Chapter 5

The Implementation of MIL

In this chapter, we describe the experience gained from implementing MIL as the primary interface language to the Monet system [BK95], and provide details of this implementation. We give an overview of its novel data structures and main-memory based algorithms. Special attention is paid to parts of the MIL language crucial for performance of the object-oriented and relational front-end applications.

Figure 5.1: Monet software architecture.

Figure 5.1 shows the Monet software architecture. The basic data structures and primitives for management of BATs are provided by the BAT kernel. This kernel includes support for persistence, transaction management and data access. The MIL operator primitives themselves are found in extension modules that can dynamically be loaded into the system. The other MIL language features, like parsing, variable handling, procedure management, resolution of overloaded operators, etc. are provided by the MIL interpreter, which coordinates execution of client applications.
5.1 Main-Memory System Design

Our design decision to target Monet towards main-memory execution of mostly read-only queries has consequences for its implementation. CPU instruction time and memory access cost are the dominant costs in main-memory systems, rather than I/O. In-memory data movement and predicate evaluation tend to take up most time during query processing [TLPZT97, ADHW99]. As main-memory optimization and cost modeling are largely unexplored research areas, main-memory system design still depends on intuitive programmer notions about what kind of coding style makes well use of memory cache, CPU registers, etc. In the design of Monet we therefore adhered to a number of rules of thumb:

1. *keep it simple.* Having a complex software architecture that offers powerful (generic) operations can easily lead to a high percentage of CPU overhead (e.g., in interpretation cost, parameter passing or buffer copying) when there is no overshadowing I/O cost. Straightforward processing algorithms and data structures, like bucket-chained hash tables or T-trees, have proven to work best in main memory [LC86a].

2. *use large granularities.* Implementation functions that work on the granularity of one tuple-at-a-time introduce a fixed amount of interpretation overhead for each tuple (stack operations, context switch). Using large granularities in the basic processing functions is an effective way to decrease the effects of such interpretation overhead.

3. *sequential memory access.* Historic cost-models for main-memory systems could safely assume absence of locality of reference on memory access. Modern custom hardware, however, has three memory levels, and uses pipelined memory transfer over the bus to enhance memory bandwidth. This makes sequential memory access significantly faster than random access. This holds for simple PC hardware, but is even more true for the new generation of scalable shared memory multi-processor computers [Sii97].

The result of applying these rules in the design of Monet are reflected in the simple sequential array structure for BAT storage, the bulk nature of the MIL operators, and the straightforward algorithms applied for their implementation.

5.2 Data Storage in Monet

The BAT data structure (Figure 5.2) is seen by database code as a pointer to a BAT descriptor. A BAT descriptor points to two column descriptors, one for the head column, the other for the tail. Each column descriptor contains column-specific information, like the type stored, and pointers to search accelerators. The bulk data structure of the BAT is the BUN heap, a main-memory array of binary tuples (BUNs). It is reachable from the BAT descriptor via a BUN descriptor. BUNs are fixed-size records that consist of a head- and a tail-field.

The heaps of a BAT are stored on disk in their exact memory layout, which enables us to map these files into virtual memory. The algorithms of Monet do not see the difference between mapped memory and normal memory. To make this direct mapping
possible, our storage scheme is carefully kept free of hard pointers. Absence of hard pointers implies that pointer swizzling [WD92] is performed lazily, on each data access. This policy only works well if data access is cheap and simple, and swizzling cost can be factored out in bulk operations. Monet therefore provides only a limited number of ways (3) to store atomic data in a BAT:

- **fixed-size atoms** are stored directly in the BUN record.

- **variable-sized atoms** store an integer in the BUN record. The integer is a byte-offset into a separate heap. This heap is a linear memory space just like the BUN heap and is reachable from the column descriptor.

- **implicit storage** virtual oid-s, defined by the additional void type, require no storage. A void column implicitly defines a column of densely ascending oid values (e.g., 100, 101, 102, 103, ...). These values are computed on-the-fly by adding the array index number of the BUN in the BUN heap to some oid base number, called “seqbase”. This seqbase (in our example 100) is stored in the column record.

The different treatment of variable-size atoms is necessary to keep the BUN heap an array of fixed-size BUNs. Implicit data storage was introduced deeply into the BAT data structure, as it is an optimization that is both greatly beneficial and often applicable. Many Monet applications map data into BATs that have one column with system-generated oid-s, and these are often dense and ascending. Virtual oid-s optimize both memory usage and value lookup: BAT sizes are cut by more than half\(^1\) and lookup can simply be done by position: when looking for oid 102 in a void column with seqbase=100 we calculate by subtraction that it is located at array index 2 in the BUN heap.

