Moa: extensibility and efficiency in querying nested data

Maurice van Keulen[1] Jochem Vonk[1] Arjen P. de Vries[2]
keulen@cs.utwente.nl vonk@cs.utwente.nl arjen@cwi.nl
Jan Flokstra[1] Henk Ernst Blok[1]
flokstra@cs.utwente.nl blok@cs.utwente.nl

[1]Centre for Telematics and Information Technology, University of Twente
Enschede, The Netherlands
[2]Centrum voor Wiskunde en Informatica
Amsterdam, The Netherlands

Abstract

Advanced non-traditional application domains such as geographic information systems and digital library systems demand advanced data management support. In an effort to cope with this demand, we present a novel multi-model DBMS architecture which provides efficient evaluation of queries on complexly structured data. A vital role in this architecture is played by the Moa language featuring a nested relational data model based on XNF^2, in which we placed renewed interest. Furthermore, the architecture allows extensibility on all of its levels providing the means to better integrate domain-specific algorithms into the system. In addition to this, the extensibility of the Moa language is designed in a way that optimization obstacles due to black-box treatment of ADTs is avoided. This combination of well-integrated domain-specific algorithms, extensibility open to optimization, and a mapping of queries on complexly structured data to an efficient physical algebra expression via a nested relational algebra, makes that the Moa system can efficiently handle complex queries from non-traditional application domains.

Keywords: Nested relational algebra, NF2, DBMS architecture, database extensibility, advanced data management

1 Introduction

Advanced non-traditional applications, such as digital library systems and geographic information systems, place high demands on their data management components. Data in these areas is intrinsically complex and voluminous in nature and queries are computationally intensive. Researchers have sought to cope with these demands in different directions. In section 2, we explore these directions focussing on data model and DBMS architecture as a motivation for our multi-model DBMS architecture, as well as the particular role the logical algebra Moa plays in this architecture.

The three layers of the multi-model DBMS architecture support different data models. In this way, the top conceptual layer can provide a data model supporting complexly structured data, while in the logical and physical layers, a query on complexly structured data is gradually transformed to efficient storage-level operations. To be able to bridge the gap between a data model supporting complex structures (e.g., a hierarchical or object-oriented data model) and a storage-level data model, we place renewed interest in (X)NF^2 data models, which were popular in the 80's. The Moa data model, used in the logical layer of the architecture, is an adaptation and extension of XNF^2 for which we succesfully achieved to find an approach towards efficient query evaluation. Another key feature of our approach is that extensibility is utilized in all layers to be able to integrate domain-specific algorithms in such a way that certain optimization obstacles concerning ADTs are avoided.

In Section 3, we present Moa's approach to query processing. Subsequently, the Moa language is presented in Section 4 explained using various examples that also illustrate optimization potential of Moa. In Section 5, we show that Moa is an extensible algebra framework and that the presented data model is a specific instantiation of the Moa framework, namely its kernel. The ideas behind Moa have been validated and gradually fine-tuned in various research projects that concerned different non-traditional application domains, such as GIS and multi-media retrieval. A short overview of these projects is presented in Section 6. Finally, we present our conclusions and current and future work in Section 7.

2 Motivation and related research

Many directions exist in which one can attempt to cope with the high data management demands of many application domains. Some of the most prominent are discussed below.

2.1 Coping with high data management demands

Special-purpose systems Obviously, one can develop a special-purpose system that focusses on the limited data management functionality that is required. For example, a digital library can be tuned to the query patterns at hand using specific information retrieval algorithms. In this way, the limited context allows both the restriction of complexity as well as a way to tackle the performance problems. On the other hand, one does re-develop, in many occasions even re-invent, data management functionality that is readily available in off-the-shelve DBMS products. Moreover, the limited genericity and flexibility of dedicated systems makes them hard to adjust to future demands.

Object-orientation Programming languages often offer a rich object-oriented data model for dealing with complexly structured data. Applications that use a relational DBMS for the management of their persistent data, however, must 'disassemble' their nested data structures into atomic components and re-assemble them upon retrieval. This is the so-called *impedance mismatch*. Notice that the notion of impedance mismatch also refers to the difference between item-oriented thinking, encouraged by imperative programming languages, and the set-oriented approach, enforced by database languages. Apparently, object-oriented applications would benefit from an object-oriented data model at the interface with the DBMS to reduce the complexity for the application developer.

One way of achieving an object-oriented data model at the interface is to use an *object wrapper*. An object wrapper, however, considers the RDBMS as a black box. The object wrapper has to provide its own query optimizer, as the optimizer of the RDBMS does not know the strategy that generates the many queries resulting from a single query at the object level. Garlic [CHN$^+$95] is an example of a system that integrates special-purpose data servers using object wrappers. It contains its own optimizer that bases its decisions on statistics and tactical information obtained from the object wrappers. Unfortunately, even with provisions like these, performing such processing outside the scope of the database system may cause serious performance degradation, see, e.g.,[dVEK98].

To effectively deal with the impedance mismatch problem, people have attempted to develop a full-fledged object-oriented DBMS. An example of an OODBMS is Objectstore [LLOW91], which offers direct support for persistent objects. Its programmatic user-interface, however, is non-declarative and it didn't handle data independence well: the class structure used in the application often dictated the physical layout of the data in the DBMS. As known from RDBMSs, data independence is essential for scalability and data distribution. More like a real DBMS is the also well-known system of O_2 [BDK92], which has a structural object-oriented data model. An optimization technique that has been applied successfully in O_2's query language OQL is the transformation of a path expression into a join, which deals with the second kind of impedance mismatch we mentioned. But O_2 as well suffers from the data independence problem we mentioned. In the end, OODBMSs never became the success as anticipated, because the performance and flexibility problems were not solved. In our opinion, the lack of data independance is at the root of these problems.

Object-relational databases As Date and Darwen point out in [DD98], extensibility with user-defined data types does not require a new data model per se. The original definition of the concept *domain* says nothing about what can be physically stored. Object-relational DBMSs are still based on the relational data model, but allow new domain types to be defined thus enriching the relational data model. Other features of these systems are, for example, references, set-valued attributes, and type inheritance.

Evaluation experiments with the Bucky benchmark [CDN+97], designed to evaluate especially the extra features of the data models in OR-DBMSs, showed that a pure relational schema achieved much better performance in most cases than a schema using object-relational features such as set-valued attributes. Furthermore, encapsulation of data and operations inside objects or ADTs affect query evaluation: optimization by the DBMS becomes infeasible, and query processing too often resolves into object-at-a-time evaluation.

A promising approach to counteract this problem is the E-ADT approach of Predator [SP97]. In the implementation of the Predator DBMS, an E-ADT can implement an optimization interface to optimize the query plan using its own algebra, it can perform the evaluation of a query plan, it can extend the catalog with its own schema information and statistics, and it can provide multiple physical implementations of values of its type.

