An Algebraic Framework for Physical OODB Design

Leonidas Fegaras David Maier
Department of Computer Science and Engineering
Oregon Graduate Institute of Science & Technology
20000 N.W. Walker Road P.O. Box 91000
Portland, OR 97291-1000, USA
email: {*fegaras,maier*}*@cse.ogi.edu*

Abstract
Physical design for object-oriented databases is still in its infancy. Implementation decisions often intrude into the conceptual design (such as inverse links and object decomposition). Furthermore, query optimizers do not always take full advantage of physical design information. This paper proposes a formal framework for physical database design that automates the query translation process. In this framework, the physical database design is specified in a declarative manner. This specification is used for generating an efficient query transformer that translates logical queries into programs that manipulate the physical database. Alternative access paths to physical data are captured as simple rewrite rules that are used for generating alternative plans for a query.

1 Introduction

One important advantage that commercial database systems offer is data independence, whereby abstract objects and the operations upon them can be significantly decoupled from their implementations. In a relational database system, for example, a database designer may choose the implementation of a database table from a number of possible structures (such as a B-tree or a hash table) as well as attach secondary indices to the table. These implementation decisions will not affect how queries are expressed in the database language but only how they are compiled and optimized. Furthermore, some systems provide a restructuring mechanism to change the implementation of parts of the database or to modify the database schema itself without losing any stored data.

Physical design for object-oriented databases is more difficult than for relational systems because the complexity of object-oriented database (OODB) data models results in a larger number of implementation choices. The database designer may consider clustering versus normalization for various nested collections in the database, create inverse links, attach secondary indices, materialize functions and views, partition large objects, etc. [15, 14, 23, 4]. It is highly desirable to have these choices isolated from the conceptual model itself, leaving the application programmer to worry only about *what* data to retrieve, not *how* to retrieve the data. Achieving the same degree of data independence in an OODB system as in a relational database system is a major challenge for object-oriented databases.

This paper presents a framework for specifying the physical design in a declarative language, called the *physical design language*. It consists of a small, but extensible, repertoire of commands (called *physical design directives*) for specifying the implementation techniques for various parts of a database. For example, one command may indicate that a specific nested collection be normalized (flattened out) into two collections. The query translator uses these commands to translate queries against the conceptual database into queries against the physical database. If normalization was chosen for a nested collection, then a logical query that manipulates this nested collection may be translated into a query that joins the two normalized collections. The physical design language described in this paper captures most of the recent proposals for OODB physical designs, including clustering, horizontal and vertical partitioning, normalization, join indices, and multiple access paths via secondary indices. Expressing a physical design as a set of independent directives simplifies the physical design process.

The query translation process in our framework consists of several stages. First, the database administrator specifies the conceptual database schema. The main concern of this person is to write functionally correct specification satisfying all the design requirements. Then, the database implementor specifies the physical design in such a way that the performance of the resulting system is acceptable for the needs of this application. This person is also responsible for tuning the database to cope with new performance requirements. Finally, the application programmer submits a logical query against the database without any knowledge of the physical design. The query translator translates the query into a physical plan that reflects the physical design and ideally runs faster than any other equivalent plan. The query evaluator executes this plan and returns the result to the application programmer.

Query translation in our framework is purely algebraic and can be easily validated for correctness. In our framework, the physical database design has an internal schema that specifies the structure of the internal database state, an *abstraction function* [11] that maps the internal schema into the conceptual schema, and a set of constraints that capture the alternative access paths (such as secondary indices, materialized functions and views). The abstraction function is a logical view of the physical database. This function always exists, since otherwise there would be some semantic information lost when the conceptual database is mapped into the physical storage. Given the conceptual schema of an OODB and a set of physical design directives, we have an automated method for generating the physical schema, the abstraction function, and the plan transformers (this is the *optimizer generation* component in Figure 1). This method is the focus of the paper. It is expressed in rule form, requiring only one rule per physical design directive, and allows extensions to more complex physical design methods.

Our physical design framework requires that both conceptual and physical data structures, as well as the operations upon them, be defined in the same language. The language used in this paper is called the *monoid comprehension calculus* [9, 10] because it is based on monoids and monoid comprehensions. Logical collection types, such as sets, lists, and bags, as well as physical data types, such as B-trees and hash tables, can be captured as monoids.

Logical queries are equivalent to queries against the conceptual database built from the internal database via the abstraction function. That is, any logical query can be transformed to a program that manipulates the physical database if we replace all references to the conceptual database state in the query with the logical view of the physical database state. The query translation process in our framework consists of substituting $\mathcal{R}(DB)$ for all occurrences of db in a logical query and normalizing the resulting program, where db is the conceptual database state, DB is the physical database state, and $\mathcal{R}$ is the abstraction function (this is the *composition* component in Figure 1).

We give a *normalization algorithm* that removes all the unnecessary intermediate logical structures, in such a way that the resulting normalized program does not actually materialize any part of the conceptual database. The resulting program (the *physical plan* in Figure 1) is thus a query that directly manipulates the physical database. That is, if the abstraction function is expressed in the monoid calculus, then any query in the monoid calculus that manipulates the conceptual database can be efficiently translated into a query that manipulates only the physical database. Even though the abstraction function builds the entire conceptual database from the physical database, no part of this construction will actually take place if we normalize the resulting query. The normalization algorithm is purely algebraic, simple, and efficient.

Access path selection is achieved by substituting $\mathcal{C}_i(DB)$ for DB in the derived physical plan, where $\mathcal{C}_i$ is a plan transformer, and then normalizing the resulting program (this step is the *plan generation* component in Figure 1). This phase can be combined with the application of commutativity and associativity rules for monoid comprehensions. There is no need of using a rewrite system for these transformations, since we only use three types of rules: an application of a plan transformer, associativity, and commutativity. In fact an optimizer based on dynamic programming, such as the one for System R [19], would be sufficient for our purpose. In that case, the *costing* component in Figure 1 could be combined with the *plan generation* component.

In addition to query translation, in this paper we report an automated method for translating database updates against the conceptual database state into updates against the physical database.

The contributions of this paper are twofold. First, we present a declarative language for specifying physical design directives for an OODB management system that captures many recent proposals for OODB physical design. Second, we present a method for translating these directives into a form that facilitates an automated translation of logical queries and updates. The program translation as well as the elimination of the intermediate logical structures in the resulting program is based on a formal model.

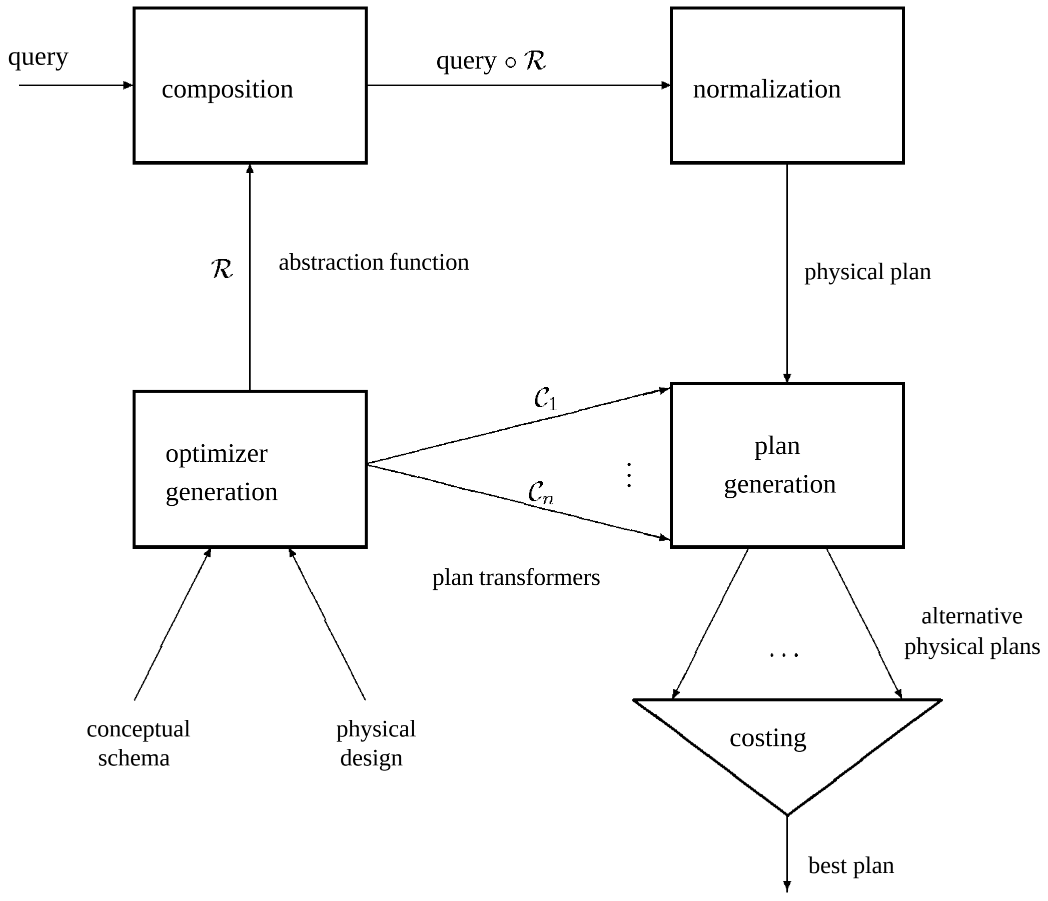

Figure 1: The Query Translation Architecture

2 Background

Queries in our framework are transformed into physical plans by a number of refinement steps. Thus, they need to be compiled into an algebraic form that captures both logical and physical operators. More importantly, the algebraic forms derived after query translation need to be normalized in a way that no intermediate logical structures are constructed during the evaluation of these forms. In this section we give a brief overview of the monoid comprehension calculus, which fulfills these two requirements. For a complete formal description of the calculus, which includes advanced data structures such as vectors, matrices and object identity, the reader is referred to our previous work [9, 10].