---

\(^1\)Alignement restrictions require a 4-byte type like oid to be aligned on 4-byte boundaries. A BUN [oid,chr] therefore occupies 8 bytes, whereas a [void,chr] occupies just 1 byte! Not aligning data would require values to be swizzled and copied on each access, at considerable extra CPU cost.
5.2.1 Storage Type Remappings

The simple nature of data storage in a BAT can be contrasted with more flexible data storage schemes that would allow a more compact data representation, for instance by using bit-wise integer encodings on low-cardinality columns [Tan99]. Such flexibility is added in Monet on a level higher than the direct data structures, by storing a BAT with a different physical type signature than it is logically perceived. Virtual oid-s are one example of such type remappings, as they implement the logical oid type in a different way. These type-differences in the BAT implementation are hidden by the MIL interpreter.

There are three type levels in the MIL implementation between which type-remappings exist (see Figure 5.3):

- **logical types** are the types known in the MIL language. These are called logical as they are not tied to one specific implementation.

- **physical types** are a superset of the logical types (a logical type may be stored in alternative ways). A physical type defines how a type is implemented. For instance, BAT's with an oid column may be stored either using oid-s or void-s. All physical types mapped on the same logical type have exactly the same MIL semantics.

- **implementation types** are a subset of the physical types, as the implementation of some physical types may re-use the implementation from others. Such a derived physical type only implements the string representation functions of Monet's atom interface, but copies all other behavior of the type it is derived from. For instance, bit is implemented by chr, and oid is (currently) implemented by int.

An enumeration type is a specific case of a logical-to-physical type remapping. The idea is to represent all values in an enumerated domain as (small) integers. In OLAP and data mining, column values often have a low cardinality. If 256 or few different values occur, 1 byte would suffice to encode the values (resp. 2 bytes for 65536 or less). A lookup table is used to translate the encoding back to the original value. The parametrized physical types enum1[BAT] and enum2[BAT] provide generic encodings into 1- and 2-byte integers. Their parameter is an encoding BAT that contains the lookup table.
The advantage of enumeration types is compact storage, which is achieved especially if the other column is void. In those cases, the BUN heap becomes a dense array of 1- or 2-byte values. Enumeration types preserve the value ordering on the encoded values in the integer codes. By doing so, operators like range-select can work directly on the encoded values. This policy, however, makes the enumeration types expensive to update, as an insert of a new value in the domain may trigger a recoding of all values in the BAT. For this reason, enumeration type storage should only be applied to BATs when updates are infrequent or bulky.

![Figure 5.4: bat[oid, str] implementation as bat[void, enum].](image)

Enumeration types in MIL are part of the physical data design (i.e., they should be explicitly created), but in their use they are transparent, because the MIL interpreter hides all storage type remappings from the user. This mainly happens during operator resolution, where the MIL interpreter looks for an operator with a signature that matches the actual parameters. Enumerated types generally do not have the same MIL operators defined on them as the types they encode. This often causes operator resolution to fail. The MIL interpreter catches such resolution misses and tries to take corrective action. Its last resort is to convert all enumerated BAT parameters back to their non-enumerated representation; though this is expensive and avoided where possible. For this reason we created specific implementations that directly handle enumeration types for certain operators (e.g., pump and multi-join map – see Section 5.3.5). In other cases, like bat.select("=", val), the MIL interpreter first tries to encode the non-BAT parameters (i.e. val), as this is cheaper than decoding the BAT.

### 5.3 MIL Operator Implementations

MIL operators are defined in an algebraic way; independent of the algorithms that implement them. Still, MIL is the target language for query-optimizing front-ends. For this reason, we introduce here the distinction between strategic and tactical query optimization, rather than the well-known distinction between logical and physical query-optimization. Query-optimizing front-ends produce MIL programs, so they decide the execution order of logical operations (the query execution “strategy”). Choosing a suitable algorithm (determining the run-time “tactics”) is done automati-
cally by the MIL operator implementations.