The E-ADT approach adheres to what is known in the field of software engineering as the *open implementation* approach [KLL+97]. Since ancient times, software has been constructed according to the principle that a module should expose its functionality, but hide

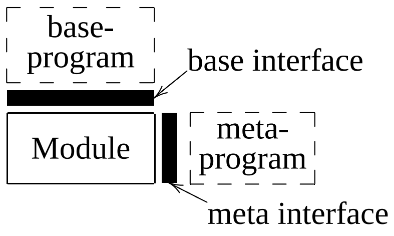


Figure 1: A module has two interfaces according to the open implementation approach.

its implementation. This *black-box abstraction* helps a developer to deal with complexity. In many cases, however, there is more than one possible implementation for the module. A developer may choose one, or try to build some intelligent code that transparently selects one. In some cases, however, the client of the module is better able to make a choice, so it is desirable to extend the interface of the module in such a way that the client can advise the module which implementation to take. This interface is called the *meta-interface* (see Figure 1). A typical example from operating systems is the memory management advice a program can give to the operating system that it, for example, will traverse a huge memory block sequentially, thus enabling the operating system to adjust its caching strategy accordingly. The optimization interface of an E-ADT is a further example of a meta-interface according to the open implementation approach.

The typical example of Predator is an E-ADT for images where $Clip(Rotate(img, 90), 10, 20, 19, 39)$ and $Rotate(Clip(img, 20, 40, 39, 49), 90)$ are equivalent, but that the latter is faster, because *Rotate* is an expensive operation and should preferrably be done on a small image. In the E-ADT, facilities exist to define that *Clip* can be pushed through a *Rotate* in the same way a *select* can be pushed through a *join*. Unfortunately, E-ADTs have their restrictions. Equivalence rules are defined solely *within* an E-ADT. We call this *intra-structure* optimization, since expressions within the E-ADT can be optimized with rules defined for that specific E-ADT. A query that, for example, selects images with a particular annotiation from a collection of images that are clipped and rotated, cannot, however, be optimized into an execution plan that will first do the selection and then the clipping and rotation. This is an example of *inter-structure* optimization where rules from different E-ADTs and/or relational operations cooperate; in this case, the select should be pushed down.

2.2 Multi-model architecture

From the above, it becomes clear that a few trade-offs have eluded researchers when trying to make DBMS technology better suitable to non-traditional applications. The first concerns data model expressiveness. The suitability of a DBMS for an application is closely related to the expressiveness of its data model. The data model of the conceptual level should fit the universe of discourse, since end-users have to understand this model of the real world in order to formulate their queries. If the application area inherently manipulates complexly structured data, a data model is required that supports the handling of such data. On the other hand, performance is expected from the DBMS, which typically means that it should be able to effectively optimize queries. The more complex the data model, however, the harder it gets to develop an effective optimizer, as the research on object-oriented DBMSs clearly showed. In the DBMS market today, the object-relational data model dominates claiming to be simple enough for query optimization, but expressive enough to handle special application areas, but as the Bucky benchmark has shown, room for improvement concerning performance exists.

To be able to deal with this trade-off, [Vri99] introduces the multi-model DBMS architecture with different data models on different layers. The conceptual level would typically have an object-oriented data model or a hierarchical semi-structured one. But instead of using such a data model throughout the DBMS architecture, we choose other, more 'simpler' data models for the logical and physical layers of the DBMS architecture. Obviously, this comes at a cost, namely additional mappings between layers, that map a query expression from one language to another. A typical choice for a data model and algebra on the physical level, would be one close to the machine, so for example, the same storage level relational algebra as ordinary RDBMSs, or, what we have used in many cases, the binary relational data model of main-memory DBMS Monet [BK99]. This leaves us with a gap to bridge going from such complex data models to a simple (binary) relational one. The XNF^2-data model [SP82] is very suitable as intermediairy data model, i.e., for the logical level. It handles complex data structures as nested relations, but still comes with an algebra that is not much more complex than an ordinary relational one. The idea of a DBMS based on XNF^2 [DKA$^+$86] lost interest when it appeared too difficult to build one that performed well. In this report, we will show how we adapted the XNF^2 data model (see Section 4) and used it effectively on the logical level of our multi-model DBMS prototype Moa (see Section 6).

The second trade-off concerns extensibility. To be able to manage complexity, one needs a module concept to hide implementation details. On the other hand, a module (or ADT)

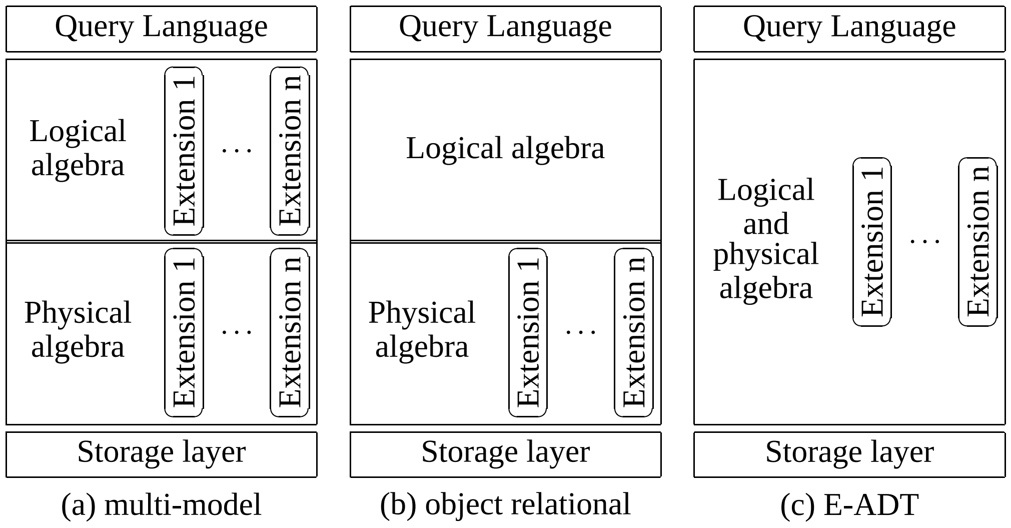


Figure 2: The multi-model DBMS architecture (a) next to the object relational (b) and E-ADT DBMS architectures (c).

as a black-box also hides implementation to the query optimizer, thus restricting its possibilities to effectively optimize queries. The open implementation approach provides an answer to this trade-off, as Predator has shown with its E-ADTs.

In effect, object-relational DBMSs only offer extensibility on the level of the physical algebra. Compared to this, the E-ADT approach of Predator seems to blur the distinction between logical and physical level. Our multi-model approach offers different kinds of extensibility on each level. A comparison of these different approaches to extensibility is shown in Figure 2. In Section 5, we show how the extensibility of Moa provides the possibility of inter-structure optimization.

Summarizing, the Moa DBMS prototype with its multi-model architecture has the potential of better meeting the demands of non-traditional application domains. By using different data models at different levels, it is possible to provide a data model supporting complex structures at the top while still being able to evaluate queries efficiently. We achieve the latter through several provisions. First, by utilizing an NF^2-based algebra as an intermediairy, we are able to more gradually and effectively translate queries on complexly structured data to efficient storage-level operations on decomposed (binary) relational data. Secondly, extensibility at all levels allows to better integrate domain-specific algorithms into the DBMS, thus improving the performance of domain-specific operations. Finally, the facilities for extensibility in the Moa language are defined in such a way that extensions are not black-boxes, but open to the optimizer, hence possible optimizations can be better exploited.

In the following three sections, we present the Moa system architecture, language, and framework, which is our realisation of a DBMS based on the multi-model DBMS architecture. The system evolved during several research projects of which more details are given in Section 6.

3 Query processing in Moa

3.1 NF^2 background

As early as 1982, Schek and Pistor argued that the relational data model is inconvenient for a domain like information retrieval [SP82]. To attack these shortcomings, they proposed to drop the first normal form (1NF) requirement to effectively allow non-atomic attribute domains such as sets of values. The 1NF relational data model can be regarded as a special case in this Non First Normal Form (or NF^2 for short) data model. This means that many definitions and theoretical conclusions of the relational model not dependent on 1NF, are also valid in NF^2.

