plan to deal with its components on an individual basis. Only when a blocking or aggregate operator is encounted, the underlying BAT is materialized.

The MAT object can not be passed as an argument to any function without first being materialized. Simply because the MAT is not known by the type system and none of the lower level operations is currently aware of their existence.

In the first approach of the MAT optimizer we assume that the last BAT in the MAT sequence is used as an accumulator. Furthermore, no semantic knowledge is used to reduce the possible superflous (semi)joins. Instead, we limit expansion to a single argument. The last one in the argument list. This is changed at a later stage when a cost-based evaluation be decide differently.

To illustrate, consider

```
      m0:= bat.new(:void,:int);
  m1:= bat.new(:void,:int);
  m2:= bat.new(:void,:int);
  b := mat.new(m0,m1,m2);
  s := algebra.select(b,1,3);
  i := algebra.count(s);
  io.print(s);
  io.print(i);
  c0 := bat.new(:int,:int);
  c1 := bat.new(:int,:int);
  c := mat.new(c0,c1);
  j := algebra.join(b,s);
  io.print(j);
```

The selection and aggregate operations can simply be rewritten into a MAT.

```
    _33 := algebra.select(m0,1,3);
    _34 := algebra.select(m1,1,3);
    _35 := algebra.select(m2,1,3);
        s := mat.new(_33,_34,_35);
        i := 0;
        _36 := aggr.count(_33);
        i := calc.+(i,_36);
        _37 := aggr.count(_34);
        i := calc.+(i,_37);
        _38 := aggr.count(_35);
        i := calc.+(i,_38);
        io.print(i);
```

The print operation does not have MAT semantics yet, which calls for collection of the MAT components in a single BAT first.

```
        s := mat.pack(_33,_34,_35);
        io.print(s);
```

For the join we have to generate all possible combinations, not knowing anything about the properties of the components. The current heuristic is to limit expansion to a single argument. This leads to

```
        b := mat.pack(m0,m1,m2);
        _39 := algebra.join(b,c0);
        _40 := algebra.join(b,c1);
        j := mat.new(_39,_40);
```

The drawback of this scheme is the explosion in MAL statements. It is choosen as a basis for experimentation. In a fullblown system we would also use iterators to avoid it.

13.13 Incremental query processing

Limitations on the addressing space in older PCs and the need for distributed storage makes that BATs ideally should be looked upon as a union of smaller BATs which are processed within the (memory) resource limitations given.

In the previous section, we already introduced the concept of a MAT, as the named union of a number of BATs to form a single virtual BAT. It can be used as a basis for incremental processing as well. For this to work, we assume that the MAT components each satisfy the (memory) resource contraints.

The operation `optimizer.partitionedQuery` takes these definitions and encapsulate them in iterator blocks, thereby minimizing the MAL program footprint.

The optimizer is designed incrementally. The focus is on supporting the SQL front-end. In particularly, the operators considered is a limited subset of MAL. Occurrence of an operator outside this set terminates the optimizer activities.

The general strategy is to select a portion of the workflow related to a MAT variable and encapsulate it with an iterator over the BAT partitions. In doing so, we replace operators with their incremental counter part, alike the MAT optimizer.

A snippet of the partitioning optimizer is shown below. It assumes that the MAT `a0` consists of the components `a1,a2,a3`. A subsequent call of the partition optimizer will replace the print command with an iterator.

```
function qry(a:bat[:void,:any_1]{mat});
        io.print(b);
end qry;
optimizer.partitionedQuery("user","qry");
```

The query is transformed into the following version.

```
function user.qry(a:bat[:void,:any_1]{mat}):void;
barrier (_5,_6):= mat.newIterator(a);
        io.print(_6);
        redo (_5,_6):= mat.hasMoreElements(a);
exit (_5,_6);
end qry;
```

The translation of a MAL block into a multi-iterator only works if the network flow is simple, i.e there exists a 'single' sink, or all the information is compressed into variables that are passed back to the caller as a single value, or is sent non-interupted to the output channels. The bottom-line is to avoid expansion. An assignment to the target is not allowed [to be checked separately]

The first implementation takes a query block and replaces the first bat variable with its partitioned equivalent. The external references should be adjusted to permit working your way through the complete graph incrementally. They denote an inclusive range. A copy of the program block is made by default. This could be optimized away later.

13.14 Strength Reduction

An effective optimization technique in compiler construction is to move invariant statements out of the loops. The equivalent strategy can be applied to the guarded blocks in MAL programs. Any variable introduced in a block and assigned a value using a side-effect free operation is a candidate to be moved. Furthermore, it may not be used outside the block and the expression may not depend on variables assigned a value within the same block.

```
        j:= "hello world";
barrier go:=true;
        i:= 23;
        j:= "not moved";
        k:= j;
        io.print(i);
        redo go:= false;
exit go;
        z:= j;
optimizer.strengthReduction();
```

which is translated into the following code:

```
        j := "hello world";
        i := 23;
barrier go := true;
        j := "not moved";
        k := j;
```

```
        io.print(i);
        redo go:= false;
exit go;
        z:= j;
```

Application is only applicable to loops and not to guarded blocks in general, because execution of a statement outside the guarded block consumes processing resources which may have been prohibited by the block condition.

For example, it doesn't make sense to move creation of objects outside the barrier.

13.15 The MAL costmodel

Cost models form the basis for many optimization decisions. The cost parameters are typically the size of the (intermediate) results and response time. Alternatively, they are running aggregates, e.g. max memory and total execution time, obtained from a simulated run. The current implementation contains a framework and example for building your own cost-based optimized.

The `optimizer.costEstimation()` works its way through a MAL program in search for relational operators and estimates their result size. The operations taken into account are The estimated size is left behind as property `RMcost`, to avoid overwriting information produced by the front-end compiler.

```
r{count=100} := bat.new(:void,:int);
s{count=1000}:= bat.new(:void,:int);
rs:= algebra.select(s,1,1);
rr:= bat.reverse(r);
j:= algebra.join(rs,rr);
optimizer.costEstimation();
```

changes the properties of the instructions as follows:

```
r{RMcost=100,count=100} := bat.new(:void,:int);
s{RMcost=1000,count=1000} := bat.new(:void,:int);
rs{RMcost=500} := algebra.select(s{RMcost=1000,count=1000},1,1);
rr{RMcost=100} := bat.reverse(r{RMcost=100,count=100});
j{RMcost=100} := algebra.join(rs{RMcost=500},rr{RMcost=100});
```

The cost estimation does not use any statistics on the actual data distribution yet. It just applies a few heuristic cost estimators.

13.16 Variable Stack Reduction

The compilers producing MAL may generate an abundance of temporary variables to hold the result of expressions. This leads to a polution of the runtime stack space, because space should be allocated and garbage collection tests should be performed.

The routine `optimizer.reduce()` reduces the number of scratch variables to a minimum. All scratch variables of the same underlying type share the storage space. The result of this optimizer can be seen using the MonetDB debugger, which marks unused variables explicitly. Experience with the SQL front-end shows, that this optimization step easily reduces the stack consumption by over 20\%.

This optimizer needs further testing. Furthermore, the other optimizers should be careful in setting the isused property, or this property can again be easily derived.

13.17 Query Execution Plans

A commonly used data structure to represent and manipulate a query is a tree (or graph). Its nodes represent the operators and the leaves the operands. Such a view comes in handy when you have to re-organize whole sections of code or to built-up an optimized plan bottom up, e.g. using a memo structure.

The MAL optimizer toolkit provides functions to overlay the body of any MAL block with a tree (graph) structure and to linearize them back into a MAL block. The linearization order is determined by a recursive descend tree walk from the anchor points in the source program.