2.1 The Monoid Comprehension Calculus

A data type T in our calculus is expressed as a monoid $\mathcal{M}$ with a unit function:

$$\mathcal{M} = (T, \text{zero}, \text{unit}, \text{merge})$$

where the function merge, of type $T \times T \rightarrow T$, is associative with left and right identity zero. If in addition merge is commutative (idempotent, i.e., $\forall x : \text{merge}(x, x) = x$), then the monoid is commutative (idempotent). For example, $(\text{set}(\alpha), \{\}, f, \cup)$, where $f(x) = \{x\}$, is a commutative and idempotent monoid while $(\text{int}, 0, g, +)$, where $g(x) = x$, is a commutative monoid. When necessary to distinguish the components of a particular monoid $\mathcal{M}$ we qualify them as $\text{zero}^{\mathcal{M}}$, $\text{unit}^{\mathcal{M}}$, and $\text{merge}^{\mathcal{M}}$.

$\mathcal{M}$	T	zero	unit(a)	merge	C/I
list	list(α)	$[\,]$	$[a]$	$+\!\!+$	
set	set(α)	$\{\}$	$\{a\}$	$\cup$	CI
bag	bag(α)	$\{\!\{\}\!\}$	$\{\!\{a\}\!\}$	$\uplus$	C
sorted[f]	list(α)	$[\,]$	$[a]$	merge[f]	CI

$\mathcal{M}$	T	zero	unit(a)	merge	C/I
sum	int	0	a	$+$	C
max	int	0	a	max	CI
some	bool	false	a	$\vee$	CI
all	bool	true	a	$\wedge$	CI

Table 1: Examples of Collection and Primitive Monoids

We have two types of monoids: *collection* and *primitive monoids*. Collection monoids capture bulk data types, while primitive monoids capture primitive types, such as integers and booleans. Table 1 presents some examples of collection and primitive monoids. The C/I column indicates whether the monoid is a commutative or idempotent monoid. The monoids *list*, *bag*, and *set* capture the well-known collection types for linear lists, multisets, and sets [7] (where $+\!\!+$ is list append and $\uplus$ is the additive union for bags). The monoid *sorted*[f] is parameterized by the function f whose range is associated with a partial order $\leq$. The merge function of this monoid merges two sorted lists into a sorted list. If x appears before y in a *sorted*[f] list, then $f(x) \leq f(y)$.

In our treatment of queries we will consider only monoid types as valid types. A *monoid type* has one of the following forms:

class_name	*(a reference to a class)*
T	*(T is a primitive type, such as* int *and* bool*)*
$T(type)$	*(T is a type constructor, such as* set, bag, *and* list*)*
$\langle\, A_1 : t_1, \ldots, A_n : t_n \,\rangle$	*(a record type)*

where $type$ and $t_1, \ldots, t_n$ are monoid types and T is a monoid. That is, collection types can be freely nested.

A *monoid comprehension* over the monoid $\mathcal{M}$ takes the form $\mathcal{M}\{\, e \mid \overline{r} \,\}$. Expression e is called the *head* of the comprehension. Each term r_i in the term sequence $\overline{r} = r_1, \ldots, r_n,\ n \geq 0$, is called a *qualifier*, and is either

- a *generator* of the form $v \leftarrow e'$, where v is a variable and e' is an expression, or
- a *filter* p, where p is a predicate.

The scope of the variable v in $\mathcal{M}\{\, e \mid \overline{e_1},\, v \leftarrow e',\, \overline{r_2} \,\}$ is limited to the rest of the comprehension, $\overline{r_2}$, and to the head of the comprehension, e. Like in most modern programming languages, the scope is textual and we have the typical scoping rules for name conflicts: e.g., the scope of the left v in $\mathcal{M}\{\, e \mid \overline{e_1},\, v \leftarrow e',\, \overline{r_2},\, v \leftarrow e'',\, \overline{r_3} \,\}$ is $\overline{r_2}$ and e'', while the scope of the right v is $\overline{r_3}$ and e.

For example, the join of two sets x and y, join$(f, p)(x, y)$, is

$$set\{\, f(a, b) \mid a \leftarrow x,\, b \leftarrow y,\, p(a, b) \,\}$$

where p is the join predicate and function f constructs an output set element given two elements from x and y. For example, if p(a,b) = (a.C=b.C) $\wedge$ (a.D $>$ 10) and f(a,b) = $\langle$ C=a.C, D=b.D $\rangle$, then this comprehension becomes:

$$set\{\ \langle\ \text{C=a.C, D=b.D}\ \rangle \ \mid\ \text{a} \leftarrow \text{x, b} \leftarrow \text{y, a.C=b.C, a.D} > 10\ \}.$$

A monoid comprehension is defined by the following reduction rules: (A formal definition based on monoid homomorphisms is presented elsewhere [10].)

$$\mathcal{M}\{\, e \mid \,\} \;\rightarrow\; \text{unit}^{\mathcal{M}}(e) \quad (1)$$

$$\mathcal{M}\{\, e \mid \text{false},\, \overline{r} \,\} \;\rightarrow\; \text{zero}^{\mathcal{M}} \quad (2)$$

$$\mathcal{M}\{\, e \mid \text{true},\, \overline{r} \,\} \;\rightarrow\; \mathcal{M}\{\, e \mid \overline{r} \,\} \quad (3)$$

$$\mathcal{M}\{\, e \mid v \leftarrow \text{zero}^{\mathcal{N}},\, \overline{r} \,\} \;\rightarrow\; \text{zero}^{\mathcal{M}} \quad (4)$$

$$\mathcal{M}\{\, e \mid v \leftarrow \text{unit}^{\mathcal{N}}(e'),\, \overline{r} \,\} \;\rightarrow\; \textbf{let}\, v = e' \,\textbf{in}\, \mathcal{M}\{\, e \mid \overline{r} \,\} \quad (5)$$

$$\mathcal{M}\{\, e \mid v \leftarrow \text{merge}^{\mathcal{N}}(e_1, e_2),\, \overline{r} \,\} \;\rightarrow\; \text{merge}^{\mathcal{M}}(\, \mathcal{M}\{\, e \mid v \leftarrow e_1,\, \overline{r} \,\},\, \mathcal{M}\{\, e \mid v \leftarrow e_2,\, \overline{r} \,\}\,) \quad (6)$$

Rules 2 and 3 reduce a comprehension in which the leftmost qualifier is a filter, while rules 4-6 reduce a comprehension in which the leftmost qualifier is a generator.

This definition of a comprehension provides an equational theory that allows us to prove the soundness of various transformations, including the translation of comprehensions into efficient joins.

The monoid comprehension is the only form of bulk manipulation of collection types supported in our calculus. But monoid comprehensions are very expressive. In fact, a small subset of these forms, namely the monoid comprehensions from sets to sets, captures precisely the nested relational algebra (since they are equivalent to the set monad comprehensions [6]). For example, the nesting operator for nested relations is

$$\text{nest}(k)\,x \;=\; set\{\, \langle\, \text{KEY} = k(e),\, \text{P} = set\{\, a \mid a \leftarrow x,\, k(e) = k(a)\,\}\,\rangle \mid e \leftarrow x\,\}$$

Similarly, the unnesting operator is

$$\text{unnest}(x) \;=\; set\{\, e \mid s \leftarrow x,\, e \leftarrow s.\text{P}\,\}$$

The last comprehension is an example of a *dependent join* in which the value of the second collection s.P depends on the value of s, an element of the first relation x. Dependent joins are a convenient way of traversing nested collections.

But monoid comprehensions go beyond the nested relational algebra to capture operations over multiple collection types, such as the join of a list with a bag that returns a set, plus predicates and aggregates. For example,

$$set\{\, (x, y) \mid x \leftarrow [1, 2],\, y \leftarrow \{\!\{3, 4, 3\}\!\}\,\} \;=\; \{(1, 3), (1, 4), (2, 3), (2, 4)\}$$

Another example is $sum\{\, a \mid a \leftarrow [1, 2, 3],\, a \geq 2\,\}$, which returns 5, the sum of all list elements greater than or equal to 2. They can also capture physical algorithms, such as the merge join:

$$sorted[f]\{\, a \mid a \leftarrow x,\, b \leftarrow y,\, f(a) = g(b)\,\}$$

where x is an instance of a sorted$[f]$ monoid and y of a sorted$[g]$ monoid (f and g are not necessarily the same). That is, this comprehension behaves exactly like a merge-join: it receives two sorted lists as input and it generates a sorted list as output. Even though the naive interpretation of this program derived from the comprehension definition (Rules 1 through 6) is quadratic, we will see later that there are some effective ways of assigning specialized execution algorithms to these programs. In that case, the program will be a real merge join. This assignment to efficient execution algorithms is possible by examining the types of the generator domains in a comprehension.

The following are some more examples of comprehensions:

$$\begin{array}{lcllcl}
\text{filter}(p)(x) & = & set\{\, e \mid e \leftarrow x,\, p(e)\,\} & \text{flatten}(x) & = & set\{\, e \mid s \leftarrow x,\, e \leftarrow s\,\} \\
x \cap y & = & set\{\, e \mid e \leftarrow x,\, e \in y\,\} & \text{length}(x) & = & sum\{\, 1 \mid e \leftarrow x\,\} \\
\text{sum}(x) & = & sum\{\, e \mid e \leftarrow x\,\} & \text{count}(x, a) & = & sum\{\, 1 \mid e \leftarrow x,\, e = a\,\} \\
\exists a \in x : e & = & some\{\, e \mid a \leftarrow x\,\} & \forall a \in x : e & = & all\{\, e \mid a \leftarrow x\,\} \\
a \in x & = & some\{\, a = e \mid e \leftarrow x\,\} & & &
\end{array}$$

The expression sum(x) adds the elements of any non-idempotent monoid x, e.g., sum$([1, 2, 3]) = 6$. The expression count(x, a) counts the number of occurrences of a in the bag x, e.g., count$(\{\!\{1, 2, 1\}\!\}, 1) = 2$.