Two types of nested algebras can be distinguished: the nest/unnest algebra, and the nested algebra, with and without explicit nesting. The respective algebras differ in the provisions that are taken for accessing attributes of relations nested within relations. In the nest/unnest algebra, tuples of relation-valued attributes are brought to the top level by unnesting these attributes. In the nested algebras, the operations to be applied to subrelations are brought to the subrelations concerned, either by using one or more algebraic operators as a navigator, or by employing path expressions. The main difference between NF^2 and eXtended NF^2 (XNF^2) data models such as that of the AIM DBMS [DKA$^+$86], is that XNF^2 data models support additional data types such as lists and allow for arbitrary nesting of type constructors. For a more detailed overview of algebras for the nested relational model, we refer to [Ste95].

The main part of the work on NF^2 concerns, however, the definition of algebras, not their function, which is to facilitate efficient query evaluation. Query processing in the context of the second type of nested algebras, where operations are brought to the subrelations, often results in nested-loop processing, which is often rather inefficient. Unnesting in a nest/unnest algebra, on the other hand, can cause data redundancy. Moreover, restructuring just to allow access to relation-valued attributes can be considered as pure computational overhead. Furthermore, such an algebra may even suffer from the infamous *COUNT-bug*, since in the presence of empty subsets, unnest is not the inverse of nest. The advantages of a nested relational model in the logical layer are evident, but its feasibility in a system where efficiency plays a role is another issue. [Ste95] has made a large contribution to the area of query optimization of nested relational algebras, among others, with the introduction of a special *nestjoin*-operator [SAB94]. Moa is an extension of XNF^2, where not only type constructors can be arbitrarily nested, but also new type constructors can be added (see Section 5).

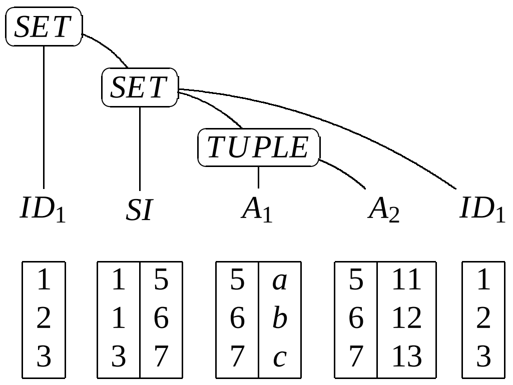


Figure 3: Nested data represented as flat data.

3.2 Mapping structure on flat data

As explained earlier, we use a 'flat' relational data model in the physical layer. The nested structures, therefore, need to be mapped to flat data, or in other words, NF^2 needs to be mapped to 1NF. Suppose, our database db is structured as a set of sets of n-tuples.[1] More formally in terms of a database universe, $db \in \mathcal{P}\mathcal{P}\, V_1 \times \ldots \times V_n$, where $\mathcal{P}$ is the powerset operator and V_i are domains of atomic values. For example, our database could look like $db = \{\{(a,11),(b,12)\},\{\},\{(c,13)\}\}$. The subsets of V_i used here are $v_1 = \{a,b,c\}$ and $v_2 = \{11,12,13\}$. Such a database can be represented as flat data in the following way:

$$\begin{aligned} ID_1 &= \text{set of as many unique id's as there are subsets in the database.} && (1)\\ ID_2 &= \text{set of as many unique id's as there are } n\text{-tuples in the database.} && (2)\\ SI &= \text{subset index as a set of pairs} \subseteq ID_1 \times ID_2 && (3)\\ A_i &= \text{columns as a set of pairs} \subseteq ID_i \times v_i \; (i \in \{1,2\}) && (4) \end{aligned}$$

The above is illustrated for our example db in Figure 3. At the bottom, the above sets are represented as flat tables and the rounded rectangles illustrate how these sets construct the original nested db.

3.3 Efficient query processing avoiding NF^2 pitfalls

In Section 3.1, we mentioned that both types of nested relational algebras are succeptible to inefficient query evaluation, either caused by nested-loop processing or by restructur-

[1]This is obviously a rude simplification, but the principles behind a generic mapping are apparent in this simplified case.

ing overhead and data redundancy. The advantage of a nest/unnest algebra is its ability to map operations on a nested structure to operations on flat data, thus avoiding nested-loop pitfalls, but at the expense of restructuring data redundancy. For the nested algebra, it is the other way around. Moa attempts to employ the advantages of both types, by using flat data processing while avoiding unnecessary restructuring. It does this by keeping an explicit structure definition in the form of type constructors connected to the flat data, and by having both nest/unnest operators as well as navigators such as *map* in the language. Furthermore, it effectively deals with the COUNT-bug by explicitly generating counteracting operations where needed, see the next section for examples.

The rounded rectangles in Figure 3 are in fact Moa type constructors and the tables at the bottom are called columns. As shown, Moa type constructors have columns or other Moa structures as arguments. The figure furthermore illustrates the distinction between a *value* and an *identified value set* (or *ivs*). Note that a column not only represents one atomic value, but a set of atomic values. By constructing a TUPLE structure using columns, the TUPLE structure actually represents a set of tuples, called an ivs, rather than one tuple value. It is the SET structure above it that introduces a partitioning of this set of tuples according to *SI*. This SET structure represents a set of sets, rather than one set, so it too is an ivs. The top-most SET doesn't introduce another partitioning, but only wraps things in a proper (nested) value. The top-most SET is, therefore, a Moa value. All structures in Moa have a value and an ivs form each with their own, albeit equally named, type constructor. The third argument of the SET ivs is used to deal with the COUNT-bug. Note that, based on *SI* alone, it is impossible to determine that an element identified by 2 exists that represents an empty subset, or that more empty subsets exist. The third argument of the SET ivs lists all elements. An empty subset, hence, is represented by an occurrence in the third argument and no occurrence in the first argument.

Each operator in our language is defined on two levels: on structure level and on data level. For example, the ivs variant of the aggregate *count* has the effect of converting a set-of-sets-valued argument to a set of atomic values on structure level, while at the same time generating a grouped count operation on the flat data underlying its argument. In other words, a query in Moa is translated into both a physical algebra expression on flat data, and the explicit (nested) structure definition of the result.

This two-level approach to query evaluation is illustrated in Figure 4. The general form of a Moa query is a Moa expression which uses columns from underlying tables (the leftmost pyramid). As explained, a column is only an abstraction for a set of atomic values. It is theoretically not obligatory that these atomic values are stored in a relation, but we

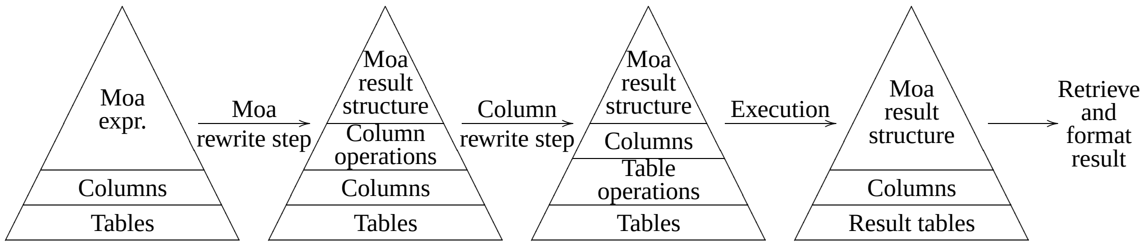


Figure 4: Moa query evaluation steps.

have, until now, used as physical storage representation of a column either a binary table (BAT) of Monet or a column from a table in some RDBMS. In the latter case, the column notation `|Emp:name|` represents the `name` attribute of table `Emp`. In the first rewrite step of the figure, Moa operations are mapped onto their respective column operations and result structure (second pyramid). Note that this is just a rewrite operation which causes only minimal overhead. Furthermore, note that this step converts a query on a nested structure to operations on flat data. In the third pyramid, the column operations have been translated to the table operations of the physicial layer. The third step performs the actual execution of the table operations producing result tables connected to a Moa result structure. In case an RDBMS is used for storage of columns and two or more columns are actually stored in the same table, operations on these columns can often be combined, for example, a selection on one column and a subsequent restriction of another to the corresponding values, becomes one select on the underlying table.