To illustrate, consider the code block:

```
#T1:= bat.new(:int,:int);
#T2:= bat.new(:int,:int);
#T3:= bat.new(:int,:int);
#T4:= bat.new(:int,:int);
a:= algebra.select(T1,1,3);
b:= algebra.select(T2,1,3);
c:= algebra.select(T3,0,5);
d:= algebra.select(T4,0,5);
e:= algebra.join(a,c);
f:= algebra.join(b,d);
h:= algebra.join(e,f);
optimizer.dumpQEP();
```

which produces an indented structure of the query plan.

```
h := algebra.join(e,f);
    e := algebra.join(a,c);
        a := algebra.select(T1,1,3);
            T1 := bat.new(:int,:int);
        c := algebra.select(T3,0,5);
            T3 := bat.new(:int,:int);
    f := algebra.join(b,d);
        b := algebra.select(T2,1,3);
            T2 := bat.new(:int,:int);
        d := algebra.select(T4,0,5);
            T4 := bat.new(:int,:int);
```

Any valid MAL routine can be overlayed with a tree (graph) view based on the flow dependencies, but not all MAL programs can be derived from a simple tree. For example, the code snippet above when interpreted as a linear sequence can not be represented unless the execution order itself becomes an operator node itself.

However, since we haven't added or changed the original MAL program, the routine `qep.propagate` produces the orginial program, where the linear order has priority. If, however, we had entered new instructions into the tree, they would have been placed in close proximity of the other tree nodes.

Special care is given to the flow-of-control blocks, because to produce a query plan section that can not easily be moved around. [give dot examples]

14 Program Debugging

In practice it is hard to write a correct MAL program the first time around. Instead, it is more often constructed by trial-and-error. As long as there are syntax and semantic errors the MAL compiler provides a sufficient handle to proceed. Once it passes the compiler we have to resort to a debugger to assess its behavior.

14.1 The MAL debugger

To ease debugging and performance monitoring, the MAL interpreter comes with a gdb-like debugger. An illustrative session elicits the functionality offered.

```
>function test(i:int):str;
> io.print(i);
> i:= i*2;
> b:= bbp.new(:int,:int);
> bat.insert(b,1,i);
> io.print(b);
> return test:= "ok";
>end test;
>user.test(1);
[ 1 ]
#-----------------#
# h     t         # name
# int   int       # type
#-----------------#
[ 1,      2       ]
```

The debugger can be entered at any time using the call mdb.start(). An overview of the available commands is readily available.

```
>mdb.start();
#mdb !end main;
mdb>help
next          -- Advance to next statement
continue      -- Continue program being debugged
break         -- set breakpoint on current instruction
break <var>     -- break on assignment to <var>
delete <var>    -- remove break point <var>
step          -- advance to next MAL instruction
module        -- display the module signatures
finish        -- finish current call
exit          -- terminate current call
quit          -- turn off debugging
list <fcn> -- list current program block
List <fcn> -- list with type information
var  <fcn>  -- print symbol table for module
optimizer <idx>  -- display program after optimizer step
print <var>    -- display value of a variable
print <var> <cnt>[<first>] -- display BAT chunk
```

```
info <var>     -- display bat variable properties
run        -- restart current procedure
where      -- print stack trace
down       -- go down the stack
up         -- go up the stack
timer      -- produce micro-second response time
io         -- produce page activity trace
help       -- this message
mdb>
```

We walk our way through a debugging session, highlighting the effects of the debugger commands. The call to mdb.start() has been encapsulated in a complete MAL function, as shown by issuing the list command. A more detailed listing shows the binding to the C-routine and the result of type resolution.

```
>mdb.start();
#end main;
mdb>l
function user.main():int;
mdb.start();
end main;
mdb>L
function user.main():int;       # 0  (main:int)
mdb.start();        # 1 MDBstart (_1:void)
end main;       # 2
```

The user module is the default place for function defined at the console. The modules loaded can be shown typeing the command 'modules' or 'module'(or 'm' for short). The former merely enumerates the module names, the latter also shows the names of the function in each. The function signatures become visible using the module and optionally the function name.

```
mdb>m alarm
#command alarm.alarm(secs:int,action:str):void address ALARMsetalarm;
#command alarm.ctime():str address ALARMctime;
#command alarm.epilogue():void address ALARMepilogue;
#command alarm.epoch():int address ALARMepoch;
#command alarm.prelude():void address ALARMprelude;
#command alarm.sleep(secs:int):void address ALARMsleep;
#command alarm.time():int address ALARMtime;
#command alarm.timers():bat[:str,:str] address ALARMtimers;
#command alarm.usec():lng address ALARMusec;
mdb>m alarm.sleep
#command alarm.sleep(secs:int):void address ALARMsleep;
mdb>
```

The debugger mode is left with a <return>. Any subsequent MAL instruction re-activates the debugger to await for commands. The default operation is to step through the execution using the 'next' ('n') or 'step' ('s) commands, as shown below.

```
>user.test(1);
```

```
#    user.test(1);
mdb>n
#    io.print(i);
mdb>
[ 1 ]
#    i := calc.*(i,2);
mdb>
#    b := bbp.new(:int,:int);
mdb>
```

The last instruction show is ready to be executed, which means the calculation has already been finished. The value assigned to a variable can be shown using a print statement, which contains the location of the variable on the stack frame, its name, its value and type. The complete stack frame becomes visible with 'values' ('v') command:

```
#    bat.insert(b,1,i);
mdb>
#    io.print(b);
mdb>v
#Stack for 'test' size=32 top=11
#[0] test        = nil:str
#[1] i   = 4:int
#[2] _2  = 0:int   unused
#[3] _3  = 2:int  constant
#[4] b   = <tmp_1226>:bat[:int,:int]   count=1 lrefs=1 refs=0
#[5] _5  = 0:int   type variable
#[6] _6  = nil:bat[:int,:int]   unused
#[7] _7  = 1:int  constant
#[8] _8  = 0:int   unused
#[9] _9  = "ok":str  constant
```

The variables marked 'unused' have been introduced as temporary variables, but which are not referenced in the remainder of the program. It also illustrates basic BAT properties, a complete description of which can be obtained using the 'info' ('i') command. A sample of the BAT content can be printed passing tuple indices, e.g. 'print b 10 10' prints the second batch of ten tuples.

14.2 Handling Breakpoints

A powerful mechanism for debugging a program is to set breakpoints during the debugging session. The breakpoints are designated by a target variable name, a [module.]operator name, or a MAL line number (#<number>). As soon as the variable recieves a new value or the operation is called the debugger is activated.

The snippet below illustrates the reaction to set a break point on assignment to variable 'i'.

```
>mdb.start();
#end main;
mdb>
>user.test(1);
```

```
#    user.test(1);
mdb>break i
breakpoint on 'i' not set
mdb>n
#    io.print(i);
mdb>break i
mdb>c
[ 1 ]
#    i := calc.*(i,2);
mdb>
```

The breakpoints remain in effect over multiple function calls. They can be removed with the `delete` statement. A list of all remaining breakpoints is obtained with `breakpoints`.

The interpreter can be instructed to call the debugger as soon as an exception is raised. Simply add the instruction `mdb.setCatch(true)`.

14.3 Runtime Inspection and Reflection

Part of the debugger functionality can also be used directly with MAL instructions. The execution trace of a snippet of code can be visualized encapsulation with `mdb.setTrace(true)` and `mdb.setTrace(false)`. Likewise, the performance can be monitored with the command `mdb.setTimer(on/off)`. Using a boolean argument makes it easy to control the (performance) trace at run time. The following snippet shows the effect of patching the test case.

```
>function test(i:int):str;
> mdb.setTrace(true);
> io.print(i);
> i:= i*2;
> b:= bbp.new(:int,:int);
> bat.insert(b,1,i);
> io.print(b);
> mdb.setTrace(false);
> return test:= "ok";
>end test;
>user.test(1);
#    mdb.setTrace(_3=true)
[ 1 ]
#    io.print(i=1)
#    i := calc.*(i=2, _5=2)
#    b := bbp.new(_7=0, _8=0)
#    bat.insert(b=<tmp_1226>, _10=1, i=2)
#-----------------#
# h     t         # name
# int   int       # type
#-----------------#
[ 1,      2       ]
#    io.print(b=<tmp_1226>)
#   261 usec!    user.test(_2=1)
```

```
>
```

The command setTimer() produces timing statistics in micro-seconds. Aside from the time spent in each statement separately, it accumulate the timeing for the complete call. The time spent on preparing the trace information is excluded from the time reported. For more detailed timing information the Linux tool *valgrind* may be of help.

```
>function test(i:int):str;
> mdb.setTimer(true);
> io.print(i);
> i:= i*2;
> b:= bbp.new(:int,:int);
> bat.insert(b,1,i);
> io.print(b);
> mdb.setTimer(false);
> return test:= "ok";
>end test;
>user.test(1);
#     6 usec#    mdb.setTimer(_3=true)
[ 1 ]
#    43 usec#    io.print(i=1)
#     5 usec#    i := calc.*(i=2, _5=2)
#    24 usec#    b := bbp.new(_7=0, _8=0)
#    10 usec#    bat.insert(b=<tmp_1226>, _10=1, i=2)
#-----------------#
# h     t         # name
# int   int       # type
#-----------------#
[ 1,      2       ]
#   172 usec#    io.print(b=<tmp_1226>)
#   261 usec#    user.test(_2=1)
```

It is also possible to activate the debugger from within a program using `mdb.start()`. It remains in this mode until you either issue a quit command, or the command mdb.stop() instruction is encountered. The debugger is only activated when the user can direct its execution from the client interface. Otherwise, there is no proper input channel and the debugger will run in trace mode.

