An Object Based Algebra for Parallel Query Processing and Optimization

Vijay M. Sarathy
Indiana University
Computer Science Department
Bloomington, Indiana 47405-4101, USA
e-mail: vijay@cs.indiana.edu

Lawrence V. Saxton
University of Regina
Department of Computer Science
Regina, Saskatchewan S4S 0A2, Canada
e-mail: saxton@cs.uregina.ca

Dirk Van Gucht
Indiana University
Computer Science Department
Bloomington, Indiana 47405-4101, USA
e-mail: vgucht@cs.indiana.edu

Abstract

The Tarski algebra provides an algebraic foundation for object-based query languages. This is demonstrated by showing how queries expressed in a graph-oriented query language (based on the functional data model) can be translated into the Tarski algebra. The graphical representation of queries in combination with the Tarski algebra is a convenient mechanism to study optimization in the context of object based query languages. We then propose extensions to the Tarski algebra that facilitate parallel query processing and address the issue of parallel query optimization in this algebraic framework. We also show how our framework helps in the study of non-monotonic query optimization.

1 Introduction

Over the last decade, a variety of new database models [10] have been introduced to deal with data applications involving *objects* with a complex external and/or internal structure. These database models can be classified into three main categories: the *complex object* models, the *function-based object* models, and *hybrids* of these.

Complex object models [1, 2, 32, 35] (also called value-based models) extend the relational model by allowing, besides flat relations, complex object types obtained as a sequence of type constructions, such as tuple, (finite) set, (finite) list, array, and pointer formation. On the other hand, function-based object models [14, 21, 28, 47] view a database as a graph of objects organized in classes, where the links between objects express single-valued and multi-valued functions (relationships) between objects.

In the area of query languages for complex object databases, researchers have successfully extended the well-known relational query languages. This has resulted in calculus, algebraic, rule-based, and SQL-like query languages for complex object databases [1, 2, 3, 31, 35, 53]. In contrast, the study of query languages for function-based object databases is less developed. In this area, the most progress has been made in the specification of SQL and rule-based languages [5, 9, 28, 39, 47].

Although proposals for function-based object algebras have appeared in the literature, such algebras strongly resemble the algebras defined for complex object databases [13, 18, 46, 53]. Defining algebras in this fashion deviates from the basic philosophy of the pure function-based approach as the level of abstraction at which these algebras operate is several layers higher than the *graph-oriented* nature of the function-based approach.

Given this situation, we propose a *simple* algebra, the *Tarski algebra*, that is appropriate to support object-based query languages [44]. Unlike previously considered algebras, the Tarski algebra operates on graphs (interpreted as conceptual binary relations[1]) rather than on objects of complex types. In that respect, the Tarski algebra is at the level of abstraction of function-based object models and is thus more natural and effective than other algebras for such database models. To demonstrate this, we show how to translate queries specified in a graph-oriented query language into the Tarski algebra. A graph-oriented query specification language was chosen because graphs are the natural representation of function-based object databases [20, 21, 30].

We then propose extensions to the basic Tarski algebra that facilitate parallel query execution and optimization in an algebraic framework. Parallelism in the context of database programming languages has received a renewed interest in recent times [27, 42]. Other related work has focussed on the expressiveness and query optimization for such languages [12, 19]. These approaches have been based on the functional programming paradigm because it naturally lends itself to the divide and conquer principle and consequently parallelism can be exploited. However, these approaches rely on a host of complicated type constructors which make their languages quickly and unnecessarily complex, which subsequently leads to difficulties in their optimization. Our techniques, on the other hand, enable us to use the divide and conquer principle for parallelism, by *partitioning* the data, within a simple underlying conceptual binary relation framework. This requires no additional type constructors and is much simpler.

Thus, the Extended Tarski algebra can be viewed not only as an algebraic foundation for object based query languages, but also as a language to specify parallelism and optimization techniques within the language itself.

In Sections 2 and 3, we discuss the techniques which establish that the Tarski algebra is an algebraic foundation for object based query languages. Section 4 discusses how the Tarski alge-

[1]It is important to stress here that these binary relations are purely conceptual. In fact, the Tarski algebra is shown to support physical data independence in Section 4.

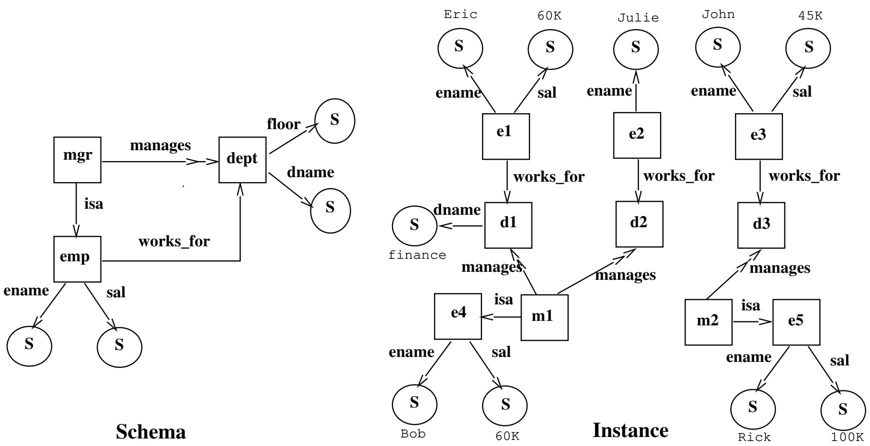

Figure 1: The object base schema of a departments database, and an instance over the departments schema.

bra and our translation algorithm are independent of the underlying physical storage structure. Section 5 discusses the extensions to the basic Tarski algebra that facilitate parallel execution of queries. Section 6 discusses several query optimization issues in this context. We finally conclude in Section 7 with a summary of our results and some future research directions.

2 Graph Representation of Queries and the Tarski Algebra

The graph-oriented representation of a function-based object database views its schema and instance as a labeled directed graph. The labeled nodes in the schema (instance) correspond to the object *classes* (*objects*) in the database, and the labeled edges correspond to the *functions* or relationships between these classes (objects) [20, 21, 41].

Consider the graph in Figure 1. The left graph represents the *object base schema* of a departments database, and the right graph represents an *instance* over the departments schema. The rectangular nodes represent *structured object classes*, whereas the circular nodes represent *basic object classes*. For example, *dept* represents a department, *emp* represents an employee, *mgr* represents a manager[2], and S an atomic string. Notice that in the object base instance S nodes have associated string values. There are two possible function types, *single-valued* ($\rightarrow$) function types, and *multivalued* ($\rightarrow\!\rightarrow$) function types. For example, the multivalued function *manages* indicates that a manager can manage several departments. The other edges indicate single valued functions.

Given this graph representation of a database, it is natural to specify queries in the form of *query graphs*. Consider the query graph in Figure 2 which specifies the following query: Find all manager-dept pairs where the manager earns the same salary as some employee in the department, and print the names of the manager and the department. The subgraph in solid lines represents the search conditions required to solve the query, while the subgraph in dotted lines represents the final attributes to be reported in the answer to the query.

For the actual computation of the answer to the query, we consider only the *subset* of the query graph that consists of the nodes in solid lines. Since we are interested in the attributes of the *mgr* and *dept* nodes as output, we designate them as *selected nodes*[3] (pointed to by bold arrows) as shown in Figure 2.

[2]A manager is an employee and inherits all the attributes of an employee.

[3]This enables us to obtain the attributes of the output easily after the query has been processed.

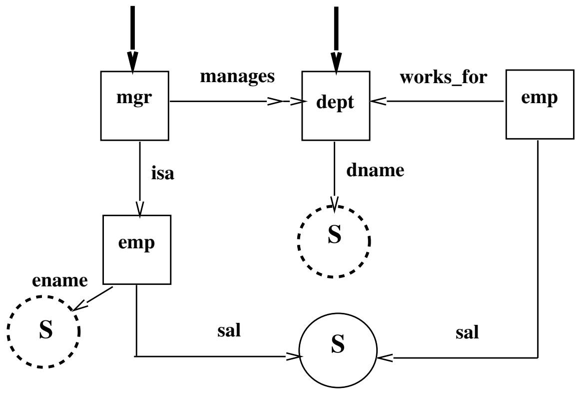

Figure 2: A query graph over the departments object base specifying manager-dept pairs where the manager earns the same salary as some employee in the department. The nodes in dotted lines represent *result* or output nodes, the bold arrows represent *selected* nodes.

