MonetDB Internals

MonetDB Internals mk Sun, 03/21/2010 - 01:47

This section contains more details on the MonetDB MAL language and the server code base.

MonetDB architecture

MonetDB architecture mk Tue, 03/30/2010 - 11:53

General overview. MonetDB belongs to the class of database management systems designed primarilly for datawarehouse environments. They are characterised by large repositories, which are mostly queried for business intelligence decisions. They also appear frequently in the area of science, where observations are collected into a warehouse for subsequent scientific analysis. The design is focussed on bulk processing wherever possible. Aimed at maximal efficient use of the hardware for large scale processing. This design objective is reflected at all levels of the architecture and the functionality offered to the user. Although MonetDB/SQL provides a full-fledged SQL interface, it has not been tuned towards high-volume transaction processing with its accompanying multi-level ACID properties.

SQL front-end. The storage layer combined with the MAL algebra provides a flexible architecture to accomodate a wide variety of query languages. The relational front-end decomposes tables by column with a dense (non-stored) OID head, and a tail column with values. For each relational table, a column with deleted positions is kept. Deltas are designed to delay updates to the main columns, and allow a relatively cheap snapshot isolation mechanism. MonetDB/SQL also maintains additional join indices to speed-up foreign key join predicates; and value indices are created on-the-fly.

Design considerations

Design considerations mk Thu, 02/20/2014 - 03:53

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 and is composed of 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 presupposes an intuition on 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 widespread 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, evaluationing a mathematical function, e.g., Gaussian, or evalauting a regular expression 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. It results in an architecture where you 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 and versatility, and is 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.

Storage model

Storage model mk Wed, 02/26/2014 - 13:03

The storage model deployed in MonetDB is a significant deviation of traditional database systems. It represents relational tables using vertical fragmentation, by storing each column in a separate {*OID,value*} table, also called a BAT (Binary Association Table). MonetDB relies on a low-level relational algebra called the *BAT algebra*, which takes BATs and scalar values as input. The complete result is always stored in (intermediate) BATs, and the result of an SQL query is a collection of BATs.

Each column, or BAT, is implemented as an ordinary C-array where the OID maps to the index in the array possibly extended with a so-called sequence-base offset. The persistent version of a BAT is a memory mapped file. OID lookup becomes a fast array indexed read into the tail column. In effect, this use of arrays in virtual memory exploits the fast in-hardware address to disk-block mapping implemented by the MMU (memory management unit) in a CPU to provide an O(1) positional database lookup mechanism.

To ensure fast access wherever possible, all (relational) operators exploit a small set of properties maintained under updates and intermediates creation.

```
seq       - the sequence base, a mapping from array index 0 into a OID value
key       - the values in the column are unique
nil       - there is at least one NIL value
nonil     - it is unknown if there NIL values
dense     - the numeric values in the colum form a dense sequence
sorted    - the column contains a sorted list for ordered domains
revsorted - the column contains a reversed sorted list
```

Execution model

Execution model mk Wed, 02/26/2014 - 13:05

The MonetDB kernel is an abstract machine, programmed in the MonetDB Assemblee Language (MAL). Each relational algebra operator corresponds to a MAL instruction. Each BAT algebra operator maps to a simple MAL instruction, which has zero degrees of freedom; it does not take complex expressions as parameter. Rather, complex expressions are broken down into a sequence of BAT algebra operators that each perform a simple operation on an entire column of values (``bulk processing''). This allows the implementation of the BAT algebra to forsake an expression interpreting engine; rather all BAT algebra operations in the implementation map onto simple array operations.
For instance, the BAT algebra expression

```
   R:bat[:oid]:=select(B:bat[:int], V:int);
```

can be implemented at the C code level like:

```
   for (i = j = 0; i < n; i++)
     if (B.tail[i] == V)
       R.tail[j++] = i;
```

The BAT algebra operators have the advantage that tight for-loops create high data access and instruction locality. Such simple loops are amenable to compiler optimization (loop pipelining, vectorization, blocking, strength reduction), and CPU out-of-order speculative execution.

Software stack

Software stack mk Wed, 02/26/2014 - 13:07

MonetDB's query processing scheme is centered around three software layers. The top is formed by the query language parser and a heuristic, language - and data model - specific optimizer to reduce the amount of data produced by intermediates and to exploit catalogue knowledge, such as foreign key constraints . The output is a logical plan expressed in MAL.

The second tier consists of a collection of optimizer modules, which are assembled into an optimization pipeline. Each optimizer takes a MAL plan and transforms it into a more efficient one, optionally sprinkled with resource management directives and flow of control directives. The modules provide facilities ranging from symbolic processing up to just-in-time data distribution and execution. The approach breaks with the hitherto omnipresent cost-based optimizers by recognition that not all decisions can be cast together in a single cost formula. Operating on the common binary-relational back-end algebra, these optimizer modules are shared by all front-end data models and query languages.

The third tier, the MAL interpreter, contains the library of highly optimized implementation of the binary relational algebra operators. They maintain properties over the object accessed to gear the selection of subsequent algorithms. For example, the Select operator can benefit both from sorted-ness of the BAT or it may call for a sample to derive the expected sizes. The interface is described in the MAL module sections.

MAL reference

MAL reference mk Tue, 03/30/2010 - 11:49

The primary textual interface to the Monetdb kernel is a simple, assembly-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 choosen in a 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 paradigm and denote a synchronization point.

MAL is the target language for query compilers 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

Literals

Literals mk Tue, 03/30/2010 - 11:51

Literals in MAL follow the lexical conventions of the programming language C. A default type is attached, e.g. the literal 1 is typed as an `int` value. Likewise, the literal 3.14 is typed `flt` rather than `dbl`.

A literal can be coerced to another type by tagging it with a type qualifier, provided a coercion operation is defined. For example, `1:lng` marks the literal 1 as of type `lng`. and `"1999-12-10":date` creates a `date` literal.

MonetDB comes with the hardwired types `bit`, `bte`, `chr`, `wrd`, `sht`, `int`, `lng`, `oid`, `flt`, `dbl`, and `str`. The kernel code has been optimized to deal with these types efficiently, i.e. without unnecessary function call overheads. In addition, the system supports temporal types `date`, `daytime`, `time`, `timestamp`, `timezone`, extensions to deal with `IPv4` addresses and URLs using `inet`, url, UUID and JSON formatted strings. An 128-integer arithmetic is compiled in on platforms that can support it.

Variables

Variables mk Tue, 03/30/2010 - 11:59

Variables are denoted by identifers and implicitly defined upon first assignment. They take on a type through a type classifier or inherit it from the context in which they are first used, see MAL Type System.

Variables are organized into two classes: user defined and internal variables. User defined variables start with a letter and temporary variables, e.g. generated internally by optimizers, start with a `X_`. In general internal variables can not be used in MAL programs directly, but they may become visible in MAL program listings or during debugging.

MAL variables are internally represented by their position into the symbol table and runtime value stack. Internal variable names are recognized by the parser and an error is produced if their name does not align with the expected position in the symbol table.

Instructions

Instructions mk Tue, 03/30/2010 - 11:53

A MAL instruction has purposely a simple format. 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 := literal;
   (t1,...,tn) := (a1,...,an);
```

Operators are grouped into user defined modules. Omission of the module name is interpreted as the `user` module.

Simple 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 internal variables to hold the result of the expression upon need. They won't show up when you list the MAL program unless they are being used elsewhere.

Each instruction should fit on a single line, which makes it easy for the parser to recover upon encountering a syntax error. 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 contains a sequence of MAL statements and a symbol table. The MAL instruction record 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. Variable- and statement- block together describe the static part of a MAL procedure. It carries enough information to produce a listing and to aid symbolic debugging.

The complete MAL syntax can be found here.

Type system

Type system mk Tue, 03/30/2010 - 11:58

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. MonetDB comes with the hardwired scalar types **bit, bte, sht, int, lng, hge, oid, flt, dbl** and **str**. 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, daytime, time, timestamp, timezone, blob,** and web supportive types **inet, url** and **json**. 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)

Type resolutions

MAL is mostly a strongly typed language. 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 overhead 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[:oid,tt] where tt is a runtime dependent type, which means that the selection operation can not be type-checked immediately. It is an example of an embedded polymorphic 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 underturmined typed argument we should postpone type checking 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.

Defining your own types

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

Flow of control

Flow of control mk Tue, 03/30/2010 - 11:54

The flow of control within a MAL program block can be changed by tagging a statement 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 primary target variable, also called the control variable.

The LEAVE and REDO are conditional flow modifiers. The control variable 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 guarded 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 guarded block 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.

The guarded 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 in accordance with the data-flow dependencies. The partial blocks order 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;
        io.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 iftpart:= i<1;
     io.print("ok");
exit iftpart;
barrier elsepart:= i>=1;
     io.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:= M.newIterator(T);
        elm:= M.getElement(T,i);
        ...
        leave i:= M.noMoreElements(T);
        ...
        redo i:= M.hasMoreElements(T);
exit i:= M.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 for BATs as follows:

```
    barrier (idx,hd,tl):= bat.newIterator(B);
```

```
    ...
    redo (idx,hd,tl):= bat.hasMoreElements(B);
exit (idx,hd,tl);
```

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

Exceptions

Exceptions mk Tue, 03/30/2010 - 23:34

MAL comes with an exception handling mechanism, similar in style as found in modern programming languages. Exceptions are considered rare situations that alter the flow of control to a place where they can be handled. After the exceptional case has been handled the following options exist a) continue where it went wrong, b) retry the failed instruction, c) leave the block where the exception was handled, or d) pass the exception to an enclosing call. The current implementation of the MAL interpreter only supports c) and d).

Exception control

The exception handling keywords are: CATCH and RAISE The CATCH marks a point in the dataflow where an exception raised can be dealt with. Any statement between the point where it is raised and the catch block is ignored. Moreover, the CATCH ...EXIT block is ignored when no errors have occurred in the preceeding dataflow structure. Within the catch block, the exception variable can be manipulated without constraints.

An exception message is linked with a exception variable of type string. If this variable is defined in the receiving block, the exception message can be delivered. Otherwise, it implicitly raises the exception in the surrounding scope. The variable ANYexception can be used to catch them irrespective of their class.

After an exception has been dealt with, the catch block can be left at the normal exit.

Both LEAVE and REDO are conditional flow of control modifiers, which trigger on a non-empty string variable. An exception raised within a catch-block terminates the function and returns control to the enclosing environment.

The argument to the catch statement is a target list, which holds the exception variables you are interested in.

The snippet below illustrates how an exception raised in the function io.read is caught using the exception variable IOerror. After dealing with it locally, it raises a new exception FATALerror for the enclosing call.

```
    io.write("Welcome");
    ...
catch IOerror:str;
    io.print("input error on reading password");
raise FATALerror:= "Can't handle it";
exit IOerror;
```

Since CATCH is a flow control modifier it can be attached to any assignment statement. This statement is executed whenever there is no exception outstanding, but will be ignored when control is moved to the block otherwise.

Builtin exceptions

The policy implemented in the MAL modules, and recognized by the interpreter, is to return a string value by default. A NULL return value indicates succesful execution; otherwise the string encodes information to analyse the error occurred.

This string pattern is strictly formatted and easy to analyse. It starts with the name of the exception variable to be set, followed by an indication where the exception was raise, i.e. the function name and the program counter, and concludes with specific information needed to interpret and handle the exception.

For example, the exception string 'MALException:Admin.main[2]:address of function missing' denotes an exception raised while typechecking a MAL program.

The exceptions captured within the kernel are marked as 'GDKerror'. At that level there is no knowledge about the MAL context, which makes interpretation difficult for the average programmer. Exceptions in the MAL language layer are denoted by 'MALerror', and query language exceptiosn fall in their own class, e.g. 'SQLerror'. Exceptions can be cascaded to form a trail of exceptions recognized during the exection.

Modules

Modules mk Sat, 04/03/2010 - 10:04

Name Space Management.
Significant speed improvement at type resolution and during the optimization phases can be gained when each module or function identifier is replaced by a fixed length internal identifier. This translation is done once during parsing. Variables are always stored local to the MAL block in which they are used.

The number of module and function names is expected to be limited. Therefore, the namespace manager is organized as a shared global table. The alternative is a namespace per client. However, this would force passing around the client identity or an expensive operation to deduce this from the process id. The price paid is that updates to the namespace should be protected against concurrent access. The current version is protected with locks, which by itself may cause quite some overhead.

The space can, however, also become polluted with identifiers generated on the fly. Compilers are adviced to be conservative in their naming, or explicitly manage the name space by deletion of non-used names once in a while.

The SQL compiler currently pollutes the name space with function names, because it guarantees a global unique name for each query plan for the duration of the server session.

Functions

Functions mk Tue, 03/30/2010 - 11:55

MAL comes with a standard functional abstraction scheme. Functions are represented by MAL instruction lists, enclosed by a function signature and end statement. The function signature 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 user.helloWorld(msg:str):str;
        io.print(msg);
        msg:= "done";
        return msg;
    end user.helloWorld;
```

The module name 'user' designates the collection to which this function belongs. A missing module name is considered a reference to the current module, i.e. the last module or atom context openend. All user defined functions are assembled in the module user by default.

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

Side-effects

Functions can be pre-pended with the keyword `unsafe`, which designates that execution of the function may change the state of the database or sends information to the client. Unsafe functions are critical for the optimizers, because their order of execution should be guaranteed. Functions that return a value of type :void are considered unsafe by default.

Inline functions.

Functions prepended with the keyword inline are a target for the optimizers to be inlined. This is particularly useful when a vectorized version can be used in cases where only scalar function is known.

Polymorphism

Polymorphism mk Tue, 03/30/2010 - 23:40

Polymorphic functions are characterised by type variables 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 it does not exists, a copy of the MAL program is generated, whereafter the type variables are replaced with their concrete types. The new MAL program 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 user.helloWorld(msg:any_1):any_1;
    io.print(msg);
    return user.helloWorld;
end helloWorld;
```

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 typed variables are propagated throughout the function body. This may invalidate type resolutions decisions taken earlier (See MAL Type System).

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.

C-functions

C-functions mk Tue, 03/30/2010 - 23:41

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. Such a polymorphic type list may only be used in the context of a pattern.

```
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(bl:bat[:any_1,:any]...):int
address IOtable;
```

Factories

Factories mk Tue, 03/30/2010 - 11:56

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 *plant* is created in the local system to handle the requests. The plant contains 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 plant 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;
```

Example factories

Example factories mk Tue, 03/30/2010 - 23:46

Complex Factories

The factory scheme can be used 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 consumption 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("relation1");
    Right:= sql.bind("relation2");
    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:= bat.chunkStep(Left);
barrier outer:= lc != nil;
    rc:= bat.chunkStep(Right);
    barrier inner:= rc != nil;
        chunk:= algebra.join(lc,rc);
        yield chunk;
        rc:= bat.chunkStep(Right);
        redo inner:= rc != nil;
    exit inner;
    lc:= bat.chunkStep(Left);
    redo outer:= lc != nil;
exit outer;
    # we have seen everything
    return nil;
end joinStep;

#factory for left branch
factory chunkStep(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 chunkStep(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.

