Centrum voor Wiskunde en Informatica
REPORT*RAPPORT*

High performance support for OO traversals in monet

P.A. Boncz, F. Kwakkel and M.L. Kersten

Computer Science/Department of Algorithmics and Architecture

CS-R9568 1995

Report CS-R9568
ISSN 0169-118X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

CWI is the National Research Institute for Mathematics and Computer Science. CWI is part of the Stichting Mathematisch Centrum (SMC), the Dutch foundation for promotion of mathematics and computer science and their applications.
SMC is sponsored by the Netherlands Organization for Scientific Research (NWO). CWI is a member of ERCIM, the European Research Consortium for Informatics and Mathematics.

Copyright © Stichting Mathematisch Centrum
P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

High Performance Support for OO Traversals in Monet

P. A. Boncz[1], F. Kwakkel[2], M. L. Kersten[2]

[1] *University of Amsterdam, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands*
boncz@fwi.uva.nl
[2] *CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands*
{kwakkel,mk}@cwi.nl

Abstract

In this paper we discuss how Monet, a novel multimodel database system, can be used to efficiently support OODB applications.

We show how Monet's offbeat view on key issues in database architecture provides both challenges and opportunities in building a high-performance ODMG-compliant programming system on top of it. In particular, we show how an OO datamodel can be mapped onto Monet's decomposed storage scheme, 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.

The complete OO7 benchmark has been implemented to measure the success of our approach. The results show that Monet achieves minimally the performance published for the fastest OODBs on the market place today.

CR Subject Classification (1991): Database systems (H.2.4) *parallel systems, query processing*, Physical design (H.2.2) *access methods*, Performance of systems (C.4)*design studies, measurement and modeling techniques, performance attributes*

Keywords & Phrases: Database techniques, performance measurements, benchmarking

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 [6].

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 [4] indicate room for further improvement and a need for more effective implementation techniques.

This article describes how we tackled the OO7 functionality using Monet [3], a novel main-memory oriented database server under development since 1992. Monet is a type-

and function-extensible database system, intended as a backend for multiple data model paradigms and application domains. Unlike most relational and OO systems, it is based on a fully-decomposed storage model [5] and exploits the facilities offered by modern operating systems.

We show how this system can be used to realize an OODB runtime system, that supports ODMG persistent programming. In particular, we study the impact of efficient database traversal to support the applications foreseen. We also analyze how query optimization and parallelization can seamlessly be integrated.

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 multiprocessor, 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 can 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 increase every year with 40% [14]. 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 microkernels, 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.

Monet is used on a daily basis for Data Mining [11] on real-life databases and a high performing GIS application [3] has been built on top of it. A basic SQL front-end exists, while work is under way developing a complete ODMG-compliant compiler and runtime system.

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.

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

- *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) [5] facilitates object evolution, and saves IO on queries that do not use all the relation's attributes, while the extra cost for re-assembling of complex objects before they are given to an application is neglectable in a main-memory setting.

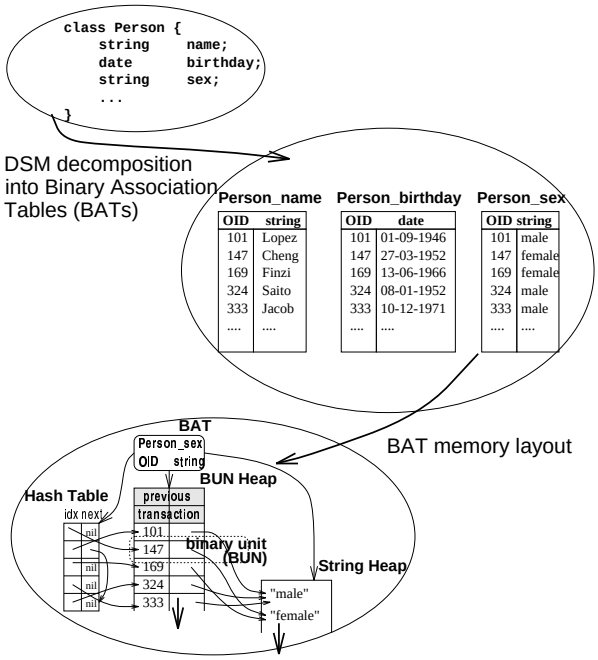

Figure 1: Monet's decomposed storage scheme

- use lean *bulk operations.* Studies like [14, 15] show that cache profiling can speed up a program by a factor of two. Deeply nested function calls cause CPUs to run out of register spaces, leading to similar performance penalties.

