The VLDB Journal (1999) 8: 101–119

The VLDB Journal

MIL primitives for querying a fragmented world*

Peter A. Boncz, Martin L. Kersten

University of Amsterdam, CWI, Kruislaan 413, NL-1098 SJ Amsterdam, The Netherlands e-mail: {boncz,mk}@cwi.nl

Edited by P. Valduriez. Received November 10, 1998 / Accepted March 22, 1999

Abstract. In query-intensive database application areas, like decision support and data mining, systems that use vertical fragmentation have a significant performance advantage. In order to support relational or object oriented applications on top of such a fragmented data model, a flexible yet powerful intermediate language is needed. This problem has been successfully tackled in Monet, a modern extensible database kernel developed by our group. We focus on the design choices made in the Monet interpreter language (MIL), its algebraic query language, and outline how its concept of tactical optimization enhances and simplifies the optimization of complex queries. Finally, we summarize the experience gained in Monet by creating a highly efficient implementation of MIL.

Key words: Database systems – Query optimization – Query languages – Main-memory techniques – Vertical fragmentation

1 Introduction

With the rapidly increasing demands of query-intensive applications like data mining and OLAP, we see performance of all-purpose commercial DBMSs become inadequate [2]. System vendors have addressed this problem by introducing specialized decision support systems (DSS) [22, 35, 42]. Such systems assume that data access is query-intensive rather than update-intensive.

Our group developed a new extensible database kernel, Monet [5], specifically targeted to query-intensive applications. It uses a fully fragmented data model that consists of binary tables only, and its query-processing infrastructure is optimized towards main-memory execution. We have demonstrated the high performance of Monet on various benchmarks, including TPC-D [3], OO7 [6], Sequoia [4], and the DD benchmark [2]. We believe Monet represents scientific state-of-the-art database technology, whose concepts, techniques and lessons have strong relevance for commercial products.

* This work was supported by SION grant no. 612-23-431

In this paper, we first outline how its architecture makes Monet different from other systems and which advantages this brings in performance, generalness and extensibility. We then focus on the question what query language framework is needed in this architecture. Our answers to this question are embodied in the design of the Monet interpreter language (MIL). We show how MIL addresses the needs of query-intensive database applications, and discuss implementation techniques that allowed us to make Monet highly efficient.

1.1 Query-intensive database architecture

Most relational DBMS products stem from a design line that originates from the late 1970s, and hence were carefully designed and tuned to the application requirements and hardware characteristics at that time. Technically speaking, their storage infrastructure remains optimized towards the needs of OLTP, which requires high performance on large numbers of small updates. Query-intensive applications like OLAP and data mining, however, have a distinctly different access pattern, as they condense large volumes of data into small, summarized, results.

In order to favor performance on single-row updates (OLTP), relational systems store their table data on disk clustered by row. Query-intensive applications, though, typically examine large percentages of the tuples in the database and must therefore often perform full table scans. Moreover, these queries typically use only a small subset of all attributes, which leads to projecting out many of the attributes of the table scanned. This means that, with row-clustered storage, only a small percentage of the I/O bandwidth generated by these table scans is actually useful.

The commercial DSS (sub)systems available deploy various architectural ideas, to cope with the high I/O load generated by their applications. We will briefly describe two such approaches: precalculation and vertical fragmentation.

1.1.1 Approach 1: Precomputation

The number of necessary table scans can be reduced by computing results from precomputed aggregates or summary tables.

One approach in this area is to store precomputed aggregates in a *multi-dimensional array* [26]. This allows for easy slice/dice operations between aggregated results. Severe problems arise when such array structures become sparse due to data skew, blowing up exponentially the size of these data structures [16].

The question of what to precompute and how to do this is often facilitated in practice by working with precooked solutions for standard business problems. In particular, products often assume a particular topology of the database schema, like a *star* or *snowflake* [22]. In such schemas, there is one big *fact table* that records "events", and maintains (hierarchical) information on the fact table attributes in small tables around it. One of the techniques used is to store precomputed aggregates in the fact table itself in 'phony' rows that have NIL (or ALL) values for the aggregated attributes. Microsoft SQLserver introduced a CUBE operator to SQL that works this way [15]. Another approach is to store *precomputed views* on frequently used subparts of the schema [9]. The inherent disadvantage of precomputation, however, is that only those queries that request precomputed data can be accelerated. Ad-hoc query execution can by definition never be fully supported with precomputation.

1.1.2 Approach 2: Vertical fragmentation

The idea of vertical fragmentation is to make table scans cheaper. When a table is decomposed into separate slices for each column, scan/project operations only need to scan those slices on which is being projected. This eliminates the I/O bottleneck from the DBMS on OLAP and data mining query loads.

Systems that use vertical fragmentation are Sybase IQ [42] and Compaq's Infocharger [11], which is inspired by an early version of Monet. Sybase IQ is especially well known for its use of various kinds of *bitmap indices*. The bitmap index is a search accelerator that has been around for quite a while [33], and works especially well for accelerating selection predicates on low-cardinality attributes. Sybase IQ extends this functionality with a patented high-cardinality bitmap schema similar to [48], which makes this technology more widely applicable.

In our view, however, bitmap indices are not the decisive factor that makes Sybase IQ efficient. In a normal select-project execution strategy, low selectivities – as typically encountered in OLAP queries – cause the project phase to degenerate into a full table scan, which renders useless any search accelerator, no matter how sophisticated. The important factor that makes it possible for IQ to successfully employ its indexing structures is in fact vertical fragmentation. IQ automatically creates a *projection index* on each attribute. These projection indices are simply vertical column slices, and are used during the project phase. Sybase IQ is even capable of supplanting scans on a projection index with a scan on its high-cardinality bitmap index. This saves additional I/O, since a high-cardinality bitmap can be seen as a projection index in compressed form.

1.2 Monet

In this section, we will present Monet, a database kernel developed since 1994 at our institute, targeted at achieving high performance on query-intensive applications. Additional objectives were to support multiple logical data models (object-oriented, relational, object-relational), providing parallelism on both shared-memory and shared-nothing hardware, and extensibility to the needs of specific application domains (GIS, multi-media).

We will give a short rationale of the main architectural choices made in Monet. More detailed information can be found in [3, 5].

1.2.1 Binary table model

Monet uses vertical fragmentation to avoid I/O where possible. It offers a kernel of DBMS primitives on binary tables (i.e., tables with two columns). This data model was introduced in literature as the Decomposed Storage Model (DSM) [12]. In the binary table model, the vertical fragmentation of the data structures is made explicit in the data model. The advantages of such an approach are:

- it is simple and elegant. Having a fixed table format both eases many design aspects of the query language and facilitates (optimization of) its implementation.
- it provides flexibility. Each application, be it an SQL front-end or a data-mining tool, can map its data on the binary table model in the way that suits best.

The binary table model has the drawback that queries must spend extra effort in recombining fragmented data, i.e. they must do extra joins. For this reason, the DSM has not been taken seriously by the database research community for a long time.

This counter-argument lost some of its power due to developments in modern custom hardware, as investing some extra memory and CPU processing (for a join) to reduce I/O can be a good trade-off. More importantly, the extra joins needed on a vertical fragmented data model are not mere 'random' joins. Vertical fragments of the same table contain identical tuple sequences, and if the join operator is aware of this, it does not need to spend effort in finding matching tuples at all. For this reason, Monet maintains fragmentation information as *properties* on each binary table, and propagates these across operations. Choosing algorithms is deferred to run time, and is done on the basis of such properties. Our experimental work with Monet shows, that by maintaining this extra run-time information, the additional join cost of vertical fragmentation can, in fact, be eliminated [3].

1.2.2 The role of main memory

When vertical fragmentation is successful in avoiding unnecessary I/O, the balance of query processing cost shifts from I/O to CPU cycles and memory access time. Past research on main-memory databases [1, 17, 38] has shown that main-memory execution needs different optimization criteria

than in I/O-dominant systems. As advances in CPU speed far outpace advances in DRAM latency, the effect of optimal use of the memory caches is becoming ever more important. Monet is optimized both in its algorithms and data structures for main-memory access. The effects of our optimizations were confirmed in experiments with the DD benchmark on a query load of data-mining requests, where Monet was measured to be more than a magnitude faster than a commercial relational DBMS [2], while both systems were CPU/memory bound.

While Monet is designed to exploit main memory when abundant, it is not an all-or-nothing main-memory system. If the database hot set exceeds main memory, the system relies on operating system (OS) support for managing virtual memory. Access to virtual memory causes page faults, and in this way I/O does play its role in the system. Relying on the OS to manage I/O (also called the *single-level store* approach) has the advantage that algorithms and data structures can stay targeted to main-memory execution, and therefore does not compromise performance when the hot set does fit in memory. On the other hand, the success or failure of our approach with single-level memory in I/O-dominant query execution, depends on how well Monet is able to provide the OS with information to steer its virtual-memory management. Our experience with virtual-memory advice on modern operating systems indicates that this indeed is feasible [4].

1.3 Overview and contributions

The first contribution of this work is to explore the database language issues for an algebra on the binary table model. This is embodied in Sect. 2 by the definition of the MIL. The second contribution comes from the considerable experience gained by the implementation of MIL. Section 3 discusses the techniques employed in Monet to make MIL into a highly efficient target language for the binary table model. Here, we also contribute the notions of strategical and tactical optimization, and show how the tactical run-time optimizations in MIL both simplify the query optimization process and enhance its quality. Finally, in Sect. 4, we draw conclusions.

2 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 (Fig. 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.

