Abstract

This paper describes extended relations, a generalization of ordinary relations in which tuple identifiers (tids) can appear as attribute values. The relational algebra is extended to include new operations that manipulate these tids; the resulting algebra is called the tid-algebra. The theorems of the tid-algebra are shown to provide a formal basis for optimizer transformations that involve high-level access structures. Three particular access structures — secondary indices, join indices, and join extensions — are examined in detail.

1. Introduction

A relation is an abstraction of physical data. The mappings from the relations seen by users to secondary storage records may be complex. Understanding them typically involves knowledge of auxiliary access structures, physical pointers, disk blocking, clustering, and so on. The query optimizer shields the user from this complexity by transforming a user query over relations into a program that accesses physical files. Such a program is called a query plan.

The generation of a query plan can be organized into different levels of abstraction [11]. At each level, the optimizer transforms the current query representation to one that involves more implementation-specific details; only at the final level is the query plan generated. Different optimization techniques may be available at different levels. In particular, algebraic transformations are useful at the higher levels of optimization. These transformations correspond to theorems in the relational algebra. For example, the following theorem asserts that a selection can be pushed inside of a join, assuming that the condition involves only attributes in the base relation:

\[ \sigma_c(r \geq a) = \sigma_c(r > a) \]

The optimizer can use this theorem as a rewrite rule, transforming a query corresponding to the left-hand-side into an equivalent one corresponding to the right-hand-side. Such an approach is especially useful in query optimizer generators [3, 7].

Although algebraic transformations are effective for query optimization, they only apply to objects for which there is an algebra. Consequently, decisions such as access path selection and operator implementation traditionally have to be made in other ways. In this paper we extend the relational algebra in order to allow more of the optimization process to be algebraic. In particular, we show how the selection of device-independent access structures (such as secondary indices) can be represented as transformations in an extended algebra. These access structures often have the same implementations as ordinary base relations.

Our results are based on the notion of an extended relation. Extended relations are generalizations of ordinary relations, in that they can have tuple references as attribute values. In Section 2 we discuss extended relations in detail. In Section 3 we introduce an algebra of extended relations, called the tid-algebra. The important new operator here is called lookup, which is used to access tuples by their references. A tuple reference can be treated as a pointer, and so doing a lookup is very efficient. Most of the useful theorems in the tid-algebra involve equating a relational algebra expression with one involving lookup operations. In this way an optimizer can algebraically transform a high-level user query into one that involves low-level pointer manipulation.

In Section 4 we discuss view definitions and their relationship to rule-based query optimization. In particular, we show three ways in which a view can be used as a rewrite rule. This discussion is particularly relevant because auxiliary storage structures can be defined as views in the tid-algebra. Thus theorems in the tid-algebra can be used to do access path selection, by transforming a query into an equivalent one that mentions appropriate auxiliary storage structures. In Sections 5 and 6 we show how this process works for two common access structures: secondary indices and join indices.

In Section 7 we consider primary storage structures. Instead of being stored as a flat file, a relation may contain pointers connecting its tuples with tuples in other relations. In particular, a relation may be "prejoined" with other ones. Such an organization is often used by object-oriented systems to support efficient database traversals. We show how view definitions in the tid-algebra support the same organization for the relational model. Consequently, a relational query involving joins can be optimized algebraically to one that simply follows pointers.
Finally, in Section 8 we summarize and suggest directions for further research.

2. Extended Relations

Tuple references (or tids) are supported by the record managers of existing relational database systems. For example, in the commercial version of INGRES [5] a tid is an integer i such that

1. \( i \mod 512 = \) the page on which the tuple is stored, and
2. \( i \div 512 = \) an offset within the page. At this offset a pointer to the tuple is stored. The tuple is stored on the same page as the pointer.

Tids resemble the surrogates used in semantic data models, and in fact can be used to implement surrogates. One advantage that tids have over surrogates is that they have a physical meaning; tuples with tids that are close in value are probably physically close as well. Thus an optimizer can take advantage of this property, such as by accessing a set of tids in sort order. However, such concerns are part of the lower levels of the query optimizer, we do not use this property of tids in this paper.

Let \( D \) and \( \text{Tidser} \) be disjoint sets, denoting the set of all attribute values and the set of all tid values respectively. The set \( U = D \cup \text{Tidser} \) is thus the universe of all values. Let \( A \) be the set of all attribute names. Each attribute has a domain, defined by a function \( \text{dom}(A) : A \rightarrow 2^D \). In general, we make no restrictions on the domain of an attribute. A domain may contain tids that refer to different extended relations, or even both tid and non-tid values.