  Tuple-oriented database algorithms, as proposed in Volcano, perform a sequence of operations for every tuple in a set-operation, easily causing register overflow and cache misses. In contrast, bulk operations iterate over a set of contiguous tuples in one go, to perform one basic operation on all, typically without additional function calls in the inner loop. This allows for more efficient use of registers and cache memory.

- employ *inter-operation parallelism.* Monet exploits shared-store and all-cache architectures. Unlike mainstream parallel database servers, like PRISMA [1] and Volcano [9], 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 [17, 18]. A prototype runs on IBM/SP1.

- allow users to *customize* the database server. Monet provides extensibility much like in Gral [10], 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 Monet kernel has a shared-memory parallel architecture). The – interpretive – MIL interface is especially apt as target language for high-level language interpreters (SQL or OQL), allowing for modular algebra translation [10], in which parallel task generation is easy. Algorithms that translate relational calculus queries to BAT algebras can be found in [13, 17]

We show in an example what the BAT algebra looks like. Consider the following SQL query on relations `company [name,telephone]` and `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);
}
```

The variables created in the query cease to exist with the end of the sequential block ({}) in which they were created. The three last joins are placed in a parallel block ([]).

In all, the original double-select, single-join, three-wide projection SQL query is transformed in a sequence of 8 BAT algebra commands.

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. We describe 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)	$\{a \mid aT \in AB\}$

MIL has a sister language called MEL (Monet Extension Language), which allows to extend MIL with new datatypes, primitives and search accelerators. MEL is a specification-language only; implementations have to be given in C/C++ compliant object code.

The algebraic MIL interface has been wrapped in an IDL specification (see Figure 3), which also allows flexible interoperability using the CORBA mechanism, and encapsulates operations executed remotely or locally.

3. ODMG programming with 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, the MAGNUM project at the University of Amsterdam and CWI aims to develop an ODMG compliant database system on top of Monet, that will (amongst others) be used as a tool to build a high-end GIS application. While the MAGNUM project has been underway since mid-1994, we will touch here – in an OO7 setting – on some of its design issues.

The ODMG system consists of two parts: an ODL parser and an ODMG runtime library to be bound with the C++ application [7].

3.1 ODL Parser

The ODL parser accepts ODL data definitions, and inserts those in the OODB data dictionary and generates C++ header files with the corresponding class definitions. Monet uses Binary Associations (BATs) as its primary and only storage entity (see Figure 1). An OO database scheme is therefore translated into a set of BATs using a standard translation technique:

1. Create a BAT for every (inherited) class attribute using a binary association between an object identifier (oid) and the attribute type. Its name is the concatenation of the class- and attribute- name. For example, the attribute x of the class *AtomicPart* is represented as a BAT called *AtomicPart_x(oid, integer)*. Notice that *AtomicPart_type(oid,str)* will be created as well.

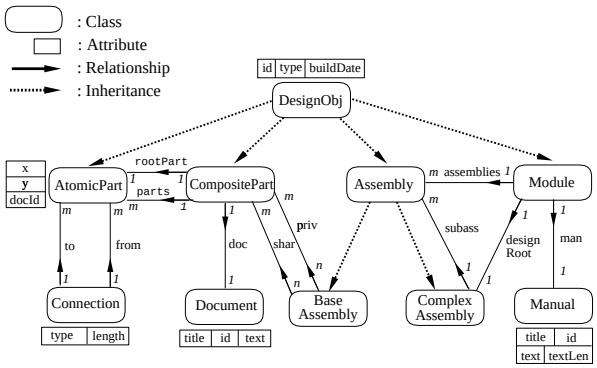

Figure 2: OODB schema: the OO7 database

2. Create a BAT for every relation in the scheme using a binary association between the object identifiers of the classes involved. For example, the BAT *parts(oid,oid)* is created for the relation 'parts', which links the *CompositeParts* to the *AtomicParts*. BATs can represent different population types. No distinction is required for 1-1, 1-n, n-1, and n-m relationships. For convenience reasons, the direction of the BAT, i.e. which side is the head and which the tail, is shown in Figure 1 by the direction of the relationship arrow.

3. Create a BAT for every class in the scheme. The class extension is described in a BAT with and object identifier (oid) as head, and the empty type (any) as tail. For example, the BAT *AtomicPart(oid,any)* describes the extension of the AtomicPart class[2].

In the case of the OO7 datamodel (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,

[2] Actually, a unary table would suffice.

contains a functional complete copy of the server code to manage *transcient* databases. This way, it can choose at runtime to cache tables and to execute operations locally or remotely.

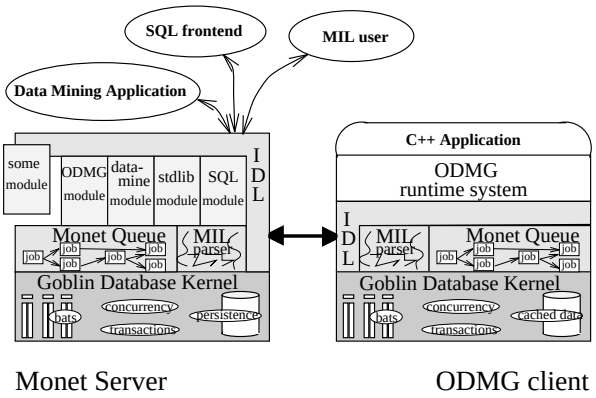

Figure 3: The Monet Server and its Clients

The challenge for an ODMG runtime library is to make efficient use of the underlying OODB characteristics. In case of Monet, the ODMG-client datamodel is different from the server's model, which puts the runtime library at the non-trivial task to translate between them. From another perspective, Monet's facilities provide opportunities for optimizations. It is relatively easy to construct an optimizing/parallelizing translator from OQL calculus to BAT algebra. As to pointer traversals, Monet's ODMG runtime uses DSM to perform *lazy instantiation* of objects: attributes are only allocated/loaded when they are used. The next section discusses another challenge: to make Monet's parallelism constructs and efficient bulk operations available for OO traversals.

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' [8]. 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 clear 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

The class traversal primitives envisioned require a few introductory definitions. We assume a class-attribute hierarchy $\mathcal{H}$, consisting of an inheritance hierarchy of classes, on which the relation attributes form a graph, and the functions:

$classes(\mathcal{H}) = \{C \mid C \text{ is a class in } \mathcal{H}\}$
$subclasses(P, \mathcal{H}) =$
$\quad \{C \mid P, C \in classes(\mathcal{H}) \wedge C \text{ is a subclass of } P\}$
$attributes(C, \mathcal{H}) =$
$\quad \{C.a \mid C \in classes(\mathcal{H}) \wedge \text{ a is an attribute of } C\}$
$class(C.a, \mathcal{H})$ and $class(obj, \mathcal{H}) =$
$\quad$ the baseclass of an attribute resp. object.

