Flattening an Object Algebra to Provide Performance

Peter Boncz ⋆ Annita N. Wilschut ◦ Martin L. Kersten⋆

◦ University of Twente
P.O.Box 217, 7500 AE Enschede,
the Netherlands
annita@cs.utwente.nl

⋆ University of Amsterdam
Kruislaan 403, 1098 SJ Amsterdam,
the Netherlands
{boncz,mk}@wins.uva.nl

Abstract

Algebraic transformation and optimization techniques have been the method of choice in relational query execution, but applying them in OODBMS is difficult due to the complexity of object-oriented query languages. This paper demonstrates that the problem can be simplified by mapping an OO data model to the binary ralational model implemented by Monet, a state-of-the-art database kernel. We present a generic mapping scheme to flatten data models and study the case of a straightforward object-oriented model. We show how flattening enabled us to implement a query algebra, using only a very limited set of simple operations. The required primitives and query execution strategies are discussed, and their performance is evaluated on the 1GB TPC-D benchmark, showing that our divide-and-conquer approach yields excellent results.

1 Introduction

During the last decade, relational database technology has grown towards industrial maturity, and the attention of the research community has shifted towards efficient support for objects in database systems [CaD96]. New application domains, like GIS, multimedia, etc, use the rich modeling facilities offered by object database systems to model application specific data.

A relational DBMS typically first translates SQL queries into relational algebra, which is subsequently optimized into an efficient physical algebra program. Algebraic optimization has proven a powerful tool in this process.

Currently, the research community tries to reuse this idea for the implementation of calculus-based object query languages [ClD92]. A lot of research is done in the complex area of translating such a language into an algebra [FeM95,GKG97,ShF94,SBB96], and the area of their optimization. A number of algebras on the object data model have been proposed, e.g. AQUA [LMS93] and KOLA [ChZ96]. The implementation of such algebras is difficult due to the combination of the very large number of operations and the complex storage model. To our knowledge, no efficient implementations of such algebras have been reported.

The contribution of this paper is twofold: first, we present the implementation of an object algebra on top of the Monet database kernel, and second, we evaluate the performance of this implementation.

Flattening an object algebra . . .

This paper shows how a high-level object data model and algebra can be mapped to the binary relational system Monet, a state-of-the-art, high-performance database kernel, used as backend for query execution. The concept of mapping the object data model to a different physical data model brings back the benefits of data independence, often hard sought for in OODBMS.

Section 3.3 discusses a formalism to describe mappings to the binary model of Monet. This process is illustrated with MOA, the Magnum[1] Object Algebra. MOA is designed as an intermediate algebraic language between a calculus-style object query language and the database execution language. MOA is a standard query algebra on a standard object datamodel, derived from well-known concepts elaborated in other algebras [ChZ96,LMS93,SBB96].

. . . **to provide performance**

This paper presents a system that can execute complex OO queries with high performance. This system translates MOA queries into Monet programs. Monet is used as a backend for execution. Demonstrating this performance posed a problem, since no complex *and* voluminous OO query benchmarks exist. The OO7 benchmark measures *navigational* performance in OO systems, but the query part of the benchmark is trivial. The BUCKY object-relational benchmark [CDN97] is not really complex in terms of joins and aggregations. The few complex OO benchmarks that we know of [ERE95] just specify queries, no database population.

Therefore we chose to use the TPC-D benchmark [TPC95]. The TPC-D queries consist of complex combinations of selections, joins, grouping, and aggregations against a relational decision support database. TPC-D can be run using different scaling factors with a database sizes from 1GB to 100GB. Numerous DBMS and OLAP system vendors have published results.

We slightly adapted TPC-D to fit an object-oriented context. Its database schema can be reformulated as a nested MOA schema (see Figure 1). The `groupby` SQL statement, maps to the OO concept of nesting and aggregation.

As we focus on query execution, we used only the query part of TPC-D, and did not – yet – look into updates. As the TPC-D queries were hand-translated from SQL into MOA, and vertical fragmentation is

[1] Two Dutch universities and the CWI cooperate in Magnum research project, which studies object database technology in the context of geographical applications.

a cornerstone of Monet, we do not comply with the benchmark implementation rules and do not intend to present our results as anything like an "official benchmark implementation". The schema, queries and 1 GB database just serve to illustrate the performance of our system. We compare our performance figures with the IBM/DB2 implementation to provide the reader with context.

Scope and outline of this paper

Object-oriented databases typically bring together *i) structural object-orientation* which allows defining structured object classes, *ii) behavioral object-orientation* which models the behavior of objects in a set of operations, *iii) inheritance* which relates object types in a subtype/supertype graph, *iv) a high-level object query language* like OQL, and *v) a persistent programming environment* like a language binding for Java or C++. Object-relational systems with *row-types* will be able to use more and more of these features starting from a relational standpoint, and add to this list the *vi) base type extensibility*.

The end-goals of the Magnum project envision a system with all these features. The work described in this paper is limited to points *i), iv)* and *vi)*. We will integrate the work concerning *ii), iii)* and *v)* described in [BKK96] in a later stage of the project.

The organization of the remainder of the paper is as follows. Section 2 describes the features of the Monet database kernel. Section 3 discusses the MOA object data model, and its the mapping on the binary Monet data model. Considerable attention is paid to the formal foundation of this mapping. Section 4 describes the MOA algebra, Monet's execution algebra and the translation of the one into the other. Section 5 outlines implementation techniques for the execution algebra in Monet, and Section 9 analyzes our TPC-D experiments and results. Finally, Section 7 summarizes and concludes the paper.

2 Monet

Monet is an extensible parallel database kernel that has been developed at the UvA and CWI since 1993. Monet has already achieved considerable successes in Data Mining [HKM95], for supporting GIS data [BQK96] and OO traversals [BKK96].

The design of Monet is based on trends in hardware technology: main memories of hundreds of megabytes are now affordable, and custom CPUs can perform over 200 MIPS. Since magnetic storage gets bigger but not faster, this means that IO is increasingly becoming a barrier in processing. Also, in the near future, standard PC hardware will come with multiple processors, so shared memory parallelism will become ever present. On the software side, we see 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 research prototypes are Mach, Chorus and Amoeba, but also commercial systems like Silicon Graphics' Irix and Sun's Solaris increasingly provide hooks for better memory and process management.

The incorporation of new datatypes like GIS data or multimedia types image, audio and video, has led to a steep increase in data volumes in databases. This causes tuples to grow wide, while a decreasing percentage of IO is really useful in queries that mainly access the small standard data types. Instead of designing a system around IO-oriented processing where hardware trends actually work against this, we concentrate on finding a data storage method that decreases the role of IO.