An extended relation scheme (or simply scheme) is a finite subset of \( A \). A tuple on scheme \( R = (A_1, \ldots, A_k) \) is a function \( f : \text{dom}(A_1) \times \cdots \times \text{dom}(A_k) \) having the property that for all \( i \in \{1, \ldots, k\}, (A_i) \in \text{dom}(A_i) \). As instance of scheme \( R \) is a set \( r \) of tuples on scheme \( R \). An extended relation on a scheme \( R \) is a set of tuples for \( R \).

Every tuple of a stored extended relation has a unique identifier: its tid. When a tuple is retrieved from storage, its tid can also be retrieved. Consequently, the scheme of every stored extended relation contains a special attribute, called its tid-attribute. The value of this attribute for any tuple is that tuple’s tid. Two tuples cannot have the same tid-attribute value, regardless of the extended relations they belong to. Tid-attributes are similar to the entity surrogate attributes of [2], and we use a similar naming convention. In particular, the tid-attribute for an extended relation \( r \) is named \( r_{\text{tid}} \).

Example 1: Figure 1 shows three extended relations, including their tid-attributes. Figure 2 shows three other extended relations derived from those of Figure 1. For example, the attribute \( r_{\text{tid}} \) in relation \( r_1 \) contains tids that refer to tuples in \( r \). Note that it is convenient for this attribute to have the same name as the tid-attribute of \( r \), although it is not required in our model.

3. The Tid Algebra

The tid-algebra is an algebra of extended relations. All the operators of the relational algebra (\( \sigma, \pi, \bowtie, \text{etc.} \)) are also operators of the tid-algebra. Moreover, their definitions are the same as in the relational algebra. Consequently, theorems of the relational algebra are also true of the tid-algebra. Sources for these theorems include [13] and [8].

An operator called lookup is special to the tid-algebra. This operator replaces tids in a specified column of an extended relation by the tuples the tids refer to. This operator is defined as follows:

Definition: Let \( T \) be a scheme containing the attribute \( A \), and let \( T' = T \backslash \{A \} \cup \{A_{\text{tid}}\} \). Let \( t \) and \( r \) be instances of \( T \) and \( R \) respectively. Then we define

\[
\text{lookup}_T(t, r) = \pi_T(t_{\text{tid}} = r_{\text{tid}})
\]

Note that if attribute \( A \) of relation \( t \) contains values other than tids for \( r \), then those \( r \) tuples will drop out of the join. Also note that, under some conditions, composition of lookups is commutative.

Theorem 1: Let \( A \) and \( B \) be attributes of extended relation \( t \). Then

---

**Figure 1:** Some Extended Relations

<table>
<thead>
<tr>
<th>r</th>
<th>r_{\text{tid}}</th>
<th>Student</th>
<th>r_{\text{tid}}</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>John</td>
<td>CS</td>
<td>t1</td>
</tr>
<tr>
<td>t2</td>
<td>Joanne</td>
<td>Math</td>
<td>t2</td>
</tr>
<tr>
<td>t3</td>
<td>Gina</td>
<td>CS</td>
<td>t3</td>
</tr>
</tbody>
</table>

**Figure 2:** More Extended Relations

<table>
<thead>
<tr>
<th>r2</th>
<th>r_{\text{tid}}</th>
<th>Dept</th>
<th>r_{\text{tid}}</th>
</tr>
</thead>
<tbody>
<tr>
<td>t4</td>
<td>CS</td>
<td>14</td>
<td>t4</td>
</tr>
<tr>
<td>t5</td>
<td>Math</td>
<td>15</td>
<td>t5</td>
</tr>
<tr>
<td>t6</td>
<td>French</td>
<td>16</td>
<td>t6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>r3</th>
<th>r_{\text{tid}}</th>
<th>r_{\text{tid}}</th>
</tr>
</thead>
<tbody>
<tr>
<td>t7</td>
<td>14</td>
<td>17</td>
</tr>
<tr>
<td>t8</td>
<td>15</td>
<td>18</td>
</tr>
<tr>
<td>t9</td>
<td>16</td>
<td>10</td>
</tr>
</tbody>
</table>
Definition:
\[ \text{lookup}_{A_1, \ldots, A_n}(t_1, t_2, \ldots, t_n) = \text{lookup}_{A_1, \ldots, A_n}(t_1, t_2, \ldots, t_n) \]

Example 2: Consider Figure 3. Extended relation \( r_4 \) is the result of the query \( \text{lookup}_{A_1, A_2}(r_3, t) \). Extended relation \( r_5 \) is the result of the query \( \text{lookup}_{A_1, A_2}(r_4, t) \), which is the same as \( \text{lookup}_{A_1, A_2}(r_3, t) \). Extended relation \( r_6 \) is the same as \( \text{lookup}_{A_1, A_2}(r_3, t) \). Note that \( r_4 \) and \( r_5 \) themselves do not have tid-attributes, because they are not stored. □

We use the operator \text{lookup} in many tid-algebra theorems. For example, the following two theorems assert that \text{lookup} can be commuted with selections and joins.