To solve such queries, it is best to resort to a mathematical framework that is faithful to the function-based approach. Since the object base schemas and instances we are working with are labeled graphs, a natural (and easy) way to conceptualize them is as a collection of binary relations [40]. Therefore, solving such queries can be accomplished by 1) encoding the object base as a collection of conceptual binary relations as shown in Figure 4 corresponding to the classes and functions in the graph instance, and 2) resorting to an algebra over binary relations introduced by Tarski[4], and augmented by Gyssens et.al and Sarathy et.al [22, 44].

The kernel of this algebra consists of four well-known operators on binary relations: union $(r \cup s)$, composition $(r \odot s)$[5], inverse (r^{-1})[6], and (finite) complementation $(\overline{r})$[7]. This algebra is *not* powerful enough to express all reasonable queries. To overcome this weakness, the basic Tarski algebra is augmented with two *constant selection* operators and certain *object-id creation* operators. The selection operators are very simple and select pairs from a relation whose left/right component is equal to a constant. For example, $\sigma^{l}_{'cindy'}(r)$ selects from relation r those pairs whose left component is equal to the constant '*cindy*'. Similarly, $\sigma^{r}_{'cindy'}(r)$ selects pairs based on the right component of r. The object-id creation[8] operators are more fundamental. They allow the creation of object identifiers for ordered-pair and finite-set objects. There are two *ordered-pair oid creation* operators, (the *left-* and *right oid creation* operators), and one *finite-set oid creation*[9] operator.

In Figure 3, we show the result of *left* and *right-oid creation* on a relation r, denoted $r^{\triangleleft}$ and $r^{\triangleright}$ respectively. Notice how each ordered pair in r has received a separate oid in the left and right oid creation operation. The left-value (right-value) of that oid is found in $r^{\triangleleft}$ $(r^{\triangleright})$. Thus, ordered-pair oid creation is a technique to create object-identifiers for ordered-pairs. If we subsequently use these oids in other ordered-pairs, we gain the capability to represent tuples of arbitrary arity and set-nesting levels. For instance, if 1 is the oid for the pair (a, b), then the ordered pair $(1, c)$ is a representation of the triple (a, b, c), and if 2 is the oid for the set $\{c, d\}$ (with finite-set oid creation),

[4] Tarski's work was a synthesis of the work of Boole, DeMorgan, Peirce, and Schroder. For some recent surveys of this work see [37].

[5] The set of ordered pairs (u, w) such that there exists a value v, such that the ordered pair $(u, v) \in r$ and $(v, w) \in s$.

[6] The set of ordered pairs (v, u) such that the ordered pair $(u, v) \in r$.

[7] Finite complementation is the complementation with respect to the active domain of a relation, i.e. the actual values involved in a relation, rather than the set of all values. This avoids infinite relations.

[8] Oid-creation can be expensive and will be used only when absolutely necessary. Later, we will show how the extensions to the Tarski Algebra can eliminate oid-creation.

[9] Due to space limitations, we refer the reader to [44] for details on the finite-set oid creation operator.

r

a	b
a	c
b	c
c	d
c	c

$r^{\lhd}$

1	a
2	a
3	b
4	c
5	c

$r^{\rhd}$

1	b
2	c
3	c
4	d
5	c

Figure 3: Example of left and right oid creation on a relation r.

Tarski Instance corresponding to Structured Objects

emp

emp_l	emp_r
e1	e1
e2	e2
e3	e3
e4	e4
e5	e5

dept

$dept_l$	$dept_r$
d1	d1
d2	d2
d3	d3

mgr

mgr_l	mgr_r
m1	m1
m2	m2

Tarski Instance corresponding to Functions

ename

emp	ename
e1	Eric
e2	Julie
e3	John
e4	Bob
e5	Rick

sal

emp	sal
e1	60K
e3	45K
e4	60K
e5	100K

works_for

emp	dept
e1	d1
e2	d2
e3	d3

manages

mgr	dept
m1	d1
m1	d2
m2	d3

isa

mgr	emp
m1	e4
m2	e5

Figure 4: The conceptual Tarski instance of the persons object base.

then $(1,2)$ represents the complex object $(a, b, \{c, d\})$.

From these examples, it can be seen that the Tarski Algebra is very different from other algebras for object-based languages in that the structured objects are not explicitly represented, but are specified by oids. In other words, complex objects are not explicitly materialized but are specified referentially. Although both approaches are equivalent, the Tarski algebra is more uniform since it is not overloaded with operations to deal with the complexity of differently typed structured objects, unlike most other algebras for function-based object databases [2, 53]. More importantly, we feel that addressing query optimization is impeded in a complex and strongly typed language [11, 53]. Our approach, enables specification of simple and general query optimization rules in a simple algebraic framework.

There are also some *derived* Tarski operators that will prove useful in subsequent sections. They are r^{π}, r^{π_l}, r^{π_r}, r^{τ}, and $r \cap s$[10], where

- $r^{\pi} = (r^{\lhd})^{-1} \odot r^{\lhd} \cup (r^{\rhd})^{-1} \odot r^{\rhd}$. This is the set of pairs (v, v), where v is an atomic value in r, and is called the *identity projection* operator.

- $r^{\pi_l} = (r^{\lhd})^{-1} \odot r^{\lhd}$. This is the set of pairs (v, v), where v is an atomic value in the *left* component of r, and is called the *left projection* operator.

- $r^{\pi_r} = (r^{\rhd})^{-1} \odot r^{\rhd}$. This is the set of pairs (v, v), where v is an atomic value in the *right* component of r, and is called the *right projection* operator.

- $r^{\tau} = (r^{\lhd} \odot (r^{\lhd})^{-1})^{\pi}$, or $(r^{\rhd} \odot (r^{\rhd})^{-1})^{\pi}$. This is the set of pairs (t, t), where t is an oid of an ordered pair of r, and is called the *ordered-pair oids* operator.

[10] This is the regular intersection of two relations and can be specified in the Tarski algebra as shown in [22].

3 The Query Graph Translation Algorithm

We now turn to translating query graphs into equivalent Tarski algebra expressions. We first need to encode an object base instance as a set of conceptual binary relations, which can be manipulated by the Tarski algebra. For each object class, we create a binary relation consisting of identical pairs for each instance of that class[11] in the instance. For each function (edge) type in the instance, we create a binary relation consisting of pairs corresponding to the instances of the two classes that the function (edge) relates. This is illustrated in Figure 4.

The key problem in the query graph translation process is to derive an algebraic expression for the search conditions specified in the *subset* query graph of Figure 2. The translation of a query graph into a Tarski expression uses the following graph reduction techniques[12].

1. The *multiple*[13] *edge reduction* technique, in which multiple function edges between a pair of objects in the query graph are replaced by a single function edge. Suppose that there are multiple edges connecting objects M and N (of any type) in the query graph. Such edges can be collected into two sets, set F of edges leaving M and arriving at N, and set T of edges leaving N and arriving at M. The technique then computes a Tarski algebra expression $\Psi_{M,N} \equiv \cap_{f \in F} f \bigcap \cap_{t \in T} t^{-1}$. The new query graph is then derived by removing from the original subset query graph all the edges between M and N and replacing them by the single new edge $(M, \Psi_{M,N}, N)$. Clearly, a multiple edge reduction reduces the query graph by at least one edge.