The language with which front- and back-end communicate, MIL, is crucial in fulfilling the design goals of the system. All front-end systems built on top of Monet, such as a relational or object-oriented DBMS, use it to communicate with the Monet back-end. This does not mean that MIL itself is an object-oriented or even a relational language; MIL just

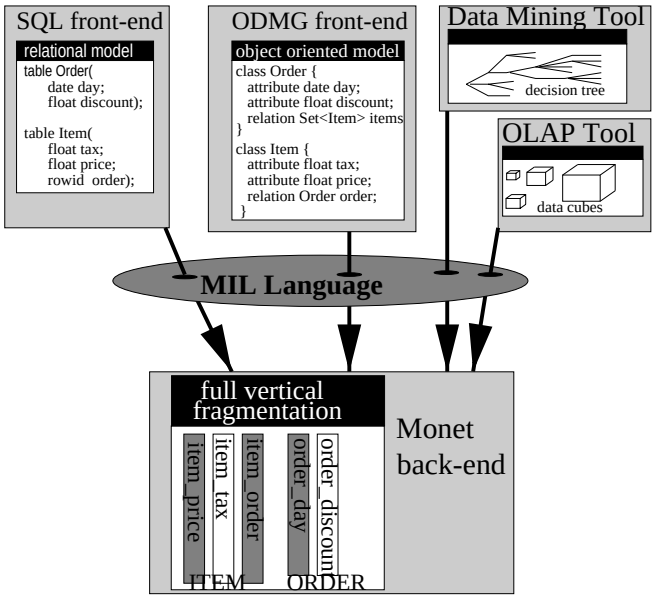

Fig. 1. Front-end/back-end architecture

provides the minimally 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 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 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 [37]. 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 [21], an early system that offered extensibility on all relevant levels (data types, algebra operators and search accelerators). The Fad [13] language is well known for its consequent functional operator style. The expressive generalness of this language proved a hindrance to run Fad programs efficiently on bulk data and perform parallelization [32]. Its successor language, Flora, used as an intermediate language in the IDEA [29] system, solved this problem by focusing 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.

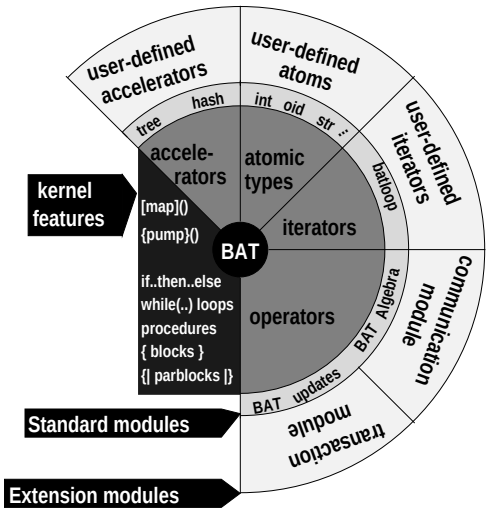

Fig. 2. The structure of MIL

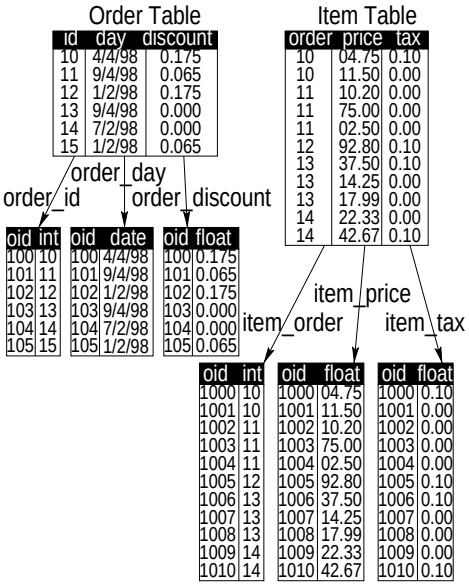

Fig. 3. Mapping of relational tables

2.1 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 $\mathscr{T}$ in MIL is:

1. $t \in \mathscr{A}_f \cup \mathscr{A}_v \Rightarrow t \in \mathscr{T}$.
2. $T_1, T_2 \in \mathscr{T} \Rightarrow \texttt{bat}[T_1, T_2] \in \mathscr{T}$.

The first rule defines atomic data types $\mathscr{A}$ (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 `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 $\mathscr{A}_f$ = {`bit`, `chr`, `sht`, `int`, `lng`, `flt`, `dbl`, `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 $\mathscr{A}_v$ = {`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) [3], the GIS domain (points, polylines, polygons) [4] and multi-media (images, video, audio) [31].

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 `type(value)` that implicitly exist for each atomic type. Casting is necessary to distinguish longs from integers and doubles from floats (e.g., `lng(42)`, `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 nil `oid`, as this value is often used. For nil values of other types, we use casts (e.g., `bit(nil)`, `int(nil)`, `dbl(nil)`).

2.1.1 Example data mapping

Suppose a relational schema with tables `Order` and `Item`, where the attribute `order` identifies the order to which an item belongs:

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

This relational data model can be stored in Monet by splitting each relational table by column [12]. 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 `oid` in the head.

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 is possible to use one of the unique columns as the head column in the BATs (like `id` for the `Order` table) field, but if not, we use system-generated `oid` numbers (see Fig. 3).

2.2 MIL language framework

The basic unit of MIL execution is the *operator*. MIL operators receive a number of input values and produce an 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,

SQL example query

```
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
ORDER BY total,id
```

MIL translation (*annotated below*)

```
ORD_NIL := select(order_discount,"between",0.0,0.06)
        a bat[oid,oid], head column with selected order-oids, nils in tail
IDS_NIL := join(order_id.reverse,ORD_NIL,"=")
          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_ITM := mark(ITM_NIL,oid(0)).reverse
         creates a bat[oid,oid] fresh oids in head, selected item-IDs in tail
UNQ_PRI := join(UNQ_ITM,item_price,"=")
           creates a bat[oid,flt] with selected item-IDs and their prices
UNQ_TAX := join(UNQ_ITM,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
table("2,1",UNQ_ITM,UNQ_TOT)
                  prints a 2-column table with item IDs and totals, with major
                ordering on the second column, and secondary ordering on the first
```

Fig. 4. A simple SQL query and a MIL translation

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 $\cdots$*type-expr*$\cdots$.

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

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

MIL is a procedural block-structured language, with standard control structures like if-then-else, and while-loops. The BAT *iterator*, denoted `bat-expr@`*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.

MIL *extension modules* introduce new atomic data types, operators, search accelerators[1] and iterators (Fig. 2). 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 at run time in MIL as scripted *procedures*. This extensibility mechanism is comparable to extension mechanisms used in relational systems like [18, 40] and differs from [34] in its choice to run extension code directly in the DBMS process space, as performance is a primary concern in Monet.

[1] 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).

2.2.1 Atomic value operators

A minimal set of operators like $=, \neq, <, >, \leq, \geq$ is present on all atomic types. Each atomic type brings with it an additional interface of specific operations.

Numerical types `sht,int,flt,dbl` and `lng` have arithmetic operators ($+$, $-$, $*$, $/$), as well as the `between(`*value, low, high*`):bit` that checks whether $low \leq value \leq high$.

Floating point `flt` and `dbl` 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 has the `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.

2.2.2 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 on BATs, so their parameters are BATs, and the result of each operator is a BAT.

We formally define the semantics of the BAT algebra operators using tables that describe the *operator signature*, followed by the equivalence symbol $\equiv$, and an algebraic definition that represents the result. We denote BATs as bags $\langle \cdots \rangle$, and – if we know that no double elements will occur – as sets: $\{\cdots\}$. The notation of a BUN is $[a, b]$. $|S|$ indicates the size of a bag or set.

Figure 4 provides a flavor of MIL execution in the example from Sect. 2.1.1. 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_ITM` and `UNQ_TOT`. Multi-column tables can be printed with the `table(str orderby, ..BAT[htpe,any]..)` operator. It prints the N-ary table that consists of all tail values that match on in the multi-join on the head columns of all BAT parameters. The optional `orderby` parameter contains a comma-separated list of columns to order the result on. In line 6 of Fig. 4, we use it to print the requested table with item-ids and total tax paid.

lookup & selections
with operator $(*f)(T, \cdots)$: `bit` $\in \{=, <, >, between, \cdots\}$)

select(bat[H,T] AB,str f,$\cdots p_i \cdots$):bat[H,oid] $\equiv$ $\langle$ [a,nil] | [a,b] $\in AB \wedge (*f)(b,\cdots p_i \cdots)$ $\rangle$
find(bat[H,T] AB, H a):$T \equiv b$ if $\exists$[a,b] $\in AB$, else T(`nil`)

The `select` operator allows for all kinds of selection predicates $(*f)()$. Equi-select, like in `select(b,"=",42)`, is one example. Line 1 of Fig. 4 shows the use of the range-select using the `between(flt,flt):bit` operator. Note that this operator selects on tail, but returns only the head column of the selected values. The tail column has the `oid` type, but is always filled with `nil` values. As we will see later, this particular kind of `oid` columns actually do not consume memory resources in the Monet implementation.

relational join with operator $(*f)(T_1, T_1, \cdots) : \texttt{bit} \in \{=, <, >, \geq, \leq, \cdots\}$
join(bat[H_1,T_1] AB, bat[T_1,T_2] CD,str $f, \cdots p_i \cdots$):bat[H_1,T_2] $\equiv \langle\ [a,d] \mid [a,b] \in AB \wedge [c,d] \in CD \wedge (*f)(b,c,\cdots p_i \cdots)\ \rangle$

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.

In lines 2, 3, 5 and 6 of Fig. 4 we use `=(oid,oid):bit` as the function $(*f)()$ for performing equi-join on the tail column of the left BAT with the head column of the right BAT. All kinds of boolean functions on the join column can be passed as an argument $(*f)()$.

Much like in the definition of the `select`, 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.

column operators
reverse(bat[H,T] AB):bat[T,H] $\equiv \langle\ [b,a] \mid [a,b] \in AB\ \rangle$
mirror(bat[H,T] AB):bat[H,H] $\equiv \langle\ [a,a] \mid [a,b] \in AB\ \rangle$
mark(bat[H,T] AB, oid o):bat[H,oid] $\equiv$ $\{\ [a_0,o],\cdots,[a_n,o+n]\ \}$ if $AB = \cup_{i=0}^{n} \langle\ [a_i,b_i]\ \rangle$

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 `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`+x =`nil`, so passing the `nil` value as second parameter will yield a BAT with `nil` in the entire tail column. The `mark` is often used for introducing a new column of unique `oid`s. As we saw in Sect. 2.1.1, 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 Fig. 4, the result of the join from line 2 gets a new unique column using the `mark`. This unique column is present in all temporaries of Fig. 4 whose name starts with `UNQ`.