The program listing functionality of the debugger is also captured in the MAL debugger module. The current code block can be listed using `mdb.list()` and `mdb.List()`. An arbitrary code block can be shown with `mdb.list(`*module,function*`)` and `mdb.List(`*module,function*`)`. A BAT representation of the current function is return by `mdb.getDefinition()`.

The symbol table and stack content, if available, can be shown with the operations `mdb.var()` and `mdb.list(`*module,function*`)` Access to the stack frames may be helpful in the context of exception handling. The operation `mdb.getStackDepth()` gives the depth and individual elements can be accessed as BATs using `mdb.getStackFrame(`*n*`)`. The top stack frame is accessed using `mdb.getStackFrame()`.

15 Execution Profiling

A key issue in the road towards a high performance implementation is to understand where resources are being spent. This information can be obtained using different tools and at different levels of abstraction. A coarse grain insight for a particular application can be obtained using injection of the necessary performance capturing statements in the instruction sequence. Fine-grain, platform specific information can be obtained using existing profilers, like valgrind (http://www.valgrind.org), or hardware performance counters.

The MAL profiler collects detailed performance information, such as cpu, memory and statement information. It is optionally extended with IO activity, which is needed for coarse grain profiling only.

The execution profiler is supported by hooks in the MAL interpreter. The default strategy is to ship an event record immediately over a stream to a separate performance monitor, formatted as a tuple. An alternative strategy is preparation for off-line performance analysis. For this case the event record is turned into a XML structure and sent over the stream (most likely linked with a file) Reflective performance analysis is supported by an event cache, emptied explicitly upon need.

15.1 Event Filtering

The profiler supports selective retrieval of such information by tagging the instructions of interest. This means that a profiler call has a global effect, all concurrent users are affected by the performance overhead. Therefore, it is of primary interest to single user sessions.

The example below illustrates how the different performance counter groups are activated, instructions are filtered for tracking, and where the profile information is retained for a posteriori analysis. The tuple format is chosen.

```
profiler.activate("time");
profiler.activate("ticks");
#profiler.activate("cpu");
#profiler.activate("memory");
#profiler.activate("io");
#profiler.activate("pc");
#profiler.activate("event");
profiler.activate("statement");
profiler.setFilter("*","insert");
profiler.setFilter("*","print");

profiler.openStream("/tmp/MonetDBevents");
profiler.start();
b:= bbp.new(:int,:int);
bat.insert(b,1,15);
bat.insert(b,2,4);
bat.insert(b,3,9);
io.print(b);
profiler.stop();
profiler.closeStream();
```

The information on CPU, IO, memory use, program counter and the event identifier are not retained. Furthermore, we are interested in all functions name `insert` and `print`. A wildcard can be used to signify any name, e.g. no constraints are put on the module in which the operations are defined.

Execution of the sample leads to the creation of a file with the following content. The ticks are measured in micro-seconds.

```
# time, ticks,  stmt,   # name
[ "15:17:56",   12,      "_27 := bat.insert(<tmp_15>,1,15);",    ]
[ "15:17:56",   2,       "_30 := bat.insert(<tmp_15>,2,4);",     ]
[ "15:17:56",   2,       "_33 := bat.insert(<tmp_15>,3,9);",     ]
[ "15:17:56",   245,     "_36 := io.print(<tmp_15>);",   ]
```

The XML-formatted output is produced by encapsulation of a code block with the pair `profiler.setOfflineProfiling()`. The default is tuple format, which can be requested explicitly using `profiler.setOnlineProfiling()`.

15.2 Event Caching

Aside from shipping events to a separate process, the profiler can keep the events in a local `BAT` group, triggered with the command `profiler.setCachedProfiling()`. Ofcourse, every measurement scheme does not come for free and may even obscure performance measurements obtained throug e.g. valgrind. The separate event caches can be accessed using the operator `profiler.getTrace(`*name*`)`. The current implementation only supports access to `time`,`ticks`,`pc`,`statement`. The event cache can be cleared with `profiler.clearTrace()`.

Consider the following MAL program snippet:

```
    profiler.setAll();
    profiler.setCachedProfiling();
    profiler.startTrace();
    b:= bbp.new(:int,:int);
    bat.insert(b,1,15);
    io.print(b);
    profiler.stopTrace();
    s:= profiler.getTrace("statement");
    t:= profiler.getTrace("ticks");
    io.print(s,t);
```

The result of the program execution becomes:

```
#-----------------#
# h     t         # name
# int   int       # type
#-----------------#
[ 1,      15      ]
#----------------------------------------------------------#
# h     t                                        t         # name
# int   str                                      int       # type
#----------------------------------------------------------#
[ 1,      "b := bbp.new(0,0);",                    51      ]
[ 2,      "$6 := bat.insert(<tmp_22>,1,15);",      16      ]
```

```
[ 3,      "$9 := io.print(<tmp_22>);",             189     ]
```

Some additional stuff

Appendix A SQL 1999 Features

The table below illustrates the features supported (S) and not supported (N) in the MonetDB/SQL distribution. Beware, some of the features are technically impossible to support without major code changes or excessive performance consequences.

Feature ID	Feature name	S/N
B011-B017	Embedded Language support. Core SQL:1999 says that at least one of Embedded Ada, Embedded C, Embedded Cobol, Embedded Fortran, Embedded MUMPS, Embedded Pascal or Embedded PL/I 1 should be supported.	N
E011	Numeric data types (FULL support)	S
E011-01	INTEGER and SMELLIEST data types (including all spellings)	S
E011-02	REAL, DOUBLE PRECISON, and FLOAT data types	S
E011-03	DECIMAL and NUMERIC data types	S
E011-04	Arithmetic operators	S
E011-05	Numeric comparison	S
E011-06	Implicit casting among the numeric data types	S
E021	Character data types	S(PARTIAL support)
E021-01	CHARACTER data type (including all its spellings)	S
E021-02	CHARACTER VARYING data type (including all its spellings)	S
E021-03	Character literals	
E021-04	CHARACTER_LENGTH function	S
E021-05	OCTET_LENGTH function	N (TODO!!!)
E021-06	SUBSTRING function	S
E021-07	Character concatenation	S
E021-08	UPPER and LOWER functions	S
E021-09	TRIM function	S
E021-10	Implicit casting among the character data types	S
E021-11	POSITION function	S
E021-12	Character comparison	S
E031	Identifiers (FULL support)	S
E031-01	Delimited identifiers	S
E031-02	Lower case identifiers	S
E031-03	Trailing underscore	S
E051	Basic query specification (FULL support)	S
E051-01	SELECT DISTINCT	S
E051-02	GROUP BY clause	S
E051-04	GROUP BY can contain columns not in select-list	S
E051-05	Select list items can be renamed	S
E051-06	HAVING clause	S
E051-07	Qualified * in select list	S
E051-08	Correlation names in the FROM clause	S
E051-09	Rename columns in the FROM clause	S
E061	Basic predicates and search conditions (FULL support)	S
E061-01	Comparison predicate	S
E061-02	BETWEEN predicate	S
E061-03	IN predicate with list of values	S
E061-04	LIKE predicate	S
E061-05	LIKE predicate: ESCAPE clause	S
E061-06	NULL predicate	S
E061-07	Quantified comparison predicate	S
E061-08	EXISTS predicate	S
E061-09	Subqueries in comparison predicate	S
E061-11	Subqueries in IN predicate	S
E061-12	Subqueries in quantified comparison predicate	S
E061-13	Correlated subqueries	S
E061-14	Search condition	S
E071	Basic query expressions	S (FULL support)
E071-01	UNION DISTINCT table operator	S
E071-02	UNION ALL table operator	S
E071-03	EXCEPT DISTINCT table operator	S
E071-05	Columns combined via table operators need not have exactly the same data type	S
E071-06	Table operators in subqueries	S
E081	Basic Privileges	S
E081-01	SELECT privilege at the table level	S
E081-02	DELETE privilege	S
E081-03	INSERT privilege at the table level	S
E081-04	UPDATE privilege at the table level	S
E081-05	UPDATE privilege at the column level	S
E081-06	REFERENCES privilege at the table level	N (not checked? needs work)
E081-07	REFERENCES privilege at the column level	N (not checked? needs work)
E081-08	WITH GRANT OPTION	S
E081-09	USAGE privilege	N
E081-10	EXECUTE privilege	N (Not checked)
E091	Set functions (FULL support)	S
E091-01	AVG	S
E091-02	COUNT	S
E091-03	MAX	S
E091-04	MIN	S

