High Performance support for OO traversals in Monet

P. A. Boncz, F. Kwakkel, M. L. Kersten
CWI, University of Amsterdam
{ boncz, kwakkel, mk}@cwi.nl *

Abstract

In this paper we discuss how Monet, a novel multi-model database system, can be used to efficiently support OODB applications. We show how Monet's off-beat view on key issues in database architecture provided both challenges and opportunities in building a high-performance ODMG-93 compliant Runtime System on top of it.

We describe how an OO data-model can be mapped onto Monet's decomposed storage scheme while maintaining physical data independence, and how OO queries are translated into an algebraic language. A generic model for specifying OO class-attribute traversals is presented, that permits the OODB to algebraicly optimize and parallelize their execution.

To demonstrate the success of our approach, we give OO7 benchmark results of our Runtime System for both the standard pointer-based object navigation, and our declarative model that uses a path-operator traversal library.

Keywords: Object oriented databases, Performance, Benchmarking, Database programming languages Database architectures, Database Techniques, Parallel Systems.

1 Introduction

Engineering design and CASE are the prototypical database applications that require the database system to support complex and evolving data structures. Queries often involve -hierarchical- traversals and have to be executed with high performance to satisfy the requirements posed by an interactive application.

OODBs have been identified as the prime vehicle to fulfill these tough demands. It is in these application domains that traditional RDBMSs suffer most from the impedance mismatch, and fail to deliver flexibility and performance [7]. In recent years several – commercial – OODBs have entered the marketplace. Since "performance" in CAD/CAM or CASE applications has many faces, the OO7 benchmark was introduced as a yardstick for their success. It measures traversal-, update- and query evaluation performance for databases of differing sizes and object connectivity. The results published [5] indicate room for further improvement and a need for more effective implementation techniques.

This article describes how we tackled the OO7 functionality with our ODMG-93 compliant Runtime System called MO_2 [17] that was developed on top of Monet [4]. Monet is a novel database kernel that uses the Decomposed Storage Model (DSM [6]) because of its effectiveness in main-memory environments. Through its use of virtual-memory techniques and operating system facilities for buffer management, Monet has been proven capable of handling both small and huge data-volumes efficiently [3].

Monet is a *multimodel* system; this means that its data can be viewed simultaneously in a relational, binary set-algebraic, or object-oriented manner. The MO_2 system is put at the task of translating between Monet's DSM- and the object-oriented data-model. This translation provides opportunity for optimization. The *lazy attribute fetching* technique employed in MO_2 is an example of this (see Section 3.1).

From the viewpoint of an OODBS, traversals specified in a persistent programming language like C++, result in a waterfall of individual object-fetches which is hard to optimize. Helped by the physical data independence present in the MO_2 system, we managed to improve on this by offering a generic model for specifying complex traversals at a high level, in the form of a template class-library. Traversals specified with this model can seamlessly be integrated with set-oriented query optimization and parallelization, for efficient execution on Monet.

*Part of this work was supported by SION grant no. 612-23-431

2 Monet Overview

Monet is a novel database server under development at the CWI and University of Amsterdam since 1992. It is designed as a backend for different data models and programming paradigms without sacrificing performance. Its development is based on our experience gained in building PRISMA [1], a full-fledged parallel main-memory RDBMS running on a 100-node multi-processor, and current market trends.

Developments in personal workstation hardware are at a high and continuing pace. Main memories of 128 MB are now affordable and custom CPUs currently perform over 50 MIPS. They rely on efficient use of registers and cache to tackle the disparity between processor and main-memory cycle time, which increases every year with 40% [13]. These hardware trends pose new rules to computer software – and to database systems – as to what algorithms are efficient.

Another trend has been the evolution of operating system functionality towards micro-kernels, i.e. those that make part of the Operating System functionality accessible to customized applications. Prominent prototypes are Mach, Chorus and Amoeba, but also conventional systems like Silicon Graphics' Irix and Sun's Solaris increasingly provide hooks for better memory and process management.

2.1 Design Principles

Given the motivation and design philosophy outlined above, we applied the following ideas in the design of Monet:

- *perform all operations in main – virtual – memory.* Monet makes aggressive use of main-memory by assuming that the database hot-set fits into main-memory. All its primitive database operations work on this assumption, no hybrid algorithms are used. For large databases, Monet relies on virtual memory by mapping large files into memory. In this way, Monet avoids introducing code to 'improve' or 'replace' the operating system facilities for memory/buffer management. Instead, it gives advice to the lower level OS-primitives on the intended behavior [1]. As Monet's tables take the same form on disk as in memory (no pointer swizzling), this memory mapping technique is completely transparent to its main-memory oriented algorithms.

- *binary relation model.* Monet stores all information in Binary Association Tables (BATs, see Figure 1). Search accelerators are automatically introduced as soon as an operator would benefit from their existence. They exists as long as the table is kept in memory; they are not stored on disk.

  This Decomposed Storage Model (DSM) [6] facilitates object evolution, and saves IO on queries that do not use all the relation's attributes. Saving on data movements is especially beneficial in main-memory environments, and easily outwheighs the extra cost for re-assembling complex objects, before they are given to an application.