The class hierarchy can have objects as *instantiations*, which are captured by the extent $\mathcal{E}$ of $\mathcal{H}$. Class extents are defined as:

$$\mathcal{E}(C) = \{obj | obj \in \mathcal{E} \wedge class(obj, \mathcal{H}) \in subclasses(C, \mathcal{H})\}$$

The paths through the class-attribute hierarchy are inductively defined as $C_1 \overline{P} C_2$ with P a *path expression*. For simplicity, we assume all attributes to be relation attributes.

$$C_1, C_2 \in classes(\mathcal{H}) \wedge class(C_1.a, \mathcal{H}) \in subclasses(C_2, \mathcal{H}) \wedge$$
$$C_1.a \in attributes(C_1, \mathcal{H}) \quad \Leftrightarrow \quad C_1 \overline{C_1.a} C_2 \tag{4.1}$$
$$C_1 \overline{P_1} C_2 \wedge C_2 \overline{P_2} C_3 \quad \Leftrightarrow \quad C_1 \overline{P_1 - P_2} C_2 \tag{4.2}$$
$$C_1 \overline{P_1} C_2 \wedge ... \wedge C_1 \overline{P_i} C_2 \quad \Leftrightarrow \quad C_1 \overline{[P_1, \cdots, P_i]} C_2 \tag{4.3}$$

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

$$o_1 \overline{C_1.a} o_n \quad \Leftrightarrow \quad o_1.a = o_n \tag{4.4}$$
$$o_1 \overline{P_1 - \cdots - P_{n-1}} o_n \quad \Leftrightarrow \quad \forall i, 1 \leq i < n : o_i \overline{P_i} o_{i+1} \tag{4.5}$$
$$o_1 \overline{[P_1, \cdots, P_{n-1}]} o_n \quad \Leftrightarrow \quad \exists i, 1 \leq i < n : o_1 \overline{P_i} o_n \tag{4.6}$$

A concrete instantiation $o_1.a_1 - .. - o_n$ of a path $\overline{P}$ between to objects o_1 and o_n 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 the path $\overline{P}$, 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 to all distinct links $o_1.a_1 - .. - o_n$ where all o_n are objects reachable by path $\overline{P}$, from o_1 (where $C_1 \overline{P} C_i, o_1 \in \mathcal{E}(C_1), \forall o_i : o_i \in \mathcal{E}(C_i)$).

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 $\overline{P}$, 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;
  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. 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.

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 $A_{\overline{A.b-B.a}}A$ (see Figure 5) and uses this set as input to traverse the two-branched path $A_{\overline{[A.c-C.e,A.d-D.e]}}E$ 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).

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()`

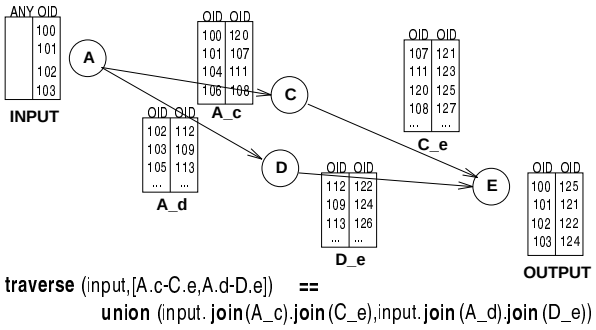

Figure 4: Executing a `traverse()` operator

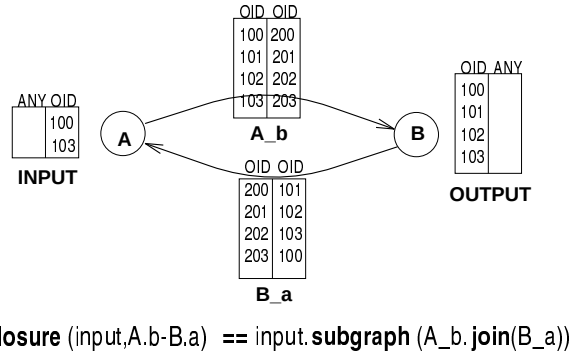

Figure 5: Executing a `closure()` operator

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* [16], 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 re-using 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 [16].

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.

Even without this, our approach to traversals still provides two important benefits. First, it allows Monet to use bulk operations for execution. Bulk operations work set-at-a-time and are more efficient than doing many tuple-operations. The second benefit comes from Monet's decomposed storage model. 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.

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. Invisible to the programmer, the class library translates the nested path operators to a sequence of MIL statements, which are executed at the Monet server.