2. The *node combination* technique, in which two objects connected by a function edge in the query graph are combined into one composite object. This technique is more complex and involves oid-creation. It is applied whenever there are edges remaining in the query graph, and none of the other reduction techniques can be applied. Suppose (M, a, N) (M, N are nodes of any type) is such an edge . Consider the relation a corresponding to this edge. Perform a left-oid-creation and right-oid-creation operation on a, (This results in the relations $a^{\triangleleft}$ and $a^{\triangleright}$, respectively.) and construct the relation containing the new oids introduced by these operations. This can be done by the Tarski expression a^{τ}. Intuitively, each element in a^{τ} corresponds *uniquely* to a pair in the relation a. Now add a new node (object) O to the query graph with label a^{τ}.

   Now, there are four cases depending on whether there are edges entering/leaving node M/N.

   - For each edge (M, b, P) in the query graph other than a (i.e. for each edge leaving M), add an edge with label $a^{\triangleleft} \odot b$ from the new node O to node P in the query graph.
   - For each edge (P, c, M) in the query graph (i.e. for each edge arriving at M), add an edge with label $a^{\triangleleft} \odot c^{-1}$ from the new node O to node P in the query graph.
   - For each edge (N, d, P) in the query graph (i.e. for each edge leaving N), add an edge with label $a^{\triangleright} \odot d$ from the new node O to node P in the query graph.
   - For each edge (P, e, N) in the query graph other than a (i.e. for each edge arriving at N), add an edge with label $a^{\triangleright} \odot e^{-1}$ from the new node O to node P in the query graph.

   Next delete nodes M and N (and also all their incident edges) from the query graph. It should be clear that this new query graph has one fewer node and one fewer edge, i.e., node combination is a graph reduction operation.

[11] We encode a class as a relation of identity pairs to maintain a uniform binary relation approach.

[12] Two other graph reduction techniques (*graph constraining* and *chain reduction*) are illustrated in [44, 45] for optimization in the translation process.

[13] Here multiple is not to be confused with multi-valued, and just means "more than one".

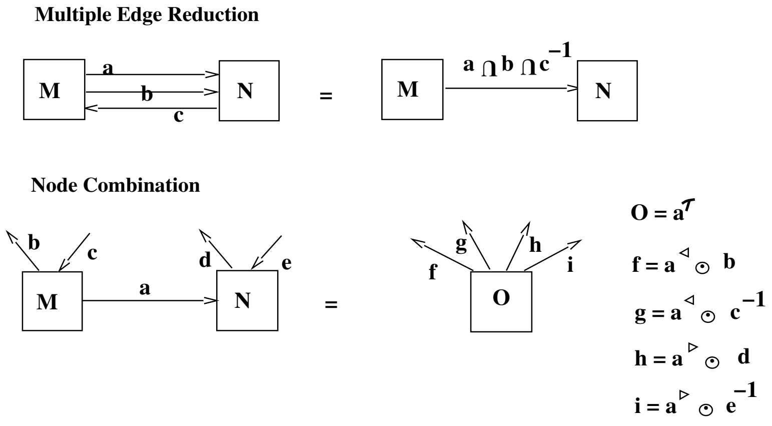

Figure 5: The main techniques used in the query graph translation algorithm.

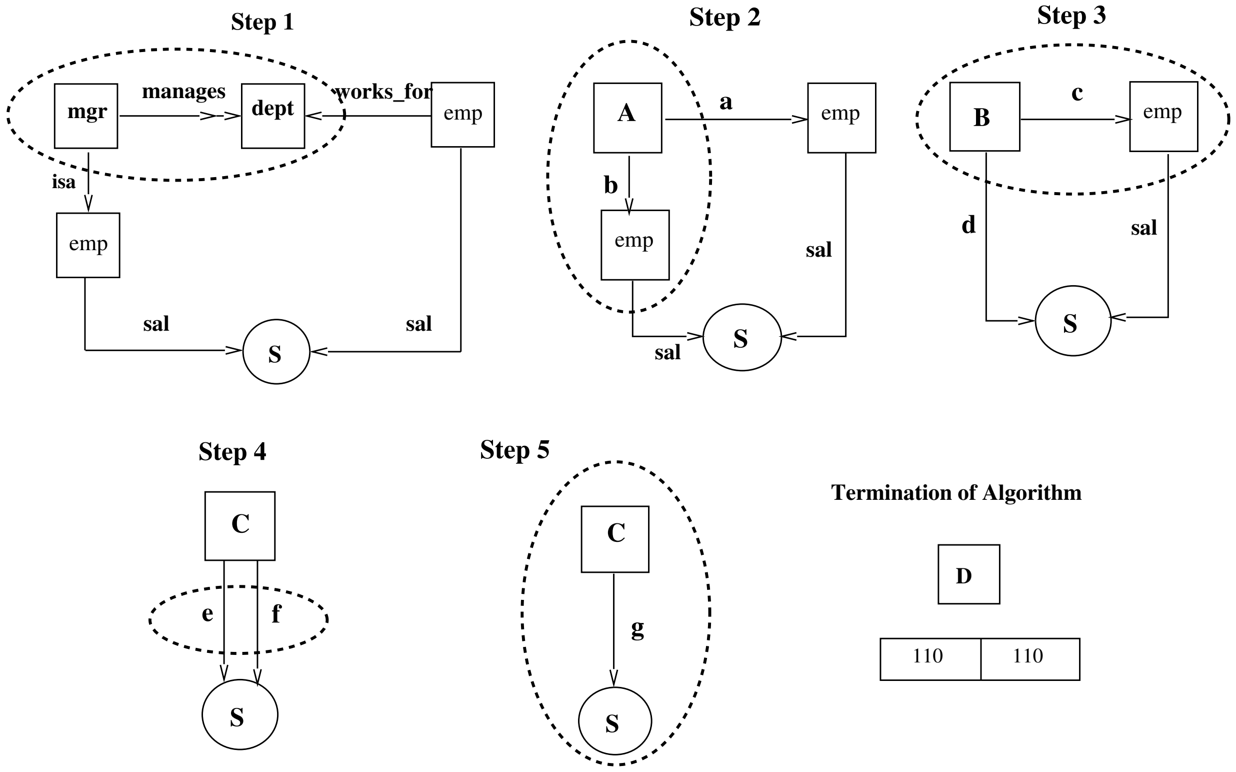

Figure 6: Step 1: A node combination of the nodes *mgr* and *dept* results in a new node A, where A stands for $manages^{\tau}$, a stands for $manages^{\triangleright} \odot works\_for^{-1}$, and b stands for $manages^{\triangleleft} \odot isa$. Step 2: A node combination of the nodes A and *emp* results in a new node B, where B stands for b^{τ}, c stands for $b^{\triangleleft} \odot a$, and d stands for $b^{\triangleright} \odot sal$. Step 3: A node combination of the nodes B and *emp* results in a new node C, where C stands for c^{τ}, e stands for $c^{\triangleleft} \odot d$, and f stands for $c^{\triangleright} \odot sal$. Step 4: An edge reduction involving the edges e and f results in a new edge g, where g stands for $e \cap f$. Step 5: A node combination of the nodes C and S results in a new node D, where D stands for g^{τ}. Termination: Final result of the query graph translation algorithm.

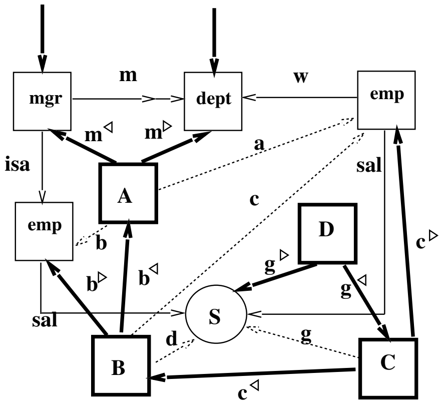

Figure 7: Trace of the node combinations performed on the query graph in Figure 2. (m stands for *manages*, and w stands for *works_for*)

The techniques of edge reduction and node combination are summarized in Figure 5. By repeatedly applying these graph reduction techniques, it is possible to reduce the subset query graph to a single node. This node (and its associated Tarski algebra expression) encodes the algebraic solution to the subset query graph. In Figure 6, we show the various steps (graph-reductions) involved in solving the query of Figure 2.

