[10] A.K. Chandra and D. Harel. Horn clauses and generalization. *Journal of Logic Programming*, 2(1):320–340, 1985.

[11] Y. Gurevich and S. Shelah. Fixed-point extensions of first-order logic. *Annals of Pure and Applied Logic*, 32:265–280, 1986. Also in *Proceedings Int. Conf. on Foundations of Computer Science*, pp. 346–353, 1985.

[12] T. Ibaraki and T. Kameda. On the optimal nesting order for computing n-relational joins. *ACM Trans. on Database Syst.*, 9(3):483–502, 1984.

[13] P.G. Kolaitis and C. Papadimitriou. Why not negation by fixpoint. In *Seventh ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems*, pages 231–239, 1988.

[14] T. J. Lehman and M. J. Carey. Query processing in main memory database management systems. In *ACM SIGMOD International Conf. on Management of Data*, 1986.

[15] T. J. Lehman and M. J. Carey. A study of index structures for main memory database management systems. In *Twelfth International Conference on Very Large Data Bases, Kyoto*, pages 294–303, 1986.

[16] U. Nanni, S. Salza, and M. Terranova. An algebraic approach to the manipulation of complex objects. In *25-th Hawaii International Conference on System Sciences*, IEEE Press, Kaloa, Hawaii, January 7-10, 1992.

[17] U. Nanni, S. Salza, and M. Terranova. The LOGIDATA+ Object Algebra. Technical Report of the International Computer Science Institute, TR-92-006, 1992.

[18] S. Salza and M. Terranova. *Efficient Implementation of Object-oriented Databases on Relational Systems.* Technical Report, IASI-CNR, 1991.

[19] E.J. Shekita and M.J. Carey. A performance evaluation of pointer-based joins. In *ACM SIGMOD International Conf. on Management of Data*, pages 300–311, 1990.

[20] J.D. Ullman. *Principles of Database and Knowledge Base Systems.* Volume 1, Computer Science Press, Potomac, Maryland, second edition, 1982.

[21] S. B. Yao. Approximating block accesses in database organisations. *Communications of the ACM*, 20(4):260–261, 1977.

multiplicity, can be computed from a user supplied quantitative characterization of the object schema, e.g. average cardinality of the set attributes etc.

This kind of approach is particularly meaningful in the case of precompiled queries. Moreover in a context of predefined transactions we may consider another level of optimization, based on the idea of improving the execution cost by means of transformations on the canonical schema. These consist in precomputing some of the joins on the link attributes. This leads to consider all the class of *admissible schemata*, where the optimal schema for a given workload, is the one with the best tradeoff between the increase in access cost due to larger relations, and the saving connected to prejoins. These issues are discussed in [18] , where a detailed cost model is presented, and the applicability of some heuristics is analized.

References

[1] S. Abiteboul and P. Kanellakis. Object identity as a query language primitive. In *ACM SIGMOD International Conf. on Management of Data*, pages 159–173, 1989.

[2] S. Abiteboul and V. Vianu. Procedural and declarative database update languages. In *Seventh ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, 1988.

[3] P. Atzeni, F. Cacace, S. Ceri, and L. Tanca. *The LOGIDATA+ model.* Rapporto LOGIDATA+ 5/20, Progetto Finallizzato Sistemi Informatici e Calcolo Parallelo, 1990.

[4] P. Atzeni and L. Tanca. *The LOGIDATA+ Rule Language.* Rapporto LOGIDATA+, Politecnico di Milano e IASI-CNR, Roma, 1990. Presented at the Workshop "Information Systems '90", Kiev.

[5] F. Bancilhon et al. The design and implementation of O_2 an object-oriented database system. In *Advances in Object-Oriented Database Systems, Proc. Second Int. Workshop on Object-Oriented Database Syst., K. Dittrich, Ed., Bad Munster, FRG*, 1988.

[6] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *ACM SIGMOD International Conf. on Management of Data*, pages 16–52, 1986.

[7] F. Cacace, S. Ceri, S. Crespi-Reghizzi, L. Tanca, and R. Zicari. Integrating object oriented data modelling with a rule-based programming paradigm. In *ACM SIGMOD International Conf. on Management of Data*, pages 225–236, 1990.

[8] S. Ceri, G. Gottlob, and Tanca L. *Logic Programming and Databases.* Springer-Verlag, Berlin, 1990.

[9] A.K. Chandra. Theory of database queries. In *Seventh ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 1–9, 1988.