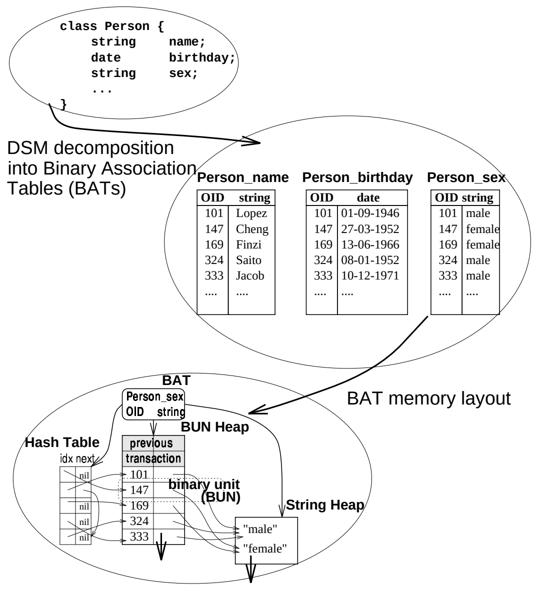

Figure 1: Monet's decomposed storage scheme

- employ *inter-operation parallelism.* Monet exploits shared-store and all-cache architectures. Unlike mainstream parallel database servers, like PRISMA [1] and Volcano [10], Monet does not use tuple- or segment-pipelining. Instead, the algebraic operators are the units for parallel execution. Their result is completely materialized before being used in the next phase of the query plan. This approach benefits throughput at a slight expense of response time and memory resources.

  A version of Monet designed to exploit efficiently distributed shared-nothing architectures is described in [15, 16]. A prototype runs on IBM/SP1.

[1]This functionality is achieved with the `mmap()`, `madvise()`, and `mlock()` Unix system calls.

- allow users to *customize* the database server. Monet provides extensibility much like in Gral [11], where a command can be added to its algebra, and its implementation linked into the kernel at any time. The Monet grammar structure is fixed, but parsing is purely table-driven on a per-user basis. Users can change the parsing tables at runtime by loading and unloading modules.

2.2 Algebraic Interface

Monet has a textual interface that accepts a set-oriented programming language called MIL (Monet Interface Language). MIL provides basic set operations, and a collection of orthogonal control structures that also serve to execute tasks in parallel. The – interpretive – MIL interface is especially apt as target language for high-level language interpreters (SQL or OQL), allowing for modular algebra translation [11], in which parallel task generation is easy. Algorithms that translate relational calculus queries to BAT algebras can be found in [12, 15]

We show in an example what the BAT algebra looks like. Consider the following SQL query on relations `company [comp#,name,telephone]` and `supply [supply#,comp, part, price]`:

```
SELECT company.name,
       company.telephone,
       supply.quantity
FROM   company, supply
WHERE  supply.comp = company.comp# AND
       supply.part = part_no AND
       supply.price < 0.50)
```

In Monet's SQL frontend, the relational database scheme will be vertically decomposed into five tables named `comp_name`, `comp_telephone`, `supply_comp`, `supply_part` and `supply_price`, where in each table the *head* contains an OID, and the *tail* contains the attribute value. The SQL query gets translated to the following MIL block:

```
{
   VAR m_supply, m_comp;
   VAR m_name, m_telephone, m_quantity;

   m_supply := SEMIJOIN(supply_part.SELECT(part_no),
                        supply_price.SELECT(0.0, 0.50));
   m_supply := MARK(m_supply);
   m_comp := JOIN(m_supply, supply_comp);
   [
      m_name      := JOIN(m_comp, comp_name);
      m_telephone := JOIN(m_comp, comp_telephone);
      m_quantity  := JOIN(m_supply, supply_quantity);
   ]
   PRINT(m_name, m_telephone, m_quantity);
}
```

In all, the original double-select, single-join, three-wide projection SQL query is transformed in a sequence of 8 BAT algebra commands. The three last joins are placed in a parallel block ([]).

The dot notation "`a.oper(b)`" is equivalent to function call notation "`oper(a,b)`". Note that `JOIN` projects out the join columns. The `MARK` operation introduces a column of unique new OIDs for a certain BAT. It is used in the example query to create the new – temporary – result relation. The below table describes in short the semantics of the BAT commands used:

BAT command	result
<AB>.mark	$\{o_i a \mid ab \in AB \wedge unique\_oid(o_i)\}$
<AB>.semijoin(CB)	$\{ab \mid ab \in AB, \exists cd \in CD \wedge a = c\}$
<AB>.join(CD)	$\{ad \mid ab \in AB \wedge cd \in CD \wedge b = c\}$
<AB>.select(Tl,Th)	$\{ab \mid ab \in AB \wedge b \geq Tl \wedge b \leq Th\}$
<AB>.select(T)	$\{ab \mid ab \in AB \wedge b = Tl\}$
<AB>.find(T)	$\{a \mid aT \in AB\}$

3 MO$_2$: ODMG programming on Monet

When one would integrate a C++ application with a MIL (or even SQL) speaking Monet server, there is an impedance mismatch. To remedy this, in the MAGNUM project (underway since 1994) at the University of Amsterdam and CWI we have developed a ODMG-93 compliant database system nicknamed MO$_2$, that will (amongst others) be used as a tool to build a high-end GIS application.

The ODMG system consists of two parts: an ODL parser and an ODMG runtime library to be bound with the C++ application [8].

3.1 ODL Parser

An ODL definition defines both persistent and transient classes managed by the database system. ODL offers the usual OO features like inheritance, methods, objects and values, constructors (Set, Array,List, Struct etc) and binary relationships between objects. Note that ODL only defines the signature of object methods, the implementation must be done in the application language for which an ODL language binding exists. In our case we support both a C++ - and Java language binding.

