Monet; a next-Generation DBMS Kernel For Query-Intensive Applications

Boncz, P.A.

Citation for published version (APA):
Chapter 4

The MIL Language

Monet was designed to work in a front-end/back-end system architecture, in order to reach its design goals of extensibility and support for multiple logical data models. It can be seen as the back-end that provides a kernel of DBMS facilities to multiple front-ends (Figure 4.1). A relational front-end maps SQL queries onto Monet requests. An ODMG front-end does the same for OQL and object access. Other front-ends, targeted to specific applications such as data-mining, may co-exist.

Front-end systems built on top of Monet, such as a relational or an object-oriented DBMS, communicate with the Monet back-end through the MIL language. This does not mean to say that MIL itself is an object-oriented or even a relational language; MIL just provides the minimal, but complete set of primitives, such that each front-end can adequately map operations on its logical model to the underlying Monet primitives. Adequately here means that our objectives for the design of Monet (performance and extensibility) should not be compromised.

Figure 4.1: The front-end/back-end architecture.

Figure 4.2 shows the general structure of MIL. The language consists of a number of control structures (the black area) and extensibility features (the concentric areas), and has the following characteristics:
• it provides all DBMS services needed by the front-ends.
• it uses one simple bulk data type, the binary table.
• its table manipulation operations form a closed algebra on this binary table model.
• it provides constructs to express various kinds of parallelism.
• it is extensible with new primitives, data types, and associated search accelerator structures.
• it is a computationally complete procedural language.

In the design of MIL, we applied valuable lessons learned in previous work on database languages. The idea of using query algebras as intermediate languages for relational query execution dates back to [Cod70]. In the context of extensible relational systems, this idea was generalized to allow modular extensibility using ADT interfaces. The extensibility interface of MIL was inspired by Gral [Güt89], an early system that offered extensibility on all relevant levels (data types, algebra operators and search accelerators). The Fad [DV92] language is well-known for its consequent functional operator style. The expressive genericness of this language proved a hindrance to run Fad programs efficiently on bulk data and perform parallelization [FGN+95]. Its successor language, Flora, used as an intermediate language in the IDEA [LVZZ94] system, solved this problem by focusing on a simple kernel of bulk operators. This decision we also followed in MIL, as well as the decision in Flora to introduce explicit language primitives for specifying parallelism.

4.1 Data Model

The MIL data model consists of an extensible set of atomic values, and one collection type, the BAT (Binary Association Table). The formal definition of the set of all types \( \mathcal{T} \) in MIL is:

1. \( t \in \mathcal{A}_f \cup \mathcal{A}_v \Rightarrow t \in \mathcal{T} \)
2. \( T_1, T_2 \in \mathcal{T} \Rightarrow \text{bat}[T_1, T_2] \in \mathcal{T} \)

The first rule defines atomic data types \( \mathcal{A} \) (both of fixed and variable size), and the latter defines the BAT type. A BAT value is a multi-set that contains binary tuples, called Binary UNits (BUNs). The left column of a BAT is called the head column and the right is called the tail column. The \( \text{bat}[T_1, T_2] \) type is parametrized by the types of its head and tail columns, and may be nested, as those types might again be BATs.

As a starting point, we have the collection of fixed-size atoms \( \mathcal{A}_f = \{ \text{bit}, \text{chr}, \text{sht}, \text{int}, \text{lng}, \text{flt}, \text{dbl}, \text{oid} \} \), respectively denoting boolean values, single characters, small, normal and long integers, normal and double floating point numbers, and object identifiers. The standard collection of variable-sized atoms \( \mathcal{A}_v = \{ \text{str} \} \) just contains the string type.

This initial set of atomic types is focused on supporting standard business applications, but the MIL data model can be extended with new atomic types. We have
implemented many new atomic types and operations on them. These types encompass enrichments in the business area (like currency and temporal types) [BWK98], the GIS domain (points, polylines, polygons) [BQK96] and multi-media (images, video, audio) [NK98].

The MIL syntax for values of the standard atomic types follows that of the C/C++ programming languages. Values can be cast to another type with conversion functions \( t(\text{value}) \), \( t \in T \), that implicitly exist for each atomic type. Casting is necessary to distinguish longs from integers and doubles from floats (e.g., \( \text{lng}(42) \), \( \text{dbl}(3.14) \)). The bit type has two values, denoted true and false.

Each type has one additional special value, called nil, that expresses the “don’t know” value. We use the nil as a shorthand for the oid(nil), as this value is often used. For nil values of other types, we use casts (e.g., \( \text{bit(nil)} \), \( \text{int(nil)} \), \( \text{dbl(nil)} \)).

### 4.1.1 Example Data Mapping

Suppose that we have a relational schema with tables Order and Item in which the attribute order identifies the order to which the item belongs:

```plaintext
| table Order(int id; date day; float discount); |
| table Item(int order; float price; float tax); |
```

A relational data model can be stored in Monet by splitting each relational table by column[CK85]. Each column becomes a BAT that holds the column values in its tail (right column). The head (left column) holds an object identifier (oid). We use the naming convention `table-name_column-name` for such BATs. The relational tuples can be reconstructed by taking all tail values of the column BATs with the same head oid.

This mapping scheme decomposes our ORDER table into order_id, order_day and order_discount, and the ITEM table into item_order, item_price and item_tax. Sometimes it may be possible to use one of the unique columns as the head column in the
4.2 MIL Execution

MIL is a procedural block-structured language, with standard control structures like if-then-else, and while-loops. The BAT *iterator*, denoted **bat-exp@iterator**, provides another way of looping. This cursor-like construct visits elements (BUNs) from a BAT, and for each element executes a MIL statement. This statement can contain the special variables $\$h$ and $\$t$, that represent the head- and tail- value of the current element, respectively. The most commonly used iterator in MIL is the *batloop*, that sequentially visits all elements of a BAT.

The basic unit of MIL execution is the *operator*. MIL operators receive a number of input values and produce a single output value. All operators can be called like `op(expr1, ..., exprN)` but MIL also allows infix notation (`expr1 op expr2`) for binary operators as well as object-oriented dot-notation `expr1.op(expr2, ..., exprN).

Multiple operators with the same name, but with different *signatures* may exist (overloading). An operator signature consists of the operator name, followed by a comma-separated list of parameters between parentheses, a colon and then the return type. Each parameter definition consists of the parameter type and a parameter name. Operators may have a variable number of parameters. In the signature, such parameters are denoted `... type-expr...`.

Most operators are *polymorphic*, which means that their signature contains (free) type-variables, denoted in this document with capital single-letter italics.\(^1\)

\(^{1}\)In MIL we use the syntax **any::<tag>** for free type variables. When such free variables have the same tag number, this implies that the types of the actual parameters passed must match each other.
4.2. MIL EXECUTION

MIL is a dynamically typed language, so function resolution is a run-time task. The execution mode of operators is to simply first interpret all parameters and materialize their results. If an operator exists with a signature that matches these actual parameter, it is invoked (else a run-time error occurs).