By careful design of the BAT data structures (see Sect. 3.3.2), the `reverse`, `mirror` and `mark` actually do not have to materialize their results, which makes them zero-cost operators.

aggregates sum($\emptyset$) $\equiv T$(0), min($\emptyset$) $\equiv T$(nil), max($\emptyset$) $\equiv T$(nil)
count(bat[H,T] AB):int $\equiv |AB|$
sum(bat[H,T] AB):$T \equiv \sum_{[a,b] \in AB} b$
max(bat[H,T] AB):$T \equiv b : [a,b] \in AB \wedge \nexists y > b, [x,y] \in AB$
min(bat[H,T] AB):$T \equiv b : [a,b] \in AB \wedge \nexists y < b, [x,y] \in AB$

The above collection of *aggregates* is by no means complete. Extension modules with new ones can be introduced easily in MIL.

set operators
unique(bat[H,T] AB):bat[H,T] $\equiv$ $\{\ [a,b] \mid [a,b] \in AB\ \}$
diff(bat[H,T] AB, bat[H,T] CD):bat[H,T] $\equiv$ $\{\ [c,d] \mid [c,d] \in CD \wedge \nexists [c,d] \in AB\ \}$
union(bat[H,T] AB, bat[H,T] CD):bat[H,T] $\equiv$ $\{\ [a,b] \mid [a,b] \in AB \vee [a,b] \in CD\ \}$
intersect(bat[H,T] AB, bat[H,T] CD):bat[H,T] $\equiv$ $\{\ [a,b] \mid [a,b] \in AB \wedge [a,b] \in CD\ \}$

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(nil)`) or equal (with `mirror`).

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 *cross-table* 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 cross-tables.

groupby operators encoding with:[id$_B(x)$,x]$\in B$, and: id$_B(x)$=id$_B(y) \Rightarrow x = y$
group(bat[oid,T] AB):bat[oid,oid] $\equiv$ $\{\ [a,o] \mid o = \text{id}_{AB}(b) \wedge [a,b] \in AB\ \}$
group(bat[oid,oid] AB, bat[oid,T] CD) : bat[oid,oid] $\equiv$ $\{\ [a,o] \mid o = \text{id}_{CD}([b,d]) \wedge [a,b], [o,b] \in AB \wedge [a,d] \in CD\ \}$

The unary `group` operation is executed on a first BAT with an `oid` head column and creates a new equivalence group for each different value in 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. Each group-id is chosen from the collection of the `oid`s from the head of its group members[2]. Cross-tables can be refined using the binary `group` operation that subdivides the groups into new equivalence subgroups taking into account an additional `bat[oid,any]`.

This re-use of head column values for the group-ids makes it easy to go back from a group-id to the tail values the grouping is based on: we know that each group-id identifies an 'example' member of the group.

horizontal fragmentation
fragment(bat[H,T]AB,bat[H,H]CD):bat[H,bat[H,T]] $\equiv$ $\{[h,\text{select}(AB,\text{"}between\text{"},l,h)] \mid [l,h] \in CD\}$
split(bat[H,T] AB,int n):bat[H,H] $\equiv \{[l,h] \mid l \leq h \wedge \exists [l,x],[h,y] \in AB\} \wedge \forall [a,b] \in AB : \exists_{unique} [l,h] : l \leq a \leq h$

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

[2] The function $id_{AB}(tail)$ in the definition could, for instance, be implemented as returning the first $head$ value from AB that has this tail value.

column. The range BAT containing the split boundaries – that is passed as a second parameter – can be produced with the `split` operator. The number of boundaries `n` is only a target; the actual number of boundaries returned depends on the distribution of the values in the head column of AB.

2.2.3 Operator constructors

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

multi-join map with: $f(T_1,\cdots,T_n) : T_r$ e.g. =,≠,<,>,+,−,∗,/,and,or,$\cdots$
$[f]$(bat$[H,T_1]$ AB_1,$\cdots$,bat$[H,T_n]$ AB_n):bat$[H,T_r] \equiv$ $\langle\ [a, f(b_1,\cdots,b_n)] \mid \forall 1 \le i \le n : [a,b_i] \in AB_i\ \rangle$

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 `*(flt,flt):flt` operator was demonstrated in line 6 of Fig. 4. It produces a new BAT with multiplied item prices and taxes.

Though not shown in the definition, we can also type the `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 0.07. Typing `const` is actually not necessary for non-BAT values; e.g., `[*](item_tax, 0.07)` will do as well.

pump with: f(bat$[T,T]$) : R e.g. count,min,max,sum,$\cdots$
$\{f\}$(bat$[H,T]$ AB, bat$[H,I]$ CD) : bat$[H,R] \equiv$ $\langle\ [a,f(S_a)] \mid [a,d] \in CD,\ S_a = \langle [b,b] \mid [a,b] \in AB \rangle\ \rangle$

If not one aggregate should be computed over a table, but multiple over some GROUPBY condition, the *pump* operator constructor is used on the aggregate (e.g., `sum()` becomes `{sum}()`).

The pump 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 between accolades, is executed. More precisely, the return value of the pump is a BAT with all head values of CD, and in the tail the result of executing $f()$ on the BAT that consists of all tail values in AB that have that head value (possibly an empty BAT). This BAT is constructed on the fly and contains the same values in both columns, to accommodate operators $f()$ that work on either head or tail column.

2.2.4 BAT updates

The BAT update operators modify their BAT operands and are therefore separated from the BAT algebra. We use the symbol $\Rightarrow$ rather than $\equiv$ to describe their semantics.

BAT management masks ∈ {`cp_normal`,`cp_enumerate`,`cp_sort`}
create(str h, str t) : bat$[H,T]$ $AB \Rightarrow$ AB := AB_{prev} := $\emptyset$
info(bat$[H,T]$ AB) : bat[str,str] $\Rightarrow$ *return [property,value] information on this bat*
copy(bat$[H,T]$ AB, int *mask*) $\Rightarrow$ CD := CD_{prev} := *independent copy of AB*

A newly created BAT is an empty bag. The `info` operator produces a meta-BAT that contains various properties (see Fig. 15) and statistics on a BAT. The reason why the `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 have them stored in a different order in the BAT data structure (`cp_sort`) or use an enumerated representation (`cp_enumerate`, see Sect. 3.2.1).

I/O and virtual memory management modes ∈ {`malloc`,`vm_normal`,`vm_rand`,`vm_seq`}
save(bat$[H,T]$ AB, str s) : bat$[H,T] \Rightarrow$ *save to persistent store*
load(str s, int *mode*) : bat$[H,T] \Rightarrow$ *load or mmap() from persistent store,*
remove(str s) $\Rightarrow$ *remove persistent image of a BAT*

BATs may be saved to persistent storage, and loaded from there. When loading to virtual memory, explicit advice can be given on the access pattern expected. In this way, the OS virtual-memory management policy can be influenced.

search accelerator management standard accelerators acc ∈ { hash,Ttree }
create(bat$[H,T]$ AB, str acc, ..X_i..) $\Rightarrow$ *create search accelerator on head of AB*
destroy(bat$[H,T]$ AB, str acc) $\Rightarrow$ *destroy search accelerator from head of AB*

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 value lookup [30].

update management modes ∈ { read, append, update, write}
delete(bat$[H,T]$ AB, H a, T b) : bat$[H,T] \Rightarrow$ $AB \wedge AB := AB \setminus \{\ [a,b]\ \}$
insert(bat$[H,T]$ AB, H a, T b) : bat$[H,T] \Rightarrow$ $AB \wedge AB := AB \cup \{\ [a,b]\ \}$
update(bat$[H,T]$ AB,H a, T b,c):bat$[H,T] \Rightarrow$ AB.delete($[a,b]$).insert($[a,c]$)
access(bat$[H,T]$ AB, int $mode$) $\Rightarrow$ *change the update access mode of a BAT*

In order for updates to be possible, write access has to be granted. BATs constructed with the BAT algebra operators have default read-only access, as this permits certain optimizations in the implementation.

New BAT elements can be inserted and deleted, or values can be replaced in a straightforward way. The `insert`, `delete` and `replace` operators do not have ACID properties themselves. What they do provide, in combination with the *transaction support* operators, are the basic primitives for building a full transaction management. In this way, transaction overhead is avoided when full ACID functionality is not required (e.g., in a bulk load of an otherwise read-only data warehouse).

transaction support
flush(bat[H,T] AB) $\Rightarrow$ AB_{prev} := AB
alpha(bat[H,T] AB) : bat[H,T] $\Rightarrow$ returns $AB \setminus AB_{prev}$
delta(bat[H,T] AB) : bat[H,T] $\Rightarrow$ returns $AB_{prev} \setminus AB$
commit(bat[bat,bat] M):bit $\Rightarrow$ returns false: *commit failed, nothing changed* true: $\forall[A,B]\in M: A{:=}A_{prev}{:=}(A \cup \text{alpha}(B))\backslash\text{delta}(B)$

In the context of the Monet/ODMG system [6], we built a transaction-processing system on top of MIL by letting each transaction work on private copies of the BATs it accesses. Monet actually uses *private virtual-memory* OS primitives[3] to efficiently `copy` BATs loaded into virtual memory: only the modified pages occupy extra memory. Monet records the changes in a BAT in an `alpha` (inserted BUNs) and `delta` (deleted BUNs) status. At transaction commit we use this to propagate the changes made in the private copies back to the master BATs. This is done with ACID properties by the `commit` – that receives pairs of (master, modified-copy) BATs – by first safeguarding all BUNs to be overwritten in the masters in a roll-back file.

2.2.5 Parallelism

The MIL execution system is multi-threaded, with worker threads taking MIL jobs from a job queue. Parallel scheduling is hence automatic on multi-processor systems. SMP parallelism is expressed with explicit MIL constructs.

Similar to the standard sequential blocks $\{\ldots\}$, MIL has *parallel blocks* denoted by $\{|\ \cdots\ |\}$. All statements in a parallel block are scheduled for independent execution. A parallel block terminates when all statements in it have been executed. Another way of expressing parallelism in MIL is by placing a *parallelism degree* between square brackets behind the @ character in BAT iterators. For instance: `bat-expr@[P]batloop()` will process `P` items from the BAT in parallel. The multi-threaded nature of Monet is explicitly present in MIL using the constructs `fork` and `kill`, with which a new thread can be started to execute a statement asynchronously.

