Contents

1 MONETDB Internals

`http://sites.computer.org/debull/A12mar/monetdb.pdf` MonetDB Internals Source Compile

1.1 Redesign considerations.

Redesign of the MonetDB software driven by the need to reduce the effort to extend the system into novel directions and to reduce the **Total Execution Cost (TEC)**.

TEC:

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

OLTP -> Online Transaction Processing -> expected most of the cost to be in (P,O) OLAP -> Online Analytical Processing -> expected most of the cost to be in (D,E,R)

1.2 Storage Model

- Represents relational tables using vertical fragmentation.
- Stores each column in a separate {(OID,value)} table, called a **BAT (Binary Association Table)**
- 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.
- **BAT** is implemented as an ordinary C-array. OID maps to the index in the array.
- Persistent version of **BAT** is a **memory mapped file**.
- **O(1) positional database lookup mechanism** (MMU - memory management unit)

1.3 All (relational) operators exploit a small set of properties:

- 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 column form a dense sequence
- sorted - the column contains a sorted list for ordered domains
- revsorted - the column contains a reversed sorted list

1.4 Execution Model

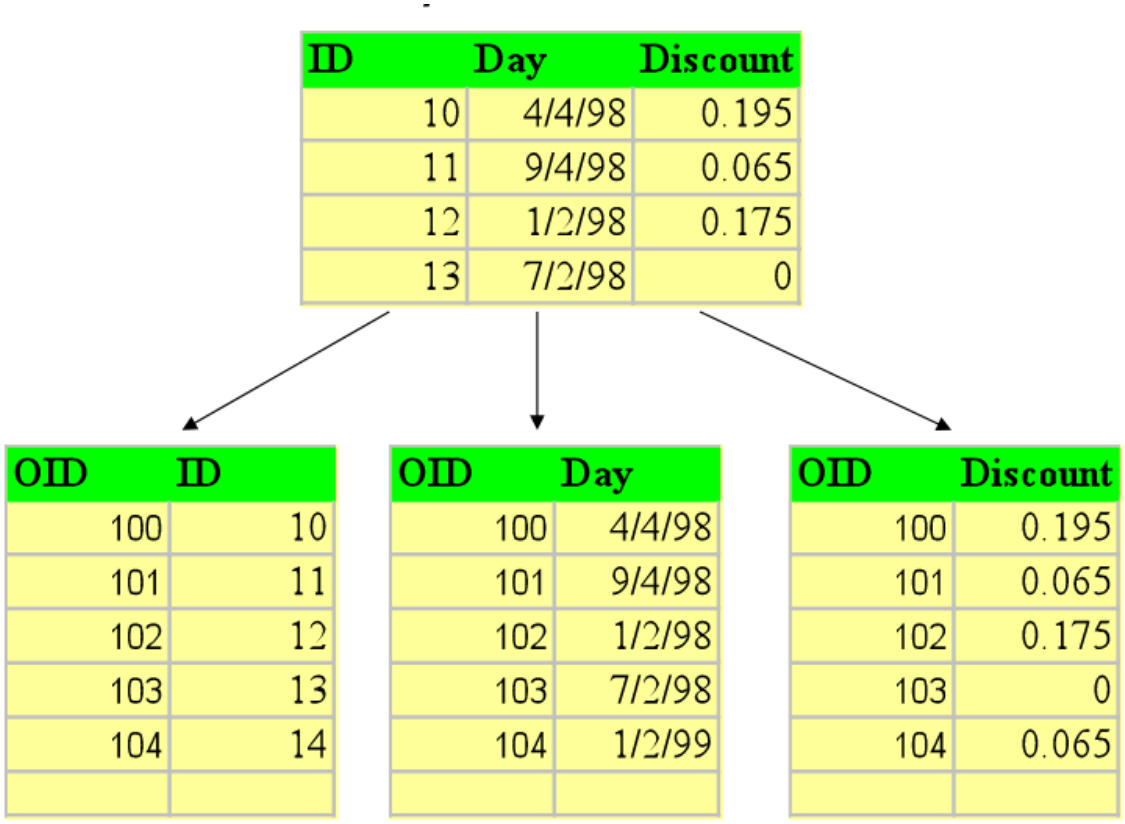
- **MonetDB** kernel is an abstract machine, programmed in the **MonetDB Assemblee Language (MAL)**.
- Each relational algebra operator corresponds to a **MAL instruction** (zero degrees of freedom).
- Each **BAT algebra operator** maps to a simple **MAL instruction**.