Bearing this in mind, the following ideas are applied in the design of Monet:

Main memory query execution All Monet's primitive database operations work on the assumption that the database hot-set fits in main-memory. It has no page-based buffer manager: Monet's algebraic operations always have direct access to the table data in main memory.

If the database gets larger, however, Monet allows making a gradual transition to IO dominated database processing. This is achieved by the transparent use of memory mapped files. Monet avoids introducing code to 'improve' or 'replace' the operating system facilities for memory/buffer management. Instead, it lets you give advice to the lower level OS-primitives on the buffer management strategy and lets the MMU do the job in hardware. Since these features themselves are controlled via the Monet Interface Language, "buffer management" itself becomes a part of query optimization, which has the additional advantage that decisions can be based on as complete knowledge as possible.

Decomposed storage model Monet implements a binary relational data model, in which all data is stored in Binary Association Tables (BATs, see Figure 2). So, structured data is decomposed over narrow tables [CoK85]. This fragmentation helps reduce chunk sizes to fit memory and saves a lot of IO. Monet exclusive accesses only those attributes that are actually used in a query.

Extensibility The Monet system is run-time extensible in various ways: *algebra commands and operators* can be added. *Base types* can be added via an ADT extension mechanism similar to Postgres [SRH90]. To augment the collection of standard types, temporal data extensions and a complete set of common GIS types have been written [BQK96]. Finally, *search accelerators* can be added to Monet. New base types cannot always be indexed efficiently with standard index structures, therefore the Monet extension mechanism allows adding new index structures, like the R-tree.

Parallelism Monet supports shared-memory parallelism via parallel iteration and parallel block execution and shared-nothing parallelism via a TCP/IP protocol. The primitives are relatively coarse-grained to preserve efficiency.

Dynamic Optimization A query engine that performs some dynamic optimization can hence simplify the work of a query optimizer, and is more robust to changes in the environment. Once the query optimizer has decided what primitives are necessary to execute a query, Monet decides at run-time which alternative implementation is most efficient at that moment.

3 Data Models

Object oriented type systems can be characterized by the concepts of *base types* and *structuring primitives.*

A class definition defines the *structure of objects*, that is mapped onto physical storage by the DBMS. Object oriented models typically have a rich hierarchy of structuring primitives, which besides the class primitive may contain concepts like tuple, set, list, and

```
class Part  <
   name        : string,
   manufacturer: string,
   brand       : string,
   type        : string,
   size        : integer,
   container   : string,
   retailPrice : float   >;

class Customer  <
   name        : string,
   address     : string,
   phone       : string,
   acctbal     : float,
   nation      : Nation,
   mktsegment  : string,
   orders      : {Order}  >;

class Nation  <
   name        : string,
   region      : Region   >;
```

```
class Supplier  <
   name        : string,
   address     : string,
   phone       : string,
   acctbal     : float,
   nation      : Nation,
   supplies    :
     {<  part     : Part,
         cost     : float,
         available: integer>} >;

class Order  <
   cust        : Customer,
   item        : {Item},
   status      : char,
   totalprice  : float,
   orderdate   : instant,
   orderpriority: string,
   clerk       : string,
   shippriority : string   >;
```

```
class Item  <
   part        : Part,
   supplier    : Supplier,
   order       : Order,
   quantity    : integer,
   returnflag  : char,
   linestatus  : char,
   extendedprice: float,
   discount    : float,
   tax         : float,
   shipdate    : instant,
   commitdate  : instant,
   receiptdate : instant,
   shipmode    : string,
   shipinstruct : string   >;

class Region  <
   name        : string,
   comment     : string   >;
```

Figure 1: MOA data model for the TPC-D database

array.

The internal structure of *base types* on the other hand cannot be accessed. It is only accessible via operations. These base types allow efficient implementation in a general-purpose programming language, and are often supported inside the database query engine by specialized search accelerators.

There is growing consensus among OO database researchers that the collection of base types and associated search accelerators should be extensible [CaD96]. This gives a database designer the choice between implementing an entity either as atomic type, or as a database structure. Examples of data entities that are typically modeled as base types are lines and polygons in spatial systems, and image and sound data for multimedia applications.

3.1 MOA logical data model

The MOA data model is similar to well-known structural object data models like [ClD92,SAB94]. At its basis, MOA accepts all atomic types of Monet (that is {`bool, short, integer, float, double, long, string`}) as base types. Since Monet can be extended with new atomic types, this automatically provides MOA with base type extensibility. The base types can be combined orthogonally using the structure primitives `SET`, `TUPLE` and `OBJECT`, which will be formalized in Section 3.3. A MOA database is formed by the collection of *class-extents*, which are sets for each type of object that contain all their instances. Figure 1 shows the MOA model for the TPC-D schema and illustrates the use of structures.

3.2 Monet physical binary model

Monet stores all data in **Binary Association Tables** (BATs). Figure 2 shows the design of the BAT structure. The left column of a BAT is referred to as *head*, the right column as *tail*. Due to the design of its datastructure, any BAT can always be viewed from two perspectives: its normal form `bat[X,Y]`, and the *mirror* `bat[Y,X]`, which has the head and tail columns swapped.

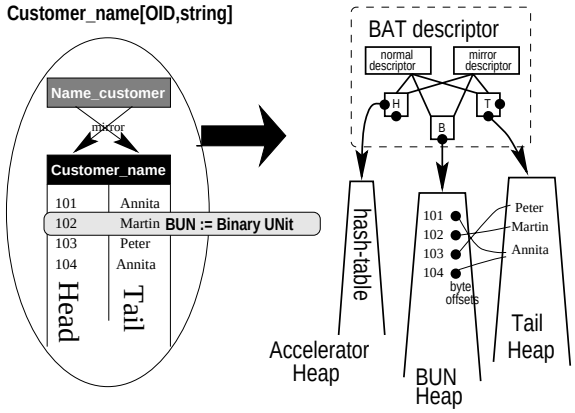

Figure 2: The Binary Association Table (BAT) and its memory layout

A BAT has at least 1 and at most 5 associated *heaps*. There is always a heap that contains the atomic value pairs, called Binary UNits (BUNs). This ensures dense, array-like storage of fixed-size datas. For atoms of variable size – such as `string` or `polygon` – both head and tail can have an extra heap (the BUNs then contain integer byte-indices into that heap). Finally, persistent search accelerators – for instance hash tables – may be stored in separate heaps, for both head and tail.