In the following section, we present the Moa language in more detail, in which the principles sketched above play an important role.

4 Moa logical language

The Moa logical language consists of two parts: the structures in which the data is modeled and the operations that can be applied on those structures. This section first introduces an example, that is used in the two subsequent subsections to illustrate the two parts of the Moa logical language.

4.1 Introduction

To illustrate the XNF^2-based model implemented in the Moa system and the logical language used therein to specify the data structure (schema) and queries, we use the well-known example of an organisation that has a number of departments and a number of employees that work in those departments, i.e., an organisation consists of a set of departments, which in turn consist of a set of employees.

A specific instantiation of such an organisation is shown in the organisation diagram presented in Figure 5. As can be seen from the figure, the organisation, a University, consists of the departments "Computer Science", "Electrical Engineering", and "Technical Healtcare". Three employees work in the computer science department, two employees work in the Electrical Engineering department, and the technical healtcare department has no employees (yet), as it is a newly founded department. The departments have a name and address attribute and the employees have a name and a salary attribute.

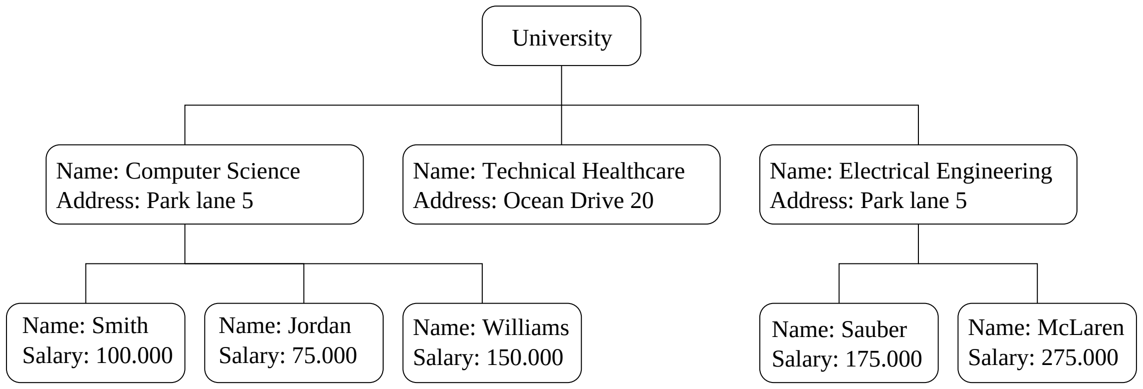


Figure 5: Example organisation structure.

In traditional 1NF relational terms, the department entity has a multi-valued attribute, i.e., the employees. It should be transformed into a seperate entity type with an additional relationship to the department entity type if this schema were to be stored in a relational

DBMS that adheres to the strict 1NF requirements. In the Moa system however, the nested structure of the schema can be preserved. Figure 6 shows the entire structure specification of the example organisation in the Moa logical language. In a first glance, without knowing the Moa logical language, one sees that the organisation is modeled more naturally, since the nesting is preserved and not flattened as is the case in the 1NF model. The next subsection explains the specification in detail by introducing the kernel structures of the Moa system.

```
SET<
  |Department:_key|,
  TUPLE<
    Atomic<|Department:DName|>: dname,
    Atomic<|Department:DAddress|>: daddress,
    SET<
      |EmpDep:_inverse|,
      TUPLE<
        Atomic<|Employee:EName|>: ename,
        Atomic<|Employee:Salary|>: salary
      >: Employee,
      |Department:_key|
    >: Employees
  >: Department
>: University
```

Figure 6: Schema of University.

4.2 Kernel structures

The structures Atomic, TUPLE, and SET in Figure 6 comprise the kernel structures that are implemented in the Moa system. These are called kernel structures being the bare minimum required to support the NF^2 data model. Other structures, e.g., LIST, are also implemented in the Moa system, but are not essential to the NF^2 data model and are therefore not considered as part of the kernel of the system, but as an *extension structure* (see Section 5).

Atomic< |*column-identifier*| > [: *label*]

For example, taken from the organisation structure:

```
Atomic<|Department:DName|>: dname
```

Figure 7: The Atomic kernel structure syntax.

The `Atomic` type constructor represents an atomic value as it is stored in the underlying database system. Figure 7 shows the syntax of the Atomic structure, together with an example taken from the organisation example. An atomic structure is specified by a keyword (Atomic), a column identifier, and an optional label.[2] The column identifier is the direct reference to the stored data of the underlying database system, which, in the case of a relational DBMS, is a column of a table. For example, `|Department:DName|` refers to the dname column of the Department table. The label of the structure is used as a convenience mechanism to be able to reference it in certain operations.

```
TUPLE< Moa-structure, ..., Moa-structure > [: label]
```

For example, taken from the organisation structure, a tuple that consists of two atomic structures. Note that any structure defined in Moa can be used as an attribute in a TUPLE structure:

```
TUPLE<
  Atomic<|Employee:EName|>: ename,
  Atomic<|Employee:Salary|>: salary
>: Employee
```

Figure 8: The TUPLE kernel structure syntax.

The `TUPLE` type constructor represents a product structure that consists of one or more Moa structures that can be of any type as shown in Figure 8. The purpose of assigning a label to a TUPLE structure is the same as for an Atomic structure, see above. The Moa tuple structure is similar to tuples found in relational database management systems.

```
SET< |key_mapping| ,
  Moa-structure,
  |key_of_enclosing_set| > [: label]
```

For example, an excerpt taken from the organisation structure: the sets containing all employees that belong to the departments.

```
SET<
  |EmpDep:_inverse|,
  TUPLE<
    ...
  >: Employee,
  |Department:_key|
>: Employees
```

In the above, '`_inverse`' and '`_key`' refer to special columns disclosing schema information, in this case the inverse of the relationship between employees and departments and the primary key of the department table, respectively. To avoid confusion, the `|EmpDep:_inverse|` column contains the relationship ('Smith',CS), ('Jordan',CS), ('Williams',CS), ('Sauber',EE), and ('McLaren',EE), where CS is 'Computer Science' and EE is 'Electrical Engineering'.

Figure 9: The SET kernel structure syntax.

The `SET` type constructor represents a collection of Moa structures. The possibility of

[2]represented by the square brackets

unrestricted nesting of type constructors makes it, in particular with this type constructor, possible to support all nestings of the NF^2 data model. The syntax of the type constructor, shown in Figure 9, consists of the SET keyword, an index mapping, the structure of the elements of this set, the index of the set that encloses the specified set, and an optional label. The index mapping is the mapping of the index[3] of the enclosing set to the index of the set being specified. Again, the label is used in the same way as the label for the atomic and tuple structures. Note that the outermost set, by contrast, is not an ivs (see Section 3.3), as it does not have an enclosing set and, hence, does not require the index mapping and the index of the enclosing set in its specification.