1.5 Software Stack

Three software layers:

- **FRONT-END Query language parser and a heuristic, language - and data model - specific optimizer**. **OUTPUT** -> logical plan expressed in MAL.
- **BACK-END Collection of optimizer modules** -> assembled into an optimization pipeline
- **MAL interpreter** -> contains the library of highly optimized implementation of the binary relational algebra operators.

2 Binary Association Tables



ID	Day	Discount
10	4/4/98	0.195
11	9/4/98	0.065
12	1/2/98	0.175
13	7/2/98	0

OID	ID
100	10
101	11
102	12
103	13
104	14

OID	Day
100	4/4/98
101	9/4/98
102	1/2/98
103	7/2/98
104	1/2/99

OID	Discount
100	0.195
101	0.065
102	0.175
103	0
104	0.065

3 MAL Reference (MonetDB Assembly Language)

- MAL program is considered a specification of intended computation and data flow behavior.
- Language syntax uses a functional style definition of actions and mark those that affect the flow explicitly.

3.1 Literals (follow the lexical conventions of C)

Hardwire Types	Temporal Types	IPv4 addresses and URLs
bit (bit)	date	inet
bte (byte)	daytime	url
chr (char)	time	UUID
wrd (word)	timestamp	json
sht (short)	-	-
int (integer)	-	-
lng (long)	-	-
oid (object id)	-	-
flt (float)	-	-
dbl (double)	-	-
str (string)	-	-

3.2 Variables

User Defined -> start with a letter **Temporary** -> start with X_ (generated internally by optimizers)

3.3 Instructions

One liners -> easy to parse

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

3.4 Type System

Strongly typed language

```
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;
```

- Polymorphic given by "any".
- Type checker (intelligent type resolution).

3.5 Flow of Control

For statement implementation:

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

If statement implementation:

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

3.6 Exceptions

(**To explore.**)

3.7 Functions

Function example

```
function user.helloWorld(msg:str):str;
    io.print(msg);
    msg:= "done";
    return msg;
end user.helloWorld;
```

Side Effects

- Functions can be pre-pended with the keyword unsafe.
- 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 -> order of execution should be guaranteed.
- Functions that return **:void** -> unsafe by default.

Inline Functions

- Functions prepended with the keyword **inline** are a target for the optimizers to be inlined. -> reduce the function call overhead.

3.8 MAL Syntax

Expressed in extended Backus–Naur form (EBNF) Wiki

Alternative constructors	(vertical bar) grouped by ()
Repetition	'+'-> at least once; '*'-> many
Lexical tokens	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]*
```

3.9 MAL Interpreter

3.10 MAL Debugger

3.11 MAL Profiler

3.12 MAL Optimizers

Triggered by experimentation and curiousity

- Alias Removal
- Building Blocks -> there are examples for a user to build a Optimizer
- Coercions Removes coercions that are not needed –> v:= calc.int(23); (sloppy code-generator or function call resolution decision)
- Common Subexpressions

```
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();
```

```
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 Expression Evaluation

```
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();
```

```
io.print(2);
io.print(2);
io.print(6);
io.print(6);
io.print(-0.279415488);
```

- Cost Model
- Data Flow Query executions without side effects can be rearranged.
- Garbage Collector
- Join Paths Looks up the MAL query and "composes" multiple joins. **algebra.join -> algebra.joinPath**
- Landscape
- Lifespans
- Macro Processing
- Memoization
- Multiplex Functions
- Remove Actions
- Stack Reduction

3.13 MAL Modules

4 MAL Algebra

Operation	MAL Cmd	C Cmd	Arguments/Return	Comment
GroupBy	groupby	ALGgroupby	gids :: bat-columntype:oid cnts :: bat-columntype:oid return :: bat-columntype:oid	Produces a new BAT with groups indentified by the head column. (The result contains tail times the head value, ie the tail contains the result group sizes.)
Find	find	ALGfind	b :: bat-columntype:any-1 t :: any-1 return :: oid	Returns the index position of a value. If no such BUN exists return OID-nil.
Fetch	fetch	ALGfetchoid	b :: bat-columntype:any-1 x :: oid return :: any-1	Returns the value of the BUN at x-th position with 0 <= x < b.count
Project	project	ALGprojecttail	b :: bat-columntype:any-1 v :: any-3 return :: bat-columntype:any-3	Fill the tail with a constant
Projection	projection	ALGprojection	left :: bat-columntype:oid rigth :: bat-columntype:any-3 return :: bat-columntype:any-3	Project left input onto right input.
Projection2	projection2	ALGprojection2	left :: bat-columntype:oid rigth1 :: bat-columntype:any-3 rigth2 :: bat-columntype:any-3 return :: bat-columntype:any-3	Project left input onto right inputs which should be consecutive.