```
// Path Class
template <class A, class B>
   class Path {
      Path(char *attr);           // outgoing attribute
      ~Path();
      int src_domain(Set<A>);     // defaults to extent
      int dst_domain(Set<B>);     // defaults to extent
   }

// Overloaded && for concatenating paths
template<class A, class B, class C>
   Path<A,C> Path::operator&&
      (const Path<A,B>& left,const Path<B,C>& right);

// Overloaded || for multiple paths.
template<class A, class B>
   Path<A,B> Path::operator||
      (const Path<A,B>& left,const Path<A,B>& right);

// PathOperator abstract class.
template <class A, class B>
  class PathOperator {
     Bag<B> collect(Ref<A> root);
     Bag<B> collect(Collection<A> root);
     int visit(Ref<A> root);
     int visit(Collection<A> root);
  }
```

```
// Traverse Operator
template <class A, class B>
   class Traverse : public PathOperator<A,B> {
       Traverse(Path<A,B> path); // no function
       Traverse(Path<A,B> path, void (fcn*)(Ref<A>,Ref<B>);
       ~Traverse();
   }

// Closure Operator
template <class A>
   class Closure : public PathOperator<A,A> {
      Closure(Path<A> path); // no functions
      Closure(Path<A> path, void (pre*)(Ref<A>,Ref<A>),
                            void (post*)(Ref<A>,Ref<A>));
      ~Closure();
   }

// Nest Operator
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, except for the attribute names used for specifying paths [3].

The library user should first build a path expression, using instances of the `Path` class. Using the *&&* and —— operators, complex paths 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.

5. OO7 on Monet

In this section we discuss how the OO7 benchmark can be expressed using the above defined traversal template classes. We then show how these traversals are translated internally to Monet's MIL primitives. Finally we show the performance results for the full OO7 benchmark, and give an performance evaluation.

The platform on which the experiments were carried out was a Sun SPARCstation 20/50Mhz workstation 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. Our configuration is in the order of 3 times faster than the Sun (SS/IPX) workstations on which E/Exodus, Objectivity, Ontos and Versant figures were obtained [4].

For the experimentation process we used the Software Testpilot [12]. The Software Testpilot is a performance assessment support system responsible for selecting values from the the wide range of OO7 parameters, (eg. number of connections, database size, type of query), construction and execution of the experiments, performance data retrieval, experiment history

[3] To make a check on this possible, the C++ syntax would have to be extended in some way.

and production of factsheets and reports. The Software Testpilot greatly simplified and speed up the experimentation process.

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 (refer T8 en T9). The memory mapping algorithm achieves similar performance as the swapping algorithm. This is shown in the traversal T2, in which the traversal has been executed for both memory mapped and non memory mapped BATs.

5.1 Traversal T1: Raw traversal speed

The first traversal of the OO7 benchmark is designed with a high degree of data locality in mind, to ensure a high number of cache hits. However, for a main memory database system like Monet, the data locality property is not an issue for a 'hot' database, because all data is kept in memory. However, the performance of this traversal is still interesting to compare its performance to the disk based OODB.

The template class C++ implementation of the T1 looks like this:

```
 1 void oo7_t1(char *dbname) {
 2   Set<BaseAssembly> baseassbly, assbly;
 3   Ref<Assembly>     root;
 4   Database          database;
 5
 6   database->open(dbname);
 7   lookup(&root, "root");
 8   lookup(&baseassbly, "BaseAssembly.extent");
 9
10   Path <ComplexAssembly, Assembly> *p1 = Path("subAss");
11   Closure <Assembly> *o1 = Closure(p1);
12   assbly = p0->collect(root)->intersection(baseassbly);
13
14   Path <CompositePart,AtomicPart> *p2 = Path("rootPart");
15   Path <BaseAssembly,CompositePart> *p3 = Path("priv");
17   Traverse <BaseAssembly,AtomicPart>*o2 = Traverse(p2 && p3)
18
19   Path <Connection,AtomicPart> *p4 = Path("to");
20   Path <AtomicPart,Connection> *p5 = Path("to");
21   Closure <AtomicPart> *o3 = Closure(p4 && p5);
22
23   Nested<BaseAssembly, AtomicPart> *o4 = Nested(o2, o3);
24   printf(o4->visit(assbly));
25   database->close();
26 }
```