3.3 Flattening the object data model

Every implementation of an object-oriented data model has to map structuring primitives to some physical representation. Some implementations use a one-to-one mapping between the logical and the physical model. A well-formalized mapping provides data independence, enabling the DBMS to choose a physical representation different from the logical one, so that it may have extra optimization possibilities during query execution.

In the case of MOA, we use full vertical decomposition [CoK85] to store structured data in BATs. The combination of BATs storing values and a *structure*

function on those BATs forms the *representation of a structured value*.

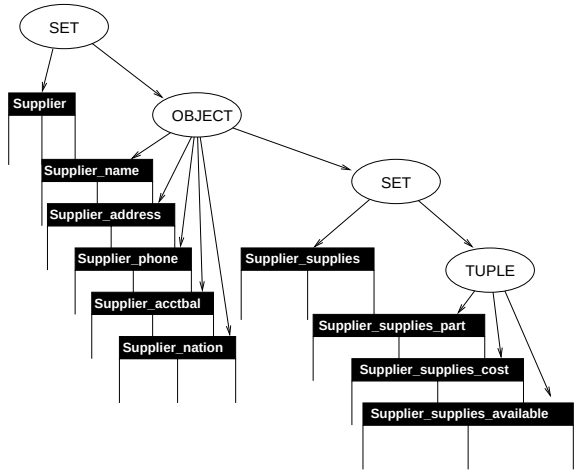

Figure 3: Mapping of the supplier table on BATs

An example

Figure 3 shows the decomposition of the TPC-D Supplier class into BATs. For example, BAT `Supplier_name` contains the values of the `name` attribute in the `Supplier` objects. All BATs that store attribute values contain an `oid` in the head and the corresponding attribute values in the tail. The `Supplier_name` BAT thus has signature `BAT[oid,string]`. The BAT `Supplier` contains all `oid`s of existing objects, and is called the *extent BAT*. The set-valued attribute `supplies` uses the BAT named `Supplier_supplies` to map supplier `oid`s to set-ids, and the attribute values in the `supplies` set are stored in BATs that can contain 0 or more BUNs for each set-id. The structure expression

```
SET(Supplier,
    OBJECT(Supplier_name, Supplier_address,
           Supplier_phone, Supplier_acctbal, Supplier_Nation,
           SET(Supplier_supplies,
               TUPLE(Supplier_supplies_part,
                     Supplier_supplies_cost,
                     Supplier_supplies_available))))
```

describes how the `Supplier` objects are created out the BATs they are decomposed over. Each structure in the type system is reflected by a structure function in the structure expression. To form a rigorous framework for the implementation of the algebra it is essential to formalize the semantics of the structure functions `SET, OBJECT`, and `TUPLE`.

The formal semantics of the mapping

To define the mapping of structures on BATs, we will formalize the type-system, the concept of a BAT and we will define containers of tuple values and of set values.

The type system is defined as follows:
basetypes: β is a type if β is an atomic Monet type.
tuple types: $\langle \tau_1, \cdots, \tau_n \rangle$ is a type, if τ_i are types.
set types: $\{\tau\}$ is a type if τ is a type.

$\mathcal{V}_\tau$ is used to denote the domain associated with type τ. $\mathcal{V}_\beta$ denotes the domain of a base type. Monet supports a base type `oid`, and $\mathcal{V}_{\texttt{oid}}$ the set of object identifiers.

A BAT$[\beta_1, \beta_2]$ is semantically equivalent to a subset of $\mathcal{V}_{\langle \beta_1, \beta_2 \rangle}$.
A head-unique BAT$[\beta_1, \beta_2]$ has unique values in the head column, so it is a subset of

$$\{\langle x_i, y_i \rangle \in \mathcal{V}_{\langle \beta_1, \beta_2 \rangle} | i \neq j \rightarrow x_i \neq x_j\}$$

An *identified value set* S is a set of pairs in which each value v_i is associated with an identifier id_i that is unique within the value set: S_τ is called an IVS over τ iff

$$S \in \mathcal{P}(\mathcal{V}_{\langle \texttt{oid}, \tau \rangle}) \wedge$$
$$\forall \langle i, v \rangle, \langle j, w \rangle \in S : \langle i, v \rangle \neq \langle j, w \rangle \Longrightarrow i \neq j$$

Identifiers can be, and actually are, reused in different value sets. In this way the concept of *synchronous value sets* is defined:
Two identified value sets $S1$ and $S2$ are **synchronous** if

$$\langle id_k, x \rangle \in S1 \Longleftrightarrow \langle id_k, y \rangle \in S2$$

So, each identifier-value pair in $S1$ has an identifier-value pair in $S2$ for which the identifiers correspond and vice versa.
We can now define the semantics of the structure functions recursively as follows.

A head-unique BAT $[\texttt{oid}, \beta]$ represents an identified value set S_β.

A head-unique BAT $[\texttt{oid}, \texttt{oid}]$ in which the tail-values refer to database objects of class X, represents an identified value set $\{\langle id_i, X_i \rangle X_i \in X \wedge oid_i = oid(X_i)\}$.

If $S_1, \cdots, S_n$ are mutually synchronous identified value sets, the structure function
TUPLE$(S_1, \cdots, S_n)$ defines a new value set:

$$\{\langle id_i, \langle v_{i1}, \cdots, v_{in} \rangle \rangle | \langle id_i, v_{ij} \rangle \in S_j\}$$

The `OBJECT` structure function is identical to the `TUPLE` structure function. The ids associated with the tuples generated are the object identifiers.

If A is a BAT $[\texttt{oid}, \texttt{oid}]$, and S is an identified value set then the structure function SET(A, S) defines the value set

$$\{\langle oid_i, \{v_j\} \rangle | \langle oid_i, id_i \rangle \in A \wedge \langle id_i, v_j \rangle \in S\}$$

A serves as an index into value set S.

If A is a BAT $[\texttt{oid}, \beta]$, then the structure function SET(A) defines the value set:

$$\{\langle oid_i, \{v_j\} \rangle | \langle oid_i, v_j \rangle \in A\}$$

This an optimization of the previous way of storing sets, for the case that the set element value is simple (i.e. a base type or an object reference).

Because the structure functions all have identified value sets as operands and result in identified value sets, they can be composed to generate complex structured data. There is a one-to-one relationship between structures in the data model and structure functions in

the physical-to-logical mapping. This implies that any data type expressible in MOA can be represented by a set of BATs and a composition of structure functions. In the remainder of this paper the symbol ($\mathcal{S}$) is used to denote some composition of structure functions.