E091-05	SUM	S
E091-06	ALL quantifier	S
E091-07	DISTINCT quantifier	S
E101	Basic data manipulation (FULL support)	S
E101-01	INSERT statement	S
E101-03	Searched UPDATE statement	S
E101-04	Searched DELETE statement	S
E111	Single row SELECT statement	S
E121	Basic cursor support	N (Cursors are not supported, no change in ..)
E121-01	DECLARE CURSOR	
E121-02	ORDER BY columns need not be in select list	S
E121-03	Value expressions in ORDER BY clause	N (should be checked)
E121-04	OPEN statement	N
E121-06	Positioned UPDATE statement	N
E121-07	Positioned DELETE statement	N
E121-08	CLOSE statement	N
E121-10	FETCH statement: implicit NEXT	N
E121-17	WITH HOLD cursors	N
E131	Null value support (nulls in lieu of values)	S
E141	Basic integrity constraints	S
E141-01	NOT NULL constraints	S
E141-02	UNIQUE constraints of NOT NULL columns	S
E141-03	PRIMARY KEY constraints	S
E141-04	Basic FOREIGN KEY constraint with the NO ACTION default	S
E141-06	CHECK constraints	N
E141-07	Column defaults	N
E141-08	NOT NULL inferred on PRIMARY KEY	N
E141-10	Names in a foreign key can be specified in any order (columns should be in the proper order)	N
E151	Transaction support	S
E151-01	COMMIT statement	S
E151-02	ROLLBACK statement	S
E152	Basic SET TRANSACTION statement	S
E152-01	SET TRANSACTION statement: ISOLATION LEVEL SERIALIZABLE clause	S
E152-02	SET TRANSACTION statement: READ ONLY and READ WRITE clauses	N
E153	Updatable queries with subqueries	S
E161	SQL comments using leading double minus	S

E171	SQLSTATE support	N
F021	Basic information schema	N
F021-01	COLUMNS view	
F021-02	TABLES view	
F021-03	VIEWS view	
F021-04	TABLE_CONSTRAINTS view	
F021-05	REFERENTIAL_CONSTRAINTS view	
F021-06	CHECK_CONSTRAINTS view	
F031	Basic schema manipulation	S
F031-01	CREATE TABLE statement to create persistent base tables	S
F031-02	CREATE VIEW statement	S
F031-03	GRANT statement	S
F031-04	ALTER TABLE statement: ADD COLUMN clause	S
F031-13	DROP TABLE statement: RESTRICT clause	S
F031-16	DROP VIEW statement: RESTRICT clause	S
F031-19	REVOKE statement: RESTRICT clause	S
F041	Basic joined table	S
F041-01	Inner join (but not necessarily the INNER keyword)	S
F041-02	INNER keyword	S
F041-03	LEFT OUTER JOIN	S
F041-04	RIGHT OUTER JOIN	S
F041-05	Outer joins can be nested	S
F041-07	The inner table in a left or right outer join can also be used in an inner join	S
F041-08	All comparison operators are supported (rather than just =)	S
F051	Basic date and time	S
F051-01	DATE data type (including DATE literal)	S
F051-02	TIME data type (including TIME literal) with fractional seconds precision of 0	S
F051-03	TIMESTAMP data type (including TIMESTAMP literal) with fractional seconds precision of 0 and 6	S
F051-04	Comparison predicate on DATE, TIME, and TIMESTAMP data types	S
F051-05	Explicit CAST between datetime types and character types	S
F051-06	CURRENT_DATE	S
F051-07	LOCALTIME	S
F051-08	LOCALTIMESTAMP	S
F081	UNION and EXCEPT in views	S
F131	Grouped operations	S
F131-01	WHERE, GROUP BY, and HAVING clauses supported in queries with grouped views	S (could use a test!)
F131-02	Multiple tables supported in queries with grouped views	S
F131-03	Set functions supported in queries with grouped views	S
F131-04	Subqueries with GROUP BY and HAVING clauses and grouped views	S

F131-05	Single row SELECT with GROUP BY and HAVING clauses and grouped views	S
F181	Multiple module support	N (limited support)
F201	CAST function	S
F221	Explicit defaults	S
F261	CASE expression	S
F261-01	Simple CASE	S
F261-02	Searched CASE	N
F261_03	NULLIF	S
F261-04	COALESCE	S
F311	Schema definition statement	S
F311-01	CREATE SCHEMA	S
F311-02	CREATE TABLE for persistent base tables	S
F311-03	CREATE VIEW	S
F311-04	CREATE VIEW: WITH CHECK OPTION	N
F311-05	GRANT statement	S
F471	Scalar subquery values	S
F481	Expanded NULL predicate	S
F501	Features and conformance views	N
F501-01	SQL_FEATURES view	
F501-02	SQL_SIZING view	
F501-03	SQL_LANGUAGES view	
F812	Basic flagging	N
S011	Distinct data types	S
S011-01	USER_DEFINED_TYPES view	N
T321	Basic SQL-invoked routines	N
T321-01	User-defined functions with no overloading	N
T321-02	User-defined stored procedures with no overloading	N
T321-03	Function invocation	N
T321-04	CALL statement	N
T321-05	RETURN statement	N
T321-06	ROUTINES view	N
T321-07	PARAMETERS view	N

Appendix B Conversion of MIL programs

B.1 MIL compilation

MonetDB comes with its programming language called MIL (Monet Interface Language). A large collection of MIL programs exists, which ideally should run without problems on the MonetDB version 5 Kernel. Unfortunately, life isn;t that easy. MIL was designed as a half-backed scripting language with dynamic typing, variable re-use with different types, little support for debugging, and permissive syntax. Furthermore, the compiler was intimately coupled with the runtime system, e.g. an identifier was immediately resolved against the modules being loaded and variables known within the prevaling scope. This was necessary to resolve their context dependent interpretation.

In MonetDB Version 4 a cleanup of MIL has already taken place which strikes a balance between forcing users to migrate to MAL and to continue using their MIL code base. It is realized with a minimalistic compiler, which produces MAL code for a large fragment of the MIL language. There is, however, no information on the runtime status. In particular, it means there is no knowledge on pre-existing variables, type names, procedures, etc..

With a little care many MIL programs can be compiled and run against the new kernel. The MonetDB test suites and predefined MIL proc's have been used to define the language boundary to be supported.

In writing MIL code, the following rules should be kept obeyed:

- do not change the type of variables within their scope.
- do not use type identifiers as ordinary variables
- properly type a new variable when it is initialized with nil, e.g. nil:int

Consider the following old MIL program stored in the file *my.mil*

```
proc hello(str s):void { s.print(); }
```

The migration step involves loading the `mil` module, whereafter the file can be compiled.

```
>include mil;
>mil.compile("my.mil");
function hello(s:str):void;
    _3 := io.print(s);
end hello;
```

B.2 Conversion Pitfalls

In this section we summarize the pitfalls and limitations encountered in using the MIL to MAL compiler.