4.3 Operations

Several operations are defined on the kernel structures presented in the previous section. This section presents some of the most important of these operations, which, at the same time, illustrate how the Moa system avoids the typical pitfalls encountered in the mapping between the NF^2 data model and the flat relational data model, as already mentioned in Section 3.3.

The syntax to specify an operation uses a notation in which the operands are placed between parenthesis and the modifiers are placed between square brackets. A modifier is an expression that is brought to a sub-structure by some operations that have navigational capabilities. The operations described in this section are:

- **count**(*operand*),
- **select**[*modifier*] (*operand*),
- **attr**(*operand*),
- **map**[*modifier*] (*operand*),
- **join**[*modifier*, *modifier*] (*operand*, *operand*), and
- **flatten**(*operand*)

The operations are illustrated using queries in the context of the organisation structure introduced before. The example queries increase in complexity, e.g., by combining several operations, and each query shows a specific characteristic of the Moa system.

[3] In relational terms, the index is the same as the primary key of a table.

Note that the structure specification of the example organisation as shown in Figure 6 is stored in the data dictionary of the system. In this way, it is available by name ("University" in the example). A name and the expression it represents are equivalent, so the expression of Figure 6 is simply substituted in the queries below.

```
count(University);
```

The result of the query is:

```
3
```

Figure 10: Query 1: Counting the number of departments in the organisation.

The query shown in Figure 10 presents an example of an aggregate function, i.e., the count operation. The count operation is applied to the organisation structure (the operand) and counts the number of elements in that structure. In this case, the organisation is a set of department tuples, so that the count operation results in the number of departments.

```
select[=(attr(THIS, dname), "Computer Science")]
  (University);
```

is equivalent to the query:

```
select[=(%dname, "Computer Science")]
  (University);
```

The result of the query is:

```
{
  <Computer Science,Park lane 5,
  {
      <Smith,100000.0>,
      <Jordan,75000.0>,
      <Wiliams,150000.0>
    }
  >
}
```

Figure 11: Query 2: Select the department named "Computer Science".

Figure 11 shows the attr and select operations. The select operation is similar to the select-operation of relational algebra. The modifier (in square brackets) specifies the selection criterion and the operand (in round brackets) specifies the set-valued argument on which the selection should be applied. The THIS keyword in the modifier refers to an element of the argument. In this way, the modifier expression is brought to the sub-structure of SET similar to the navigation operators of the nested algebras mentioned in Section 3.1.

The attr operation (shorthand for attribute) is Moa's equivalent of projection, i.e., it evaluates to a particular attribute of its tuple-valued argument. 'dname' is the label specified for the atomic structure that represents the department name, see Figure 6. As an abbreviation, the attr operation can be replaced by a direct reference to the desired attribute using the label of the structure prefixed with a '%', which is shown in Figure 11 as the second query.

The example query selects the department named "Computer Science". The result of the query is the entire department structure including all attributes of that department, i.e., the name, address, and the set of employees belonging to it.

```
map[TUPLE<%dname, count(%Employees)>]
  (University);
```

The result of the query is:

```
{
  <Computer Science,3>,
  <Electrical Engineering,2>,
  <Technical Healtcare,0>
}
```

Figure 12: Query 3: Proper count handling.

The query of Figure 12 counts the number of employees per department. Not explicitly shown, but this query is not simply rewritten into a grouped count on the `|EmpDep:_inverse|` column shown in Figure 9, because of the so-called COUNT-bug mentioned in Section 3.3. If it was, it would have failed to produce the result for the 'Technical Healthcare' department. The third parameter of the SET type constructor is used to identify the departments for which to count the employees, see also Section 4.2.

This query also introduces the map operation and a structure constructor. The map operation is a pure navigational operator: it evaluates the modifier for each of the elements of its operand and collects the results in a SET structure. In this example, the count operation is performed on the elements of the organisation structure, i.e., the count is performed on the departments. The THIS keyword is present in the modifier here, but it is hidden in the '%' shorthand that we saw earlier. The structure constructor TUPLE<...> is used to specify the structure of the result elements. In this example, the result will be a tuple that contains the department name and the number of employees.

The avoidance of the nested-loop processing pitfall, as mentioned in Section 3.3, is shown by the query in Figure 13. The query selects all employees, grouped by department, and converts their salaries from dutch Guilders into Euros by dividing the salary through the currency conversion factor (i.e., 2.20371). The nested-loop pitfall is avoided by applying the data processing (the division operation) on the one column that contains the salary

```
map[
  map[TUPLE<%ename, /(%salary, 2.20371)>]
    (%Employees)]
      (University)
```

The result of the query is:

```
{
  {
    <Smith,45378.0216090139>,
    <Jordan,34033.5162067604>,
    <Wiliams,68067.0324135208>
  },
  {},
  {
    <Sauber,79411.5378157743>,
    <McLaren,124789.559424788>
  }
}
```

Figure 13: Query 4: Illustrate the set-oriented processing.

values, as shown in Figure 6. The other operations, like the group by, are processed using rewrite steps, as explained in Section 3.3.

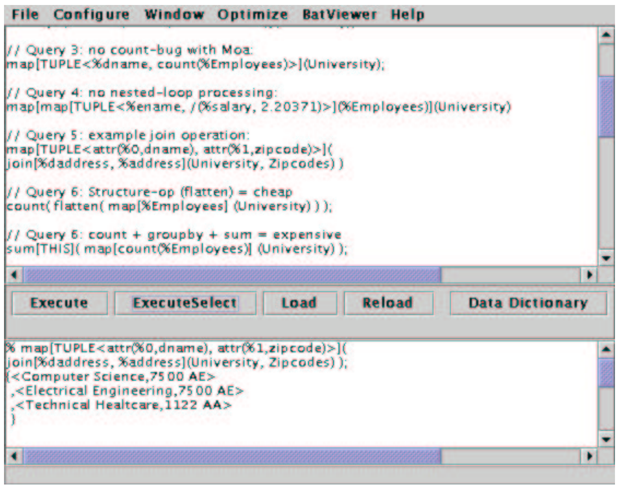
```
map[TUPLE<attr(%0,dname), attr(%1,zipcode)>](
  join[%daddress, %address]
    (University, Zipcodes) )
```

The result of the query is:

```
{
  <Computer Science,7500 AE>,
  <Electrical Engineering,7500 AE>,
  <Technical Healtcare,1122 AA>
}
```

Figure 14: Query 5: Joining two sets.