4 Query Execution

In the Magnum project, we aim at supporting a declarative object query language like IQL [AbK92] and ODMG-OQL [Cat94]. The preparatory stage, the translation of a declarative object query language into an object algebra, has been studied extensively [SAB94,SBB96] and an implementation of the ideas developed is on its way.

This section describes the MOA query algebra, which is designed to be an intermediate language. In the context of this paper however, it is our source language and the Monet Interpreter Language (MIL) is our target language.

4.1 MOA Query Algebra

The MOA query algebra is a standard object algebra. It contains the operations select, project, join, semijoin, union, intersection, difference, subset, in, nest, unnest, and aggregates that operate on sets; it allows access to attributes of tuples and objects; it supports operations on the atomic types and allows for method invocations on objects. Example descriptions of similar algebras may be found in [ClD92,LMS93, SBB96].

```
project[<date : year, sum(project[revenue](%2)) : loss>](
   nest[date](
      project[<year(order.orderdate) : date,
               *(extendedprice,-(1.0, discount)) : revenue>](
         select[=(order.clerk, "Clerk#000000088"),
                =(returnflag, 'R')](Item))))
```

The MOA version of TPC-D query 13 displayed above provides a flavor of the algebra. This query analyzes the quality of work of a certain clerk. It combines two selections on `Item` – sold by a certain clerk, and having a return flag indicating that it was sent back with defects – computes a revenue lost per returned item, and then sums the losses over each year. The grouping of losses per year is done using nesting. The result of the query is projected into a set of <`year, loss`> tuples.

4.2 Monet Execution Algebra

The Monet Interface Language (MIL) consists of the BAT-algebra, which contains basic set operations, and a collection of control structures. BAT-algebra operations materialize their result and never change their operands.

The above primitives are sufficient to execute the majority of MOA constructs on Monet. Each BAT algebra primitive has a fixed semantics regarding what it expects in the columns of its parameters. If necessary you just use the `mirror` command to flip head and tail of a BAT; an operation free of cost. The `semijoin` operation is important, since it is heavily used for reassembling vertically partitioned fragments (in Section 5.2 we will elaborate an efficient implementation of this operation). Note that the equi-`join` projects out the join columns, in order to keep the operation closed in the binary model. The `unique` produces its result by removing the duplicates from its operand.

MIL command	informal semantics
AB.mirror	$\{ba \mid ab \in AB\}$
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.unique	$\{ab \mid ab \in AB\}$
AB.group	$\{ao_b \mid ab \in AB \wedge o_b = \text{unique\_oid}(b)\}$
AB.group(CD)	$\{ao_{bd} \mid ab \in AB \wedge cd \in CD \wedge a = c \wedge o_{bd} = \text{unique\_oid}(b,d)\}$
$[f]$(AB)	$\{af(b) \mid ab \in AB\}$
$[f]$(AB,$\cdots$,XY)	$\{af(b,\cdots,y) \mid ab \in AB, \cdots, xy \in XY \wedge a = \cdots = x\}$
$\{g\}$(AB)	$\{ag(S_a) \mid a \in A \wedge S_a = \{b \mid ab \in AB\}\}$

Figure 4: BAT primitives for executing OO queries

Operations on values (like arithmetic), and aggregate operations on BATs (like sum, avg, etc) are also part of MIL, but are omitted for brevity, just like the theta-join and some set-operations (difference, intersection, etc).

grouping The `group` operation introduces new `oid`s for uniquely occurring values in a BAT column. In this definition, the unique_oid($\cdots$) function returns a new `oid` for each unique (combination of) parameter(s). This operation is used to implement SQL `groupby` and MOA `nest`. For groupings on one attribute the unary version is used. For groupings on multiple attributes, this is followed up by binary `group` invocations till all attributes are processed. This is illustrated in Figure 5 by the grouping that occurs on the objects of interest according to `year` (the `group` operation assigns new `oid`s, that are used as key for all three result BATs of the query).

method invocation The **multiplex** constructor $[X]$ allows bulk application of any algebraic operation on all tail values of a BAT. Multiple BAT parameters can be given, in which case the algebraic operation is applied on all combinations of tail values over the natural join on head values. This operation is used to vectorize computation of expressions, and invocation of methods. As an example, in Figure 5 the expression `(1-price)*discount` is vectorized in successive `[*]` and `[-]` operations.

aggregation The **set-aggregate** constructor is used for bulk aggregation. It is defined for each aggregate function Y that maps a set to some value. The set-aggregate version $\{Y\}()$ groups over the head of the BAT and calculates for each formed set of tail values an aggregate result. With this construct, we can execute nested aggregates in one go, rather than having to do iterative calls to some function on nested collections.

4.3 MOA to MIL transformation

The idea behind the algebra implementation is to translate a query on the representation of the structured operands into a representation of the structured query result. Figure 6 illustrates this process: the query is a MOA expression on a structure expression on BATs, and its translation is a MIL program on the operand BATs that generates result BATs, which in turn are operands of another structure expression that represents the result.

Formally, this implementation is described as follows: Assume that we execute MOA-operation `moa` on the structured data value X. X is stored in BATs $X_1, \cdots, X_n$, and there is a structure function $\mathcal{S}_X$, such

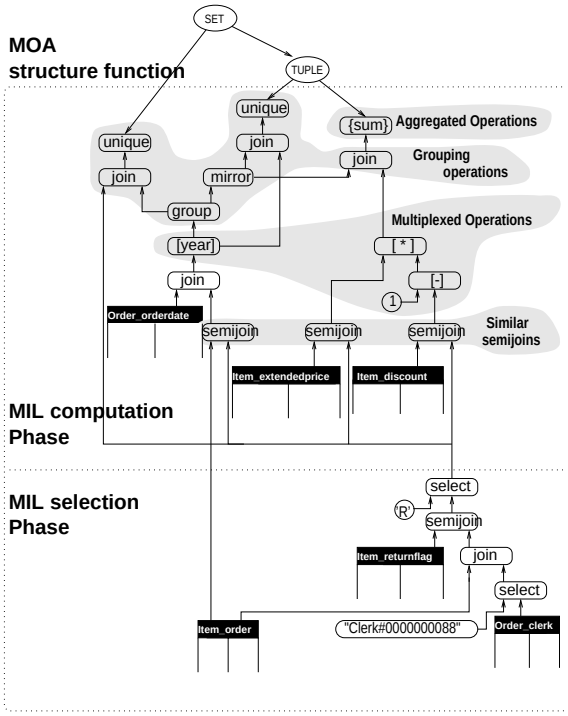

Figure 5: TPC-D query 13 as a MIL tree

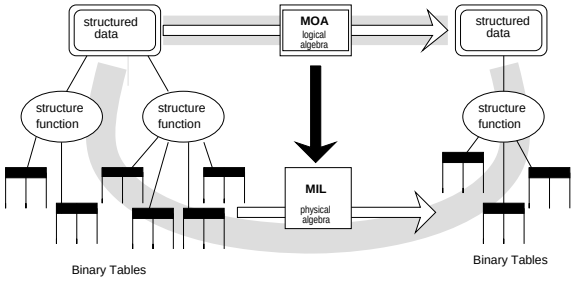

Figure 6: MOA query execution by translation to MIL

that $\mathcal{S}_X(X_1, \cdots, X_n) = X$. The implementation of operation `moa` consists of a MIL-program `mil`, that results in BATs $Y_1, \cdots Y_m$, and a structure function $\mathcal{S}_Y$, taking $Y_1, \cdots, Y_m$ as operands, so that