BAT copying

Operation	MAL Cmd	C Cmd	Arguments/Return	Comment
Copy	copy	ALGcopy	b :: bat-columntype:any-1 return :: bat-columntype:any-1	Returns physical copy of a BAT.
Exist	exist	ALGexist	b :: bat-columntype:any-1 return :: bit	Returns whether 'val' occurs in b.

select ALGselect1 ALGselect2 ALGselect1nil ALGselect2nil
thetaselect ALGthetaselect1 ALGthetaselect2
selectNotNil ALGselectNotNil
sort ALGsort11 ALGsort12 ALGsort13 ALGsort21 ALGsort22 ALGsort23 ALGsort31 ALGsort32 ALGsort33
unique ALGunique2 ALGunique1
Join operations crossproduct ALGcrossproduct2
join ALGjoin ALGjoin1
leftjoing ALGleftjoin ALGleftjoin1
outerjoin ALGouterjoin
semijoin ALGsemijoin
thetajoin ALGthetajoin
band join ALGbandjoin
rangejoin ALGrangejoin
difference ALGdifference
intersect ALGintersect
Projection operations firstn ALGfirstn **reuse** ALGreuse

slice ALGslice$_{oid}$ ALGslice ALGslice$_{int}$ ALGslice$_{lng}$ **subslice** ALGsubslice$_{lng}$
Common BAT Aggregates
count ALGcount$_{bat}$ ALGcount$_{nil}$ **count$_{nonil}$** ALGcount$_{nonil}$
count ALGcountCND$_{bat}$ ALGcountCND$_{nil}$ **count$_{nonil}$** ALGcountCND$_{nonil}$

---

command select(b:bat[:any$_1$], low:any$_1$, high:any$_1$, li:bit, hi:bit, anti:bit) :bat[:oid] address ALGselect1 comment "Select all head values for which the tail value is in range. Input is a dense-headed BAT, output is a dense-headed BAT with in the tail the head value of the input BAT for which the tail value is between the values low and high (inclusive if li respectively hi is set). The output BAT is sorted on the tail value. If low or high is nil, the boundary is not considered (effectively - and

- infinity). If anti is set, the result is the complement. Nil

values in the tail are never matched, unless low=nil, high=nil, li=1, hi=1, anti=0. All non-nil values are returned if low=nil, high=nil, and li, hi are not both 1, or anti=1. Note that the output is suitable as second input for the other version of this function.";

command select(b:bat[:any$_1$], s:bat[:oid], low:any$_1$, high:any$_1$, li:bit, hi:bit, anti:bit) :bat[:oid] address ALGselect2 comment "Select all head values of the first input BAT for which the tail value is in range and for which the head value occurs in the tail of the second input BAT. The first input is a dense-headed BAT, the second input is a dense-headed BAT with sorted tail, output is a dense-headed BAT with in the tail the head value of the input BAT for which the tail value is between the values low and high (inclusive if li respectively hi is set). The output BAT is sorted on the tail value. If low or high is nil, the boundary is not considered (effectively - and + infinity). If anti is set, the result is the complement. Nil values in the tail are never matched, unless low=nil, high=nil, li=1, hi=1, anti=0. All non-nil values are returned if low=nil, high=nil, and li, hi are not both 1, or anti=1. Note that the output is suitable as second input for this function.";

command select(b:bat[:any$_1$], low:any$_1$, high:any$_1$, li:bit, hi:bit, anti:bit, unknown:bit) :bat[:oid] address ALGselect1nil comment "With unknown set, each nil != nil";

command select(b:bat[:any$_1$], s:bat[:oid], low:any$_1$, high:any$_1$, li:bit, hi:bit, anti:bit, unknown:bit) :bat[:oid] address ALGselect2nil comment "With unknown set, each nil != nil";

command thetaselect(b:bat[:any$_1$], val:any$_1$, op:str) :bat[:oid] address ALGthetaselect1 comment "Select all head values for which the tail value obeys the relation value OP VAL. Input is a dense-headed BAT, output is a dense-headed BAT with in the tail the head value of the input BAT for which the relationship holds. The output BAT is sorted on the tail value.";

