Specifying a Database System to Itself

David Maier

Dept. of Computer Science & Engineering
Oregon Graduate Institute
Beaverton 97007
USA

Abstract

Query processors have long been monolithic pieces of software that resist extension. Query capabilities are needed in many applications of persistent object bases, but object models and storage managers are evolving so rapidly that matching a monolithic query processor to each of them is infeasible. The EREQ project seeks to structure object query processors into well-defined software components that can be independently modified, extended and exchanged. This paper sets forth our initial ideas on what the major components will be and how we will specify the interfaces to each.

1 Introduction

The EREQ Project[1] is a recently launched effort to devise a common internal architecture for query processing in persistent object bases. A goal of the work is to specify the interfaces to the query optimizer component precisely enough that we can interchange query optimizers between query processing systems. We are conducting our investigations in an object-oriented database setting, but expect them to apply to other models having user-defined database types with complex state. We expect to test our results through reconciling the internal architectures of the ENCORE, Revelation and EXODUS query processor prototypes [1-3].

Some of the interfaces to a query optimizer are obvious, such as an object algebra for input expressions and a query plan language for its output to the query evaluating layer. There are other less obvious or less direct interfaces. There are connections to one or more programming languages, if the query language is embedded in a host programming language or an algebra expression can contain calls to programming language functions. An optimizer needs information from a schema manager on type definitions and name bindings. Statistics and selectivity information on instances, as well as existence of auxiliary access paths must be available. Other possible inputs are physical properties of the underlying computer system and search heuristics. At each of these interfaces, there is a question of how much of the semantics of the entities crossing the boundary the optimizer needs to understand. At the object algebra interface, it may not need “full” semantics (the

[1]EREQ stands for "Encore-Revelation-Exodus Query System." It is a joint project of Stanley Zdonik at Brown University, David DeWitt and Michael Carey at University of Wisconsin at Madison, David Maier of Oregon Graduate institute, and Goetz Graefe of University of Colorado at Boulder.

denotational meaning of operators). It needs to know about algebraic identities involving the operators, but it does not necessarily need to know how to evaluate a particular operator. At the query plan interface, the optimizer needs to know more than what logical function an operator procedure computes. For reasonable cost estimation, it must know how an algorithm affects physical properties of the data, such as sort order.

Another facet of query optimization is the stability of the information. Information that is stable for the one-shot execution of an ad hoc query may be variable over the life of a query embedded in an application program. Depending on how the validity of query plans is tracked, the optimizer may have to differentiate information that is stable between compilations of the database system from information that is stable between schema changes from information that is only valid until the next database update.

The mind of the questioning reader may still be back at the first paragraph, asking "Why?" That is, what is so important or useful about the ability to change query optimizers in a query processing system? There is some utility in the research setting, being able to compare the efficiency and efficacy of different query optimization strategies in the same environment. One might also speculate on being able to upgrade the query optimizer module of an installed database system, say with a module from a third-party vendor. What we are really after with this exercise in interchangeability is discovering good formalisms or notations with which to describe the various interfaces. The goal is not so much defining a particular query processing architecture as in having a *meta-architecture* in which changes to interfaces resulting from extensions and evolution can be captured. A meta-architecture is a system in which to define particular architectures. The demands of applications for data modeling features, the sophistication of access methods and hardware platform capabilities all increase steadily. Monolithic query processing engines are proving unmanageable to maintain in the face of such changes. A more modular architecture demands a precise means to describe evolving module boundaries, so that work can proceed on different parts of a query processing system in parallel.

A secondary issue is what does "interchangeable" mean in this context? How much rebinding is allowed when exchanging optimizers? Is it simply a matter of relinking code, or are there configuration tables and other supporting data structures the optimizer uses that must be modified? Other possibilities are that the optimizer is generated by an optimizer generator, and must be regenerated in the new environment, or that the optimizer is extensible and requires certain additions in order to conform. Simple relinking is probably too much to expect, but we hope the process can be done automatically—by running tools rather than necessitating any hand-coding.

This project is scarcely begun, so it is generous to even call the ideas here preliminary. This paper marks a trailhead, rather than a point of progress down any path.

2 A Reference Architecture

In this section we describe a generic architecture for query processing, in order to give an overview of the major components involved and their interactions. Figure 1 depicts the architecture.