Materialized Views

An area where factories might be useful are support for materialized views, i.e. the result of a query is retained for ease of access. A simple strategy is to prepare the result once and return it on each successive call. Provided the arguments have not been changed. For example:

```
factory view3(l:int, h:int):bat[:oid,:str];
    a:bat[:oid,:int]:= bbp.bind("emp","age");
    b:bat[:oid,:str]:= bbp.bind("emp","name");
barrier always := true;
    l@1d := l;
    h@1d := h;
    s := algebra.select(a,l,h);
    d := algebra.semijoin(b,c);
barrier available := true;
    yield d;
    leave available := calc.!=(l@1d,l);
    leave available := calc.!=(h@1d,h);
    redo available := true;
exit available;
    redo always;
exit always;
end view3;
```

The code should be extended to also check validity of the BATs. It requires a check against the last transaction identifier known.

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?

Ownership

Ownership mk Tue, 03/30/2010 - 23:44

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 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
    bat.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, factories.getCaller(), which returns a client index, factories.getModule() and factories.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:= factories.getCaller();
    yield user.random(seed, clientid);
    redo always;
exit always;
end random;
```

MAL syntax

MAL syntax mk Tue, 03/08/2011 - 16:22

The MAL syntax is summarized below 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 small capitals.

```
program      : (statement ';') *
statement    : moduleStmt [helpinfo] | definition [helpinfo]
             | includeStmt | stmt
moduleStmt   : MODULE ident | ATOM ident [':'ident]
includeStmt  : INCLUDE identifier | INCLUDE string_literal
definition   : [UNSAFE] COMMAND header ADDRESS identifier
             | [UNSAFE] PATTERN header ADDRESS identifier
             | [INLINE | UNSAFE ] FUNCTION header statement* END name
             | FACTORY header statement* END name
helpinfo     : COMMENT string_literal
header       : name '(' params ')' result
name         : [ moduleName '.' ] name
result       : typeName | '(' params ')'
params       : binding [',' binding]*
binding      : identifier typeName
typeName     : scalarType | columnType | ':' any ['_' digit]
scalarType   : ':' identifier
columnType   : ':' BAT '[' ':' colElmType ']'
colElmType   : scalarType | anyType
stmt         : [flow] varlist [':=' expr ]
flow         | RETURN  | BARRIER | CATCH | LEAVE | REDO | RAISE
varlist      : variable | '(' variable [',' variable ] * ')'
variable     : identifier
expr         : fcncall | [factor operator] factor
factor       : literal_constant | NIL | var
fcncall      : moduleName '.' name '(' [args] ')'
args         : factor [',' factor]*
```

MAL interpreter

MAL interpreter mk Tue, 03/30/2010 - 12:03

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

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, FACcall, 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.

The PATcall initiates a call which contains the MAL context, i.e. f(MalBlkPtr mb, MalStkPtr stk, InstrPtr pci) The mb 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.

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.

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. 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.

Exception handling

Exception handling mk Tue, 03/30/2010 - 20:18

Exception handling

Calling a built-in or user-defined routine may lead to an error or a cached status message to be dealt with in MAL. To improve error handling in MAL, an exception handling scheme based on catch-exit blocks. The catch statement identifies a (string-valued) variable, which carries the exception message from the originally failed routine or raise exception assignment. During normal processing catch-exit blocks are simply skipped. Upon receiving an exception status from a function call, we set the exception variable and skip to the first associated catch-exit block. MAL interpretation then continues until it reaches the end of the block. If no exception variable was defined, we should abandon the function alltogether searching for a catch block at a higher layer.

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 then 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 a successfull call, it is a NULL string. Otherwise it contains an encoding of the exceptional state encountered. This message starts with the exception identifer, followed by contextual details.

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: BATs and strings.

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

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

Execution engine

Execution engine mk Thu, 02/20/2014 - 04:02

The execution engine comes in several flavors. The default is a simple, sequential MAL interpreter. For each MAL function call it creates a stack frame, which is initialized with all constants found in the function body. During interpretation the garbage collector ensures freeing of space consumptive tables (BATs) and strings. Furthermore, all temporary structures are garbage collected before the funtion returns the result.

This simple approach leads to an accumulation of temporary variables. They can be freed earlier in the process using an explicit garbage collection command, but the general intend is to leave such decisions to an optimizer or scheduler.
The execution engine is only called when all MAL instructions can be resolved against the available libraries. Most modules are loaded when the server starts using a bootstrap script *mal_init.mal*. Failure to find the startup-file terminates the session. It most likely points to an error in the MonetDB configuration file.

During the boot phase, the global symbol table is initialized with MAL function and factory definitions, and loading the pre-compiled commands and patterns. The libraries are dynamically loaded by default. Expect tens of modules and hundreds of operations to become readily available.
Modules 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 prepared that concurrent threads may be actively changing its structure. Especially, dropping modules may cause severe 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 hindrance.

MAL debugger

MAL debugger mk Tue, 03/30/2010 - 12:12

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.

Note, the MAL debugger described here can be used in conjunction with the textual interface client *mclient* only. The JDBC protocol does not permit passing through information that 'violates' the protocol.

Breakpoints

Breakpoints mk Tue, 03/30/2010 - 12:14

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.]function name, or a MAL line number (#<number>).

The snippet below illustrates the reaction to set a break point on assignment to variable 'i'.

```
mal>mdb.start();
#end main;
mdb>
mal>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).

Debugger features

Debugger features mk Tue, 03/30/2010 - 12:13

To ease debugging and performance monitoring, the MAL interpreter comes with a gdb-like debugger. An illustrative session elicits the functionality offered.

```
mal>function test(i:int):str;
mal>       io.print(i);
mal>       i:= i*2;
mal>       b:= bat.new(:int);
mal>       bat.append(b,i);
mal>       io.print(b);
mal>       return test:= "ok";
mal>end test;
mal>user.test(1);
[ 1 ]
#-----------------#
# h     t         # name
# void   int      # type
#-----------------#
[ 0@0,   2        ]
```

The debugger can be entered at any time using the call mdb.start(). An overview of the available commands is readily available.

```
mal>mdb.start();
#mdb !end main;
mdb>help
   next             -- Advance to next statement
   continue         -- Continue program being debugged
   catch            -- Catch the next exception
   break [<var>]    -- set breakpoint on current instruction or <var>
   delete [<var>]   -- remove break/trace point <var>
   debug <int>      -- set kernel debugging mask
   step             -- advance to next MAL instruction
   module           -- display a module signatures
   atom             -- show atom list
   finish           -- finish current call
   exit             -- terminate executionr
   quit             -- turn off debugging
   list <obj>       -- list current program block
   list # [+#],-#   -- list current program block slice
   List <obj> [#]   -- list with type information[slice]
   list [#] <obj>   -- list program block after optimizer <#>
   List # [+#],-#   -- list current program block slice
   var  <obj>       -- print symbol table for module
   optimizer <obj>  -- display optimizer steps
   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
   trace <var>      -- trace assignment to variables
   trap <mod>.<fcn> -- catch MAL function call in console
   help             -- this message
mdb>
```

The term <obj> is an abbreviation for a MAL operation <mod>.<fcn>, optionally extended with a version number, i.e. [<nr>]. The var denotes a variable in the current stack frame. Debugger commands may be abbreviated.

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.

```
mal>mdb.start();
#end main;
mdb>l
function user.main():void;
   mdb.start();
end user.main;
mdb>L
function user.main():int;       # 0  (main:int)
   mdb.start();        # 1 MDBstart (_1:void)
end user.main;       # 2
```

The user module is the default place for function defined at the console. The modules loaded can be shown typeing the command 'module' (or 'm' for short). The function signatures become visible using the module and optionally the function name.

```
mdb>m alarm
#command alarm.ctime():str address ALARMctime;
#command alarm.epoch():int address ALARMepoch;
```

```
#command alarm.sleep(secs:int):void address ALARMsleep;
#command alarm.time():int address ALARMtime;
#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.

```
mal>user.test(1);
#    user.test(1);
mdb>n
#    io.print(i=1);
mdb>
[ 1 ]
#    i := calc.*(i=1,2:int);
mdb>
#    b=nil:bat[:int] := bat.new(0:int);
mdb>
```

The last instruction shown is next to be executed. The result 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=<tmp_2>[0],i=2);
mdb>
#    io.print(b=<tmp_2>[1]);
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.

Inspection

Inspection mk Tue, 03/30/2010 - 12:16

The debugger commands available for inspection of the program and symbol tables are:

list (List) [<mod>.<fcn>['['<nr>']']]
: A listing of the current MAL block, or one designated by the <mod>.<fcn> is produced. The [<nr>] extension provides access to an element in the MAL block history. The alternative name 'List' also produces the type information.

optimizer [<mod>.<fcn>['['<nr>']']]
: Gives an overview of the optimizer actions in the history of a SQL query. Intermediate results can be accessed using the list command.

atoms
: Lists the atoms currently known

modules [<mod>]
: Lists the modules currently known. An optional <mod> argument produces a list of all signatures within the module identified.

dot <mod>.<fcn>['['<nr>']'] [<file>]
: A dataflow diagram can be produced using the dot command. It expects a function identifier with an optional history index and produces a file for the Linux program dot, which can produce a nice, multi-page graph to illustrate plan complexity.

```
mdb>dot user.main
```

This example produces the user.main.dot in the current working directory. The program call

```
dot -Tpdf user-tst-0.dot -o user-tst-0.pdf
```

creates a PDF file with the graphs. The Linux pdfposter utility can be used to produce a proper printing. Alternatively, the Adobe reader professional can break it up into multiple pages.

Runtime status

Runtime status mk Tue, 03/30/2010 - 12:16

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). The following snippet shows the effect of patching the test case.

```
mal> function test(i:int):str;
mal>    mdb.setTrace(true);
mal>    io.print(i);
mal>    i:= i*2;
mal>    b:= bat.new(:oid,:int);
mal>    bat.insert(b,0@0,i);
mal>    io.print(b);
mal>    mdb.setTrace(false);
mal>    return test:= "ok";
mal> end test;
mal> user.test(1);
#    io.print(i=1);
[ 1 ]
#    i := calc.*(i=1,2);
#    b := bat.new(:oid,:int);
#    bat.insert(b=<tmp_1001>[0],0@0,i=2);
#    io.print(b=<tmp_1001>[1]);
#-----------------#
# h     t         # name
# void  int       # type
#-----------------#
[ 0@0,    2       ]
mal>
```

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().

MAL profiler

MAL profiler mk Tue, 03/30/2010 - 12:18

A key issue in the road towards a high performance application is to understand where resources are being spent. This information can be obtained using different tools and at different levels of abstraction. A coarse-grained insight for a particular application can be obtained using injection of the necessary performance capturing statements in the MAL instruction sequence. Fine-grained, platform specific information can be obtained using existing profilers, like valgrind (http://www.valgrind.org), or hardware performance counters.

MonetDB comes with a set of profiling tools, including ***stethoscope***, ***tomograph*** and ***tachograph***, geared at understanding the internal working, scheduling, and potential resource bottlenecks raised by running queries. They are all based on a profiler event stream started by the system upon request. Together, they facilitate both *online and off-line inspection* of database queries.

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.

SECURITY WARNING

Profiling a system is a potential security leak. Therefore, at any time there can only be one user accessing the profile information.

Trace Format

Trace Format zhang Sun, 07/19/2015 - 18:05

Upon request, a MonetDB server will produce an event stream with performance profiling information.

The event stream consists of individual text lines with comma-separated values and decorative tuple format square brackets ('[' and ']'). The field types are fixed. The bulk of the event records are recordings made at the *"start"* and *"done"* state of a MAL instruction call. The last column *"stmt"* contains the actual call and its arguments. It aids in the analysis to understand why some instructions are taking a lot of time. The prelude of the event record contains:

	Field	Description	Example
1	event	An event counter	38
2	time	The wall-clock time in microseconds	"13:11:16.710180",
3	pc	Name of the query execution plan name; followed by the program counter in the execution plan (denoted in the square brackets '[' and ']') indicating the position of the MAL instruction in its defining block; and finally a unique call identifier to distinguish recursion.	"user.s5_1[14]12",
4	thread	Id of the worker thread processing this MAL instruction.	3
5	state	The instruction status ("start" or "done") *. The "wait" event signals that a worker thread has stumbled upon an empty instruction queue, e.g. all remaining MAL instructions are blocked by an instruction to produce an intermediate result.	"done",
6	usec	The estimated (at "start") or actual execution time (at "done") for the MAL instruction, measured in microseconds	207,
7	rssMB	Memory Resident Set Size (RSS), i.e., the portion of memory occupied by a process that is held in main memory, in MB.	54
8	tmpspace	Estimated cumulative query plan footprint, in MB. For a query, the maximal value of "tmpspace" gives a fairly good estimation of the memory consumption of this query.	0,
9	inblock	The number of disk blocks read **	0,
10	outblock	The number of disk blocks written **	0,
11	majflt	The number of major page faults	0,
12	nswap	The number of swaps	0,
13	switch	The number of context switches	0,
14	stmt	The MAL statement being executed, with argument statistics	"sql.exportResult(X_21=\"104d2\":streams,X_16=4:int);"

* In addition, the *"state"* field denotes two special events. The *"wait"* event is received when the worker threads cannot find an eligible MAL instruction and have to wait for other worker threads to deliver results. The *"ping"* event provides a synopsis of the CPU processing loads.

** Please be aware that these values reflect system wide activities, thus they include not only MonetDB activities. Additionally, these I/O counts do not reflect the actual amount of data read/written from/to the hardware drives, as this information is generally not available to all users of a Linux system.

Stethoscope

Stethoscope mk Tue, 03/30/2010 - 12:20

This program is up for revision in Q1 2020

The program stethoscope is a simple Linux application that can attach itself to a running MonetDB server and extracts the profiler events from concurrent running queries. Stethoscope is an online-only inspection tool, i.e., it only monitors the execution state of the current queries and outputs the information in `STDOUT` for immediate inspection. For example, the following command tracks the microsecond ticks for all database instructions denoted in MAL on a database called "voc":

```
$ stethoscope -u monetdb -P monetdb -d voc
```

A synopsis of the calling conventions:

```
stethoscope [options]
  -d | --dbname=<database_name>
  -u | --user=<user>
  -P | --password=<password>
  -p | --port=<portnr>
  -h | --host=<hostname>
  -o | --output=<local file name>
  -b | --beat=<delay> in milliseconds (default 50)
```

Before stethoscope starts parsing command line options, it first tries to read configuration information from a ".monetdb" file. If the environment variable `DOTMONETDBFILE` is set, it reads the file pointed to by that variable instead. When unset, stethoscope searches for a ".monetdb" file in the current working directory, and if that does not exist, in the current user's home directory. This file can contain defaults for command line keys, e.g., `user`, `password`, `language`, `save_history`, `format` and `width`.

The output produced by stethoscope consists of heartbeat events and MAL instruction traces. The heartbeat is issued at regular intervals and contains as main piece of information the recent CPU load of all cores as made available by the (Linux) operating system. The text below shows an excerpt of the stethoscope output for the query `"SELECT id FROM _tables;"`:

```
$ stethoscope -d voc -b 50000
-- opened UDP profile stream numa-db:50010 for localhost
# event,        time,   pc,     thread, state,  usec,   rssMB,  tmpspace,       inblock,        oublock,        majflt, nswap,  switch, stmt,   # name
...
[ 6,    "13:11:15.060773",      "user.ping[0]0",        4,      "ping", 0,      58,     0,      0,      0,      0,      0,      1,      "[  0.00  0.00  0.00  0.00  0.01  0.01  0.01  0.00  1.00  1.00  0.01  0.01  0.01  1.00  1.00  1.00  1.00  0.01  1.00  0.01  0.00  0.07  0.00  0.00  0.01  0.01  0.00  0.01  0.00  0.00  0.05  0.02  0.00  0.00  0.00  0.04  0.01  0.01  0.01  0.01  0.01  0.01  0.01  0.01  0.00  0.00  0.00  0.01 ]",  ]
...
[ 8,    "13:11:16.703220",      "user.s5_1[0]12",       3,      "start",        0,      54,     0,      0,      0,      0,      0,      484,    "function user.s5_1{autoCommit=true}():void;",       ]
...
[ 10,   "13:11:16.704600",      "user.s5_1[1]12",       3,      "start",        0,      54,     0,      0,      0,      0,      0,      0,      "X_23=0@0:void := querylog.define(\"select id from _tables;\":str,\"default_pipe\":str,16:int);",    ]
[ 11,   "13:11:16.704742",      "user.s5_1[1]12",       3,      "done ",        140,    54,     0,      0,      0,      0,      0,      0,      "X_23=0@0:void := querylog.define(\"select id from _tables;\":str,\"default_pipe\":str,16:int);",    ]
[ 12,   "13:11:16.704881",      "user.s5_1[2]12",       3,      "start",        0,      54,     0,      0,      0,      0,      0,      0,      "barrier X_37=false:bit{transparent} := language.dataflow();",       ]
[ 13,   "13:11:16.705111",      "user.s5_1[3]12",       28,     "start",        0,      54,     0,      0,      0,      0,      0,      2,      "X_2=0:int := sql.mvc();",     ]
[ 14,   "13:11:16.705305",      "user.s5_1[3]12",       28,     "done ",        193,    54,     0,      0,      0,      0,      0,      0,      "X_2=0:int := sql.mvc();",     ]
[ 15,   "13:11:16.705419",      "user.s5_1[4]12",       28,     "start",        0,      54,     0,      0,      0,      0,      0,      0,      "X_3=nil:bat[:oid,:oid] := sql.tid(X_2=0:int,\"sys\":str,\"_tables\":str);", ]
[ 16,   "13:11:16.705468",      "user.s5_1[5]12",       6,      "start",        0,      54,     0,      0,      0,      0,      0,      0,      "X_6=nil:bat[:oid,:int]{file=\"/home/demouser27/tst/./bat/07/736\", rows=45, hlb>=0@0:oid, hub<45@0:oid} := sql.bind(X_2=0:int,\"sys\":str,\"_tables\":str,\"id\":str,0:int);",      ]
[ 17,   "13:11:16.705459",      "user.s5_1[7]12",       22,     "start",        0,      54,     0,      0,      0,      0,      0,      0,      "X_12=nil:bat[:oid,:int]{file=\"/home/demouser27/tst/./bat/07/736\", hlb>=45@0:oid, hub<1048621@0:oid} := sql.bind(X_2=0:int,\"sys\":str,\"_tables\":str,\"id\":str,1:int);",        ]
[ 18,   "13:11:16.705719",      "user.s5_1[5]12",       6,      "done ",        250,    54,     0,      0,      0,      0,      0,      1,      "X_6=[45]:bat[:oid,:int]{file=\"/home/demouser27/tst/./bat/07/736\", rows=45, hlb>=0@0:oid, hub<45@0:oid} := sql.bind(X_2=0:int,\"sys\":str,\"_tables\":str,\"id\":str,0:int);",    ]
[ 19,   "13:11:16.705799",      "user.s5_1[7]12",       22,     "done ",        339,    54,     0,      0,      0,      0,      0,      2,      "X_12=[0]:bat[:oid,:int]{file=\"/home/demouser27/tst/./bat/07/736\", hlb>=45@0:oid, hub<1048621@0:oid} := sql.bind(X_2=0:int,\"sys\":str,\"_tables\":str,\"id\":str,1:int);",      ]
[ 20,   "13:11:16.706113",      "user.s5_1[4]12",       28,     "done ",        693,    54,     0,      0,      0,      0,      0,      5,      "X_3=[43]:bat[:oid,:oid] := sql.tid(X_2=0:int,\"sys\":str,\"_tables\":str);",      ]
[ 21,   "13:11:16.705459",      "user.s5_1[6]12",       12,     "start",        0,      54,     0,      0,      0,      0,      0,      1,      "(X_9=nil:bat[:oid,:oid]{file=\"/home/demouser27/tst/./bat/07/736\", rows=45, hlb>=0@0:oid, hub<45@0:oid},r1_9=nil:bat[:oid,:int]) := sql.bind(X_2=0:int,\"sys\":str,\"_tables\":str,\"id\":str,2:int);",    ]
[ 22,   "13:11:16.706800",      "user.s5_1[6]12",       12,     "done ",        1339,   54,     0,      0,      0,      0,      0,      2,      "(X_9=[0]:bat[:oid,:oid]{file=\"/home/demouser27/tst/./bat/07/736\", rows=45, hlb>=0@0:oid, hub<45@0:oid},r1_9=[0]:bat[:oid,:int]) := sql.bind(X_2=0:int,\"sys\":str,\"_tables\":str,\"id\":str,2:int);",      ]
[ 23,   "13:11:16.706948",      "user.wait[0]0",        6,      "wait", 0,      54,     0,      0,      0,      0,      0,      3,      "[  0.01  0.00  0.00  0.01  0.01  0.01  0.01  0.01  0.01  0.01  0.01  0.01  0.01  0.01  0.01  0.01  0.01  0.01  0.01  0.00  0.01  0.00  0.00  0.01  0.01  0.01  0.00  0.01  0.00  0.00  0.01  0.01  0.00  0.00  0.00  0.00  0.01  0.01  0.01  0.01  0.01  0.01  0.01  0.01  0.01  0.01  0.01  0.01 ]",      ]
...
[ 25,   "13:11:16.708063",      "user.s5_1[8]12",       45,     "start",        0,      54,     0,      0,      0,      0,      0,      0,      "X_14=nil:bat[:oid,:int] := sql.delta(X_6=[45]:bat[:oid,:int],X_9=[0]:bat[:oid,:oid],r1_9=[0]:bat[:oid,:int],X_12=[0]:bat[:oid,:int]);",      ]
[ 26,   "13:11:16.708329",      "user.s5_1[8]12",       45,     "done ",        265,    54,     0,      0,      0,      0,      0,      1,      "X_14=[45]:bat[:oid,:int] := sql.delta(X_6=[45]:bat[:oid,:int],X_9=[0]:bat[:oid,:oid],r1_9=[0]:bat[:oid,:int],X_12=[0]:bat[:oid,:int]);",    ]
[ 27,   "13:11:16.708445",      "user.s5_1[9]12",       45,     "start",        0,      54,     0,      0,      0,      0,      0,      0,      "X_15=nil:bat[:oid,:int] := algebra.leftfetchjoin(X_3=[43]:bat[:oid,:oid],X_14=[45]:bat[:oid,:int]);",    ]
[ 28,   "13:11:16.708561",      "user.s5_1[9]12",       45,     "done ",        116,    54,     0,      0,      0,      0,      0,      1,      "X_15=[43]:bat[:oid,:int] := algebra.leftfetchjoin(X_3=[43]:bat[:oid,:oid],X_14=[45]:bat[:oid,:int]);",  ]
...
[ 30,   "13:11:16.708994",      "user.s5_1[2]12",       3,      "done ",        4112,   54,     0,      0,      0,      0,      0,      3,      "barrier X_37=false:bit{transparent} := language.dataflow();",       ]
[ 31,   "13:11:16.709419",      "user.s5_1[11]12",      3,      "start",        0,      54,     0,      0,      0,      0,      0,      2,      "X_16=0:int := sql.resultSet(1:int,1:int,X_15=[43]:bat[:oid,:int]);",       ]
[ 32,   "13:11:16.709529",      "user.s5_1[11]12",      3,      "done ",        109,    54,     0,      0,      0,      0,      0,      0,      "X_16=4:int := sql.resultSet(1:int,1:int,X_15=[43]:bat[:oid,:int]);",       ]
[ 33,   "13:11:16.709617",      "user.s5_1[12]12",      3,      "start",        0,      54,     0,      0,      0,      0,      0,      0,      "sql.rsColumn(X_16=4:int,\"sys._tables\":str,\"id\":str,\"int\":str,32:int,0:int,X_15=[43]:bat[:oid,:int]);",       ]
[ 34,   "13:11:16.709717",      "user.s5_1[12]12",      3,      "done ",        100,    54,     0,      0,      0,      0,      0,      0,      "sql.rsColumn(X_16=4:int,\"sys._tables\":str,\"id\":str,\"int\":str,32:int,0:int,X_15=[43]:bat[:oid,:int]);",       ]
[ 35,   "13:11:16.709816",      "user.s5_1[13]12",      3,      "start",        0,      54,     0,      0,      0,      0,      0,      0,      "X_21=nil:streams := io.stdout();", ]
[ 36,   "13:11:16.709896",      "user.s5_1[13]12",      3,      "done ",        79,     54,     0,      0,      0,      0,      0,      0,      "X_21=\"104d2\":streams := io.stdout();",    ]
[ 37,   "13:11:16.709973",      "user.s5_1[14]12",      3,      "start",        0,      54,     0,      0,      0,      0,      0,      0,      "sql.exportResult(X_21=\"104d2\":streams,X_16=4:int);",      ]
[ 38,   "13:11:16.710180",      "user.s5_1[14]12",      3,      "done ",        207,    54,     0,      0,      0,      0,      0,      0,      "sql.exportResult(X_21=\"104d2\":streams,X_16=4:int);",      ]
[ 39,   "13:11:16.710264",      "user.s5_1[15]12",      3,      "start",        0,      54,     0,      0,      0,      0,      0,      0,      "end user.s5_1;",      ]
[ 40,   "13:11:16.710325",      "user.s5_1[15]12",      3,      "done ",        61,     54,     0,      0,      0,      0,      0,      0,      "end user.s5_1;",      ]
...
[ 43,   "13:11:16.711494",      "user.s5_1[0]12",       3,      "done ",        8273,   54,     0,      0,      0,      0,      0,      0,      "function user.s5_1{autoCommit=true}():void;",       ]
...
^Csignal 2 received
```

Tomograph

Tomograph mk Fri, 08/31/2012 - 13:03

The program ***tomograph*** uses the information in the event stream to illustrate the parallel processing behavior of individual queries. The information can be used to identify possible expensive and blocking portions in a query. It helps the developers to understand the actual parallel behavior and may spot areas for improvement. Tomograph is an *off-line only inspection* tool and it is only available on Linux platforms.

The tomograph is depricated as of Q1 2020.

Assume a MonetDB server is running to serve the database "voc". First, in terminal 1 start tomograph on the database as the following:

```
$ tomograph -d voc
-- Output directed towards cache/voc_*
-- Stop capturing with or after 32 pages
```

Then, in terminal 2, start a mclient to execute a query:

```
$ mclient -d voc -s "select count(*) from tables;"
```

This triggers tomograph to respond with something like below in terminal 1:

```
-- page 0 :set time zone interval '+02:00' hour to minute
-- page 1 :select count(*) from tables
```

This indicates that tomograph has captured two queries. The first query "`set time zone...`" is executed automatically once at the start of each `mclient` to set the time zone for this client. For each captured query, tomograph will generate one page with its execution information. The tomograph output of the set time zone query can be safely ignored.

After our real SQL query has finished, go to terminal 1 and press `Ctrl-c` to terminate tomograph, which will try to plot the execution information of each captured query into a separated PDF file, and glue the individual PDF files into one file. This requires the tools `gnuplot` and `gs`. If this succeeds, tomograph will terminate with a message like below:

```
^Csignal 2 received
-- exec:gnuplot cache/voc_00.gpl;
-- exec:gs -q -dNOPAUSE -sDEVICE=pdfwrite -sOUTPUTFILE=cache/voc.pdf -dBATCH cache/voc_00.pdf
-- done: cache/voc.pdf
```

Otherwise, tomograph will terminate with an error message and instructions to manually generate the plots and glue the PDF files, like below:

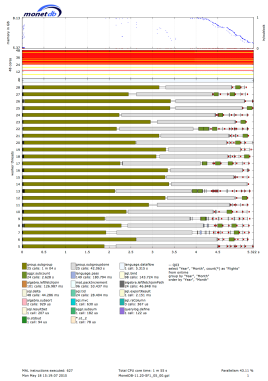
```
"Csignal 2 received
-- exec:gnuplot cache/voc_00.gpl;
<error message>

To finish the atlas make sure gnuplot is available and run:
gnuplot cache/voc_00.gpl
gnuplot cache/voc_01.gpl
gs -dBATCH -dNOPAUSE -sDEVICE=pdfwrite -sOUTPUTFILE=cache/voc.pdf -dBATCH cache/voc_00.pdf cache/voc_01.pdf
```

For each captured query, tomograph produces three files:

```
$ ls cache
voc_00.dat  voc_00.gpl  voc_00.trace  voc_01.dat  voc_01.gpl  voc_01.trace
```

The *".trace"* files contain the event stream with raw query execution information. The *".dat"* and *".gpl"* files are generated by tomograph by extracting information from the *".trace"* files. These files can be regenerated by passing the corresponding *".trace"* file to the *"--input"* of tomograph. The figure below shows an example tomograph graph for one query:



In a tomograph graph, the top most part ("memory in GB" and "in/outblock") illustrates the memory RSS (shown as blue dots) and I/O activities (reads as grey dots, writes as red dots) as reported by the OS during each heartbeat event. Note that all values displayed here reflect system wide activities, thus they include not only MonetDB activities. Also note that the I/O counts shown here do not reflect the actual amount of data read/written to the hardware drives, as this information is generally not available to all users of a Linux system.

The second part ("cores") shows a heat map of the available CPU cores in the system, one line per CPU. This information is gathered from a heartbeat monitor in the server. The arrival rate can be controlled using the "--beat" option. The colours show the level of core utilization (white: <2%, yellow: >2%, gold: >25%, orange: >50%, orangered: >75%, red: >90%). Please be aware that the information given here includes the activities of the whole system, instead of only that of MonetDB.

The third part ("worker threads") shows the activities of all MonetDB worker threads. Therefore, this part contains the most interesting information to find the bottlenecks in query performance. Note that the activities of the software threads here are not *directly* linked to the CPU activites shown in the second part, as tomograph does not give any information about which core is executing which thread (although some deductions can be made). Along the time axes, we show for each software thread the MAL instructions in progress. A hollow box with smaller height indicates that the thread was waiting for the next instruction to become available for execution. The little red-filled boxes, often placed at the end of a thread, indicate there is no instruction left at that moment.

Finally, at the bottom of a tomograph, we provide detailed statistics on the most frequently executed MAL instructions. Among others, the "parallelism" is calculated by: (sum of the maximum worker running time) × (numbers of cores) ÷ (reported number of ticks spent on each MAL instruction).

A synopsis of the calling conventions:

```
tomograph [options]
  -d | --dbname=<database_name>
  -u | --user=<user>
  -P | --password=<password>
  -p | --port=<portnr>
  -h | --host=<hostname>
  -T | --title=<plot title>
  -r | --range=<starttime>-<endtime>[ms,s]
  -i | --input=<profiler event file>
  -o | --output=<dir/file prefix > (default 'cache/<dbname>')
  -b | --beat=<delay> in milliseconds (default 5000)
  -A | --atlas=<number> maximum number of queries (default 1)
  -D | --debug
  -? | --help
```

The tomograph trace files can be also used for *off-line inspection* of specific time ranges, without having to contact the MonetDB server and re-run a query. Most notably is to zoom into a range of the execution trace using the option *"--range"*. The ranges may be expressed in seconds (*s*), milliseconds (*ms*), or microseconds (default). The trace file can also be used to locate the details of each instruction.

Tachograph

Tachograph zhang Sun, 07/19/2015 - 21:30

This program is deprecated as of Q1 2020

The program ***tachograph*** monitors the event stream to determine the progress of a query execution.

A synopsis of the calling conventions:

```
tachograph [options]
  -d | --dbname=<database_name>
  -u | --user=<user>
  -p | --port=<portnr>
  -h | --host=<hostname>
  -b | --beat=<delay> in milliseconds (default 5000)
  -i | --interactive=0 | 1> show trace on stdout (default 1)
  -o | --output=<webfile>
  -c | --cache=<query pool location> (default 'cache')
  -q | --queries=<query pool capacity>
  -w | --wait=<delay time> in milliseconds (default 5000)
  -? | --help
```

Assume a MonetDB server is running to serve the database "voc". First, in terminal 1 start tachograph on the database as the following:

```
$ tachograph -d voc
-- opened UDP profile stream mama-db:50810 for localhost
```

Then, in terminal 2, start an `mclient` to execute a (long-running) query:

```
$ mclient -d voc -s "select * from tables a, tables b, tables c, tables d, tables e;"
```

This triggers tachograph to respond with something like below in terminal 1:

```
CACHE ID:0
set time zone interval '+02:00' hour to minute

[##################################################] 100%  00:00:00
CACHE ID:1
select * from tables a, tables b, tables c, tables d, tables e
[#########################..........] 68%  00:00:20     language.dataflow()
```

This indicates that tachograph has captured two queries. The first query "`set time zone...`" is executed automatically once at the start of each `mclient` to set the time zone for this client. For each captured query, tachograph shows on the command line:

1. progress bar,
2. percentage of completion,
3. elapsed time, and
4. MAL instruction currently being executed (if any).

The progress bar and percentage of completion are computed using the ETC (Expected Time to Completion) of the MAL instructions. To terminate tachograph at any time, press `Ctrl-c`. For each query, tachograph saves the execution information in four files (by default in the directory "cache"):

```
$ ls cache
tachograph_0.json     tachograph_0.trace    tachograph_1_stmt.csv
tachograph_0_mal.csv  tachograph_1.json     tachograph_1.trace
tachograph_0_stmt.csv  tachograph_1_mal.csv  tachograph.trace
```

The events streams are saved in the *".trace"* files. The *".csv"* files contain more readable information of the MAL statements extracted from the *".trace"* files stored in CSV format. The *".csv"* files can be easily loaded into a database for in-database analysis of the query execution, which are currently not used yet.

The most interesting output files here are the *".json"* files, which contain the same information as the *".trace"* files, but converted into JSON format. In addition, each *"stmt"* is accompanied by a *"beautystmt"*, in which the variables in the MAL statement are mapped back to tables and columns. This makes the event streams much more human readable. The table below shows an excerpt of the *".json"* file generated for the query `"select count(*) from tables;"`:

```
{ "tachograph":0.1,
  "system":{ MonetDBversion:"11.20.0", release:"unreleased", host:"x86_64-unknown-linux-gnu", threads:"48", memory:"93.703 GB", oid:"64", packages:["huge"]},   ],
  "qid":"user_s1_2{autoCommit=true}",
  "tag":"133",
  "query":"select count(*) from tables",
  "started": 8580532413,
  "duration":0,
  "instructions":34
},
{
"qid":"user_s1_2{autoCommit=true}",
"tag":133,
"pc":1,
"time": 1160,
"status": "start",
"estimate": 0,
"stmt": "X_49=0@0:void := querylog.define(\"select count(*) from tables;\":str,\"default_pipe\":str,34:int)",
"beautystmt": "querylog.define(\"select count(*) from tables;\",\"default_pipe\",34)",
"prereq":[]
},
---
{
"qid":"user_s1_2{autoCommit=true}",
"tag":133,
"pc":27,
"time": 5139,
"status": "start",
"estimate": 0,
"stmt": "X_38=0:ard := aggr.count(X_37=tmp_710:bat[:oid,:int]);",
"beautystmt": "count(tmp._tables.id[6])",
"prereq":[, 26]
},
---
```

Each *".json"* file starts with a special block containing meta-information, e.g., hardware resources and software versions. Then it contains one block for each line of event in the *".trace"* file.

MAL optimizers

MAL optimizers mk Tue, 03/30/2010 - 12:10

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. 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.

Experience shows that construction and debugging of a front-end specific optimizer is simplified when you retain information on the origin of the MAL code produced as long as possible. For example, the snippet `sql.insert(col, 1208, "hello")` can be the target of simple SQL rewrites using the module name as the discriminator.

Pipeline validation. The pipelines used to optimize MAL programs contain dependencies. For example, it does not make much sense to call the deadcode optimizer too early in the pipeline, although it is not an error. Moreover, some optimizers are merely examples of the direction to take, others are critical for proper functioning for e.g. SQL.

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

Landscape

Landscape mk Tue, 03/30/2010 - 12:06

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.

Constant expressions 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(R.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 re-use, 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 alias removal. Impact: relevant for stored queries (MAL functions) Prereq: alias removal, common terms .

Alias Removal 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 Term Optimizer 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: Alias Removal

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

Join Path Optimizer 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 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 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:

Range Propagation Goal: look for constant ranges in select statements and propagate them through the code. Rationale: partitioned tables and views may give rise to expressions that contain multiple selections over the same BAT. If their arguments are constant, the result of such selects can sometimes be predicted, or the multiple selections can be cascaded into a single operation. Impact: high, should be followed by alias removal and dead code removal

Result Cacher 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 quer optimizers. Impact: high

Staged Execution 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 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 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 (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 (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 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

Demand Driven Interpreter (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 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 Evaluation 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 common term optimizer to avoid clashes. Status: Used in the SQL optimizer.

Code Inliner 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. Impact: medium Status: Used in the SQL optimizer to handle SQL functions.

Code Outliner 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 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 garqabe collection call at the end of the BAT's lifespan. Impact: large Status: Implemented. Algorithm based on end-of-life-span analysis.

Foreign Key replacements 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 using a join index. Impact: large Status: Implemented in the SQL strategic optimizer.

Building blocks

Building blocks mk Tue, 03/30/2010 - 12:07

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.

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.

Building Your Own Optimizer

Implementation of your own MAL-MAL optimizer can best be started from refinement of one of the examples included in the code base. Beware that only those used in the critical path of SQL execution are thoroughly tested. The others are developed up to the point that the concept and approach can be demonstrated.

The general structure of most optimizers is to actively copy a MAL block into a new program structure. At each step we determine the action taken, e.g. replace the instruction or inject instructions to achieve the desired goal.

A tally on major events should be retained, because it gives valuable insight in the effectiveness of your optimizer. The effects of all optimizers is collected in a system catalog.

Each optimizer ends with a strong defense line, `optimizerCheck()` It performs a complete type and data flow analysis before returning. Moreover, if you are in debug mode, it will keep a copy of the plan produced for inspection. Studying the differences between optimizer steps provide valuable information to improve your code.

The functionality of the optimizer should be clearly delineated. The guiding policy is that it is always safe to not apply an optimizer step. This helps to keep the optimizers as independent as possible.

It really helps if you start with a few tiny examples to test your optimizer. They should be added to the Tests directory and administered in Tests/All.

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

One of the major hurdles is to test interference of the optimizer. The test set is a good starting point, but does not garantee that all cases have been covered.

In principle, any subset of optimizers should work flawlessly. With a few tens of optimizers this amounts to potential millions of runs. Adherence to a partial order reduces the problem, but still is likely to be too resource consumptive to test continously.

Lifespans

Lifespans mk Tue, 03/30/2010 - 12:09

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 a MAL instruction calls for either a recalculation or further propagation. Unclear what will be the best strategy. For the time being we just recalc.

See is all arguments mentioned in the instruction at point pc are still visible at instruction qc and have not been updated in the mean time. Take into account that variables may be declared inside a block. This can be calculated using the BARRIER/CATCH and EXIT pairs.

The safety property should be relatively easy to determine for each MAL function. 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

```
    G1 := f(A,B,C);
    ...
    G1 := g(D,E,F);
    ...
    I2:= f(A,B,C);
    ...
    I2:= 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 side effects).

Alias removal

Alias removal mk Tue, 03/30/2010 - 16:01

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:=0;
        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");
```

A special case is backward propagation of constants. The following snippet is the result of the JITO emptyset. It can be further reduced to avoid useless assignments.

```
    _53 := sql.bind("sys","_tables","type",0);
    (_54,_56,_58,_60) := bat.partition(_53);
    _53 := nil;
    _56 := nil;
    _54 := nil;
    _75 := _67;
    _67 := nil;
    _83 := _75;
    _75 := nil;
```

Coercions

Coercions mk Tue, 03/30/2010 - 16:03

A simple optimizer that removes coercions that are not needed. They may result from a sloppy code-generator or function call resolution decision. For example:

```
        v:= calc.int(23);
```

becomes a single assignment without function call.

The primary role is a small illustration of coding an optimizer algorithm.

Common subexpressions

Common subexpressions mk Tue, 03/30/2010 - 16:03

5.2.6 Common Subexpression Elimination

Common subexpression elimination merely involves a 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.commonTerms() 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 non-empty barrier block.

```
    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.commonTerms();
```

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;
    l := calc.+(24,27);
    l3 := calc.+(l,24);
```

Constant expressions

Constant expressions mk Tue, 03/30/2010 - 16:04

5.2.7 Constant Expression Evaluation

Expressions produced by compilers involving only constant arguments can be evaluated once. It is particular relevant in functions that are repeatably called. One time queries would not benefit from this extra step.

Consider the following snippet, which contains recursive use of constant arguments

```
    a:= 1+1;        io.print(a);
    b:= 2;          io.print(b);
    c:= 3*b;        io.print(c);
    d:= calc.flt(c);io.print(d);
    e:= mmath.sin(d);io.print(e);
    optimizer.aliasRemoval();
    optimizer.evaluate();
```

The code produced by the optimizer would be

```
    io.print(2);
    io.print(2);
    io.print(6);
    io.print(6.);
    io.print(-0.279415498);
```

Likewise we attempt to catch barrier blocks based on constants.

Cost model

Cost model mk Tue, 03/30/2010 - 16:05

5.2.8 Costmodel Approach

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 an example for building your own cost-based optimized.

The optimizer.costModel() works its way through a MAL program in search for relational operators and estimates their result size. The estimated size is left behind as the property rows.

```
    r{rows=100} := bat.new(:oid,:int);
    s{rows=1000} := bat.new(:oid,:int);
    rs:= algebra.select(s,1,1);
    rr:= bat.reverse(r);
    j:= algebra.join(rs,rr);
    optimizer.costModel();
```

changes the properties of the instructions as follows:

```
    r{rows=100} := bat.new(:oid,:int);
    s{rows=1000} := bat.new(:oid,:int);
    rs{rows=501} := algebra.select(s,1,1);
    rr{rows=100} := bat.reverse(r);
    j{rows=100} := algebra.join(rs,rr);
```

The cost estimation does not use any statistics on the actual data distribution yet. It relies on the rows property provided by the front-end or other optimizers. It just applies a few heuristic cost estimators. However, it ensures that empty results are only tagged with rows=0 if the estimate is accurate, otherwise it assumes at least one result row. This property makes it possible to safely pass the result of the cost estimation to the emptySet optimizer for code reduction.

Data flow

Data flow mk Tue, 03/30/2010 - 16:06

5.2.9 The dataflow optimizer

MAL programs are largely logical descriptions of an execution plan. At least as it concerns side-effect free operations. For these sub-plans the order of execution needs not to be a priori fixed and a dataflow driven evaluation is possible. Even using multiple worker threads to work their way through the dataflow graph.

The dataflow optimizer analyses the code and wraps all instructions eligible for dataflow driven execution with a guarded block:

```
BARRIER X_12:= language.dataflow();
   .... side-effect-free MAL calls ...
```

EXIT X_12;

Ofcourse, this is only necessary if you can upfront determine there are multiple threads of execution possible.

Upon execution, the interpreter instantiates multiple threads based on an the number of processor cores available. Subsequently, the eligible instructions are queued and consumed by the interpreter worker threads. A worker thread tries to continue processing with an instruction that needs the latest intermediate result produced. This improves locality of access and shortens the lifetime of temporary structures.

Dataflow blocks may not be nested. Therefore, any dataflow block produced for inlined code is removed first.

Dead code

Dead code mk Tue, 03/30/2010 - 16:07

5.2.10 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(:oid,:int);
V10 := bat.new(:int,:oid);
V16 := algebra.markH(V7);
V17 := algebra.join(V16,V7);
V19 := bat.new(:oid,:int);
V22 := bat.new(:oid,: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");
```

A refinement of the dead code comes from using arguments that ceased to exist due to actions taken by an optimizer. For example, in the snippet below the pushranges optimizer may conclude that variable V31 becomes empty and simply injects a 'dead' variable by dropping the assignment statement. This makes other code dead as well.

```
V30 := algebra.select( V7, 10,100);
V31 := algebra.select(V30,-1,5);
V32 := aggr.sum(V31);
io.print(V32);
```

[implementation pending]

Garbage collector

Garbage collector mk Tue, 03/30/2010 - 16:08

5.2.13 Garbage Collection

Garbage collection of temporary variables, such as strings and BATs, takes place upon returning from a function call.

Join paths

Join paths mk Tue, 03/30/2010 - 16:10

5.2.15 Join Paths

The routine optimizer.joinPath() walks through the program looking for join operations and cascades them into multiple join paths. To illustrate, consider

```
a:= bat.new(:oid,:oid);
b:= bat.new(:oid,:oid);
c:= bat.new(:oid,:str);
j1:= algebra.join(a,b);
j2:= algebra.join(j1,c);
j3:= algebra.join(b,b);
j4:= algebra.join(b,j3);
```

The optimizer will first replace all arguments by their join sequence. The underlying instructions are left behind for the deadcode optimizer to be removed.

```
a:= bat.new(:oid,:oid);
j1:= algebra.join(a,b);
j2:= algebra.joinPath(a,b,c);
j3:= algebra.join(b,b);
j4:= algebra.joinPath(b,b,b);
```

The same operation is applied to sequences of leftjoin operations.

Any join result used multiple times should be calculated once. A list of inlined paths is kept around to detect those.

In principle, the joinpaths may contain common subpaths, whose materialization would improve performance.

The SQL front-end produces often produces snippets of the following structure

```
t1:= algebra.join(b,c);
z1:= algebra.join(a,t1);
    ...
t2:= algebra.join(b,d);
z2:= algebra.join(a,t2);
```

The joinpath would merge them into

```
z1:= algebra.joinPath(a,b,c);
    ...
z2:= algebra.joinPath(a,b,d);
```

which are handle by a heuristic looking at the first two argments and re-uses a materialized join.