MIL also accepts the C/C++ shorthand for assignment combined with operator execution (e.g., \( i := 1 \) means \( i := i+1 \)). In order to provide flexible execution of variable functions, a string name of an operator can be \textit{dereferenced} to call it as in \((\texttt{str-expr})(\texttt{expr1}, \ldots, \texttt{exprN})\).

MIL \textit{extension modules} introduce new atomic data types, operators, iterators, and search accelerators (Figure 4.2). Search accelerators are data structures related to BAT columns, that are maintained under updates by the system. They do not introduce semantics, but are generally used to accelerate execution of certain operators (e.g., a hash table accelerates equi-selections and equi-joins). The core of the language is introduced by the standard module collection, which database extenders can augment with their own. Extension modules are implemented in C/C++. Alternatively, new operators can be defined run-time in MIL as scripted \textit{procedures}:

\begin{verbatim}
proc operator-signature <MIL statement>

The statement is called the \textit{body} of the procedure. When a procedure is executed, its body – in which the formal parameters from the signature have their scope – is interpreted. A value can be returned from a procedure body with the \texttt{return} keyword.

As procedures may be defined to receive a variable number of arguments, the MIL body can access those with the special construct $\$(\texttt{int-expr})$, that retrieves the \(i\)-th parameter. The actual number of parameters is obtained with \$0. All arguments from the \(i\)-th until the last can be passed into another operator that with the $\$(\texttt{int-expr}..)$ construct.

As an example, let us define the \texttt{avg}(T, \ldots T.):T. operator that averages its arguments, using a procedure:

\begin{verbatim}
proc avg(any:: l  v , \ldots any::l.. ) : any:: l  { 
    var i := 1;
    while( i <= $0 )
        v :=+ $\{(i :=+ 1)\};
    return v / $0;
}
\end{verbatim}

Though procedures are interpreted instead of executed as a C/C++ function – like normal MIL operators – this distinction is hidden on the language level, and procedures can be used like any other operator.

This extensibility mechanism is comparable to extension mechanisms used in relational systems like [SAH87, OHUS96] and differs from [Ora97a] in its choice to run extension code directly in the DBMS process space, as performance is a primary concern in Monet.

4.2.1 Atomic Value Operators

The minimal set of operators \(-,\neq,\lt,\gt,\leq,\geq,\texttt{hash}()\) is present on all atomic types.\footnote{The \texttt{hash(any)}:\texttt{int} returns an \texttt{int} 32-bit hash-number.}

Each atomic type brings with it an additional interface of specific operations:
CHAPTER 4. THE MIL LANGUAGE

**Numerical Types**

The **sht, int, fit, db1** and **lng** have arithmetic operators (+, −, *, /), as well as the `between(value, low, high)` bit that checks whether low ≤ value ≤ high.

**Floating Point**

The **fit** and **db1** have math operators `cos`, `sin`, `tan`, etc.

**Strings**

Have a series of (sub)string and matching operations.

**Object Identifiers**

The `newoid(int size):oid` operator requests a system-wide unique range of fresh oid-s. The function returns the start value of this consecutive range.

**Booleans**

The bit type and or, not operators defined on it.

Note that the convention for all MIL operators is to respect the “don’t know” semantics of the `nil` value. For example, `int(nil)*2` and `int(nil)>10` evaluate to `int(nil)` and `bit(nil)`, respectively.

### 4.2.2 BAT Algebra

The focus of MIL execution is to enable efficient bulk operations on mass data stored in BATs. This core functionality is offered by a **BAT algebra** of MIL operators. These operators

- have an algebraic definition, as provided below.
- are free of side-effects, which makes the algebra apt as a language for optimization with rewrite systems.
- form a closed algebra with BATs as the only collection-type.

We formally define the semantics of the BAT algebra operators using tables that show in their left column the *operator signature*, and in the right column a definition of its result. Just like in C++, if trailing parameters in a signature have a specified default value, when one omits these trailing parameters when an calling an operator, their default values are used. The notation of a BUN is `[a, b]`. We denote BATs as bags `{}`, and if we know that no double elements will occur as sets: `{}`. In the case of bags, the notation is pragmatic but somewhat loose, as we use set-like definitions `|x| < cartesianproduct :< condition >`. With the special semantics that the bag consists of those `x` corresponding to each element in the cartesian product for which the condition holds. Also, in some of these bag-definitions the `∪` and `\` are used, in which cases the loss-less bag-union resp. the bag-minus are meant. `|S|` indicates the size of a collection. BATs can always be seen as a list of BUNs, that is, with some **BUN-ordering**. Only a few MIL algebra operators define such an ordering, so for most BATs we just do not know anything about their BUN-ordering. The notation of a BAT as a certain BUN-list `B = [[h₁, t₁], ..., [h|B|, t|B|]]` we shorten as `B = [B]`.

<table>
<thead>
<tr>
<th>MIL operator signature</th>
<th>Definition of the Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>find(bat[H,T] AB, H a) → T</code></td>
<td><code>b if \[a,b] \in AB, else T(nil)</code></td>
</tr>
<tr>
<td><code>select(bat[H,T] AB, Tv₁, Tv₂ = v₁, (a,b) \in AB : bit r₁ = true, r₂ = r₁)</code></td>
<td><code>(((isnil(v₁) \lor v₁ &lt; b) \land (isnil(v₂) \lor v₂ &lt; b) \lor (isnil(v₁) \land isnil(v₂)))</code></td>
</tr>
<tr>
<td></td>
<td><code>\lor (r₁ \land v₁ = b) \lor (r₂ \land v₂ = b) \lor (isnil(v₁) \land r₁ \land isnil(v₂) \land r₂ \land isnil(b))</code></td>
</tr>
</tbody>
</table>

*Range and equi-selections*
4.2. MIL EXECUTION

The select supports the selection predicates \( \text{isnil}(t,v) \), \( t \leq v \), \( t < v \), \( t = v \), \( t > v \), and \( t \geq v \) as well as \( v_1 \leq t \leq v_2 \), \( v_1 < t \leq v_2 \), \( v_1 \leq t < v_2 \), and \( v_1 < t < v_2 \):

- `select(b,nil);`  # select on isnil(tail(b))
- `select(b,42);`  # select on tail(b) = 42
- `select(b,10,20);`  # select on 10 <= tail(b) <= 42
- `select(b,10,20,false,true);`  # select on 10 < tail(b) < 42
- `select(b,10,20,true,false);`  # select on 10 <= tail(b) < 42
- `select(b,10,20,false,false);`  # select on 10 < tail(b) < 42
- `select(b,10,nil);`  # select on 10 <= tail(b)
- `select(b,10,nil,false);`  # select on 10 < tail(b)
- `select(b,11,20);`  # select on tail(b) <= 20
- `select(b,11,20,false,false);`  # select on tail(b) < 20

The BAT algebra is closed on the BAT type, so the result of the join is again a binary table. This is achieved by projecting out the join columns; the result consists of the outer columns of the left and right BAT where their inner columns matched.