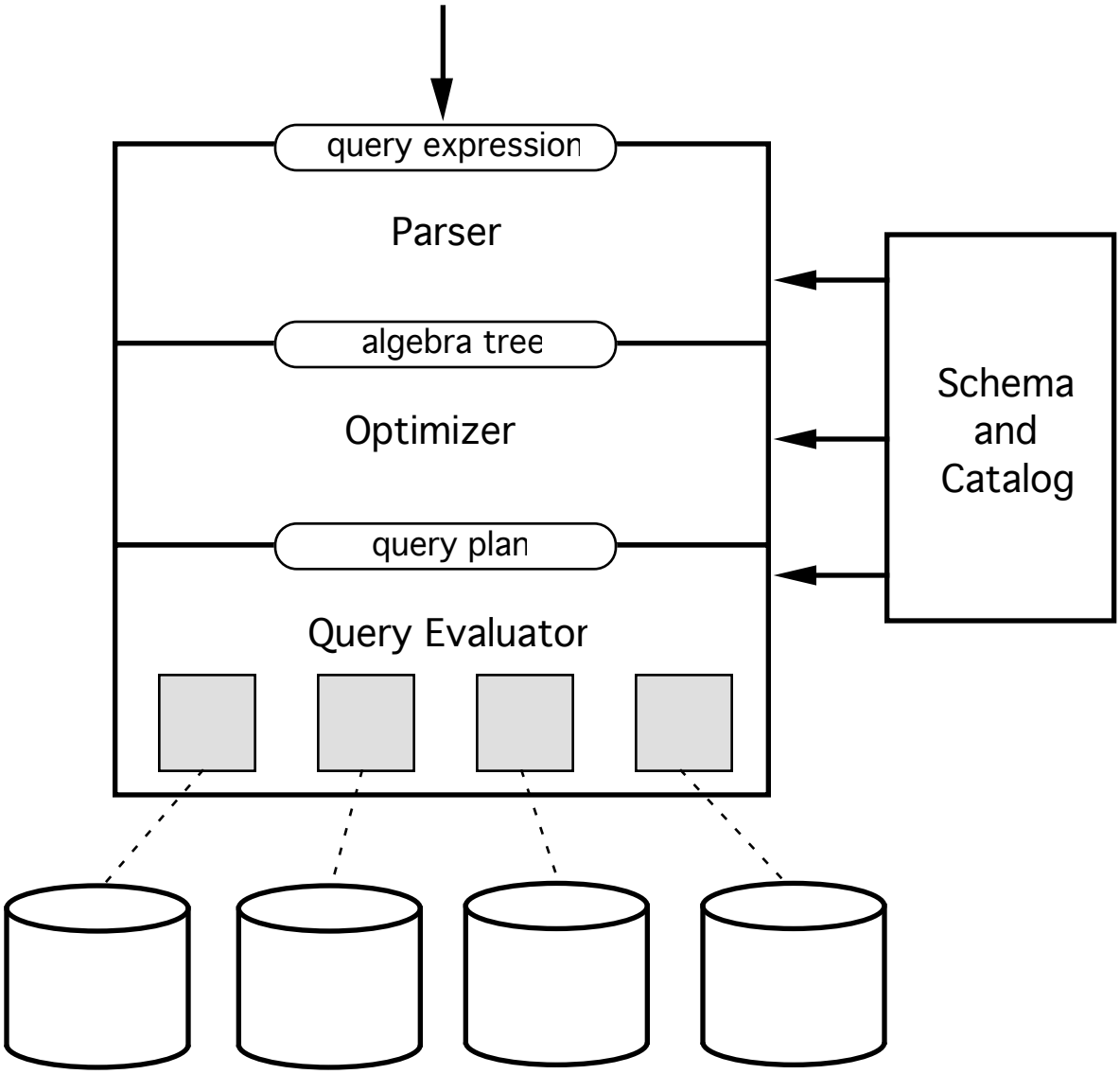


Figure 1. Sample Query Processing Architecture.

Query expressions, either from ad hoc queries or extracted from programs, are parsed into expression trees over a logical object algebra. The optimizer transforms these expressions into query plans for immediate or later evaluation by a query evaluation layer. The schema manager and catalog provide logical and physical information on types and instances to the other three layers. The source form of queries does not concern us much. It might be an SQL-like syntax, an iterator construct in a persistent language, or some form of comprehension notation. In any case, scanning and parsing the source is straightforward.

The logical algebra will be a generalization of the relational algebra. However, rather than dealing with simply sets of records over base types, the values are formed by free composition of a variety of constructors, such as tuple, set (or bag), array and reference. Algebraic expressions can also operate on instances of user-defined abstract types, through the interfaces of those types. An algebra expression might also contain embedded code segments from some programming language. The operations in the algebra will include arithmetic or calculi on the base types (such as addition and Boolean negation),

selectors and accessors on some of the constructed types (field selector, array indexing, dereference) and bulk operations on other types (selection, unnest, map, and so forth).

It is the job of the optimizer both to create choices and to make choices. It creates choices by developing algebraic expressions equivalent to the input expression and by exploring alternative query plans. It makes choices by selecting among the algebraic expressions on the basis of heuristics, and among query plans on the basis of cost estimates. The alternative algebra expressions come from identities such as

```
join(R, S) = join(S, R)

select(select(R, P), Q) = select(R, and(P, Q))
```

In general the number of equivalent expressions that can be developed for a given input expression can be large or even infinite. Hence the optimizer needs to employ search heuristics and strategies to limit the amount of search space it examines.

The alternative query plans for a given algebra expression arise through choosing among different physical operator procedures for a given logical operation, such as hash-join or merge-join for the logical join operation. There need not be a direct match between each physical algorithm and a single logical operation. A join procedure at the query plan level might perform a combination of join and selection as expressed at the logical level. A single physical procedure might also implement more than one logical algebra operation. For example, a join procedure can be used for intersection, generally.

Query plans are typically directed expression graphs whose internal nodes specify physical operator procedures and whose leaves specify database objects. A query plan might not be a tree if common subexpressions are shared. In addition to specifying which physical procedures are applied in which order, a query plan might contain scheduling information, such as whether to apply a certain operator in a demand-driven or data-driven manner. The query plan language can be viewed as a physical, or more concrete, algebra. Many of the physical procedures will have direct counterparts among the logical algebra operations, as far as function goes. Such procedures might also have physical parameters, specifying options such as sort orders or duplicate information that are not part of the semantics of the logical algebra. There can also be physical procedures that are no-ops at the logical level. Such an operator is inserted to affect physical properties of intermediate results, such as partitioning and placement in the memory hierarchy, or to modify the processing of the query plan, such as to use parallelism.