```
_13:= algebra.join(a,b);
z1:= algebra.join(_13,c);
    ...
z2:= algebra.join(_13,d);
```

An alternative is to make recognition of any common re-useable paths an integral part of the joinPath body.

```
x3:= algebra.join(a,b);
    r3:= bat.reverse(x3);
    j1:= join(c,r3);

    rb:= bat.reverse(b);
    ra:= bat.reverse(a);
    j1:= algebra.joinpath(c,rb,ra);
```

As a final step in the speed up of the joinpath we consider clustering large operands if that is expected to improve IO behavior.

Macro processing

Macro processing mk Tue, 03/30/2010 - 16:10

5.2.16 Macro and Orcam Processing

The optimizers form the basis for replacing code fragments. Two optimizers are focused on code expansion and contraction. The former involves replacing individual instructions by a block of MAL code, i.e. a macro call. The latter depicts the inverse operation, a group of instructions is replaced by a single MAL assignment statement, i.e. a orcam call.

The macro facility is limited to type-correct MAL functions, which means that replacement is not essential from a semantic point of view. They could have been called, or the block need not be compressed. It is equivalent to inline code expansion.

The macro and orcam transformations provide a basis to develop front-end specific code generation templates. The prototypical test case is the following template:

```
function user.joinPath( a:bat[:any_1,:any_2],
                b:bat[:any_2,:any_3],
                c:bat[:any_3,:any_4]):bat[:any_1,:any_4]
address fastjoinpath;
    z:= join(a,b);
    zz:= join(z,c);
    return zz;
end user.joinPath;
```

The call optimizer.macro("user", "joinPath") hunts for occurrences of the instruction call in the block in which it is called and replaces it with the body, i.e. it in-lines the code. Conversely, the optimizer.orcam("user", "joinPath") attempts to localize a block of two join operations and, when found, it is replaced by the direct call to joinPath. In turn, type resolution then directs execution to a built-in function fastjoinpath.

The current implementation is limited to finding a consecutive sequence, ending in a return-statement. The latter is needed to properly embed the result in the enclosed environment. It may be extended in the future to consider the flow of control as well.

5.2.17 Known issues

The functions subject to expansion or contraction should be checked on 'proper' behavior.

The current implementation is extremely limited. The macro optimizer does not recognize use of intermediate results outside the block being contracted. This should be checked and it should block the replacement, unless the intermediates are part of the return list. Likewise, we assume here that the block has a single return statement, which is also the last one to be executed.

The macro optimizer can only expand functions. Factories already carry a significant complex flow of control that is hard to simulate in the nested flow structure of an arbitrary function.

The orcam optimizer can not deal with calls controlled by a barrier. It would often require a rewrite of several other statements as well.

```
pattern optimizer.macro(targetmod:str,targetfcn:str):void
address OPTmacro
comment "Inline the code of the target function.";
pattern optimizer.macro(mod:str,fcn:str,targetmod:str,targetfcn:str):void
address OPTmacro
comment "Inline a target function used in a specific function.";

pattern optimizer.orcam(targetmod:str,targetfcn:str):void
address OPTorcam
comment "Inverse macro processor for current function";
pattern optimizer.orcam(mod:str,fcn:str,targetmod:str,targetfcn:str):void
address OPTorcam
comment "Inverse macro, find pattern and replace with a function call.";
```

Memoization

Memoization mk Tue, 03/30/2010 - 16:11

5.3 Memo-based Query Execution

Modern cost-based query optimizers use a memo structure to organize the search space for an efficient query execution plan. For example, consider an oid join path 'A.B.C.D'. We can start the evaluation at any point in this path.

Its memo structure can be represented by a (large) MAL program. The memo levels are encapsulated with a `choice` operator. The arguments of the second dictate which instructions to consider for cost evaluation.

```
...
scheduler.choice("getVolume");
T1:= algebra.join(A,B);
T2:= algebra.join(B,C);
T3:= algebra.join(C,D);
scheduler.choice("getVolume",T1,T2,T3);
T4:= algebra.join(T1,C);
T5:= algebra.join(A,T2);
T6:= algebra.join(T2,D);
T7:= algebra.join(B,T3);
T8:= algebra.join(C,D);
scheduler.choice("getVolume",T4,T5,T6,T7,T8);
T9:= algebra.join(T4,D);
T10:= algebra.join(T5,D);
T11:= algebra.join(A,T6);
T12:= algebra.join(A,T7);
T13:= algebra.join(T1,T8);
scheduler.choice("getVolume",T9,T10,T11,T12,T13);
answer:= scheduler.pick(T9, T10, T11, T12, T13);
```

The `scheduler.choice()` operator calls a builtin `getVolume` for each target variable and expects an integer-valued cost. In this case it returns the total number of bytes uses as arguments.

The target variable with the lowest cost is chosen for execution and remaining variables are turned into a temporary NOOP operation.(You may want to re-use the memo) They are skipped by the interpreter, but also in subsequent calls to the scheduler. It reduces the alternatives as we proceed in the plan.

A built-in naive cost function is used. It would be nice if the user could provide a private cost function defined as a `pattern` with a polymorphic argument for the target and a `:lng` result. Its implementation can use the complete context information to make a decision. For example, it can trace the potential use of the target variable in subsequent statements to determine a total cost when this step is taken towards the final result.

A complete plan likely includes other expressions to prepare or use the target variables before reaching the next choice point. It is the task of the choice operator to avoid any superfluous operation.

The MAL block should be privately owned by the caller, which can be assured with `scheduler.isolation()`.

A refinement of the scheme is to make cost analysis part of the plan as well. Then you don't have to include a hardwired cost function.

```
Acost:= aggr.count(A);
Bcost:= aggr.count(B);
Ccost:= aggr.count(C);
T1cost:= Acost+Bcost;
T2cost:= Bcost+Ccost;
T3cost:= Ccost+Dcost;
scheduler.choice(T1cost,T1, T2cost,T2, T3cost,T3);
T1:= algebra.join(A,B);
T2:= algebra.join(B,C);
T3:= algebra.join(C,D);
...
```

Multiplex functions

Multiplex functions mk Tue, 03/30/2010 - 16:14

5.3.2 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 positional 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]:= mal.multiplex("calc.+",b,1);
        optimizer.multiplex();
```

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

Remote actions

Remote actions mk Tue, 03/30/2010 - 16:19

Remote Queries

MAL variables may live at a different site from where they are used. In particular, the SQL front-end uses portions of remote BATs as replication views. Each time such a view is needed, the corresponding BAT is fetched and added to the local cache.

Consider the following snippet produced by a query compiler,

```
mid:= mapi.reconnect("s0_0","localhost",50000,"monetdb","monetdb","mal");
b:bat[:oid,:int] := mapi.bind(mid,"rvar");
c:=algebra.select(b,0,12);
io.print(c);
d:=algebra.select(b,5,10);
low:= 5+1;
e:=algebra.select(d,low,7);
i:=aggr.count(e);
```

```
io.printf(" count %d\n",i);
io.print(d);
```

which uses a BAT

rvar

stored at the remote site

db1

.

There are several options to execute this query. The remote BAT can be fetched as soon as the bind operation is executed, or a portion can be fetched after a remote select, or the output for the user could be retrieved. An optimal solution depends on the actual resources available at both ends and the time to ship the BAT.

The remote query optimizer assumes that the remote site has sufficient resources to handle the instructions. For each remote query it creates a private connection. It is re-used in subsequent calls .

The remote environment is used to execute the statements. The objects are retrieved just before they are locally needed.

```
mid:= mapi.reconnect("s0_0","localhost",50000,"monetdb","monetdb","mal");
mapi.rpc(mid,"b:bat[:oid,:int] :=bbp.bind(\"rvar\");");
mapi.rpc(mid,"c:=algebra.select(b,0,12);");
c:bat[:oid,:int]:= mapi.rpc(mid, "io.print(c);");
io.print(c);
mapi.rpc(mid,"d:=algebra.select(b,5,10);");
low:= 5+1;
mapi.put(mid,"low",low);
mapi.rpc(mid,"e:=algebra.select(d,low,7);");
mapi.rpc(mid,"i:=aggr.count(d);");
i:= mapi.rpc(mid,"io.print(i);");
io.printf(" count %d\n",i);
io.print(d);
```

To reduce the number of interprocess communications this code can be further improved by glueing the instructions together when until the first result is needed.

Stack reduction

Stack reduction mk Tue, 03/30/2010 - 16:20

Stack Reduction

The compilers producing MAL may generate an abundance of temporary variables to hold the result of expressions. This leads to a pollution of the run-time 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%.

MAL modules

MAL modules mk Tue, 03/30/2010 - 12:21

The MAL module signatures are collected in individual files named .../lib(64)/<modulename>.mal. The server is bootstrapped by processing a MAL script, called .../lib/MonetDB5/mal_init.mal, with the names of the modules to be loaded. For each module file encountered, the object library lib_<modulename>.so is searched for in the location identified by monet_mod_path=exec_prefix/lib/MonetDB5:exec_prefix/lib/MonetDB5/lib:exec_prefix/lib/MonetDB5/bin. 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_PATH, 2) the libraries specified in /etc/ld.so.cache and 3) /usr/lib followed by /lib.

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

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

Alarm

Alarm mk Tue, 03/30/2010 - 16:42

Timers and Timed Interrupts

This module handles various signaling/timer functionalities. The parameterless routines, *usec*, *time* and *ctime* provide access to the CPU clock. They return an integer and string, respectively. The functions are considered unsafe, because they affect the timing experienced by the user.

```
command alarm.sleep{unsafe}(secs:int):void
address ALARMsleep
comment "Sleep a few seconds";

command alarm.usec{unsafe}() :lng
address ALARMusec
comment "Return time in microseconds.";

command alarm.time{unsafe}() :int
address ALARMtime
comment "Return time in milliseconds.";

command alarm.epoch{unsafe}() :int
address ALARMepoch
comment "Return the current time as UNIX epoch.";

command alarm.ctime{unsafe}() :str
address ALARMctime
comment "Return the current time as a C-time string.";

command alarm.prelude():void
address ALARMprelude
comment "Initialize alarm module."
```

Algebra

Algebra mk Tue, 03/30/2010 - 16:43

8.37 BAT Algebra

This modules contains the most common algebraic BAT manipulation commands. We call them *algebra*, because all operations take values as parameters, and produce new result values, but *do not modify their parameters*. Unlike the previous Monet versions, we reduce the number of functions returning a BAT reference. This was previously needed to simplify recursive bat-expression and manage reference counts. In the current version we return only a BAT identifier when a new bat is being created.

All parameters to the modules are passed by reference. In particular, this means that string values are passed to the module layer as (str *) and we have to de-reference them before entering the gdk library. This calls for knowlegde on the underlying BAT typs`s

BAT

BAT mk Tue, 03/30/2010 - 16:44

8.39 Binary Association Tables

This module contains the commands and patterns to manage Binary Association Tables (BATs). The relational operations you can execute on BATs have the form of a neat algebra, described in algebra.mx

But a database system needs more that just this algebra, since often it is crucial to do table-updates (this would not be permitted in a strict algebra).

All commands needed for BAT updates, property management, basic I/O, persistence, and storage options can be found in this module.

All parameters to the modules are passed by reference. In particular, this means that string values are passed to the module layer as (str *) and we have to de-reference them before entering the gdk library. (Actual a design error in gdk to differentiate passing int/str) This calls for knowledge on the underlying BAT types`s

BAT extensions

BAT extensions mk Tue, 03/30/2010 - 12:22

The kernel libraries are unaware of the MAL runtime semantics. This calls for declaring some operations in the MAL module section and register them in the kernel modules explicitly.

A good example of this borderline case are BAT creation operations, which require a mapping of the type identifier to the underlying implementation type.

```
pattern bat.new(ht:oid, tt:any_1) :bat[:oid,:any_1]
address CMDBATnew
comment "Creates a new empty transient BAT, with head- and tail-types as indicated.";

pattern bat.new(ht:oid, tt:any_1, size:int) :bat[:oid,:any_1]
address CMDBATnew
comment "Creates a new BAT with sufficient space.";

pattern bat.new(ht:oid, tt:any_1, size:lng) :bat[:oid,:any_1]
address CMDBATnew
comment "Creates a new BAT and allocate space.";

pattern bat.new_persistent(ht:oid, tt:any_1) :bat[:oid,:any_1]
address CMDBATnew_persistent
comment "Creates a new empty transient BAT in the persistent farm, with head- and tail-types as indicated.";

pattern bat.new_persistent(ht:oid, tt:any_1, size:int) :bat[:oid,:any_1]
address CMDBATnew_persistent
comment "Creates a new BAT in the persistent farm with sufficient space.";

pattern bat.new_persistent(ht:oid, tt:any_1, size:lng) :bat[:oid,:any_1]
address CMDBATnew_persistent
comment "Creates a new BAT in the persistent farm and allocate space.";

pattern bat.new(b:bat[:oid,:any_1] ) :bat[:oid,:any_1]
address CMDBATnewDerived;
pattern bat.new(b:bat[:oid,:any_1], size:lng) :bat[:oid,:any_1]
address CMDBATnewDerived
comment "Create a new BAT by structure inheritance.";

command bat.new(nme:str):bat[:oid,:any_1]
address CMDBATderivedByName
comment "Localize a bat by name and produce a clone.";

pattern bat.partition(b:bat[:oid,:any_1]):bat[:oid,:any_1]...
address CMDBATpartition
comment "Create a serie of slices over the BAT argument. The BUNs are distributed evenly.";

command bat.new(nme:str):bat[:oid,:any_1]
address CMDBATderivedByName
comment "Localize a bat by name and produce a clone.";

pattern bat.partition(b:bat[:oid,:any_1]):bat[:oid,:any_1]...
address CMDBATpartition
comment "Create a serie of slices over the BAT argument. The BUNs are distributed evenly.";

pattern bat.partition(b:bat[:oid,:any_1], pieces:int, n:int):bat[:oid,:any_1]
address CMDBATpartition2
comment "Create the n-th slice over the BAT broken into severral pieces.";

command bat.imprints(b:bat[:oid,:bte]) :void
address CMDBATimprints;
command bat.imprints(b:bat[:oid,:sht]) :void
address CMDBATimprints;
command bat.imprints(b:bat[:oid,:int]) :void
address CMDBATimprints;
command bat.imprints(b:bat[:oid,:lng]) :void
address CMDBATimprints;
command bat.imprints(b:bat[:oid,:flt]) :void
address CMDBATimprints;
command bat.imprints(b:bat[:oid,:dbl]) :void
address CMDBATimprints
comment "Check/create an imprint index on the BAT";

command bat.imprintsize(b:bat[:oid,:bte]):lng
address CMDBATimprintsize;
command bat.imprintsize(b:bat[:oid,:sht]):lng
address CMDBATimprintsize;
command bat.imprintsize(b:bat[:oid,:int]):lng
address CMDBATimprintsize;
command bat.imprintsize(b:bat[:oid,:lng]):lng
address CMDBATimprintsize;
command bat.imprintsize(b:bat[:oid,:flt]):lng
address CMDBATimprintsize;
command bat.imprintsize(b:bat[:oid,:dbl]):lng
address CMDBATimprintsize
comment "Return the size of the imprints";
```

BBP

BBP mk Tue, 03/30/2010 - 12:23

The BBP module provides inspection routines over the BAT buffer pool.