The calculus has a semantic well-formedness requirement that a comprehension be over an idempotent or commutative monoid if any of its generators are over idempotent or commutative monoids. For example, $list\{\, x \mid x \leftarrow \{1, 2\}\,\}$ is not a valid monoid comprehension, since it maps a set (which is both commutative and idempotent) to a $list$ (which is neither commutative nor idempotent), while $sum\{\, x \mid x \leftarrow \{\!\{1, 2\}\!\}\,\}$ is valid (since both bag and sum are commutative). This requirement can be easily checked during compile time [9].

We will use the following convention to represent variable bindings in a comprehension:

$$\mathcal{M}\{\, e \mid \overline{r},\, x \equiv u,\, \overline{s}\,\} \;\longrightarrow\; \mathcal{M}\{\, e[u/x] \mid \overline{r},\, \overline{s}[u/x]\,\} \tag{7}$$

where $e[u/x]$ is the expression e with u substituted for all the free occurrences of x (i.e., $e[u/x]$ is equivalent to **let** $x = u$ **in** e). A term of the form $x \equiv u$ is called a *binding* since it binds the variable x to the expression u. For example,

$$set\{\; b.D \;\mid\; a \leftarrow x, b \equiv y, a.B = b.C \;\} \;=\; set\{\; y.D \;\mid\; a \leftarrow x, a.B = y.C \;\}$$

2.2 Program Normalization

The monoid calculus can be put into a canonical form by an efficient rewrite algorithm, called the *normalization algorithm* (described in detail elsewhere [10]). The evaluation of these canonical forms generally produces fewer intermediate data structures than the initial unnormalized programs. Moreover, the normalization algorithm improves program performance in many cases. The normalization algorithm will be used as a prephase to our query evaluator since canonical forms are a convenient program representation that facilitate program transformation. The physical design framework described in Section 3 uses this algorithm to eliminate value coercions introduced when mapping logical queries into physical programs.

The normalization algorithm is a pattern-based rewriting algorithm. One example of a rewriting rule that this algorithm uses is unnesting nested comprehensions (i.e., comprehensions that contain a generator whose domain is another comprehension):

$$\mathcal{M}\{\, e \mid \overline{r},\, v \leftarrow \mathcal{N}\{\, e' \mid \overline{t}\,\},\, \overline{s}\,\} \quad\longrightarrow\quad \mathcal{M}\{\, e \mid \overline{r},\, \overline{t},\, v \equiv e',\, \overline{s}\,\} \tag{8}$$

Rules 7 and 8 are the most complex rules of the normalization algorithm. The other rules include trivial reductions, such as a projection over a tuple construction results into a tuple component. Rule 8 may require some variable renaming to avoid name conflicts. The following is an example of a program normalization that requires variable renaming. The program filter(p)(filter(q) x) is computed by

set{ a | a ← *set*{ a | a ← x, q(a) }, p(a) }
= *set*{ a | a ← *set*{ b | b ← x, q(b) }, p(a) }

(by renaming variable a to b) and is normalized into

⟶ *set*{ a | b ← x, q(b), a ≡ b, p(a) } *(by Rule 8)*
⟶ *set*{ b | b ← x, q(b), p(b) } *(by Rule 7)*

A path $path$ is a $name$ (the identifier of a bound variable, or the identifier of a persistent variable, or the name of a class extent) or an expression $path'.name$ (where $name$ is an attribute name of a record and $path'$ is a path). If the generator domains in a comprehension (i.e., expressions e in $v \leftarrow e$) do not contain any non-commutative merges (such as the list append), then these domains can be normalized into paths [10]. In the next section we will use the following shorthand: A *path expression* (as it is defined in [12]) is an expression of the form $db.pth_1.pth_2 \ldots .pth_{n+1}$, where each pth_i is a path and db is the conceptual database state, and whose interpretation in our calculus is

$$set\{\, v_n.pth_{n+1} \mid v_1 \leftarrow db.pth_1,\, v_2 \leftarrow v_1.pth_2,\, \ldots,\, v_n \leftarrow v_{n-1}.pth_n\,\}$$

In addition to the normalization rules, there are other important program transformations that explore the commutativity properties of monoids. In particular, if $\mathcal{M}$ is a commutative monoid, then we have the following join commutativity rule:

$$\mathcal{M}\{\, e \mid \overline{r},\, v_1 \leftarrow e_1,\, v_2 \leftarrow e_2,\, \overline{s}\,\} \;\longrightarrow\; \mathcal{M}\{\, e \mid \overline{r},\, v_2 \leftarrow e_2,\, v_1 \leftarrow e_1,\, \overline{s}\,\}$$

which holds only when term e_2 does not depend on v_1. The following transformation, which is valid for any monoid $\mathcal{M}$, pushes a selection before a join if $pred$ does not depend on v:

$$\mathcal{M}\{\, e \mid \overline{r},\, v \leftarrow e_1,\, pred,\, \overline{s}\,\} \;\longrightarrow\; \mathcal{M}\{\, e \mid \overline{r},\, pred,\, v \leftarrow e_1,\, \overline{s}\,\}$$

3 Physical Design

In this section we show how to translate queries against the conceptual database into queries against the physical database in a way that reflects a user-specified physical design. The translation process is described through examples that illustrate the basic idea. The physical design language is presented in Section 4 while the rules for generating the query translator from a physical design are presented in Section 5. In the first example we normalize a nested relation. We intentionally kept this example simple so that one can easily express the abstraction function and the plan transformers by simply observing the conceptual and the physical schema. These observations will help us understand how these programs are generated automatically by the optimizer-generation component of our translator. We use these programs to translate a logical query into a physical plan and to derive alternative plans. The second example is more complex. It is based on a conceptual OODB schema with a complex physical design. The purpose of this example is to support our claim that the same theory can be easily scaled up to capture more complex designs.

3.1 Example 1: Mapping Nested Relations into Flat Relations

Consider the following NF^2 conceptual database schema:

$$\text{db: set}(\langle\ \text{A: int, B: set}(\langle\ \text{C: int, D: int}\ \rangle),\ \text{E: int}\ \rangle)$$

Suppose that we want to implement this schema using flat table structures. The standard approach is to normalize the nested collection into two tables T1 and T2: table T1 holds the outer set while table T2 holds the union of all the inner sets. Then, whenever a query manipulates the initial nested collection, this nested collection is reconstructed via an implicit join. Furthermore, suppose that we want to implement the set as a B-tree indexed by A and we want to add a secondary index (also implemented as a B-tree) indexed by E. Using our physical design language (that will be described in detail in Section 4), this specification is expressed by the following physical design directives:

```
directives = { implement( db, sorted[A] ),        (1)
               normalize( db.B ),                 (2)
               secondary( db, E ) }               (3)
```

Directive *(1)* indicates that the outer set be implemented as a B-tree indexed by A. Directive *(2)* indicates that the nested set (reached by the path expression db.B) be normalized. Directive *(3)* indicates that there will be a secondary index attached to the outer set. One possible internal (physical) schema that captures this design is the following:

```
DB: ⟨ T1: sorted[A](⟨ A: int, B: ⟨⟩, E: int ⟩),
      T2: sorted[#](⟨ #: TID, INFO: ⟨ C: int, D: int ⟩ ⟩),
      T3: sorted[E](⟨ #: TID, E: int ⟩) ⟩
```

where $\langle\rangle$ is the empty record, which indicates that the B attribute in T1 is of no interest, since the inner set in the conceptual database is normalized into T2. Each record in the physical schema is associated with a *tuple identifier* (of type TID) that holds the actual location of this record on disk. The tuple identifier of a record x is accessed by $@x$. The # attributes in T2 and T3 hold tuple identifiers. Sequence T1 is implemented as a sequence sorted by A, that is, $\forall x, y \in \mathsf{T1} : @x \leq @y \Rightarrow x.\mathsf{A} \leq y.\mathsf{A}$. A similar equation holds for the secondary index T3. Sequence T2 is indexed by the # attribute, that is, $\forall x, y \in \mathsf{T2} : @x \leq @y \Rightarrow x.\# \leq y.\#$. If $x \in \mathsf{T2}$ is a child of $y \in \mathsf{T1}$, then $x.\# = @y$. The inner set of the conceptual database is implemented as a sorted[#] sequence so that the join between T1 and T2 over the join predicate $x.\# = @y$, which reconstructs the nested set, can be performed as a merge join. Similarly, for each $x \in \mathsf{T1}$ there is $y \in \mathsf{T3}$ such that $y.\# = @x$ and $y.\mathsf{E} = x.\mathsf{E}$.

Let $\mathcal{R}$ be the abstraction function that maps the physical schema DBtype to the conceptual schema dbtype. That is, if db of type dbtype is the database state as a user sees it and DB of type DBtype is the actual database state as it is stored on disk, then $\mathsf{db} = \mathcal{R}(\mathsf{DB})$. For our example, we have:

$$\begin{aligned}\mathcal{R}(\text{DB}) = \textit{set}\{\ \langle\ & \text{A = a.A},\\ & \text{B} = \textit{set}\{\ \langle\ \text{C=b.INFO.C, D=b.INFO.D}\ \rangle\ \mid\ \text{b} \leftarrow \text{DB.T2, b.\#=@a}\ \},\\ & \text{E = a.E}\ \rangle\\ \mid\ & \text{a} \leftarrow \text{DB.T1}\ \}\end{aligned}$$

In addition, there is a relationship between the table T1 and its secondary index T3. This relationship can be captured by the function $\mathcal{C}$ (a plan transformer), which represents a referential integrity constraint on the physical schema:

$$\begin{aligned}\mathcal{C}(\text{DB}) = \langle\ & \text{@=@DB},\\ & \text{T1} = \textit{sorted}[\text{A}]\{\ \langle\ \text{@=@a, A=a.A, B=a.B, E=b.E}\ \rangle\\ & \qquad\qquad\mid\ \text{a} \leftarrow \text{DB.T1, b} \leftarrow \text{DB.T3, b.\#=@a}\ \},\\ & \text{T2 = DB.T2, T3 = DB.T3}\ \rangle\end{aligned}$$

The equation $\mathcal{C}(\mathsf{DB}) = \mathsf{DB}$ is true for any database instance DB because of the information redundancy introduced by the secondary index. This equation indicates that the values stored in table T1 can also be retrieved by joining T1 with T3. That is, if a tuple b of the secondary index T3 is located (e.g., by providing the value b.E), then the associated tuple a of T1 is located by the equijoin. The tuple identifier @ of the resulting tuples in T1 is set to @a so that the tuples in T1 have the same tuple identifiers as those generated by the comprehension. That is, the TID @ is handled as a record attribute, even though it does not occupy any physical space. This function makes the tuple identifiers of all the records in DB equal to the tuple identifiers generated by the expression in the $\mathcal{C}$ definition.