5.3.1 Tactical vs. Strategical Optimization

In normal query optimization, the physical algebra contains algorithm-specific primitives; the query optimizer chooses both strategy and tactics. MIL separates these two concepts, which alleviates (though not eliminates) a number of problems found in classical query optimization:

1. queries are optimized to be executed in isolation. The real situation of the execution system, however, is determined by a load of multiple queries and the database status at time of execution (including buffer management and available search accelerators), which might favor altogether different decisions [KdB94].

2. errors in estimates of intermediate result characteristics quickly propagate in complex queries where the estimates of one operator are calculated from parameters that are themselves the result of previous estimates [IC91]. Such estimation errors lead directly to wrong decisions made by the optimizer.

3. a very detailed model of query processing creates a huge search space for complex queries, whose search itself gets to be resource-consuming [GLPK94].

Problems 1 and 2 are dealt with by the tactical phase at run-time, so it can take into account the system state. Monet’s policy of materializing all results now becomes a benefit. When an operator starts, all information about its parameters is known. The optimization decisions are based on real information, not on estimates. This is the main difference between our approach and the so-called 'choose-plan operator' dynamic query optimization approach of [GW89, CG94, KD98]. In the 'choose-plan operator' approach, just before query execution, the estimates on the base operators are updated, and variables in the query are bound; then a new query optimization is done on the entire tree to see which alternative is best. This approach hence makes a decision based on much more actual information than normal QO – hence alleviates problem one – but as it is just an optimization closer to the moment of execution, it still suffers in errors made by estimation functions in the model (problem 2).

It is important to note that separating the query optimization in a strategical and tactical phase assumes that the strategical phase can do without physical details. The target of optimization cannot be formulated in terms execution time, as this depends on the (physical) algorithms chosen. A useful alternative target is minimization of the number of intermediate tuples generated. In this case, the price paid for our simplification is making the assumption that the best plan corresponds to the strategy that generates the least number of intermediate tuples.

The notion of strategical and tactical query optimization should not be confused with the classical notion of logical and physical query optimization [Gra93a], in which the logical phase depends on heuristics and the physical phase on a cost model. In the strategical phase we already determine the eventual execution order of logical algebra operations, so this optimization process includes both logical and physical optimization. We might for instance use cost models to estimate the selectivities of various MIL operators. The abstraction of all physical alternatives (e.g., merge-, hash- and nested loop-join) into their single logical operator (join) just causes a strong reduction of the search space, hence alleviates problem three.
5.3. MIL OPERATOR IMPLEMENTATIONS

As strategic optimization is a task of the front-end, further discussion of it falls outside the scope of this thesis.\(^2\)

5.3.2 Data Structure Optimizations

Tactical optimizations imply that the implementations of the MIL operations themselves decide at run-time how they will produce their logical result. Some MIL operators can exploit the decomposed nature of the BAT data structure (Figure 5.2) and actually produce their result without doing any real work.

![Diagram showing mirrored and void column views of BAT descriptors.](image)

**Figure 5.5:** mirror (left) and mark implementations (right).

**reversed view** the BAT data structure contains two BAT descriptors (see Figure 5.2); one *normal* and one *reversed*. These two descriptors differ only in that they have their column descriptor pointers swapped. As such, they represent two different *views* on the same BAT. The implementation of the MIL *reverse* operator on a BAT makes use of these views. It just jumps from one view to the other; making this operation free of cost.

**mirrored view** The mirror MIL operator creates a new BAT descriptor that has both head and tail column descriptor pointers pointing to the original head column descriptor. The resulting BAT appears to have two identical columns.

**void view** Virtual *oid-*s are introduced by the *mark* operator, that creates a new column descriptor stating the column data type to be *void*. As *void* values are computed just by position, the data in the BUN heap is not looked at, and can therefore share the BUN heap from the operand BAT.

**slice views** The slice operator always returns a view BAT, whose BUN heap points to a horizontal slice of BUN heap of its first parameter. The same technique is used when a BAT column contains ascending values and a range-*select* or *fragment* is done on it. In such cases, we just provide an alternative BUN descriptor that points at that subset (see Figure 5.6).

**enumeration views** BATs with enumeration types provide various opportunities for view optimizations. Consider the unary *group* operator, that replaces the tail

\(^2\text{In [MPK00] one can find a discussion of such a system based on Monet, with the additional aim of exploiting synergy between multiple queries; i.e. multi-query optimization.}\)
column with oid-s. Each such oid uniquely identifies a tail value. It therefore suffices to replace the encoding BAT of the enumeration type with an alternative encoding BAT that maps onto oid values (see Figure 5.6). We then just create a view with a different enumeration type in the column descriptor; this enumeration type points to our new encoding BAT.