The remainder of this module contains operations that rely on the MAL runtime setting, but logically belong to the kernel/bat module.

module bbp;

`command open():void`

`pattern releaseAll():void`

`pattern iterator(nme:str):lng`

address CMDbbpiterator comment "Locate the next element in the box.";

`pattern prelude():void`

address CMDbbpprelude comment "Initialize the bbp box.";

`pattern bind(name:str):bat[:any_1,:any_2]`

address CMDbbpbind comment "Locate the BAT using its logical name";

`command getHeadType() :bat[:int,:str]`

address CMDbbpHeadType comment "Map a BAT into its head type";

`command getTailType() :bat[:int,:str]`

address CMDbbpTailType comment "Map a BAT into its tail type";

`command getNames() :bat[:int,:str]`

address CMDbbpNames comment "Map BAT into its bbp name";

`command getRNames() :bat[:int,:str]`

address CMDbbpRNames comment "Map a BAT into its bbp physical name";

`command getName( b:bat[:any_1,:any_2]):str`

address CMDbbpName comment "Map a BAT into its internal name";

`command getCount() :bat[:int,:lng]`

address CMDbbpCount comment "Create a BAT with the cardinalities of all known BATs";

`command getRefCount() :bat[:int,:int]`

address CMDbbpRefCount comment "Create a BAT with the (hard) reference counts";

`command getLRefCount() :bat[:int,:int]`

address CMDbbpLRefCount comment "Create a BAT with the logical reference counts";

`command getLocation() :bat[:int,:str]`

address CMDbbpLocation comment "Create a BAT with their disk locations";

`command getHeat() :bat[:int,:int]`

address CMDbbpHeat comment "Create a BAT with the heat values";

`command getDirty() :bat[:int,:str]`

address CMDbbpDirty comment "Create a BAT with the dirty/ diffs/clean status";

```
command getStatus() :bat[:int,:str]
```

address CMDbbpStatus comment "Create a BAT with the disk/load status";

```
command getKind():bat[:int,:str]
```

address CMDbbpKind comment "Create a BAT with the persistency status";

```
command getRefCount(b:bat[:any_1,:any_2]) :int
```

address CMDgetBATrefcnt comment "Utility for debugging MAL interpreter";

```
command getLkRefCount(b:bat[:any_1,:any_2]) :int
```

address CMDgetBATlrefcnt comment "Utility for debugging MAL interpreter";

```
command getDiskSpace() :lng
```

address CMDbbpDiskSpace comment "Estimate the amount of disk space occupied by dbfarm";

```
command getPageSize():int
```

address CMDgetPageSize comment "Obtain the memory page size";

Calculator

Calculator mk Tue, 03/30/2010 - 16:46

8.53 Basic arithmetic

This module is an extended version of the V4 arithmetic module. It implements the arithmetic operations on the built-in types, *chr*, *bte*, *sht*, *int*, *flt*, *dbl* and *lng*. All combinations are implemented. Limited combinations are implemented for *bit*, *oid* and *str*.

```
[binary operators]
```

The implemented operators are first of all all comparison that return a TRUE/FALSE value (*bit* values), i.e. <=, <, ==, !=, >=, and >=.

The module also implements the operators +, -, * and /. The rules for the return types operators is as follows. If one of the input types is a floating point the result will be a floating point. The largest type of the input types is taken.

The *max* and *min* functions return the maximum and minimum of the two input parameters.

```
[unary operators]
```

This module also implements the unary *abs()* function, which calculates the absolute value of the given input parameter, as well as the - unary operator.

The *inv* unary operation calculates the inverse of the input value. An error message is given when the input value is zero.

```
[bitwise operators]
```

For integers there are some additional operations. The % operator implements the congruent modulo operation. The << and >> are the left and right bit shift. The *or*, *and*, *xor* and *not* for integers are implemented as bitwise boolean operations.

```
[boolean operators]
```

The *or*, *and*, *xor* and *not* for the bit atomic type in MIL (this corresponds to what is normally called boolean) are implemented as the logic operations.

```
[random numbers]
```

This module also contains the rand and srand functions. The *srand()* function initializes the random number generator using a seed value. The subsequent calls to *rand()* are pseudo random numbers (with the same seed the sequence can be repeated).

The general interpretation for the NIL value is "unknown". This semantics mean that any operation that receives at least one NIL value, will produce a NIL value in the output for sure.

The only exception to this rule are the "==" and "!=" equality test routines (it would otherwise become rather difficult to test whether a value is nil).

Clients

Clients mk Tue, 03/30/2010 - 16:24

8.8 Client Management

Each online client is represented with an entry in the clients table. The client may inspect his record at run-time and partially change its properties. The administrator sees all client records and has the right to adjust global properties.

module clients;

```
pattern setListing(flag:int):int
```

address CLTsetListing comment "Turn on/off echo of MAL instructions: 2 - show mal instruction, 4 - show details of type resolutoin, 8 - show binding information.";

```
pattern setHistory(s:str)
```

address CLTsetHistory comment "Designate console history file for readline.";

```
pattern getId():int
```

address CLTgetClientId comment "Return a number that uniquely represents the current client.";

```
pattern getInfo( ):bat[:str,:str]
```

address CLTinfo comment "Pseudo bat with client attributes.";

```
pattern getScenario():str
```

address CLTgetScenario comment "Retrieve current scenario name.";

```
pattern setScenario(msg:str):str
```

address CLTsetScenario comment "Switch to other scenario handler, return previous one.";

```
pattern quit():void
```

address CLTquit comment "Terminate the client session.";

```
pattern quit(idx:int):void
```

address CLTquit comment "Terminate the session for a single client using a soft error. It is the privilige of the console user.";

Administrator operations

```
command getLogins( ):bat[:int,:str]
```

address CLTLogin comment "Pseudo bat of client login time.";

```
command getLastCommand( ):bat[:int,:str]
```

address CLTLastCommand comment "Pseudo bat of client's last command time.";

```
command getActions( ):bat[:int,:int]
```

address CLTActions comment "Pseudo bat of client's command counts.";

```
command getTime( ):bat[:int,:lng]
```

address CLTTime comment "Pseudo bat of client's total time usage(in usec).";

```
command getUsers( ):bat[:int,:str]
```

address CLTusers comment "Pseudo bat of users logged in.";

```
pattern stop(id:int)
```

address CLTstop comment "Stop the query execution at the next eligible statement.";

```
pattern suspend(id:int):void
```

address CLTsuspend comment "Put a client process to sleep for some time. It will simple sleep for a second at a time, until the awake bit has been set in its descriptor";

```
command wakeup(id:int):void
```

address CLTwakeup comment "Wakeup a client process";

```
pattern setTimeout(q:int,x:int):void
```

address CLTsetTimeout comment "Abort a query after q seconds (q=0 means run undisturbed). The session timeout aborts the connection after spending too many seconds on query processing.";

```
pattern getTimeout(){q:int,x:int}
```

address CLTgetTimeout comment "A query is aborted after q seconds (q=0 means run undisturbed). The session timeout aborts the connection after spending too many seconds on query processing.";

```
command shutdown(forced:bit):void
```

address CLTshutdown comment "Close all client connections. If forced=false the clients are moved into FINISHING mode, which means that the process stops at the next cycle of the scenario. If forced=true all client processes are immediately killed";

Debugger

Debugger mk Tue, 03/30/2010 - 16:28

8.13 MAL debugger interface

This module provides access to the functionality offered by the MonetDB debugger and interpreter status. It is primarilly used in interactive sessions to activate the debugger at a given point. Furthermore, the instructions provide the necessary handle to generate information for post-mortum analysis.

To enable ease of debugging and performance monitoring, the MAL interpreter comes with a hardwired gdb-like text-based debugger. A limited set of instructions can be included in the programs themselves, but beware that debugging has a global effect. Any concurrent user will be affected by breakpoints being set.

The prime scheme to inspect the MAL interpreter status is to use the MAL debugger directly. However, in case of automatic exception handling it helps to be able to obtain BAT versions of the critical information, such as stack frame table, stack trace, and the instruction(s) where an exception occurred. The inspection typically occurs in the exception handling part of the MAL block.

Beware, a large class of internal errors can not easily captured this way. For example, bus-errors and segmentation faults lead to premature termination of the process. Similar, creation of the post-mortum information may fail due to an inconsistent state or insufficient resources.

```
module mdb;

pattern start():void
address MDBstart
comment "Start interactive debugger";
pattern start(clientid:int):void
address MDBstart
comment "Start interactive debugger on a client";
pattern start(mod:str,fcn:str):void
address MDBstartFactory
comment "Start interactive debugger on a running factory";

pattern stop():void
address MDBstop
comment "Stop the interactive debugger.";

pattern inspect(mod:str,fcn:str):void
address MDBinspect
comment "Run the debugger on a specific function";

command modules():bat[:oid,:str]
address CMDmodules
comment "List available modules";

pattern setTrap(mod:str, fcn:str, b:bit):void
address MDBtrapFunction
comment "Suspend upon a call to the MAL function.";

pattern setTrap(idx:int):void
address mdbTrapClient
comment "Call debugger for a specific process.";

pattern setTrace(b:bit):void
address MDBsetTrace
comment "Turn on/off tracing of current routine";

pattern setTrace(b:str):void
address MDBsetVarTrace
comment "Turn on/off tracing of a variable ";

pattern setCatch(b:bit):void
address MDBsetCatch
comment "Turn on/off catching exceptions";

command getDebug():int
address MDBgetDebug
comment "Get the kernel debugging bit-set. See the MonetDB configuration file for details";

command setDebug(flg:str):int
address MDBsetDebugStr
comment "Set the kernel debugging bit-set and return its previous value. The recognized options are: threads, memory, properties,
io, transactions, modules, algorithms, estimates.";
command setDebug(flg:int):int
address MDBsetDebug
comment "Set the kernel debugging bit-set and return its previous value.";

command getException(s:str):str
address MDBgetExceptionVariable
comment "Extract the variable name from the exception message";
command getReason(s:str):str
address MDBgetExceptionReason
comment "Extract the reason from the exception message";
command getContext(s:str):str
address MDBgetExceptionContext
comment "Extract the context string from the exception message";

pattern list():void
address MDBlist
comment "Dump the current routine on standard out.";
pattern listMapi():void
address MDBlistMapi
comment "Dump the current routine on standard out with Mapi prefix.";
pattern list(Mstr,Fstr):void
address MDBlist3
comment "Dump the routine M.F on standard out.";
pattern List():void
address MDBlistDetail
comment "Dump the current routine on standard out.";
pattern List(Mstr,Fstr):void
address MDBlistDetail
comment "Dump the routine M.F on standard out.";
pattern var():void
address MDBvar
comment "Dump the symboltable of current routine on standard out.";
pattern var(Mstr,Fstr):void
address MDBvar3
comment "Dump the symboltable of routine M.F on standard out.";
pattern lifespan(Mstr,Fstr):void
address MDBlifespan
comment "Dump the current routine lifespan information on standard out.";

pattern grab():void
address mdbGrab
comment "Call debugger for a suspended process.";
pattern trap():void
address mdbTrap
comment "A suspended process for debugging.";

pattern dot(s:str):void
address MDBshowFlowGraph
comment "Dump the data flow of the current routine in a format recognizable by the command 'dot' to the file s";

pattern dot(Mstr,Fstr,s:str):void
address MDBshowFlowGraph
comment "Dump the data flow of the function M.F in a format recognizable by the command 'dot' on the file s";

pattern getStackDepth():int
address MDBStkDepth
comment "Return the depth of the calling stack.";

pattern getStackFrame(s:int):{bat[:oid,:str] ,bat[:oid,:str]}
address MDBGetStackFrame
pattern getStackFrame():{bat[:oid,:str] ,bat[:oid,:str]}
address MDBgetStackFrame
comment "Collect variable binding of current (n-th) stack frame.";
pattern getStackTrace():{bat[:oid,:str], ,bat[:oid,:str]}
address MDBStkTrace;

pattern dump()
address MDBdump
comment "Dump instruction, stacktrace, and stack";

pattern getDefinition():bat[:oid,:str]
address MDBgetDefinition
comment "Returns a string representation of the current function with typing information attached";
```

Factories

Factories mk Tue, 03/30/2010 - 16:25

8.9 Factory management

The factory infrastructure can be inspected and steered with the commands provided here.

module factories;

```
command getPlants():{mod:bat[:oid,:str], fcn:bat[:oid,:str]}
```

address FCTgetPlants comment "Retrieve the names for all active factories.";

```
command getCaller():int
```

address FCTgetCaller comment "Retrieve the unique identity of the factory caller.";

```
command getOwners():bat[:oid,:str]
```

address FCTgetOwners comment "Retrieve the factory owners table.";

```
command getArrival():bat[:oid,:timestamp]
```

address FCTgetArrival comment "Retrieve the time stamp the last call was made.";
command getDeparture():bat[:oid,:timestamp]
address FCTgetDeparture comment "Retrieve the time stamp the last answer was returned.";
pattern shutdown(mb:str, f:str):void
address FCTshutdown comment "Close a factory.";

I/O

I/O mk Tue, 03/30/2010 - 16:26

8.11 Input/Output module

The IO module provides simple ascii-io rendering options. It is modeled after the tuple formats, but does not attempt to outline the results. Instead, it is geared at speed, which also means that some functionality regarding the built-in types is duplicated from the atoms definitions.

A functional limited form of formatted printf is also provided. It accepts at most one variable. A more complete approach is the tablet module.