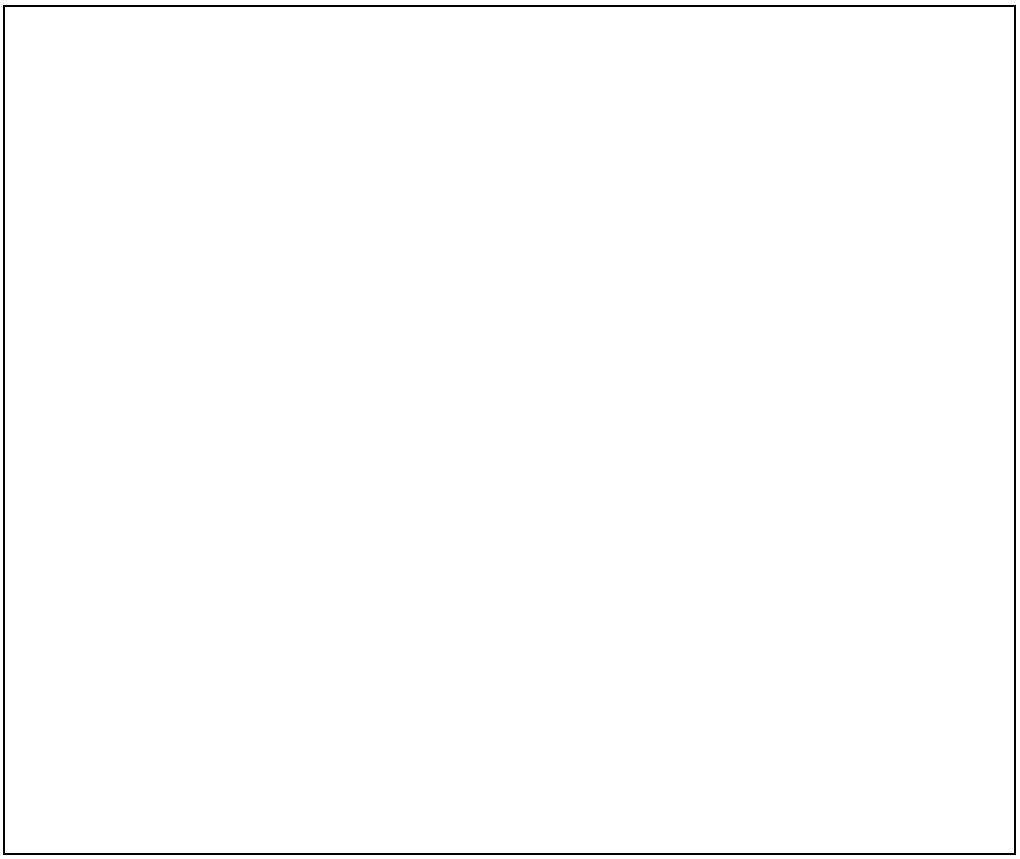

Figure 10: The relational query tree

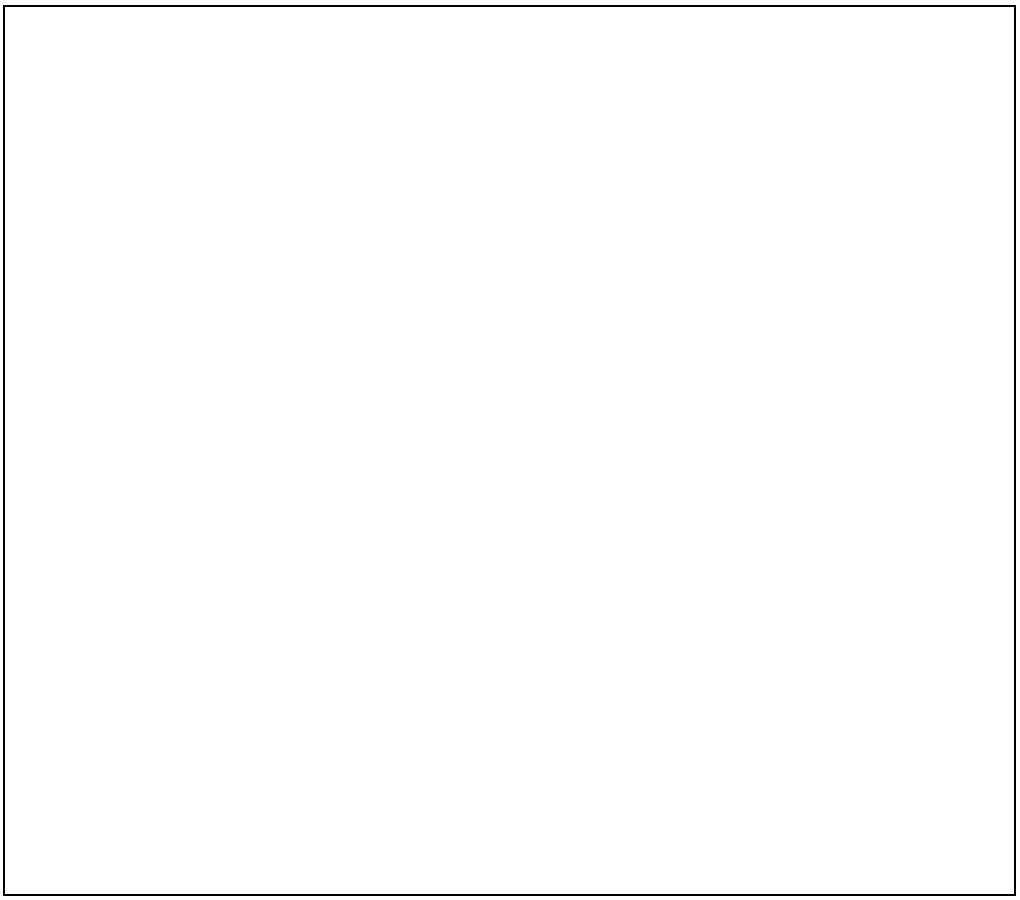

Figure 9: The instance tree

the same set. To do so, we introduce the notion of *instance tree*, defined as a subtree of the value tree, in wich for every set node only one child, if any, is retained. We then say that an object satisfies a condition if and only if it exists for that object an instance tree that satisfies the condition, and that a selection query $\mathcal{Q}(\mathcal{C};\mathcal{B})$, on the class $\mathcal{C}$ selects all the objects in $\mathcal{C}$ that satisfy the condition $\mathcal{B}$.

For instance considering the object selection query:

$$\begin{aligned}\texttt{Q1} : (&\texttt{STUDENT}; \texttt{STUDENT.curriculum.course} =' \texttt{MATH1}' \\ &\wedge\ \texttt{STUDENT.curriculum.test.grade} =' \texttt{A}' \\ &\wedge\ \texttt{STUDENT.curriculum.test.year} = 1990)\end{aligned}$$

we see that the object of Figure 8 is not selected by the query, although each atomic condition is satisfied by an atomic value in the value tree. Instead the query:

$$\begin{aligned}\texttt{Q2} : (&\texttt{STUDENT}; \texttt{STUDENT.curriculum.course} =' \texttt{MATH1}' \\ &\wedge\ \texttt{STUDENT.curriculum.test.grade} =' \texttt{A}' \\ &\wedge\ \texttt{STUDENT.attends.course} =' \texttt{MATH2}')\end{aligned}$$

selects the object, since the condition is satisfied by the instance tree of Figure 9.

According to what we have seen in Section 3 an object selection query can be readily translated into a a relational query on the canonical schema. The resulting relational expression is in general quite complex, and its optimization is a crucial problem. This is because evaluating conditions on objects with complex nested structure, may require to perform a large number of joins, mostly on the link attributes introduced by the mapping of the schema.

To discuss the optimization strategy we focus on a sub-class of the selection queries, the *sd-queries* in which the condition $\mathcal{B}$ is *decomposable*, i.e. has conjuntive form $\mathcal{B} = \mathcal{B}_1 \wedge \mathcal{B}_2 \wedge \ldots \wedge \mathcal{B}_k$, and each $\mathcal{B}_i$ contains only atomic conditions on the attributes of a single tuple constructor. For these queries the optimization problem has a very clean formulation, since they can be translated into relational SPJ queries, consisting in a series of selections on the base relations of the canonical schema, followed by a chain of joins on link attributes. As an example Figure 10 shows the relational query tree for query `Q2`.

In most RDBMs, such multijoin queries are usually computed with a nested loop algorithm, provided that indices are maintained on all the join attributes. This avoids the materialization of the intermediate results, and exploits the selectivity of the select and the join conditions. In our case, at least for the sd-queries, this strategy fits very well, since all the joins are performed on the link and oid attributes, and it is very reasonable to keep an index on each link and oid attribute.

The optimization of the nested loop computation of the multijoin queries has been given a good deal of attention in the literature. The basic problem is the ordering of the joins to maximize the selectivity along the query tree, and hence to minimize the number of page fetches. The problem is NP-complete, and therefore intractable for large number of joins. Neverthless some reasonably efficient euristic algorithms have been proposed [12, 19].

In our case the cost model can be easily defined, since the joins are on link attributes. Hence estimates of their extensional and statistical parameters, such as originality and

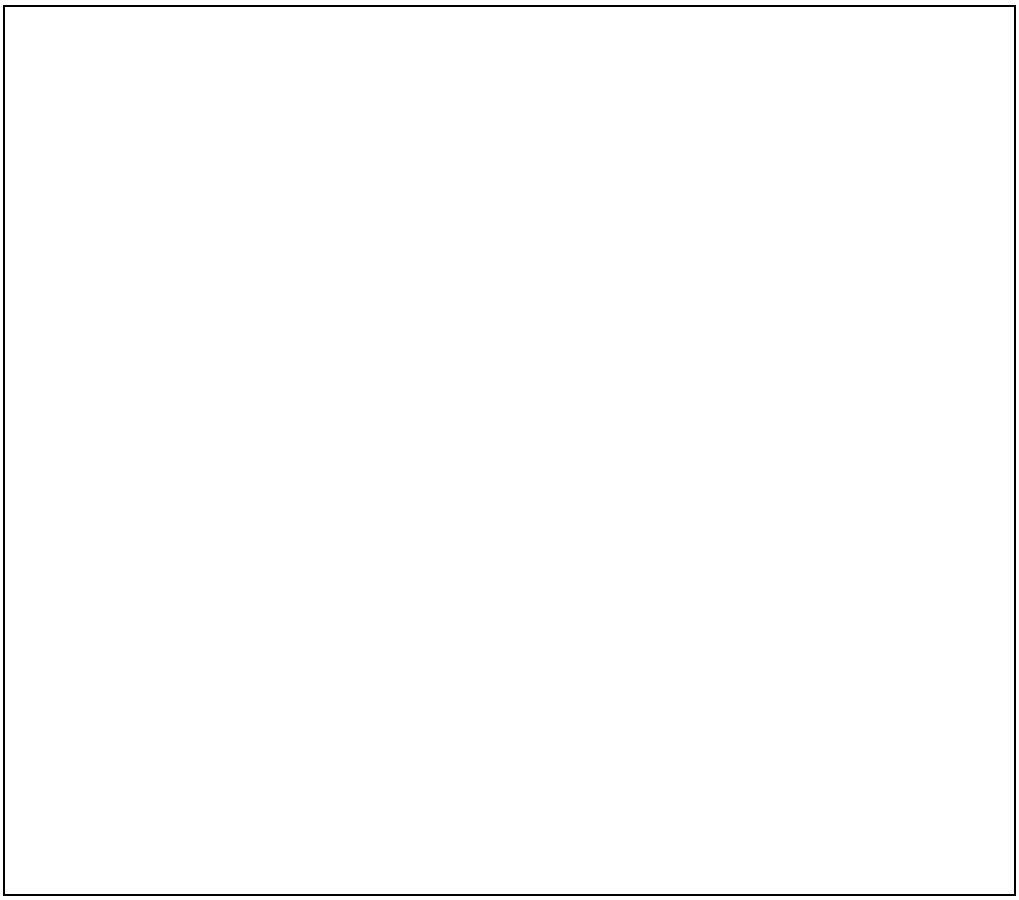

Figure 8: The value tree

6 Object Selection Queries

In this section we discuss the issues connected to the translation of non-recursive object oriented queries into relational queries. The results extend of course also to the translation of the algebraic expressions which are part of fixpoint blocks.