The unary multi-join map can use a similar optimization (e.g., [*]( tax, 0.007) can just be executed on the encoding BAT) if the resulting values are unique. The one-column version of the unique operator also can represent its result by a view on this encoding BAT.

cast views a map cast occurs when an (implicit) conversion function is passed into the multi-join map (e.g., [bit](bat[any, chr])). If the cast target has the same implementation type as the tail of the BAT (as chr and bit indeed have), the multi-join map can create a view with a tail column descriptor that contains the target type.

All these optimizations are highly efficient and exploit the freedom that MIL has in choosing the best way an operator can be implemented at run-time.

<table>
<thead>
<tr>
<th>property</th>
<th>semantics</th>
<th>implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>int type</td>
<td>(physical) type number</td>
<td>field in the column descriptor</td>
</tr>
<tr>
<td>bit enum</td>
<td>true ⇔ enumerated type</td>
<td>derived from the physical type</td>
</tr>
<tr>
<td>bit dense</td>
<td>the column contains a densely ascending range</td>
<td>field in the column descriptor</td>
</tr>
<tr>
<td>bit sorted</td>
<td>true ⇔ ascending value sequence</td>
<td>field in the column descriptor</td>
</tr>
<tr>
<td>bit constant</td>
<td>true ⇔ all equal values</td>
<td>sorted(b) and min(b)=max(b)</td>
</tr>
<tr>
<td>oid align</td>
<td>unique identifier for this value sequence</td>
<td>field in the column descriptor</td>
</tr>
<tr>
<td>bit key</td>
<td>true ⇔ no duplicates this value sequence</td>
<td>field in the column descriptor</td>
</tr>
<tr>
<td>bit hash</td>
<td>true ⇔ hash-table on this column exists</td>
<td>derived from the accelerator-list</td>
</tr>
<tr>
<td>bit Ttree</td>
<td>true ⇔ T-tree on this column exists</td>
<td>derived from the accelerator-list</td>
</tr>
<tr>
<td></td>
<td><strong>BAT properties</strong></td>
<td></td>
</tr>
<tr>
<td>bit set</td>
<td>true ⇔ no duplicate BUNs exist in this BAT</td>
<td>field in the BAT descriptor</td>
</tr>
<tr>
<td>bit mirrored</td>
<td>true ⇔ head and tail column are identical</td>
<td>short for align(h)=align(t)</td>
</tr>
<tr>
<td>int count</td>
<td>the exact number of BUNs in the BAT.</td>
<td>computed from BUN descriptor</td>
</tr>
</tbody>
</table>

Figure 5.6: Slice and Enumeration Views.
5.3. MIL OPERATOR IMPLEMENTATIONS

5.3.3 Property-Driven Tactical Optimization

Not all MIL operators get a free ride in terms of their implementation. For these operators the Monet implementation contains a multitude of algorithms. Selecting a good alternative at run-time happens in three levels:

**operator overloading** Some of the selection work is off-loaded to the command resolution in the MIL interpreter. Figure 5.8 shows that specific implementations for the equi- and range- predicates are available for the MIL select operator. The binary select handles the \(=(\text{tail}(b), \text{value})\) and \(\text{isnil}(\text{tail}(b))\) predicates. The other select handles all range-conditions.

**algorithm selection** For the equi- and range-selects, the MIL interpreter can choose between hash-lookup, T-tree search, binary search and sequential scan (Figure 5.8). The tactical decisions made here are partially based on general system information about the CPU load, I/O activity and memory consumption. The most important information, though, comes from the properties that Monet maintains on all BATs (Figure 5.7).

Note that in the equi-select both the \(=(\text{tail}(b), \text{value})\) and \(\text{isnil}(\text{tail}(b), \text{value})\) predicates can be handled by the same algorithm, due to the fact that in Monet, the nil is implemented as a normal domain value (i.e., the smallest value). In the case of the range-predicates (which must filter out nil-s), this also means that in the physical C implementation, all range predicates have a lower bound that at least is nil (thus the range-selects for \(<(\text{bat}(\text{tail}), \text{value})\) and \(\leq(\text{bat}(\text{tail}), \text{value})\) need not be implemented).