A query plan may be executed immediately or catalogued away for later use. The query evaluation layer might interpret plans directly, or compile them into some lower level operations. Execution of plans can take place in data-driven rather than demand driven mode. Also, physical operator procedures might evaluate in “batch” rather than stream mode, constructing the entire result relation of an operation at once, rather than a record at a time. (The possibilities here must also be known by the query optimizer, whether it is making the decision on evaluation mechanism, or it is given that information from without.) In Figure 1, the query evaluation layer comprises the physical operator procedures, buffer and file management routines, access methods, data structure realizations, and connections to the concurrency and recovery services of the DBMS. In general, the evaluator will have parallel processing capabilities. By data structure realizations, we mean that a given type constructor, such as tuple or set, might have more than one physical implementation. For example, small sets might be implemented with single records, while large sets might use files of records or B-trees.

The schema and catalog contain type definitions, type implementations and declarations of persistent variables. It is the persistent variables that are the starting points for queries. We posit that the persistent variables reside in some sort of name space, where constraints on and between the variables can be declared. The schema and catalog also must be able to provide physical information on individual instances, such as sizes, ranges of values and existence of indices. One feature that sets query optimization off from programming language optimization is recognizing that instances of the same type may have widely different physical properties.

2.1 An Example of Query Processing

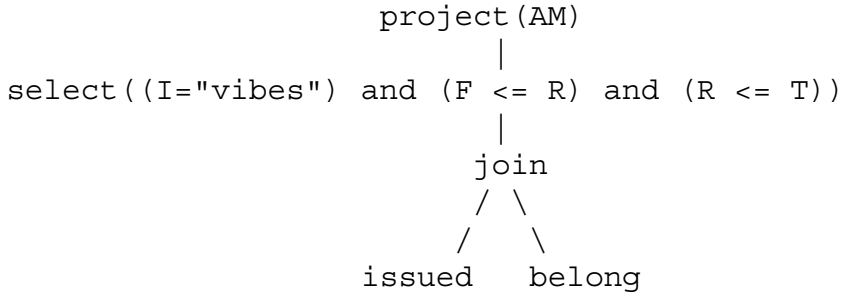
We illustrate this process with a relational database example. Consider relations

```
issued(Album, Group, RecordedOn, DateOfIssue)
belong(Group, Member, Instrument, From, To)
```

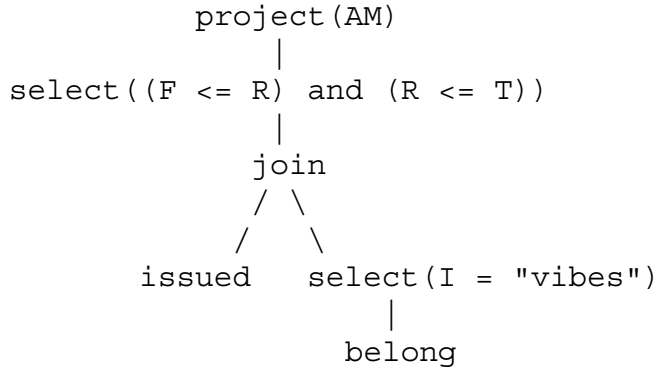
Where `issue` gives the dates that a musical group recorded and issued an album, and `belong` gives the range of dates a particular musician played a given instrument with a specific group. Consider a query to find an album with a vibraphone player on it, plus the player. Such a query might appear in a relational calculus language as

```
{[i.Album, b.Member]|i from issued, b from belong,
        i.Group = b.Group, b.Instrument = "vibes",
        between(b.From, i.RecordedOn, b.To)}
```

This form of the query can be parsed and expressed in the logical algebra as follows. (Attributes have been abbreviated to a single letter here.)

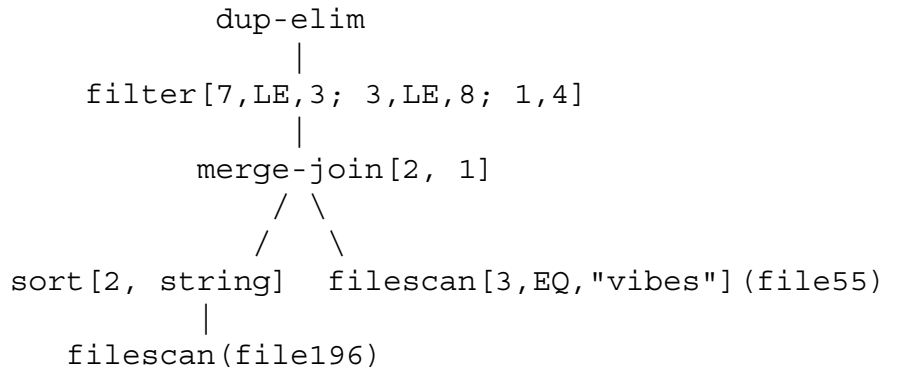


On the basis of a general heuristic or a cost estimate, this logical algebra expression can be transformed by moving a portion of the selection condition below the join:



The query plan phase then selects particular physical algorithms for each logical operator, filling in the appropriate arguments, and binding identifiers to files or offsets

based on catalog information on the location and physical layout of relations. The choices are based on statistics kept for the relations and cost estimates for the physical procedures:



Here we assume the evaluation strategy is demand driven, with each physical algorithm providing a stream of records as input to the algorithm above it. There are a couple points to note here. One is that the correspondence between logical operators and physical algorithms is not necessarily one-to-one. For example, the `filescan` on the right branch of the join implements both access to the `belong` relation (located in `file55`) and a `select`. Also, the top `select-project` pair is realized with a `filter` procedure, which passes records selectively and removes fields, followed by a procedure for eliminating duplicate record values. The second point is that certain procedures may be sensitive to physical properties of the data and additional operations may be inserted to achieve the requisite properties. For example, `merge-join` requires its input streams to be sorted on the join attributes, hence the need for the `sort` procedure. Here we assume that `file55` is already in the desired order. Note that the `sort` procedure is a no-op at the logical level, since relations are treated as unordered sets of tuples at that level.