Once we have applied the translation algorithm to the subset query graph in Figure 2, we have to compute the result of the query. In Figure 7, we show the *trace* of successive node combinations that led to the binary relation D. The bold nodes and edges indicate the successive node combinations. If we expand (conceptually) what the oids in the relation D mean, it is actually a quintuple[14]. For each object that appeared originally in the subset query graph there is a *unique* path from the object D to that object along the *bold* nodes and edges. Once these paths are known, it is necessary to determine the *projections* of the conceptual quintuple relation onto the selected nodes (mgr and $dept$). To do so, the projection paths to mgr and $dept$ will be used. The projections are computed as the composition of the edge labels appearing on the projection paths. Thus, to obtain the *projection* on mgr, we have the expression $g^{\triangleleft} \odot c^{\triangleleft} \odot b^{\triangleleft} \odot m^{\triangleleft}$, and on *dept*, we have the expression $g^{\triangleleft} \odot c^{\triangleleft} \odot b^{\triangleleft} \odot m^{\triangleright}$.

Once we have these, computing the required attributes by appropriate compositions is simple and is outlined in [44, 45]. This establishes that the Tarski algebra can be used as an algebraic foundation for object based query languages.

4 Physical Data Independence of the Tarski Algebra

The fact that the Tarski algebra operates on binary relations might suggest that the underlying physical storage organization of an object base needs to consist of binary relations. In this section, we show how the Tarski algebra can be used with several proposed physical organizations[15] for manipulating complex objects, and show how the Tarski algebra is independent of the underlying physical storage structure. The representation of the object base as a collection of binary relations is purely conceptual.

The *decomposed storage* model (DSM) [52] is a storage organization based on binary relations in which an n-ary relation is broken into several binary relations consisting of surrogate-attribute

[14] The quintuple captures the five objects in the subset query graph of Figure 2.

[15] Various techniques and their relative merits/demerits for implementing complex objects are discussed in [25, 52].

pairs. Clearly, the DSM lends itself naturally as a physical storage model to support the Tarski algebra. The DSM is best suited for join queries that do not report more than a few attributes in the final result.

However, queries that need to report a large number of related attributes of an object, are better supported by the *normalized storage* model (NSM), which is the standard storage organization for relational databases, in which all the attributes of a structured object are clustered together in one n-ary relation. All the operators in the Tarski algebra can be easily implemented in this setting since the n-ary relations store binary relations implicitly. All that is necessary is a mapping between the conceptual view of the database as a collection of binary relations and the actual normalized physical organization.

The DSM and the NSM are two special cases of the more general *partial decomposed storage* model (P-DSM), in which, the partitioning of conceptual versus physical attribute representations is mixed. In this model attributes are clustered together in such a manner so as to take maximal advantage of both the NSM and the DSM. Again, a similar argument as made in the case of the NSM clearly shows that the Tarski algebra can easily be implemented over the P-DSM.

5 The Extended Tarski Algebra

5.1 Problems With The Basic Tarski Algebra

As shown in the previous section, the basic Tarski algebra offers a simple algebraic foundation for object based query languages. However, there are some limitations with the basic Tarski algebra, which we outline in this section.

The Tarski algebra operates on binary relations, and one possible storage structure to support such an algebra is the **D**ecomposed **S**torage **M**odel (DSM) as discussed in Section 4. As argued in [17, 29], the principal motivation for the use of the DSM is the advantages it offers for query processing in parallel. The advantages are due to the vertical fragmentation of large relations into a set of smaller binary relations. Given this, the basic Tarski algebra can be thought of as an algebra suitable for parallel query processing based on a *vertical fragmentation* of the database. However, vertical fragmentation is just one dimension of data fragmentation which can lead to effective parallel query processing. Furthermore, the resulting parallelism is entirely at the physical level and the user has no control over the degree of parallelism at the query language level. The other, and more frequently studied data fragmentation beneficial for parallel query processing is the *horizontal fragmentation* of data [42]. As such, the basic Tarski algebra has no mechanisms to specify such horizontal data fragmentation. In this section, we extend the basic Tarski algebra to enable horizontal data fragmentation (partitioning) and specification of parallelism within the algebra itself. The result is a simple and elegant algebraic foundation for parallel object based query processing. These extensions also enable us to algebratize many queries with universal quantification elegantly, which is a significant feature of our algebra. The extensions also enable us to specify different algorithms for a single operation within the algebra itself.

Although the basic Tarski algebra is simple, the *oid-creation* operator was critical in the translation of object based queries. For many queries, oid-creation can be avoided by using the *chain reduction* technique as shown in [45]. However, for solving many complicated queries such as that shown in Figure 8, oid-creation becomes an absolute necessity. Oid-creation could potentially become an expensive operation. Our extensions to the basic Tarski algebra also enable us to eliminate the necessity for oid-creation.

The basic Tarski algebra does not provide any mechanisms for dealing with queries involving aggregate functions. We have extended the algebra to include some important aggregate functions.

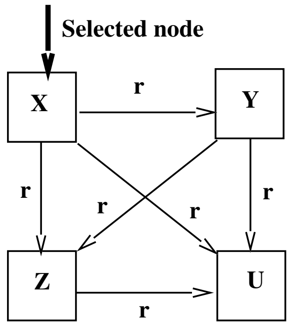

Figure 8: Query graph that absolutely requires oid-creation for a solution in the basic Tarski algebra.

The aggregate functions also enable us to specify optimization techniques within the algebra itself. Also, we have added certain generalized *bonding* operators in addition to the simple composition operator. All these changes make the algebra easier to work with and makes query formulation more intuitive as will be shown by examples later.

5.2 Extensions for Parallelism

In this section, we introduce the extensions to the basic Tarski algebra that enable specification of parallelism within the algebra. The two main concepts in this context are:

1. *Horizontal partitioning* of relations to decompose a query into several smaller subqueries, and
2. *Accumulators* that "accumulate" the results of the subqueries to provide the final result of the query.

5.2.1 Horizontal Partitioning of Relations

The horizontal partitioning of relations is motivated by the *divide and conquer* principle, to break down a problem into several independent subproblems. The independent data partitioning specifies *independent fan-out* parallelism [42] since the independent subproblems can be solved independently in parallel. The data partitioning is expressed in the algebra itself giving the user explicit control over the degree of parallelism. This is in contrast to a DSM approach which uses vertical partitioning and achieves parallelism only at the physical level giving the user no control over the parallelism [17, 29].

The horizontal partitioning is achieved by adding the **for** construct to the algebra. This is specified as follows:

$$\textbf{for}\ [x, y]\ \textbf{in}\ (\textit{tarski-expression}1)\ \textit{accumulate}\ (\textit{tarski-expression}2)$$

The tarski-expressions 1 and 2 can be arbitrarily complex, giving the user flexibility over the degree of parallelism. Since we are uniformly operating in the conceptual world of binary relations, there are no typing problems, and expressions can be nested within other expressions to create complex expressions. The "**for** $[x, y]$" partitions tarski-expression1 into its component pairs and assigns x to the left component and y to the right component of each pair, i.e. $[x, y]$ is a relation corresponding to each pair in the binary relation computed by tarski-expression1. As a shorthand, we also allow the use of constants such as $[x,'2']$, where y is bound to the constant 2. In the following subsection, we explain what the *accumulate* operator does.

5.2.2 Accumulation of Subquery Results

In the **for** construct illustrated above, the accumulate operator "accumulates" the results of tarski-expression2 for each pair in tarski-expression1. The values x and y specify the pairs in tarski-expression1 and may occur in tarski-expression2, and hence tarski-expression2 may evaluate to different results for different $[x, y]$. (x and y are treated as constants in tarski-expression2.) The relations occurring in tarski-expression2 are also partitioned depending on the values of x and y. This partitioning can be very well exploited by a storage mechanism which stores a single relation on different disks depending on the values of the attributes. Tarski-expression2[16] could be evaluated in parallel for each pair $[x, y]$ in tarski-expression1, the results of which are then accumulated to provide the final answer to the query. This specifies *fan-in* parallelism. The *accumulate* operator has to satisfy the following properties [19].