The current tactical optimization procedures try to make best use of the information provided by the properties using heuristics, sometimes supplemented
CHAPTER 5. THE IMPLEMENTATION OF MIL

```plaintext
01 PROC select(BAT[ANY::1,ANY::2] b, ANY::2 val) : BAT[ANY::1,ANY::2] {
02   VAR i := b.info;
03
04   IF (i.find("dense").bit) {
05     RETURN positional_equi_select(b,v);
06   } ELSE IF (i.find("tail.hash").bit) {
07     RETURN hash_equi_select(b,v);
08   } ELSE IF (i.find("tail.sorted").bit) {
09     RETURN binsearch_equi_select(b,v);
10   } ELSE IF (i.find("tail.Ttree").bit) {
11     RETURN Ttree_equi_select(b,v);
12 } }
13 RETURN scan_equi_select(b,v);
```

Figure 5.9: MIL procedure for tactical optimization of equi-select.

by a simple cost model. For selections, positional lookup is the most effective method, followed by hash-lookup, binary search and T-tree search. These rules are optimal under main-memory conditions.

We implemented these tactical optimizations as scripted MIL procedures, like in Figure 5.9. This makes it easy to experiment with more complex cost models (e.g., by using virtual memory usage statistics and result size estimates, or even sampling).

An important feature of property-driven tactical optimization is that each operator implementation propagates all relevant properties onto its result BAT. If the scan_equi_select is executed on a BAT that has a sorted head column, it will propagate this property on its result.

type-specific expansions When an algorithm has been selected, the Monet implementation makes an additional automatic choice for a type-specific routine.

MIL operators are generally type-generic. This means that data access (for instance, comparing two values) goes through some atomic ADT function interface. Calling functions in the inner loop of an algorithm should be avoided in programs optimized for main-memory. In order to optimize main-memory performance, Monet has for each algorithm multiple implementation routines that are specific to a certain type. We call such type-specific implementations macro-expansions as we generate them automatically from one source base using a macro package.

Monet is an extensible system and new atomic types may appear at run-time, so there always needs to be one generic implementation that still uses the ADT routines. All type-specific expansions are optional. Code expansion is an optimization technique that trades off code size for performance, so only those cases that benefit most and are likely to be used should be expanded.

Figure 5.8 shows, that the equi-select hash implementation has macro-expansions for the types chr, sht, int, lng, and str. These are called chr_hash_equi_select, sht_hash_equi_select, etc. There is also one ADT_hash_equi_select that goes through the ADT interface, that is used for all other types. Note that the int expansion is also used for selecting oid values, as int is the implementation type of oid.3

---

3Moreover, as a special optimization, the equi-select remaps f1t selections to the int implemen-
Type-specific implementations are selected automatically, and are therefore not visible on the MIL level.

### 5.3.4 MIL Operator Implementation Overview

The different algorithms implemented for unary MIL operators are shown in bold text by Figure 5.10. This table shows per row which properties need be set for them to be chosen (leftmost column), as well as the macro-expansions applicable for each algorithm (rightmost column). As the decision which code-expansions to generate largely depends on the nature of the algorithm, these code-expansions and algorithm type share the horizontal dimension in Figure 5.10. The logical operators listed at the top of each column have a parameter $h$ or $t$, indicating the column (head or tail) of the BAT-parameter on which the properties should apply.

![Algorithm Overview for unop(b)](image)

The MIL-script for the equi-select of Figure 5.9 can be reconstructed by checking the equi-select column from top to bottom. The algorithm of the first row in which the property condition holds is chosen for execution.

A more complex example is the unique operator, which is split in two cases. The left column (labeled $X = h$) singles out the special case that only the head column is relevant for the uniqueness of the BUNs: if we know that both columns are equal (mirrored(b)), or one contains all the same values (constant(b)). In those cases, the standard two-column unique implementation invokes the single-column one, which is more efficient.

Figure 5.11 gives a similar algorithm overview for binary operators. Similar to unique, the binary set operators intersect, union and diff are implemented both in their 1- and 2-column versions.

We can read Figure 5.11 from top to bottom to find out which algorithm is chosen. Note that all binary operators except diff and group are symmetrical, in which cases the possibility of execution with swapped parameters is also taken into account. In contrast to the unary operators, there are no fall-back algorithms that are executed when no properties are set. The reason for this is that it is more efficient to enforce one property (e.g., by creating a hash-table or by sorting) than to execute a nested loop
algorithm. The decision which property to enforce and – for symmetrical operators – on which BAT, depends on memory statistics and result size estimates.