The ODL parser accepts the data definitions, inserts those in the OODB data dictionary, and generates C++ header files with the corresponding class definitions.

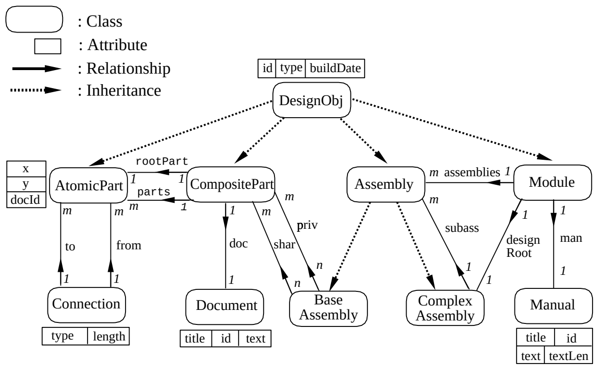

Figure 2: OODB schema: the OO7 database

In the C++ language binding, object references are implemented in a template class `Ref<T>` following a "smart pointer" approach.

The mapping of the ODMG object model to the Monet data-model is influenced by three considerations:

- The mapping must allow translation of OQL to an efficient MIL program. In particular this means that object reconstruction from the decomposed tables – which is costly – must be avoided, set operations are performed on object identifiers only, and indexes can be used for the execution of selections and join operations on arbitrary attributes.
- The mapping must offer a large degree of data independence. Both C++ and Java applications must be able to share the data stored in the database. Furthermore, the addition of attributes to existing classes must have a low impact on the stored objects.
- In object hierarchy traversals such as OO7 a large number of objects are visited but only a small fraction of the attributes of the intermediate objects is used. "Lazy attribute fetching" is therefore a good technique to reduce disk I/O.

Objects are therefore fully decomposed in our design. Each attribute is stored as binary relation between the object identifier and the attribute value in `class_attribute` BATs. Each relationship is represented by a binary relation between the object identifiers of the two related objects.

In the case of the OO7 data-model (see Figure 2) the complete mapping of the 10 classes, 33 attributes, and 11 relationships of the OO7 database leads to 54 BATs in the Monet database.

3.2 ODMG Runtime

The ODMG runtime library deals with runtime object management. ODMG unites two paradigms, namely C++ persistent programming, where pointer traversals seamlessly permits browsing through the database tables, and the OQL part – a high level query language for bulk data retrieval.

Moving from a non-standardized OODB – as there exist many – to an ODMG compliant one, calls for a tight integration of the high-level OQL into the existing persistent programming system. Since OQL favors function-shipping solutions, preferably using complex query optimization and parallelization of tasks, there is a conflict of data allocation strategies, because the pointer traversal part of ODMG requires a data-shipping strategy to achieve efficiency.

The architecture of the Monet ODMG runtime system therefore incorporates a dual design. The client and server are peer-to-peer systems with different roles. The server system(s) are masters of their data: they are responsible for transaction management. The client, however, contains a functional complete copy of the server code to manage *transient* databases. This way, it can choose at runtime to cache tables and to execute operations locally or remotely.

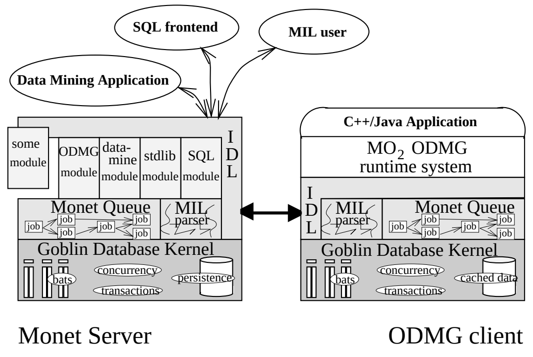

Figure 3: The Monet Server and its Clients

Hiding the object implementation through the use of accessor functions `get_attr()` and `put_attr()` eliminates the need for object reconstruction at the client for navigational access. In other words, the data representation is the same at the client and server, so that query processing can be performed at either side. For this purpose the accessor functions and method code for new classes can be dynamically linked to the database server program.

At the moment of this writing, navigational access is supported and work is well underway to support full

OQL. Section 5 presents the performance results for the traversal queries of the OO7 benchmark. In the future we will further optimize the Runtime System and reduce the granularity of locking it applies.

4 Efficient Object Traversals

Though OODBs provide for a seamless integration of application and database, they have also been criticized on the grounds that what effectively happens in OO class-attribute traversals is CODASYL-like "pointer chasing" [9]. Apart from the data independence issue, a piece of C++ code – say a complex loop – with object-referencing operations inside cannot be easily analyzed by the OODB to optimize complex traversals, let alone parallelize them. Such tasks are left to the programmer.

We think that this aspect of OODBs is a step backwards. For this reason, we felt there is a need for a way to specify OODB class-attribute traversals at a higher-level level of abstraction, such that they become amendable for optimization and efficient processing using a parallel platform.

Such high level constructs – generic as they may be – will never posses the expressive power as an arbitrary piece of C++ program. However, the model presented here captures a wide spectrum of traversals encountered in practice, at least those specified in the OO7 traversal queries.