An abstract query is a function f over the conceptual database db. For example:

$$f(\mathsf{db}) = sum\{\ \mathsf{y.C}\ \mid\ \mathsf{x} \leftarrow \mathsf{db},\ \mathsf{y} \leftarrow \mathsf{x.B},\ \mathsf{x.A{=}10},\ \mathsf{y.D} > 5\ \}$$

The implementation of $f(\mathsf{db})$ is $F(\mathsf{DB}) = f(\mathcal{R}(\mathsf{DB}))$:

$$\begin{aligned}
F(\mathsf{DB}) &= sum\{\ \mathsf{y.C}\ \mid\ \mathsf{x} \leftarrow \mathcal{R}(\mathsf{DB}),\ \mathsf{y} \leftarrow \mathsf{x.B},\ \mathsf{x.A{=}10},\ \mathsf{y.D} > 5\ \}\\
&= sum\{\ \mathsf{y.C}\ \mid\ \mathsf{x} \leftarrow set\{\ \langle\ \mathsf{A{=}a.A},\ \mathsf{B{=}}set\{\ \langle\ \mathsf{C{=}b.INFO.C},\ \mathsf{D{=}b.INFO.D}\ \rangle\\
&\qquad\qquad\qquad\qquad\qquad \mid\ \mathsf{b} \leftarrow \mathsf{DB.T2},\ \mathsf{b.\#{=}@a}\ \},\ \mathsf{E{=}a.E}\ \rangle\\
&\qquad\qquad\qquad \mid\ \mathsf{a} \leftarrow \mathsf{DB.T1}\ \},\\
&\qquad\qquad \mathsf{y} \leftarrow \mathsf{x.B},\ \mathsf{x.A{=}10},\ \mathsf{y.D} > 5\ \}
\end{aligned}$$

If we normalize this expression using our normalization algorithm, we get:

$$\begin{aligned}
&\longrightarrow sum\{\ \mathsf{y.C}\ \mid\ \mathsf{a} \leftarrow \mathsf{DB.T1}, && \textit{(by Rule 8)}\\
&\qquad \mathsf{x} \equiv \langle\ \mathsf{A{=}a.A},\ \mathsf{B{=}}set\{\ \langle\ \mathsf{C{=}b.INFO.C},\ \mathsf{D{=}b.INFO.D}\ \rangle\\
&\qquad\qquad\qquad \mid\ \mathsf{b} \leftarrow \mathsf{DB.T2},\ \mathsf{b.\#{=}@a}\ \},\ \mathsf{E{=}a.E}\ \rangle,\\
&\qquad \mathsf{y} \leftarrow \mathsf{x.B},\ \mathsf{x.A{=}10},\ \mathsf{y.D} > 5\ \}\\
&\longrightarrow sum\{\ \mathsf{y.C}\ \mid\ \mathsf{a} \leftarrow \mathsf{DB.T1}, && \textit{(by Rule 7)}\\
&\qquad \mathsf{y} \leftarrow set\{\ \langle\ \mathsf{C{=}b.INFO.C},\ \mathsf{D{=}b.INFO.D}\ \rangle\\
&\qquad\qquad \mid\ \mathsf{b} \leftarrow \mathsf{DB.T2},\ \mathsf{b.\#{=}@a}\ \},\\
&\qquad \mathsf{a.A{=}10},\ \mathsf{y.D} > 5\ \}\\
&\longrightarrow sum\{\ \mathsf{y.C}\ \mid\ \mathsf{a} \leftarrow \mathsf{DB.T1},\ \mathsf{b} \leftarrow \mathsf{DB.T2},\ \mathsf{b.\#{=}@a}, && \textit{(by Rule 8)}\\
&\qquad \mathsf{y} \equiv \langle\ \mathsf{C{=}b.INFO.C},\ \mathsf{D{=}b.INFO.D}\ \rangle,\ \mathsf{a.A{=}10},\ \mathsf{y.D} > 5\ \}\\
&\longrightarrow sum\{\ \mathsf{b.INFO.C}\ \mid\ \mathsf{a} \leftarrow \mathsf{DB.T1},\ \mathsf{b} \leftarrow \mathsf{DB.T2},\\
&\qquad \mathsf{b.\#{=}@a},\ \mathsf{a.A{=}10},\ \mathsf{b.INFO.D} > 5\ \} && \textit{(by Rule 7)}
\end{aligned}$$

We see that the initial dependent join, which was over a nested collection, is flattened into an 1NF join. Notice that DB.T1 is sorted by both @ and A attributes while DB.T2 is sorted by @ and #. That is, the derived program has the functionality of a sort-merge join since the join predicate is b.#=@a. This functionality can be deduced directly from the types of the comprehension generators. In contrast to most query optimization approaches, the programs derived in our framework are guaranteed to be correct since our framework uses transformations that are purely algebraic and meaning preserving.

The alternative access path of using the secondary index T3 can be derived from the equation $F'(\mathsf{DB}) = F(\mathcal{C}(\mathsf{DB}))$:

$$\begin{aligned}
&F'(\mathsf{DB}) = sum\{\ \mathsf{b.INFO.C}\ \mid\ \mathsf{a} \leftarrow \mathcal{C}(\mathsf{DB}).\mathsf{T1},\ \mathsf{b} \leftarrow \mathcal{C}(\mathsf{DB}).\mathsf{T2},\\
&\qquad\qquad \mathsf{b.\#{=}@a},\ \mathsf{a.A{=}10},\ \mathsf{b.INFO.D} > 5\ \}\\
&= sum\{\ \mathsf{b.INFO.C}\ \mid\ \mathsf{a} \leftarrow sorted[\mathsf{A}]\{\ \langle\ \mathsf{@{=}@c},\ \mathsf{A{=}c.A},\ \mathsf{B{=}c.B},\ \mathsf{E{=}d.E}\ \rangle\\
&\qquad\qquad\qquad \mid\ \mathsf{c} \leftarrow \mathsf{DB.T1},\ \mathsf{d} \leftarrow \mathsf{DB.T3},\ \mathsf{d.\#{=}@c}\ \},\\
&\qquad\qquad \mathsf{b} \leftarrow \mathsf{DB.T2},\ \mathsf{b.\#{=}@a},\ \mathsf{a.A{=}10},\ \mathsf{b.INFO.D} > 5\ \} && \textit{(by } \mathcal{C} \textit{ def)}\\
&\longrightarrow sum\{\ \mathsf{b.INFO.C}\ \mid\ \mathsf{c} \leftarrow \mathsf{DB.T1},\ \mathsf{d} \leftarrow \mathsf{DB.T3},\ \mathsf{d.\#{=}@c},\\
&\qquad \mathsf{a} \equiv \langle\ \mathsf{@{=}@c},\ \mathsf{A{=}c.A},\ \mathsf{B{=}c.B},\ \mathsf{E{=}d.E}\ \rangle, && \textit{(by Rule 8)}\\
&\qquad \mathsf{b} \leftarrow \mathsf{DB.T2},\ \mathsf{b.\#{=}@a},\ \mathsf{a.A{=}10},\ \mathsf{b.INFO.D} > 5\ \}\\
&\longrightarrow sum\{\ \mathsf{b.INFO.C}\ \mid\ \mathsf{c} \leftarrow \mathsf{DB.T1},\ \mathsf{d} \leftarrow \mathsf{DB.T3},\ \mathsf{b} \leftarrow \mathsf{DB.T2}, && \textit{(by Rule 7)}\\
&\qquad \mathsf{d.\#{=}@c},\ \mathsf{b.\#{=}@c},\ \mathsf{c.A{=}10},\ \mathsf{b.INFO.D} > 5\ \}
\end{aligned}$$

The resulting program is an alternative plan to evaluate the initial logical query. It is a 3-way sort-merge join that corresponds to the alternative access path associated with the secondary index T3. Both programs $F'(\mathsf{DB})$ and $F(\mathsf{DB})$ should be considered by the query optimizer for costing. If there were many integrity constraints because of multiple access paths, then an optimization step would consist of selecting one of the plan transformers $\mathcal{C}$, substituting $\mathcal{C}(\mathsf{DB})$ for DB in the current program, and normalizing the resulting program. The optimization process consists of the exploration of all the alternative programs generated by applying this optimization step multiple times as well as of using the commutativity and associativity properties of monoids.

3.2 Example 2: OODB Physical Design

The example presented here translates an OODB query into a physical plan that reflects an OODB physical design. The conceptual database schema is the following:

```
class hotel = ⟨ name: string, address: string, facilities: set(string),
                rooms: set(⟨ beds: int, price: int ⟩) ⟩
              extent: hotels;

class city = ⟨ name: string, hotels: bag(hotel),
               places_to_visit: list(⟨ name: string, address: string ⟩) ⟩
             extent: cities;
```

where the extent name is a collection of all instances of a class. The database schema db associated with this specification is the aggregation of all class extents along with a number of persistent variables. To make our examples short, though, we will assume that there are no persistent variables. In that case, db has type:

$$\langle \text{ hotels: set(hotel), cities: set(city) } \rangle$$

As we mentioned earlier, physical design in our framework consists of a set of physical design directives specified by the database implementor. In order to reduce the number of required physical directives, we assume a default implementation for the database. Then the physical design directives are commands to change these defaults.