3 Interface Specification Issues

Evidently, the logical algebra and query plan expressions represent two key interfaces in our reference architecture. There are also one or more interfaces to the schema and catalog component. Less obvious interfaces include the one between an application programming language and the query subsystem, both for invoking queries and returning results. Another is from the query processor to information about the capabilities of the underlying computing platform: available memory, scratch space on disk, number of processors, processor loads, and so forth.

In the rest of this paper, we discuss these interfaces, with particular attention to the parser-optimizer interface and the optimizer-query evaluator interface. Clearly, in specifying the interfaces between the layers, there is at a minimum a need for an operational fit. That is, the values passed down by one layer must be expected by the layer below. But there is much to decide about the interface beyond the format of their values. The main issue for is now is what notation or formalism to use to describe each interface. That choice, in turn, depends on what needs to be described.

A word on terminology. We use the adjective “logical” to refer to the object algebra and the software layers around it. We use “physical” to indicate operator procedures and other parts of the query evaluator layer.

4 The Parser-Optimizer Interface: The Logical Algebra

In our reference architecture, it is quite clear what passes from the parser to the optimizer: expressions from a logical object algebra. A data algebra is characterized by a collection of base types and data structures, with their associated operators. The relational algebra of the relational data model is probably the best formalized aspect of database systems. (Still, definitions of relational algebra [4-6] exhibit variations in several areas: typed or untyped domains, the particular choice of base types and associated operators, labelled or positional column notation, provision for constant relations, and so forth. Also, even though the operators can be precisely specified, typing systems that handle highly polymorphic operators such as join are non-trivial [7].) The relational algebra has one data structuring construct, the relation, which cannot be applied iteratively. We expect an object algebra will contain several constructors, such as set, tuple, and array, that may be composed freely. We also expect it to incorporate object identity and the ability to operate on user-defined types through messages. There have been numerous proposals for object algebras, such as those of Beeri and Kornatsky [8] and Osborn [9]. We expect that ours will resemble those of Vance [10] and Vandenberg [11].

It might seem, then, that specifying the parser-optimizer interface is a matter of selecting a semantic formalism, such as set theory, equational specifications or denotational semantics, and giving a definition to each operator in the algebra. However, from the point of view of the optimizer architecture, that approach might be both too much and too little. Yes, the optimizer does receive algebra expressions as input, but it does not necessarily need to know their precise meaning to do its job. The optimizer is concerned with what are valid transformations on algebra expressions; it does not necessarily need to know enough about the algebra to evaluate those expressions. For example, the equivalence

```
intersect(union(Q, R), S) =
        union(intersect(Q, S), intersect(R, S))
```

holds whether `Q`, `R` and `S` are sets or multisets (where intersection and union take the minimum and maximum of element cardinalities, respectively). For transforming expressions, the precise meaning of `intersect` and `union` does not matter, as long as the appropriate algebraic properties hold. (Of course, there may be other operations where sets and multisets differ.) On the other hand, given the formal semantics of the algebra operators, deducing useful equivalences automatically may be non-trivial. Hence, it seems that an optimizer needs not so much the formal definition of the operators as the legal transformations that hold on them.

Positing that expression transformations are the description the optimizer needs for the object algebra, what system do we use for capturing them? A legal transform might be captured simply as a chunk of code. Given an input expression, the code uses arbitrary means to reformulate it to a transformed output expression. Such a solution seems less than satisfactory. Captured as a chunk of code, transformations would not be easily manipulable, say to recognize commonalities in patterns for efficient search or to combine transformations that are used together frequently. A better starting point is likely a term-rewriting system. Many algebraic equivalences are easily captured with rewrite rules, such as commutativity of join:

```
join(R, S) --> join(S, R)
```

and the relationship between nested selections and logical conjunctions:

```
select(select(R, P), Q) --> select(R, and(P, Q))
```

4.1 Complications

While we believe term-rewriting is a promising approach, we foresee a number of challenges in extending it to a fully expressive system for all the kinds of transforms we anticipate. We enumerate some of the potential problem areas below, which we illustrate for the most part with examples from relational algebra.

4.1.1 Conditional Transforms

Some transforms may only apply if certain relationships hold among the arguments. Consider pushing selection through projection:

```
project(select(R, P), X) -->
        select(project(R, X), P)
```

This transform is only valid when the predicate `P` is "over" the attributes in `X`. We will either need an expressive pattern-matching mechanism or the ability to attach conditions to transforms to treat such cases.

4.1.2 Complex Transforms

The exact details of a transformation can get quite complicated, particularly for the auxiliary arguments to operators. Consider the following basic schema for pushing selection through join:

```
select(join(R, S), P) -->
        select(join(select(R, P1), select(S, P2)), P3)
```

It is not a simple matter to describe the decomposition of `P` into the conjunction of `P1`, `P2` and `P3` so that each mentions the appropriate attributes.