4.1 Some Definitions

Our class traversal primitives require a few introductory definitions. We assume a class-attribute graph $\mathcal{G}$, which is a directed graph, where the nodes are classes, and the vertices the `relation` attributes in them.[2] These *attribute-vertices* start at the class of which the attribute is a member, and point to the class of the attribute. The class-attribute graph $\mathcal{G}$ also captures a second set of vertices representing inheritance relations, forming a forest of DAGs. Unless stated otherwise, when we use the term "reachable" we mean "reachable by attribute-vertices".

We define the following functions:

$$\begin{array}{l} classes(\mathcal{G}) = \{C \mid C \text{ is a node in } \mathcal{G}\} \\ subclasses(P, \mathcal{G}) = \{C \mid P, C \in classes(\mathcal{G}), \\ \qquad C \text{ reachable from } P \text{ via inheritance-vertices}\} \end{array}$$

The class-attribute graph can have objects as *instantiations* associated with it, which are captured by the extent $\mathcal{E}$:

$$\begin{array}{l} \mathcal{E} = \{obj \mid obj \text{ instantiation of } C \in classes(\mathcal{G})\} \\ \mathcal{E}(P) = \{obj \mid obj \text{ instantiation of } C \in subclasses(P, \mathcal{G})\} \end{array}$$

The complex paths we want to use are broader than the definition given in [2], in the sense that we want to be able to specify paths containing cycles and multiple subpaths from one class to the other.

Below we inductively define all possible paths $C_1 \underset{P}{\overline{\ \ }} C_2$ as specified by a *path expression* $\underset{P}{\overline{\ \ }}$:

$$\begin{array}{rcll} \multicolumn{4}{l}{C_1 \in classes(\mathcal{G}) \wedge} \\ C_2 \in subclasses(C_1.\alpha) & \Leftrightarrow & C_1 \underset{C_1.\alpha}{\overline{\quad\quad}} C_2 & (1) \\ C_1 \underset{P_1}{\overline{\ \ }} C_2 \wedge C_2 \underset{P_2}{\overline{\ \ }} C_3 & \Leftrightarrow & C_1 \underset{P_1 - P_2}{\overline{\quad\quad\quad}} C_2 & (2) \\ C_1 \underset{P_1}{\overline{\ \ }} C_2 \wedge \ldots \wedge C_1 \underset{P_n}{\overline{\ \ }} C_2 & \Leftrightarrow & C_1 \underset{[P_1, .., P_n]}{\overline{\quad\quad\quad\quad}} C_2 & (3) \end{array}$$

The notion of *reachability* of two objects $o_1 \in \mathcal{E}(C_1)$, $o_n \in \mathcal{E}(C_n)$, by a path $C_1 \underset{P}{\overline{\ \ }} C_n$ can similarly defined over path expressions $\underset{P}{\overline{\ \ }}$ as:

$$\begin{array}{rcll} o_1 \underset{C_1.\alpha}{\overline{\quad\quad}} o_n & \Leftrightarrow & o_1.\alpha = o_n & (4) \\ o_1 \underset{P_1 - \ldots - P_{n-1}}{\overline{\quad\quad\quad\quad\quad}} o_n & \Leftrightarrow & \forall i, 1 \leq i < n : o_i \underset{P_i}{\overline{\ \ }} o_{i+1} & (5) \\ o_1 \underset{[P_1, \ldots, P_{n-1}]}{\overline{\quad\quad\quad\quad\quad}} o_n & \Leftrightarrow & \exists i, 1 \leq i < n : o_1 \underset{P_i}{\overline{\ \ }} o_n & (6) \end{array}$$

A concrete instantiation $o_1.\alpha_1 - o_2.\alpha_2 - .. - o_n$ of a path $C_1 \underset{P}{\overline{\ \ }} C_2$between objects $o_1 \in \mathcal{E}(C_1)$ and $o_n \in \mathcal{E}(C_2)$ is called a *link*. The fact that two objects are reachable by a path implies that there is a (set of distinct) link(s) between them, following that path, and vice versa.

With this in mind, we can finally define the function `reachable`$(\frac{o_1}{P})$ as the *bag* of objects o_n, corresponding 1-1 to all distinct links $o_1.\alpha_1 - .. - o_n$ that are instantiations of path $C_1 \underset{P}{\overline{\ \ }} C_2$, where $o_1 \in \mathcal{E}(C_1) \wedge \forall o_n : o_n \in \mathcal{E}(C_2)$. We refer to the order in which the objects are put into the bag (which is arbitrary), as the *traversal order*.

4.2 Path Operators

Path expressions provide a handle to define the concept of *path operators*, which specify complex traversal patterns. Because the order in which a traversal algorithm visits the objects in a class-attribute hierarchy is important, we will define our path operators using pseudocode. Also, a traversal algorithm can visit a node more than once. It is for these reasons, that the path operators defined here work on ordered sets and ordered bags.