<table>
<thead>
<tr>
<th>property</th>
<th>positional lookup</th>
<th>expansions</th>
</tr>
</thead>
<tbody>
<tr>
<td>dense(t)</td>
<td>array lookup</td>
<td>1: {void}</td>
</tr>
<tr>
<td>align(l)</td>
<td>equal</td>
<td>1: {ADT}</td>
</tr>
<tr>
<td>constant(l)</td>
<td></td>
<td>6: {ADT}</td>
</tr>
<tr>
<td>mirrored(l)</td>
<td></td>
<td>8: {ADT}</td>
</tr>
<tr>
<td>not used</td>
<td>hash</td>
<td></td>
</tr>
<tr>
<td>sorted(l)</td>
<td>merge</td>
<td></td>
</tr>
<tr>
<td>sort(l)</td>
<td>binary search</td>
<td></td>
</tr>
<tr>
<td>T-tree(l)</td>
<td>T-tree search</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#operators</th>
<th>#algorithms</th>
<th>#expansions</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>47 (total)</td>
<td>269 (total)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.11: Algorithm Overview for \( \text{binop}(l, r) \).

### 5.3.5 Operators With Implicit Multi-Joins

The binary group, binary order, pump \{op\} and multi-join map \{op\} all define a result in terms of a multi-way equi-join on their parameters. Putting some hard-coded equi-join algorithm inside their implementations is not a good idea, because this leads to code duplication, but more importantly, such equi-joins would prevent any extended, optimized implementations of join to be used. In Section 6.5.2 we will see a particularly relevant example, where the MIL equi-join is overloaded to use the cache-friendly radix-cluster/decluster algorithm if certain conditions are met (i.e. other algorithms would thrash the caches).

Therefore, the implementation strategy for such operations is to assume the equi-join is trivial, i.e. each parameter is a densely ascending sequence, such that the join parameters already represent the join result that can be processed by a simple scan. For any other case, scripted MIL procedures perform the equi-join as a pre-processing step and thereafter call the “trivial-join” based implementations.

#### The order Operator

Figure 5.12 shows that the binary order is implemented by a C implemented operator \( \text{Corder} \) that assumes all its three parameters to be trivial equi-joins. A MIL procedure is used to handle the case where real equi-joins are necessary, in a pre-processing step. We also show details of the C implementation of \( \text{Corder} \), which assumes that the second parameter (group-bat) is already sorted on tail. This in turn implies that all groups are consecutive and can be processed by a merge algorithm.

51 would be expected from the table. The binary group, and order though, are code-expanded on 8 tail types.
5.3. MIL OPERATOR IMPLEMENTATIONS

MEL Declaration Of CTorder
OPERATOR CTorder (bat[oid,any::0] o, bat[oid,any] s, bat[oid,any] v) : bat[oid,int]

MIL Procedure That Resolves Equi-Join If Necessary

# dense() is used to assure a void column. always falls into the second (free) case
proc dense(bat[oid,any] b) : bat[void,b] { return b.reverse.order.mark.reverse; }
proc dense(bat[void,any] b) : bat[void,b] { return b; } # normal case

proc order(bat[any::1,any] b, bat[any::1,any] v) : bat[any::1,int] {
  if (v.info.find("dense").bit) {
    var s := b.order; # free if already sorted
    var sb := s.mark.reverse,
    var st := s.reverse.mark.reverse,
    var sa := sh.join(v);
    if (sa.count > sa.count) {
      return CTorder(sh, st, sa); # general, optimized, case
    }
  } # unexpected misses occurred in dense join!
  var xs := a.reverse.mark.reverse;
  var xs := a.mark.reverse;
  return CTorder(xs.join(sh).dense, xs.join(st).dense, xa);
}

## generic 2-way equi-join
var s := b.order;
var s := s.reverse.mark.reverse;
var yv := v.mark.reverse;
var yv := v.reverse.mark.reverse;
var yv := v.reverse.mark.reverse;
var yv := v.order.mark.reverse;
var yv := join(yo, yo.reverse);
var yv := join(yo, yo.reverse);
var yv := join(yo, yo.reverse);
var yv := join(yo, yo.reverse);
return CTorder(rx.join(xo).dense, rx.join(xs).dense, ry.join(yv).dense);
}