<table>
<thead>
<tr>
<th>MIL operator signature</th>
<th>Definition of the Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>join(bat[T0,T1]AB,bat[T1,T2]CD)</code></td>
<td>{ [[a,d]\mid [a,b]\in AB,\ [c,d]\in CD;\ b=c} }</td>
</tr>
<tr>
<td><code>outerjoin(bat[T0,T1]AB,bat[T1,T2]CD)</code></td>
<td>{ [[a,nil]\mid [a,b]\in AB;\ [b,c]\in CD;\ b=c} }</td>
</tr>
</tbody>
</table>

Figure 4.4 provides a flavor of MIL execution in the example from Section 4.1.1. The depicted sequence of MIL statements retrieves all item-IDs from orders with a certain discount, and the tax paid over them. As Monet never materializes N-ary tables, the result of our query is a relational table that is again decomposed, in the BATs UNQ.ITH and UNQ.TOT. We use the standard `printf(str format, ...)` printing operator for printing results to standard output. This operator is part of the (standard) IO extension module.

**SQL example query**

```sql
SELECT item.id AS id,
       item.price*item.tax AS total
WHERE order.id = item.order AND
      order.discount BETWEEN 0.00 AND 0.06
```

**MIL translation (annotated below)**

```mil
ORD.DSC := select(order.discount,0.0,0.06)
IDS.NIL := join(order_id.reverse,ORD.DSC)
creates a bat[oid,oid] with selected order-IDs in head, nil tail
ITM.NIL := join(item.order,IDS.NIL)
creates a bat[oid,oid] with selected item-IDs in head, nil tail
UNQ.ITH := mark(ITM.NIL).reverse
creates a bat[oid,oid] fresh oids in head, selected item-IDs in tail
UNQ.PRI := join(UNQ.ITH,item.price)
creates a bat[oid,flt] with selected item-IDs and their prices
UNQ.TAX := join(UNQ.ITH,item.tax)
creates a bat[oid,flt] with selected item-IDs and their taxes
UNQ.TOT := [*] (UNQ.PRI,UNQ.TAX)
creates a bat[oid,flt] with selected item-IDs and totals
[printf("\%d \%d\n", UNQ.ITH, UNQ.TOT)]
```

Figure 4.4: A simple SQL query and a MIL translation.

Much like in the definition of the select, which selects on tail, the operators in the BAT algebra have fixed semantics on which columns of their BAT parameters they work, and from which columns result values are derived. If an operator needs to work on the opposite column of some BAT, MIL allows to view each BAT with head and tail
column swapped. This reverse view on a BAT is delivered by the reverse operator. The mirror allows to view a BAT as if it had its head column superimposed on the tail column, yielding a BAT with two identical columns. The rationale to have these column-swapping operators is that it allows the other BAT algebra operators to make fixed assumptions about their parameter formats - thus reducing the degree of freedom in the operator signatures and simplifying their implementation and optimization.

<table>
<thead>
<tr>
<th>MIL operator signature</th>
<th>Definition of the Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>reverse(bat[H,T] AB)→bat[T,H]</td>
<td>${AB[i=1[a_i,b_i]]}$, $AB = {AB[i=1[a_i,b_i]]}$</td>
</tr>
<tr>
<td>mirror(bat[H,H] AB)→bat[H,H]</td>
<td>${AB[i=1[a_i,a_i]]}$, $AB = {AB[i=1[a_i,b_i]]}$</td>
</tr>
<tr>
<td>mark(bat[H,T] AB, oid o = 0)→bat[H,oid]</td>
<td>${AB[i=1[a_i,a_i+1]]}$, $AB = {AB[i=1[a_i,b_i]]}$</td>
</tr>
<tr>
<td>project(bat[H,T1] AB, T2c)→bat[H, T2]</td>
<td>${AB[i=1[a_i,c]]}$, $AB = {AB[i=1[a_i,b_i]]}$</td>
</tr>
<tr>
<td>slice(bat[H,T] AB int lo, hi)→bat[H,T]</td>
<td>$\min(AB[lo+hi], a_i, b_i)$, $AB = {AB[i=1[a_i,b_i]]}$</td>
</tr>
<tr>
<td>order(bat[H,T] AB)→bat[H,T]</td>
<td>${AB[k=1[h_i,k_i]]} = {{a_i,b_i}</td>
</tr>
<tr>
<td>order(bat[H,T1] AB, bat[H,T2] AC)→bat[H,oid]</td>
<td>${AB[k=1[h_i,k_i]]} = {{a_i,b_i}</td>
</tr>
</tbody>
</table>

MIL operators with list-semantics, using the below definitions:

<table>
<thead>
<tr>
<th>id(z) respects $=<em>{n,\leq</em>{n}}$</th>
<th>$id(a,b) = id(c,d) \Leftrightarrow (a,b) = (c,d)$, $id(a,b) \leq id(c,d) \Leftrightarrow (a,b) \leq (c,d)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>lexicographic $=<em>{n,\leq</em>{n}}$</td>
<td>$\leq_{n,\leq_{n}}$</td>
</tr>
<tr>
<td>nil-proof $=<em>{n,\leq</em>{n}}$</td>
<td>$a =<em>{n,b} \Leftrightarrow (a =</em>{n,b}) \lor (\text{isnil}(a) \land \text{isnil}(b))$, $a \leq_{n,b} \Leftrightarrow (a \leq_{n,b}) \lor \text{isnil}(a)$</td>
</tr>
</tbody>
</table>

The mark also introduces a new tail column, but fills it with an ascending range of oid-s that starts with the second parameter value. Note that nil+nil=nil, so passing the nil value as second parameter will yield a BAT with nil in the entire tail column. This same effect can be reached for any constant value with the project. The mark is often used for introducing a new column of unique oid-s. As we saw in Section 4.1.1, such unique columns are used in MIL to couple BATs that represent a decomposed table. This not only goes for persistent tables, but also for intermediate results of query processing, as those can be viewed as temporary tables. In line 4 of Figure 4.4, the result of the join from line 2 gets a new unique column using the mark. This unique column is present in all tempsoraries of Figure 4.4 whose name starts with UNQ.

The slice returns a positionally selected subset of its parameter BAT. If the BUN-ordering of this BAT is unknown, the result can be considered a random selection without putting back. The unary order returns a bag-wise identical BAT as its only parameter, but guarantees that the tail column is ordered. The binary order is much similar to the binary group discussed later, but guarantees that the generated oid-s tail column is ordered and respects the lexicographical $\leq_{n}$ ordering on tail value combinations (See Section 7.3.5 for a detailed discussion how this operator facilitates multi-column ORDER BY).

By careful design of the BAT data structures (see Section 5.3.2), the reverse, mirror, slice and mark actually do not have to materialize their results, which makes them zero-cost operators.