Name spaces. The naming of functions has undergone a clean up action. Part of this name re-casting is dealt with in a transparent way for the user.

Multiplex operations. One of the old key features dropped is the generic multiplex operation. Although this language construct was dealt with rather efficiently in the MIL interpreter, we have opted for a more mundane approach. The multiplex operations were mostly used in the context of arithmetic expressions and coercions over BATs. In particular, those that have a `:void` head.

These operations are now directly supported by C implementations in the modules `batcalc, batstr, batmath, batcolor`. The generic case is still available in the form of a optimizer request, i.e. `optimizer.multiplex(S,A1,...,An)` The first argument should be a string literal that denotes the element operation over the remaining arguments. The optimzier bakes an interator block to produce the required result.

Beware, in the older version the semantics of the multiplex was based on join semantics over the head of the argument BATs. This has been dropped as well.

Signature changes. Aside from syntax restrictions imposed, several command signatures and command names have been changed. They have to translated partly by hand or the MIL program should be prepared for a proper translation.

For example, the BAT update routines do not return their target variable. This means you can not cascade a series of calls, e.g. the statement

```
a.select(1,2).access(BAT_WRITE).rename("C");
b.insert(0,1).insert(1,2);
```

should be split into multiple steps as follows:

```
a.select(1,2); a.access(BAT_WRITE); a.rename("C");
b.insert(0,1); b.insert(1,2);
```

Runtime dependencies. A major rewrite of the MIL program is required when it inspects the runtime environment. See the bbp and inspection modules for information on the operations available.

Appendix C MAL Instruction Overview

The table below gives a condensed overview of the operations defined in each of the modules. Further details can be obtained using the MAL commandline interface (See tobedone).

aggr.avg
aggr.count
aggr.max
aggr.min
aggr.prod
aggr.product
aggr.size
aggr.sum

alarm.alarm
alarm.ctime
alarm.epilogue
alarm.epoch
alarm.prelude
alarm.sleep
alarm.time
alarm.timers
alarm.usec

algebra.bandjoin
algebra.between
algebra.betweenCO
algebra.betweenOC
algebra.betweenOO
algebra.cardinality
algebra.copy
aggr.count
aggr.count_no_nil
algebra.difference
algebra.exist
algebra.fetch
algebra.fetchjoin
algebra.find
algebra.fragment
algebra.hashjoin
algebra.hashsplit
algebra.histogram
algebra.indexjoin
algebra.intersect
algebra.join
algebra.joinPath
algebra.kdiff
algebra.kintersect
algebra.kunion
algebra.kunique
algebra.like
algebra.markH
algebra.markT
aggr.max
algebra.merge
algebra.mergejoin
aggr.min
algebra.number
algebra.outercrossproduct
algebra.outerjoin
algebra.position
algebra.product
algebra.project
algebra.sample
algebra.sdifference
algebra.select
algebra.selectH
algebra.semijoin
algebra.sintersect
algebra.slice
algebra.sort
algebra.sortHT
algebra.sortT
algebra.sortTH
algebra.split
algebra.sum
algebra.sunion
algebra.sunique
algebra.thetajoin
algebra.topN
algebra.tunique
algebra.uhashsplit
algebra.union
algebra.unique
algebra.uselect

array.grid
array.print
array.product
array.project

bat.append
bat.delete
bat.getAccess
bat.getAlpha
bat.getBatSize
bat.getCapacity
bat.getDelta
bat.getHeadType
bat.getHeat
bat.getName
bat.getRole
bat.getSequenceBase
bat.getTailType
bat.hasAppendMode
bat.hasReadMode
bat.hasWriteMode
bat.info
bat.insert
bat.isCached
bat.isPersistent
bat.isSorted
bat.isSortedReverse
bat.isSynced
bat.isTransient
bat.isaKey
bat.isaSet
bat.load
bat.mirror
bat.new
bat.pack
bat.replace
bat.reverse
bat.revert
bat.save
bat.setAccess
bat.setAppendMode
bat.setCold
bat.setColumn
bat.setGarbage
bat.setHash
bat.setHot
bat.setKey
bat.setMemoryAdvise
bat.setMemoryMap
bat.setName
bat.setPersistent
bat.setReadMode
bat.setRole
bat.setSequenceBase
bat.setSet
bat.setSorted
bat.setTransient
bat.setWriteMode
bat.unload
bat.unpack

batcalc.!=
batcalc.%
batcalc.*
batcalc.+
batcalc.++
batcalc.-
batcalc.–
batcalc./
batcalc.<
batcalc.<=
batcalc.==
batcalc.>
batcalc.>=
batcalc.and
batcalc.dbl
batcalc.flt
batcalc.ifthen
batcalc.ifthenelse
batcalc.init
batcalc.int
batcalc.lng
batcalc.not
batcalc.oid
batcalc.or
batcalc.project
batcalc.sht
batcalc.xor

batcolor.blue	batcolor.green	batcolor.red	batcolor.value
batcolor.cb	batcolor.hsv	batcolor.rgb	
batcolor.color	batcolor.hue	batcolor.saturation	
batcolor.cr	batcolor.luminance	batcolor.str	
batmath.acos	batmath.cos	batmath.fmod	batmath.sinh
batmath.asin	batmath.cosh	batmath.log	batmath.sqrt
batmath.atan	batmath.exp	batmath.log10	batmath.tan
batmath.atan2	batmath.fabs	batmath.pow	batmath.tanh
batmath.ceil	batmath.floor	batmath.sin	
batstr.+	batstr.length	batstr.search	batstr.toLower
batstr.==	batstr.ltrim	batstr.startsWith	batstr.toUpper
batstr.bytes	batstr.match	batstr.str	batstr.trim
batstr.chrAt	batstr.r_search	batstr.string	batstr.unicodeAt
batstr.endsWith	batstr.rtrim	batstr.substitute	
bbp.bind	bbp.getDirty	bbp.getNames	bbp.open
bbp.close	bbp.getHeadType	bbp.getObjects	bbp.prelude
bbp.commit	bbp.getHeat	bbp.getRNames	bbp.release
bbp.deposit	bbp.getKind	bbp.getRefCount	bbp.releaseAll
bbp.destroy	bbp.getLRefCount	bbp.getStatus	bbp.take
bbp.discard	bbp.getLocation	bbp.getTailType	bbp.toString
bbp.getCount	bbp.getName	bbp.iterator	
blob.nitems	blob.null	blob.toblob	blob.tostring
box.close	box.discard	box.open	box.take
box.deposit	box.getBoxNames	box.release	box.toString
box.destroy	box.iterator	box.releaseAll	
bstream.create	bstream.destroy	bstream.read	
calc.!=	calc.>=	calc.getBATidentifier	calc.oid
calc.%	calc.>>	calc.ifthenelse	calc.or
calc.*	calc.abs	calc.int	calc.ptr
calc.+	calc.and	calc.inv	calc.setoid
calc.-	calc.bat	calc.isnil	calc.sht
calc./	calc.between	calc.isnotnil	calc.str
calc.<	calc.bit	calc.lng	calc.void
calc.<<	calc.chr	calc.max	calc.xor
calc.<=	calc.dbl	calc.min	
calc.==	calc.flt	calc.newoid	
calc.>	calc.getBAT	calc.not	
chop.getHead	chop.hasMoreChunks	chop.newChunkIterator	
chop.getTail	chop.hasMoreElements	chop.newIterator	
clients.clearDebug	clients.getLogins	clients.quit	clients.setListing
clients.getId	clients.getScenario	clients.setDebug	clients.setScenario
clients.getInfo	clients.getUsers	clients.setHistory	
color.blue	color.green	color.print	color.saturation
color.cb	color.hsv	color.printf	color.str
color.color	color.hue	color.red	color.value
color.cr	color.luminance	color.rgb	color.ycc
const.close	const.discard	const.open	const.releaseAll