In the default implementation, objects from two different classes are not clustered together. That is, the hotels extent will be stored in a different storage collection than the cities extent, while each cities.hotels bag will be a bag of OIDs[1] that reference hotels. But the database implementor can cluster cities and hotels together by stating the right physical directive. The default implementation for a nested collection, such as the hotels.rooms, is the direct storage model [23]: all hierarchical object structures are stored in preorder form. For example, hotels and hotels.rooms are clustered together, with the rooms of a hotel stored adjacent to the hotel.

The following is an example of physical design directives specified by the database implementor during the physical design of the previous OODB example:

```
directives = { implement( cities, sorted[name] ),          (1)
               implement( hotels, sorted[name] ),          (2)
               secondary( hotels, address ),               (3)
               normalize( cities.hotels ),                 (4)
               join_index( hotels.rooms ) }                (5)
```

Directives *(1)* and *(2)* indicate that both cities and hotels will be implemented as B-trees indexed by name. Directive *(3)* indicates that a secondary index on attribute address will be attached to hotels. Directive *(4)* indicates that cities.hotels will be normalized. The conceptual nested collection is reconstructed by a join. Directive *(5)* requests a binary join index for hotels.rooms. This directive implies that hotels.rooms be normalized and that there will be an additional index for accelerating the join between the normalized tables.

According to these physical design directives, the physical schema DB for our OODB example is the following: (it is automatically generated by a program described in Section 5)

[1] We decided to capture OIDs as tuple identifiers only to make the algorithms and examples easier to understand. A better alternative for OIDs might be to use surrogates, i.e., system generated unique numbers.

```
⟨ hotels: sorted[name](⟨ name: string, address: string,
                         facilities: list(string), rooms: ⟨⟩ ⟩),
  cities: sorted[name](⟨ name: string, hotels: ⟨⟩,
                         places_to_visit: list(⟨ name: string, address: string ⟩) ⟩),
  cities_hotels: sorted[#](⟨ #: TID, INFO: TID ⟩),
  hotels_rooms: sorted[#](⟨ #: TID, INFO: ⟨ beds: int, price: int ⟩ ⟩),
  hotels_rooms_JI: sorted[FROM](⟨ FROM: TID, TO: TID ⟩),
  hotels_address: sorted[address](⟨ #: TID, address: string ⟩) ⟩
```

The abstraction function $\mathcal{R}$(DB), which is also generated automatically, is the following:

```
𝓡(DB) = ⟨ hotels = set{ ⟨ name = b.name, address = b.address,
                          facilities = set{ x | x ← b.facilities },
                          rooms = set{ ⟨ beds=r.INFO.beds, price=r.INFO.price ⟩
                                     | i ← DB.hotels_rooms_JI, r ← DB.hotels_rooms,
                                       i.FROM=@b, i.TO=@r } ⟩
                      | b ← DB.hotels },
          cities = set{ ⟨ name = a.name,
                          hotels = bag{ @x | b ← DB.cities_hotels, x ← DB.hotels,
                                             b.#=@a, @x=b.INFO },
                          places_to_visit = list{ ⟨ name=c.name, address=c.address ⟩
                                                | c ← a.places_to_visit } ⟩
                      | a ← DB.cities } ⟩
```

That is, the set of rooms in a hotel b is reconstructed by joining the normalized table hotels_rooms with the join index hotels_rooms_JI. The set of all hotel references cities.hotels in a city a is reconstructed by joining the normalized table cities_hotels with the hotels extent.

The plan transformer generated (because of the secondary index) is the following:

```
𝒞(DB) = ⟨ @=@DB,
          hotels = sorted[address]{ ⟨ @=@x, name=x.name, address=y.address,
                                      facilities=x.facilities, rooms=x.rooms ⟩
                                  | x ← DB.hotels, y ← DB.hotels_address, y.#=@x },
          cities=DB.cities, cities_hotels=DB.cities_hotels, hotels_rooms=DB.hotels_rooms,
          hotels_rooms_JI=DB.hotels_rooms_JI, cities_address=DB.cities_address ⟩
```

We now translate a logical query against our OODB schema into a physical plan:

```
set{ h.name | c ← db.cities, h ← c.hotels, p ← c.places_to_visit,
              c.name="Portland", h.name=p.name }
```

This query finds all hotels in Portland that are also interesting places to visit. It is translated into

```
set{ h.name | c ← 𝓡(db).cities, h ← c.hotels, p ← c.places_to_visit,
              c.name="Portland", h.name=p.name }
```

which, when normalized by the Rules 8 and 7, becomes

```
set{ x.name | a ← DB.cities, c ← a.places_to_visit, b ← DB.cities_hotels,
              x ← DB.hotels, @x=b.INFO, b.#=@a,
              a.name="Portland", x.name=c.name }
```

Observe that this query is purely in terms of physical storage structures and has no nested comprehensions, hence it is not reconstructing any of the structures in the conceptual database. The resulting program is still a dependent join since c is derived from a.places_to_visit. But the collection DB.cities.places_to_visit is not normalized. Therefore, all places to visit are clustered together with the cities. Hence, when a city a is retrieved, all places to visit in a are retrieved as well.

If we use the secondary index secondary(hotels,address), the previous program becomes

$$\textit{set}\{\ \text{x.name}\ \mid\ \text{a} \leftarrow \mathcal{C}(\text{DB}).\text{cities},\ \text{c} \leftarrow \text{a.places\_to\_visit},\ \text{b} \leftarrow \mathcal{C}(\text{DB}).\text{cities\_hotels},\ \text{x} \leftarrow \mathcal{C}(\text{DB}).\text{hotels},\ @\text{x}=\text{b.INFO},\ \text{b.\#}=@\text{a},\ \text{a.name}=\text{“Portland”},\ \text{x.name}=\text{c.name}\ \}$$

which, when normalized by the Rules 8 and 7, becomes

$$\textit{set}\{\ \text{y.name}\ \mid\ \text{a} \leftarrow \text{DB.cities},\ \text{c} \leftarrow \text{a.places\_to\_visit},\ \text{b} \leftarrow \text{DB.cities\_hotels},\ \text{y} \leftarrow \text{DB.hotels},\ \text{z} \leftarrow \text{DB.hotels\_address},\ \text{z.\#}=@\text{y},\ @\text{y}=\text{b.INFO},\ \text{b.\#}=@\text{a},\ \text{a.name}=\text{“Portland”},\ \text{y.name}=\text{c.name}\ \}$$

4 Physical Design Specification

The following is the detailed description of the physical design directives. This description is by no means a complete list. It can be easily extended to incorporate new physical design techniques, new storage structures, and new physical algorithms. Such extensions are easy to incorporate because, as we will see next, each design technique can be expressed in a declarative way, in a form of a rule that is independent of the other rules. We have been experimenting with vertical partition of collections, hierarchical join indices [23], implementation of OIDs with surrogates, materialized functions and views, and denormalization [17] (where two collections that are not nested together are stored as a nested collection), but we decided not to include them here to simplify the exposition of the translation algorithms. The physical design directives are the following:

- implement($path, \mathcal{M}$): sets the implementation of the collection reached by the path expression $path$ to $\mathcal{M}$. (The monoid $\mathcal{M}$ represents a storage structure, such as an ordered list, a hash table, etc.)
- secondary($path, attrb$): attaches a secondary index on attribute $attrb$ to the collection reached by $path$ (in addition to the possible primary index specified by the implement directive). The secondary index may be attached to a deeply nested collection.
- normalize($path$): normalizes the nested collections reached by $path$ into one collection. Each element of this collection contains a reference (a TID) to its owner object. The original nested collection can be reconstructed by joining the $path$ with this collection.
- join_index($path$): is like normalize ($path$) but it also creates a binary join index to speed up the join between the $path$ and the normalized collection.
- cluster($path$): $path$ should be either a reference to a class or a collection of class references (such as set(person)). It clusters the class instances reached by $path$ together with the $path$ (instead of storing these instances into the class extent).
- partition($path, f$): specifies a horizontal partition of the collection reached by $path$. Function f is the partition function. Two elements x and y of the collection belong to the same partition if $f(x) = f(y)$. If the collection e (an instance of $\mathcal{M}$) is reached by $path$, then the horizontal partitions are computed as follows:

$$\textit{sorted}[\text{KEY}]\{\,\langle\, \text{KEY} = f(x),\ \text{PARTITION} = \mathcal{M}\{\, a \mid a \leftarrow e,\ f(a) = f(x)\,\}\,\rangle \mid x \leftarrow e\,\}$$

5 The Optimizer Generator

The following algorithms generate the physical schema, the abstraction function, and the semantic constraints from the conceptual schema and from the physical design directives. To make the algorithms simple, we assumed that the physical design directives have been checked for semantic correctness and for possible conflicts before they fed to these algorithms (e.g., all expression paths in the directives are valid within the conceptual database schema).