<table>
<thead>
<tr>
<th>MIL operator signature</th>
<th>Definition of the Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>count(bat[H,T] AB)→int</td>
<td>$</td>
</tr>
<tr>
<td>sum(bat[H,T0] AB)→T1</td>
<td>$\sum_{i=1}^{n} b$</td>
</tr>
<tr>
<td>max(bat[H,T] AB)→T</td>
<td>$b =</td>
</tr>
<tr>
<td>min(bat[H,T] AB)→T</td>
<td>$b =</td>
</tr>
</tbody>
</table>

aggregates; on boundary condition $AB = \emptyset$: count, sum return zero, others return $t(\text{nil})$.
The above collection of aggregates is by no means complete. Extension modules with new ones can be introduced easily in MIL.

### MIL operator signature

<table>
<thead>
<tr>
<th>MIL operator signature</th>
<th>Definition of the Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>unique(bat[H,T] (AB)) (-)) bat[H,T]</td>
<td>{ (a, b)</td>
</tr>
<tr>
<td>diff(bat[H,T] (AB), bat[H,T] (CD)) (-)) bat[H,T]</td>
<td>{ (c, d)</td>
</tr>
<tr>
<td>union(bat[H,T] (AB), bat[H,T] (CD)) (-)) bat[H,T]</td>
<td>{ (a, b)</td>
</tr>
<tr>
<td>intersect(bat[H,T] (AB), bat[H,T] (CD)) (-)) bat[H,T]</td>
<td>{ (a, b)</td>
</tr>
</tbody>
</table>

The classical operations on sets, formed by the BUNs of a BAT, are displayed above. If only one column of the parameter BATs is of interest, one should first make the other column constant (with mark(mill)) or equal (with mirror).

### MIL operator signature

<table>
<thead>
<tr>
<th>MIL operator signature</th>
<th>Definition of the Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>group(bat[oid,T] (AB)) (-)) bat[oid,oid]</td>
<td>{ (a, \text{id}(b))</td>
</tr>
<tr>
<td>group(bat[oid,oid] (AB), bat[oid,T] (AC)) (-)) bat[oid,oid]</td>
<td>{ (a, \text{id}(b, c))</td>
</tr>
</tbody>
</table>

Relational group by, or object-oriented nest/unnest, require specific support on the flat binary algebra. Such groupings may involve multiple attributes. In MIL, groupings are materialized in a grouping BAT that holds in the head column identifiers of all objects of interest, and in the tail a unique group identifier. The group operators construct such grouping-BATs.

The unary group operator is executed on a first bat[oid,any] and creates a new equivalence group for each different value from the tail column. The result is formed by a BAT with the same head column as the input, with a group-id in the tail column for each BUN. These group-id’s from a dense collection of oids \([0,1,\ldots,N-1]\), with \(N\) the number of distinct tail values in the original BAT. Such groupings can be refined using the binary group operator that subdivides the groups into new equivalence subgroups, taking into account an additional bat[oid,any].

### MIL operator signature

<table>
<thead>
<tr>
<th>MIL operator signature</th>
<th>Definition of the Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>fragment(bat[H,T] (AB), bat[H,H] (CD)) (-)) bat[H,T]</td>
<td>{ (h, \text{select}(AB, l, h)) { (l, h) (\in CD) }</td>
</tr>
<tr>
<td>split(bat[H,T] (AB), int (n)ranges) (-)) bat[H,H]</td>
<td>{ (l, h) (\exists l \leq h) (\exists x, (x, h, y) \in AB) }</td>
</tr>
</tbody>
</table>

Relation: \(S = \text{split}(AB, n)\) creates range-partition \(\forall_{(a,b) \in AB}: \exists_{\text{unique},l,h} \{ l \leq a \leq h \} \)

The MIL data model supports nested BATs, as produced by the fragmentation operator fragment. This operator performs a range-fragmentation of a BAT according to the head column. The range-BAT containing the split boundaries – that is passed as a second parameter – can be produced with the split operator. The ranges is only a target; the actual number of ranges depends on the distribution of the values in the head column of \(AB\).

### 4.2.3 Operator Constructors

The \(\{f\}\) and \([f]\) are special MIL syntax constructs that implicitly define a new operator for each already defined operator \(f\).

### MIL operator signature

<table>
<thead>
<tr>
<th>MIL operator signature</th>
<th>Definition of the Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(f)(bat[H,T] (AB_1,\ldots,\text{bat[H,T],} n) (AB_n)) (-)) bat[H,T]</td>
<td>{ (a_1, f(b_1,\ldots,b_n)) }</td>
</tr>
</tbody>
</table>
| multi-join map of operator \(f(T_1,\ldots,T_n) \rightarrow T_r \in \{-,\neq,\lt,\leq,\gt,\geq,+,\cdot,/,\text{and,or,not,}\} \) | }
The multi-join map constructs an operator that does an implicit equi-join on the head columns of multiple BATs and executes the operator that was passed between the square brackets on the result of this join (all matching combinations of tail values). The result of the multi-join map is again a BAT, that contains the head value for each match and in the tail the result of the corresponding operator execution.

The multi-join map of the \( \{f \} (BD, BX) \) operator was demonstrated in line 6 of Figure 4.4. It produces a new BAT with for all selected items the multiplied price and tax.

Though not shown in the definition, we can also type the \texttt{const} keyword in front of an actual parameter and pass any kind of value (not necessarily a BAT) into the map operator. In this case, that parameter is not taken into the multi-join, and this value is passed as a constant into all operator executions. For example, \( \{*, item\_tax, const 0.07 \} \) multiplies all prices by .07. Typing \texttt{const} is actually not necessary for non-BAT values; e.g., \( \{*, item\_tax, 0.07 \} \) will do as well. In order to keep the multi-join map meaningful, at least one parameter should still be a (non-\texttt{const}) BAT.

<table>
<thead>
<tr>
<th>MIL operator signature</th>
<th>Definition of the Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>( {f}(bat[T_0,T_1] \ AB \ bat[T_2,T_2] \ CB \ bat[T_2,T_2] \ CD, \ bat[T_2,.bit] \ CE=CB, \ project(true).\ unique) ) ( \rightarrow ) ( T_4 \in { ) \texttt{count, min, max, sum}, \ldots } |</td>
<td></td>
</tr>
</tbody>
</table>

The \textit{pump} \( \{f\} \) constructs a new operator that works on a set of bags, where each bag is represented by a BAT. On each such BAT, the operator that was between accolades, is executed. Its first parameter is called the “extent” as a result BUN is produced for all elements in this bag: for each BUN, the head value is placed in the head of the result, together with the result of \( f() \) in the tail, computed as explained in the following. The second parameter is called the “grouping” and associates each group (i.e. each tail value of the extent) with a possibly empty set of identifying values (typically, this is a unique set of \texttt{oid}-s). By joining this set of identifying values\(^3\) with the head of the third parameter (referred to as “attribute”), we arrive at a bag of attribute-values. For this bag, a BAT constructed that contains all values in both head and tail, and on this BAT, the operator \( f() \) is evaluated.