4.1.3 Approximate Transforms

There are transforms that give approximate but not exact equivalence, such as associativity of floating-point addition. In some cases exact numerical equivalence may not be required. However, it is difficult to imagine distinguishing permissible cases without the use of programmer-defined directives [12]. Other kinds of approximation come up in substituting value equality for identity, and vice versa [13].

4.1.4 Variadic Operations

To cut down on the size of the search space for optimization, it is useful to group multiple applications of a binary operator into a single application of a multiway operator. For example, for choosing join orders, one might want to collapse a group of adjacent joins to a single pseudo-operation

```
join(join(R1, R2), join(join(R3, R4), R5)) -->
        multijoin(R1, R2, R3, R4, R5)
```

and then proceed to partition the argument set back to binary joins in another order. One could possibly use list notation for arguments

```
join(R, multijoin(L)) --> multijoin(cons(R, L))
```

but that still leaves the problem of providing transforms to change list orderings. Some work on transforms in object algebras with variadic operators already exists [14].

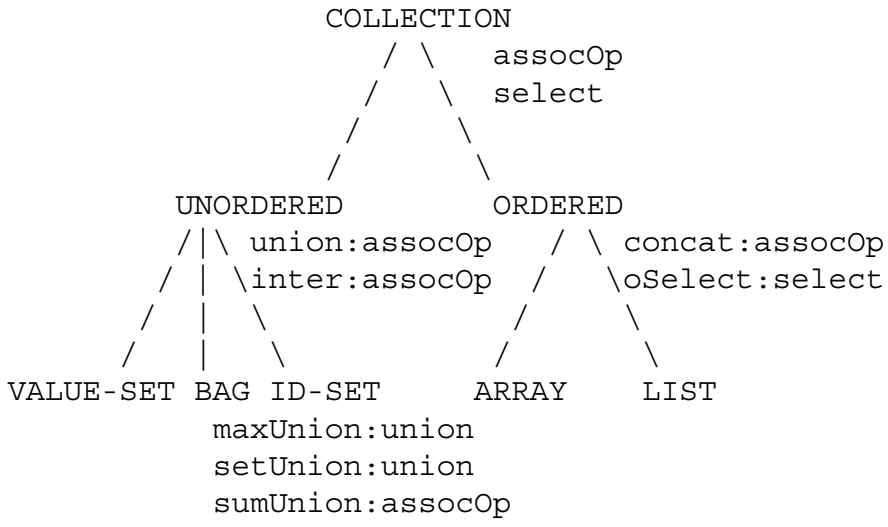
4.1.5 Non-Local Transforms

Some transforms seem to have a non-local character that is not easily expressed with term rewriting [15]. Here, relief might come from transformation grammar techniques used in natural language processing for specifying non-local transforms on sentential forms [16].

4.1.6 Exploiting Similarities

We expect that our object algebra will contain multiple bulk types, and it may well be that these bulk types have some similar operations (such as selection) with similar legal transforms. Having "duplicated" transforms in several bulk types probably is not a problem in increasing the size of the search space—the result type of an expression node will limit the choice to the instance of the transform for that type. However, if special heuristics or strategies are provided in the optimizer, those tactics might get repeated for each of several similar operations. For example, there might be a strategy that attempts to split predicates in selection conditions so one part can be pushed through the operation below the selection. That strategy likely applies equally well for a selection on bags, sets and lists, and we would rather not have to specify it three times.

An approach to explore here is to organize the bulk types into a "subalgebra" hierarchy [17], with abstractions of bulk types at the top, and concrete bulk types at the bottom, such as illustrated below:



The more abstract types could define operations and transforms that have one or more specializations at the lower levels. Heuristics or strategies can then be expressed in terms of the abstracted operators and applied to the concrete versions of those. Note that having transforms on the abstract operations does not prevent having additional transforms on the specialized versions. We think the work on comprehensions of bulk types [18, 19] is an excellent start on such an organization. It seems to capture the essence of some common operations on bulk types at an abstract level, though we expect some extensions will be needed for types such as multidimensional arrays.

Such an organization for bulk types could also accommodate user-defined bulk types. By inserting such a type at the appropriate point in the hierarchy, the query optimizer would be able to apply transforms to operations on that type.

4.2 Instance Parameters

We touch upon one final area in terms of specifying the logical algebra to the optimizer, and it is related to the specificity of the typing system for the data types in the algebra. There are tradeoffs in how specific the types are, between the amount of information one can develop statically about an expression versus how generically an expression will operate. For example, for multidimensional arrays, one might specify the ranges of the dimensions (`Array[50, 100]`), just the dimensionality (`Array[*,*]`) or just have a single type for all dimensions of arrays. In the context of database processing, given a particular type system, it is important to observe that the typing for an expression must hold over all executions of that expression, whereas there may be more detail that can be articulated on a specific execution. Consider an expression that aggregates a three-dimensional array across two of its dimensions:

```
reduce(+, reduce(+, M))
```

In a type system that captures dimensionality but not ranges of dimensions, the best that the type information can tell us is that the result is one-dimensional. However, for a particular execution of the expression where `M` has shape 800 by 120 by 200, the optimizer may want to know that the inner expression produces an 800 by 120 matrix and the result has size 800. (It would be unusual to analyze a single execution of an expression in the programming language setting, since it is doubtful the cost of the analysis would ever be recovered. However, in a database environment, the sizes of the arguments might be large and there could be a payoff from optimizing a single execution.)