The commands to load and save a BAT from/to an ASCII dump are efficient, but work only for binary tables.

```
module io;
pattern stdin():bstream
address io_stdin comment "return the input stream to the database client";
pattern stderr():streams
address io_stderr comment "return the error stream for the database console";
pattern stdout():streams
address io_stdout comment "return the output stream for the database client";
pattern print(val:any_1,tst:any...):void
address fOprint_val comment "Print a MAL value tuple .";
pattern print(bl:bat[:any_1,:any]...):void
address fOtable comment "BATs are printed with '#' for legend lines, and the BUNs on seperate lines between brackets, containing each to comma separated values (head and tail). If multiple BATs are passed for printing, print() performs an implicit natural join, producing a multi attribute table.";
pattern ftable( filep:streams, b1:bat[:any_1,:any], b:bat[:any_1,:any]... ):void
address fOftable comment "Print an n-ary table to a file.";
pattern print(order:int,b:bat[:any_1,:any], b2:bat[:any_1,:any]...):void
address fOxtable comment "The same as normal table print, but enforces to use the order of BAT number [1..argc] to do the printing.";
pattern table(b1:bat[:any_1,:any], b2:bat[:any_1,:any]...):void
address fOtable comment "Print an n-ary table. Like print, but does not print oid column";
pattern table(order:int, b1:bat[:any_1,:any], b2:bat[:any_1,:any]...):void
address fOtable comment "Print an n-ary table.";
pattern ftable(fp:streams, order:int, b1:bat[:any_1,:any], b:bat[:any_1,:any]...):void
address fOftable comment "Print an n-ary table to a file.";
pattern print(val:any_1):void
address fOprint_val comment "Print a MAL value tuple .";
pattern print(val:bat[:any_1,:any_2]):void
address fOprint_val comment "Print a MAL value tuple .";
pattern prompt(val:any_1):void
address fOprompt_val comment "Print a MAL value without brackets.";
pattern printf(fmt:str,val:any...):void
address fOprintf comment "Select default format ";
pattern printf(fmt:str):void
address fOprintf comment "Select default format ";
pattern printf(filep:streams,fmt:str,val:any...):void
address fOprintfStream comment "Select default format ";
pattern printf(filep:streams,fmt:str):void
address fOprintfStream comment "Select default format ";
command data(fname:str):str
address fOdatafile comment "Signals receipt of tuples in a file fname. It returns the name of the file, if it still exists.";
command export(b:bat[:any_1,:any_2], filepath:str):bit
address fOexport comment "Export a BAT as ASCII to a file. If the 'filepath' is not absolute, it is put into the .../dbfarm/$DB directory. Success of failure is indicated.";
command import(b:bat[:any_1,:any_2], filepath:str) :bat[:any_1,:any_2]
address fOimport comment "Import a BAT from an ASCII dump. The new tuples are *inserted* into the parameter BAT. You have to create it! Its signature must match the dump, else parsing errors will occur and FALSE is returned.";
```

Imprints

Imprints mk Sun, 12/28/2014 - 16:31

Imprints are a novel secondary index structure to speed up scans over large tables described in the ACM SIGMOD paper "Column imprints: a secondary index structure". They are automatically created over columns of fixed type and carry a maximal storage overhead of 12%. An effective compression scheme brings this often down to much less storage.

```
command bat.imprints(b:bat[:oid,:bte]) :void
address CMDBATimprints;
command bat.imprints(b:bat[:oid,:sht]) :void
address CMDBATimprints;
command bat.imprints(b:bat[:oid,:int]) :void
address CMDBATimprints;
command bat.imprints(b:bat[:oid,:lng]) :void
address CMDBATimprints;
command bat.imprints(b:bat[:oid,:flt]) :void
address CMDBATimprints;
command bat.imprints(b:bat[:oid,:dbl]) :void
address CMDBATimprints;
comment "Check for existence or create an imprint index on the BAT.";

command bat.imprintsize(b:bat[:oid,:bte]):lng
address CMDBATimprintsize;
command bat.imprintsize(b:bat[:oid,:sht]):lng
address CMDBATimprintsize;
command bat.imprintsize(b:bat[:oid,:int]):lng
address CMDBATimprintsize;
command bat.imprintsize(b:bat[:oid,:lng]):lng
address CMDBATimprintsize;
command bat.imprintsize(b:bat[:oid,:flt]):lng
address CMDBATimprintsize;
command bat.imprintsize(b:bat[:oid,:dbl]):lng
address CMDBATimprintsize
comment "Return the storage size of the imprints index structure.";
```

Inspect

Inspect mk Tue, 03/30/2010 - 16:25

8.10 Inspection

This module introduces a series of commands that provide access to information stored within the interpreter data structures. It's primary use is debugging. In all cases, the pseudo BAT operation is returned that should be garbage collected after being used.

The main performance drain would be to use a pseudo BAT directly to successively access it components. This can be avoided by first assigning the pseudo BAT to a variable.

```
module inspect;
command getWelcome():str
address INSPECTgetWelcome comment "Return the server messsage of the day string";
pattern getDefinition(mod:str,fcn:str) :bat[:str,:str]
address INSPECTgetDefinition comment "Returns a string representation of a specific function.";
pattern getSignature(mod:str,fcn:str) :bat[:str,:str]
address INSPECTgetSignature comment "Returns the function signature(s).";
pattern getAddress(mod:str,fcn:str) :bat[:str,:str]
address INSPECTgetAddress comment "Returns the function signature(s).";
pattern getComment(mod:str,fcn:str) :bat[:str,:str]
address INSPECTgetComment comment "Returns the function help information.";
pattern getSource(mod:str,fcn:str):str
address INSPECTgetSource comment "Return the original input for a function.";
pattern getKind():bat[:oid,:str]
address INSPECTgetKind comment "Obtain the instruction kind.";
pattern getModule():bat[:oid,:str]
address INSPECTgetAllModules comment "Obtain the function name.";
pattern getFunction():bat[:oid,:str]
address INSPECTgetAllFunctions comment "Obtain the function name.";
pattern getSignature():bat[:oid,:str]
address INSPECTgetAllSignatures comment "Obtain the function signatures.";
pattern getAddress():bat[:oid,:str]
address INSPECTgetAllAddresses comment "Obtain the function address.";
pattern getSize():lng
address INSPECTgetSize comment "Return the storage size for the current function (in bytes).";
pattern getSize(mod:str):bat[:str,:lng]
address INSPECTgetModuleSize comment "Return the storage size for a module (in bytes).";
pattern getSize(mod:str,fcn:str):lng
address INSPECTgetFunctionSize comment "Return the storage size for a function (in bytes).";
pattern getType(v:bat[:any_1,:any_2]) (ht:str, tt:str)
address INSPECTgetTypeName comment "Return the concrete type of a variable (expression).";
pattern getType(v:any_1) :str
address INSPECTgetTypeName comment "Return the concrete type of a variable (expression).";
command getTypeName(v:int):str
address INSPECTgetname comment "Get the type name associated with a type id.";
pattern getTypeIndex(v:bat[:any_1,:any_2]) (ht:int, tt:int)
address INSPECTgetTypeIndex comment "Return the type index of a BAT head and tail.";
pattern getTypeIndex(v:any_1):int
address INSPECTgetTypeIndex comment "Return the type index of a variable. For BATs, return the type index for its tail.";
pattern equalType(l:any, r:any):bit
address INSPECTequalType comment "Return true if both operands are of the same type";
command getAtomNames():bat[:int,:str]
address INSPECTatom_names comment "Collect a BAT with the atom names.";
command getAtomSuper():bat[:int,:str]
address INSPECTatom_sup_names comment "Collect a BAT with the atom names.";
command getAtomSizes():bat[:int,:int]
address INSPECTatom_sizes comment "Collect a BAT with the atom sizes.";
command getEnvironment():bat[:str,:str]
address INSPECTgetEnvironment comment "Collect the environment variables.
```

Iterators

Iterators mk Tue, 03/30/2010 - 16:23

BAT Iterators

Many low level algorithms rely on an iterator to break a collection into smaller pieces. Each piece is subsequently processed by a block. For very large BATs it may make sense to break it into chunks and process them separately to solve a query. An iterator pair is provided to chop a BAT into fixed size elements. Each chunk is made available as a BATview. It provides read-only access to an underlying BAT. Adjusting the bounds is cheap, once the BATview descriptor has been constructed.

The smallest granularity is a single (oid,value) pair, which can be used to realize an iterator over the individual BAT elements. For larger sized chunks, the operators return a view.

All iterators require storage space to administer the location of the next element. The BAT iterator module uses a simple lng variable, which also acts as a cursor for barrier statements.

The larger chunks produced are currently static, i.e. their size is a parameter of the call. Dynamic chunk sizes are interesting for time-series query processing. (See another module)

```
module iterator;

command new(b:bat[:oid,:any_2], size:lng) (:lng,:bat[:oid,:any_2])
address ITNewChunk
comment "Create an iterator with fixed granule size. The result is a view.";

command next(b:bat[:oid,:any_2], size:lng) (:lng, :bat[:oid,:any_2])
address ITNextChunk
comment "Produce the next chunk for processing.";

pattern new(b:bat[:oid,:any_2]) (h:oid, t:any_2)
address ITRunIterator
comment "Process the buns one by one extracted from a void table.";

pattern next(b:bat[:oid,:any_2]) (h:oid, t:any_2)
address ITRunNext
comment "Produce the next bun for processing.";

command next(step:oid,last:oid):oid
address ITNext_oid;
command next(step:sht,last:sht):sht
address ITNext_sht;
command next(step:int,last:int):int
address ITNext_int;
command next(step:lng,last:lng):lng
address ITNext_lng;
command next(step:flt,last:flt):flt
address ITNext_flt;
command next(step:dbl,last:dbl):dbl
address ITNext_dbl
comment "Advances the iterator with a fixed value";
```

Groups

Groups mk Tue, 03/30/2010 - 16:46

8.55 The group module

This module contains the primitives to construct, derive, and perform statistical operations on BATs representing groups. The default scheme in Monet is to assume the head to represent the group identifier and the tail an element in the group.

Groups play an important role in datamining, where they are used to construct cross-tables. Such cross tables over a single BAT are already supported by the histogram function. This module provides extensions to support identification of groups in a (multi-)dimensional space.

The module implementation has a long history. The first implementation provided several alternatives to produce/derive the grouping. A more complete (and complex) scheme was derived during its extensive use in the context of the Data Distilleries product. The current implementation is partly a cleanup of this code-base, but also enables provides better access to the intermediate structures produced in the process, i.e. the histogram and the sub-group mapping. They can be used for various optimization schemes at the MAL level.

The prime limitation of the current implementation is that an underlying database of `oid->any` BATs is assumed. This enables representation of each group using an oid, and the value representation of the group can be accordingly be retrieved easily. An optimized implementation in which we use positional integer id's (as embodied by Monet's void type) is also available.

This limitation on (v)oid-headers is marginal. The primitive GRPsplit produces for any BAT two copies with both a (v)oid header.

8.55.1 Algorithms

There are several approaches to build a cross table. The one chosen here is aimed at incremental construction, such that re-use of intermediates becomes possible. Starting with the first dimension, a BAT is derived to represent the various groups, called a *GRP BAT* or cross-table BAT.

8.55.2 Cross Table (GRP)

A cross table is an <oid,oid> BAT where the first (head) denotes a tuple in the cross table and the second (tail) marks all identical lists. The tail-oids contain group identifiers; that is, *this value is equal* **iff** *two tuples belong to the same group*. The group identifiers are chosen from the domain of the tuple-identifiers. This simplifies getting back to the original tuples, when talking about a group. If the tuple-oid of 'John' is chosen as a group-id, you might view this as saying that each member of the group is 'like John' with respect to the grouping-criterion.

Successively the subgroups can be identified by modifying the GRP BAT or to derive a new GRP BAT for the subgroups. After all groups have been identified this way, a BAT histogram operation can be used to obtain the counts of each data cube. Other aggregation operations using the MIL set aggregate construct `{bat}` can be used as well; note for instance that `histogram == {b.reverse()}`.

The Monet interface module specification is shown below. Ideally we should defined stronger type constraints, e.g. command group.new(attr:bat[_any_1]

The group macro is split along three dimensions:

[type:]	Type specific implementation for selecting the right hash function and data size etc.;
[clustered:]	The select the appropriate algorithm, i.e., with or without taking advantage of an order of values in the parent groups;
[physical properties:]	Values , choosing between a fixed predefined and a custom hashmask. Custom allows the user to determine the size of the hashmask (and indirectly the estimated size of the result). The hashmask is 2^n - 1 where n is given by the user, or 1023 otherwise, and the derived result size is 4 ... 2^n.

Further research should point out whether fitting a simple statistical model (possibly a simple mixture model) can help choose these parameters automatically; the current idea is that the user (which could be a domain-specific extension of the higher-level language) knows the properties of the data, especially for IR in which the standard grouping settings differ significantly from the original datamining application.

Language extension

Language extension mk Tue, 03/30/2010 - 16:27

8.12 Language Extensions

Iterators over scalar ranges are often needed, also at the MAL level. The barrier and control primitives are sufficient to mimic them directly.

The modules located in the kernel directory should not rely on the MAL datastructures. That's why we have to deal with some bat operations here and delegate the signature to the proper module upon loading.

Running a script is typically used to initialize a context. Therefore we need access to the runtime context. For the call variants we have to determine an easy way to exchange the parameter/return values.

```
module language;

pattern call(s:str):void
address CMDcallString comment "Evaluate a MAL string program.";
pattern call(s:bat[:oid,:str]):void
address CMDcallBAT comment "Evaluate a program stored in a BAT.";
pattern source(f:str):void
address CMDevalFile comment "Merge the instructions stored in the file with the current program.";

command raise(msg:str) :str
address CMDraise
comment "Raise an exception labeled
    with a specific message.";
command assert(v:bit,term:str):void
address MALassertBit;
command assert(v:sht,term:str):void
address MALassertSht;
command assert(v:int,term:str):void
address MALassertInt;
command assert(v:lng,term:str):void
address MALassertLng;
command assert(v:str,term:str):void
address MALassertStr;
command assert(v:oid,term:str):void
address MALassertOid;
pattern assert(v:any_1,pname:str,oper:str,val:any_2):void
address MALassertTriple
comment "Assertion test.";

pattern dataflow():bit
address MALstartDataflow
comment "The current guarded block is executed using dataflow control. ";

pattern sink(v:any...):void
address MALgarbagesink
comment "Variables to be considered together when triggering garbage collection.
Used in the dataflow blocks to avoid early release of values.";

pattern pass(v:any_1):any_1
address MALpass
comment "Cheap instruction to disgard storage while retaining the dataflow dependency";

pattern register(m:str,f:str,code:str,help:str):void
address CMDregisterFunction
comment"Compile the code string to MAL and register it as a function.";
```

Logger

Logger mk Tue, 03/30/2010 - 16:48

8.57 The Transaction Logger

In the philosophy of MonetDB, transaction management overhead should only be paid when necessary. Transaction management is for this purpose implemented as a separate module and applications are required to obey the transaction policy, e.g. obtaining/releasing locks.

This module is designed to support efficient logging of the SQL database. Once loaded, the SQL compiler will insert the proper calls at transaction commit to include the changes in the log file.

The logger uses a directory to store its log files. One master log file stores information about the version of the logger and the transaction log files. This file is a simple ascii file with the following format: `6DIGIT-VERSION\n[log file number \n]*]*` The transaction log files have a binary format, which stores fixed size logformat headers (flag,nr,bid), where the flag is the type of update logged. The nr field indicates how many changes there were (in case of inserts/deletes). The bid stores the bid identifier.

The key decision to be made by the user is the location of the log file. Ideally, it should be stored in fail-safe environment, or at least the log and databases should be on separate disk columns.

This file system may reside on the same hardware as the database server and therefore the writes are done to the same disk, but could also reside on another system and then the changes are flushed through the network. The logger works under the assumption that it is called to safeguard updates on the database when it has an exclusive lock on the latest version. This lock should be guaranteed by the calling transaction manager first.

Finding the updates applied to a BAT is relatively easy, because each BAT contains a delta structure. On commit these changes are written to the log file and the delta management is reset. Since each commit is written to the same log file, the beginning and end are marked by a log identifier.

A server restart should only (re)process blocks which are completely written to disk. A log replay therefore ends in a commit or abort on the changed bats. Once all logs have been read, the changes to the bats are made persistent, i.e. a bbp sub-commit is done.

MAPI Interface

MAPI Interface mk Tue, 03/30/2010 - 16:29

8.15 MAPI interface

The complete Mapi library is available to setup communication with another Mserver.