The fourth, optional, parameter of the complex pump imposes a selection on all attribute-values. This variant was introduced for computing \textit{data cubes} in data mining MIL queries, such as those generated by the tools of Data Distilleries [BRK98]. Such query loads work with a large number of different selections at the same time (that correspond to the data mining hypotheses under statistical validation), such that it becomes interesting to re-use the full BATs storing the grouping and the attributes for all tuples instead of producing intermediate subsets of these BATs for each selection:

\begin{verbatim}
sub_oid := oid_sel.select(true).mark.reverse;
sub_val := sub_oid.join(oid_val);
sub_grp := sub_oid.join(oid_grp);
\end{verbatim}

\(^3\)A past version of the pump \( \{f\} (BD, BX) \) had two parameters: a \( BD \) attribute-group BAT that contained attribute values in the head column, and group-values in tail. The extent \( BX \) was the second parameter of which only the head column was used, both as group-value and as head column for the result. The pump was changed to the current definition as we found that in many query processing situations, the group- and attribute-values are already placed in distinct BATs, and joining them together is not really necessary as this join can be handled inside the pump by a trivial scan over both (e.g. the case of equal, unique and ordered head columns), and this past pump can without loss of performance or expressiveness be rewritten to \( \{f\} (BX.mirror, BD.reverse.mark.reverse, BD.mark.reverse). \)
4.2. MIL EXECUTION

res_agrr := \( f(\text{sub}_\text{val}, \text{sub}_\text{grp}, \text{res}_\text{grp}) \);

\[ \Rightarrow \]

res_agrr := \( f(\text{res}_\text{grp}, \text{oid}_\text{grp}, \text{oid}_\text{val}, \text{oid}_\text{sel}) \);

This again supposes that it is trivial for the pump to construct the equi-join between selection-, attribute- and group-BATs. As we will see in Section 5.3.5, the Monet implementations of pump as well as the implementation of the multi-join map is based on this assumption, and uses pre-processing with the MIL equi-join for handling the more complex kinds of joins.

4.2.4 BAT Updates

The BAT update operators modify their BAT operands and are therefore separated from the BAT algebra.

<table>
<thead>
<tr>
<th>MIL operator type signature</th>
<th>Informal semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{bat}(\text{str} , h, \text{str} , t, \text{int} , \text{size} = 100) \rightarrow \text{bat}[H, T] AB</td>
<td>AB := AB_{prev} := \emptyset</td>
</tr>
<tr>
<td>\text{info}(\text{bat}[H, T] AB) \rightarrow \text{bat}[\text{str}, \text{str}]</td>
<td>\text{return {property, value} information on this bat}</td>
</tr>
<tr>
<td>\text{copy}(\text{bat}[H, T] AB \text{int mode} \rightarrow \text{CP}_{\text{NORMAL}}) \rightarrow \text{bat}[H, T] CD</td>
<td>CD := CD_{prev} := \text{independent copy of AB}</td>
</tr>
<tr>
<td>mode \in { \text{CP}<em>{\text{NORMAL}}, \text{CP}</em>{\text{ENUMERATE}} }</td>
<td></td>
</tr>
</tbody>
</table>

A newly created BAT is an empty bag. The \text{info} operator produces a meta-BAT that contains various properties (see Section 5.7) and statistics on a BAT. The reason why the \text{copy} operator is in the update interface is that copying has no meaning in an algebra. The copy produced is an identical set of BUNs, but may use an enumerated representation (\text{CP}_{\text{ENUMERATE}}, see Section 5.2.1).

<table>
<thead>
<tr>
<th>MIL operator type signature</th>
<th>Informal semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{create}(\text{bat}[H, T] AB, \text{str} , acc, \ldots)</td>
<td>\text{create search accelerator on head of AB}</td>
</tr>
<tr>
<td>\text{destroy}(\text{bat}[H, T] AB, \text{str} , acc)</td>
<td>\text{destroy search accelerator from head of AB}</td>
</tr>
<tr>
<td>Standard accelerators acc \in { \text{hash, tree} }</td>
<td></td>
</tr>
</tbody>
</table>

Search accelerators are part of Monet’s extensibility interface. MIL comes standard with the bucket-chained hash table structure and the T-tree, both successful main-memory data structures for accelerating value lookup [LC86a].

<table>
<thead>
<tr>
<th>MIL operator type signature</th>
<th>Informal semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{insert}(\text{bat}[H, T] AB, H a, T b) \rightarrow \text{bit}</td>
<td>AB := AB \cup {a, b}</td>
</tr>
<tr>
<td>\text{insert}(\text{bat}[H, T] AB, H a, T b) \rightarrow \text{bit}</td>
<td>AB := AB \cup CD</td>
</tr>
<tr>
<td>\text{delete}(\text{bat}[H, T] AB, H a, T b) \rightarrow \text{bit}</td>
<td>AB := AB \setminus {a, b}</td>
</tr>
<tr>
<td>\text{delete}(\text{bat}[H, T] AB, \text{bat}[H, T] CD) \rightarrow \text{bit}</td>
<td>AB := AB \setminus CD</td>
</tr>
<tr>
<td>\text{update}(\text{bat}[H, T] AB, H a, T b) \rightarrow \text{bit}</td>
<td>AB.\text{delete}(AB.\text{select}(&quot;=,b&quot;).\text{insert}([a, b]))</td>
</tr>
<tr>
<td>\text{update}(\text{bat}[H, T] AB, \text{bat}[H, T] CD) \rightarrow \text{bit}</td>
<td>AB.\text{delete}(CD.\text{mirror}.\text{join}(AB,&quot;=&quot;).\text{insert}(CD))</td>
</tr>
<tr>
<td>\text{access}(\text{bat}[H, T] AB, \text{int} , mode)</td>
<td>\text{change the update access mode of a BAT}</td>
</tr>
</tbody>
</table>

New BAT elements can be inserted and deleted, or values can be replaced in a straightforward way. Considering the fact that each BAT is a BUN-list (in an often unknown ordering), the \text{insert} guarantees that the only effect is that BUNs are appended to this list. Similarly the \text{replace} guarantees that the BUN-ordering is left intact. All update primitives exist both in single-value and bulk (bag-at-a-time) versions.
In order for updates to succeed, write access (\texttt{BAT\_WRITE}) has to be granted with the \texttt{access} operator. BATs constructed with the BAT algebra operators have default read-only (\texttt{BAT\_READ}) access, though, as this permits maximum optimizations in the Monet implementation, in particular, the sharing of memory resources between different BATs. For this same reason of enabling specific resource sharing optimizations, there also exist the append- and update-modes (\texttt{BAT\_APPEND}, resp. \texttt{BAT\_UPDATE}). The former allows only \texttt{insert}-s, while the latter also allows \texttt{replace}-s.