To discuss the problem we restrict to a specific class of queries, that consist in extracting from a class the set of objects that satisfy a given condition. We shall call these *object selection queries*, and we will see in a while that these queries are far more powerful than a relational selection.

In order to define conditions on structured objects, we first introduce, referring to an instance of the OODB, the notion of *value tree* of an object.

- The node of the tree are of four different types: *tuple*, *set*, *oid* and *value*; these represent respectively the constructors in the type of the object, the oid of the object and subojects, and the base values.
- The root of the tree is a node of type *oid*, and has only one child of type *tuple*, which represent the outermost constructor in the object type.
- Each node of type tuple has one child for every attribute in the corresponding constructor; if the attribute is of base type, the node is a leaf and represents the value of the attribute; if the attribute is of object type the node represents the oid of the suboject; if the attribute is of set type the node is of set type.
- Each node of set type has one child for every element in the corrisponding set.
- Each node of oid type is the root of the value tree of the corresponding object.

Note that even in the case of recursive type definition the value of an object is still represented by a tree. In such cases the tree may become infinite, but this is not a problem since, as we shall see in a while, we only need to consider a finite portion of it.

Referring to the object schema in Figure 2, a sample value tree of an object of the class STUDENT is represented in Figure 8.

The purpose of the value tree is to expand the set constructors, explicitly representing all the *combinations* of values in the object. Let us now consider the *atomic* values in the tree, i.e. the base values in the leaves, and the oid in the oid nodes. Each of these atomic values is characterized by a *value path*, defined as the sequence of attributes traversed in the tree by the path from the root to the node. We may then define an *atomic* condition, by associating it to a value path, and say that the condition is satisfied by any atomic value on the value tree that has that path.

For instance referring to the value tree of Figure 8 the atomic condition:

$$\texttt{STUDENT.curriculum.test.year} = \texttt{1990}$$

is satisfied by two different atomic values.

To set up more complex conditions, i.e boolean expressions built on atomic conditions, we need then to allow distinction between atomic values belonging to different elements of

– N_{pc}: number of values common to both attributes.

In fact, according to the defintion of Θ_p and Θ_c:

$$\Theta_p = \frac{N}{N_c} \qquad \Theta_c = \frac{N}{N_p}$$

Assuming a *regular* structure in the graph, the shape of the graph can be desumed from the values of Θ_p and Θ_c. We have two cases:

- $\Theta_p = \Theta_c$. In this case we assume for the graph a rectangular shape. The number of layers h and the number of nodes b can then be expressed as:

$$b = N_p - N_{pc}$$

$$h = 1 + \left\lceil \frac{N}{b\Theta_c} \right\rceil$$

- $\Theta_p \neq \Theta_c$. In this case we assume for the graph a trapezoidal shape, and express accordingly h and the the number b_i of nodes in layer i. For instance, if $\Theta_p < \Theta_c$, and referring to Figure 7:

$$b_0 = N_c - N_{pc}$$

$$b_{h-1} = N_p - N_{pc}$$

$$h = 1 + \left\lceil \log_{\frac{\Theta_c}{\Theta_p}} \frac{b_0}{b_{h-1}} \right\rceil$$

To give an example of how this can lead to practical results, let us refer to the procedural code in Figure 6, and show how to compute an upper bound to the number of accesses performed in the MSDB to the relation `A` at each iteration of the EXTRACT at line `09`.

To get the upper bound we take the node corresponding to the constant in the query (i.e. `a`) in the layer 0 of the graph. This is, of course, the worst case, and requires $h-1$ iterations. During the computation, iteration i passes from a subset $\mathcal{N}_{i-1}$ of the nodes of the layer $i-1$ to the subset of the nodes of layer i that are connected to the nodes in $\mathcal{N}_{i-1}$. Now, if we assume that the nodes are randomly connected, and if we call n_i the cardinality of $\mathcal{N}_i$, it can be proved that:

$$n_i = \begin{cases} 1 & i = 0 \\ \left\lceil b_i(1 - \frac{1}{b_i})^{\lceil n_{i-1}\Theta_p \rceil} \right\rceil & i = 1, \ldots, h-1 \end{cases}$$

Then the number of tuples t_i accessed in the base relation in iteration i is given by:

$$t_i = \lceil \Theta_p n_{i-1} \rceil \qquad i = 1, \ldots, h-1$$

The actual number of I/O accesses needed at each iteration can then be computed by using the well known formula of Yao [21].

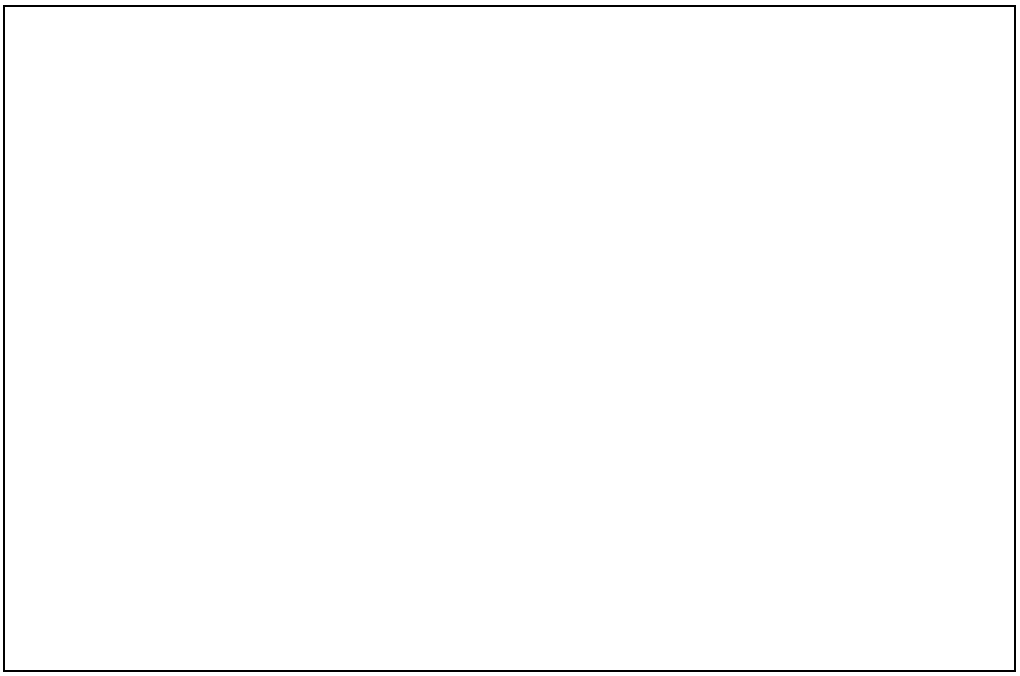

Figure 7: The relation graph

difference operations, that instead are integrated and efficiently performed by the SWAP command in the MMDB.

Normally the optimizer must choose between the remaining two options, i.e. tight and loose coopling, and decide which extensional relations are to be entirely loaded. This is done by estimating and comparing, on one side the I/O cost connected to all the accesses to the mass storage relations needed to perform the semijoins when working in tight coupling mode, and on the other side the cost of loading the whole relations. In the latter case all the blocks of the relation are accessed, but sequentially.

The problem then becomes that of estimating the number of iterations in the seminaive evaluation, and the number of tuples of a given extensional relation that are accessed at each iteration. To see how this can be done, let us consider a binary relation with both the attributes on the same domain, and represent it as a graph where the nodes are the values in the domain, and the arcs represent the tuples of the relation (Figure 7).