$$\mathcal{S}_Y(\texttt{mil}(X_1, \cdots, X_n)) = \texttt{moa}(X)$$

Because the operations in MIL and in MOA, and the structure functions have a formal semantics, it is possible to prove the correctness of implementation of MOA on MIL. A detailed discussion of this issue is beyond the scope of this paper. Informally, a correct implementation can be described as an implementation for which both gray paths in Figure 6 yield the same result.

For each operation in MOA, a transformation rule for the translation of the operation into a MIL program and structure function is generated. The MOA implementation consists of a straightforward term rewriter. Figure 5 shows the MIL translation of TPC-D query 13. We illustrate the simplicity of the transformation rules with a few examples:

4.3.1 Selection

The syntax for a selection in MOA is: `select`$[f()](X)$, in which X is an expression, that evaluates to the set $\{x\}$, and $f()$ is a boolean operation on the elements in X. The semantics of this selection expression is $\{x | x \in X \wedge f(x)\}$. The transformation rule for selections is:

select[f()](SET(A, X)) → SET(semijoin(A, $\mathcal{T}$ (f(X))), X).

Because the selection operation operates on sets, the translation of the operands of a syntactically correct MOA selection expression is always of the form `SET(A, X)`, with A as set index and X as identified value set. In this rule, $\mathcal{T}$`(f(x))` is the translation of the selection predicate $f()$ on the operands value-set X. The selection predicate has to be a boolean function on the value set; it is translated via its own transformation rule into a BAT containing the ids of the qualifying values in value set X. A new setindex is generated via a semijoin.

4.3.2 Operations on set-valued attributes

Structures in MOA may be nested, and therefore, set-valued attributes may occur. The Supplier class in the TPC-D benchmark is an example. Assume that we want to retrieve, for each supplier, the set of parts that are out of stock, so that `available` is equal to 0. In MOA this query is expressed as follows:

project[<%name,
select[%available = 0](%supplies)>](Supplier).

This query contains a selection on set-valued attribute supplies. The transformation rule for selection set-valued attributes is identical to the rule a selection on a single set. If operand X in Section 4.3.1 is interpreted as an identified set of set values, the transformation of this expression results in the correct identified set of reduced set values.

Here we see one of the beneficial effects of storing nested sets in a flattened model: instead of executing repeated selections for each nested set, we can do all work together in one selection on the flattened representation. Similar efficient translations are made for other nested set-operations like union, difference, and intersection.

5 Monet Implementation

This section describes some aspects of the Monet implementation that are heavily used in the TPC-D implementation.

5.1 Property Management

The Monet kernel generally contains multiple implementations for each algebraic operation. For instance, for the `semijoin` there is a `hashsemijoin` implementation, but also a `mergesemijoin`, that assumes the join columns of both BATs to be ordered. The most particular variant is the `syncsemijoin`, that using the knowledge that the join columns are exactly equal just returns a `copy` of its left operand BAT.

The philosophy of Monet is that the algebraic commands do an additional *dynamic optimization* step just before execution. Depending on the state of the system, and the state of the operands, a run-time choice between the available algorithms can be made. To this end, Monet keeps track of various properties of

permanent and intermediate BATs. We focus here on three BAT properties that are maintained by the kernel on each column. The following are examples of such properties:

`ordered(BAT)` is `true`, if the head column of stored in ascending order.

`key(BAT)` is `true`, if the head column of the BAT does not contain duplicate elements.

`synced(BAT1,BAT2)` is `true`, if the BUNs in both BATs correspond by position. The most common case for this is that the head columns of the two BATs are exactly identical.

Once set, these properties are actively guarded by the kernel. When updates occur, they are rechecked, and switched off if necessary. Each MIL command has a *propagation rule* for propagating the properties of its parameters onto its result. For example, a semijoin will propagate the `key` properties on both head and tail of its left operand onto the result, a `rangeselect` will propagate the `ordered` information on both head and tail to the result.

5.2 The Datavector Accelerator

OLAP queries as found in the TPC-D benchmark, typically consist of two phases:

- first, they select an interesting subset of objects; using some *selection-attributes*.
- then, in a second phase, computation of expressions and aggregations on other attributes of the selected objects takes place. Let us call these the *value-attributes*.

These trends can be observed also in TPC-D query 13 and are indicated in Figure 5.

When the database hotset outgrows main memory, algorithms using sorted tables like merge-join, merge-semijoin, and binary search selection tend to work best in Monet, because they have sequential access patterns and can better be supported by the OS virtual memory pager.

For doing the selection on the *selection-attributes*, one would prefer to have attribute BATs ordered on attribute value (tail column), in order to use binary search selection.

To do computation and aggregation on the *value-attributes* of the selected objects, one needs to do semijoins between the *value-attribute* BATs and the made selection. Observe that although multiple semijoins may be necessary, many of those will be very similar: they will semijoin the same selected `oid`s from the attribute BATs.

This leads to conflicting clustering-requirements: *selection-attributes* require sorting on tail, whereas *value-attributes* require sorting on `oid`. Also, attributes have different roles – value or selection – in different queries.

The solution explored here is to store all attributes ordered on tail; this favors the access from values to `oid`s (e.g. selections, and joins on attribute values). The path in the opposite direction, from `oid`s to values is then tackled by using a fully vectorized representation of the n-ary table into one vector of `oid`s and n vectors with attribute values, that are all stored in `oid` order. They can easily be represented in Monet using unary BATs. [2] Note that the MOA mapping of objects already gave us the unary vector of `oid`s, as the extent BAT (see Section 3.3). We use the discussed `synced` kernel property (see Section 5.1) to let Monet ensure us that the vectors correspond by position. The original BATs used in the MOA mapping, that are – as said – kept sorted on tail, then have a reference to their "value vector" by means of a new Monet *search accelerator* extension dubbed `datavector`.