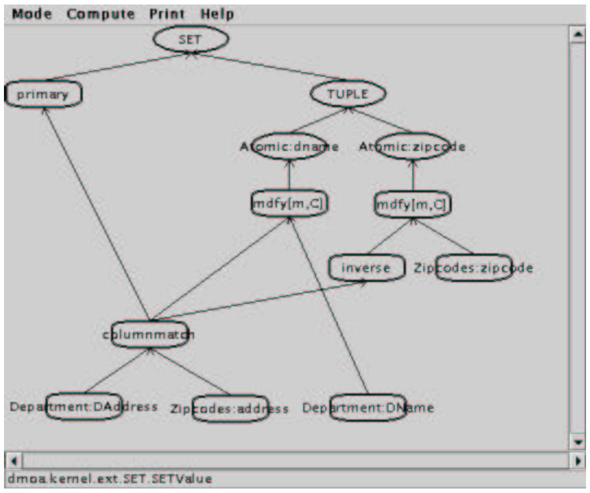
The join operation joins two structures based on equality of certain attributes of those structures, similar to the join operation in relational algebra. In the example shown in Figure 14, the organisation structure and the zipcodes structure are joined based on the same value of the address attribute. The zipcodes structure is a set of tuples, in which each tuple relates an address to a zipcode. The result of the query is a set of tuples, created by the tuple-constructor, containing the department name and its corresponding zipcode. The ‘%0’ and the ‘%1’ refer to the first and second argument of the operand, i.e., the ‘%0’ refers to the “University” set and the ‘%1’ refers to the “Zipcodes” set, of which the tuple constructor only takes the dname and the zipcode attributes.



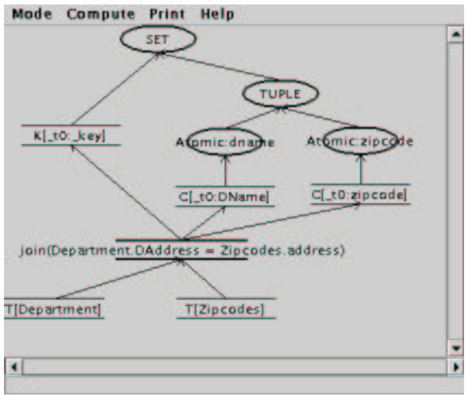
```
// Query 3: no count-bug with Moa:
map[TUPLE<%dname, count(%Employees)>](University);

// Query 4: no nested-loop processing:
map[map[TUPLE<%ename, /(%salary, 2.20371)>](%Employees)](University)

// Query 5: example join operation:
map[TUPLE<attr(%0,dname), attr(%1,zipcode)>](
join[%daddress, %address](University, Zipcodes) )

// Query 6: Structure-op (flatten) = cheap
count( flatten( map[%Employees] (University) ) );

// Query 6: count + groupby + sum = expensive
sum[THIS]( map[count(%Employees)] (University) );
```

```
% map[TUPLE<attr(%0,dname), attr(%1,zipcode)>](
join[%daddress, %address](University, Zipcodes) );
{<Computer Science,7500 AE>
 ,<Electrical Engineering,7500 AE>
 ,<Technical Healtcare,1122 AA>
 }
```

(a) Input/output window.



(b) Column algebra tree view.



(c) Target language tree view.

Figure 15: Screendump of Moa System executing Query 5

Figure 15 shows a screen dump of the Moa system in which this query (Query 5) has been executed. The graphical user interface, shown in Figure 15(a), is used to execute the queries and the display the result. Figure 15(b) and Figure 15(c) show, using graphs, some intermediairy results of the rewrite steps that are explained in Section 3.3. The column algebra tree is the state of the query after rewriting to column algebra. The target language tree shows the operations as they will be executed by the target database system in which the actual data is stored. In this case, the target database is a relational database system.[4] Therefore, the query is eventually translated to SQL. A detailed explanation of the column algebra and target database mapping is beyond the scope of this report; a report on this topic is currently being written.

Note that all operations described above, can either affect the structure of the operands, the data itself, or, in most cases, both. The map and attr operations are examples of operations that only affect the structure, select and count affect both. Another operation that mainly has an effect on the structure, is flatten, which converts a set of sets to one set by taking the union of all sets, i.e., it removes one level of nesting. Figure 16 shows the use of an application of this operation.

Figure 16 also demonstrates an optimization opportunity. Both queries aimed at counting the number of employees in the organisation, are equivalent. It is more efficient to first execute a flatten on the structure, creating a structure that contains the set of all employees, and then count those employees, than to count the employees per department and then take the sum. The fact that the structure operation is more efficient, is clearly shown in Figure 17. Figure 17(a) is the target database mapping of the query using the flatten operation, which rewrites to one join and an aggregate, while Figure 17(b) shows the alternative query, which rewrites to two joins and two aggregates, which is evidently more expensive to execute.

Besides the operations that are described in this section, other operations are available in the Moa system, e.g., the nest and unnest operations. However, due to space limitations, it is infeasible to present an exhaustive list.

[4]The Moa system currently supports IBM DB2, PostgreSQL, and MySQL.

```
count(
  flatten(
  map[%Employees]
      (University) ) );
```

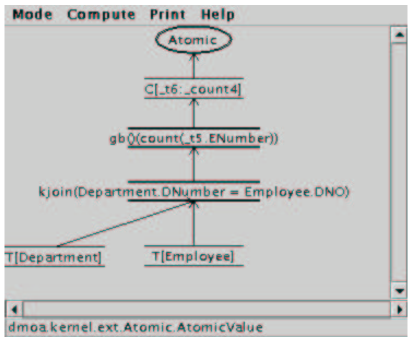
The above query is more efficient than the query below.

```
sum[THIS](
  map[count(%Employees)]
  (University) );
```

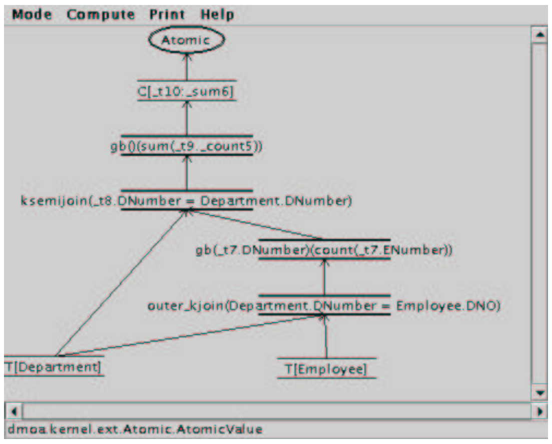
The result of the query is:

```
5
```

Figure 16: Query 6: Illustration of optimization potential.



(a) Target language tree of first alternative.


Mode
Compute
Print
Help
Atomic
C[_t10:_sum6]
gb()(sum(_t9._count5))
ksemijoin(_t8.DNumber = Department.DNumber)
gb(_t7.DNumber)(count(_t7.ENumber))
outer_kjoin(Department.DNumber = Employee.DNO)
T[Department]
T[Employee]
dmoa.kernel.ext.Atomic.AtomicValue

(b) Target language tree of second alternative.

Figure 17: Illustration of optimization potential.

5 Extensibility

Besides the features of the Moa system described in the previous sections, another key feature of the Moa system is extensibility. New structures with accompanying operations can be defined. In fact, the structures we have seen so far, SET, TUPLE, and Atomic, do not differ in any way from user-defined structures other than being pre-defined in the system. Hence, Moa can be better classified as a generic framework in which structures can be arbitrarily nested. The immediate consequence of the latter combined with our choice of kernel structures is that this defines an NF^2 data model. Since both collection structures, like LIST, can be defined as well as 'normal' domain-specific structures, Moa can be said to be extensible XNF^2, or rather X^{2}NF^2.

Another novel aspect of Moa is that an operation is 'implemented' in terms of (abstract) column algebra operators (see Section 3.3), thus allowing a subsequent global optimization of the column algebra expression of an entire query, hence providing an opportunity for the much desired inter-structure optimizations.

To illustrate the extensibility possibilities of the Moa system, it is extended with a simple new structure 'FV' containing a few operations. The name of the new structure 'FV' is an abbreviation for feature vector. Its use in a query is shown in Figure 18. The example is taken from the multimedia domain, content-based querying is realised by extracting features from images and retrieving images based on some distance measure between features. Examples of features are color histogram and color structure.