1. It should be *consistent* with the basic Tarski algebra,
2. It should be *commutative*, and
3. It should be *associative.*

These properties ensure that the results can be accumulated in any order, and therefore ensure that the parallelism is truly independent.

We have identified three important accumulation operators, that are commutative and associative.

1. $\bigcup$ (Generalized Union)
2. $\bigcap$ (Generalized Intersection)
3. $\bigoplus$ (Generalized Symmetric Difference)

The $\bigcup$ operator performs a union of the subquery results to provide the final answer, the $\bigcap$ operator performs an intersection of the subquery results, and the $\bigoplus$ operator performs a symmetric difference[17] of the subquery results.

As a quick example to illustrate the **for** construct and the $\bigcup$ accumulator, consider the following specification for the composition of two binary relations r and s:

$$\textbf{for} \;\; [y, y] \;\; \textbf{in} \;\; (r^{\pi_r}) \;\; \bigcup (\sigma^r_y(r) \odot \sigma^l_y(s))$$

Here, for each distinct value in the right column of r, we compute the composition of the subrelations $\sigma^r_y(r)$ and $\sigma^l_y(s)$, and then accumulate the results. ($\sigma^r_y(r)$ selects from relation r those pairs whose right component is equal to y.) We will see more complicated examples in Section 5.4, where each of the three accumulation operations are useful.

It is important to point out that most other approaches that use the divide and conquer approach to parallelism [12, 19, 27, 42] use the functional programming paradigm, and rely on a host of complicated type constructors. This not only makes these languages unduly complex, but also makes query optimization more difficult and less general because each extra type constructor necessitates special type-specific optimization rules. Typically, several variants of the same optimization technique are used because of the nuances of the complicated type constructors. Our approach, on the other hand, uses a simple conceptual underlying binary relational schema and is uniform and elegant making query formulation simple and query optimization easier and more general. We show examples of queries in Section 5.4 that can be solved in our extended algebra. We also show how these extensions enable us to eliminate oid-creation.

[16] Frequently, we use the expression $\{x, y\}$ in tarski-expression2 to denote a binary relation with a pair consisting of x and y.

[17] Symmetric difference of two sets is defined as elements that belong to either of the two sets but not to both.

5.3 Other Extensions : Generalized Bonding Operators and Aggregate Functions

The basic Tarski algebra has the *composition* operator as a primitive. This operator is similar to an *equijoin* in the relational model. However, there is no general join operator. We have extended the basic algebra with a few useful *generalized bonding* operators. These bonding operators are join operations with join conditions different from equality. We must point out here that the bonding is performed with the right column of the first relation and the left column of the second relation in a manner very similar to composition. We later show examples where we use the generalized bonding operators to make query formulation simpler. These operators are specified as follows:

- $r \overset{\geq}{\odot} s$ This is the set of pairs (u, w) such that there exist values v and x, such that the ordered pair $(u, v) \in r$ and $(x, w) \in s$, and $v \geq x$.
- $r \overset{\leq}{\odot} s$ This is the set of pairs (u, w) such that there exist values v and x, such that the ordered pair $(u, v) \in r$ and $(x, w) \in s$, and $v \leq x$.
- $r \overset{\neq}{\odot} s$ This is the set of pairs (u, w) such that there exist values v and x, such that the ordered pair $(u, v) \in r$ and $(x, w) \in s$, and $v \neq x$.

There are also two important derived operators that are very useful in the context of several frequent but difficult queries.

- $r \overset{all}{\odot} s$ This is the set of pairs (u, w) such that $\forall v$, $(u, v) \in r \Rightarrow (v, w) \in s$.
- $r \overset{no}{\odot} s$ This is the set of pairs (u, w) such that $\nexists v$, such that $(u, v) \in r$ and $(v, w) \in s$.

We will show later how to simulate $r \overset{all}{\odot} s$ in the extended Tarski algebra with the **for** construct, and show how this becomes useful in the context of queries with universal quantification.

We also have added aggregation operations, specified as **aggregate**(*tarski-expression*), which perform an aggregation on the right column of the binary relation corresponding to the tarski-expression. Typical examples of such aggregate functions are **count**, **avg** etc. We show examples later of queries that use these aggregate functions, and also show how they may be used to specify optimization techniques within the algebra itself.

Before we proceed, we would like to introduce the **if then else** construct. This is a derived construct that can be simulated in the basic Tarski algebra [22] and is specified as follows:

if (*tarski-expression*1) **then** (*tarski-expression*2) **else** (*tarski-expression*3)

The semantics of this construct are straightforward, if tarski-expression1 evaluates to a non-empty binary relation, then the binary relation that tarski-expression2 evaluates to is returned, else the binary relation that tarski-expression3 evaluates to is returned.

5.4 Examples

In this section, we will illustrate how several difficult queries can be solved in a parallel manner with the Extended Tarski algebra by "partitioning" and "accumulation". Most of our examples are chosen from the set of 66 queries in [36] to show the expressive power of our algebra. The examples are chosen to cover different types of queries, from very simple to very difficult, to show the range of the algebra, and are numbered as in [36]. Some of our own examples are also presented and are ordered alphabetically. Some of the examples may be solved with the basic Tarski algebra itself. The schema used for these queries is shown in Figure 9.

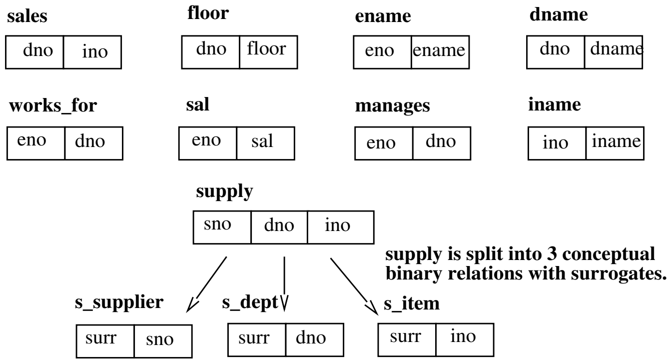

Figure 9: Schema used for the example queries in the Extended Tarski algebra.

Query 3 : Find the items sold by the departments on the 2nd floor

$[(\sigma^r_{'2'}(floor))^{-1} \odot (sales)]^{\pi_r}$

This is a very simple example, and is solved in the basic Tarski algebra. We can also solve this in the Extended Tarski algebra as follows:

$\textbf{for } [d, '2'] \textbf{ in } (floor) \bigcup ($
$\qquad [\sigma^l_d(sales)]^{\pi_r} \;)$

For each department on the 2nd floor, we just compute the items sold and accumulate them.

Query 4 : Find the items sold by all departments on the 2nd floor

$\textbf{for } [d, '2'] \textbf{ in } (floor) \bigcap ($
$\qquad [\sigma^l_d(sales)]^{\pi_r} \;)$

Here, for each department on the 2nd floor, we compute its set of items, and perform an intersection of the results, which gives us the items sold by all departments on the 2nd floor. It is interesting to contrast this query with Query 3, where the only difference is that Query 3 used the $\bigcup$ operator. The $\bigcap$ accumulator plays a significant role in the specification of queries with universal quantification. We now contrast this solution, with the solutions in the basic Tarski algebra[18] and SQL.

- **Basic Tarski algebra solution.**
  $dept2 = \sigma^r_{'2'}(floor)$
  $temp = [\textbf{complement}(sales^{-1}) \odot dept2]^{\pi_l}$
  This gives the items which are not sold by all departments on the 2nd floor.
  $result = iname^{\pi_l} - temp$

- **SQL solution.**
  select *ino*
  from *iname i*
  where not exists (select $*$
  from *floor f*
  where $floor = '2'$

[18] We use set difference which can be simulated in the basic Tarski algebra as shown in [22].

and not exists (**select** $*$
from *sales*
where $ino = i.ino$ **and** $dno = f.dno$))

It is clear that both the basic Tarski algebra and the SQL solutions are complicated. This example clearly illustrates that query formulation in our extended algebra is simpler and more intuitive. Also, it is important to point out that the basic Tarski algebra and the SQL solutions use negation, set difference, or complementation to solve the query, whereas the Extended Tarski algebra solution does not. This, we feel will lead to a better understanding of non-monotonic query optimization, because optimization in the context of negation has been a tough problem in conventional query languages. We elaborate more on this in Section 6.1.