const.deposit	const.hasMoreElements	const.prelude	const.take
const.destroy	const.newIterator	const.release	const.toString
crackers.crack	crackers.getPiece	crackers.new	crackers.selectPieces
crackers.drop	crackers.hasMoreElements	crackers.newIterator	
crackers.dropView	crackers.info	crackers.select	
date.!=	date.<=	date.>	date.isnil
date.<	date.=	date.>=	
daytime.!=	daytime.<=	daytime.>	daytime.isnil
daytime.<	daytime.=	daytime.>=	
factory.getCaller	factory.getOwners	factory.getTimeStamp	factory.setLocation
factory.getLocations	factory.getPlants	factory.getTimeStampOut	factory.shutdown
group.avg	group.max	group.refine	
group.count	group.min	group.size	
group.group	group.prelude	group.sum	
inspect.getAtomNames	inspect.getDefinition	inspect.getModule	inspect.getTypeName
inspect.getAtomSizes	inspect.getFunction	inspect.getSignature	
inspect.getAtomSuperNames	inspect.getKind	inspect.getType	
io.export	io.print	io.prompt	io.stdout
io.ftable	io.printf	io.stderr	io.table
io.import	io.printft	io.stdin	
lock.create	lock.set	lock.try	
lock.destroy	lock.tostr	lock.unset	
mal.assert	mal.newRange	mal.raise	mal.source
mal.call	mal.nextElement	mal.register	
manual.create	manual.createHelp	manual.createOverview	manual.help
mapi.epilogue	mapi.listen_ssl	mapi.resume	mapi.suspend
mapi.listen	mapi.prelude	mapi.stop	mapi.trace
mdb.List	mdb.getExceptionVariable	mdb.inspect	mdb.setTrace
mdb.getDefinition	mdb.getStackDepth	mdb.list	mdb.start
mdb.getExceptionContext	mdb.getStackFrame	mdb.setCatch	mdb.stop
mdb.getExceptionReason	mdb.getStackTrace	mdb.setTimer	mdb.var
mmath.acos	mmath.exp	mmath.log	mmath.sqrt
mmath.asin	mmath.fabs	mmath.log10	mmath.srand
mmath.atan	mmath.finite	mmath.pow	mmath.tan
mmath.atan2	mmath.floor	mmath.rand	mmath.tanh
mmath.ceil	mmath.fmod	mmath.round	
mmath.cos	mmath.isinf	mmath.sin	
mmath.cosh	mmath.isnan	mmath.sinh	
mtime.add	mtime.date_sub_sec_interval	mtime.milliseconds	mtime.start_dst
mtime.adddays	mtime.day	mtime.minutes	mtime.time_synonyms
mtime.addmonths	mtime.dayname	mtime.month	mtime.timestamp
mtime.addyears	mtime.daynum	mtime.monthname	mtime.timezone
mtime.compute	mtime.dayofweek	mtime.monthnum	mtime.tzone
mtime.current_date	mtime.dayofyear	mtime.msecs	mtime.tzone_local
mtime.current_time	mtime.daytime	mtime.olddate	mtime.weekday
mtime.current_timestamp	mtime.diff	mtime.oldduration	mtime.weekofyear
mtime.date	mtime.dst	mtime.prelude	mtime.year

mtime.date_add_month_interval
mtime.date_add_sec_interval
mtime.end_dst
mtime.hours
mtime.rule
mtime.seconds
optimizer.aliasRemoval
optimizer.clrDebug
optimizer.commonExpressionRemoval
optimizer.costEstimation
optimizer.deadCodeRemoval
optimizer.dumpQEP
optimizer.emptySet
optimizer.expressionAccumulation
optimizer.factorize
optimizer.garbageCollector
optimizer.macroContraction
optimizer.macroExpansion
optimizer.multiplex
optimizer.optimize
optimizer.partitionedQuery
optimizer.peephole
optimizer.prelude
optimizer.reduce
optimizer.setDebug
optimizer.showFlowGraph
optimizer.showPlan
optimizer.singleton
optimizer.strengthReduction
partitions.close
partitions.compress
partitions.delete
partitions.deposit
partitions.destroy
partitions.discard
partitions.dump
partitions.getLastPartition
partitions.getRange
partitions.hasMoreElements
partitions.insert
partitions.newIterator
partitions.open
partitions.prelude
partitions.release
partitions.releaseAll
partitions.setCapacity
partitions.take
partitions.toString
pcre.compile
pcre.match
pcre.null
pcre.select
pcre.uselect
profiler.activate
profiler.clearTrace
profiler.closeStream
profiler.clrFilter
profiler.deactivate
profiler.dumpTrace
profiler.getTrace
profiler.openStream
profiler.setAll
profiler.setCachedProfiling
profiler.setEndPoint
profiler.setFilter
profiler.setNone
profiler.setOfflineProfiling
profiler.setOnlineProfiling
profiler.setStartPoint
profiler.start
profiler.startTrace
profiler.stop
profiler.stopTrace
rule.define
sema.create
sema.destroy
sema.down
sema.up
statistics.close
statistics.deposit
statistics.destroy
statistics.discard
statistics.dump
statistics.forceUpdate
statistics.getCount
statistics.getHistogram
statistics.getHotset
statistics.getMax
statistics.getMin
statistics.getObjects
statistics.getSize
statistics.hasMoreElements
statistics.newIterator
statistics.open
statistics.prelude
statistics.release
statistics.releaseAll
statistics.take
statistics.toString
statistics.update
status.cpuStatistics
status.gdkEnv
status.gdkThread
status.ioStatistics
status.memStatistics
status.memUsage
status.mem_cursize
status.mem_maxsize
status.vmUsage
status.vm_cursize
status.vm_maxsize
str.+
str.STRprelude
str.chrAt
str.codeset
str.endsWith
str.iconv
str.length
str.ltrim
str.nbytes
str.r_search
str.rtrim
str.search
str.startsWith
str.str
str.string
str.substitute
str.toLower
str.toUpper
str.trim
str.unicode
str.unicodeAt
stream.blocked
stream.close
stream.flush
stream.openRead
stream.openReadBytes
stream.openWrite
stream.openWriteBytes
stream.readInt
stream.readStr
stream.socketRead
stream.socketReadBytes
stream.socketWrite
stream.socketWriteBytes
stream.writeInt
stream.writeStr
tablet.display
tablet.dump
tablet.finish
tablet.firstPage
tablet.getPage
tablet.getPageCnt
tablet.input
tablet.lastPage
tablet.load
tablet.nextPage
tablet.output
tablet.page
tablet.setColumn
tablet.setColumnBracket
tablet.setColumnDecimal
tablet.setColumnName
tablet.setColumnNull
tablet.setColumnPosition
tablet.setDelimiter
tablet.setFormat
tablet.setPivot
tablet.setProperties
tablet.setRowBracket
tablet.setStream

tablet.header
tablet.prevPage
tablet.setColumnWidth
tablet.setTableBracket
thread.exec
thread.isDead
thread.kill
thread.suspend
thread.terminate
thread.wait
timestamp.!=
timestamp.<
timestamp.<=
timestamp.=
timestamp.>
timestamp.>=
timestamp.epoch
timestamp.isnil
timestamp.unix_epoch
transaction.abort
transaction.alpha
transaction.clean
transaction.commit
transaction.delta
transaction.sync
tzone.str
tzone.timestamp
unix.getenv
unix.setenv
url.getAnchor
url.getBasename
url.getContent
url.getContext
url.getDirectory
url.getDomain
url.getExtension
url.getFile
url.getHost
url.getPort
url.getProtocol
url.getQuery
url.getQueryArg
url.getRobotURL
url.getUser
url.isURL
url.newurl
url.url
user.main

Appendix D MAL Instruction Help

The table below summarizes the first commentary line encountered in the system associated with a MAL operation.