The `traverse()` traverses a path $\underset{P}{\overline{\ \ }}$, starting from one object o_1, producing the `reachable`$(\frac{o_1}{P})$ bag of objects as a result:

```
FUNCTION traverse(src: OBJECT; p: PATHEXP;
                  f: FUNCTION) : BAG;
  VAR obj: OBJECT, dst : BAG;
```

[2]We only describe relation attributes here, since value-attributes are not important in path definitions.

```
  FORALL obj IN reachable(src, p) DO
    IF (f) THEN f(src,obj) FI;
    append(dst, obj);
  OD;
  RETURN dst;
END;
```

This trivial piece of pseudocode implies that some ordering criterion exist in visiting the destination objects reachable from the source object. The definition also allows some function to be executed on all nodes when they are visited.[3]

The second operator computes the closure set over a *cyclic* path starting at an input bag:

```
FUNCTION closure(src: OBJECT; dst: INOUT SET;
                  p: PATHEXP; f1, f2: FUNCTION);
  VAR obj: OBJECT, dst SET;
  FORALL obj IN reachable(src, p) DO
    IF (f1) THEN f1(src,obj) FI;
    IF (NOT obj IN dst) THEN
      append(dst, obj);
      closure(obj, dst, path, f1, f2);
      IF (f2) THEN f2(src,obj) FI;
    FI;
  OD;
  RETURN dst;
END;
```

This version of the closure operation uses a depth-first algorithm. Likewise, a breadth-first closure and other variants can be defined. Again, user-defined functions can be executed on the nodes when they are visited, or just when they are included in the closure.

4.3 Executing Path Operators

Path operators can be nested to specify complex traversal patterns. For example, the expression $traverse(closure(input,A.b{-}B.a),[A.c{-}C.e,A.d{-}D.e])$ starts traversal at the bag *input* and computes the closure over the loop $A\overline{\underset{A.b-B.a}{}}A$ (see Figure 5) and uses this set as input to traverse the two-branched path $A\overline{\underset{[A.c-C.e,A.d-D.e]}{}}E$ between A and E (see Figure 4).

Efficient execution our path operators by Monet is relatively easy. As illustrated by the pseudo-code, the `traverse()` operator is equivalent to the join operator in Monet (see Section 2.2). Since the Monet's decomposition model vertically fragments all classes in binary relations named "`class_attribute`", traversing a path $P = A.a_1 - ... - A.a_n$ means joining the input bag with relation $\texttt{A\_a}_1$, then joining its result with $\texttt{A\_a}_2$, and again, until the last join with $\texttt{A\_a}_n$, which forms the result of the operator. If the path has multiple branches $P = [P_1, ..., P_n]$, all branches are traversed first, and the results united (see Figure 4).

[3]further on, we will use a `traverse()` on an ordered source bag rather than a single source object, which executes a traverse on all its elements in order, returning the concatenation of all resulting bags.

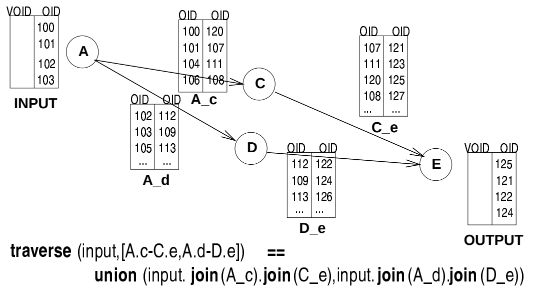

Figure 4: Executing a `traverse()` operator

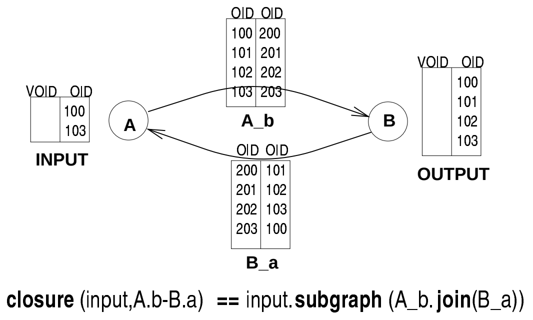

Figure 5: Executing a `closure()` operator

The `closure()` is implemented in Monet using the MIL operation `subgraph()`, which expects a single `[oid,oid]` BAT as the path relation. In contrast to `traverse()`, the `closure()` does not start joining the input bag with the "`class_attribute`" tables. Instead, it starts reducing the relations along the path *internally* using joins and unions until only a singular `[OID,OID]` path relation remains. Only then, it is fed as parameter with the input bag into Monet's `subgraph()` command.

4.4 Traversal Optimization

As opposed to a C++ pointer-based traversal, our nested path traversal operators give the OODB the whole view, such that it can work set-at-a-time and can optimize and parallelize traversal execution.

The relation attributes that connect the classes are essentially *join indices* [14], or OODB *nested indices* [2]. The closure operator first transforms a complex path to a singular one. In fact, it constructs a join index between the starting class and the ending class (which for `closure()` are one and the same). Although the construction cost for a join index maybe high, this investment can be turned into profit by reusing it in later traversals.

For the traversal operators, the join index technique may also be applied. A complete path or (multiple) subpaths in the path can be collapsed into a join index. It is clear, that the OODB optimizer has many options, of which some will, and others will not, be beneficial. The OODB should therefore use cost models incorporating parameters like mean fanout from objects along the classes-attribute path, the number of tuples in the input bag, the likelihood that a certain traversal will be executed again and others [14].

Path operators provide possibilities for parallel execution of traversals. Since both the `traverse()` as the `closure()` execute independent algorithms for the elements on their input bag, this bag can be fragmented, and given to different processing nodes for execution. The only communication required between nodes is at the end, when the results are united.

This article does not seek to investigate join index or traversal optimization cost-models; we just want to point out that high-level traversals provide opportunities for optimization and parallelization. The benchmark numbers in Section 5 have been obtained without attempting any optimization or parallelization.