Clients may initialize a private listener to implement specific services. For example, in an OLTP environment it may make sense to have a listener for each transaction type, which simply parses a sequence of transaction parameters.

Authorization of access to the server is handled as part of the client record initialization phase.

This library internally uses pointer handles, which we replace with an index in a locally maintained table. It provides a handle to easily detect havoc clients.

A cleaner and simplier interface for distributed processing is available in the module remote.

```
module mapi;
command listen():int
	address SERVERlisten_default comment "Start a Mapi server with the default settings.";
command listen(port:int):int
	address SERVERlisten_port comment "Start a Mapi listener on the port given.";
command listen(port:int, maxusers:int):int
	address SERVERlisten2 comment "Start a Mapi listener.";
command listen(port:int, maxusers:int, cmd:str):int
	address SERVERlisten3 comment "Start the Mapi listener on <port> for <maxusers>. For a new client connection MAL procedure <cmd>(Stream s_in, Stream s_out) is called.If no <cmd> is specified a new client thread is forked.";
command stop():void
	address SERVERstop comment "Terminate connection listeners.";
command suspend():void
	address SERVERsuspend comment "Suspend accepting connections.";
command resume():void
	address SERVERresume comment "Resume connection listeners.";
command malclient(in:streams, out:streams):void
	address SERVERclient comment "Start a Mapi client for a particular stream pair.";
command trace(mid:int,flag:int):void
	address SERVERtrace comment "Toggle the Mapi library debug tracer.";
pattern reconnect(host:str, port:int, usr:str, passwd:str,lang:str):int
	address SERVERreconnectWithoutAlias comment "Re-establish connection with a remote mserver.";
pattern reconnect(host:str, port:int, db_alias:str, usr:str, passwd:str,lang:str):int
	address SERVERreconnectAlias comment "Re-establish connection with a remote mserver.";
command reconnect(mid:int):void
	address SERVERreconnect comment "Re-establish a connection.";
pattern connect(host:str, port:int, usr:str, passwd:str,lang:str):int
	address SERVERconnect comment "Establish connection with a remote mserver.";
command disconnect(dbalias:str):int
	address SERVERdisconnectWithAlias comment "Close connection with a remote Mserver.";
command disconnect():int
	address SERVERdisconnectALL comment "Close connections with all remote Mserver.";
command setAlias(dbalias:str)
	address SERVERsetAlias comment "Give the channel a logical name.";
command lookup(dbalias:str):int
	address SERVERlookup comment "Retrieve the connection identifier.";
command disconnect(mid:int):void
	address SERVERdisconnect comment "Terminate the session.";
command destroy(mid:int):void
	address SERVERdestroy comment "Destroy the handle for an Mserver.";
command ping(mid:int):int
	address SERVERping comment "Test availability of an Mserver.";
command query(mid:int, qry:str):int
	address SERVERquery comment "Send the query for execution";
command query_handle(mid:int, qry:str):int
	address SERVERquery_handle comment "Send the query for execution.";
pattern query_array(mid:int, qry:str, arg:str...):int
	address SERVERquery_array comment "Send the query for execution replacing '?' by arguments.";
command prepare(mid:int, qry:str):int
	address SERVERprepare comment "Prepare a query for execution.";
command finish(hdl:int):int
	address SERVERfinish comment "Remove all remaining answers.";
command get_field_count(hdl:int):int
	address SERVERget_field_count comment "Return number of fields.";
command get_row_count(hdl:int):lng
	address SERVERget_row_count comment "Return number of rows.";
command rows_affected(hdl:int):lng
	address SERVERrows_affected comment "Return number of affected rows.";
command fetch_row(hdl:int):int
	address SERVERfetch_row comment "Retrieve the next row for analysis.";
command fetch_all_rows(hdl:int):lng
	address SERVERfetch_all_rows comment "Retrieve all rows into the cache.";
command fetch_field(hdl:int,fnr:int):str
	address SERVERfetch_field_str comment "Retrieve a single field.";
command fetch_field(hdl:int,fnr:int):int
	address SERVERfetch_field_int comment "Retrieve a single int field.";
command fetch_field(hdl:int,fnr:int):lng
	address SERVERfetch_field_lng comment "Retrieve a single lng field.";
command fetch_field(hdl:int,fnr:int):sht
	address SERVERfetch_field_sht comment "Retrieve a single sht field.";
command fetch_field(hdl:int,fnr:int):void
	address SERVERfetch_field_void comment "Retrieve a single void field.";
command fetch_field(hdl:int,fnr:int):oid
	address SERVERfetch_field_oid comment "Retrieve a single void field.";
command fetch_field(hdl:int,fnr:int):chr
	address SERVERfetch_field_chr comment "Retrieve a single chr field.";
command fetch_field_array(hdl:int):bat[:int,:str]
	address SERVERfetch_field_bat comment "Retrieve all fields for a row.";
command fetch_line(hdl:int):str
	address SERVERfetch_line comment "Retrieve a complete line.";
command fetch_reset(hdl:int):int
	address SERVERfetch_reset comment "Reset the cache read line.";
command next_result(hdl:int):int
	address SERVERnext_result comment "Go to next result set.";
command error(mid:int):int
	address SERVERerror comment "Check for an error in the communication.";
command getError(mid:int):str
	address SERVERgetError comment "Get error message.";
command explain(mid:int):str
	address SERVERexplain comment "Turn the error seen into a string.";
pattern put(mid:int, nme:str, val:any_1):void
	address SERVERput comment "Send a value to a remote site.";
pattern put(nme:str, val:any_1):str
	address SERVERputLocal comment "Prepare sending a value to a remote site.";
pattern rpc(key:int,qry:str...):any
	address SERVERmapi_rpc_single_row comment "Send a simple query for execution and fetch result.";
pattern rpc(key:int,qry:str):bat[:any_1,:any_2]
	address SERVERmapi_rpc_bat;
command rpc(key:int,qry:str):void
	address SERVERquery comment "Send a simple query for execution.";
pattern bind(key:int,rschema:str,rtable:str,rcolumn:str,i:int):bat[:any_1,:any_2]
	address SERVERbindBAT comment "Bind a remote variable to a local one.";
pattern bind(key:int,rschema:str,rtable:str,i:int):bat[:any_1,:any_2]
	address SERVERbindBAT comment "Bind a remote variable to a local one.";
pattern bind(key:int,remoteName:str):bat[:any_1,:any_2]
	address SERVERbindBAT comment "Bind a remote variable to a local one.";

	mapi.listen();
```

Manual

Manual mk Tue, 03/30/2010 - 16:29

8.14 Manual Inspection

This module introduces a series of commands that provide access to the help information stored in the runtime environment.

The manual bulk operations ease offline inspection of all function definitions. It includes an XML organized file, because we expect external tools to massage it further for presentation.

```
module manual;
pattern help(text:str)
	address MANUALhelp comment "Produces a list of all <module>.<function> that match the text pattern. The wildcard '*' can be used for <module> and <function>. Using the '(' asks for signature information and using ')' asks for the complete help record.";
pattern search(text:str)
	address MANUALsearch comment "Search the manual for command descriptions that match the regular expression 'text'";
pattern createXML(mod:str):void
	address MANUALcreate1 comment "Generate a synopsis of a module";
pattern createXML():void
	address MANUALcreate0 comment "Produces a XML-formatted manual over all modules loaded.";
pattern section(mod:str):void
	address MANUALcreateSection comment "Generate a synopsis of a module for the reference manual";
pattern index():void
	address MANUALcreateIndex comment "Produces an overview of all names grouped by module.";
pattern summary():void
	address MANUALcreateSummary comment "Produces a manual with help lines grouped by module.";
pattern completion(pat:str):bat[:int,:str]
	address MANUALcompletion comment "Produces the wordcompletion table.";
```

PCRE library

PCRE library mk Tue, 03/30/2010 - 16:33

8.19 PCRE library interface

The PCRE library is a set of functions that implement regular expression pattern matching using the same syntax and semantics as Perl, with just a few differences. The current implementation of PCRE (release 4.x) corresponds approximately with Perl 5.8, including support for UTF-8 encoded strings. However, this support has to be explicitly enabled; it is not the default.

Profiler

Profiler mk Tue, 03/30/2010 - 16:32

8.18 Performance profiler

A key issue in developing fast programs using the Monet database back-end requires a keen eye on where performance is lost. Although performance tracking and measurements are highly application dependent, a simple to use tool makes life a lot easier (See stethoscope and tomograph).

Activation of the performance monitor has a global effect, i.e. all concurrent actions on the kernel are traced, but the events are only sent to the client who initiated the profiler thread.

The profiler event can be handled in several ways. The default strategy is to ship the event record immediately over a stream to a performance monitor. An alternative strategy is preparation of off-line performance analysis, where event information is retained in tables.

To reduce the interference of performance measurement with the experiments, the user can use an event cache, which is emptied explicitly upon need.

```
module profiler;
pattern activate(name:str...):void
pattern deactivate(name:str...):void
pattern openStream():void
    address CMDopenProfilerStream comment "Send the events to output stream";
pattern openStream(fnm:str):void
command closeStream():void
    address CMDcloseProfilerStream comment "Stop sending the event records";
pattern setAll():void
pattern setFilter(v:any):void
    address CMDsetFilterVariable comment "Generate an event record for every instruction where v is used."
pattern clrFilter(mod:str,fcn:str):void
pattern setStartPoint(mod:str,fcn:str):void
pattern start():void
    address CMDstartProfiler comment "Start performance tracing";
command noop():void
pattern stop():void
    address CMDstopProfiler comment "Stop performance tracing";
command reset():void
    address CMDclearTrace comment "Clear the profiler traces";
command dumpTrace():void
command getTrace(m:str):bat[:int,:any_1]
    address CMDgetTrace comment "Get the trace details of a specific event";
pattern getEvent():(:lng,:lng,:lng)
command cleanup():void
    address CMDcleanup comment "Remove the temporary tables for profiling";
command getDiskReads():lng
    address CMDgetDiskReads comment "Obtain the number of physical reads";
command getDiskWrites():lng
    address CMDgetDiskWrites comment "Obtain the number of physical reads";
command getUserTime():lng
    address CMDgetUserTime comment "Obtain the user timing information.";
command getSystemTime():lng
    address CMDgetSystemTime comment "Obtain the user timing information.";
pattern getFootprint():lng
```

Remote

Remote mk Tue, 03/30/2010 - 16:34

8.20 Remote querying functionality

Communication with other mservers at the MAL level is a delicate task. However, it is indispensable for any distributed functionality. This module provides an abstract way to store and retrieve objects on a remote site. Additionally, functions on a remote site can be executed using objects available in the remote session context. This yields in four primitive functions that form the basis for distribution methods: get, put, register and exec.

The get method simply retrieves a copy of a remote object. Objects can be simple values, strings or BATs. The same holds for the put method, but the other way around. A local object can be stored on a remote site. Upon a successful store, the put method returns the remote identifier for the stored object. With this identifier the object can be addressed, e.g. using the get method to retrieve the object that was stored using put.

The get and put methods are symmetric. Performing a get on an identifier that was returned by put, results in an object with the same value and type as the one that was put. The result of such an operation is equivalent to making an (expensive) copy of the original object.

The register function takes a local MAL function and makes it known at a remote site. It ensures that it does not overload an already known operation remotely, which could create a semantic conflict. Deregister a function is forbidden, because it would allow for taking over the remote site completely. C-implemented functions, such as io.print() cannot be remotely stored. It would require even more complicated (byte?) code shipping and remote compilation to make it work. Currently, the remote procedure may only returns a single value.

The choice to let exec only execute functions was made to avoid problems to decide what should be returned to the caller. With a function it is clear and simple to return that what the function signature prescribes. Any side effect (e.g. io.print calls) may cause havoc in the system, but are currently ignored.

This leads to the final contract of this module. The methods should be used correctly, by obeying their contract. Failing to do so will result in errors and possibly undefined behaviour.

The resolve() function can be used to query Merovingian. It returns one or more databases discovered in its vincinity matching the given pattern.

```
module remote;

# module loading and unloading funcs

command prelude():void
    address RMTprelude comment "Initialise the remote module.";
command epilogue():void
    address RMTepilogue comment "Release the resources held by the remote module.";

    # global connection resolve function

command resolve(pattern:str):bat[:oid,:str]
    address RMTresolve comment "resolve a pattern against Merovingian and return the URIs";

    # session local connection instantiation functions

command connect(uri:str, user:str, passwd:str):str
    address RMTconnect comment "returns a newly created connection for uri, using user name and password";
command connect(uri:str, user:str, passwd:str, scen:str):str
    address RMTconnectScen comment "returns a newly created connection for uri, using user name, password and scenario";
command disconnect(conn:str):void
    address RMTdisconnect comment "disconnects the connection pointed to by handle (received from a call to connect())";

    # core transfer functions

pattern get(conn:str, ident:str):any
    address RMTget comment "retrieves a copy of remote object ident";
pattern put(conn:str, object:any):str
    address RMTput comment "copies object to the remote site and returns its identifier";
pattern register(conn:str, mod:str, fcn:str):void
    address RMTregister comment "register <mod>.<fcn> at the remote site";
pattern exec(conn:str, mod:str, func:str):str
    address RMTexec comment "remotely executes <mod>.<func> and returns the handle to its result";
pattern exec(conn:str, mod:str, func:str)(:str, :str)
    address RMTexec comment "remotely executes <mod>.<func> and returns the handle to its result";
pattern exec(conn:str, mod:str, func:str, :str...):str
    address RMTexec comment "remotely executes <mod>.<func> using the argument list of remote objects and returns the handle to its result";
pattern exec(conn:str, mod:str, func:str, :str...)(:str, :str)
    address RMTexec comment "remotely executes <mod>.<func> using the argument list of remote objects and returns the handle to its result";
```

Transactions

Transactions mk Tue, 03/30/2010 - 16:36

8.23 Transaction management

In the philosophy of Monet, transaction management overhead should only be paid when necessary. Transaction management is for this purpose implemented as a module. This code base is largely absolute and should be re-considered when serious OLTP is being supported. Note, however, the SQL front-end obeys transaction semantics.

```
module transaction;
command sync():bit
    address TRNglobal_sync comment "Save all persistent BATs";
command commit():bit
    address TRNglobal_commit comment "Global commit on all BATs";
command abort():bit
    address TRNglobal_abort comment "Global abort on all BATs";
command subcommit(b:bat[:any_1,:str]):bit
    address TRNsubcommit comment "commit only a set of BATnames, passed in the tail (to which you must have exclusive access!)";
pattern commit(c:any...)
    address TRNtrans_commit comment "Commit changes in certain BATs.";
pattern abort(c:any...)
    address TRNtrans_abort comment "Abort changes in certain BATs.";
pattern clean(c:any...)
    address TRNtrans_clean comment "Declare a BAT clean without flushing to disk.";
command prev(b:bat[:any_1,:any_2]):bat[:any_1,:any_2]
    address TRNtrans_prev comment "The previous stae of this BAT";
command alpha(b:bat[:any_1,:any_2]) :bat[:any_1,:any_2]
    address TRNtrans_alpha comment "List insertions since last commit.";
command delta(b:bat[:any_1,:any_2]) :bat[:any_1,:any_2]
    address TRNtrans_delta comment "List deletions since last commit.";
```