As the MIL update operators do not perform any locking, the MIL users need to ensure themselves that a BAT is not being updated concurrently (that is a hard rule). The Monet implementation can do BAT resource sharing between a read-only BAT and a writable BAT if the latter is append-only (in that case the already existing part of a BAT can be shared, since inserts only append). As for concurrency guarantees, multiple MIL users may read an append-only BAT. The same even goes for an update-only BAT, if the readers \textit{themselves} ensure that they only read elements that are not being replaced. Thus, in case of append-only and update-only BATs, we can have concurrent reads while \texttt{insert}s and in the latter access mode also \texttt{replace}s are ongoing. In their Monet implementation, however, \texttt{insert} and \texttt{replace} both can potentially cause a re-allocation of memory resources, which would make such concurrent sharing unsafe.

All update operators therefore return a \texttt{bit} success status. Any update operator on read-only BATs fails always, just like \texttt{delete}s on all but \texttt{BAT\_WRITE} BATs. In fact, all updates on such fully writable BATs always succeed.\footnote{Resource exhaustion causes an exception, not just a failed return status.} Inserts into append-only and inserts and replaces into update-only BATs \textit{may or may not} succeed, that depends on whether the implementation needs to reallocate. In case of such a failure, it is the responsibility of the MIL user to change the BAT access mode manually into \texttt{BAT\_WRITE}. It is then also up to the MIL users (e.g. a transaction system) to take explicit steps first, that prevent concurrent reads.

<table>
<thead>
<tr>
<th>MIL operator type signature</th>
<th>Informal semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{persistence} (\texttt{bat[H,T]}) (\texttt{AB, str s} ): \texttt{bit}</td>
<td>add to persistent store. name must not be in use.</td>
</tr>
<tr>
<td>\texttt{load} ((\texttt{str s, int mode=MALLOC}) \rightarrow \texttt{bat[H,T]})</td>
<td>\texttt{read()} or \texttt{mmap()} from persistent store into memory</td>
</tr>
<tr>
<td>\texttt{remove} ((\texttt{str s})) (\texttt{_commit()}) (\texttt{_mode} \in {\texttt{MALLOC, VM_NORMAL, VM_RANDOM, VM_SEQUENTIAL}})</td>
<td>remove from persistent store at next commit</td>
</tr>
<tr>
<td>global database commit</td>
<td></td>
</tr>
</tbody>
</table>

BATs normally exist until their last MIL variable reference disappears. By use of the \texttt{persistence} operator, however, BATs are associated with a persistent (disk image) name. This means that before releasing the memory resources of a persistent BAT, it is written to disk (if dirty or not yet saved). As such, a Monet database consists of a collection of persistent BATs. The \texttt{load} operator allows one to obtain a persistent BAT by name.

At a successful \texttt{commit}, the set of persistent BATs is guaranteed to be permanent, and all its BAT images clean. Also, all persistent resources (i.e. disk images) of previously persistent BATs that were \texttt{removed} since the previous \texttt{commit} are released. A database crash and subsequent restart brings back the exact same collection of persistent BATs with the same content, as well as cleans up disk images that did not make the last \texttt{commit}. To make this recovery possible, Monet keeps the exact disk image at last \texttt{commit} (i.e. writes before the next \texttt{commit} goes to different disk files).
This transaction functionality is intended to support only the most basic atomicity and durability needs. The rationale is that in the Monet and MIL design philosophy, query processing is separated as much as possible from transaction processing, in order never to let transaction overhead stand in the way of query operator performance, as there are quite a few query-intensive application areas that need very little transaction functionality and very much performance (e.g. data mining, multi-media retrieval, GIS). Moreover, in Section 7.4, we show that with two simple extension modules that introduce explicit locking and I/O, sophisticated and high performance transaction systems can be built on top of the basic MIL transaction functionality.

4.3 Object-Oriented Example

We now use part of the decision support database from the TPC-D and TPC-H benchmarks [Tra95, Tra99] to illustrate how an object-oriented data model can be stored in Monet BATs, and how query processing can be implemented using MIL primitives.

4.3.1 Mapping The Object Model

The object-oriented model supplants the flat relational tables with a nested type system in which Object and Set form the basic building blocks for database types. Both concepts can be refined using inheritance, and methods can be defined on them. A standardised object-oriented data model has been defined by OMG [CAB+94, CBB+97] together with an object-oriented equivalent of the SQL query calculus, named OQL. We rephrase our example schema from Section 4.1.1 in an object-oriented way, as follows:

```java
class Order {
    attribute date day;
    attribute float discount;
    relation Set<Item> items;
}
class Item {
    attribute float price;
    attribute float tax;
    relation Order order inverse Order.items;
}
```

The object-oriented data mapping into BATs is similar to the relational example. Simple object attributes are mapped just like relational columns into `table.attribute` BATs. Relation attributes, i.e. those that refer to an object, simply store an oid in the tail of such a BAT. Relation attributes allow to avoid one level of indirection as compared to the relational mapping (for instance, we can now directly join orders with items on its “order” attribute, instead of first having to join on the relational “order.id” attribute). The object-oriented data-model also allows to specify referential consistency using `inverse` relationships.

The possibility to nest collection types, however, leads to one extra BAT in the mapping of a class. This BAT is called the `extent`, and holds the oids of all objects in the collection. Set-valued attributes are stored in `table.attribute` BATs just like ordinary attributes, with the difference that each oid in the extent can occur zero or more times in the head of this BAT (instead of exactly one time). The set-value of such an attribute is formed by all tail values in this BAT with its oid in the head. In this way, nested collections are flattened into flat binary tables. Note that the empty set is encoded by the absence of an oid (the extent is necessary to detect its absence).
Figure 4.5: Mapping Objects onto BATs.

We represent the extent of a class with a `bat[oid,oid]` of which the head holds the oid-s of all objects in the class. Its tail column can be used to store the identifiers of the objects in the direct superclass. In top-level classes like `Item` and `Order`, we could store system-wide object identifiers in the tail.\(^5\) Making a difference between local and system-wide object identifiers is interesting, as local identifiers need to be unique only in their (sub)class, and hence might be implemented with a smaller data type. In this example, we use the same identifiers in both columns.

A final note concerns relation attributes with an inverse relationship, like `Order.item`s and `Item.order`. Here, we refrain from materializing an `order.item`s BAT. Whenever it is used, we can instead use the reverse view on the `item.order` BAT. This way, the problem of keeping both inverses consistent is implicitly solved by the data structure.

### 4.3.2 Query Translation

An example OLAP query on our schema asks per year totals of tax paid on discounted items:

```sql
SELECT year, sum(total)
FROM
  ( SELECT price * tax AS total,
          year(item.order.day) AS year
  FROM   item
  WHERE  order.discount BETWEEN 0.00 AND 0.06 )
GROUP BY year;
```

The MIL equivalent of this OQL single join query contains six BAT joins.\(^6\) The join in line 7 actually corresponds with the OQL join between orders and items, the other

---

\(^5\)Alternatively, one could choose to always store system-wide object identifiers in the tail of the extent.