Query 5 : Find the items sold by at least two departments on the second floor

$$\textbf{for } [i,i] \textbf{ in } (iname^{\pi_l}) \bigcup (\\ \textbf{if } (\textbf{count}[(\sigma^r_i(sales))^{-1} \odot (\sigma^r_{'2'}(floor))^{\pi_l}] \geq' 2') \textbf{ then } (\{i,i\}) \textbf{ else } (\{\}))$$

This query is quite easy to formulate and understand, because of the usage of the **count** function, and the **if then else** construct.

Query 2 : Find the items sold by no departments on the 2nd floor

This query can be easily formulated with the **if then else** construct.

$$\textbf{for } [i,i] \textbf{ in } (iname^{\pi_l}) \bigcup (\\ \textbf{if } (\sigma^r_{'2'}[(\sigma^r_i(sales))^{-1} \odot floor]) \textbf{ then } (\{\}) \textbf{ else } (\{i,i\}))$$

Now, we also illustrate a second way to solve the query using two accumulators to show that we can provide different degrees of parallelism for the same query.

$$\textbf{for } [d,f] \textbf{ in } (floor) \bigcup (\\ \textbf{for } [u,'2'] \textbf{ in } (floor) \bigcap (\\ [(\sigma^l_d(sales))^{\pi_r} - (\sigma^l_u(sales))^{\pi_r}]))$$

In this query, the $\bigcap$ operator ensures that items are sold by no departments on the 2nd floor, because it performs an intersection of the items sold by each department that are not sold by any department on the 2nd floor.

Query 8 : Find the departments where all the employees earn less than their manager

$$temp(d,d) = [(\sigma^r_d(works\_for))^{-1} \odot sal] \overset{\geq}{\odot} [(\sigma^r_d(manages))^{-1} \odot sal]^{-1}$$

$temp(d,d)$ is non-empty iff there is some dept where an employee earns more than his manager because of the semantics of the $\overset{\geq}{\odot}$ operator, and this is exploited to solve the query.

$$\textbf{for } [d,d] \textbf{ in } (dname^{\pi_l}) \bigcup (\\ \textbf{if } (temp(d,d)) \textbf{ then } (\{\}) \textbf{ else } (\{d,d\}))$$

This query shows a very interesting use of the $\overset{\geq}{\odot}$ operation, to provide a simple solution.

Query 24 : List the items supplied to all departments by all suppliers

$$Item\_no(s,d) = \{\{[\sigma^r_s(s\_supplier)]^{\pi_l} \cap [\sigma^r_d(s\_dept)]^{\pi_l}\} \odot s\_item\}^{\pi_r}$$

This gives all the items by supplier s to dept d, after which we can solve the query with two accumulators.

for $[s,s]$ **in** $(s\_supplier)^{\pi_r} \bigcap$ (
for $[d,d]$ **in** $(s\_dept)^{\pi_r} \bigcap$ (
$Item\_no(s,d)$))

This is also a very interesting query and is a generalization of Query 4. This is actually a very difficult query to solve in SQL, and again, we see that query formulation in the Extended Tarski algebra can be very simple and intuitive. An item that survives both the $\bigcap$ operations will be supplied to all departments by all suppliers.

Query 35 : List the suppliers that are the only supplier of some item

for $[s,i]$ **in** (s_item) $\bigcup$ (
if (**count**$(\sigma^r_i(s\_supplier^{-1} \odot s\_item)) =' 1'$)
then $([\sigma^l_s(s\_supplier)]^{\pi_r})$ **else** $(\{\})$)

This query is quite easy to formulate and understand, again because of the usage of the **count** function, without which the query is hard to formulate.

Query 50 : Find, for each department the average salary in the department

for $[d,d]$ **in** $(dname^{\pi_l}) \bigcup$ (
avg$[(\sigma^r_d(works\_for))^{-1} \odot sal]$)

This query shows how the aggregate function **avg** is useful.

Query 66 : Is it true that all the departments that sell items of type 'A' are on the 3rd floor ?

$$temp('A',f) = [(\sigma^r_{'A'}(iname))^{\pi_l} \odot sales^{-1}] \odot floor$$

This computes, for items of type 'A', the floors of the departments that sell it.

if $(temp('A',f) \overset{\neq}{\odot} \{'3','3'\})$ **then** $(\{0,0\})$ **else** $(\{1,1\})$

Here $\{'3','3'\}$ is a pair that encodes the 3rd floor. This query uses the $\overset{\neq}{\odot}$ operator. This operation ensures that the result of the **if** will be non-empty iff some department that sells items of type 'A' is not on the 3rd floor. We encode false as $(\{0,0\})$ and true as $(\{1,1\})$.

Query A : Find the employees who do not work in more than one department

for $[d,d]$ **in** $(dname^{\pi_l}) \bigoplus$ (
$[\sigma^r_d(works\_for)]^{\pi_l}$)

This query shows an interesting use of the $\bigoplus$ accumulator. The semantics of this operation eliminate all employees who work in more than one department.

6 Query Optimization

6.1 Optimization of Non-monotonic Queries

In Query 4 (and in Query 24) in the previous section, we showed how a non-monotonic[19] query can be solved without the use of negation, complementation, or set difference. This is important in the context of optimization, because it is hard to optimize queries that use negation, complementation, or set difference. Since the Extended Tarski algebra formulation is entirely positive (uses intersection), we feel it is much better suited for query optimization. For example, selections can be moved inside an intersection, but not inside a negation, complementation, or set-difference. This is significant, because it opens the door for many non-trivial optimizations in the context of non-monotonic queries. We now show one more example that brings out this aspect very elegantly.

Universal quantifiers are frequently used in many non-monotonic queries. Algebratizing such queries and optimizing them is quite challenging in conventional algebras, and they are often solved by introducing negation even when the query does not involve explicit negation. In our extended algebra, we have found ways to algebratize many queries with universal quantification without introducing negation. Consider the expression $r \overset{all}{\odot} s$, outlined in Section 5.3. This expression is non-monotonic because adding some pairs to r may cause the result to shrink. Also, this expression frequently occurs in the formulation of many difficult queries involving universal quantification. We show how to simulate this in the Extended Tarski algebra without any negation or complementation.

$$
\begin{array}{l}
\textbf{for } [x,x] \textbf{ in } (r^{\pi_l}) \bigcup (\\
\qquad\qquad\qquad\qquad \textbf{for } [u,v] \textbf{ in } (\sigma^l_x(r)) \bigcap (\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\{u,v\} \odot \sigma^l_v(s)\ )\ )
\end{array}
$$

In this formulation, the key operator is the $\bigcap$ which ensures that a pair (a,b) is part of the result if the set of values associated with a in the left component of r is a subset of the set of values associated with b in the right component of s. The $r \overset{all}{\odot} s$ operation is used very frequently in many non-monotonic queries that involve universal quantification, and this positive formulation is a significant step toward optimization of non-monotonic queries[20]. Also, the query can be executed in parallel and the query formulation suggests an effective algorithm for parallel execution.

6.2 Physical Level Optimization

Most query languages for databases leave all of the task of optimization to the query optimizer. In this section, we show how the **count** function enables specification of some optimization techniques in the algebra itself. This is best illustrated by a simple example. Suppose that there are binary relations r, s, and t, and assume that we are interested in the operation $r \odot s \odot t$. Since composition is associative, this can be done in one of two ways, $(r \odot s) \odot t$, or $r \odot (s \odot t)$. It might be better to perform the composition in one of the above ways depending on the sizes of the individual relations. Such optimization decisions can be specified in the algebra itself, as follows:

$$\textbf{if } [\textbf{count}(r \odot s) < \textbf{count}(s \odot t)] \textbf{ then } [(r \odot s) \odot t] \textbf{ else } [r \odot (s \odot t)]$$

In this case, the **count** function need only use an estimation for the size of the joins $r \odot s$ and $s \odot t$ instead of computing the joins directly. It is clear that more complicated optimization techniques can be specified within the algebra in a similar fashion.