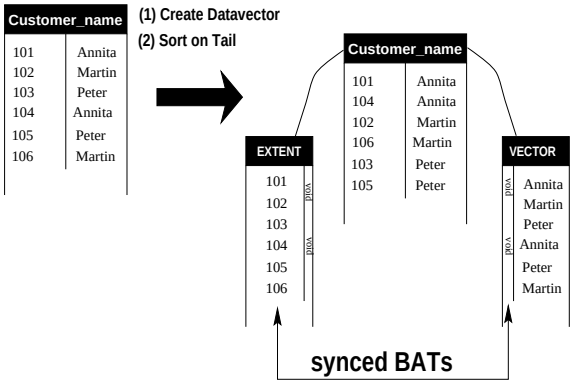

Figure 7: Datavector Creation through Project and Sort

Through all this we actually achieve a fully vectorized storage (represented by the extent BAT and all datavectors), supplemented by an inverted list index on all attributes (the "normal" BATs containing the `oid`,attribute combinations).

5.2.1 Datavector Semijoin

Just like the presence of a hash-table on an operand might lead the `join` to choose a `hashjoin` implementation, might the presence of a datavector influence the execution strategy of some operations. The most important operation in our context is `semijoin`, since it is instrumental in the phase of getting to the *value-attributes* of an OLAP query. We hence introduced a **datavector semijoin**; which is displayed below in pseudo code:

```
01 PROCEDURE datavector_semijoin(BAT[oid,any] A, BAT[oid,_] B)
02 BEGIN
03      oid EXTENT[size(A)] := extent(A);
04      any VECTOR[size(A)] := datavector(A);
05      int LOOKUP[] := positions(B);
06
07      IF NOT EXIST(LOOKUP) THEN
08         INT IDX := 0
09         LOOKUP := new INT[size(B)];
10         FORALL BUNS [X,_] IN BAT B DO
11              IF (POS := probedlookup(EXTENT,X)) THEN
12              LOOKUP[INC(IDX)] := POS;
13              FI
14         OD
15      FI
16      BAT RESULT[oid,anyNY];
17      FORALL Z in [1..size(LOOKUP)] DO
18         INT POS := LOOKUP[Z];
19         RESULT.insert(EXTENT[POS], VECTOR[POS]]);
20      DO
21      RETURN RESULT;
22 END
```

[2] Unary BATs are of course a contradiction. We mean BATs that have the zero-space type `void` in one column.

Associated with the left operand BAT are both the extent and the datavector (lines 3-4). For ease of reading both are displayed here as simple arrays. If it is the first time that B is used as right operand for a semijoin, then lookup has to be performed (line 7). All elements of B are looked up in the extent (line 10-11). The extent is always kept sorted, so this lookup can be implemented efficiently using *probe-based* binary search. For each hit, the array index is saved in the `LOOKUP` array (line 12). It is kept there for later use, so subsequent semijoins with B not re-do the lookup effort. The insertion phase walks through the `LOOKUP` array, and fetches the matching head and tail values from respectively `EXTENT` and `VECTOR` (lines 17-19).

5.2.2 IO Cost Model

On the one hand, Monet benefits from the full vertical fragmentation (less IO, narrow tables), on the other, it has to face the extra semijoins to recombine fragments. This section analyses the resulting performance trade-off via a small performance model, in which the costs the Monet approach are compared to non-decomposed relational approach. As doing some semijoins poses no performance problem in main memory, we focus here on the IO bound situation. That is, we assume cold memory mapped BATs, such that every access to them will cause page faults.

We are interested for expected number E of B-byte disk pages to be retrieved (or: virtual memory page faults) for doing a selection with selectivity s, followed by a projection to p attributes in an n-ary table. This n-ary table has X rows which are $n*w$ bytes wide, where w is taken uniform as the byte width of one value:

$E_{rel}(s) = \lceil \frac{sX}{C_{inv}} \rceil + \lceil \frac{X}{C_{rel}} \rceil * (1 - (1-s)^{C_{rel}})$ is the expected number of disk blocks to fetch when using a relational strategy where the database table is stored without decomposition. The first component is the IO cost of discovering which tuples participate in the selection. This can most efficiently be done using an index; in this case we assume an inverted-list, implemented as an array of `[value,tuple-pointer]` records. The number of inverted-list tuples per page $C_{inv} = \lfloor \frac{B}{2w} \rfloor$. The second component models – unclustered – retrieval itself. It is a multiplication of the number of pages with the probability that at least one row in a page is selected. The number of rows per page $C_{rel} = \lfloor \frac{B}{(n+1)*w} \rfloor$.

$E_{dv}(s) = \lceil \frac{sX}{C_{bat}} \rceil + (p+1)(\lceil \frac{X}{C_{dv}} \rceil * (1 - (1-s)^{C_{dv}}))$ expresses the costs for the Monet approach. The first component represents doing the selection on a BAT. We have all data BATs sorted on tail, which is in fact like having an inverted list on each attribute. The second component of the formula represents doing p datavector semijoins to get the requested attribute values. The lookup into the extent performed during the first datavector semijoin counts as one semijoin more, hence the factor $p+1$. The number of BUNs per page of a BAT $C_{bat} = \lfloor \frac{B}{2w} \rfloor$, and the number of datavector values per page $C_{dv} = \lfloor \frac{B}{w} \rfloor$. .

Figure 8 displays the projected cost with parameters derived from the 1GB TPC-D `Item` table ($X = 6000000, n = 16, w = 4, B = 4096$). The fat line shows the model for the relational storage case. The thin lines show the Monet case for various values of p. It shows that Monet's datavector semijoin strategy is generally more efficient than the relational approach, apart from very low values of s, e.g. the crossover point for $n = 16, p = 3$ is at $s \simeq 0.004$.

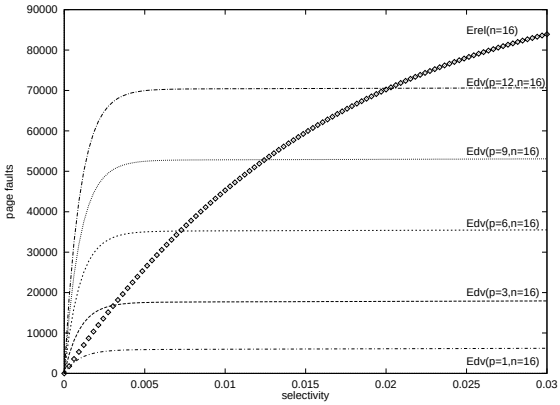

Figure 8: select-project IO cost according to selectivity for relational vs datavector approach

6 TPC-D Experiments

We used the DBGEN program to generate the 1GB database in ASCII files. We then loaded these into Monet our system, using its bulk load utility, which took 1:28 hour. This utility correctly sets the properties *key*, *ordered*, and *synced* for each generated BAT.