Shared-nothing parallelism is offered in MIL through the communication interface outlined below:

remote MIL execution
connect(str $host$, int $port$):int $\Rightarrow$ *connect with a remote Monet server*
close(int *fd*) : int $\Rightarrow$ *close connection*
export(int *fd*, str id, T $value$) $\Rightarrow$ *send a value with a certain ID to a remote site*
import(str id):T $\Rightarrow$ *block until a value came in for ID.*
mil(int *fd*, str mil_expr) $\Rightarrow$ *asynchronous MIL execution*
rpc(int *fd*, str mil_expr, $\cdots$*params*$\cdots$):T $\Rightarrow$ *export params, remote execution, import result*

We used these primitives as the building blocks for scalable distributed data structures (SDDS), and to experiment with distributed parallel join strategies [25].

[3] In UNIX systems, the this functionality is provided by the `MAP_PRIVATE` flag of the `mmap()` system call.

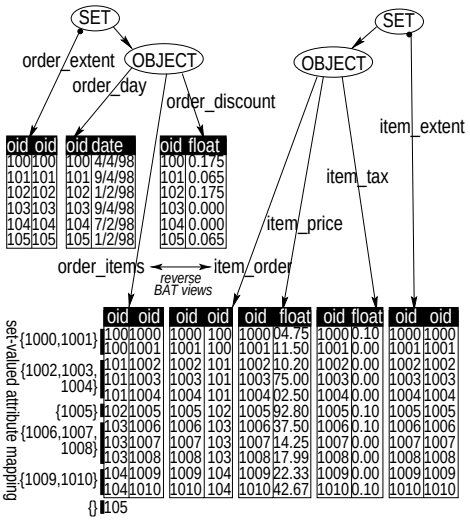

Fig. 5. Mapping objects onto BATs

2.3 Object-oriented example

We now use part of the decision support database from the TPC-D benchmark [43] to illustrate how an object-oriented data model can be stored and queried in Monet.

2.3.1 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 standardized object-oriented data model has been defined by the ODMG [7], together with an object-oriented equivalent of the SQL query calculus, named OQL. We rephrase our example schema from Sect. 2.1.1 in an object-oriented way, as follows:

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

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 present in the relational mapping (i.e., 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 `oid`s 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

OQL query

```
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;
ORDER BY year;
```

MIL translation (*annotated below*)

```
01 ORD_NIL := select(order_discount,"between",0.0,0.06)
     bat[oid,oid]   select all orders of interest
02 ORD_SEL := ORD_NIL.mark(oid(0))
     bat[oid,oid]   put unique id's in the tail
03 SEL_DAY := join(ORD_SEL.reverse,order_day,"=")
     bat[oid,date]  get [id,day] values
04 SEL_YEA := [year](SEL_DAY)
     bat[oid,int]   extract [id,year] values
05 GRP_SEL := group(SEL_YEA).reverse
     bat[oid,oid]   group on year
06 GRP_GRP := unique(GRP_SEL.mirror)
     bat[oid,oid]   all unique grp-ids
07 GRP_YEA := join(GRP_GRP,SEL_YEA,"=")
     bat[oid,int]   unique grp-ids and years
08 ITM_SEL := join(item_order,ORD_SEL,"=")
     bat[oid,oid]   [item,id] oid combinations
09 UNQ_ITM := ITM_SEL.mark(oid(0)).reverse
     bat[oid,oid]   renumber tail column
10 SEL_UNQ := ITM_SEL.reverse.mark(oid(0))
     bat[oid,oid]   renumber head column
11 UNQ_PRI := join(UNQ_ITM,item_price,"=")
     bat[oid,flt]   get bat[pos,price] values
12 UNQ_TAX := join(UNQ_ITM,item_tax,"=")
     bat[oid,flt]   get bat[pos,tax] values
13 UNQ_TOT := bat[*](UNQ_PRI,UNQ_TAX)
     bat[oid,flt]   compute bat[pos,price*tax]
14 GRP_UNQ := join(GRP_SEL,SEL_UNQ,"=")
     bat[oid,oid]   substitute sel for grp
15 GRP_TOT := join(GRP_UNQ,UNQ_TOT,"=")
     bat[oid,flt]   substitute pos for grp
16 GRP_SUM := {sum}(GRP_TOT,GRP_GRP)
     bat[oid,flt]   bat[grp,sum(tot)] results
17           table("1",GRP_YEA,GRP_SUM)
```

Fig. 6. OQL query and a MIL translation

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 this).

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[4]. 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 that have an inverse relationship, like `Order.items` and `Item.order`. Here, we refrain from materializing an `order_items` 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] Alternatively, one could choose to always store system-wide object identifiers in the tail of the extent.

2.3.2 Query translation

The example OLAP query in Fig. 6 on our schema asks per-year totals of tax paid on discounted items. The MIL equivalent of this single-join OQL query contains seven BAT joins. The join in line 8 actually corresponds with the OQL join between orders and items, the other six joins are a consequence of the vertical fragmentation applied in Monet. While this may seem a waste of effort, we describe in Sect. 3.4 how the MIL operators keep track of the relatedness of vertical fragments and how they avoid doing unnecessary work when joining those.

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. A specific translation technique for the decomposed model can be found in [28, 45].

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 [8, 41]. 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 implementation of object algebras have been reported on large databases.

With the object-oriented MOA front-end [3] 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 very efficient.

The fragmentation of the nested object-oriented data model onto binary tables brings some additional intrinsic optimizations. Traversing a relation attribute (see line 8) boils down to executing a MIL `join` operator on a `bat[oid,oid]`. This is very efficient, as it comes down to the join optimization technique of *join indices* proposed in [44]. 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 `intersect` operator, e.g., this OQL query may intersect many sets:

```
select intersect(items1, items2) from Orders,
```

but translates in MIL to the single bulk operator:

```
intersect(order_items1, order_items2)
```

2.3.3 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 *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

	MIL Statement	signature of created bat
01	BOUNDS := split(order_extent, N);	bat[oid,oid]
02	FRG_EXT := fragment(order_extent, BOUNDS);	bat[oid,bat[oid,oid]]
03	FRG_DIS := fragment(order_discount, BOUNDS);	bat[oid,bat[oid,flt]]
04	FRG_DAY := fragment(order_day, BOUNDS);	bat[oid,bat[oid,date]]
05	FRG_I_O := fragment(item_order.reverse, BOUNDS);	bat[oid,bat[oid,oid]]
06	GRP_YEA := new(oid,int);	
07	GRP_SUM := new(oid,flt);	
08	BOUNDS@bat[P]batloop() {	
09	ORD_DIS := select(FRG_DIS.find($h), "between", 0.00, 0.06);	bat[oid,flt]
10	ORD_SEL := ORD_DIS.mark(newoid(count(ORD_DIS)));	bat[oid,oid]
11	SEL_DAY := join(ORD_SEL.reverse, FRG_DAY.find($h), "=");	bat[oid,date]
12	SEL_YEA := bat[year](SEL_DAY);	bat[oid,date]
13	GRP_SEL := group(SEL_YEA).reverse;	bat[oid,oid]
14	GRP_GRP := unique(GRP_SEL.mirror);	bat[oid,oid]
15	GRP_YEA.insert(join(GRP_GRP, SEL_YEA, "="));	
16	ITM_SEL := join(FRG_I_O.find($h), ORD_SEL, "=");	bat[oid,oid]
17	POS_ITM := ITM_SEL.mark(0).reverse;	bat[oid,oid]
18	SEL_POS := ITM_SEL.reverse.mark(0);	bat[oid,oid]
19	POS_PRI := join(POS_ITM, item_price, "=");	bat[oid,flt]
20	POS_TAX := join(POS_ITM, item_tax, "=");	bat[oid,flt]
21	POS_TOT := bat[*](POS_PRI, POS_TAX);	bat[oid,flt]
22	GRP_POS := join(GRP_SEL, SEL_POS, "=");	bat[oid,oid]
23	GRP_TOT := join(GRP_POS, POS_TOT, "=");	bat[oid,flt]
24	GRP_SUM.insert({sum}(GRP_TOT, GRP_GRP));	
25	}	
26	GLB_GRP := group(GRP_YEA).reverse;	bat[oid,oid]
26	GLB_GLB := unique(GLB_GRP.mirror);	bat[oid,oid]
27	GLB_YEA := join(GLB_GLB, GRP_YEA, "=");	bat[oid,int]
28	GLB_TOT := join(GLB_GRP, GRP_SUM, "=");	bat[oid,flt]
29	GLB_SUM := {sum}(GLB_TOT, GLB_GLB);	bat[oid,flt]
30	table("1",GLB_YEA,GLB_SUM);	

Fig. 7. Pipelined MIL execution of the example query

program iterate over these chunks. Standard decomposition rules for fragmented query processing [14] 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 [15].

Figure 7 shows a "pipelined" version of our sample MIL program that was created by fragmenting the `Order` table on `oid`. Balanced chunk sizes are guaranteed by the `split` MIL operator. All `Order` BATs are then fragmented with these split boundaries (lines 2–5)[5]. The main MIL program body is then placed inside a loop over all chunks (lines 8–24). Wherever one of the `Order` BATs was used, it is substituted by the current chunk from this BAT. The SUM aggregate gets decomposed in a local `{sum}` and a global `{sum}`. The local aggregate results are grouped and reaggregated 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 8 with some parallelism degree P (`BOUNDS@[P]batloop()`).

3 The implementation of MIL

In this section, we describe the experience gained from implementing MIL as the primary interface language to the Monet system [5], 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.

[5] In Sect. 3.4.1 we describe how the `split` and `fragment` operators can be made to consume just constant time.