\(^6\)Each intermediate result can be destroyed right after its last use. This is done by assigning its MIL variable e.g. to `nil`. Statements doing so have been omitted here for brevity.
five joins are a consequence of the vertical fragmentation applied in Monet. While this may seem a waste of effort, we will see later how the MIL operators keep track of the relatedness of vertical fragments and how they avoid doing unnecessary work when joining those.

<table>
<thead>
<tr>
<th>OQL query</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT year, sum(total) FROM (SELECT price * tax AS total, year(item.order.day) AS year FROM item WHERE item.order.discount BETWEEN 0.00 AND 0.06) GROUP BY year;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MIL translation (annotated below)</th>
</tr>
</thead>
<tbody>
<tr>
<td>01 ORD_DIS := select(order.discount, 0.0, 0.06)</td>
</tr>
<tr>
<td>02 SEL_ORD := ORD_DIS.mark.reverse</td>
</tr>
<tr>
<td>03 SEL_DAY := join(SEL_ORD, order.day)</td>
</tr>
<tr>
<td>04 SEL_YEA := (year)(SEL_DAY)</td>
</tr>
<tr>
<td>05 SEL_GRP := group(SEL_YEA)</td>
</tr>
<tr>
<td>06 GP_GRP := unique(SEL_GRP.reverse.mirror)</td>
</tr>
<tr>
<td>07 ITM_SEL := join(SEL_ORD, item.order.reverse).reverse</td>
</tr>
<tr>
<td>08 SUB_ITM := ITM_SEL.mark.reverse</td>
</tr>
<tr>
<td>09 SUB_SEL := ITM_SEL.reverse.mark.reverse</td>
</tr>
<tr>
<td>10 SUB_PRI := join(SUB_ITM, item.price)</td>
</tr>
<tr>
<td>11 SUB_TAX := join(SUB_ITM, item.tax)</td>
</tr>
<tr>
<td>12 SUB_TOT := [+] (SUB_PRI, SUB_TAX)</td>
</tr>
<tr>
<td>13 SUB_GRP := join(SUB_SEL, SEL_GRP)</td>
</tr>
<tr>
<td>14 RES_GRP := GRP_GRP.mark.reverse</td>
</tr>
<tr>
<td>15 RES_SUM := {sum}(RES_GRP, SUB_GRP, SUB_TOT)</td>
</tr>
<tr>
<td>16 RES_SEL := {min}(RES_GRP, SEL_GRP, SEL_GRP.mirror)</td>
</tr>
<tr>
<td>17 RES_YEA := join(RES_SEL, SEL_YEA)</td>
</tr>
<tr>
<td>18 RES_RES := join(RES_SEL, SEL_YEA)</td>
</tr>
</tbody>
</table>

Many optimizing query execution engines on the relational model have been built successfully in the past decades, by following the strategy of transformation of relational calculus to relational algebra with optimizing rewrite systems. The MIL binary algebra on the data model of Monet can be seen as an instance of such a physical algebra. This makes translation of SQL into MIL a special case of the traditional relational execution path. A specific translation technique for the decomposed model can be found in [KCJ'87, vdB94].

For supporting object-oriented systems, database researchers have tried to repeat the successes in the relational field by proposing a number of object-oriented query algebras [SLVZ95, CZ96]. They offer a nested object data model for supporting complex objects and support multiple collection types like Set, List and Bag. These languages were designed as input languages for algebraic optimizer systems that produce a physical query plan. Their implementation, however, turned out to be difficult due to the combination of a large number of operations and the complex storage model. To our
knowledge, no efficient implementations of object-algebras have been reported on large databases.

With the object-oriented MOA front-end [BWK98] for Monet we showed that despite the additional mapping of a logical data model (object-oriented) to the physical binary tables, ad-hoc query processing can be supported efficiently.

The fragmentation of the nested object-oriented data model onto binary tables brings some additional intrinsic optimizations. Traversing a relation attribute (see line 7) boils down to executing a MIL join operator on a \texttt{bat[oid,oid]}. This is very efficient as it comes down to the join optimization technique of \textit{join indices}, proposed in [Val87]. Additional optimizations are achieved on set-operator expressions on nested sets. Intersecting two set-valued attributes on a collection of objects, for instance, is executed with just one MIL \texttt{intersect} operator, e.g. the following OQL query may intersect many sets:

\begin{verbatim}
    select intersect(items1, items2) from Orders
\end{verbatim}

but translates in MIL to the single bulk operator:

\begin{verbatim}
    intersect(order_items1, order_items2)
\end{verbatim}

### 4.3.3 Optimized Translations

MIL operators have the execution policy of full materialization of their result. This choice was made mainly to allow for more main-memory specific optimization in the implementation of MIL operators. If intermediate results are larger than the available memory, this simple policy quickly becomes suboptimal. A \textit{pipelined} execution, where chunks flow through an operator tree, then performs better.

In a "pipelined" MIL program, we use on-the-fly horizontal fragmentation of tables in chunks, and let the MIL program iterate over these chunks. Standard decomposition rules for fragmented query processing must be applied in order to produce correct results using this additional horizontal fragmentation. For instance, a selection on a fragmented table can be executed on each chunk but results must afterwards be collected with a union. Aggregate computations must use decomposition rules of the aggregate in a local function and a global function [GBLP96].

Below we show a “pipelined” version of our sample MIL program, that was created by fragmenting the \texttt{Order} table on \texttt{oid}. Balanced chunk-sizes are guaranteed by the \texttt{split} MIL operator. All \texttt{Order} BATs are then fragmented with these split boundaries (lines 1-4).\footnote{In Section 5.3.2 we describe how the \texttt{split} and \texttt{fragment} operators can be made to consume just constant time.} The main MIL program body is then placed inside a loop over all chunks (lines 8-23). Wherever one of the \texttt{Order} BATs was used, it is substituted by the current chunk from this BAT. The SUM aggregate gets decomposed in a local \{\texttt{sum}\} and a global \{\texttt{sum}\}. The local aggregate results are grouped and re-aggregated using the global function after the loop terminates (lines 25-29).
The next step is to parallelize the pipelined program, by letting MIL work on multiple chunks in parallel. This is simply achieved by using a parallel batloop in line 7 with some parallelism degree P \( \text{BOUND}P \text{batloop}() \{ \ldots \} \). Not shown are lock statements that are now required around the insert statements. As an additional optimization, independent statements like line 1-4 could be placed in a parallel block \( \{ \ldots \} \).

### 4.4  MIL Extensibility

When a database is used for more than administrative applications alone, the need for additional functionality quickly arises [Sto86]. First of all, new application domains typically require complex user-defined data-types, such as for instance polygon or point. Secondly, one often needs to define new predicates and functions on them (intersect(p1, p2) or surface(p), for example). Also, new application domains often create a need for new relational operators, such as spatial join or polygon overlay. In order to evaluate queries using the new predicates, functions and relational operators, one needs new search accelerators (such as for instance R-Trees). Finally, applications using a database as back-end want the option to perform certain application-specific operations near to the data. If a database server allows one to link additional server code on top of it, the communication penalties of creating a separate server process, encapsulating the database (a "client-level" server), can be avoided.