It is modeled as an identifier and a number of feature values. An operation that is specific for this new structure is the distance operation. This operation calculates the distance of two feature vectors, based on the distance measure below (*i* ranges over the individual feature values).

$$distance(\vec{v}_1, \vec{v}_2, \vec{w}) = \sum_i |\vec{w}[i](\vec{v}_1[i] - \vec{v}_2[i])|$$

The smaller the distance between features, the more similar the images corresponding to those feature are. The query of Figure 18 returns the three most similar images based on the features of the images stored in the database and the fixed feature vector FV<"", 2.0, 2.0>. The example is simplified for reasons of clarity; real-life feature vectors can consist of hundreds of values. Other operations shown in the figure are sort, which sorts the elements of a collection in either ascending or descending order, and a top operation, which returns only the first *n* elements of its operand. These operations are examples of the support for ordered sequences in Moa.

```
top[3](
  sort['ASC',%dist](
    select[sm(%dist,2.3)](
      map[TUPLE<dist(this,FV<"",2.0,2.0>):dist,%MediaURL>](
      SET<
        |FeatBase:_key|,
        FV<
          Atomic<|FeatBase:MediaURL|>:MediaURL,
          Atomic<|FeatBase:Value1|>,
          Atomic<|FeatBase:Value2|>
>>))));
```

The result of the query is:

```
{
  <2.15,http://.../artfinder/AV-Content/V2/8610B12.jpg>,
  <2.2,http://.../artfinder/AV-Content/V2/DSCN0492.JPG>,
  <2.25,http://.../artfinder/AV-Content/V2/deaf00_symposium_day2_06.JPG>
}
```

Figure 18: Example query illustrating the feature vector (FV) extension.

6 Application areas and context

The ideas for Moa and multi-model architecture originate from the Magnum-project [WvZF$^+$98, BWK98, BQK96]. In this project, a structurally object-oriented DBMS was developed for the purpose of efficiently integrating spatial and thematic data in a single data manager. Reseach has shown in the early nineties, that full and efficient integration of GIS functionality in an extensible relational or an object-oriented DBMS based on ADT-like GIS extensions, was difficult. Therefore, *decomposition* and *extensibility* were the key features of this project.

In terms of the multi-model architecture, the Magnum system consisted of two layers: the main-memory DBMS Monet as physical layer and Moa as logical layer. This architecture could be extended in two ways. First, new base types could be defined in Monet, e.g., point, line, and polygon, together with a large set of spatial operators on these primitive base types. Secondly, Moa's structural extensibility was used to support structures like polygonal maps, triangulations, and rasters, next to the conventional tuple, set, and list. Moa mapped these structures to Monet's binary data model, meaning that the highly structured data was decomposed in many binary tables. Experiments showed that the Magnum system performed well on the Sequoia benchmark [SFGM93].

The good experiences with the Moa/Monet combination, especially with the combination of base type and structural extensibility, sparked off new efforts. If the generic features of this DBMS could be applied to realise a well-performing GIS, why wouldn't that work for other non-traditional domains as well? In the Mirror and AMIS projects, the ideas for a highly extensible DBMS architecture based on Moa and Monet were further developed in the realm of text and multimedia retrieval. Mirror concentrated on a generic multimedia retrieval framework based on belief networks. Experiments showed its feasibility for content-based retrieval for text, images (based on color and texture features), and music (based on rhythm features) [dVvDBA99, Vri99]. Since parallelisation and fragmentation in the physical layer is orthogonal to the logical layer, the architecture design seems to be better prepared to scale up. The AMIS-project explored this idea by studying the optimization of top-N IR-queries in a fragmented context [BdVBA01, BHC$^+$01].

In all three projects described above, the integration of data and algorithms from non-traditional application domains in a single data manager is a central theme. Much thematic (tabular) data related to GIS or multimedia objects, however, resides in RDBMSs with existing applications running on them. Transferring this data to Monet, hence, is, from an information system engineering point-of-view, not a viable option. Therefore,

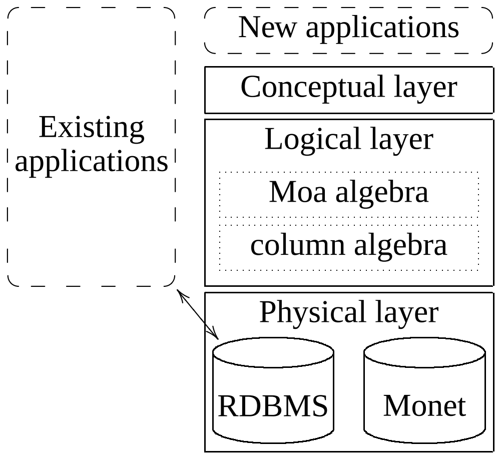


Figure 19: SUMMER federated multi-model architecture.

in the SUMMER-project, the multi-model architecture was taken one step further by using Moa as a kind of *data management middleware*, driving both Monet and ‘normal’ RDBMSs (see Figure 19) simultaneously. This allows the ‘addition’ of, for example, multimedia retrieval functionality to an existing federated information system. The Moa algebra was extended with a column algebra inspired by Monet’s binary algebra to obtain more independence from the physical layer. Also in SUMMER, we started the development of an XML-based conceptual layer. At the time of writing, the SUMMER-project is still running, and no references to this work are available, yet.

7 Conclusions and future work

In this report, we presented the multi-model DBMS architecture and the Moa logical algebra which plays an important role therein in an attempt to cope with high data management demands in non-traditional application domains. In order to support applications like GIS or digital libraries, one needs an expressive conceptual data model supporting complexly structured data. Expressiveness, however, is not the only requirement. Since the managed data is often voluminous and queries are complex in nature, performance is an important aspect as well. The multi-model architecture supports extensibility in all three layers thus enabling to integrate domain-specific algorithms in an effective way. Furthermore, the extensibility mechanism of the Moa language used in the logical layer of this architecture, has been designed in such a way that optimization across extensions (inter-structure optimization) is possible. This alleviates the problem with usual ADT-based extension mechanisms that an ADT is a black-box for the optimizer, thus prohibiting pushing, for example, projections and selections through ADT-operators.

To be able to bridge the gap between an expressive conceptual data model at the top and an efficient simple physical data model at the bottom, the nested relational approach proved effective. We placed renewed interest in it, adapted and extended an XNF^2 algebra, and worked on new ways for efficient query evaluation. This resulted in the Moa language presented in Sections 4 and 5. We regard its role to be vital in the success of the multi-model architecture.

In several projects, a prototype DBMS evolved into what is now called the Moa system. Addressing different application domains, the genericity, extensibility, and performance of the system were put to the test. An overview of those projects was given in Section 6.

In current and future projects, we will continue the work on the DBMS prototype and the ideas and languages applied therein. First, we are currently developing a conceptual layer based on XML as data model and XQuery as query language. This will make the DBMS suitable to be used in web-based environments, providing a more convenient way of managing large XML data volumes with, among others, integrated and efficient multimedia retrieval. Secondly, we are extending the DBMS prototype with distribution support. In this way, the Moa system can function as a kind of data management middleware facilitating the construction of federated systems. Thirdly, in our research group, the Moa system is used as an experimentation platform, which imposes a continuous demand for perfecting the extensibility and efficiency of the system. Finally, we recently started to put more effort into better exploiting the optimization potential, inter-structure optimization in particular. In a PhD project, we are exploring the realm of category theory in search for ways to fundamentally improve the Moa and column algebra. Furthermore, in a project which starts in the winter of 2002 concerning information retrieval based on the work of [Hie01], we intend to improve the optimizer.