command thetaselect(b:bat[:any$_1$], s:bat[:oid], val:any$_1$, op:str) :bat[:oid] address ALGthetaselect2 comment "Select all head values of the first input BAT for which the tail value obeys the relation value OP VAL and for which the head value occurs in the tail of the second input BAT. Input is a dense-headed BAT, output is a dense-headed BAT with in the tail the head value of the input BAT for which the relationship holds. The output BAT is sorted on the tail value.";

command selectNotNil(b:bat[:any$_2$]):bat[:any$_2$] address ALGselectNotNil comment "Select all not-nil values";

command sort(b:bat[:any$_1$], reverse:bit, nilslast:bit, stable:bit) :bat[:any$_1$] address ALGsort11 comment "Returns a copy of the BAT sorted on tail values. The order is descending if the reverse bit is set. This is a stable sort if the stable bit is set."; command sort(b:bat[:any$_1$], reverse:bit, nilslast:bit, stable:bit) (:bat[:any$_1$], :bat[:oid]) address ALGsort12 comment "Returns a copy of the BAT sorted on tail values and a BAT that specifies how the input was reordered. The order is descending if the reverse bit is set. This is a stable sort if the stable bit is set."; command sort(b:bat[:any$_1$], reverse:bit, nilslast:bit, stable:bit) (:bat[:any$_1$], :bat[:oid], :bat[:oid]) address ALGsort13 comment "Returns a copy of the BAT sorted on tail values, a BAT that specifies how the input was reordered, and a BAT with group information. The order is descending if the reverse bit is set. This is a stable sort if the stable bit is set."; command sort(b:bat[:any$_1$], o:bat[:oid], reverse:bit, nilslast:bit, stable:bit) :bat[:any$_1$] address ALGsort21 comment "Returns a copy of the BAT sorted on tail values. The order is descending if the reverse bit is set. This is a stable sort if the stable bit is set."; command sort(b:bat[:any$_1$], o:bat[:oid], reverse:bit, nilslast:bit, stable:bit) (:bat[:any$_1$], :bat[:oid]) address ALGsort22 comment "Returns a copy of the BAT sorted on tail values and a BAT that specifies how the input was reordered. The order is descending if the reverse bit is set. This is a stable sort if the stable bit is set."; command sort(b:bat[:any$_1$], o:bat[:oid], reverse:bit, nilslast:bit, stable:bit) (:bat[:any$_1$], :bat[:oid], :bat[:oid]) address ALGsort23 comment "Returns a copy of the BAT sorted on

tail values, a BAT that specifies how the input was reordered, and a BAT with group information. The order is descending if the reverse bit is set. This is a stable sort if the stable bit is set."; command sort(b:bat[:any$_1$], o:bat[:oid], g:bat[:oid], reverse:bit, nilslast:bit, stable:bit) :bat[:any$_1$] address ALGsort31 comment "Returns a copy of the BAT sorted on tail values. The order is descending if the reverse bit is set. This is a stable sort if the stable bit is set."; command sort(b:bat[:any$_1$], o:bat[:oid], g:bat[:oid], reverse:bit, nilslast:bit, stable:bit) (:bat[:any$_1$], :bat[:oid]) address ALGsort32 comment "Returns a copy of the BAT sorted on tail values and a BAT that specifies how the input was reordered. The order is descending if the reverse bit is set. This is a stable sort if the stable bit is set."; command sort(b:bat[:any$_1$], o:bat[:oid], g:bat[:oid], reverse:bit, nilslast:bit, stable:bit) (:bat[:any$_1$], :bat[:oid], :bat[:oid]) address ALGsort33 comment "Returns a copy of the BAT sorted on tail values, a BAT that specifies how the input was reordered, and a BAT with group information. The order is descending if the reverse bit is set. This is a stable sort if the stable bit is set.";

command unique(b:bat[:any$_1$], s:bat[:oid]) :bat[:oid] address ALGunique2 comment "Select all unique values from the tail of the first input. Input is a dense-headed BAT, the second input is a dense-headed BAT with sorted tail, output is a dense-headed BAT with in the tail the head value of the input BAT that was selected. The output BAT is sorted on the tail value. The second input BAT is a list of candidates."; command unique(b:bat[:any$_1$]) :bat[:oid] address ALGunique1 comment "Select all unique values from the tail of the input. Input is a dense-headed BAT, output is a dense-headed BAT with in the tail the head value of the input BAT that was selected. The output BAT is sorted on the tail value.";