[19] A non-monotonic query is one whose result may shrink if new tuples are added to the database.

[20] Of course, $r \overset{all}{\odot} s$ does not help in solving all queries with universal quantification, but is useful for solving many such frequently occurring queries.

6.3 Specification of Different Algorithms for an Operation

For the same operation, the Extended Tarski algebra is capable of providing different algorithms for parallel execution in an algebraic framework. Depending on the configuration of data, any of the algorithms might be chosen to execute the queries more efficiently, giving the user more flexibility to choose a particular algorithm.

We illustrate this by showing two more algorithms for specifying the composition operator of the basic Tarski algebra, in addition to the one we illustrated in Section 5.2.2.

1. Nested for loops.
   $r \odot s = \textbf{for } [x, y] \textbf{ in } (r) \bigcup ($
   $\qquad \textbf{for } [y, z] \textbf{ in } (s) \bigcup (\ \{x, z\}\ )$

2. Iteration over composition components.
   $r \odot s = \textbf{for } [c, c] \textbf{ in } (r^{\pi_r} \cap s^{\pi_l}) \bigcup ($
   $\qquad \sigma^r_c(r) \odot \sigma^l_c(s)\ )$
   This is similar to an index lookup join, where we iterate over the common components via an index.

6.4 Elimination of Oid-Creation

As briefly mentioned earlier, oid-creation is an absolute necessity for solving complex queries as shown in Figure 8. This could be a potentially expensive operation. With the **for** construct extension, this query graph can be solved without oid-creation and in a more optimized manner as follows:

$$\begin{aligned}&\textbf{for } [y, z] \textbf{ in } (r) \bigcup (\\ &\qquad \textbf{for } [u, u] \textbf{ in } ((\sigma^l_y(r))^{\pi_r} \cap (\sigma^l_z(r))^{\pi_r}) \bigcup (\\ &\qquad\qquad [\sigma^r_y(r) \cap \sigma^r_u(r) \cap \sigma^r_z(r)]^{\pi_l}\ )\ )\end{aligned}$$

This example is fairly easy to follow from the query graph in Figure 8 as we simply enforce the constraints of the query graph in the algebra. We compute the U nodes that are constrained by the relationship r to nodes Y and Z, and then compute the X nodes that are connected to the U, Y, and Z nodes.

As another example to show how the **for** construct can be used to eliminate oid-creation, consider the very simple operation $r \cap s$, that is not a primitive in the Tarski algebra. This is used very frequently, but requires oid-creation[21] to simulate it in the basic Tarski algebra. This can be very easily done in the Extended Tarski Algebra with the **for** construct as follows:

$$\begin{aligned}&\textbf{for } [x, y] \textbf{ in } (r) \bigcup (\\ &\qquad \textbf{for } [u, v] \textbf{ in } (s) \bigcup (\\ &\qquad\qquad \{x, x\} \odot \{u, v\} \odot \{y, y\}\ )\ )\end{aligned}$$

This shows that the Extended Tarski algebra is better suited for effective query optimization. This also shows that the Extended Tarski algebra (without oid-creation) is equivalent to the basic Tarski algebra in expressive power.

[21]Due to finite complementation in the basic Tarski algebra, $r \cap s$ cannot be expressed as $\overline{\overline{r} \cup \overline{s}}$ using DeMorgan's laws.

6.5 Other Optimizations

In [44], we outlined several syntactic and semantic graph level optimization techniques for the translation of object based queries into the Tarski algebra. We also showed some algebraic rewrite optimization rules. These techniques, in conjunction with the techniques in the previous subsections are very useful for good query optimization in the context of parallel object based query processing.

7 Summary and Future Research

This paper makes the following contributions:

- Shows that the Tarski algebra is a strong and simple algebraic foundation for object based query languages.
- Facilitates specification of parallel query processing strategies in an algebraic framework.
- Facilitates specification of optimization techniques in an algebraic framework.
- Maintains a simple underlying framework without the need for complicated type constructors, thereby keeping the language simple and allowing general optimization rules which are not type-specific unlike other most other complex object and object based algebras.
- Provides for effective optimization of non-monotonic queries.

We showed that many non-monotonic queries can be solved without the use of negation, which could lead to better optimization of such queries. This is a very promising future research topic and needs to be addressed in more detail by extending it to all queries with universal quantification in general. Also, we are interested in exploring the implementation issues involved in a parallel framework that the algebra proposes. There are also several optimization rewrite rules that can be used for optimization in the context of the generalized bonding operators which need to be explored.

References

[1] S. Abiteboul, P. Fischer, and H.J. Schek, editors. Nested Relations and Complex Objects in Databases. *# 361 in Lecture Notes in Computer Science*, Springer-Verlag, 1989.

[2] S. Abiteboul and V. Kanellakis. Database Theory Column: Query Languages for Complex Object Databases. *ACM SIGACT NEWS* 21, 2, Summer 1990, pp. 9–19.

[3] S. Abiteboul and V. Kanellakis. Object Identity as a Query Language Primitive, In *Proc. ACM SIGMOD*, May 1989, pp. 159–173.

[4] S. Abiteboul and V. Vianu. Procedural Languages for Database Queries and Updates. *Journal of Computer and System Sciences* 41, 1990, pp. 181–229.

[5] J. Annevelink. Database Programming Languages: A Functional Approach. In *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, Denver, Colorado, 1991, pp. 318–327.

[6] M. Atkinson, F. Bancilhon, D.J. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The Object-Oriented Database System Manifesto. In *Proc. 1st Int'l Conf. on Deductive and Object-Oriented Databases*, Kyoto, Japan, 1989, pp. 40–57.

[7] F. Bancilhon. Object Oriented Database Systems. In *Proc. 7th ACM SIGACT-SIGMOD-SIGART Symp. on Princ. Database Systems*, Austin, Texas, 1988, pp. 152–162.

[8] F. Bancilhon, T. Briggs, S. Khoshafian, and P. Valduriez. FAD, A Powerful and Simple Database Language. In *Proc. of the 13th Int'l Conf. on Very Large Databases*, 1987.

[9] F. Bancilhon, S. Cluet, C. Delobel. A Query Language for the O_2 Object-Oriented Database System. In *2nd Int'l. Workshop on Database Programming Languages*, Oregon, 1989, pp. 122–138.

[10] C. Beeri. New Data Models and Languages - The Challenge. In *ACM SIGACT-SIGMOD-SIGART Symp. on Princ. Database Systems*, San Diego, California, 1992, pp. 1–15.

[11] C. Beeri, Y. Kornatzky. Algebraic Optimization of Object-Oriented Query Languages. In *Proc. 3rd Int'l Conf. on Database Theory*, Paris, France Dec. 1990, pp. 72–88.

[12] V. Breazu-Tannen, P. Buneman, and S. Naqvi. Structural Recursion as a Query Language. In *Proc. 3rd Int'l Workshop on Database Programming Languages*, Greece, Sept. 1991.

[13] V. Breauzu-Tannen et.al. (pp. 23–28), S. Vandenberg and D.J. DeWitt (pp. 48–53), and S. Daniels et.al. (pp. 58–63). *Special Issue on Foundations of Object-Oriented Database Systems. Data Engineering*, 14, 2, June 1991.

[14] V. Bussche and J.Paredaens. The Expressive Power of Structured Values in Pure OODB's. In *Proc. 10th ACM SIGACT-SIGMOD-SIGART Symp. on Princ. Database Systems*, 1991, pp. 291–299.

[15] M.J. Carey, D.J. DeWitt, G. Graefe, D.M. Haight, J.E. Richardson, D.T. Schuch, E.J. Shekita, and S.L. Vandenberg. The EXODUS Extensible DBMS Project: An Overview. In *Readings in Object-Oriented Database Systems.* Edited by S.B. Zdonik and D. Maier, Morgan Kaufmann Publ., 1989.