$\mathcal{T}([\![\text{base\_type}]\!], path) \rightarrow \text{base\_type}$

$\mathcal{T}([\![\text{class\_name}]\!], path)\ :\ \text{cluster}(path)$
$\quad \rightarrow \mathcal{T}([\![type(\text{db.class\_extent})]\!], \text{db.class\_extent})$

$\mathcal{T}([\![\text{class\_name}]\!], path) \rightarrow \text{TID}$

$\mathcal{T}([\![t]\!], path)\ :\ \text{normalize}(path) \rightarrow \langle\ \rangle$

$\mathcal{T}([\![t]\!], path)\ :\ \text{join\_index}(path) \rightarrow \langle\ \rangle$

$\mathcal{T}([\![\langle\, A_1 : t_1, \ldots, A_n : t_n \,\rangle]\!], path)$
$\quad \rightarrow \langle\, A_1 : \mathcal{T}([\![t_1]\!], path.A_1), \ldots, A_n : \mathcal{T}([\![t_n]\!], path.A_n) \,\rangle$

$\mathcal{T}([\![\mathcal{M}(t)]\!], path)\ :\ \text{partition}(path, f)$
$\quad \rightarrow \text{sorted[KEY]}(\langle\, \text{KEY} : \text{co-domain}(f),\ \text{PARTITION} : \mathcal{T}([\![\mathcal{M}(t)]\!], path) \,\rangle)$

$\mathcal{T}([\![\mathcal{M}(t)]\!], path)\ :\ \text{implement}(path, \mathcal{N}) \rightarrow \mathcal{N}(\mathcal{T}([\![t]\!], path))$

$\mathcal{T}([\![\text{list}(t)]\!], path) \rightarrow \text{list}(\mathcal{T}([\![t]\!], path))$

$\mathcal{T}([\![\mathcal{M}(t)]\!], path) \rightarrow \text{sorted[@]}(\mathcal{T}([\![t]\!], path))$

Figure 2: Generation of the Physical Schema

Algorithm 1 (Generation of the Physical Schema) *The rules for schema transformation are presented in Figure 2. The expression $\mathcal{T}([\![type]\!], path)$ takes the conceptual schema (the type of $path$) and returns the physical schema. The algorithm is expressed by rules of the form:*

$$\mathcal{T}([\![type]\!], path)\ :\ condition \rightarrow type$$

The condition checks whether a specific directive exists in the set of physical design directives. Only the first rule whose head matches the current type and whose condition matches one of the directives is executed. The matched directive is not used again. For example, the rule that checks for a partition directive can only be used once for each directive, hence allowing multiple horizontal partitions for the same collection.

The conceptual database db is mapped into the physical schema $\mathcal{T}([\![\text{dbtype}]\!], \text{db})$, which is a record since dbtype is also a record. The resulting record is extended with the following record attributes that contain the normalized collections and the alternative access paths. (The identifier $\underline{path}$ is the concatenation of all the attribute names in the path. For example, if $path = \text{A.B.C}$, then $\underline{path} = \text{A\_B\_C}$. In addition, $type(path)$ returns the type of the $path$.):

- for each normalize($path$) or join_index($path$), include the record attribute
  $\underline{path} : \text{sorted[\#]}(\langle\, \# : \text{TID}, \text{INFO} : \mathcal{T}([\![type(path)]\!], path) \,\rangle)$
- for each secondary($path, A$), include the record attribute
  $\underline{path}\_A : \text{sorted}[A](\langle\, \# : \text{TID}, A : \mathcal{T}([\![type(path.A)]\!], path.A) \,\rangle)$
- for each join_index($path$), include the record attribute
  $\underline{path}\_\text{JI} : \text{sorted[FROM]}(\langle\, \text{FROM} : \text{TID}, \text{TO} : \text{TID} \,\rangle)$

Algorithm 2 (Generation of the Abstraction Function) *The abstraction function R is generated by the rules in Figure 3. Expression $\mathcal{E}([\![type]\!], path, e_1, e_2)$ takes an abstract schema, a path expression, an expression e_1 (the current constructed expression), and expression e_2 (@e_2 references the last collection that contains e_1), and generates*

$\mathcal{E}([\![\text{basic\_type}]\!], path, e_1, e_2) \rightarrow e_1$

$\mathcal{E}([\![\text{class\_name}]\!], path, e_1, e_2)$: cluster($path$)
$\quad \rightarrow @(\mathcal{E}([\![type(\text{db.class\_extent})]\!], \text{db.class\_extent}, e_1, e_2))$

$\mathcal{E}([\![\text{class\_name}]\!], path, e_1, e_2)$
$\quad \rightarrow pick\{\, @(\mathcal{E}([\![type(\text{db.class\_extent})]\!], \text{db.class\_extent}, x, x))$
$\qquad\qquad \mid x \leftarrow \text{DB.class\_extent}, \; @x = e_1 \,\}$

$\mathcal{E}([\![\langle\, A_1 : t_1, \ldots, A_n : t_n \,\rangle]\!], path, e_1, e_2)$
$\quad \rightarrow \langle\, A_1 = \mathcal{E}([\![t_1]\!], path.A_1, e_1.A_1, e_2), \ldots, A_n = \mathcal{E}([\![t_n]\!], path.A_n, e_1.A_n, e_2) \,\rangle$

$\mathcal{E}([\![\mathcal{M}(t)]\!], path, e_1, e_2)$: join_index($path$)
$\quad \rightarrow \mathcal{M}\{\, \mathcal{E}([\![t]\!], path, r.\text{INFO}, r) \mid i \leftarrow \text{DB.}\underline{path}\text{\_JI}, \; r \leftarrow \text{DB.}\underline{path}, \; i.\text{FROM} = @e_2, \; i.\text{TO} = @r \,\}$

$\mathcal{E}([\![\mathcal{M}(t)]\!], path, e_1, e_2)$: normalize($path$)
$\quad \rightarrow \mathcal{M}\{\, \mathcal{E}([\![t]\!], path, x.\text{INFO}, x) \mid x \leftarrow \text{DB.}\underline{path}, \; x.\# = @e_2 \,\}$

$\mathcal{E}([\![\mathcal{M}(t)]\!], path, e_1, e_2)$: partition($path, f$)
$\quad \rightarrow \mathcal{M}\{\, y \mid x \leftarrow e_1, \; y \leftarrow \mathcal{E}([\![\mathcal{M}(t)]\!], path, x.\text{PARTITION}, x.\text{PARTITION}), \; f(y) = x.\text{KEY} \,\}$

$\mathcal{E}([\![\mathcal{M}(t)]\!], path, e_1, e_2) \rightarrow \mathcal{M}\{\, \mathcal{E}([\![t]\!], path, x, x) \mid x \leftarrow e_1 \,\}$

Figure 3: Generation of the Abstraction Function

the piece of the abstraction function that corresponds to this type. All free variable names that appear in a rule action need to be made unique to avoid the variable capture problem. The entire abstraction function is generated by $\mathcal{E}([\![\text{dbtype}]\!], \text{db}, \text{DB}, \text{DB})$.

The primitive monoid *pick* in the third rule is over tuple identifiers. Its zero value is null, its unit function is the identity function, and its merge function satisfies $\text{merge}^{pick}(\text{null}, x) = x$, otherwise $\text{merge}^{pick}(x, y) = x$. For example, *pick*$\{$ @x | x ← DB.hotels, @x=h $\}$ dereferences a hotel from the class extent DB.hotels using the TID h. If there is no such hotel, then it returns null. If there are more than one hotel (this never happens, since TIDs are unique), then it returns the first one.

The $f(y) = x.\text{KEY}$ predicate in the next-to-last rule in Figure 3, which checks for a partition, is redundant because of the way this partition was constructed. But, if there were a generator $v \leftarrow e$ in a comprehension, where e is partitioned by f, and a predicate $f(v) = constant$, then it is translated into $x \leftarrow e, \; y \leftarrow x.\text{PARTITION}, \; f(y) = x.\text{KEY}, \; f(y) = constant$, which implies $x.\text{KEY} = constant$. That way, only the partition with the specified KEY is retrieved.

Algorithm 3 (Generation of the Semantic Constraints) *For each such directive* secondary($lpath, attrb$)*, we generate the function*

$$\mathcal{C}_{lpath}(\text{DB}) = \mathcal{S}([\![\text{DBtype}]\!], \text{DB}, ppath)$$

where $\text{DBtype} = \mathcal{T}([\![\text{dbtype}]\!], \text{db})$ *is the physical database type and* $ppath$ *is the physical path expression that corresponds to the logical path expression* $lpath$*. Function* $\mathcal{S}$ *is defined as follows:*

$\mathcal{S}([\![\langle\, A_1 : t_1, \ldots, A_n : t_n \,\rangle]\!], e, A_i.path)$
$\quad \rightarrow \langle\, @ = @e, \; A_1 = e.A_1, \ldots, A_i = \mathcal{S}([\![t_i]\!], e.A_i, path), \ldots, A_n = e.A_n \,\rangle$

$\mathcal{S}([\![\mathcal{M}(\langle\, A_1 : t_1, \ldots, A_n : t_n \,\rangle)]\!], e, \emptyset)$
$\qquad \rightarrow \mathcal{M}\{\, \langle\, @ = @x, \; A_1 = x.A_1, \ldots, attrb = y.attrb, \ldots, A_n = x.A_n \,\rangle$
$\qquad\qquad \mid x \leftarrow e, \; y \leftarrow \text{DB.}\underline{lpath}\_attrb, \; y.\# = @x \,\}$

$\mathcal{S}([\![\mathcal{M}(t)]\!], e, path) \rightarrow \mathcal{M}\{\, \mathcal{S}([\![t]\!], x, path) \mid x \leftarrow e \,\}$

where $\emptyset$ denotes the empty path expression.

For example, if we had specified the directive

secondary(cities.places_to_visit, name)

we would have the following constraint:

```
C_{c.ptv} DB = ⟨ @=@DB, hotels=DB.hotels,
            cities = sorted[name]{ ⟨ @=@a, name=a.name, hotels=a.hotels,
                                     places_to_visit = list{ ⟨ @=@b, name=c.name, address=b.address ⟩
                                                           | b ← a.places_to_visit,
                                                             c ← DB.cities_places_to_visit_name,
                                                             c.#=@b } ⟩
                                   | a ← DB.cities }, . . . ⟩
```

This is a secondary index attached to a nested collection, i.e., we can access any place_to_visit by providing its name only, without having to go through the cities extent.

6 Translation of Updates

In this section we are concerned with the translation of user-level database updates over the conceptual database into updates over the internal database. For example, if there was a secondary index attached to a table, then, when we insert an item into this table, we would like the secondary index to be updated as well.

Database updates can be captured by extending the definition of monoid comprehensions with the following comprehension qualifiers: Qualifier $path$:= u destructively replaces the value stored at $path$ with u, qualifier $path$ += u merges the singleton u with $path$, and qualifier $path$ -= u deletes all elements in the collection reached by $path$ equal to u.

For example, if the abstract database db is of type set(int), then

some{ true | a ← db, a $>$ 10, a += 1 }

increments every database element greater than 10 by one. It returns true if there is at least one update performed.