Using an approach similar to the one introduced in [6], we make the assumption that the graph is layered, with h layers, and define Θ_p and Θ_c as the average number of arcs respectively entering and leaving a node. In our example the average number of parents and children for every individual. These parameters can then be estimated from the following values, that, in turn, can be easily extracted from the MSDB catalog:

- N : number of tuples in the relation.
- N_p: number of distinct values assumed by the first attribute.
- N_c: number of distinct values assumed by the second attribute.

Figure 6: Procedural code for the semi-naive evaluation of Figure 5

5 Optimizing the Fixed Point Computation

An optimization phase is provided to generate a procedural code that exploits the different options available in the architecture, to perform the computation etheir directly in main memory, or in mass storage through the RDBMS. As we already pointed out, this is especially interesting in the semi-naive evaluation of recursive sets of equations.

Basically there are three different options in performing the semi-naive evaluation:

- *Mass Storage Evaluation.* All the relational operations are performed on the MSDB through the EXECUTE command, and all the intermediate are stored in the MSDB too.
- *Tight Coupling.* Most of the computation takes place in the MMDB. Repeated interaction with the RDBMS takes place at each iteration, when, using the STORE and EXTRACT commands, the differential relations are moved to the MSDB, to perform a semijoin and bring back the result to the MMDB for the next iteration.
- *Loose Coupling.* All the extensional relations involved in the query are initially moved to the MMDB where all the computation takes place.

The first option proves to be valuable only in extreme situations, when both the base relations and the intermediate results have a very large size. Actually the RDBMS is especially designed to handle efficiently such cases. In other situations this choice is not reasonable because of the overhead in the RDBMS, and of the inefficiency of the union and

- The *Communication Primitives* allow to move the relations between the MMDB and the MSDB, to print the result of a query, and to request the RDBMS to execute a subquery:
  - LOAD: moves an extensional relation to the MMDB.
  - STORE: moves a temporary relation from the MMDB to the MSDB.
  - EXTRACT: requests the RDBMS to compute a SQL query and to move the result to the MMDB.
  - EXECUTE: requests the RDBMS to compute a SQL query, and to keep the result in mass memory for further computation.
- The *Control Primitives* allow to express the iterative computation that, as we have seen earlier, is required by the semi-naive evaluation:
  - FIXPOINT: begins a block of procedural code that has to be iterated until some specified integral relations stabilize.
  - ENDFIX: marks the end of a fixpoint loop.
  - HALT: stops the computation.

As an example let us consider the very simple system, that defines the transitive closure of a relation P:

$$A(\$1, \$2) = \sigma_{\$2=a} P(\$1, \$2)$$

$$A(\$1, \$2) = A(\$1, \$2) \cup A(\$1, \$2) \bowtie_{\$1=\$2} P(\$1, \$2)$$

which corresponds to the semi-naive computation:

a1: $A(\$1, \$2) \leftarrow \emptyset$
b1: $\partial A(\$1, \$2) \leftarrow \sigma_{\$2=a} P(\$1, \$2)$
c1: $\partial A(\$1, \$2) \leftarrow \partial A(\$1, \$2) \bowtie_{\$1=\$2} P(\$1, \$2)$
d1: $\partial A(\$1, \$2) \leftarrow \partial A(\$1, \$2) - (A(\$1, \$2) \cap \partial A(\$1, \$2))$
e1: $A(\$1, \$2) \leftarrow A(\$1, \$2) \cup \partial A(\$1, \$2)$

The corresponding procedural code is given in Figure 6. Note that the computation is partially performed in main memory. The interaction with the RDBMS is limited to lines 5 and 9. At every iteration the new tuples in the relation `DELTA-A` are stored in the MSDB (line 8), and then the RDBMS is requested to compute a semijoin and to move the result to the MMDB (line 9). At the end of the iteration the integral relations are updated, and only the new tuples are left in the differential relations (line 10).

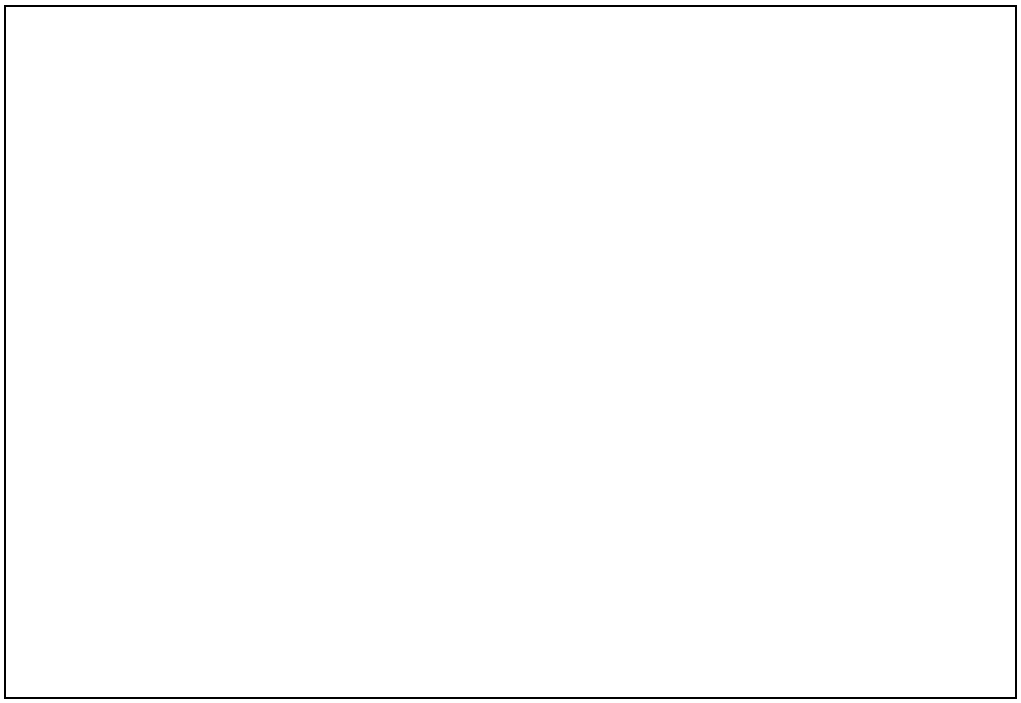

Figure 5: Semi-naive evaluation of a fixpoint block

The procedural code is generated in such a way that indices are always maintained on the join attributes. If there is an index only on one attribute (which is usually the case), the relations are sorted and the join is performed through a merge in linear time. Otherwise a nested loop is used.

More specifically the procedural code provides three classes of primitives:

- The *MMDB Primitives* allow to create the temporary relations and the corresponding access structures, and to execute in main memory some relational operations:
  - CREATE: creates a new relation with a given schema in the MMDB or in the *Mass Storage Database* (MSDB), and sets up indexes on the specified attributes (or groups of attributes). It has to be specified if the relation is an *integral* or a *differential* one, since, as we will see later, different physical representations are used.
  - COPY: performs a copy on the specified relation. This is used to initialize the temporary relations.
  - PROJECT: performs a projection on a temporary relation. The result is stored too in the MMDB.
  - JOIN: performs a join. Both the operands and the result are in the MMDB.
  - SWAP: performs the union-difference operation, needed in the semi-naive evaluation. The operands are an integral relation and the corresponding differential relation. The SWAP adds to the integral relation the new tuples, and deletes from the differential relation the old tuples.

in Section 6. In this section we address the problem of the efficient evaluation of sets of mutually recursive equations.

The fixed point is computed with the *semi-naive algorithm* [6]. This algorithm notably solves by iteration a system of algebraic equations. For every unknown relation in the system two relations are maintained during the computation. The first one, called *differential relation*, contains the tuples produced during the current iteration, and the second one, the *integral relation*, cumulates all the tuples produced by the previous iterations. Only the new tuples are used in the next iteration. The computation terminates when no new tuples have been produced during the last iteration.

More specifically the system of equations generated by the *Relational Mapper* has the form:

$$R_i = E_i^0(B_1, \ldots, B_k) \qquad i = 1, \ldots, m$$

$$R_i = E_i(R_1, \ldots, R_m, B_1, \ldots, B_k) \qquad i = 1, \ldots, m$$

where the R_i are the unknown relations to be computed, $B_1, \ldots, B_k$ the base relations of the canonical database involved in the computation, and the first set of equations define the base values for the R_i. The E_i and E_i^0 are expressions of the relational algebra.