For each class, an `extent[oid,void]` was created by taking one attribute-BAT, and projecting out the tail column. Initially, all tables were sorted on `oid` (head column), so it was cheap to create datavectors on them: we just had to make a projection on tail column for each attribute BAT. Creating the extents and the datavectors took about half an hour.

In order to efficiently execute selections and joins on attribute values, we then reordered all tables on tail values. This took an additional hour. In total, the TPC-D database occupies 1.6GB of diskspace (of which 300MB in data vectors, and 1.3 GB as base data).

All MOA versions of the TPC-D queries were fed through the translator – which takes no significant time – and executed in a sequence on the Monet backend. Figure 9 shows the absolute performance results in elapsed seconds. To provide more insight, we also include stats on the number of page faults, the selectivity in the main table (`Item` has 6 million tuples), the total size of all intermediate results, and the maximum memory consumption during query execution.

6.1 Hardware

The hardware platform used for experimentation was a Sun SuperSparc 20/61 (performing at 98.1 SPECint) running Solaris 5.3. The machine had two internal 4GB Seagate ST15150W disks (9 ms access time, 6MB/s throughput), of which one was used as root file system and swap area, the other one for storing our TPC-D data. The only other known TPC-D numbers for the 1GB benchmark are the official IBM numbers, obtained on a PentiumPro 200 Mz PC

Qx	**DB2 (sec)**	**Monet (sec)**	total (MB)	**max (MB)**	Item **select%**	page faults	**comment**
1	668.4	1098.1	800	95	98%	73K	billing aggregates over 700MB table
2	6.7	14.1	10	4	n.a.	518	cheapest part supplier for a region
3	179.7	99.8	76	60	56%	368	find top-10 valuable orders
4	88.3	52.3	6	6	4%	16K	priority assessment, customer satisfaction
5	148.2	172.2	98	44	15%	13K	revenue per local supplier
6	95.2	48.7	45	32	15%	132	benefits if discounts abolished
7	261.3	109.5	124	29	30%	966	value of shipped goods between 2 nations
8	54.2	117.5	39	29	30%	4.1K	part market share change for a region
9	2321.6	77.1	70	7	3%	25K	line of parts profit for year and nation
10	221.7	83.1	22	8	4%	18K	top-20 customers with problematic parts
11	6.4	8.9	9	3	n.a.	97	significant stock per nation
12	128.8	214.4	76	25	15%	11K	cheap shipping affecting critical orders
13	24.1	37.9	0	0	0.1%	4.7k	loss due to returned orders of a clerk
14	64.7	35.2	8	4	1%	384	market change after a campaign date
15	55.6	41.1	33	22	4%	47	identify the top supplier
load	4740	10080					ascii import and accelerator creation
QppD	43.8	59.1					geometric mean-based query per hour rate

Figure 9: TPC-D Results In Elapsed Seconds

(performing at 351 SPECint) running Windows NT 3.51. This configuration had an ultrawide SCSI controller, with four Ultrastore XP disks (9 ms access time, 10MB/s throughput). Both platforms had a total of 128MB memory. Note that the IBM is about 3 times more powerful than our hardware.

6.2 Analysis

We see the best side of Monet in e.g. in queries 3, 4, 6, 7, 9, 10 and 14. The IO cost model (Section 5.2.2) also shows Monet to be at a relative disadvantage on low selectivity values or when very small results are obtained. This effect can indeed be seen in the relatively lagging performance of queries 2, 11 and 13 (which has $p = 3$, and $s = 0.001$). Only on query 1, the database hot-set outgrows main-memory size. This query has a selectivity 98% over the 6 million line items. Under such conditions, our algebraic buffer management starts to save intermediate results to disk to make room in main memory. A test run with explicit buffer management omitted, choked the system by excessive swapping. This shows the viability of our approach of including OS buffer management advise as an algebraic alterative in the query transformation. It should however be noted that Monet's policy of materializing intermediate results here is a disadvantage. Lagging performance for queries 5, 8 and 12 is related to a high number of page faults when processing complex sequences of joins. We think that more optimization could improve these numbers.

6.2.1 Detailed Performance Trace

To discuss Monet's performance in more detail, Figure 10 shows the execution results of a simplified[3] version of the MOA translation of Q13 to MIL.

The query starts with selecting all orders from `Order_clerk[oid,string]` for a certain clerk. Efficient binary search can be used, and the results are all stored consecutively, so this operation causes very few page faults. The returned 1459 orders are then joined with the `Item_order[oid,oid]` to get to the line items. Actually the `orders[oid,`

[3] All buffer management operations have been omitted. For full MOA and MIL scripts see the Monet web pages at tt http://www.cwi.nl/∼monet.

elapsed ms	faults		**MIL statement**
21	238	*1*	_orders := select(Order_clerk, "Clerk#000000088")
16102	7	*2*	_items := join(Item_order, _orders)
12932	3663	*3*	_returns := semijoin(Item_returnflag, _items)
5	0	*4*	_ritems := select(_returns, 'R')
2415	250	*5*	_critems := semijoin(Item_order, _ritems)
1653	331	*6*	_years := [year](join(_critems,Order_orderdate))
5	0	*7*	_class := group(_years)
6	0	*8*	INDEX := join(_ritems.mirror, _class).unique
7	0	*9*	YEAR := join(_class.mirror, _years).unique
2022	232	*10*	_prices := semijoin(Item_extendedprice,_critems)
2420	247	*11*	_discount := semijoin(Item_discount,_critems)
4	0	*12*	_factor := [-](1.0,_discount)
4	0	*13*	_rlprices := [*](_prices,_factor)
9	0	*14*	_losses := join(_class.mirror, _rlprices)
4	0	*15*	LOSS := {sum}(_losses)

Figure 10: Q13 Detailed Monet Execution Results

`oid]` is also ordered on tail, so the mergejoin implementation is used. In line 3 we semijoin them to get returnflags. This semijoin will go into the datavector-semijoin implementation, since the `Item_returnflags[oid,char]` is not sorted on `oid`, but has a datavector attached to it. The selection on the `_returns[oid,char]` with 5929 elements is cheap. The following join (line 5) is again a merge-join, since we still have sorted `oid`s. In lines 7-9 the grouping in classes according to year of the order is determined. Note the use of the multiplex `[year]()` operator to extract years from the sets of dates. The semijoin in 10 is again a datavector-semijoin. This is cheap, because the previous datavector-semijoin (line 3) has already blazed the trail into the extent. The costs are just the costs of fetching values from the vector; this repeats itself in line 11. The two multiplex operations that follow can be executed very efficiently, since the Monet kernel knows that the BATs `_prices[oid,float]` and `_discount[oid,float]` are `synced`. Both stem from a semijoin with a 100% match with the small relation `_critems[oid, oid]`, so they again are synced.