The ODMG runtime library translates the path operators in this code to MIL statements. Since the ODMG runtime is still under development, we translated the traversals in the OO7 by hand, by simply following the algorithms: 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 *CompexAssembly.subAss*, already represented in Monet by the binary association `CompexAssembly_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 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.

The (absolute) hot times for traversal are shown in Figure 6. Figure 7 shows that the cold times are maximal a factor 1.7 slower than the hot times, and for longer execution times not more than 1.4. This shows that the impact of BAT loading in main memory is not a predominant cost factor.

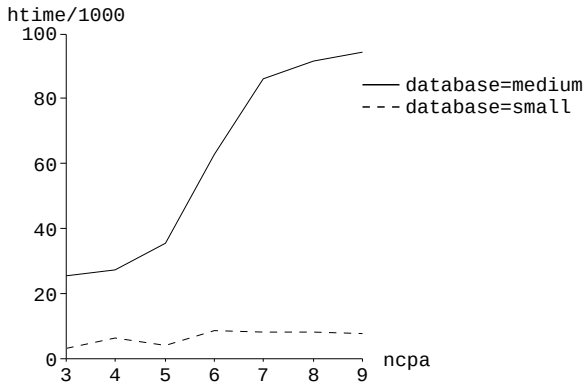

Figure 6: Traversal T1: Elapsed hot time

We observe that for small database sizes Monet's performance is similar to the fastest competitor when corrected with a factor 1/3 for difference in hardware configuration, except for the Versant figures, which show a slightly faster response for a (small) hot database. For medium database sizes, Monet is clearly the winner.

Figure 8 shows the main memory size after each T1 traversal. Observe that for the small database the preallocated memory size is sufficient, and that the medium database grows to 105 MB, which is more than the available physical memory.

5.2 Traversals T2 - T9

The traversal T2(a,b,c) focuses on the update performance of the DBMS. The three types of updates patterns in this traversal are: a) update one atomic part per composite part, b) update every atomic part as it is encountered, c) update each atomic part in a composite part four times. The update operation is to swap the x and y attributes of each atomic part as it is encountered in the same way as traversal T1. Our traversal classes permit implementing this by allowing a function parameter to be passed to the path operators, that is executed on all objects as they are visited.