command crossproduct(left:bat[:any$_1$], right:bat[:any$_2$], max$_{one}$:bit) (l:bat[:oid],r:bat[:oid]) address ALGcrossproduct2 comment "Returns 2 columns with all BUNs, consisting of the head-oids from 'left' and 'right' for which there are BUNs in 'left' and 'right' with equal tails";

command join(l:bat[:any$_1$],r:bat[:any$_1$],sl:bat[:oid],sr:bat[:oid],nil$_{matches}$:bit,estimate:lng) (:bat[:oid],:bat[:oid]) address ALGjoin comment "Join";

command join(l:bat[:any$_1$],r:bat[:any$_1$],sl:bat[:oid],sr:bat[:oid],nil$_{matches}$:bit,estimate:lng) :bat[:oid] address ALGjoin1 comment "Join; only produce left output";

command leftjoin(l:bat[:any$_1$],r:bat[:any$_1$],sl:bat[:oid],sr:bat[:oid],nil$_{matches}$:bit,estimate:lng) (:bat[:oid],:bat[:oid]) address ALGleftjoin comment "Left join with candidate lists";

command leftjoin(l:bat[:any$_1$],r:bat[:any$_1$],sl:bat[:oid],sr:bat[:oid],nil$_{matches}$:bit,estimate:lng) :bat[:oid] address ALGleftjoin1 comment "Left join with candidate lists; only produce left output";

command outerjoin(l:bat[:any$_1$],r:bat[:any$_1$],sl:bat[:oid],sr:bat[:oid],nil$_{matches}$:bit,estimate:lng) (:bat[:oid],:bat[:oid]) address ALGouterjoin comment "Left outer join with candidate lists";

command semijoin(l:bat[:any$_1$],r:bat[:any$_1$],sl:bat[:oid],sr:bat[:oid],nil$_{matches}$:bit,max$_{one}$:bit,estimate:lng) (:bat[:oid],:bat[address ALGsemijoin comment "Semi join with candidate lists";

command thetajoin(l:bat[:any$_1$],r:bat[:any$_1$],sl:bat[:oid],sr:bat[:oid],op:int,nil$_{matches}$:bit,estimate:lng) (:bat[:oid],:bat[:oid address ALGthetajoin comment "Theta join with candidate lists";

command bandjoin(l:bat[:any$_1$],r:bat[:any$_1$],sl:bat[:oid],sr:bat[:oid],c1:any$_{1,c2}$:any$_{1,li}$:bit,hi:bit,estimate:lng) (:bat[:oid],:b address ALGbandjoin comment "Band join: values in l and r match if r - c1 <[=] l <[=] r + c2";

command rangejoin(l:bat[:any$_1$],r1:bat[:any$_1$],r2:bat[:any$_1$],sl:bat[:oid],sr:bat[:oid],li:bit,hi:bit,anti:bit,symmetric:bit,esti (:bat[:oid],:bat[:oid]) address ALGrangejoin comment "Range join: values in l and r1/r2 match if r1 <[=] l <[=] r2";

command difference(l:bat[:any$_1$],r:bat[:any$_1$],sl:bat[:oid],sr:bat[:oid],nil$_{matches}$:bit,nil$_{clears}$:bit,estimate:lng) :bat[:oid] address ALGdifference comment "Difference of l and r with candidate lists";

command intersect(l:bat[:any$_1$],r:bat[:any$_1$],sl:bat[:oid],sr:bat[:oid],nil$_{matches}$:bit,max$_{one}$:bit,estimate:lng) :bat[:oid] address ALGintersect comment "Intersection of l and r with candidate lists (i.e. half of semi-join)";

pattern firstn(b:bat[:any], n:lng, asc:bit, nilslast:bit, distinct:bit) :bat[:oid] address ALGfirstn comment "Calculate first N values of B"; pattern firstn(b:bat[:any], s:bat[:oid], n:lng, asc:bit, nilslast:bit, distinct:bit) :bat[:oid] address ALGfirstn comment "Calculate first N values of B with candidate list S"; pattern firstn(b:bat[:any], s:bat[:oid], g:bat[:oid], n:lng, asc:bit, nilslast:bit, distinct:bit) :bat[:oid] address ALGfirstn comment "Calculate first N values of B with candidate list S"; pattern firstn(b:bat[:any], n:lng, asc:bit, nilslast:bit, distinct:bit) (:bat[:oid],:bat[:oid]) address ALGfirstn comment "Calculate first N values of B"; pattern firstn(b:bat[:any], s:bat[:oid], n:lng, asc:bit, nilslast:bit, distinct:bit) (:bat[:oid],:bat[:oid]) address ALGfirstn comment "Calculate first N values of B with candidate list S"; pattern firstn(b:bat[:any], s:bat[:oid], g:bat[:oid], n:lng, asc:bit, nilslast:bit, distinct:bit) (:bat[:oid],:bat[:oid]) address ALGfirstn comment "Calculate first N values of B with candidate list S";