We refer to the kinds of information that can describe data values beyond their types as *instance parameters*. The optimizer needs to know how to calculate the instance parameters of the results of algebraic expressions, given the values of those parameters on the inputs. We think abstract interpretation [20, 21] of the algebraic operators will be a good mechanism to capture this information. For example, abstracting lists to their lengths, the abstract interpretation of concatenation could be expressed by

```
LEN[concat(M, N)] = LEN[M] + LEN[N].
```

5 The Optimizer-Evaluator Interface

The query plan that the optimizer passes to the query evaluator represents the same function as the object algebra expression that was passed to the optimizer by the parser. However, the query plan embodies choices on particular physical procedures to implement each logical operator, on evaluation order of subexpressions and on additions of special procedures to modify physical properties of data. The principal information that the optimizer needs about this interface is the correspondence between logical algebra operators and the procedures in the evaluator that implement them. It also needs to know cost estimates for executing those procedures to use in the planning process. Good cost estimates in turn derive from knowing physical properties of procedure inputs.

5.1 The Logical-to-Physical Correspondence

Capturing the logical-operator-to-physical-procedure correspondence seems well handled by term-rewriting rules, even in the non-one-to-one case. For instance, one of the translations used in the example of Section 2.1 was

```
project(select(R, P), X) -->
        dup-elim(filter(R, P; X)).
```

The only unusual case is physical operators, such as `sort`, that are logical no-ops. The optimizer needs to know that they can be introduced into a plan at arbitrary points.

Again, the optimizer does not need to know precisely what function a physical procedure computes. It only needs to now how it relates to the logical algebra operators. (Of course, for a correctly functioning query processor, the physical procedures must conform to the definitions of the algebra operators.)

5.2 Cost Modeling and Physical Properties

In translating an algebra expression to a query plan, the query optimizer is attempting to minimize some costs, such as total CPU usage, elapsed time, memory usage or a combination of those. The basic input to the cost estimation process is statistics and layout information on the data objects at the leaves of the query plan. That information is obtained from the schema and catalog manager, which is treated in the next section. The rest of the process involves propagating this information through the procedures in the query plan to calculate the cost contribution of each. What we need is a behavioral model of each procedure that captures its costs as a function of properties of its inputs. To continue the process, we also need to compute or estimate properties for the output of the procedure. There is also the possibility that some procedures expect or require certain physical properties to hold on their inputs, and those requirements must be checked in producing a valid query plan. An example is the `merge-join` procedure in the query plan of Section 2.1. Thus, it is useful to classify procedures as to whether they preserve, enforce, require or destroy physical properties. Examples of such information are that selection preserves duplicates (or the absence thereof) and hash join destroys sort order.

One approach to capturing the behavior of physical procedures that we plan to investigate is to model the various costs and physical properties as abstract interpretations of the physical procedures. For example, suppose we have the following plan using nested-loops join:

```
nl-join(R(AB), S(AC))
```

where `R` is the outer relation and `S` is the inner relation. If `R` is sorted on `A` and `S` is sorted on `C`, then the resulting relation will be lexicographically sorted on `AC`.

To express how various operations interact with sort order as an abstract interpretation, we need a domain of sort orders and an interpretation function. Assume we are in the relational case, where sort orders can be described as sequences of attributes. Let **`SO[`**`P`**`]`** mean the sort-order interpretation of a plan `P`. The creator of new physical procedures would then be responsible for extending the abstract interpretation to that procedure. Consider

```
SO[nl-join[X, Y, Z](R(XY), S(YZ))] ==
        concat(SO[R], restrict(SO[S], X, Z))
```

Here the `restrict` operator computes what ordering exists on `Z`-values in `S` for tuples with equal `X`-values. To express that stream selection (as opposed to indexed selection) preserves sort order, we have

```
SO[stream-select(S, P)] == SO[S].
```

Some operators will destroy any sort order on their operands:

```
SO[hash-project(R, X)] == null-order
```

The abstract interpretation can then be used as input to cost functions on procedures and to express input requirements for procedures, for instance

```
merge-join[X, Y, Z](R(XY), R(XZ)) requires
        restrict(SO[R], empty, X) =
              restrict(SO[S], empty, X) and
        covers(SO[R], X)
```

which says that the sort order of R and S must imply equal sort orders on the attributes in X and they must be sorted on all those attributes.

As with using term rewriting for algebraic transforms, there are challenges in adapting abstract interpretation to cost and physical property modeling. We mention some of the problems here.

5.2.1 Defining the Domain for Interpretation

The basis of an abstract interpretation is a domain of interpretation. In some cases the domain will be simple, such as integers or Booleans for time cost or presence of duplicates. In other cases, such as describing the clustering of an object or sort orders on collections with complex objects, the domains need to be quite complex. Can the domain be general enough to capture user-defined values, such as a special sort ordering on a type. In addition, one may need to define new functions on the domain, such as `restrict` above, to express the behavior of a certain procedure. A final question is how abstract to make the domain for a particular property. For example, if we are interested in the distribution of values in a collection, we might capture that information as a max-median-min triple, as a histogram or even as a count of all values present. In making this choice it is important to remember that many of these interpretations are approximations, and there is no point in extending the precision of the domain much past the accuracy of the estimate.

5.2.2 Relationships on the Domain

In looking to satisfy physical requirements of a procedure in a query plan, the optimizer needs to know implications between domain values. For example, if some procedure on relations needs its input sorted on AB, then the optimizer should be able to conclude an ABC-sorted relation fulfills this requirement.

5.2.3 Interdependence of Properties

In traditional usage in programming languages and compilers, abstract interpretations are defined independently, for such purposes as typing and strictness analysis. For our purposes, some of the interpretations will depend on the value of others. For example, result size estimates might take into account the presence of duplicates, as might an interpretation on distribution of values.