4.5 ODMG Traversal Library

The high-level traversals discussed in this section are incorporated in Monet's ODMG system by means of a C++ template library. The library allows the programmer to construct nested path operators on his own classes.

```
// Path Class
template <class A, class B>
   class Path {
      Path(char *attr);           // outgoing attribute
      ~Path();
   }

// Path construction: && and || operators
template<class A, class B, class C>
   Path<A,C> Path::operator&&
      (const Path<A,B>& left,const Path<B,C>& right);

template<class A, class B>
   Path<A,B> Path::operator||
      (const Path<A,B>& left,const Path<A,B>& right);

// Path Operators: Traverse, Closure and Nest
template <class A, class B>
  class PathOperator {
     Bag<B> collect(Collection<A> root);
     int visit(Collection<A> root);
  }

template <class A, class B>
   class Traverse : public PathOperator<A,B> {
       Traverse(Path<A,B> path, void (fcn*)(Ref<A>,Ref<B>);
       ~Traverse();
   }

template <class A>
   class Closure : public PathOperator<A,A> {
      Closure(Path<A> path, void (pre*)(Ref<A>,Ref<A>),
                            void (post*)(Ref<A>,Ref<A>));
      ~Closure();
   }

template <class A, class C>
   class Nested : public PathOperator<A,C> {
       template <class B>
         Nested(Operator<A,B> op1; Operator<B,C> op2);
       ~Nested();
   }
```

The template classes are a direct translation of the model previously defined. They allow for all type checking to be done at runtime.[4]

The library user should first build a path expression, using instances of the `Path` class. Using the && (*and*) and || (*or*) operators, complex paths of resp. equation (2) and (3) can be assembled. With such paths as parameters, the operators `Closure` and `Traverse` can be constructed on them. Path operators can be nested using the `Nested` operator. Materialization of a path operator is done by either the `collect()` method (which returns all visited objects), or the `visit()` method, which just returns a visit count.

4.6 Example

We now give an example of the use of these classes in a ODMG C++ binding, by showing how the OO7 Traversal 1 can be expressed:

```
void oo7_t1(char *dbname) {
  Set<BaseAssembly> baseassbly, assbly;
  Ref<Assembly>     root;
  Database          database;

  database->open(dbname);
  lookup(&root, "root");
  lookup(&baseassbly, "BaseAssembly.extent");

  Path <ComplexAssembly, Assembly> *p1 = Path("subAss");
  Closure <Assembly> *o1 = Closure(p1);
  assbly = p0->collect(root)->intersection(baseassbly);

  Path <CompositePart,AtomicPart> *p2 = Path("rootPart");
  Path <BaseAssembly,CompositePart> *p3 = Path("priv");
  Traverse <BaseAssembly,AtomicPart>*o2 = Traverse(p2 && p3)

  Path <Connection,AtomicPart> *p4 = Path("to");
  Path <AtomicPart,Connection> *p5 = Path("to");
  Closure <AtomicPart> *o3 = Closure(p4 && p5);

  Nested<BaseAssembly, AtomicPart> *o4 = Nested(o2, o3);
  printf("T1 #visited objects: %d.\n", o4->visit(assbly));
  database->close();
}
```

[4]except for attribute names: to do that, the C++ syntax would have to be extended in some way.

We will now – step by step – show how the ODMG Runtime translates the above code to MIL: after some initialization, we fetch the *root* of the assembly hierarchy, and the extent of all *BaseAssemblies* (lines 7-8). This translates in the following MIL operations:

```
root := designRoot.find("root");
baseassbly := BaseAssembly;
```

Lines 10-12 specify the path expression: *closure(root,ComplexAssembly.subass)* of which the result is intersected with the previously fetched *BaseAssemblies*. Monet constructs a BAT named *closure_root* with the previously fetched *root* as only element. Since the closure is to be computed on the single cyclic path *ComplexAssembly.subAss*, already represented in Monet by the binary association `ComplexAssembly_subass`, it doesn't have to be reduced to one with joins and unions. We just call the `subgraph()` with it, and the *closure_root* as source BAT, and semijoin the result with the BaseAssemblies.

```
closure_root := new(oid,oid);
closure_root.insert(root,root);
leaves := subgraph(closure_root,
                  ComplexAssembly_subAss).semijoin(baseAssembly)
```

To retrieve all private root parts that are reachable from the selected base assemblies, we have to do a *traverse(leaves,BaseAssembly.priv−CompositePart.rootPart)*. This traversal is executed in MIL with two joins:

```
rootparts := leaves.join(BaseAssembly_priv)
                   .join(CompositePart_rootPart);
```

The *traverse*() operator above can in fact be nested in the *closure(leaves,AtomicPart.to−Connection.to)*, by substituting it in for "*leaves*" to obtain the nested path operator, as specified in lines 14-23. To execute the closure, two steps have to be taken. First, the closure path has to be reduced to a single relation, as follows:

```
connections := join(AtomicPart_to.reverse,
                    Connection_to);
```

(Note that joining two class-attribute relations might be an expensive operation, so an optimizing OODB might decide to save the *connections* BAT as a join-index for later re-use).

As a second step, the closure has to be executed on all elements in the *leaves* BAT, which contains all starting positions. Since we materialized the operator with `visit()`, we only have to return a visit count:

```
visited := 0; tmp := new(oid,oid);
rootparts@batloop() {
    tmp.clear;
    tmp.insert($2,$2);
    visited := visited +
            tmp.subgraph(connections).count;
}
```

The above piece of script, in which `visited` contains the final result, is the most expensive part of the T1 traversal. The *"@batloop()"* is a cursor-like Monet iterator, that iterates through all elements of a BAT, executing a MIL statement-block on all of them. Iterators in MIL can also be invoked in parallel: putting *"@[N]batloop()"* would have executed the block on at most N elements from the BAT in parallel, providing an easy way to parallelize traversals.