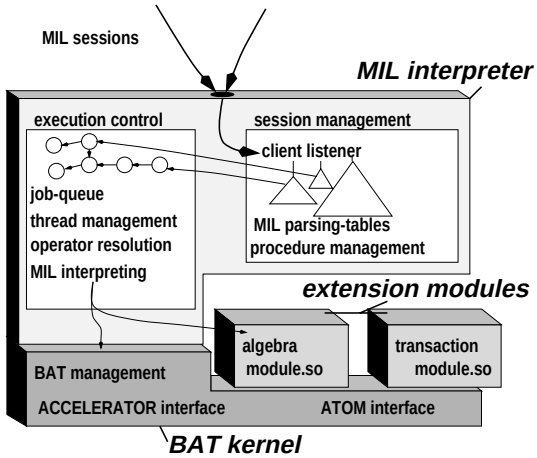

Fig. 8. Monet software architecture

Figure 8 shows the Monet software architecture. The basic data structures and primitives for management of BATs are provided by the *BAT kernel*. This includes support for persistency, 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.

3.1 Main-memory system design

Our design decision to target Monet at 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 [17, 46]. 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 good 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 [30].
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 [39].

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.

3.2 Data storage in Monet

The BAT data structure (Fig. 9) 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* [47] 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:

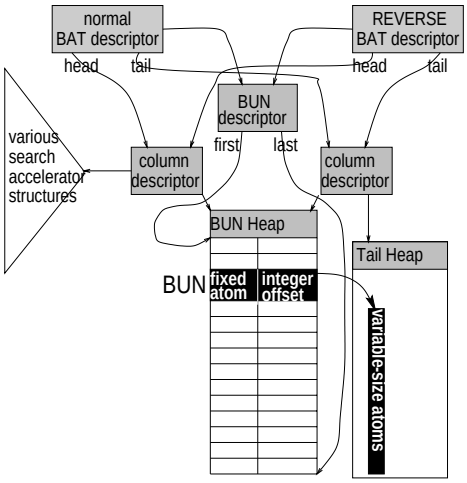

Fig. 9. BAT data structure

- *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 `oids`, 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 records. 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 `oids`, and these are often dense and ascending. Virtual `oids` optimize both memory usage and value lookup: BAT sizes are cut by more than half 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.

3.2.1 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 [11]. 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 `oids` 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 Fig. 10).

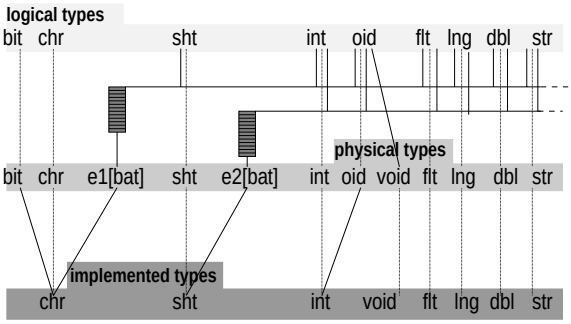

Fig. 10. Type remappings on three levels

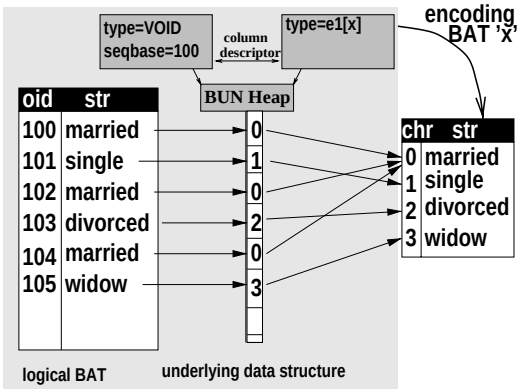

Fig. 11. `bat[oid,str]` stored as `bat[void,e1[x]]`

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, BATs 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 reuse 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 fewer different values occur, 1 byte would suffice to encode the values (2 bytes for 65536 or less). A lookup table is used to translate the encoding back to the original value. The parametrized physical types `e1[BAT]` and `e2[BAT]` provide *enumeration* 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.

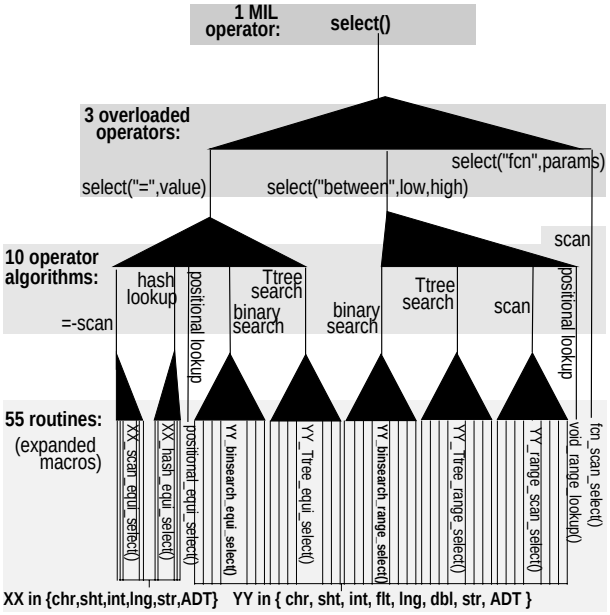

Fig. 12. One MIL operator, many implementations

3.3 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 *strategical* 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 automatically by the MIL operator implementations.

3.3.1 Tactical vs. strategical optimization

In traditional query optimization, the primitives in the physical algebra are algorithm-specific; 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 [27].

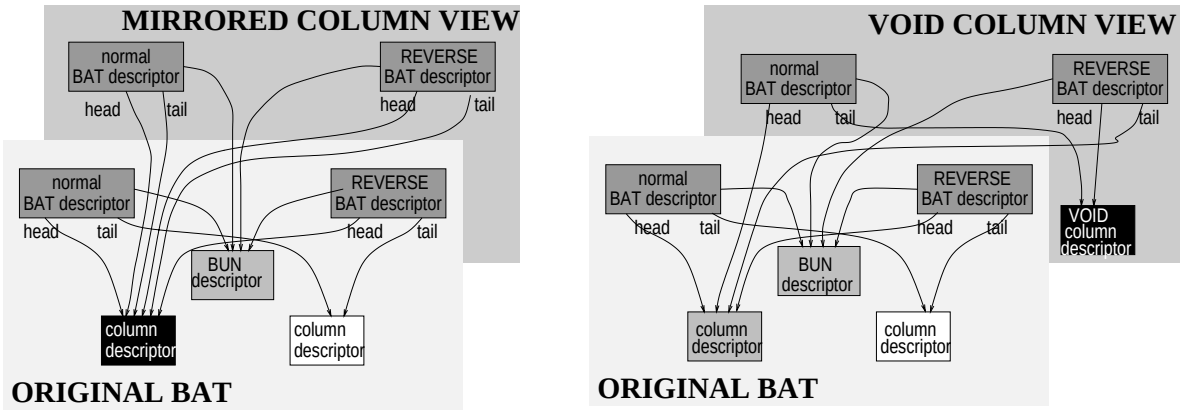

Fig. 13. `mirror` (*left*) and `mark` implementations (*right*)

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 [23]. 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 [36].

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 [10, 20, 24]. 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 thus 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 of 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 [19], in which the logical phase depends on heuristics and the physical phase on a cost model. In the strategical phase, we already decide 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 physical alternatives (e.g., merge, hash and nested-loop join) into their single logical operator (join) just causes a reduction of the search space, hence alleviates problem three.

As strategic optimization is a task of the front-end, further discussion of it falls outside the scope of this paper.

3.3.2 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 (Fig. 9) and actually produce their result without doing any real work.

Reversed view. The BAT data structure contains two BAT descriptors (see Fig. 9); 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. When a BAT column contains ascending values and a range-`select` or `fragment` is done on it, the result represents a contiguous subarray of BUNs in its BUN heap. In such cases, we just have to provide an alternative BUN descriptor that points at that subset (see Fig. 14).

Enumeration views. BAT with enumeration types provide various opportunities for view optimizations. Consider the unary `group` operator that replaces the tail column

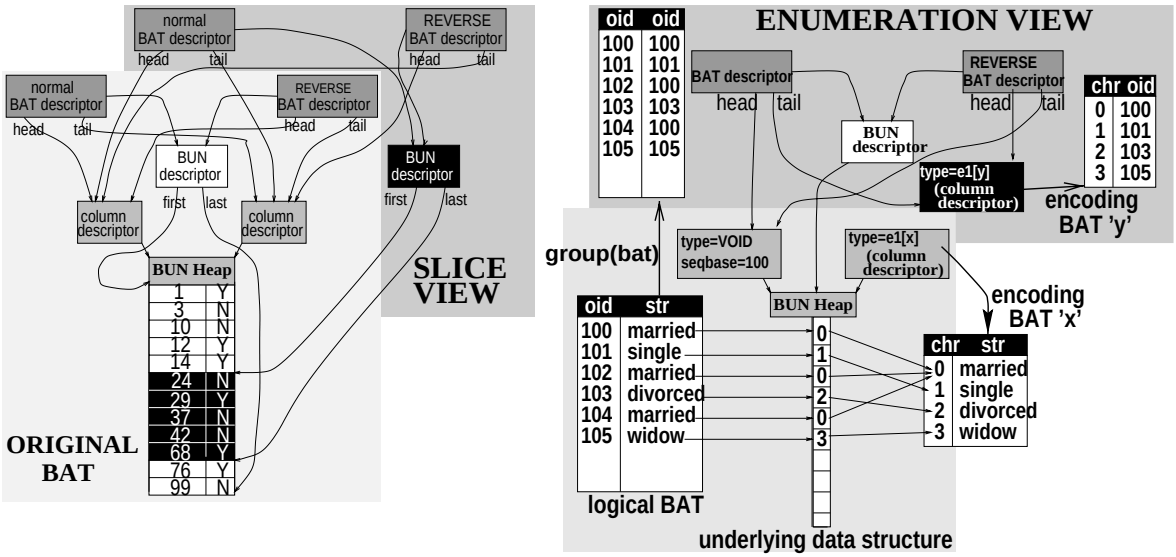

Fig. 14. Slice and enumeration views

property	semantics
column properties	
`int` **type**	(physical) type number
`bit` **enum**	`true` ⇔ enumerated type
`bit` **dense**	the column is a densely ascending range
`bit` **sorted**	`true` ⇒ ascending value sequence
`bit` **constant**	`true` ⇒ all equal values
`oid` **align**	unique identifier for this value sequence
`bit` **key**	`true` ⇒ no duplicates this value sequence
`bit` **hash**	`true` ⇔ hash-table on this column exists
`bit` **Ttree**	`true` ⇔ T-tree on this column exists
BAT properties	
`bit` **set**	`true` ⇒ no duplicate BUNs in the BAT
`bit` **mirrored**	`true` ⇒ head and tail column are identical
`int` **count**	the exact number of BUNs in the BAT.