[16] M.P. Consens and A.O. Mendelzon. GraphLog: A Visual Formalism for Real Life Recursion. In *Proc. 9th ACM SIGACT-SIGMOD-SIGART Symp. on Princ. Database Systems*, Nashville, Tenn., 1990, pp. 404–416.

[17] G. Copeland and S. Khoshafian. A Decomposition Storage Model. In *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, Austin, Texas, 1985, pp. 268–279.

[18] U. Dayal. Queries and Views in an Object-Oriented Data Model. In *Proc. of 2nd Database Programming Languages Workshop*, 1989, pp. 80–102.

[19] M. Erwig, and U.W. Lipeck. A Functional DBPL Revealing High Level Optimizations. In *Proc. 9th ACM SIGACT-SIGMOD-SIGART Symp. on Princ. Database Systems*, Nashville, Tenn., 1990, pp. 417–424.

[20] M. Gyssens, J. Paredaens, and Dirk Van Gucht. A Graph Oriented Object Database Model. In *Proc. 9th ACM SIGACT-SIGMOD-SIGART Symp. on Princ. Database Systems*, Nashville, Tenn., 1990, pp. 417–424.

[21] M. Gyssens, J. Paredaens, and Dirk Van Gucht. A Graph Oriented Object Model for Database End-User Interfaces. In *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, Atlantic City, New Jersey, 1990, pp. 24–33.

[22] M. Gyssens, L. Saxton, and Dirk Van Gucht. Tagging as an Alternative to Object Creation. Presented at *The Dagstuhl Seminar on Query Processing in Object Oriented, Complex Object, and Nested Relation Databases.*

[23] M. Guo, S.Y.W. Su, and H. Lam. An Association Algebra For Processing Object-Oriented Databases. In *Proc. Seventh IEEE Int'l Conf. on Data Engineering*, 1991.

[24] L.M. Haas, J.C. Freytag, G.M. Lohman and H. Pirahesh. Extensible Query Processing in Starburst IBM Almaden Research Center, San Jose, California.

[25] A. Hafez, and G. Ozsoyoglu. Storage Structures for Nested Relations. In *Special Issue on Nested Relations*, IEEE Computer Society Technical Committee on Data Engineering, Sept. 1988, pp. 31–38.

[26] E.N. Hanson, T.M. Harvey, and M.A. Roth. Experiences in DBMS Implementation Using an Object-Oriented Persistent Programming Language and a Database Toolkit. In *Proc. OOPSLA 1991*, pp. 314–328.

[27] B. Hart, S. Danforth, and P. Valduriez. Parallelizing FAD, A Database Programming Language. In *Proc. Int'l Symposium on Databases in Distributed and Parallel Systems*, Austin, Texas, Dec. 1988.

[28] R. Hull and M. Yoshikawa. ILOG: Declarative Creation and Manipulation of Object Identifiers. D. McLeod, R. Sacks-Davis, and H. Schek, editors, In *Proc. 16th Int'l Conf. on Very Large Databases*, Brisbane, Australia, 1990.

[29] S. Khoshafian, G. Copeland, T. Jagodits, H. Boral and P. Valduriez. A Query Processing Strategy for the Decomposed Storage Model. In *IEEE Data Engineering*, 1987. pp. 636–643.

[30] S. Khoshafian and P. Valduriez. Sharing, Persistence, and Object Orientation: A Database Perspective. Advances in Database Programming Languages. *ACM Press, Frontier Series*, pp. 221–240.

[31] M. Kifer, W. Kim, and Y. Sagiv. Querying Object Oriented Databases. In *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, San Diego, California, 1992, pp. 393–402.

[32] M. Kifer, G. Lausen, and J. Wu. Logical Foundations of Object Oriented and Frame-Based Languages. *Technical Report 90/14*, Dept. of Comp. Science, SUNY Stony Brook, 1990.

[33] W. Kim and F.H. Lochovsky, editors. Object-Oriented Concepts, Databases, and Applications. *ACM Press (Frontier Series)*, 1989.

[34] W. Kim, D.S. Reiner, and D.S. Batory. Query Processing in Database Systems. *Springer Verlag Publishers*, 1985.

[35] G. Kuper and M. Vardi. A New Approach to Database Logic. In *Proc. of the 3rd ACM Symposium on Principles of Database Systems*, pp. 86–96, ACM Press, 1984.

[36] M. Lacroix and A. Pirotte. Example Queries in Relational Languages. *Technical Note No. 107* Brussels: M. B. L. E.

[37] R.D. Maddux. The Origin of Relation Algebras in the Development and Axiomatization of the Calculus of Relations. In *Studia Logica L, 3/4*, pp. 421–455, 1991.

[38] D. Maier. The Theory of Relational Databases. *Computer Science Press*, 1983.

[39] M. Mannino, I. Choi, D. Batory. An Overview of an Object Oriented Data Language. In *Proc. of IEEE Data Engineering Conf.*, 1989.

[40] O. Ore. Theory of Graphs. *American Mathematical Society*, 1962.

[41] J. Paredaens, J. Van den Bussche, D. Van Gucht, V. Sarathy, and L. Saxton. An Overview of GOOD. In *SIGMOD Record*, Volume 21, Number 1, March 1992.

[42] D.S. Parker, E. Simon, and P. Valduriez. SVP - A Model Capturing Sets, Streams, and Parallelism. In *Proc. of the 18th Int'l Conf. on Very Large Databases*, Vancouver, Canada, 1992.

[43] A. Rosenthal and C.G. Legaria. Query Graphs, Implementing Trees, and Freely-Reorderable Outerjoins. In *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, Atlantic City, New Jersey, 1990, pp. 291–299.

[44] V. Sarathy, L. Saxton and D. Van Gucht. Translating Query Graphs into Tarski Algebra Expressions. *Technical Report # 342*, Dept. of Computer Science, Indiana University, December 1991.

[45] V. Sarathy, L. Saxton and D. Van Gucht. Algebraic Foundation and Optimization for Object Based Query Languages. To appear in *Proc. Ninth IEEE Int'l Conf. on Data Engineering*, Vienna, Austria, 1993.

[46] G. Shaw and S. Zdonik. An Object-Oriented Query Algebra. In *Data Engineering*, September 1989, pp. 12(3):29–36.

[47] D. Shipman. The Functional Data Model and the Data Language DAPLEX. In *Readings in object-oriented database systems*. Edited by S.B. Zdonik and D. Maier, Morgan Kaufmann Publ., 1989.

[48] M. Stonebraker, editor. Readings in Database Systems. *Morgan Kaufmann Publ.*, 1988.

[49] A. Tarski. On the Calculus of Relations. *Journal of Symbolic Logic*, 6, 1941, pp. 73–89.

[50] A. Tarski and S. Givant. A Formalization of Set Theory Without Variables. *American Mathematical Society*, Providence, Rhode Island, 1986.

[51] J.D. Ullman. Principles of Database and Knowledge-Base Systems, Volume II. *Computer Science Press*, 1989.

[52] P. Valduriez, S. Khoshafian, and G. Copeland. Implementation Techniques for Complex Objects. In *Proceedings of the 12th Int'l Conf. on Very Large Databases*, Kyoto, August 1986, pp. 101–109.

[53] S.L. Vandenberg and D.J. DeWitt. Algebraic Support for Complex Objects with Arrays, Identity and Inheritance. In *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, Denver, Colorado, 1991, pp. 158–167.

[54] W. Wong and K. Youseffi. Decomposition - A Strategy for Query Processing. *ACM Transactions on Database Systems*, Vol. 1, No. 3, pp. 223–241.

[55] C. Zaniolo. The Database Language GEM. Readings in Database Systems. *Morgan Kaufmann Publ.*, 1988.

[56] S.B. Zdonik and D. Maier, editors. Readings in Object-Oriented Database Systems. *Morgan Kaufmann Publ.*, 1989.