C Pseudo Code For CTorder Operator Implementation
typedef struct { oid head, int tail } bun; // order by result tuples
// CTorder helper function
int sort_renumber(bun *start, bun *end, char* base, int ID) {
  // put cluster in sorted order
  qsort.offset_longcopy.<T2> (start, end, sizeof(oid), base);
  // replace offsets by ascending IDs
  <T2> prev = (<T2>) (base + start.tail);
  for(ID++; start < end; start++) {
    <T2> cur = (<T2>) (base + start.tail);
    if (cur != prev) { ID++; prev = cur; }
    start.tail = ID;
  }
  return ID;
}

bun Result[N]
CTorder(oid 0[N], <T1> S[N], <T2> V[N]) {
  int clusterStart = 0, ID = 0;
  <T1> prev = S[0];
  for(int i=0, offset=0; i<N; i++, offset+=sizeof(<T2>)) {
    if (S[i] != prev) {
      ID = sort_renumber(Result+clusterStart, Result+i, V, ID);
      clusterStart = i; // new cluster starts here
    }
    Result[i].head = 0[1];
    Result[i].tail = offset;
  }
  sort_renumber(Result+clusterStart, Result+N, V, ID);
  return Result;
}

Figure 5.12: Implementing order as a MIL Procedure around CTorder
Figure 5.13: MIL Implementation Of Multi-Way Equi-Join In a Scripted Procedure

This merge algorithm assembles per group all BUNs in the output relation, where the int tail initially contains a byte-offset to the corresponding values. Each group forms a consecutive chunk in the result-BAT, and is sorted by a type-specific (code-expanded) Quick-Sort inside Monet. The sub-sorted chunk is then scanned, and for each different tail value, a new (ascending) oid is generated, which is placed over its old value (which was the beforementioned byte-offset). This algorithm is also illustrated later in this thesis in Figure 7.9.

The unary COrder is directly implemented by the Monet Quick-Sort, as listed in Figure 5.10. Monet does not use the standard library qsort as this one is buggy on some OSes\(^6\) and much performance can be gained by exploiting the knowledge on data types and tuple layout inside the database kernel, again using the technique of code-expansions. Monet implements Quick-Sort for all base types (8) with versions with or without offsets (2) using different variants for record swapping: 4-byte word swap, 8-byte word swap, and byte iteration (3) making for a total of \(8 \times 2 \times 3 = 48\) expanded qsort implementations. In Figure 5.10, the swapping method and addressing mode (offset or not) are counted as 6 separate algorithms. By sorting on a type and record-width known at compile-time, more than a factor 5 of performance is gained with respect to (non-buggy) standard library qsort-s, that compare each pair of values with a dereferenced (late binding) comparison routine that is passed to it as a parameter and swaps records with memcpy.

\(^6\)On Solaris, qsort never returns on medium sized sorts with many duplicates.
5.3. MIL OPERATOR IMPLEMENTATIONS

<table>
<thead>
<tr>
<th>#params</th>
<th>type expansions</th>
<th>expansions #params + 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>enum1</td>
<td>fixed-size</td>
</tr>
<tr>
<td>2</td>
<td>enum2</td>
<td>variable-size</td>
</tr>
<tr>
<td>1</td>
<td>normal*</td>
<td>fixed-size</td>
</tr>
<tr>
<td>1</td>
<td>normal*</td>
<td>ADT</td>
</tr>
</tbody>
</table>

Figure 5.14: 253 multi-join map implementations.

**pump {op} and multi-join map [op]**

The multi-join map implementation assumes that all its BAT-parameters have dense head columns, such that they already represent a multi-join result. If that is not the case, the MIL interpreter first invokes the multijoin shown in Figure 5.13, which similarly to order uses the MIL join to create BATs with dense head columns. This multijoin receives a nested-BAT that contains all $1 \leq n \leq N$ BAT[any::0,any::n]-parameters that must be joined on head column. The result $N + 1$ result “columns” of the form BAT[void,any::n] for $0 \leq n \leq N$. That is, the first column holds the head value that matched, and all remaining BATs the “columns” of the table that is the result of the multi-join. The multijoin is implemented as a MIL procedure, and follows a two phase approach that puts all BATs to be joined in a linear join tree, and then creates join indices from the bottom to the top. The second phase then uses the join indices to project the columns from the leaves up to the root level.