command reuse(b:bat[:any$_1$]):bat[:any$_1$] address ALGreuse comment "Reuse a temporary BAT if you can. Otherwise, allocate enough storage to accept result of an operation (not involving the heap)";

command slice(b:bat[:any$_1$], x:oid, y:oid) :bat[:any$_1$] address ALGslice$_{oid}$ comment "Return the slice based on head oid x till y (exclusive).";

command slice(b:bat[:any$_1$], x:lng, y:lng) :bat[:any$_1$] address ALGslice comment "Return the slice with the BUNs at position x till y.";

command slice(b:bat[:any$_1$], x:int, y:int) :bat[:any$_1$] address ALGslice$_{int}$ comment "Return the slice with the BUNs at position x till y.";

command slice(b:bat[:any$_1$], x:lng, y:lng) :bat[:any$_1$] address ALGslice$_{lng}$ comment "Return the slice with the BUNs at position x till y.";

command subslice(b:bat[:any$_1$], x:lng, y:lng) :bat[:oid] address ALGsubslice$_{lng}$ comment "Return the oids of the slice with the BUNs at position x till y.";

module aggr;

command count(b:bat[:any]) :lng address ALGcount$_{bat}$ comment "Return the current size (in number of elements) in a BAT."; command count (b:bat[:any], ignore$_{nils}$:bit) :lng address ALGcount$_{nil}$ comment "Return the number of elements currently in a BAT ignores BUNs with nil-tail iff ignore$_{nils}$==TRUE."; command count$_{nonil}$ (b:bat[:any$_2$]) :lng address ALGcount$_{nonil}$ comment "Return the number of elements currently in a BAT ignoring BUNs with nil-tail";

command count(b:bat[:any], cnd:bat[:oid]) :lng address ALGcountCND$_{bat}$ comment "Return the current size (in number of elements) in a BAT."; command count (b:bat[:any], cnd:bat[:oid], ignore$_{nils}$:bit) :lng address ALGcountCND$_{nil}$ comment "Return the number of elements currently in a BAT ignores BUNs with nil-tail iff ignore$_{nils}$==TRUE."; command count$_{nonil}$ (b:bat[:any$_2$], cnd:bat[:oid]) :lng address ALGcountCND$_{nonil}$ comment "Return the number of elements currently in a BAT ignoring BUNs with nil-tail";

command cardinality(b:bat[:any$_2$]) :lng address ALGcard comment "Return the cardinality of the BAT tail values.";

#SQL uses variable head types command min(b:bat[:any$_2$]):any$_2$ address ALGminany comment "Return the lowest tail value or nil.";

command min(b:bat[:any$_2$], skipnil:bit):any$_2$ address ALGminany$_{skipnil}$ comment "Return the lowest tail value or nil.";

command max(b:bat[:any$_2$]):any$_2$ address ALGmaxany comment "Return the highest tail value or nil.";

command max(b:bat[:any$_2$], skipnil:bit):any$_2$ address ALGmaxany$_{skipnil}$ comment "Return the highest tail value or nil.";

pattern avg(b:bat[:any$_2$]) :dbl address CMDcalcavg comment "Gives the avg of all tail values";

pattern avg(b:bat[:any$_2$], scale:int) :dbl address CMDcalcavg comment "Gives the avg of all tail values";

command stdev(b:bat[:any$_2$]) :dbl address ALGstdev comment "Gives the standard deviation of all tail values"; command stdevp(b:bat[:any$_2$]) :dbl address ALGstdevp comment "Gives the standard deviation of all tail values"; command variance(b:bat[:any$_2$]) :dbl address ALGvariance comment "Gives the variance of all tail values"; command variancep(b:bat[:any$_2$]) :dbl address ALGvariancep comment "Gives the variance of all tail values";

command covariance(b1:bat[:any$_2$],b2:bat[:any$_2$]) :dbl address ALGcovariance comment "Gives the covariance of all tail values"; command covariancep(b1:bat[:any$_2$],b2:bat[:any$_2$]) :dbl address ALGcovariancep comment "Gives the covariance of all tail values"; command corr(b1:bat[:any$_2$],b2:bat[:any$_2$]) :dbl address ALGcorr comment "Gives the correlation of all tail values";