aggr.avg	aggr.count_no_nil	aggr.min	aggr.sum
aggr.cardinality	aggr.histogram	aggr.product	
aggr.count	aggr.max	aggr.size	
alarm.alarm	alarm.epoch	alarm.time	
alarm.ctime	alarm.prelude	alarm.timers	
alarm.epilogue	alarm.sleep	alarm.usec	
algebra.antijoin	algebra.groupby	algebra.merge	algebra.sort
algebra.bandjoin	algebra.hashjoin	algebra.mergejoin	algebra.sortHT
algebra.between	algebra.hashsplit	algebra.number	algebra.sortHead
algebra.betweenCO	algebra.indexjoin	algebra.outerjoin	algebra.sortTH
algebra.betweenOC	algebra.intersect	algebra.position	algebra.split
algebra.betweenOO	algebra.join	algebra.project	algebra.sunion
algebra.copy	algebra.joinPath	algebra.revert	algebra.sunique
algebra.crossproduct	algebra.kdifference	algebra.sample	algebra.thetajoin
algebra.difference	algebra.kintersect	algebra.sdifference	algebra.topN
algebra.exist	algebra.kunion	algebra.select	algebra.tunique
algebra.fetch	algebra.kunique	algebra.selectH	algebra.uhashsplit
algebra.fetchjoin	algebra.like	algebra.semijoin	algebra.union
algebra.find	algebra.markH	algebra.sintersect	algebra.unique
algebra.fragment	algebra.markT	algebra.slice	algebra.uselect
array.grid	array.print	array.product	array.project
bat.append	bat.hasReadMode	bat.new	bat.setMemoryAdvise
bat.delete	bat.hasWriteMode	bat.pack	bat.setMemoryMap
bat.getAccess	bat.info	bat.replace	bat.setName
bat.getAlpha	bat.insert	bat.reverse	bat.setPersistent
bat.getBatSize	bat.isCached	bat.revert	bat.setReadMode
bat.getCapacity	bat.isPersistent	bat.save	bat.setRole
bat.getDelta	bat.isSorted	bat.setAccess	bat.setSequenceBase
bat.getHeadType	bat.isSortedReverse	bat.setAppendMode	bat.setSet
bat.getHeat	bat.isSynced	bat.setCold	bat.setSorted
bat.getName	bat.isTransient	bat.setColumn	bat.setTransient
bat.getRole	bat.isaKey	bat.setGarbage	bat.setWriteMode
bat.getSequenceBase	bat.isaSet	bat.setHash	bat.unload
bat.getTailType	bat.load	bat.setHot	bat.unpack
bat.hasAppendMode	bat.mirror	bat.setKey	
batcalc.!=	batcalc./	batcalc.dbl	batcalc.not
batcalc.%	batcalc.<	batcalc.flt	batcalc.oid
batcalc.*	batcalc.<=	batcalc.ifthen	batcalc.or
batcalc.+	batcalc.==	batcalc.ifthenelse	batcalc.project
batcalc.++	batcalc.>	batcalc.init	batcalc.sht
batcalc.-	batcalc.>=	batcalc.int	batcalc.str
batcalc.–	batcalc.and	batcalc.lng	batcalc.xor
batcolor.blue	batcolor.green	batcolor.red	batcolor.value

batcolor.cb	batcolor.hsv	batcolor.rgb	
batcolor.color	batcolor.hue	batcolor.saturation	
batcolor.cr	batcolor.luminance	batcolor.str	
batmath.acos	batmath.cos	batmath.fmod	batmath.sinh
batmath.asin	batmath.cosh	batmath.log	batmath.sqrt
batmath.atan	batmath.exp	batmath.log10	batmath.tan
batmath.atan2	batmath.fabs	batmath.pow	batmath.tanh
batmath.ceil	batmath.floor	batmath.sin	
batstr.!=	batstr.length	batstr.search	batstr.toUpper
batstr.+	batstr.like	batstr.startsWith	batstr.trim
batstr.==	batstr.ltrim	batstr.str	batstr.unicodeAt
batstr.bytes	batstr.match	batstr.string	
batstr.chrAt	batstr.r_search	batstr.substitute	
batstr.endsWith	batstr.rtrim	batstr.toLower	
bbp.bind	bbp.getDirty	bbp.getNames	bbp.open
bbp.close	bbp.getHeadType	bbp.getObjects	bbp.prelude
bbp.commit	bbp.getHeat	bbp.getRNames	bbp.release
bbp.deposit	bbp.getKind	bbp.getRefCount	bbp.releaseAll
bbp.destroy	bbp.getLRefCount	bbp.getStatus	bbp.take
bbp.discard	bbp.getLocation	bbp.getTailType	bbp.toString
bbp.getCount	bbp.getName	bbp.iterator	
blob.nitems	blob.null	blob.toblob	blob.tostring
box.close	box.discard	box.open	box.take
box.deposit	box.getBoxNames	box.release	box.toString
box.destroy	box.iterator	box.releaseAll	
bstream.create	bstream.destroy	bstream.read	
calc.!=	calc.>=	calc.getBATidentifier	calc.not
calc.%	calc.>>	calc.ifthenelse	calc.oid
calc.*	calc.abs	calc.int	calc.or
calc.+	calc.and	calc.inv	calc.ptr
calc.-	calc.bat	calc.isnil	calc.setoid
calc./	calc.between	calc.isnotnil	calc.sht
calc.<	calc.bit	calc.length	calc.str
calc.<<	calc.chr	calc.lng	calc.void
calc.<=	calc.dbl	calc.max	calc.xor
calc.==	calc.flt	calc.min	
calc.>	calc.getBAT	calc.newoid	
chopper.getHead	chopper.getTail	chopper.hasMoreElements	chopper.newIterator
clients.addUser	clients.getLogins	clients.newPassword	clients.setScenario
clients.checkPermission	clients.getPasswords	clients.quit	clients.trace
clients.delUser	clients.getPermissions	clients.setDebug	
clients.getId	clients.getScenario	clients.setHistory	
clients.getInfo	clients.getUsers	clients.setListing	
color.blue	color.green	color.print	color.saturation
color.cb	color.hsv	color.printf	color.str
color.color	color.hue	color.red	color.value
color.cr	color.luminance	color.rgb	color.ycc

const.close	const.epiloque	const.prelude	const.toString
const.deposit	const.hasMoreElements	const.release	
const.destroy	const.newIterator	const.releaseAll	
const.discard	const.open	const.take	
crackers.addOne	crackers.crack_SM	crackers.newIterator	crackers.zcrackOrdered_SM
crackers.addTwo	crackers.crack_sm	crackers.printIndex	crackers.zcrackOrdered_is
crackers.crack	crackers.crack_validate	crackers.range	crackers.zcrackOrdered_sm
crackers.crackOrdered_MK	crackers.destroy	crackers.select	crackers.zcrackOrdered_validate
crackers.crackOrdered_SM	crackers.getPiece	crackers.selectPieces	crackers.zcrack_MK
crackers.crackOrdered_is	crackers.hasMoreElements	crackers.setGranule	crackers.zcrack_SM
crackers.crackOrdered_sm	crackers.info	crackers.setLimit	crackers.zcrack_sm
crackers.crackOrdered_validate	crackers.new	crackers.uselect	crackers.zcrack_validate
crackers.crack_MK	crackers.newIndex	crackers.zcrackOrdered_MK	
date.!=	date.<=	date.>	date.date
date.<	date.=	date.>=	date.isnil
daytime.!=	daytime.<=	daytime.>	daytime.isnil
daytime.<	daytime.=	daytime.>=	
factory.getCaller	factory.getOwners	factory.getTimeStampIn	factory.setLocation
factory.getLocations	factory.getPlants	factory.getTimeStampOut	factory.shutdown
group.avg	group.max	group.prelude	group.size
group.count	group.min	group.refine	group.sum
group.derive	group.new	group.refine_reverse	
inet.!=	inet.>	inet.host	inet.new
inet.<	inet.>=	inet.hostmask	inet.setmasklen
inet.<<	inet.>>	inet.isnil	inet.text
inet.<<=	inet.>>=	inet.masklen	
inet.<=	inet.abbrev	inet.netmask	
inet.=	inet.broadcast	inet.network	
inspect.getAddress	inspect.getComment	inspect.getModule	inspect.getTypeIndex
inspect.getAddresses	inspect.getDefinition	inspect.getSignature	inspect.getTypeName
inspect.getAtomNames	inspect.getEnvironment	inspect.getSignatures	
inspect.getAtomSizes	inspect.getFunction	inspect.getSize	
inspect.getAtomSuperNames	inspect.getKind	inspect.getType	
io.export	io.print	io.prompt	io.stdout
io.ftable	io.printf	io.stderr	io.table
io.import	io.printft	io.stdin	
language.assert	language.newRange	language.raise	language.source
language.call	language.nextElement	language.register	
lock.create	lock.set	lock.try	
lock.destroy	lock.tostr	lock.unset	
manual.completion	manual.createHelp	manual.help	
manual.create	manual.createOverview	manual.search	
mat.expand	mat.iterator	mat.pack	
mat.info	mat.new	mat.print	
mdb.List	mdb.getExceptionVariable	mdb.inspect	mdb.setTrace
mdb.getDefinition	mdb.getStackDepth	mdb.list	mdb.start
mdb.getExceptionContext	mdb.getStackFrame	mdb.setCatch	mdb.stop