The result of the query are the three BATs `IN-`

`DEX[void, oid]`, `YEAR[oid,int]` and `LOSS[ oid, flt]`. MOA looks at them through through the structure function `SET(INDEX, TUPLE( YEAR, LOSS))`.

As we have seen, the active use of properties by the Monet kernel enables it to successfully choose efficient implementations at run-time, the datavector-semijoin being a winner among them: in many TPC-D queries it reduces the cost of multiple semijoins by more than half. Due to this intelligent semijoin execution, Monet is able to avoid being punished for its use of full vertical fragmentation, and is able to reap the benefits – namely – doing IO on very thin tables.

7 Conclusions

The large scale experiment reported here demonstrates progress in two key areas of modern database management. First, the experiments demonstrate convincingly that a DBMS kernel based on binary associations and a strong bias to exploit main-memory algorithms can be scaled to accommodate a disk-based decision support benchmark. Second, the mapping of an object-algebra to the binary relational platform using transformation rules can be achieved and proved correct. Taken together our results mark progress in developing small, yet extensible database kernels, which are applicable to a wide variety of database application scenarios using an object-oriented interface (See also, [BKK96,BQK96,HKM95]).

Currently, we are integrating the work on object database language bindings and method invocation described in [BKK96] into the algebraic context. More structure primitives, like the list, bag, and array, will be included in MOA. Research goals for Monet include further scaling of the database kernel technology to over 100GB databases via the exploitation of parallelism. Our experience from PRISMA [WFA95] is being used in a project were the MOA implementation will be extended to generate heterogeneously parallel MIL programs.

8 References

[AbK92] S. Abiteboul & P. C. Kanellakis, “Object Identity as a Query Language Primitive," in *Building an Object-Oriented database System. The story of O_2*, F. Bancilhon, C. Delobel & P. Kanellakis,eds., Mergan Kaufmann, San Mateo, California, 1992.

[BKK96] P. A. Boncz, F. Kwakkel & M. L. Kersten, “High Performance support for OO traversals in Monet," in *Proceedings British National Conference on Databases (BNCOD96) 1996*.

[BQK96] P. A. Boncz, C. W. Quak & M. L. Kersten, “Monet and its Geographic extensions," in *Proceedings of the 1996 EDBT Conference, Avignon, France*.

[CaD96] M. J. Carey & D. J. DeWitt, “Of objects and databases: a decade of turmoil," in *Proceedings of the 22th International Conference on Very Large Data Bases, Bombay, India, September 3-6, 1996.*, 3–15.

[CDN97] M. J. Carey, D. J. DeWitt, J. F. Naughton, M. Asgarian, P.Brown, J. E. Gehrke & D. N. Shah, “The BUCKY Object Relational Benchmark," in *Proceedings of the SIGMOD conference on management of data, Tucson, Arizona, USA, May 1997*.

[Cat94] R. G. G. Cattell, ed., *The Object Database Standard: ODMG-93*, Morgan Kaufmann Publishers, San Mateo, California, USA, 1994.

[ChZ96] M. Cherniak & S. B. Zdonik, “Rule languages and internal algebras for rule-based optimizers," in *Proceedings of the SIGMOD conference on management of data, Montreal, Canada, June 1996*.

[ClD92] S. Cluet & C. Delobel, “A general framework for the optimization of object-oriented queries," in *Proceedings of ACM-SIGMOD 1992 International Conference on Management of Data, San Diego, CA, June 2–5, 1992*, 383–392.

[CoK85] G. P. Copeland & S. N. Koshafian, “A decomposition storage model.," in *Proceedings of ACM-SIGMOD 1985 International Conference on Management of Data, Austin, TX, May 28–31, 1985*, 268–279.

[ERE95] EREQ, “The bakeoff query corpus," available at http://www.cse.ogi.edu/projects/ereq/bakeoff/schema.html, 1995.

[FeM95] L. Fegaras & D. Maier, “An algebraic framwork for physical OODB design," in *Proceedings of the 5th workshop on Database Programming Languages, Italy, 1995*.

[GKG97] T. Grust, J. Kröger, D. Gluche, A. Heuer & M. H. Scholl, “Query evaluation in CROQUE - calculus and algebra coincide.," in *Proceedings British National Conference on Databases (BNCOD15) 1997*.

[GuS95] R. H. Gueting & M. Schneider, “Realm-Based spatial data types: The rose algebra," *VLDB Journal* 4 (1995).

[HKM95] M. Holsheimer, M. L. Kersten & M. L. Mannila, “A perspective on databases and data mining.," in *Proc. Knowledge Discovery in Database ’95 Montreal, Can. (KDD95),.*

[LMS93] T. W. Leung, G. Mitchell, B. Subramanian, B. Vance, S. Vandenberg & S. B. Zdonik, “The Aqua data model and algebra," in *Proceedings of the 4th workshop on Database Programming Languages, 1993.*

[ShF94] T. Sheard & L. Fegaras, “Optimizing algebraic programs," OGI tech-report 94-004, Oregon Graduate Institute of Science and Technology, 1994.

[SAB94] H. J. Steenhagen, P. M. G. Apers, H. M. Blanken & R. A. de By, “From Nested-Loop to Join Queries in OODB," in *Proceedings of Twentieth International Conference on Very Large Data Bases, Santiago, Chile, September 12–15, 1994*.

[SBB96] H. J. Steenhagen, R. A. de By & H. M. Blanken, “Translating OSQL Queries into Efficient Set Exoessions," in *Proceedings of the 1996 EDBT Conference, Avignon, France.*.

[SRH90] M. Stonebraker, L. A. Rowe & M. Hirohama, “The implementation of POSTGRES," *IEEE Transactions on Knowledge and Data Engineering* 2 (March 1990).

[TPC95] TPC, “TPC Benchmark D," Transaction Processing Performance Council, 1995.

[WFA95] A. N. Wilschut, J. Flokstra & P. M. G. Apers, “Parallel evaluation of multi-join queries," in *Proceedings of the SIGMOD conference on management of data, San Jose, California, USA, May 1995*.