Referring to this system the schema of the semi-naive evaluation is given in Figure 5. The first two group of steps ($a_1, \ldots, a_m$ and $b_1, \ldots, b_m$) are executed only once to initialize the integral and differential relations; the remaining steps are iterated until the integral relations stabilize. More specifically steps ($c_1, \ldots, c_m$) incrementally compute the *new* tuples, using the expressions ∂E_i, that are the *derivatives* of the E_i. The last two groups of steps ($d_1, \ldots, d_m$ and $e_1, \ldots, e_m$) perform a peculiar *union-difference* operation between the integral relations and the corresponding differential relations. That is the new tuples are added to the integral relations and the old tuples are deleted from the differential relations.

Hence the semi-naive evaluation requires to maintain several, rapidly evolving intermediate relations, and the repeated executions of relational operations, some efficiently supported by the RDBMS (like Join and Select), and some not, like the union-difference performed at the end of each iteration. For these reasons the architecture of the prototype system includes the *Main Memory Database* (MMDB), and specific primitives have been defined in the instruction set of the procedural code.

The MMDB provides for the storage of the intermediate relations, their access structures, and for the efficient execution in main memory of the relational operations.

Different physical structures are provided for the integral and differential relations. Both are sorted, but the latter are managed as a simple list in main memory, while the former ones are organized in some kind of *paged* structure that allows, if needed, a partial overflow to mass memory.

As for the index structures in main memory, the problem has been thoroughly investigated in [14] and [15]. According to this analysis, and taking into account that in our application the relations are highly *dynamical*, we adopted the B*-tree, which makes a reasonable tradeoff between the access cost and the memory cost. The size of the blocks was chosen in order to allow, if needed, an easy overflow of the index to mass memory. When this happens, the blocks are gradually paged out and managed with a LRU policy.

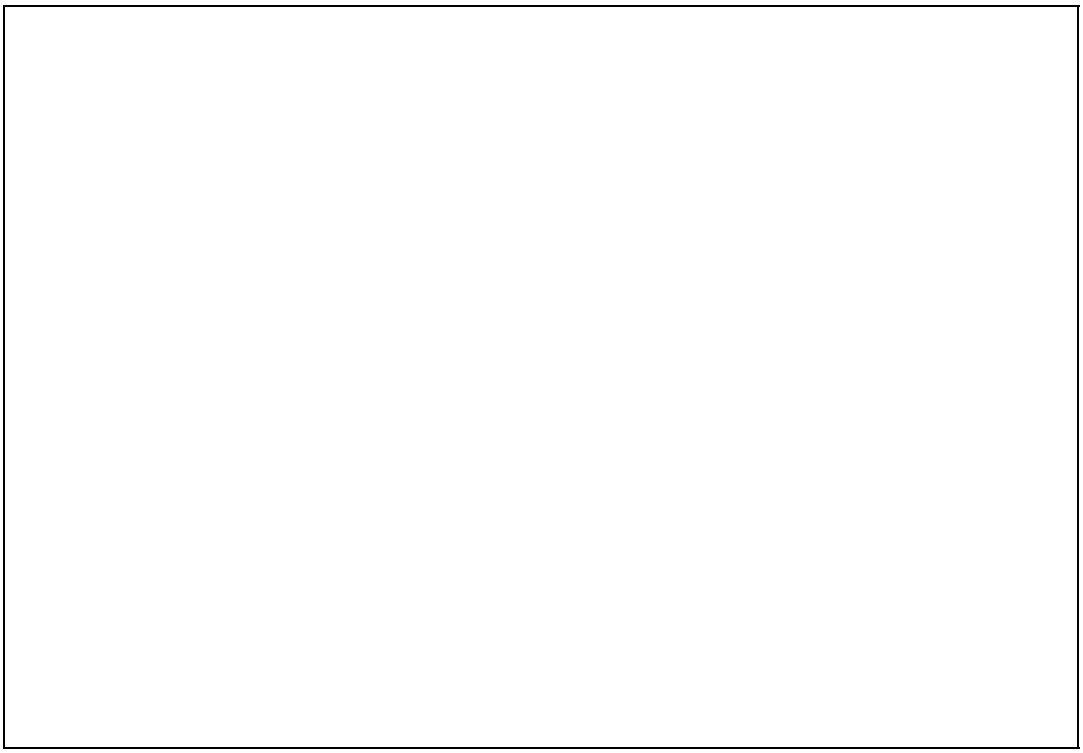

Figure 4: The canonical schema

algebra. More precisely, computing $\mathcal{E}$ transforms the object schema $\mathcal{O}$ in a new schema $\mathcal{O}'$. We then say that a set of relational expressions $(E_1, E_2, \ldots, E_k)$ is equivalent to $\mathcal{E}$ according to the mapping Θ_C, if the canonical schema Σ_C is transformed into a new schema Σ'_C, such that Σ'_C corresponds to $\mathcal{O}'$ in the mapping Θ_C.

It can be easily shown that $E_1, E_2, \ldots, E_k$ can be computed for any $\mathcal{E}$, and are expressions of the relational algebra, with the only addition of the *oid-invention* if restructuring primitives are included in the object algebra, as proposed in [16]. The result of the original query can be obtained by applying the inverse transformation Θ_C^{-1}.

Therefore the set of fixpoint algebraic equations generated by the language interface can be computed on the canonical database, through the repeated evaluation of relational expressions. We address the problem in more detail in the following section.

4 The Procedural Code and the Fixed Point Evaluator

As we have shown in the previous sections, LOGIDATA+ programs are first translated into fixpoint blocks of object algebra equations, and then transformed, according to the canonical mapping, into fixpoint systems of relational algebraic equations. To perform the evaluation the *Code generator* generates a procedural program, written in an internal code, that is later executed by the *Fixed Point Evaluator*.

Non recursive equations are directly translated into relational queries and the computation is performed by the mass memory RDBMS. We will discuss how to optimize this step

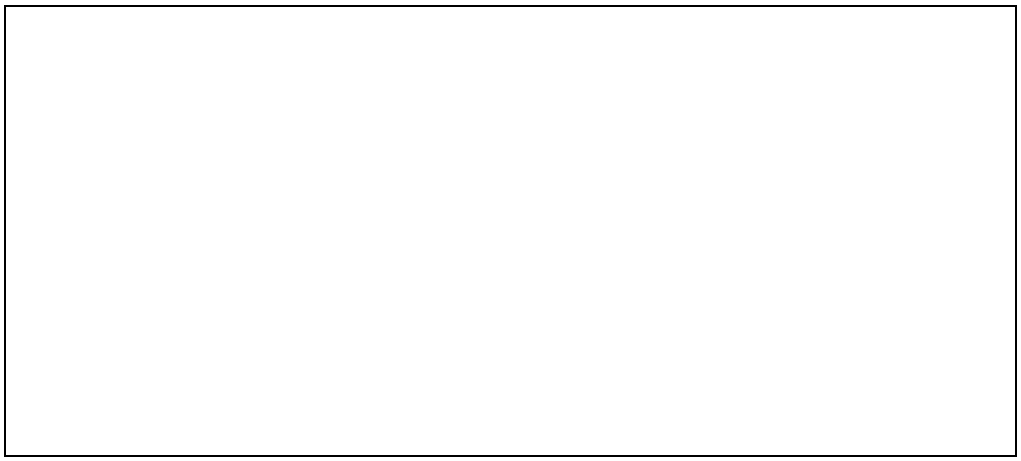

Figure 3: The object graph

schema Σ_C that we call*canonical schema*, defined as follows:

- There is a relation R_i in Σ_C for each node n_i in the object graph. If $(a_1^i, a_2^i, ..., a_k^i)$ is the tuple constructor corresponding to the node, then the relation R_i has schema $(b_0^i, b_1^i, b_2^i, ..., b_k^i)$.
- Attributes in the canonical schema are either of *base type*, corresponding to base types in the object schema, or of *oid type*, or of *link type*.
- If a_h^i is an attribute of base type the corresponding attribute b_h^i in R_i has the same type.
- If n_i is a class node, then R_i is called a *c-relation* and b_0^i has oid type (oid attribute), and is a key to the relation.
- If n_i is a non-class node with a set link from attribute a_h^j of node n_j, then the corresponding relation R_i is called a *s-relation*, and both b_h^j in R_j and b_0^i are of link type (link attributes).
- If a_h^i is an attribute of object type, then b_h^i has oid type.