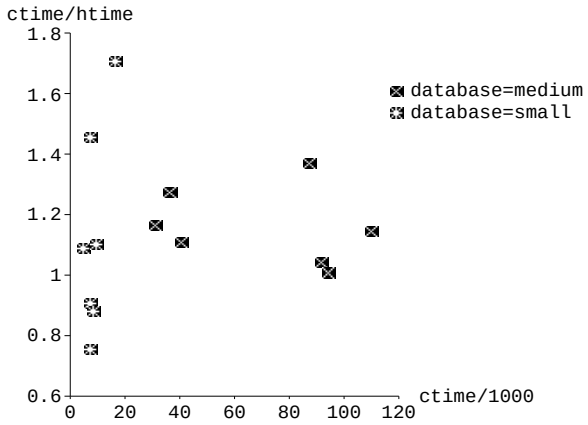

Figure 7: Traversal T1: Cold time overhead

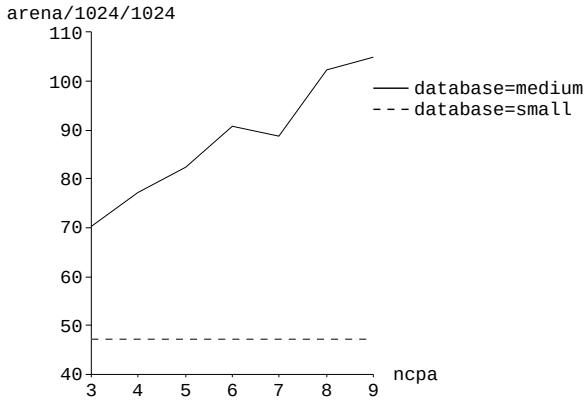

Figure 8: Traversal T1: Main memory size

Figure 10 shows the cold times for T2(a,b,c) for a small database and Figure 9 shows the cold time for T2(a,b,c) for a medium database. For the medium database size, the Monet figures are (hardware corrected) for T2(b,c) about 1.5 times better then the fastest of the competitors and for T2a an order of magnitude faster. Here the benefit of a decomposed storage model is clearly demonstrated. Not all atomic parts need to be retrieved and written back, but only the BATs representing the x and y coordinates of the root (atomic) parts.

For the small database the performance is about the same to slightly worse than the fastest competitor (with the exception of Versant, which performs much worse on these update traversals).

Figure 11 shows that the main memory consumption of the medium database after the execution of the traversals t2b and t2c, for memory mapped and non-memory mapped BATs are nearly the same, whether they fit in physical memory or not.

Moreover, Figure 12 shows similar response times for memory mapped and non-memory

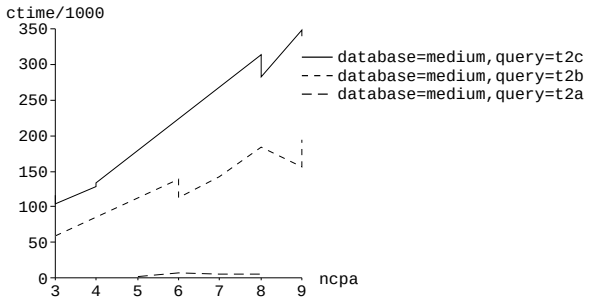

Figure 9: Traversal T2: Cold time for medium db

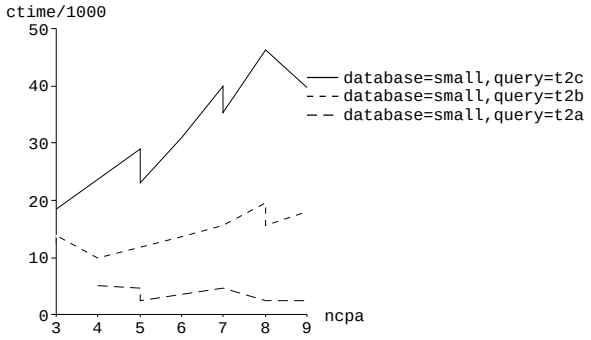

Figure 10: Traversal T2: Cold time for small db

mapped BATs. From these observations we conclude that when complete BATs are required, memory mapped BATs behave similar to BATs in swap space.

Traversal T3 repeats traversal T2, except that now the update is on the date field, which is indexed. The specific update is to increment the date if it is odd, and decrement the date if it is even.

As shown in Figure 13, the performance of the update operation using a batmap of a user defined operation is similar to T2. The competitors that make use of the indexed property of the data field benefit with response times up to 2 times faster than Monet *for small databases sizes.* However, Monet maintains automatic indexing and is therefore comparable to the Objectivity performance figures, showing similar performance figures (20 - 100 s, hardware corrected). For medium database sizes, only T3a numbers can be compared, because data on T3(b,c) have not published. The figures for T3a are an order of magnitude faster, due to the decomposed storage model of the the Monet kernel.

Traversal T6 measures sparse traversal speed. and is nearly the same is T1, except that the connection graph is not traversed. It is therefore much faster then T1 and independent of the number of connections parameter. For the medium database, the cold time is around 4s, and hot time around 1s. For the small database, the cold time is around 1.5s, and the hot time around 0.9s.