5.2.4 Properties Versus Realizations

As we mentioned earlier, we anticipate multiple physical realizations of a given data structure. It may be that a given physical property does not apply to all realizations. For example, sort order might not apply for a set realized as a bit vector.

5.2.5 Parameter and Procedure Selection

The scenario we have sketched so far in conjunction with property computation and cost estimation is too simplistic. It is not always a case of generating a candidate query plan and then making various abstract interpretations of it. In order to limit search in the optimizer, it is useful to turn the situation around and ask if there is a particular combination of procedure and parameters that achieves a given cost or satisfies a given property. For example, a sort-based projection can accommodate a variety of output orderings based on a sort-order parameter to it. Different input orderings for merge join give different output orderings. The optimizer would like to ask about which parameter choices and input properties could produce a given output property, and in that way guide the generation of query plans for consideration. Going a step further in that direction, the optimizer might want to know whether any of the physical procedures corresponding to the logical join operator can deliver its result in a particular sort order. We have not yet decided on how to encode this information. It might be as search pragmas, as separate "property forcing" functions associated with the abstract interpretations, or as ancillary rules connected with each physical procedure.

6 Optimizer Interfaces to the Schema and Catalog

Unlike the interfaces discussed in Sections 4 and 5, which were internal to the query processor, the schema and catalog interface is an external interface to another module in the persistent object base. We should probably be writing in terms of multiple interfaces for the parser, optimizer and evaluator layers, but we have only begun to articulate the different information requirements of these levels. Here we mainly categorize the kinds of information that the query processor needs from the schema or catalog.

6.1 Definitions

We expect the schema to maintain definitions of user-defined types and implementations for those types. A type will be described in terms of the signatures of its operations. An implementation of a type will be expressed in terms of an internal representation of the instances of the type and methods for the operations of the type. In general, we wish to allow multiple implementations for a single type. It may be the schema manager's job as well to provide information about subtype relationships between types.

6.2 Bindings

We assume that part of the database schema is a name space where persistent identifiers reside. (In a relational database system, this name space holds the identifiers (names) of relations and view definitions.) For a given identifier, various kinds of information may be bound to it, such as a type, integrity constraints, an implementation, an object identifier, instance parameters and state. The query processor needs to inquire about these bindings for identifiers that show up in expressions it is processing.

A catalog or data directory might have further information on location and layout of data objects, as well as keep track of auxiliary access paths that exist. The query processor will ask about the existence of access paths, and needs a notation that expresses what each path is good for. Most auxiliary access paths in existing database systems are indexes or hash tables. However, we imagine that researchers will discover new kinds of structures, and the notation must be general enough to describe the utility of these. As an example of such a path, consider a collection `Albums` of record album objects, where every record algorithm has a distributor. Suppose the distributors are company objects

drawn from a collection `Companies`. There is no explicit collection of just those companies that distribute record albums in this model. It might be useful to maintain such a collection, however, for processing queries on `Albums`. That is, we want to maintain an auxiliary structure `Distributors`, defined as

```
{c in Companies|a in Albums, distributor(a) = c}
```

Then, for instance, given a query for all albums distributed by Canadian companies, the query processor could choose to iterate over `Distributors` rather than the likely much larger collections `Albums` and `Companies`.

6.3 Statistics

Another kind of information we expect the schema or catalog to contain is statistics on objects. A statistic in this context is usually a count or a value distribution, formed by aggregation or reduction of the data object. In the persistent object base setting, other useful abstractions of data may suggest themselves, such as the average fan-in or fan-out when objects reference other objects.

A key problem here is that statistics are derived from object instances, where the expression that the optimizer receives contains object names. If the name space treats identifiers as variables that may be reassigned arbitrarily, it will be hard for the query processor to obtain appropriate statistics at query optimization time. Note that in relational systems this problem usually does not arise, because each relation name is statically bound to a particular relation instance (though the contents of that instance can change).

6.4 Stability of Information

The kinds of information we discussed in Sections 4 and 5 that the optimizer needs about the object algebra and query plan language is relatively static. We anticipate that information being constant across many databases and over time within a given database system. On the other hand, the information that the query processor obtains from the schema or catalog is likely to change over the life of an application (and hence during the lifetime of some queries). The query processor and schema manager need some way to communicate about the stability of information. The optimizer either must avoid using information that might change during the lifetime of a query plan, or some service must record the appropriate dependencies so a plan can be marked out of date when information it incorporates has changed.

We are thinking about classifying schema and catalog information in terms of the kind of change that can invalidate it. The most volatile information changes because of a database update, such as modifying the state of an object, creating a new object or assigning to a persistent variable. Less volatile is information about physical organizations, which can change because of adding or deleting indexes, reclustering objects and computing new statistics, and so forth. Schema changes probably happen less frequently, with modifications such as adding an implementation or a type and changing a type interface.

One way of viewing the problem is to treat obtaining information from the schema and catalog as partial evaluation of an expression, and classify database changes as to how they affect the environment in which that evaluation takes place.

7 Other Interfaces

We briefly mention two other interfaces for which we have done much less analysis. One is the connection of the query processor to a host programming language. There might be little difficulty if the persistent object base is actually a persistent programming language with a type system matching that of the object algebra. In other cases the host language will be non-persistent or not support bulk types directly. The most challenging situation is where some mapping of type systems and values must take place between the query processor and the programming language. We would then need notation to specify the target formats of values in the programming language. IDL [22] is a possible starting point here. If a query can contain embedded fragments from the host language, what is needed to make the linkage to execute such code during query evaluation? Is the source code translated into a form that can execute directly on the database format of objects or will the database make "up-calls" to the application? In the latter case, must the database reformat objects into the form expected by the programming language (for example, changing object IDs to memory addresses or converting strings from start-offset format to self-delimiting format) before passing them up as arguments?