The link attributes are introduced to represent the unnesting of set constructors. Similarly oid attributes are used to implement the object identity, and to represent the part-of relationship.

According to these definitions we may map the OODB into a pure relational *canonical database*, in which in every c-relation there is a tuple for every object in the corresponding class, and for every s-relation there is a tuple for every element of every set. The canonical schema for the OODB of Figure 2 is reported in Figure 4.

Note that all the information contained in the OODB is preserved in the canonical schema, and that a reverse transformation Θ_C^{-1} can be defined from Σ_C to $\mathcal{O}$.

Once we have defined the mapping Θ_C between the object schema and canonical schema, we may consider the problem of evaluating a query consisting of an expression $\mathcal{E}$ of the object

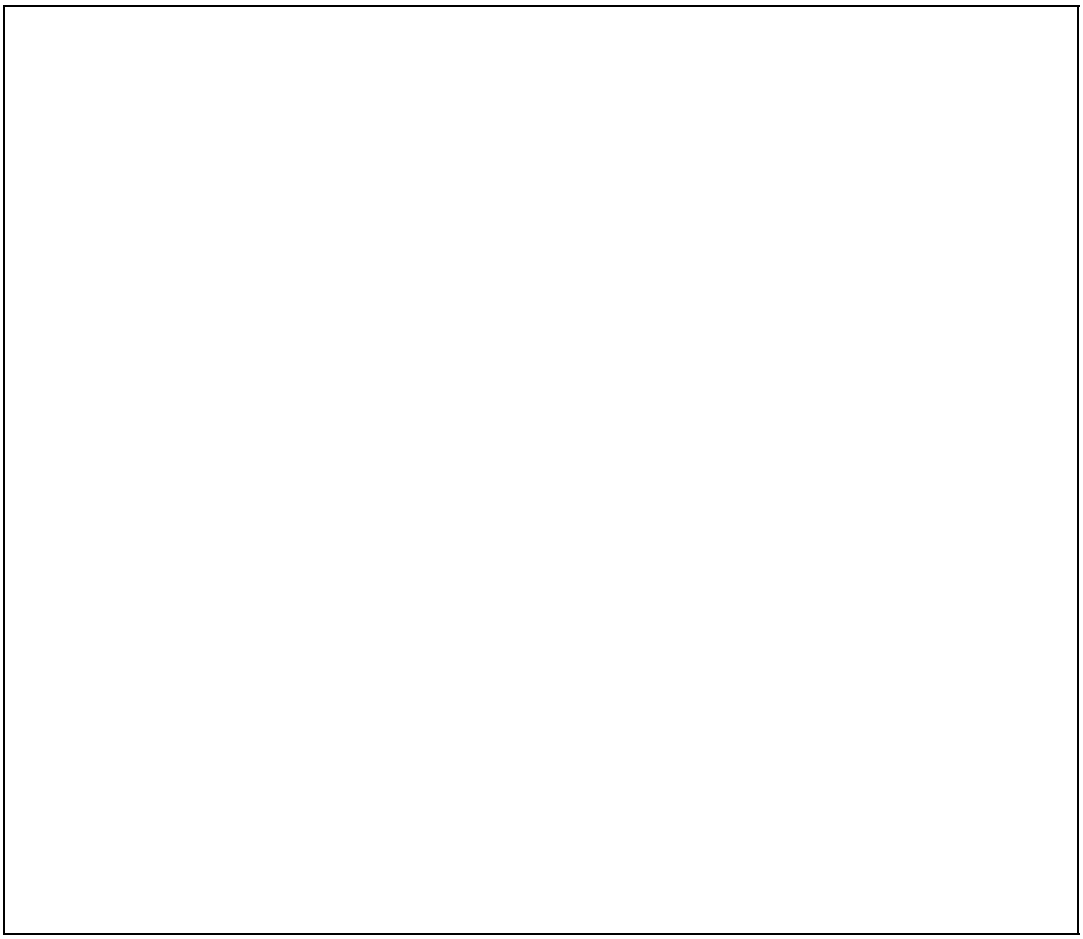

Figure 2: A sample object schema

Note that we require a tuple constructor inside each set constructor. This, as we shall see later, is to have a more direct representation of the schema into the relational model. For similar reasons we assume that tuple constructors cannot be directly nested. Both assumptions do not produce any loss generality.

According to our definitions each class corresponds to an object type. Furthermore, if ω_1 and ω_2 are objects types, and ω_2 appears in the definition of ω_1, we say that ω_2 is *part-of* ω_1.

The structure of the objects in the schema can be effectively represented by the *object graph*. The graph contains a node for every type in the schema that has a tuple constructor at the outermost level. The nodes corresponding to object types are called *class nodes*. The nodes are connected by two different types of arcs. The *set links* and the *oid links*. The former represent a set constructor, and the latter a part-of relationship.

The definition of a sample object schema and the corresponding object graph are presented in Figures 2 and 3. In the figure solid lines represent set-links and dashed lines represent oid-links. Note that the graph only represents the nested structure of the objects, and the part-of relations between the classes. Attributes of base type are not explicitly represented. The graph may be cyclic if recursive type definitions are allowed.

We now address the problem of mapping the object oriented schema with a relational schema. This obtained through a transformation Θ_C, that generates a normalized relational

primitives which, for example, allow the user to require a view V defined during the current working section to become permanent together with any other things in the working environment which C is based on. Of course this management results effective also in the case that many users are simultaneously using the system. The support of a DBMS with built-in locking features on data will result useful in the implementation of these features in the prototype. The hierarchy of environments is only visible to the schema manager, that provides the mapping from any variable/predicate referred in a particular environment to a unique collection of data (class or relation), or to a view within the schema.

3 Mapping Objects into Relations

For sake of simplicity in this and the following sections we shall refer to a data model, which is a simplified version of the LOGIDATA+ model, with *tuple* and *set* constructors and object identity. The model actually contains the relevant aspects of most object oriented models. Therefore the results we present in the paper can be easily extended to a more general framework.

More formally given a finite set of *domains* $\mathbf{D}_1, \ldots, \mathbf{D}_D$ with *domain names* $\mathcal{D}_1, \ldots, \mathcal{D}_D$, a countable domain of *object identifiers* Ω, and a countable set of *attribute names* $\mathcal{A}_1, \mathcal{A}_2, \ldots$, we refer to an *object schema* $\mathcal{O}$ composed by the following elements:

- A finite set of *types names*, or shortly *types*, $\theta_1, \ldots, \theta_\Theta$, and the corresponding *type definitions*.
- A finite set of *classes* $\mathbf{C}_1, \ldots, \mathbf{C}_C$ with names $\mathcal{C}_1, \ldots, \mathcal{C}_C$.

The domain Ω contains the object identifiers associated to the objects, that are unique in all the database. The classes are collections of objects of the same type.

Type definitions allow to build structured types from the basic types associated to the domains, and, for *object types*, to add the identity. A *value set*, i.e. the set of all possible values, is associated to each type:

- A *type* θ is either a *value type* τ or an *object type* ω.
- Each domain name $\mathcal{D}_i$ is a *value type* (called *base type*), and the corresponding value set is $\mathbf{D}_i$.
- If $\theta_1, \ldots, \theta_n$ are types with value sets $\mathbf{V}_1, \ldots, \mathbf{V}_n$, then $\tau = (\mathcal{A}_1 : \theta_1, \ldots, \mathcal{A}_n : \theta_n)$ defines a *tuple type* τ, with value set $\mathbf{V}_\tau = \mathbf{V}_1 \times \ldots \times \mathbf{V}_n$. Round brackets denote the *tuple constructor*.
- If θ is a tuple type and $\mathbf{V}$ is the corresponding value set then $\tau = \{\theta\}$ defines a *set type* τ with value set $\mathbf{V}_\tau = \textsc{part}(\mathbf{V})$, i.e. the powerset of V. Curly brackets denote the *set constructor*.
- If τ is a *tuple* value type with value set $\mathbf{V}$, and $\mathcal{C}$ is a class name, then $\omega = [\mathcal{C}, \tau]$ is an *object type*, and the corresponding value set is $\mathbf{V}_\omega = \Omega \times \mathbf{V}$.

the rules according the strata. Actually the sequencer individuates the strongly connected components of the dependency graph, finding out the inherent sequential structure of the program. In general an overall sequential structure arises when there are portions of the program which are not mutually recursive. As an example, if a program P uses a view-predicate V, in the dependency graph the connected component containing the node V (included by the view handler) has no edge coming from nodes corresponding to intensional predicates defined within the program (note that the program can not redefine some predicate which the view V is defined on). This means that the predicates corresponding to the nodes in that connected component might be instantiated before the rest of the program.