Fig. 15. BAT and column properties

```
01 PROC select(BAT[ANY::1,ANY::2] b, STR "="
02             ANY::2 val) : BAT[ANY::1,ANY::2]
04 {
05     VAR i := b.info;
06
07     IF (i.find("tail.type") = "void") {
08             RETURN positional_equi_select(b,v);
09     } ELSE IF (i.find("tail.hash").bit) {
10             RETURN hash_equi_select(b,v);
11     } ELSE IF (i.find("tail.sorted").bit) {
12             RETURN binsearch_equi_select(b,v);
13     } ELSE IF (i.find("tail.Ttree").bit) {
14             RETURN Ttree_equi_select(b,v);
15     }
16     RETURN scan_equi_select(b,v);
17 }
```

Fig. 16. Procedure for algorithm selection in equi-select

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 Fig. 14). 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 be executed on the encoding BAT of the `tax` tail column). The one-column version of the `unique` operator also can represent its result by a view on an encoding BAT if it is executed on an enumerated column.

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.

3.3.3 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 offloaded to the command resolution in the MIL interpreter. Figure 12 shows that specific implementations for the "=" and "between" predicates are available for the MIL `select` operator (equi- or range-select). All other predicates are handled by a simple scan that invokes a predicate function on all tuples and retains those yielding `true` in the result.

Algorithm selection. For the equi- and range-selects, the MIL interpreter can choose between hash-lookup, T-tree search, binary search and sequential scan (Fig. 12).

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 (Fig. 15).

We implemented these tactical optimizations as MIL procedures, like in Fig. 16. 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).

The current tactical optimization procedures try to make best use of the information provided by the properties using heuristics, sometimes supplemented by a simple cost

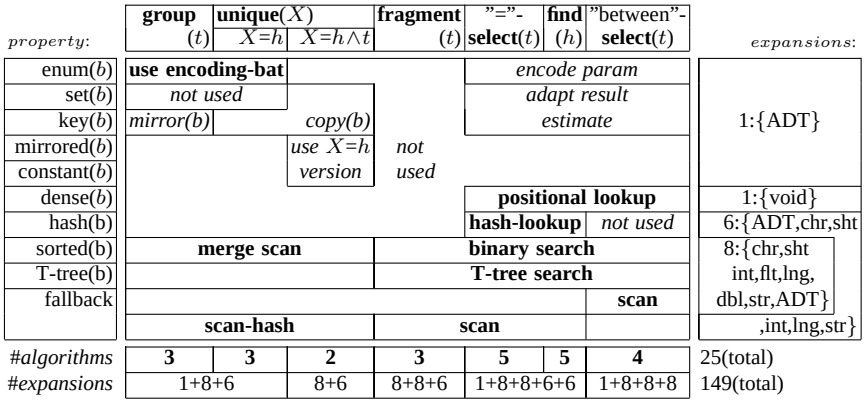

Fig. 17. Algorithm overview for unary BAT algebra operators **unop**(b).

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.

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, etc.

Type-specific expansions. When an algorithm has been selected, the Monet implementation makes an automatic extra 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 12 shows, for instance, 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, and 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` .

Type-specific implementations are selected automatically, and are therefore not visible on the MIL level.

3.3.4 MIL operator implementation overview

The different algorithms implemented for unary MIL operators are shown in bold text by Fig. 17. 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 do largely depends on the nature of the algorithm, these code expansions and algorithm type share the horizontal dimension in Fig. 17. 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.

The MIL script for the equi-select of Fig. 16 can be reconstructed by checking the '='-select column in Fig. 17 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(b)` 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. This happens 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(b)` implementation invokes the single-column one, which is more efficient.

Figure 18 gives a similar algorithm overview for binary operators. The join is overloaded with specific implementations for the $<, \leq, =, >, \geq$ predicates. Other join predicates are handled by a simple nested-loop algorithm that invokes a predicate-testing function for each pair of matching tuples. Similar to `unique`, the binary set operators `intersect`, `union` and `diff` are implemented both in their 1- and 2-column versions.

In a pure main-memory situation, we can read Fig. 18 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 fallback 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.

property:	**group** (h,h)	=-**join** (t,h)	**intersect**(X,X) **union**(X,X) **diff**(X,X) $X{=}h$	$X{=}h \wedge t$	{≤,<, ≥,>} -**join** (t,h)	*expansions*:
align(l)= align(r)	**array lookup** (if key(l)∨key(r))		equal bags⇒ **trivial impl's**	*one column insignificant⇒ use $X = h$ version*	*not used*	1:{ ADT }
constant(l)= constant(r)		*cart. prod.*			*cart. prod.*	
mirrored(l)∧ mirrored(r)	*not used*					
dense(l)	**positional lookup**				*not used*	1:{void}
hash(l)	**hash lookup**					6:{ADT,chr, sht,int,lng,str}
sorted(l)∧ sorted(r)	*not used*	**merge algorithm**				8:{chr,sht int,flt lng,dbl, str,ADT}
sorted(l)		**binary search**				
T-tree(l)		**T-tree search**				
#operators	1	1	3	3	4	
#algorithms	3	6		5	3	54 (total)
#expansions	6+6+6 [6]	1+1+6+8+8+8		1+6+8+8+8	8+8+8	335 (total)

Fig. 18. Algorithm overview for binary BAT algebra operators **binop**(l,r)

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

The pump and multi-join map primitives are mainly executed on BATs that have an `oid` head column. For this reason, the `oid` cases are specifically optimized using code expansions. This can be contrasted with the other MIL operators which are optimized for all standard data types.

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 Sect. 3.3.3) 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 pump combines tail values with a common head value. Consequently, the head 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 enumeration types `e1[b]`, `e2[b]` and `oid` physical types, respectively). These are combined with three algorithms: merge-, hash- and T-tree grouping.

The multi-join map `[op]` has the additional complication that a variable number of BAT parameters ($/P$) may be passed. Its fallback implementation is a generic adaptive N-ary implementation that uses all properties in an interpretative manner. As the multi-join result is an N-ary table, it cannot be represented as a BAT. For this reason, results produced are returned by calling a call-back function for each tuple. This implementation, which is also used for the BAT-`print`, is not main-memory efficient. Two optimized algorithms are available, namely array lookup (if all head columns are identical), or a merge algorithm (if all head columns are sorted).

[6] 1+1+6 would be expected from the table. The binary group, though, always gets `oid`s in its head columns. No expansions would be necessary (1+1+1), but then again the grouping algorithm is based on hashing on the tail column of the r parameter. So, each algorithm is expanded on tail for hashing (6 expansions).

head-type		*algorithm*		*tail-type*		
ADT		hash-group		e1[B]		fixed-size
chr				e2[B]		
sht		merge-group		normal		variable-size
int	*	Ttree-group	*		*	
4		3		3		2

Fig. 19. 72 pump implementations

head	*algorithm*	*/P*	*tail expansions*				*expansions*$^{/P}$	
ADT	array lookup	3	e1[B]		fixed-size		3	27
		2					6	36
		1	e2[B]	*	variable-size	=	6	6
int	merge lookup	2					6	36
		3	normal				3	27
	multi-join	N	3		2		1	1
ADT		N					1	1

Fig. 20. 134 multi-join map implementations

In all, the 20 MIL primitives are implemented using around 100 algorithms. These algorithms are macro-expanded into around 700 highly efficient implementation functions 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 MB.

3.4 MIL execution trace

We illustrate MIL execution in Monet by running our example query of Sect. 2.3 on the database specified by the TPC-D benchmark [43]. The mapping of an object-oriented data model onto BATs depicted in Fig. 5 is a sub-part of the object-oriented version [3] of the TPC-D schema.

The order table in the TPC-D database has 1.5M rows (we use SF=1), and the item table has 6M rows. We store all BATs from Fig. 2 with the physical `void` type in the head column, except for the `order_items`, which is not materialized (the reverse view on `item_order` is used instead). In this example, we use enumeration types in the tail columns of the BATs that store the columns `tax`, `discount`, `price` and `day`. This compact representation results in the entire TPC-D database occupying just 600 MB of disk space in Monet, instead of the 1 GB that it normally occupies in relational DBMS products.

The table below shows a trace of executing the MIL script for our example query. It describes the exact implementations chosen for each MIL operator in the script and the BAT and column properties of all results created. Using these properties, one can check back to Figs. 17 and 18 to see why the tactical optimization chose that particular implementation function, and also see how each operator implementation *propagates* its properties onto their BAT result.