<table>
<thead>
<tr>
<th>group-type ($T_1$)</th>
<th>algorithm</th>
<th>value-type ($T_3$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADT</td>
<td>hash-group</td>
<td>fixed-size</td>
</tr>
<tr>
<td>chr</td>
<td>sel-hash-group</td>
<td>enum1</td>
</tr>
<tr>
<td>sht</td>
<td>merge-group</td>
<td>enum2</td>
</tr>
<tr>
<td>int</td>
<td>sel-merge-group*</td>
<td>normal*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>variable-size</td>
</tr>
</tbody>
</table>

4 4 3 2

Figure 5.15: 96 pump implementations.

The pump and multi-join map parameters often receive BAT parameters that have enumerated tail columns. In order to avoid conversions, their optimized implementations decode such types on the fly. We again use macro-expansions (see Section 5.3.3) to avoid having to check for each value whether it is enumerated (into either 1 or 2-byte integers) or not. Another expansion dimension that speeds up data access to the tail columns is created for fixed or variable-size atoms.

The multi-join map [op] has the additional complication that a variable number of BAT parameters may be passed. To eliminate looping overhead, and overhead for assembling a variable number of arguments for each invocation of op, code-expansions are made for the multi-join map with one, two and three parameters, always also code-expanded on their result type (hence $\#\text{params} + 1$).

The pump {op} combines “attribute” values with a common “grouping” value. Consequently, the grouping columns of such BATs tend to have a relative low cardinality, and are therefore often represented with an enumeration type that exploits this. For this reason, in addition to the generic ADT version, type-specific expansions are made for the chr, sht and int types (these are the implementation types for the enum1[b], enum2[b] and oid physical types, respectively). These are combined with four algorithms: merge- and hash-grouping, both in variants without and with a selection (the fourth optional parameter of the pump).
5.4 Conclusion

This chapter has given an overview of the implementation of Monet, and particularly of the MIL interpreter and the BAT algebra. This also entailed a detailed look at the data storage methods employed. Monet combines the decomposed storage model with a physical representation as a dense BUN array stored in main memory or mapped in virtual memory. The data type representation in the array is of fixed size and possibly compressed using *enumeration types*. The possible data storage optimizations are hidden in a data type layering of *logical types* (the MIL types), that can be stored in multiple *physical types*. Some physical types share the same implementation, hence *implementation types* form the lowest data type layer in Monet.

A number of techniques have been applied in Monet in order to optimize the performance of the BAT algebra operators. The most spectacular of such optimizations are the *view-implementations*, that reduce the operator cost to constant time, by constructing a result that shares its underlying BAT memory resources with its operands. Some of the most often used MIL operators (*reverse*, *mirror*, and *mark*) fall into this category.

One of the advantages of the full materialization in MIL is that when an operator starts executing, its parameters are completely available. This allows for a number of dynamic query optimization strategies that are hard to realize in the operator model [GW89, Gra93a], such as those that depend on the actual size, uniqueness, or ordering of intermediate query results. Also, it enables the MIL interpreter to choose itself which physical algorithm is most efficient to execute a particular (logical) algebra operator. This optimization process is called *tactical* optimization, and is separated here from *strategical* optimization, which aims at determining a good order of logical operators. We argue that abstracting away from physical algorithms reduces the query plan search space, which is an important problem in query optimization. We consider strategical optimization a front-end task (i.e. outside the scope of Monet), while tactical optimization is automatic in Monet.

Tactical query optimization in MIL is driven in the first place by the *properties* which are maintained on each BAT, and *propagated* by all operators. Some of the tactical optimizations can be done simply by MIL operator overloading (i.e. on the operator signature), others by explicit checking in MIL procedures that wrap around the various operator alternatives. The lowest level of tactical optimization is the use of the programming technique of *code-expansions* [Ker89]. Monet is implemented in a macro language from which C routines are extracted. Typically, each BAT algebra operator is code-expanded on the data type being processed by it. The main effect of code-expansions is that expensive constructs as dereferenced function calls (vs. C++ methods with late binding), variable-size data structure boundaries, and interpreted predicate evaluation, which lead to CPU-wise poorly predictable and thus inefficient code, can be eliminated. Thanks to the low degree of freedom in BAT algebra operator signatures, the amount of code expansion generated stays limited. In all, the 25 BAT algebra operators are implemented with 99 algorithms – which we summarized for all operators in this chapter. These algorithms are macro-expanded into 838 highly efficient implementation routines that can be invoked by the MIL interpreter. As these functions are all fairly simple, the cost in binary code size of all these expansions still remains moderate: when compiled with space-optimization on PC hardware, the Monet binary occupies about 1.5 MB.