A simple syntactical transformation of the rules converts them into algebraic form, and furthermore all the rules with the same head are grouped to give raise to a single *LOA equation*. Any LOA equation has a form $V = \mathcal{E}$, where V is the name of the variable/predicate being defined, and $\mathcal{E}$ is a *LOA expression* (whose evaluation will be described in the following sections). Moreover V corresponds to a node in the dependency graph, namely to the node with the same name. The other nodes in the dependency graph, by virtue of the view handler, can only be nodes corresponding to extensional predicates.

The stratifier/sequencer is in charge to perform a fragmentation of the program into a sequence of *blocks*, each of them corresponding to a strongly connected component of the dependency graph. Each block consists of either a single LOA equation or a *fixpoint blocks*. Any fixpoint block will be interpreted as a FIXPOINT operator applied to the set of equations it includes. The total ordering of the blocks in the resulting sequence must fulfill the constraint that any equation is not allowed to refer to variables/predicates which are defined in the following blocks.

Note that if the program is stratified, in the dependency graph a negated arc can not occur within a strongly connected component. This means that if a fixpoint block will include rules with negation in the body, the negated predicates either are defined in a previous blocks, or are extensional. The resulting sequence of blocks is what we call a *LOA program*.

The language interface also manages the working and permanent *environments* as described above, allowing a reasonable context to work interactively. In more detail, the schema manager will be able to support a hierarchy of environments, according the following behaviour, which resembles the behaviour of interactive systems in very different contexts. The LOGIDATA schema (and database) has to be considered partitioned in environments. An environment A is *based on* the environment B if the (meta)data contained in A are allowed to refer data in B. The constraint is that this relationship defines a partial order among environments. As an example, suppose that G is a single global environment which is permanent. Then several environment $U_1, U_2, \ldots, U_N$ based on G can be defined. Each U_i may correspond to a user, or a group of users, and contains the customized permanent data, such as views. At any time an interactive user logs into the system, a new empty *working environment* is built. In this manner the side effects of user programs affects only the working environment, and are reflected in the subsequent programs only within the current working session. At any time the user may ask to make permanent a consistent portion of the working environment. This is allowed by the schema manager only if the portion of the environment to be transferred does not contain references to the working environment. Given the definition of a LOGIDATA schema, is not difficult to implement

transformed until it becomes a LOA expression. Of course several rules with the same head will produce a single assignment statement, as shown later.

A user program may refer to *extensional* predicates, corresponding to classes or relationships having associated collections of data in the extensional data base, and to *intentional* predicates which are defined by means of rules. The definition of an intensional predicate may appear either within the program itself, or in the schema (in either the working or the permanent environment). These are called *views*, by analogy with the standard database terminology.

An additional constraint, analogously to the one common in the relational model, is the *safety* of the rules. A variable x which appear as argument in a built-in predicate (such as an arithmetic comparison) must have a bounded domain. If x appears in the same rule as argument of a non negated intensional or extensional predicate, it has a bounded domain. Otherwise in the rule there must be a chain of equality predicates imposing that x is equal to another variable y having a bounded domain.

The main modules in the language interface are the following:

- the LOGIDATA *compiler* takes user programs as input, applies possible rewriting techniques, and generates a set of *rectified* rules;
- the *separator* individuates the part of the program concerning addressing of output and management of the object schema, excluding them from being processed together with the declarative part of the program;
- the *view handler* individuates the predicates used in the program corresponding to views in the schema, and includes the rules defining them inside the program itself: the resulting program refers only to predicates defined therein or to extensional predicates;
- the *stratifier/sequencer* finds outs the stratification of the program according standard techniques and individuates the inherent sequential structure of the program (if any) providing a fragmentation of the program in a sequence of *blocks*;

The external processing of user programs in our prototype is now drawn in more detail.

The compiler, which is not extensively described in this paper, performs a first scanning of the program and applies rewriting techniques to the source LOGIDATA+ program. Furthermore in this first phase the rules are *rectified* [20], in the sense that a renaming of variables with possible substitution of constants within each rule is performed, so that any rule with the same head has exactly the same sequence of variables as arguments. The *separator* module can be considered a part of the compiler and simply separates the part of the program consisting in primitives to direct output and/or handling the environment.

The *view handler* is in charge to include in the program the rules defining the predicates corresponding to the view-predicates. One of the functions of the schema handler is in fact to store such intensional definitions in "compiled" form. Of course this mechanism to describe views by means of rules (with possible negation) is strictly more powerful than a purely algebraic description (i.e. not using operators to handle recursion).

The *stratifier/sequencer*, as remarked before, operates in a standard fashion (as described, for example, in [20, 8]). In particular it builds the dependency graph which will be used to find the stratification of the negation, thus giving a partial ordering among

carry on the fixed point computation, by maintaining in main memory the intermediate results, and requires a further level of optimization, to find the tradeoff between fast computation in main memory and moving data back and forth between MSDB e MMDB.

The paper is organized as follows. In the next Section we present the language interface, and discuss the process of rewriting the rule-based programs and translating them into the object algebra. Next in Section 3 we show how to map the object schema into a relational schema. Sections 4 and 5, present the *lower* part of the architecture, and discuss how to optimize the fixed point computation. Finally Section 6 deals with the translation and the optimization of an important class of object oriented queries.

2 The language interface

This section is devoted to the description of the language interface of our prototype which provides an interactive environment where the user can execute a sequence of *LOGIDATA+ programs*. Each program may simply consist of a query to be answered, or may have side effects on both the schema or the data, or, in general, it might consist of any combination of these basic functions. Side effects affect the behaviour of the system either in the current working section, or permanently, as required by the user, who is given an explicit control over the evolution of the working *environment*, that is the collection of intensional and extensional data which are considered in the evaluation of the programs in the current working section. A remarkable example of an explicit control over the evolution of data and/or metadata is shown in [7].

In the current state of the implementation, a program may include negation in the body with the additional constraint to be stratified. When intensional predicates are to be instantiated a *stratified* semantics is used [10]. This might be later generalized to an *inflationary* semantics [11], which can be applied to a wider class of programs and is strictly more expressive [2, 9]. The choice of an inflationary semantics received further legitimation in [13]. Nevertheless, while dealing with a complex data model (based on object identity), this approach does not guarantee that the evaluation of the program terminates, as shown in [1].

The unit of interaction with the object base is a *LOGIDATA program*, consisting in a set of rules, plus a set of (possibly implicit) directives to direct the output and to handle the evolution of the environment. These include the specification of what among the new classes or relationship defined within the program has to be instantiated or what intensional or extensional data are to be made permanent, thus affecting what is called the *permanent environment*. More precisely, in the middle of a working section, both the LOGIDATA schema and database can be thought to be partitioned into a permanent environment and a working (temporary) one. The latter in general contains references to the permanent environment and is affected by the side effects of the programs. The user can make permanent a consistent set of data and/or metadata, with the constraint that no permanent data/metadata can refer to what is contained in the working environment.