The execution trace fully lists all operators used, including the `load()` operator (that loads a persistent BAT into memory and gives virtual-memory advice) and the `free()`, that is automatically invoked by the MIL interpreter if a BAT is no longer used. The rightmost columns show the amount of reserved virtual memory ('res'), the actual amount

lineno.	example query execution trace: MIL_implementation_function → result	BAT signature: head	BAT signature: tail	BAT Properties[7]: head	BAT Properties[7]: tail	BAT Properties[7]: count	memory(MB): diff	memory(MB): res	memory(MB): use	memory(MB): hot
	load("order_discount", vm_seq); → order_disc	void	e_1[flt]	$D_{100}A_o$	A_{oi}	1.5M	1.5	1.5	0.0	0.0
01	**chr_scan_range_select**(order_disc,$e_{1[.00]}$,$e_{1[.06]}$) → ord_nil	oid	void	SKA_1	A_2	0.9M	3.6	5.1	3.6	0.0
	free(order_disc) → *in_memory*(ord_nil)						-1.5	3.6	3.6	0.0
02	**view_mark**(ord_nil,x) → ord_sel	oid	void	SKA_1	D_xA_3	0.9M	0.0	3.6	3.6	0.0
	load("order_day", vm_seq); → order_day	void	e_2[date]	$D_{100}A_o$	A_{od}	1.5M	3.0	6.6	3.6	0.0
03	**positional_equi_join**(**view_reverse**(ord_sel),order_day) → sel_day	void	e_2[date]	D_xA_3	A_4	0.9M	1.8	8.4	5.4	0.0
	free(order_day) → *in_memory*(ord_nil,sel_day)						-3.0	5.4	5.4	0.0
04	**norfix_array_map**(sel_day,**date2year**()) → sel_yea	void	sht	D_xA_3	A_5	0.9M	1.8	7.2	7.2	0.0
	free(sel_day) → *in_memory*(ord_nil,sel_yea)						-1.8	5.4	5.4	0.0
05	**view_reverse**(**sht_scanhash_group**(sel_yea)) → grp_sel	e_1[oid]	void	A_6	D_xA_3	0.9M	0.9	6.3	6.3	0.0
06	**enum_unique**(**view_mirror**(grp_sel)) → grp_grp	e_1[oid]	e_1[oid]	SKA_7	SKA_7	7	0.0	6.3	6.3	0.0
07	**positional_equi_join**(grp_grp,sel_yea) → grp_yea	e_1[oid]	sht	SKA_7	A_8	7	0.0	6.3	6.3	0.0
	free(sel_yea) → *in_memory*(ord_nil,grp_sel)						-1.8	4.5	4.5	0.0
	load("item_order", vm_seq); → item_order	void	oid	$D_{1000}A_i$	SA_{io}	6.0M	24.0	28.5	4.5	0.0
08	**int_merge_equi_join**(item_order,ord_sel) → itm_sel	oid	oid	SKA_8	SA_9	3.6M	28.8	57.3	33.3	0.0
	free(item_order,ord_nil) → *in_memory*(grp_sel,itm_sel)						-27.6	29.7	29.7	0.0
09	**view_reverse**(**view_mark**(itm_sel,0)) → pos_itm	void	oid	D_0A_8	SKA_8	3.6M	0.0	29.7	29.7	0.0
10	**view_mark**(**view_reverse**(itm_sel),0) → sel_pos	oid	void	SA_9	D_0A_8	3.6M	0.0	29.7	29.7	0.0
	load("item_price", vm_seq); → item_price	void	e_2[flt]	$D_{1000}A_i$	SA_{ip}	6.0M	12.0	41.7	29.7	0.0
11	**positional_equi_join**(pos_itm,item_price) → pos_pri	void	e_2[flt]	D_0A_8	A_{10}	3.6M	7.2	48.9	36.9	0.0
	free(item_price) → *in_memory*(itm_sel,pos_pri,grp_sel)						-12.0	36.9	36.9	0.0
	load("item_tax", vm_seq); → item_tax	void	e_1[flt]	$D_{1000}A_i$	SA_{it}	6.0M	6.0	42.9	36.9	0.0
12	**positional_equi_join**(pos_itm,item_tax) → pos_tax	void	e_1[flt]	D_0A_8	A_{11}	3.6M	3.6	46.5	40.5	0.0
	free(item_tax) → *in_memory*(itm_sel,pos_pri,pos_tax,grp_sel)						-6.0	40.5	40.5	0.0
13	**e2fix_e1fix_array_map**(pos_pri,pos_tax,**flt_flt_mult**()) → pos_tot	void	flt	D_0A_8	A_{12}	3.6M	14.4	54.9	54.9	0.0
	free(pos_pri,pos_tax) → *in_memory*(itm_sel,pos_tot,grp_sel)						-10.8	44.1	44.1	0.0
14	**positional_equi_join**(grp_sel,sel_pos) → grp_pos	e_1[oid]	void	A_{13}	D_0A_8	3.6M	3.6	47.7	47.7	0.0
	free(itm_sel,grp_sel) → *in_memory*(grp_pos,pos_tot)						-29.7	18.0	18.0	0.0
15	**array_equi_join**(grp_pos,pos_tot) → grp_tot	e_1[oid]	flt	A_{13}	A_{11}	3.6M	28.8	46.8	46.8	0.0
	free(grp_pos,pos_tot) → *in_memory*(grp_tot)						-28.0	28.8	28.8	0.0
	create(grp_tot, "hash", 5)						17.3	46.1	46.1	2.9
16	**chr_norfix_hash_pump**(grp_tot,grp_grp,**flt_scan_sum**()) → grp_sum	e_1[oid]	flt	SKA_7	A_{14}	7	0.0	48.2	48.2	46.1

of committed memory ('use') and the minimum amount of memory each operator needs to fit its hot set into main memory ('hot').

3.4.1 Memory management

Query execution starts by loading the order_discount BAT. The vm_seq flag tells Monet to map the file that stores the BUN array into virtual memory, instead of directly loading it, and to give the OS virtual-memory advice, telling that access to this mapped region will be sequential. Given such advice, the Solaris OS uses memory prefetching and DMA to load large chunks of pages in the background, while the CPU can continue processing. It also places all swapped-in pages directly on the swap-out list. This has the effect that when a MIL operator sequentially scans the BAT, the OS uses only a few memory pages to swap it in.

Up to line 15, the hot set column of the execution trace is zero, since all algorithms used had sequential access. Such sequential main-memory algorithms take optimal profit from complex bus and memory cache architectures found in modern computing hardware, and have a small hot set, so they may process virtual-memory sizes that exceed the main-memory size and still deliver good performance. The pump operator in line 16, however, first constructs a hash table with the create() operator, and then executes chr_norfix_hash_pump() that uses this hash table to iterate over the group oids from grp_grp. The pump has random access to both the BUN array and the hash table; its hot set hence is 46.1 MB. The tactical optimization of the pump actually computes this number beforehand, and compares it with the run time statistics on available memory. If too little is available, it could decide to go for the chr_norfix_merge_pump() implementation that first sorts the grp_tot BAT. This algorithm performs a little more slowly in main memory, but requires a smaller hot set (and also profits from the *slice view* technique from Sect. 3.3.2). Problems with hot sets that do not fit the main memory can also be prevented by generating 'pipelined' MIL programs (see Sect. 2.3.3).

3.4.2 Join processing

The tactical optimization resolves all joins in lines 3, 7, 11, 12 and 14 to *positional* joins. This is a cheap kind of join that computes a BUN array position by subtracting the *seqbase* property value from each oid that is looked up. Such joins normally have a hit rate of exactly 1, in which case the non-join column of the smaller operand reappears identically in the result. The positional_equi_join() exploits this by propagating a void non-join column to the result. Due to this optimization, the intermediate BATs stays relatively small.

[7] Properties are abbreviated as follows: $S \Leftrightarrow$ column is sorted, $K \Leftrightarrow$ column is key, $D_{seqbase} \Leftrightarrow$ column is densely ascending from $seqbase$ (also implies SK), A_{id} is the alignment-id of a column (in order to easily detect equal columns).

The join in line 15 is even simpler: from the equality of the *align* properties of the join columns (head of `pos_tot` and tail of `grp_pos`) it can be deduced that both columns are identical. This join can hence be constructed without any lookup by simply iterating through both BATs and combining both tuples in the join result. A similar condition in line 13 leads to execution of the map operator [*] with `e2fix_e1fix_array_map`. The 'e2fix_e1fix' indicates a macro-expanded implementation for a binary map operator on two BATs, of which the first has a `e2[]` tail type that encodes fixed-size values, and the second a `e1[]` tail type that also encodes fixed-size values. As explained in Sect. 3.3.5, all unary, binary, and tertiary combinations of { 'e1', 'e2', 'nor' } and { 'fix','var' } are coded out in 69 separate `XX_array_map()` implementation routines. On line 4, for example, we used the `norfix_array_map()` for a unary [year] map.

In the OQL query, the joins of lines 3, 7, 11, 12 and 14 are not present; they are a necessary result of the vertical fragmentation applied in Monet. It is interesting to see that tactical optimization is sufficient to neutralize the extra joins introduced by vertical fragmentation. All such joins get executed by the positional or array implementations, which just move data without lookup. This effectively eliminates the disadvantages of vertical fragmentation, and leaves us with its advantages, namely a reduction of I/O cost. Whereas our example query (with its 60% selectivity) would require a full scan of 1 GB of relational data structures, Monet just scans 40 MB of data from disk. In query-intensive tasks like these, we have shown in the TPC-D and DD Benchmarks that Monet achieves an order of magnitude of performance improvement over conventional DBMS technology [2, 3].

4 Conclusion

The research described in the paper reports on progress in three key areas of modern database management. First, we defined a simple-yet-powerful query algebra on the binary table model. The MIL language is quite small, yet has proven successful in supporting relational, object-oriented and other database applications efficiently. It provides constructs for parallelism and is extensible in all its dimensions.

Second, we show how important physical query optimization decisions can be deferred to run time, both enhancing their quality and simplifying the query optimization process. What makes MIL stand out from other database languages is that it is both a logical and an execution language. By providing a direct implementation for this logical algebra language, the Monet system separates query optimization in a strategical and a tactical phase. Query-optimizing systems producing MIL code must transform a high-level query in a sequence of appropriate MIL primitives. This includes determining a good (join)order, but excludes translation to physical primitives. Choosing an algorithms is performed at run time inside each MIL operator during the process of tactical query optimization. By using *properties* maintained on relation fragments and full propagation of these properties across operators, MIL conserves maximum information about the data that is being processed, which is also combined with run time system statistics. By off-loading these activities from the compile-time phase, this approach simplifies the task of query optimizers, and results in a system that takes better decisions, as more run time information is taken into account.

Third, it provides insight in the techniques employed and lessons learned by implementing MIL in the Monet system. We were successful in achieving our goal of constructing a system that provides high performance on query-intensive application areas like and data mining [2–4, 6]. This success can firstly be attributed to the use of vertical fragmentation, which enables MIL to avoid much I/O otherwise spent in table scans and reduces the volumes of data movement during query processing. Another reason is that the simple and concise definition of MIL allowed us to put much optimization effort in its implementation. Techniques like implicit storage, type remappings and view implementations were effective in eliminating the extra join-overhead that is encountered when relational or object-oriented applications are fully decomposed into the binary table model. Main-memory optimization methods like code expansions were employed throughout the system to make it perform well on modern hardware architectures, in which memory latency and bandwidth are increasingly limiting factors for achieving high performance.