A more complex example related to the previous OODB schema is the following:

```
some{ true | c ← db.cities, c.name="Portland", h ← c.hotels, h.name="Benson",
             r ← h.rooms, r.beds=1, r.price += 100 }
```

It increases the price of a single room in Portland's Benson hotel by $100.

If database updates modify primitive values only, then the query translation process described in Section 3 is sufficient for update translation too (since a conceptual path that reaches a primitive value is always translated into a physical path, while a conceptual path that reaches a collection may be translated into a complex comprehension.) For example, if we substitute $\mathcal{R}(\text{DB})$ for db in the last comprehension and normalize we get:

```
some{ true | a ← DB.cities, a.name="Portland", b ← DB.cities_hotels,
             x ← DB.hotels, x.name="Benson", @x=b.INFO, b.#=@a,
             i ← DB.hotels_rooms_JI, s ← DB.hotels_rooms,
             i.FROM=@x, i.TO=@s, s.INFO.beds=1, s.INFO.price += 100 }
```

Notice that the update s.INFO.price += 100 is over the physical database.

The difficult case is when we have an update over a collection type, such as the insertion of a new hotel:

```
some{ true | c ← db.cities, c.name="Portland",
             c.hotels += ⟨ name="Hilton", address="Park Ave", facilities={},
                           rooms = { ⟨ beds=1, price=100 ⟩, ⟨ beds=2, price=150 ⟩ } ⟩ }
```

$\mathcal{U}([\![\text{class\_name}]\!], path, \mathit{from}, \mathit{to})$: cluster($path$)
  $\rightarrow \mathcal{U}([\![type(\text{db.class\_extent})]\!], \text{db.class\_extent}, \mathit{from}, \mathit{to})$

$\mathcal{U}([\![\text{class\_name}]\!], path, \mathit{from}, \mathit{to}) \rightarrow [\ \text{DB.class\_extent += } @1\ \mathit{to}\ ]$

$\mathcal{U}([\![\langle\, A_1 : t_1, \ldots, A_n : t_n\, \rangle]\!], p, \mathit{from}, \mathit{to})$
  $\rightarrow \mathcal{U}([\![t_1]\!], p.A_1, \mathit{from}.A_1, \mathit{to}.A_1) +\!\!+ \cdots +\!\!+ \mathcal{U}([\![t_n]\!], p.A_n, \mathit{from}.A_n, \mathit{to}.A_n)$

$\mathcal{U}([\![\mathcal{M}(t)]\!], p, \mathit{from}, \mathit{to})$
  $\rightarrow [\ x \leftarrow \mathcal{B}([\![\mathcal{T}([\![\mathcal{M}(t)]\!], p)]\!], \mathit{from})\ ] +\!\!+ \mathcal{I}([\![\mathcal{M}(t)]\!], p, x, \mathit{to}) +\!\!+ \mathcal{U}([\![t]\!], p, x, \mathit{to})$

$\mathcal{U}([\![t]\!], path, \mathit{from}, \mathit{to}) \rightarrow [\,]$

$\mathcal{I}([\![\mathcal{M}(t)]\!], path, \mathit{from}, \mathit{to})$: normalize($path$)
  $\rightarrow [\ \text{DB}.\underline{path}\ \text{+=}\ @2\, \langle\ \# = @1,\ \text{INFO} = \mathcal{B}([\![\mathcal{T}([\![t]\!], path)]\!], \mathit{from})\ \rangle\ ]$

$\mathcal{I}([\![\mathcal{M}(t)]\!], path, \mathit{from}, \mathit{to})$: join_index($path$)
  $\rightarrow [\ \text{DB}.\underline{path}\text{\_JI += } \langle\ \text{FROM} = @1,\ \text{TO} = @2\ \rangle\ ]$

$\mathcal{I}([\![\mathcal{M}(t)]\!], path, \mathit{from}, \mathit{to})$: secondary($path, attrb$)
  $\rightarrow [\ \text{DB}.\underline{path}\_attrb\ \text{+=}\ \langle\ \# = @1,\ attrb = \mathcal{B}([\![\mathcal{T}([\![t]\!], path)]\!], \mathit{from}).attrb\ \rangle\ ]$

$\mathcal{I}([\![\mathcal{M}(t)]\!], path, \mathit{from}, \mathit{to})$: partition($path, f$)
  $\rightarrow [\ x \leftarrow \mathit{to},\ x.\text{KEY} = f(\mathit{from}),\ x.\text{PARTITION += } \mathcal{B}([\![\mathcal{T}([\![t]\!], path)]\!], \mathit{from})\ ]$

$\mathcal{B}([\![\langle\, A_1 : t_1, \ldots, A_n : t_n\, \rangle]\!], e) \rightarrow \langle\ A_1 = \mathcal{B}([\![t_1]\!], e.A_1), \ldots, A_n = \mathcal{B}([\![t_n]\!], e.A_n)\ \rangle$

$\mathcal{B}([\![\mathcal{M}(t)]\!], e) \rightarrow \mathcal{M}\{\, \mathcal{B}([\![t]\!], x) \mid x \leftarrow e\, \}$

$\mathcal{B}([\![t]\!], e) \rightarrow e$

Figure 4: Update Generation

This conceptual update needs to be translated into the following internal update:

```
sum{ 1 | a ← DB.cities, a.name="Portland",
         DB.hotels += @1 ⟨ name="Hilton", address="Park Ave", facilities=[ ], rooms=⟨⟩ ⟩,
         DB.cities_hotels += ⟨ #=@a, INFO=@1 ⟩,
         DB.hotels_address += ⟨ #=@1, address="Park Ave" ⟩,
         DB.hotels_rooms += @2 ⟨ #=@1, INFO=⟨ beds=1, price=100 ⟩ ⟩,
         DB.hotels_rooms_JI += ⟨ FROM=@1, TO=@2 ⟩,
         DB.hotels_rooms += @2 ⟨ #=@1, INFO=⟨ beds=2, price=150 ⟩ ⟩,
         DB.hotels_rooms_JI += ⟨ FROM=@1, TO=@2 ⟩ }
```

That is, we may need to perform multiple internal updates for a single conceptual update. Insertions to a collection in the internal database may be tagged by a natural number n: $path$ += $@n\ u$. The update $path$ += $@n\ u$ inserts u into the collection reached by $path$ but it also binds the memory register numbered n to the TID of the newly inserted tuple. The value of this register can be retrieved by evaluating $@n$. Our physical design language requires only two registers: $@1$ and $@2$.

Algorithm 4 (Update Generation) *For each conceptual database update of the form* $path$ += e*, where* $path$ *is an* $\mathcal{M}(T)$ *collection,* $\mathcal{U}([\![type]\!], ppath, path, e)$ *generates a list of qualifiers that update the physical database (*$ppath$ *is* *the logical path expression that corresponds to $path$, e.g., if $path =$* s.price *then $ppath =$* db.hotels.rooms.price *and $type$ is the type of $ppath$.) The algorithm is given in Figure 4. It uses the following support functions:*

- $\mathcal{I}([\![\mathcal{M}(t)]\!], path, from, to)$*: it generates additional updates for normalized tables, join indices, secondary indices, etc. All applicable rules are executed and the generated qualifier lists are appended.*
- $\mathcal{B}([\![t]\!], e)$*: translates the logical expression e into a physical expression that reflects the physical type t. For example,*

```
𝓑(⟨ name: string, address: string, facilities: list(string), rooms: ⟨⟩ ⟩,
  ⟨ name="Hilton", address="Park Ave", facilities={},
    rooms = {⟨ beds=1, price=100 ⟩,...} ⟩)
= ⟨ name="Hilton", address="Park Ave", facilities=[ ], rooms=⟨⟩ ⟩
```

For example, the update generation algorithm generates the following list of qualifiers for the conceptual update x.hotels += e:

```
[ DB.hotels += @1 ⟨ name=e.name, address=e.address, facilities=e.facilities, rooms=⟨⟩ ⟩,
  DB.cities_hotels += ⟨ #=@x, INFO=tid(@1) ⟩,
  DB.hotels_address += ⟨ #=tid(@1), address=e.address ⟩,
  c ← e.rooms,
  DB.hotels_rooms += @2 ⟨ #=tid(@1), INFO=⟨ beds=c.beds, price=c.price ⟩ ⟩,
  DB.hotels_rooms_JI += ⟨ FROM=tid(@1), TO=tid(@2) ⟩ ]
```

Database deletions can be handled in the same way as insertions (by substituting -= for +=). Updates of the form path := e, where path is a collection, can be translated into:

```
some{ true | x ← path, path -= x, y ← e, path += y }
```

7 Related Work

Our framework is based on monoid homomorphisms, which were first introduced as an effective way to capture database queries by V. Tannen and P. Buneman [5, 7, 6]. Their form of monoid homomorphism (also called structural recursion over the union presentation – SRU) is more expressive than our calculus. Operations of the SRU form, though, require the validation of the associativity, commutativity, and idempotence properties of the monoid associated with the output of this operation. These properties are hard to check by a compiler [7], which makes the SRU operation impractical. They first recognized that there are some special cases where these conditions are automatically satisfied, such as for the ext$(f)(A)$ operation. In our view, SRU is too expressive, since inconsistent programs cannot always be detected in that form. To our knowledge, there is no normalization algorithm for SRU forms in general. (I.e., SRU forms cannot be put in canonical form.) On the other hand, ext(f) is not expressive enough, since it does not capture operations that involve different collection types and it cannot express predicates and aggregates. We believe that our monoid comprehension calculus is the most expressive subset of SRU where inconsistencies can always be detected at compile time, and, more importantly, where all programs can be put in canonical form.