#### 4.4.1 Other Systems

Postgres [SAH87] and Informix [Ger95] are examples of an extended relational systems, allowing for the introduction of new data types and access methods via prefixed
ADT interfaces. This works fine for new data types, predicates on them, and their accelerators, but does not allow for addition of new relational operators. In recent years, database researchers have spent much effort on Object-Oriented databases. In these systems, the programmer has more control, but to the point that data independence is compromised and the system gets hard to debug [eaENS89]. Another effort to achieve customizability has been the “extensible-toolkit” approach, where a database can be assembled by putting together a set of “easily” customizable modules (see [CD87]). Putting together such a system remains a serious work, however. One of the most appealing approaches to the problem we find in the Gral system [Güt89], which accepts a many-sorted algebra. Such an algebra can by its nature easily be extended with new operations.

4.4.2 MIL Extension Modules

Monet’s extension system most resembles Gral, supporting new data types, new search accelerators, and user-defined primitives (embodying both new predicates and new relational operators).

Monet extensions are packaged in modules, that can be specified in the Monet Extension Language (MEL). It requires you to specify ADT interfaces for new atomic types and accelerators, together with mappings to implementation functions in C/C++ compliant object code for all ADT operations and user-defined primitives.

Both module-specification and implementation object-code are fed into the mel parser and glue code generator. Implementation and glue code together form a shared library that can be dynamically loaded from MIL with the load(module,...); and unloaded with the drop(module,...) primitives.

Atomic Types

The ADT interface for atomic types assures that the MIL query algebra operators will work on user-defined types. For instance, one of the standard ADT operations is AtomHash(), which ensures that hash-based join works on BATs of any type. The ADT interface also contains routines to copy values to and from a heap, and to convert them to and from their string representations (for user interaction). Below we show how an atom can be specified, and which ADT operations should be defined:

\[
\text{ATOM } \langle \text{name} \rangle (\langle \text{fixed-size} \rangle, \langle \text{byte-alignment} \rangle)
\]

\[
\text{FromStr } := \langle \text{fcn} \rangle; \quad \text{# parse string to atom}
\]

\[
\text{ToStr } := \langle \text{fcn} \rangle; \quad \text{# convert an atom to string}
\]

\[
\text{Compare } := \langle \text{fcn} \rangle; \quad \text{# compare two atoms}
\]

\[
\text{Hash } := \langle \text{fcn} \rangle; \quad \text{# compute hash value}
\]

\[
\text{Length } := \langle \text{fcn} \rangle; \quad \text{# compute length of an atom}
\]

\[
\text{Null } := \langle \text{fcn} \rangle; \quad \text{# create a null atom}
\]

\[
\text{Put } := \langle \text{fcn} \rangle; \quad \text{# put atom in a BAT}
\]

\[
\text{Get } := \langle \text{fcn} \rangle; \quad \text{# get atom from a BAT}
\]

\[
\text{Delete } := \langle \text{fcn} \rangle; \quad \text{# delete atom from a BAT}
\]

\[
\text{Heap } := \langle \text{fcn} \rangle; \quad \text{# generate a new atom heap}
\]

END \langle \text{name} \rangle;

In case of a fixed-sized atom, the Put, Get and Delete operations, perform the trivial task of updating some BUNs in the BAT. In case of a variable-sized atomic type, they have the additional task of updating the heap.
ACCELERATOR <name>
  Build := <fcn>; # build accelerator on a BAT
  Destroy := <fcn>; # destroy accelerator
  Insert := <fcn>; # adapt acc. under BUN insert
  Delete := <fcn>; # adapt acc. under BUN delete
  Commit := <fcn>; # adapt acc. for transaction commit
  Rollback := <fcn>; # adapt acc. for transaction abort
  Cluster := <fcn>; # cluster a BAT on accelerator order
END <name>;

New Primitives

The MIL grammar has a fixed structure but depends on purely table-driven parsing. This allows for the run-time addition of new commands, operators, and iterators. Moreover, every user has an individual keyword-table, such that different users can speak different "dialects" of MIL at the same time. All system tables have been implemented as BATs and are accessible to the user via persistent variables for debugging purposes.

In order to do type-checking at the highest possible level, the MIL has been equipped with a polymorphism mechanism. A certain command, operation or iterator can have multiple definitions, with differing function signatures. Upon invocation, the Monet Interpreter decides which implementation has to be called, based on the types of the actual parameters, and matching operator and MIL procedure signatures in reverse order of definition.

OPERATOR <name>(<type-list>) : <type> := <fcn>;
ITERATOR <name>(<type-list>) := <fcn>;

The above shows the MEL syntax for specifying new primitives.

---

8 Extension code that "knows" the accelerator, typically accesses it with a C-macro or C++ inline function.
4.5 Conclusion

In this chapter, we have formally defined the MIL language, which is a computationally complete, dynamically typed, polymorphic, parallel programming language that stores (large) data collections in Binary Association Tables (BATs), and defines on this data model a 25-operator BAT algebra, an update interface and an extension language (MEL). Query algebras are not exactly new [Cod70], and have been used in a wide variety of database systems (e.g. [Güt89, DV92, LVZZ94, FGN+95]). Yet, MIL is a new query algebra as it is defined on the decomposed storage model (DSM) [CK85, KCJ+87] with the focus on extensibility and query-intensive applications.

In MIL, we have sought to define the minimal set of primitives that is powerful enough to allow a wide range of application scenarios. This has resulted in an algebra where the operators manipulate one, two or at most a few BATs (i.e. columns) at a time. This is a new direction for database query languages, which normally manipulate larger collections of columns (i.e. organized in tables or objects) – a direction pursued both to keep MIL neutral in the sense of data structures (and hence as easily applicable as possible to widely different logical data models) and because algebra operators with few parameters have a low degree of freedom, which opens up certain memory and CPU optimization opportunities in their implementation.

The fact that MIL is a column-at-a-time algebra typically leads to query graphs where the crucial nodes – such as those MIL operators computing selections and joins – are shared by multiple nodes higher in the query graph, that e.g. project columns through these joins and selections. In the generic iterator-model for query processing operators [Gra93a] the nested-iterator execution is broken for such nodes in order to materialize a sharable result. Given that such “forced materializations” are bound to happen often in MIL query plans, plus the fact that the nested iterator model might negatively impact CPU efficiency due to its recursive function calling, we decided to try a policy of full materialization of results in MIL operators. We showed in this chapter, that the resource consumption problems that are caused by such an approach can be handled by generating iterative “pipelined” MIL queries that process data in fragments. The advantage of such MIL queries is that these are directly apt for parallel query processing.

Some anecdotal evidence of the expressive power of MIL has been given in the various examples, where RDBMS and OODBMS data is stored in BATs and SQL/OQL queries on this data are translated into MIL.