Acknowledgements

Special thanks to Annita Wilschut, who unfortunately no longer works in the field of computer science. It were her ideas that stood at the birth of Moa. All participants of the SUMMER project are acknowledged for their contribution to the work presented in this report. Furthermore, we would like to thank the Data Mining and Knowledge Discovery group of CWI for many years of pleasant and fruitful cooperation.

References

[BDK92] F. Bancilhon, C. Delobel, and P. Kanellakis, editors. *Building an Object-Oriented Database System : The Story of O_2*. Morgan Kaufmann Publishers, 1992.

[BdVBA01] H.E. Blok, A.P. de Vries, H.M. Blanken, and P.M.G. Apers. Experiences with IR Top N optimization in a main memory DBMS: Applying 'the database approach' in new domains. In Brian Read, editor, *Proceedings of the British National Conference on Databases (BNCOD 18), Advances in Databases*, volume 2097 of *Lecture Notes in Computer Science*, pages 126–151. Springer, July 2001.

[BHC$^+$01] H.E. Blok, D. Hiemstra, S. Choenni, F. de Jong, H.M. Blanken, and P.M.G. Apers. Predicting the cost-quality trade-off for information retrieval queries: Facilitating database design and query optimization. In *Proceedings of the 2001 ACM CIKM International Conference on Information and Knowledge Management, Atlanta, Georgia, USA*, pages 207–214. ACM, November 2001.

[BK99] Peter A. Boncz and Martin L. Kersten. MIL primitives for querying a fragmented world. *VLDB Journal*, 8(2):101–119, 1999.

[BQK96] Peter A. Boncz, Wilko Quak, and Martin L. Kersten. Monet and its geographic extensions: A novel approach to high performance GIS processing. In Peter M. G. Apers, Mokrane Bouzeghoub, and Georges Gardarin, editors, *Advances in Database Technology - EDBT'96, 5th International Conference on Extending Database Technology, Avignon, France, March 25-29, 1996, Proceedings*, volume 1057 of *Lecture Notes in Computer Science*, pages 147–166. Springer, 1996.

[BWK98] Peter A. Boncz, Annita N. Wilschut, and Martin L. Kersten. Flattening an object algebra to provide performance. In *Proceedings of the Fourteenth International Conference on Data Engineering, February 23-27, 1998, Orlando, Florida, USA*, pages 568–577. IEEE Computer Society, 1998.

[CDN$^+$97] Michael J. Carey, David J. DeWitt, Jeffrey F. Naughton, Mohammad Asgarian, Paul Brown, Johannes Gehrke, and Dhaval Shah. The BUCKY object-relational benchmark (experience paper). In Peckham [Pec97], pages 135–146.

[CHN$^+$95] William F. Cody, Laura M. Haas, Wayne Niblack, Manish Arya, Michael J. Carey, Ronald Fagin, Myron Flickner, D. Lee, Dragutin Petkovic, Peter M. Schwarz, Joachim Thomas II, Mary Tork Roth, John H. Williams, and Edward L. Wimmers. Querying multimedia data from multiple repositories by content: the Garlic project. In Stefano Spaccapietra and Ramesh

Jain, editors, *Visual Database Systems 3 (VDB 3), Visual Information Management, Proceedings of the third IFIP 2.6 working conference on visual database systems, March 27–29, 1995, Lausanne, Switzerland*, volume 34 of *IFIP Conference Proceedings*, pages 17–35. Morgan Kaufmann, 1995.

[DD98] C.J. Date and H. Darwen. *Foundation for Object/Relational Databases: the Third Manifesto*. Addison-Wesley, 1998.

[DKA$^+$86] Peter Dadam, Klaus Küspert, F. Andersen, Henk M. Blanken, R. Erbe, Jürgen Günauer, Vincent Y. Lum, Peter Pistor, and Georg Walch. A DBMS prototype to support extended NF^2 relations: An integrated view on flat tables and hierarchies. In Carlo Zaniolo, editor, *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 28-30, 1986*, pages 356–367. ACM Press, 1986.

[dVEK98] Arjen P. de Vries, Brian Eberman, and David E. Kovalcin. The design and implementation of an infrastructure for multimedia digital libraries. In Barry Eaglestone, Bipin C. Desai, and Jianhua Shao, editors, *Proceedings of the 1998 International Database Engineering and Applications Symposium (IDEAS), Cardiff, Wales, U.K., July 8-10, 1998*, pages 103–120. IEEE Computer Society, 1998.

[dVvDBA99] Arjen P. de Vries, Mark G. L. M. van Doorn, Henk M. Blanken, and Peter M. G. Apers. The Mirror MMDBMS architecture. In Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, editors, *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 758–761. Morgan Kaufmann, 1999.

[Hie01] Djoerd Hiemstra. *Using Language Models for Information Retrieval*. PhD thesis, University of Twente, January 2001.

[KLL$^+$97] Gregor Kiczales, John Lamping, Christina Videira Lopes, Chris Maeda, Anurag Mendhekar, and Gail Murphy. Open implementation design guidelines. In *Proceedings of the 19th international conference on Software engineering*, pages 481–490. ACM Press, 1997.

[LLOW91] Charles Lamb, Gordon Landis, Jack A. Orenstein, and Danel Weinreb. The objectstore database system. *CACM*, 34(10):50–63, 1991.

[Pec97] Joan Peckham, editor. *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*. ACM Press, 1997.

[SAB94] Hennie J. Steenhagen, Peter M. G. Apers, and Henk M. Blanken. Optimization of nested queries in a complex object model. In Matthias Jarke, Janis A. Bubenko Jr., and Keith G. Jeffery, editors, *Advances in*

Database Technology - EDBT'94. 4th International Conference on Extending Database Technology, Cambridge, United Kingdom, March 28-31, 1994, Proceedings, volume 779 of *Lecture Notes in Computer Science*, pages 337–350. Springer, 1994.

[SFGM93] Michael Stonebraker, James Frew, Kenn Gardels, and Jeff Meredith. The Sequoia 2000 benchmark. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993*, pages 2–11. ACM Press, 1993.

[SP82] Hans-Jörg Schek and Peter Pistor. Data structures for an integrated data base management and information retrieval system. In *Eigth International Conference on Very Large Data Bases, September 8-10, 1982, Mexico City, Mexico, Proceedings*, pages 197–207. Morgan Kaufmann, 1982.

[SP97] Praveen Seshadri and Mark Paskin. PREDATOR: An OR-DBMS with enhanced data types. In Peckham [Pec97], pages 568–571.

[Ste95] Hennie Steenhagen. *Optimization of Object Query Languages*. PhD thesis, University of Twente, 1995.

[Vri99] A.P. de Vries. *Content and Multimedia Database Management Systems*. PhD thesis, University of Twente, 1999.

[WvZF$^+$98] Annita N. Wilschut, Roelof van Zwol, Jan Flokstra, Nick Brasa, and Wilko Quak. Road collapse in Magnum. In Robert Laurini, Kia Makki, and Niki Pissinou, editors, *ACM-GIS '98, Proceedings of the 6th international symposium on Advances in Geographic Information Systems, November 6-7, 1998, Washington, DC, USA*, pages 20–27. ACM, 1998.