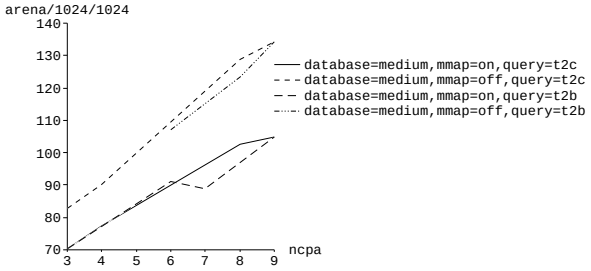

Figure 11: Traversal T2: Main memory consumption

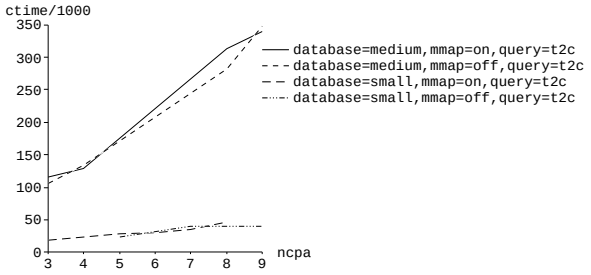

Figure 12: Traversal T2: Impact of memory mapping

Traversal T8 and T9 are operations on a large text block: *Manual_text*. 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*).

The performance data shows 0.5s faster response in the memory mapped version of T9 compared to the ordinary version 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 show 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:

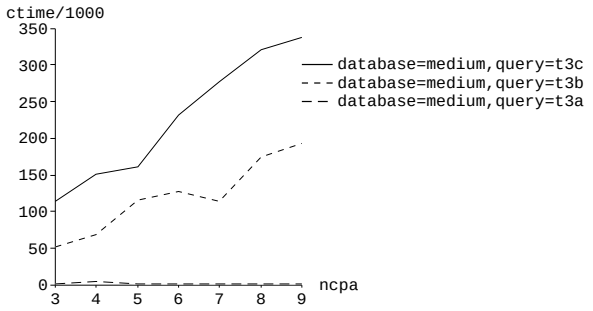

Figure 13: Traversal T3: elapse time for medium db

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

Figure 14: T8 and T9 on 1 MB Manual Text

Query	Cold time	Hot time	Competitor
Q2 (1%)	0.62s	0.61s	> 6s
Q3 (10%)	0.67s	0.64s	> 10s
Q4	0.85s	0.64s	> 0.5s
Q5	0.84s	0.82s	> 5s
Q7A (100%)	1.94	1.78s	> 10s
Q7B (100%)	2.5	2.2s	> 10s

The overall results of Monet on the OO7 queries, and traversal ,using our path operator strategy, compare favourably with previously published results [4].

6. Conclusion

In this article we discussed how OO applications can be supported with Monet, a novel extensible database system with an unusual underlying architecture. We used a derivate of the well-known concept of path-expressions to specify OODB class-attribute traversals on a high level of abstraction. Monet's virtual memory techniques, decomposed storage scheme, and bulk operations provide many opportunities to optimize and parallelize OODB traversals. The complete OO7 benchmark has been implemented on Monet to show the effectiveness of our approach, as demonstrated by the excellent results.

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

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

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

6. 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.

7. R.G.G. Catell et al. *The Object Database Standard.* Morgan Kaufman, 1993.

8. 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.

9. 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.

10. 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.

11. M. Holsheimer, M. L. Kersten, and A. Siebes. Data Surveyor: searching for nuggets in parallel. In *Knowledge Discovery in Databases.* MIT Press, Cambridge, MA, USA, 1995.

12. M.L. Kersten and F. Kwakkel. Design and implementation of a dbms performance assessment tool. In *Proceedings of the 4th intern. DEXA conference*, pages 265 – 276, Prague, Czech Republic, 1993.

13. 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.

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

15. A. Shatdahl, C. Kant, and J.F. Naughton. Cache conscious algorithms for relational query processing. In *Proceedings of the 20th VLDB Conference, Santiago, Chile.*, pages 510–521, September 1994.

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

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

18. 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.