Monad comprehensions were first introduced by P. Wadler [24] as a generalization of list comprehensions (which already exist in some functional languages). Monoid comprehensions are related to monad comprehensions, but they are considerably more expressive. In particular, monoid comprehensions can mix inputs from different collection types and may return output of a different type. This mixing of types is not possible for monad comprehensions, since they restrict the inputs and the output of a comprehension to be of the same type. Monad comprehensions were first proposed as a convenient and practical database language by P. Trinder [21, 20], who also presented many algebraic transformations over these forms as well as methods for converting comprehensions into joins. The monad comprehension syntax was also adopted by P. Buneman and V. Tannen [8] as an alternative syntax to monoid homomorphisms. The comprehension syntax was used for capturing operations that involve collections of the same type while structural recursion was used for expressing the rest of the operations (such as converting one collection type to another, predicates, and aggregates). Our normalization algorithm is highly influenced by L. Wong's work on normalization of monad comprehensions [25]. He presented some powerful rules for flattening nested comprehensions into canonical comprehensions whose generators are over simple paths. These canonical forms are equivalent to our canonical forms for monoid homomorphisms.

Our schema transformation technique is influenced by the Genesis extensible database management system [2, 3]. Genesis introduced a technology that enables customized database management systems to be developed rapidly, using user-defined modules as building blocks. A transformation model is used to map abstract models to concrete implementations. This map is done with possibly more than one level of conceptual to internal mappings, transferring abstract models to more implementation-oriented ones, until a primitive layer is reached. For each type transformer, the database implementor is responsible for writing the program transformers that translate abstract schemas into concrete schemas, and the operation expanders that translate any operation on an abstract type to a sequence of operations on the concrete type. This framework is more general than ours since it allows any mapping from abstract to concrete schemas while ours is guided by the physical design directives. We believe that our approach of using design directives to guide the mapping leaves little space for errors and can be easily modified and extended.

A similar technique for mapping conceptual schemas into internal schemas was used by M. Scholl [17, 18]. More specifically, he considered the problems of clustering and denormalization in a relational database system, that is, mapping flat tables into nested structures in which related objects are clustered together. He also used abstraction functions, called conceptual-to-internal mappings, to capture the schema transformation, but he required these functions to be invertible. He used normalization techniques for obtaining efficient nested queries from the conceptual flat queries, which were based on the algebraic equivalences between the NF^2 expressions. He recognized that these algebraic transformations can only be effective if they are combined with a redundancy elimination phase where all redundant joins are removed. Even though our physical design framework has different objectives, our approach is very similar to this approach. Our proposed system is more automated since most of the query translation work is done when compiling the design directives.

Another approach for physical OODB independence was proposed for the PIOS system [1, 16]. PIOS includes a language, called SDL (a storage definition language), that allows one to specify the mapping from the logical to the physical schema in a form similar to our physical design directives. The mappings supported are vertical and horizontal partitioning of classes and object clustering. The physical schema is computed automatically from these specifications and logical operations are mapped to physical operations. Other approaches for physical OODB design include Lanzelotte's work on OODB query optimization [13], which is based on a graph physical design language, and the GMAP system [22] that uses a search-based algorithm to match for applicable access paths in a query.

8 Conclusion

Object-oriented database systems have long been criticized for not supporting sufficient levels of data independence. The main reason for this criticism is that early OODB systems used simple pointer chasing to perform object traversals, which did not allow many opportunities for optimization. There are many recent system proposals though, such as GemStone, O2, and ODMG, that use more sophisticated methods for object traversals. These systems support a declarative language to express queries, and advanced physical structures and alternative access paths to speed up the bulk manipulation of objects. Since object models are more complex than the relational model, most OODB systems are lacking a formal theory for query translation and optimization that could capture the new advanced physical design proposals that are necessary to speed up object queries.

In this paper we presented a formal framework for achieving a complete data independence in an OODB system. The physical design process in this framework consists of the specification of a set of physical design directives that describe in declarative form the physical design of parts of the logical database schema. We use these directives to generate a program (the abstraction function) that automatically transforms any logical query or update into a physical program. These transformations are purely algebraic and can be easily validated for correctness, since they are based on a formal framework. The generation of the abstraction function itself is achieved by a rule-based system, which can be easily extended to incorporate more advanced physical design directives.

9 Acknowledgements

The authors are grateful to Tim Sheard for helpful comments on the paper. This work is supported by the Advanced Research Projects Agency, ARPA order number 18, monitored by the US Army Research Laboratory under contract DAAB-07-91-C-Q518, and by NSF Grants IRI 9118360 and IRI 9509955.

References

[1] N. Aloia, S. Barneva, and F. Rabitti. Supporting Physical Independence in an Object Database Server. In *Workshop on Object-Oriented Programming ECOOP'92*, pp 396–412, September 1992. LNCS 615.

[2] D. Batory, J. Barnett, J. Garza, K. Smith, K. Tsukuda, B. Twichell, and T. Wise. Genesis: An Extensible Database Management System. *IEEE Transactions on Software Engineering*, 14(11):1711–1729, November 1988.

[3] D. S. Batory, J. R. Barnett, J. F. Garza, K. P. Smith, K. Tsukuda, B. C. Twichell, and T. Wise. Genesis: A Reconfigurable Database Management System. Technical report, Department of Computer Science, University of Texas at Austin, March 1986. TR-86-07.

[4] E. Bertino. A Survey of Indexing Techniques for Object-Oriented Database Management Systems. In J. Freytag, D. Maier, and G. Vossen, editors, *Query Processing for Advanced Database Systems*, pp 384–418. Morgan Kaufmann, 1994.

[5] V. Breazu-Tannen, P. Buneman, and S. Naqvi. Structural Recursion as a Query Language. In *Proceedings of the Third International Workshop on Database Programming Languages: Bulk Types and Persistent Data, Nafplion, Greece*, pp 9–19. Morgan Kaufmann Publishers, Inc., August 1991.

[6] V. Breazu-Tannen, P. Buneman, and L. Wong. Naturally Embedded Query Languages. In *4th International Conference on Database Theory, Berlin, Germany*, pp 140–154. Springer-Verlag, October 1992. LNCS 646.

[7] V. Breazu-Tannen and R. Subrahmanyam. Logical and Computational Aspects of Programming with Sets/Bags/Lists. In *18th International Colloquium on Automata, Languages and Programming, Madrid, Spain*, pp 60–75. Springer-Verlag, July 1991. LNCS 510.

[8] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension Syntax. *SIGMOD Record*, 23(1):87–96, March 1994.

[9] L. Fegaras. A Uniform Calculus for Collection Types. Oregon Graduate Institute Technical Report 94-030. Available by anonymous ftp from `cse.ogi.edu:/pub/crml/tapos.ps.Z`.

[10] L. Fegaras and D. Maier. Towards an Effective Calculus for Object Query Languages. *ACM SIGMOD International Conference on Management of Data, San Jose, California*, May 1995. Available by anonymous ftp from `cse.ogi.edu:/pub/crml/sigmod95.ps.Z`.

[11] L. Fegaras and D. Stemple. Using Type Transformation in Database System Implementation. In *Proceedings of the Third International Workshop on Database Programming Languages: Bulk Types and Persistent Data, Nafplion, Greece*, pp 337–353. Morgan Kaufmann Publishers, Inc., August 1991.

[12] A. Kemper and G. Moerkotte. Advanced Query Processing in Object Bases Using Access Support Relations. In *Proceedings of the Sixteenth International Conference on Very Large Databases, Brisbane, Australia*, pp 290–301. Morgan Kaufmann Publishers, Inc., August 1990.

[13] R. Lanzelotte, P. Valduriez, and J. Ziane, M. Cheiney. Optimization of Nonrecursive Queries in OODBs. *Deductive and Object-Oriented Databases, Munich, Germany*, pp 1–21, 1991.

[14] D. Maier and J. Stein. Indexing in an Object-Oriented DBMS. In *International Workshop on Object-Oriented Database Systems, Asilomar, CA*, pp 171–182, September 1986.

[15] D. Maier, J. Stein, A. Otis, and A. Purdy. Development of an Object-Oriented DBMS. In *Proceedings of the Object-Oriented Programming Systems, Languages and Applications, Portland, Oregon*, 1986.

[16] F. Rabitti, L. Benedetti, and F. Demi. Query Processing in PIOS. In *Sixth International Workshop on Persistent Object Systems, Tarascon, France*, pp 408–431, September 1994. Proceedings to be published in the Springer-Verlag *Workshops in Computer Science* series.

[17] M. Scholl. Theoretical Foundation of Algebraic Optimization Utilizing Unnormalized Relations. In *Proceedings International Conference on Database Theory*, pp 380–396. Springer-Verlag, September 1986. LNCS 243.

[18] M. Scholl. Physical Database Design for an Object-Oriented Database System. In J. Freytag, D. Maier, and G. Vossen, editors, *Query Processing for Advanced Database Systems*, pp 420–447. Morgan Kaufmann, 1994.

[19] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access Path Selection in a Relational Database Management System. *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Boston, Massachusetts*, pp 23–34, May 1979.

[20] P. Trinder. Comprehensions: A Query Notation for DBPLs. In *Proceedings of the Third International Workshop on Database Programming Languages: Bulk Types and Persistent Data, Nafplion, Greece*, pp 55–68. Morgan Kaufmann Publishers, Inc., August 1991.

[21] P. Trinder and P. Wadler. Improving List Comprehension Database Queries. In *in Proceedings of TENCON'89, Bombay, India*, pp 186–192, November 1989.

[22] O. Tsatalos, M. Solomon, and Y. Ioannidis. The GMAP: A Versatile Tool for Physical Data Independence. In *Proceedings of the 20th VLDB Conference, Santiago, Chile*, September 1994.

[23] P. Valduriez, S. Khoshafian, and G. Copeland. Implementation Techniques of Complex Objects. In *Proceedings of the Twelfth International Conference on Very Large Databases, Kyoto, Japan*, pp 101–109, 1986.

[24] P. Wadler. Comprehending Monads. *Proceedings of the ACM Symposium on Lisp and Functional Programming, Nice, France*, pp 61–78, June 1990.

[25] L. Wong. Normal Forms and Conservative Properties for Query Languages over Collection Types. *Proceedings of the 12th ACM Symposium on Principles of Database Systems, Washington, DC*, pp 26–36, May 1993.