What is initially a set of rules in a program is transformed, through the evaluation process, in a set of LOA equations that eventually will be interpreted as a sequence of assignment statements. In any case any equation (or rule) has a left side (*head*) which is the name of a predicate to be defined, and a right hand side (*body*) that will be successively

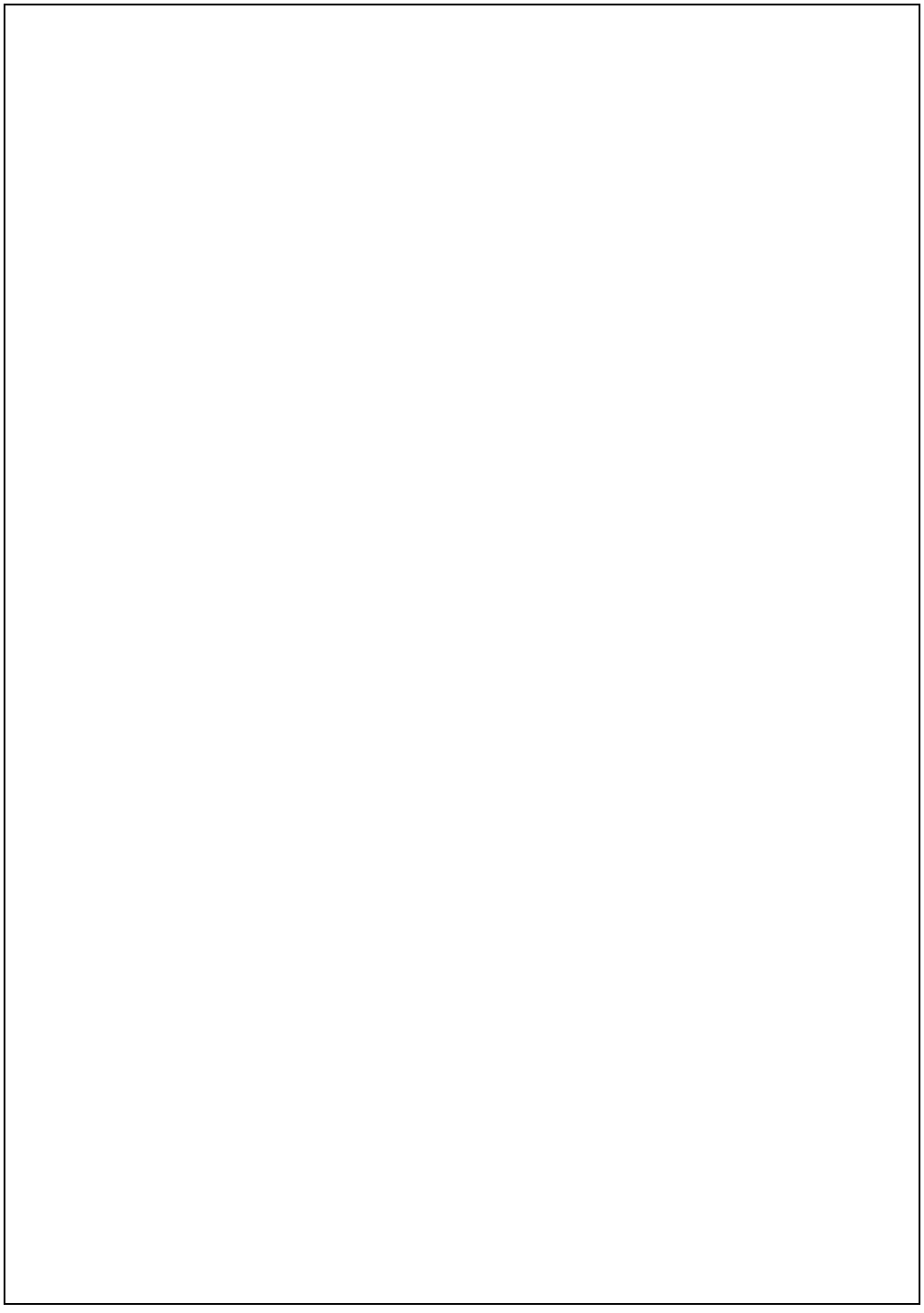

Figure 1: The LOGIDATA+ Prototype Architecture

1 Introduction

This paper presents the architecture of the LOGIDATA+ prototype system, an experimental object oriented database management system, supporting a rule-based language, and developed within the LOGIDATA+ project of the Italian National Research Council (CNR). The primary goal of the prototype was to implement the main features the data model [3] and language, but a good deal of effort has been devoted to study several problems related to the efficient implementation of complex objects, and to analyze various solutions in terms of cost and performance. The development of such a prototype led to deal with several implementation problems. The first issue is the persistent storage of complex objects, studied in terms of access cost, and processing cost of transactions. Other interesting problems are connected to the language interface, like rewriting of rule based programs in a context with structured data types, and dynamic management of temporary environments. Finally a crucial issue is how to perform efficiently the fixed point computations on large mass memory sets of data.

A main implementation choice has been to utilize an existing, commercial, relational DBMS, to implement the persistent storage of the objects, instead of building the prototype on a file system with variable length records, like it has been done for example in O_2 [5].

Our choice has several motivations. First, of course, feasibility and ease of implementation. But, beside this, it can be claimed that such choice is rather reasonable also in terms of performance, at least for a large class of end user applications. These are all those applications that do not require as a basic operation the instantiation of complex objects. For instance, most business applications that already perform reasonably well on relational systems, but that could take great advantage in being designed and maintained in an object oriented framework.

The architecture of the prototype system is presented in Figure 1. The diagram shows several layers in the architecture, which correspond to the various phases of the transformation, and of the evaluation of LOGIDATA+ programs.

The first layer corresponds to the *language interface*. The input language is the LOGIDATA+ language [4], and the output language is LOA (LOGIDATA+ Object Algebra), an algebra for complex objects developed within the project [17]. The programs are first transformed by using rewriting techniques, then stratified and translated into a sequence of blocks, each formed by a system of LOA equations. Evaluating a program requires to compute the fixed point of these blocks.

The second layer is composed by the *schema manager* and the *relational mapper*. The first module generates and manages the mapping between the object oriented schema and the relational schema, and the relational mapper translates the LOA equations into the relational algebra, according to that mapping. This requires a first level of optimization.

The last layer corresponds to the fixpoint evaluation of the systems of relational algebraic equations. This is performed by the *procedural code generator* that produces a program written in a special internal code, and by the *fixed point evaluator* which finally executes the procedural program.

The procedural code has primitives to perform relational operations both on relations stored in the *Mass Storage DataBase* (MSDB) managed by the RDBMS, and on temporary structures stored in the *Main Memory Database* (MMDB). This gives the way to efficiently

The LOGIDATA+ Prototype System*

Umberto Nanni† Silvio Salza‡ Mario Terranova‡

TR-92-007

February 1992

Abstract

In this paper we present a prototype system developed within *LOGIDATA+*, a national project funded by the Italian National Research Council (CNR). The prototype supports a rule based language on a data model with structured data types, object identity and sharing. The system has an interactive user interface, with a unit of interaction consisting of a *LOGIDATA+ program*, used to extract information from the knowledge base and/or modify the schema. A program consists of a set of rules, and of additional directives to handle the data output and/or the updates to the schema. The intermediate results and the updates to the schema, affect a temporary *working environment* connected to the working session, but can also be saved in a (user or global) *permanent environment*. The system uses *LOA* (LOGIDATA+ Object Algebra) as an intermediate internal language. User programs are compiled, with a set of transformations including rewriting and stratification, and then translated into LOA programs, i.e. sequences of fixpoint systems of algebraic equations. The object oriented schema is mapped into a relational schema, and the database is actually managed by a relational DBMS, that provides the basic support for the permanent storage of data as well as for concurrency control and recovery. The object algebra expressions can then be mapped into relational algebra expressions, thus relying on the efficiency of the RDBMS for the access to mass storage structures and the efficient execution of set-oriented operations. Moreover a *main memory database* has been included in the architecture, to improve the performance in the evaluation of the fixpoint systems, by keeping in main memory the intermediate results.

---

*This work is partially supported by the project "Sistemi Informatici e Calcolo Parallelo" of the Italian National Research Council (CNR).

†Dipartimento di Matematica Pura e Applicata, University of L'Aquila, via Vetoio, Coppito - 67010 - L'Aquila, Italy. Currently visiting the International Computer Science Institute, 1947 Center St., Berkeley, CA 94704, U.S.A.

‡Istituto di Analisi dei Sistemi ed Informatica del CNR, Viale Manzoni 30, 00185, Rome, Italy.