mdb.getExceptionReason	mdb.getStackTrace	mdb.setTimer	mdb.var
mkey.bulk_rotate_xor_hash	mkey.hash	mkey.rotate	
mmath.acos	mmath.exp	mmath.log	mmath.sqrt
mmath.asin	mmath.fabs	mmath.log10	mmath.srand
mmath.atan	mmath.finite	mmath.pow	mmath.tan
mmath.atan2	mmath.floor	mmath.rand	mmath.tanh
mmath.ceil	mmath.fmod	mmath.round	
mmath.cos	mmath.isinf	mmath.sin	
mmath.cosh	mmath.isnan	mmath.sinh	
mserver.connect	mserver.explain	mserver.malclient	mserver.stop
mserver.epilogue	mserver.listen	mserver.prelude	mserver.suspend
mserver.error	mserver.listen_ssl	mserver.resume	
mtime.add	mtime.dayname	mtime.month	mtime.time_synonyms
mtime.adddays	mtime.daynum	mtime.monthname	mtime.timestamp
mtime.addmonths	mtime.dayofweek	mtime.monthnum	mtime.timestamp_add_month_interval
mtime.addyears	mtime.dayofyear	mtime.msec	mtime.timestamp_add_sec_interval
mtime.compute	mtime.daytime	mtime.msecs	mtime.timestamp_sub_month_interval
mtime.current_date	mtime.diff	mtime.olddate	mtime.timestamp_sub_sec_interval
mtime.current_time	mtime.dst	mtime.oldduration	mtime.timezone
mtime.current_timestamp	mtime.end_dst	mtime.prelude	mtime.tzone
mtime.date	mtime.epilogue	mtime.rule	mtime.tzone_local
mtime.date_add_month_interval	mtime.hours	mtime.seconds	mtime.weekday
mtime.date_add_sec_interval	mtime.local_timezone	mtime.start_dst	mtime.weekofyear
mtime.date_sub_sec_interval	mtime.milliseconds	mtime.time_add_sec_interval	mtime.year
mtime.day	mtime.minutes	mtime.time_sub_sec_interval	
optimizer.aliasRemoval	optimizer.dumpQEP	optimizer.macroExpansion	optimizer.reduce
optimizer.clrDebug	optimizer.emptySet	optimizer.multiplex	optimizer.setDebug
optimizer.coercions	optimizer.expressionAccumulation	optimizer.multitable	optimizer.showFlowGraph
optimizer.commonExpressionRemoval	optimizer.factorize	optimizer.optimize	optimizer.showPlan
optimizer.costEstimation	optimizer.garbageCollector	optimizer.partitionedQuery	optimizer.singleton
optimizer.crack	optimizer.generators	optimizer.peephole	optimizer.strengthReduction
optimizer.deadCodeRemoval	optimizer.macroContraction	optimizer.prelude	
partitions.close	partitions.discard	partitions.hasMoreElements	partitions.release
partitions.compress	partitions.dump	partitions.insert	partitions.releaseAll
partitions.delete	partitions.epilogue	partitions.newIterator	partitions.setCapacity
partitions.deposit	partitions.getLastPartition	partitions.open	partitions.take
partitions.destroy	partitions.getRange	partitions.prelude	partitions.toString
pbm.close	pbm.epilogue	pbm.getNextName	pbm.setReadMode
pbm.compress	pbm.generator	pbm.getRange	pbm.setWriteMode
pbm.deposit	pbm.getComponents	pbm.newIterator	pbm.take
pbm.destroy	pbm.getLast	pbm.open	
pbm.discard	pbm.getNames	pbm.prelude	
pbm.dump	pbm.getNextElement	pbm.releaseAll	
pcre.compile	pcre.null	pcre.select	
pcre.like	pcre.prelude	pcre.sql2pcre	
pcre.match	pcre.replace	pcre.uselect	
pqueue.dequeue_max	pqueue.enqueue_min	pqueue.topn_min	

pqueue.dequeue_min
pqueue.enqueue_max
pqueue.init
pqueue.topn_max
pqueue.topreplace_max
pqueue.topreplace_min
profiler.activate
profiler.cleanup
profiler.clearTrace
profiler.closeStream
profiler.clrFilter
profiler.deactivate
profiler.dumpTrace
profiler.getTrace
profiler.openStream
profiler.setAll
profiler.setCachedProfiling
profiler.setEndPoint
profiler.setFilter
profiler.setNone
profiler.setOfflineProfiling
profiler.setOnlineProfiling
profiler.setStartPoint
profiler.start
profiler.startTrace
profiler.stop
profiler.stopTrace
sema.create
sema.destroy
sema.down
sema.up
statistics.close
statistics.deposit
statistics.destroy
statistics.discard
statistics.dump
statistics.epilogue
statistics.forceUpdate
statistics.getCount
statistics.getHistogram
statistics.getHotset
statistics.getMax
statistics.getMin
statistics.getObjects
statistics.getSize
statistics.hasMoreElements
statistics.newIterator
statistics.open
statistics.prelude
statistics.release
statistics.releaseAll
statistics.take
statistics.toString
statistics.update
status.batStatistics
status.cpuStatistics
status.getDatabases
status.getPorts
status.getThreads
status.ioStatistics
status.memStatistics
status.memUsage
status.mem_cursize
status.mem_maxsize
status.vmStatistics
status.vm_cursize
status.vm_maxsize
str.+
str.STRepilogue
str.STRprelude
str.ascii
str.chrAt
str.codeset
str.endsWith
str.iconv
str.insert
str.length
str.like
str.locate
str.ltrim
str.nbytes
str.prefix
str.r_search
str.repeat
str.replace
str.rtrim
str.search
str.space
str.startsWith
str.str
str.string
str.stringlength
str.substitute
str.substring
str.suffix
str.toLower
str.toUpper
str.trim
str.unicode
str.unicodeAt
streams.blocked
streams.close
streams.flush
streams.openRead
streams.openReadBytes
streams.openWrite
streams.openWriteBytes
streams.readInt
streams.readStr
streams.socketRead
streams.socketReadBytes
streams.socketWrite
streams.socketWriteBytes
streams.writeInt
streams.writeStr
tablet.display
tablet.dump
tablet.finish
tablet.firstPage
tablet.getPage
tablet.getPageCnt
tablet.header
tablet.input
tablet.lastPage
tablet.load
tablet.nextPage
tablet.output
tablet.page
tablet.prevPage
tablet.setColumn
tablet.setColumnBracket
tablet.setColumnDecimal
tablet.setColumnName
tablet.setColumnNull
tablet.setColumnPosition
tablet.setColumnWidth
tablet.setDelimiter
tablet.setFormat
tablet.setPivot
tablet.setProperties
tablet.setRowBracket
tablet.setStream
tablet.setTableBracket
timestamp.!=
timestamp.<
timestamp.<=
timestamp.=
timestamp.>
timestamp.>=
timestamp.epoch
timestamp.isnil
timestamp.unix_epoch
transaction.abort
transaction.alpha
transaction.clean
transaction.commit
transaction.delta
transaction.sync
tzone.str
tzone.timestamp
unix.getenv
unix.setenv
url.getAnchor
url.getDomain
url.getProtocol
url.isURL

url.getBasename
url.getContent
url.getContext
url.getDirectory
url.getExtension
url.getFile
url.getHost
url.getPort
url.getQuery
url.getQueryArg
url.getRobotURL
url.getUser
url.new
url.url
user.main
zrule.define