5 OO7 Experiments

Our client platform was a Sun SPARCstation 20/50Mhz running Solaris, with 96 MB main-memory, 16KB data-, 20KB instruction and 1MB secondary cache, 1 Gb local disk (raw throughput: 10 MB/s) and 0.5 GB swap space. It was connected with a similar Monet server via non-exclusive NFS over 10Mbit non-isolated ethernet. In this section we present results for three systems:

- MO_2 with C++ pointer-based navigational access. The implementation is a 95% match with the standard implementation available with the OO7 benchmark (see `ftp.cs.wisc.edu/oo7`).
- MO_2 with the Path Operator primitives. This implementation has only been done on the real traversal queries: T1 and T6 (full and sparse traversal), T2a,b,c (sparse,full and 3-fold attribute swap), and T3a,b,c (sparse,full and 3-fold indexed attribute update).
- A competitor from [5] as a reference. Since E/Exodus was the overall winner, we used these numbers. They have been hardware corrected with a factor 5.

All numbers mentioned with the experiments are in seconds of elapsed time.

5.1 Traversal Queries

In the below table, we observe that for navigational MO_2 the numbers for the small database are slightly worse than the competition. Here Monet's approach with hash-lookups instead of pointer swizzling proves to bear slightly more overhead. For the medium database, the performance is very similar. We expect to improve the performance on sparse traversals (T6) by optimizations in the ODMG Runtime. The T1 is clearly faster, because of Monet's use of DSM: only the starting and ending points of the traversal must be materialized completely. This means that all attributes of intermediate classes, that are not relational attributes or not involved in the traversal, need not be accessed.

	Competitor (E/Exodus)			**MO_2** (navigational access)			**MO_2** (path operators)		
fanout	**3**	**6**	**9**	**3**	**6**	**9**	**3**	**6**	**9**
	Small Database, Cold								
T1	7.0	8.5	10.1	10.6	17.0	23.4	3.6	15.1	8.6
T6	3.8	3.8	3.8	3.2	4.8	6.4	1.5	1.5	1.3
T2a	7.3	8.8	10.3	11.4	17.6	23.8	2.8	2.9	2.9
T2b	7.9	9.3	12.0	12.2	18.6	25.0	12.3	13.7	18.0
T2c	8.1	9.5	12.2	13.4	20.0	26.8	18.5	26.3	30.9
T3a	8.0	9.8	12.3	11.0	17.2	23.6	1.3	1.5	3.2
T3b	17.5	20.2	28.0	11.6	18.2	24.8	14.4	10.1	17.8
T3c	43.8	48.9	70.5	12.4	19.0	25.6	16.5	18.6	32.0
	Small Database, Hot								
T1	2.1	2.6	3.0	6.6	12.0	17.8	3.3	8.8	7.8
T6	0.2	0.2	0.2	0.3	0.3	0.3	1.0	1.6	1.0
	Medium Database, Cold								
T1	146.9	193.1	238.7	104.5	176.3	240.0	23.9	66.2	84.4
T6	5.9	5.9	5.9	12.6	14.0	17.0	1.5	4.1	1.6
T2a	151.9	200.2	246.8	116.2	201.9	314.9	7.2	8.2	9.3
T2b	193.7	245.6	290.0	138.3	205.1	320.3	52.1	103.5	135.7
T2c	192.7	242.4	291.8	152.1	244.0	352.0	94.1	151.8	192.5
T3a	166.5	216.8	266.0	113.2	200.0	313.3	1.4	1.6	1.8
T3b				119.9	212.9	318.8	47.4	103.6	152.1
T3c				134.4	229.8	330.8	92.4	143.1	237.3
	Medium Database, Hot								
T1				90.7	159.5	220.3	25.5	62.7	94.3
T6				0.3	0.4	0.5	1.1	2.2	2.1

Note that for MO_2 T2 (swap-update) is in general more expensive than update T3 (toggle on single – indexed – attribute), whereas all competitors have T2 cheaper than T3. This is because the decomposed storage model implies that T2 accesses two tables, whereas T3 only one. This outweighs the extra cost of adapting the index in T3.

As for the MO_2 numbers obtained with the path-operator class-library: with a factor 1-4 of difference this approach is the clear all-out winner. Algebraic MIL operations work set-at-a-time and are more efficient than doing many single object-traversals.[5] These results show crisp and clear that in order to achieve greater efficiency in complex OO traversals, one should use higher-level operations – whether it be path operators like we use, or some traversal-extension to a OQL-like language – such that the OODB can execute them efficiently, using standard query processing techniques (optimization, parallelization).

[5]The reason that T3a and T2a seem even two orders of magnitude faster, is that the path-operator implementation only visits the to-be-updated objects. If all objects would have been visited, times would have been more similar to T1

5.2 Queries T8-T9

Traversal T8 and T9 are operations on a large text block: *Manual_text* (1 MB). T8 counts all occurrences of the character 'i'. T9 compares the first and last character. The following performance figures are obtained for a medium database size (1MB of *Manual_text*).

Query	Memory map		Non Memory map	
	cold	hot	cold	hot
T8 (count)	0.654	0.602	0.999	0.558
T9 (compare)	1.343	0.955	1.898	1.123

Monet allows you to specify for all BATs one of two different memory managements strategies: loaded in memory, or mapped into virtual memory. For all measurements we used the latter strategy. This ensures BAT pages are swapped in memory only when required, and are therefore more efficient on low cache hit ratios. The performance data shows a faster response in the mapped version of T9 because here the first and last page are retrieved only.