The other interface is to the status of computational capabilities of the environment in which the query processor is running. Some of those factors may be fairly constant, such as the number of processors and communications bandwidth. Others may change more frequently, such as memory available for buffers, processor loads and unused data space. We need a mechanism for describing resources and a methodology for factoring information about those resources into the cost model for query plans.

8 Remarks and Future Work

We have presented our ideas on how to articulate various internal and external interfaces for a query processor in a persistent object system. It is perhaps important to mention two interfaces that we do not intend to specify right away, namely the external query language and the interface to the underlying persistent object store. Our reasons are twofold. One is that we do not see as much similarity at those boundaries among the prototypes of the participating groups, and we think the parser and evaluator can insulate the query optimizer from them in any case. The second reason is that we do not want to preempt other ongoing efforts to gain consensus on module boundaries in persistent object systems [23].

Most of the ideas presented herein are as yet unsullied by the realities of implementation attempts. In the coming year we will make our initial essay at defining the interfaces of the query processor, and then try to adapt the prototype query processors of the various participants in the EREQ project to those interfaces. We will then be able to attempt an optimizer transplant. In all likelihood the heart will reject the body, but we should have enough feedback to refine our interfaces so that the next attempt at the operation is successful.

9 Acknowledgements

Most of these ideas grew out of many discussions with other members of the Revelation group—Scott Daniels, Goetz Graefe, Tom Keller and Bennet Vance—and have been influenced by the work of other members of the EREQ project—Mike Carey, Dave DeWitt, Gail Mitchell, Scott Vandenberg and Stan Zdonik. This work was partially supported by NSF grant IRI 8920642.

References

[1] Zdonik, SB and Mitchell, G. ENCORE: An object-oriented approach to database modeling and querying. *Data Engineering* 14(2), June 1991.

[2] Daniels, S, Graefe, G, Keller, T, et al. Query optimization in Revelation, an overview. *Data Engineering* 14(2), June 1991.

[3] Vandenberg, SL and DeWitt, DJ. Algebraic query processing in EXTRA/EXCESS. *Data Engineering* 14(2), June 1991.

[4] Elmasri, R and Navathe, SB. *Fundamentals of Database Systems*, Benjamin/Cummings, 1989.

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

[6] Ullman, JD. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, 1988.

[7] Buneman, OP. Data types for database programming. Proc. of the Appin Conf. on Data Types and Persistence: 295-308, Dept. of Computing Science report PPRR 16, Univ. of Glasgow, August 1985.

[8] Beeri, C and Kornatsky, Y. Algebraic optimization of object-oriented query languages. *Proc. Third Intl. Conf. on Database Theory*:72-88, Springer-Verlag LNCS 470, Paris, December 1990.

[9] Osborn, SL. Identity, equality and query optimization. *Advances in Object-Oriented Database Systems:*346-351, Springer-Verlag LNCS 334, October 1988.

[10] Vance, B. Toward an object-oriented query algebra. Computer Science & Engineering report 91-08, Oregon Graduate Inst., May 1991.

[11] Vandenberg, SL and DeWitt, DJ. Algebraic support for complex objects with arrays, identity and inheritance. *Proc. 1991 ACM SIGMOD Intl. Conf. on Management of Data:*158-167, Denver, May 1991.

[12] Lieuwen, DF and DeWitt, DJ. Optimizing loops in database programming languages. To appear, *Proc. of the Third Intl. Workshop on Database Programming Languages*, Morgan Kaufmann, 1991.

[13] Shaw, GM and Zdonik, SB. Object-oriented queries: Equivalence and optimization. In *Deductive and Object-Oriented Databases:*264-278, Elsevier Science Publishers, 1990.

[14] Straube, DD and Özsu, MT. Queries and query processing in object-oriented database systems. *ACM Trans. on Information Systems* 8(4):387-430, October 1990.

[15] Mitchell, G, Zdonik, SB and Dayal, U. An architecture for query processing in persistent object stores. Dept. of Computer Science report 91-38, Brown Univ., June 1991.

[16] Winograd, T. *Language as a Cognitive Process, volume 1: Syntax*, Addison-Wesley, 1983.

[17] Wegner, P and Zdonik, SB. Models of Inheritance. In *Database Programming Languages: Proceedings of the Second Intl. Workshop*, Morgan Kaufmann, 1989.

[18] Trinder, P. Comprehensions, a query notation for DBPLs. To appear, *Proc. of the Third Intl. Workshop on Database Programming Languages*, Morgan Kaufmann, 1991.

[19] Wadler, PL. Comprehending monads. *Proc. ACM Conf. on Lisp and Functional Programming:*61-78, Nice, France, June 1990.

[20] Abramsky, S and Hankin, C (eds.). *Abstract Interpretation of Declarative Languages*, Ellis Horwood, 1987.

[21] Mycroft, A. *Abstract Interpretation and Optimizing Transformations for Applicative Programs*, PhD Thesis, Univ. of Edinburgh, 1981.

[22] Snodgrass, R. *The Interface Description Language: Definition and Use*, Computer Science Press, 1989.

[23] Thatte, SM. A modular and open object-oriented database system. *SIGMOD Record* 20(1):47-52, March 1991.