At this moment, we have a mature implementation of Monet that is deployed on a commercial basis as part of the data mining toolkits of Data Distilleries [8], including a packaged version as an Oracle data cartridge. In our group, Monet is also used as a research system in various areas, like multi-media, scalable distributed data structures, and cost modeling. We plan to publish parallel performance results on large versions of the TPC-D benchmark in the near future.

Acknowledgements. We would like to thank Carel van den Berg, Niels Nes, Wilko Quak, Annita Wilschut, Fred Kwakkel, Florian Waas, Arjen de Vries, Jonas Karlsson, Marco Fuykschot, Stefan Manegold, André van der Hoeven, Menzo Windhouwer and Jan Flokstra for their past and present contributions to the Monet system.

References

1. Analyti A, Pramanik S (1992) Fast search in main memory databases. In: Stonebraker M (ed) Proc. SIGMOD Conf, June 1992, San Diego, Calif. ACM Press, New York, pp 215–224
2. Boncz PA, Rühl T, Kwakkel F (1998) The Drill Down Benchmark. In: Gupta A, Shmueli O, Widom J (eds) Proc. VLDB Conf, August 1998, New York, N.Y. Morgan Kaufman, San Francisco, pp 628–632
3. Boncz PA, Wilschut A, Kersten ML (1998) Flattening an object algebra to provide performance. In: Proc. ICDE Conf, February 1998, Orlando, Fla. Computer Society Press, New York, pp 568–577
4. Boncz PA, Quak WC, Kersten ML (1996) Monet and its Geographical Extensions: A novel approach to high-performance GIS processing. In: Apers PMG, Bouzeghoub M, Gardarin G (eds) Proc. EDBT Conf, March 1996, Avignon, France. Springer, Berlin Heidelberg New York, pp 77–78
5. Boncz PA, Kersten ML (1995) Monet: An impressionist sketch of an advanced database system. In: Proc. BIWIT Workshop, July 1995, San Sebastian, Spain. Computer Society Press, New York, pp 240–251
6. Boncz PA, Kwakkel F, Kersten ML (1996) High performance OO traversals in Monet. Lecture Notes in Computer Science, Vol. 1094. Springer, Berlin Heidelberg New York, pp 152–160

[8] See `http://www.ddi.nl/`

7. Cattell RGG (1994) ODMG-93: A standard for OODBMSs. SIGMOD Rec 23(2): 480–480
8. Cherniack M, Zdonik SB (1996) Rule languages and internal algebras for rule-based optimizers. In: Jagadish HV, Mumick IS (eds) Proc. SIGMOD Conf, June 1996, Atlantic City, N.J. ACM Press, New York, pp 401–412
9. Colby LS, Cole RL, Haslam E, Jazayeri N, Johnson G, McKenna WJ, Schumacher L, White D (1998) Red brick vista: Aggregate computation and management. In: Proc. ICDE Conf, February 1998, Orlando, Fla. Computer Society Press, New York, pp 174–177
10. Cole RL, Graefe G (1994) Optimization of dynamic query evaluation plans. SIGMOD Rec 23(2): 150–160
11. Compaq Inc (1998) Whitepaper. Infocharger. Compaq Inc, Houston, Tex.
12. Copeland GP, Khoshafian S (1985) A decomposition storage model. In: Navathe SB (ed) Proc. SIGMOD Conf, May 1985, Austin, Tex. ACM Press, New York, pp 268–279
13. Danforth S, Valduriez P (1992) A FAD for data intensive applications. TKDE 4(1): 34–51
14. Doppelhammer J, Höppler T, Kemper A, Kossmann D (1997) Database performance in the real-world TPC-D and SAP-11/3. In: Peckham J (ed) Proc. SIGMOD Conf, June 1997, Tucson, Ariz. ACM Press, New York, pp 123–134
15. Gray J, et al. (1997) Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. Data Min Knowl Discovery 1(1): 29–53
16. Agarwal S, et al. (1996) On the computation of multidimensional aggregates. In: Vijayaraman M, Buchmann PA, Mohan C, Sarda NL (eds) Proc. VLDB Conf, July 1996, Bombay, India. Morgan Kaufman, San Francisco, pp 506–521
17. Garcia Molina H, Salem K (1992) Main memory database systems: An overview. TKDE 4(6): 509–516
18. Gerber B (1995) Informix online XPS. In: Carey MJ, Schneider DA (eds) Proc. SIGMOD Conf, May 1995, San Jose, Calif. ACM Press, New York, p 463
19. Graefe G (1993) Query evaluation techniques for large databases. ACM Comput Surv 25(2): 73–170
20. Graefe G, Ward K (1989) Dynamic query evaluation plans. SIGMOD Rec 18(2): 358–366
21. Güting RH (1989) Gral: An extensible relational database system for geometric applications. In: Apers PMG, Wiederhold G (eds) Proc. VLDB Conf, August 1989, Amsterdam, The Netherlands. Morgan Kaufman, San Francisco, pp 33–44
22. Informix Corp. (1998) MetaCube ROLAP User Guide. Informix Corp., Menlo Park, Calif.
23. Ioannidis YE, Christodoulakis S (1991) On the propagation of errors in the size of join results. SIGMOD Rec 20(2): 268–277
24. Kabra N, DeWitt DJ (1998) Efficient mid-query reoptimization of sub-optimal query execution plans. In: Haas LM, Tiwary A (eds) Proc. SIGMOD Conf, June 1998, New York, N.Y. ACM Press, New York, pp 106–117
25. Karlsson JS, Kersten ML (1998) Live optimization in SDDS join operations. Technical report. CWI, University of Amsterdam, Amsterdam, The Netherlands
26. Kenan Technologies (1995) An Introduction to Multidimensional Database Technology. Kenan Technologies, Camdrige, Mass.
27. Kersten ML, Boer MFN de (1994) Query optimisation strategies for browsing sessions. In: Proc. ICDE Conf, February 1994, Houston, Tex. Computer Society Press, New York, pp 478–487
28. Khoshafian S, Copeland G, Jagodits T, Bora H, Valduriez P (1987) A query processing strategy for the decomposed storage model. In: Proc. ICDE Conf, February 1987, Los Angeles, Calif. Computer Society Press, New York, pp 636–644
29. Lanzelotte RSG, Valduriez P, Zait M, Ziane M (1994) Industrial-strength parallel query optimization: issues and lessons. Inf Syst 19(4): 311–330
30. Lehman TJ, Carey MJ (1986) A study of index structures for main memory database management systems. In: Chu WW, Gardarin G, Ohsuga S, Kambayashi Y (eds) Proc. VLDB Conf, August 1986, Kyoto, Japan. Morgan Kaufman, San Francisco, pp 294–303
31. Nes NJ, Kersten ML (1998) The ACOI algebra: A query algebra for image retrieval systems. In: Embury SM, Fiddian NJ, Gray WA, Jones AC (eds) Proc. BNCOD Conf, July 1998, Cardiff, Wales. Springer, Berlin Heidelberg New York, pp 77–88.
32. Novak M, Gardarin G, Valduriez P (1994) Flora: A functional-style language for object and relational algebra. Lecture Notes in Computer Science, Vol. 856. Springer, Berlin Heidelberg New York, pp 37–46
33. O'Neil PE (1989) Model 204 architecture and performance. In: Gawlick D, Haynie MN, Reuter A (eds) Proc. International Workshop on High-Performance Transaction Systems, September 1987, Springer, Berlin Heidelberg New York, pp 40–59
34. Oracle Corp. (1997) Oracle8 SQL Reference. Oracle Corp., Redwood Shores, Calif.
35. Oracle Corp. (1997) Whitepaper. Express Server: Delivering OLAP to the Enterprise. Oracle Corp., Redwood Shores, Calif.
36. Pellenkoft A, Galindo-Legaria CA, Kersten ML (1997) The complexity of transformation-based join enumeration. In: Jarke M, Carey MJ, Dittrich KR, Lochovsky FH, Loucopoulos P, Jeusfeld MA (eds) Proc. VLDB Conf, August 1997, Athens, Greece. Morgan Kaufman, San Francisco, pp 306–315
37. Selinger PG, Astrahan MM, Chamberlin DD, Lorie RA, Price TG (1979) Access path election in a relational database management system. In: Bernstein PA (ed) Proc. SIGMOD Conf, May 1979, Boston, Mass ACM Press, New York, pp 23–34
38. Shatdahl A, Kant C, Naughton JF (1994) Cache-conscious algorithms for relational query processing. In: Bocca JB, Jarke M, Zaniolo C (eds) Proc. VLDB Conf, September 1994, Santiago, Chile. Morgan Kaufman, San Francisco, pp 510–521
39. Silicon Graphics Inc. (1997) Origin Servers Technical Report, April. Silicon Graphics Inc., Mountain View, Calif.
40. Stonebraker M, Anton J, Hirohama M (1987) Extendability in POSTGRES. Database Eng Bull 10(2): 16–23
41. Subramanian B, Leung TW, Vandenberg SL, Zdonik SB (1995) The AQUA approach to querying lists and trees in object-oriented databases. In: Yu PS, Chen LP (eds) Proc. ICDE Conf, March 1995, Taipei, Taiwan. Computer Society Press, New York, pp 80–89
42. Sybase Corp (1996) Whitepaper: Adaptive Server IQ. Sybase Corp., Emeryville, Calif.
43. Transaction Processing Performance Council (1995) TPC Benchmark D, 1.2.3 edition. Transaction Processing Performance Council, San Jose, Calif.
44. Valduriez P (1987) Join indices. ACM Trans Database Syst 12(2): 218–246
45. Berg CA van den, Kersten ML (1994) An analysis of a dynamic query optimization scheme for different data distributions. In: Freytag JC, Vossen G, Maier D (eds) Query Processing for Advanced Database Applications. Morgan Kaufmann, San Francisco, Calif., pp 449–470
46. Whang K, Krishnamurthy R (1990) Query optimization in a memory-resident domain relational calculus database system. ACM Trans Database Syst 15(1): 67–95
47. White SJ (1994) Pointer Swizzling Techniques for Object-Oriented Database Systems. PhD thesis. University of Wisconsin, Madison, Wis.
48. Wu MC, Buchmann AP (1998) Encoded bitmap indexing for data warehouses. In: Proc. ICDE Conf, February 1998, Orlando, Fla. Computer Society Press, New York, pp 220–230