5.3 Queries Q1-Q7

The performance results of the queries for the medium/9 database are shown below. We first discuss the results of Q1, in two different versions: in Q1 Monet only returns a set of OIDs, thereby taking advantage of its vertical decomposition strategy. The Q1a reconstructs tuples, by performing joins on the attributes. The difference in cold situations is due to BAT loading and building of index structures. The hot times clearly show the tuple reconstruction cost stays small (0.4 s).

Query	Cold time	Hot time	Memory Size
Q1	0.77s	0.56s	2.0-2.8 Mb
Q1a	1.82	0.94s	6.0-6.9 Mb

The other query results are as follows:

Query	Cold time	Hot time	Competitor
Q2 (1%)	0.62s	0.61s	> 4s
Q3 (10%)	0.67s	0.64s	> 7s
Q4	0.85s	0.64s	> 0.4s
Q5	0.84s	0.82s	> 3s
Q7A (100%)	1.94	1.78s	> 7s
Q7B (100%)	2.5	2.2s	> 7s

6 Conclusion

It was discussed how OO applications can be supported with Monet, a novel DBS with an unusual architecture. Firstly, our ODMG compliant MO_2 system maps an OO persistent programming interface onto decomposed tables in Monet. This mapping provides physical data independence, often hard sought for in OODBs. It also optimizes object-navigation by "lazy attribute fetching". Secondly, we introduced a derivate of the well-known concept of path-expressions to define path-operators. These provide an alternative – higher level – interface for specifying class-attribute traversals. We implemented the OO7 benchmark using MO_2 with and without this path-operator library. The comparison of results between the two and other OODBs shows that though bare MO_2 performs well, results superior to any competitor were achieved by the path operators, proving the importance of high-level query processing to performance in OODBs.

References

[1] P. M. G. Apers, C. A. van den Berg, J. Flokstra, P. W. P. J. Grefen, M. L. Kersten, and A. N. Wilschut. PRISMA/DB: A parallel main memory relational DBMS. *IEEE Trans. on Knowledge and Data Eng.*, 4(6):541, December 1992.

[2] E. Bertino and W. Kim. Indexing techniques for queries on nested objects. *IEEE Transactions on Knowledge and Data Engineering*, 1(2), June 1989. Also published in/as: Mathematisch Centrum (Amsterdam), now CMCSC, TR-ACT-OODS,132-89, Mar.1989.

[3] P. A. Boncz, , W. C. Quak, and M. L. Kersten. Monet and its Geographical Extensions: A novel approach to high performance GIS processing. In *Proc. EDBT'96 Conference, Avignon (France)*, March 1996.

[4] P. A. Boncz and M. L. Kersten. Monet: An impressionist sketch of an advanced database system. In *Proc. IEEE BIWIT workshop, San Sebastian (Spain)*, July 1995.

[5] M. Carey, D. J. DeWitt, and J. F. Naughton. The DEC OO7 benchmark. In *Proc. ACM SIGMOD Conf.*, page 12, Washington, DC, May 1993.

[6] G. Copeland and S. Khoshafian. A decomposition storage model. In *Proc. ACM SIGMOD Conf.*, page 268, Austin, TX, May 1985.

[7] J. Duhl and C. Damon. A performance comparison of object and relational databases using the Sun benchmark. In *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications, ACM SIGPLAN Notices*, page 153, November 1988.

[8] R.G.G. Catell et al. *The Object Database Standard*. Morgan Kaufman, 1993.

[9] et al. Neuhold,E. and Stonebraker,M. Future directions in DBMS research. *ACM SIGMOD RECORD*, 18(1), March 1989. Also published in/as: ICCS, Berkeley, TR-88-1, Sep.1988.

[10] G. Graefe. Encapsulation of parallelism in the volcano query processing system. In *19 ACM SIGMOD Conf. on the Management of Data, Atlantic City*, May 1990.

[11] R. H. Guting. Gral: An extensible relational database system for geometric applications". In *Proceedings of the 15th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Amsterdam*, August 1989.

[12] S. Khoshafian, G. Copeland, T. Jagodits, H. Boral, and P. Valduriez. A query processing strategy for the decomposed storage model. In *Proc. IEEE CS Intl. Conf. No. 3 on Data Engineering, Los Angeles*, February 1987.

[13] A.R. Lebeck and D.A. Wood. Cache profiling and the spec benchmarks: A case study. *IEEE Computer*, 27(10):15–26, October 1994.

[14] P. Valduriez. Join indices. *ACM Trans. on Database Sys.*, 12(2):218, June 1987.

[15] C. A. van den Berg. *Dynamic Query Optimization*. PhD thesis, CWI (Center for Mathematics and Computer Science), February 1994.

[16] C. A. van den Berg and M. L. Kersten. An analysis of a dynamic query optimisation scheme for different data distributions. In J. Freytag, D. Maier, and G.Vossen, editors, *Advances in Query Processing*, pages 449–470. Morgan-Kaufmann, San Mateo, CA, 1994.

[17] C. A. van den Berg and A. van den Hoeven. Monet meets OO7. In *OO Database Systems Symposium of the Engineering Systems Design and Analysis Conference, Montpellier (France)*, July 1996.