Theorem 2: If \( e \) is a condition involving only attributes in \( r \) other than \( A \), then \[ \alpha_e(\text{lookup}_A(r, t)) = \text{lookup}_A(\alpha_e(r), t) \]

Theorem 3: If condition \( e \) refers only to attributes of \( r \) and \( s \) other than \( B \), where \( B \) is an attribute of \( s \), then \[ r \bowtie_e s \iff \text{lookup}_B(r \bowtie_e s) = \text{lookup}_B(r \bowtie_e s) \]

4. Operators and Views

A view definition is the bindig of an expression in a query language to a name. For example, the view named \text{oldperson} can be defined by

\[ \text{oldperson} := \sigma_{\text{age} \geq 65}(\text{person}) \]

A view definition establishes an equivalence between the name and the expression defining it. For example, the query \( \tau_{\text{age} = 65}(\text{oldperson}) \) is equivalent to the query \( \tau_{\text{age} = 65}(\sigma_{\text{age} \geq 65}(\text{person})) \). Views can be parameterized. For example, a more general view definition is the following:

\[ \text{old}(A, R) := \sigma_{A \geq 65}(R) \]

We can then define the previous view by saying \( \text{oldperson} := \text{old}(\text{age}, \text{person}) \). A parameterized view is also called a user-defined operator. For example, the two definitions of \text{lookup} in Section 3 can be thought of as view definitions in this way.

An optimizer can treat view definitions as transformation rules in three different ways. First, the view definition can be thought of as an abbreviation mechanism; the name of the view simply abbreviates the defining expression. The optimizer transforms queries involving a view name to the equivalent queries involving the view expression. This technique is called query modification [12]. With query modification, the view definition can be read from left to right as a rewrite rule. The above definition of \text{old} and the second definition of \text{lookup} in Section 3 are examples of views as abbreviations.

Second, a nonparameterized view may be materialized, so that the value of the view is kept explicitly in a stored extended relation. In this case, the optimizer can treat the view definition as a rewrite rule from right to left. That is, it should translate occurrences of the view expression in a query into the view name. In this way, the view expression will be computed once. Associated with materialized views are the extra storage cost and the cost of keeping them current in the presence of updates to their operands. A view should be materialized if these costs are not as high as the costs of always recomputing the view expression.

The third way in which view definitions can assist optimization is when the view expression is of a special form that has an efficient implementation. In this case, the optimizer should treat the definition as a rewrite rule from right to left, regardless of whether the view is materialized. For example, consider the first definition of \text{lookup}. This definition asserts that the \text{lookup} operator behaves the same as a certain join expression. However, we distinguish this particular kind of join because it has a very efficient implementation. In particular, each tid can be thought of as a pointer, so doing a lookup only involves accessing the first operand of \text{lookup} and following pointers. Thus whenever a user query matches the right hand side of the definition, the optimizer should transform it to the corresponding query matching the left hand side.

High-level access structures also correspond to this third use of views. In the next two sections we consider high-level access structures, and show how they can be specified as parameterized view definitions in the tid-algebra. This formulation leads to equivalence theorems that can be expressed as term rewrite rules, and input to an optimizer generator based on such rules. The right members of the equalities in the theorems are likely to have more efficient implementations than the left members, so each left member corresponds to the right hand side of a rewrite rule, and each right member to the right hand side. In Section 7 we give a similar treatment to an
5. Secondary Indices

Perhaps the most well-known high-level access structure is the dense secondary index. A secondary index can be defined as a parameterized view in the tid-algebra as follows:

**Definition:** Let $A$ be an attribute of stored extended relation $r$. The index of $r$ on $A$ is defined as follows:

$$\text{index}_A(r) = \pi_{rA, rTid}(r)$$

In other words, $\text{index}$ is an operator in the tid-algebra that takes two arguments: an extended relation and an attribute to index on. Note that indices need not be previously materialized in order to be useful. To compute a join, for example, the most efficient query plan might involve creating an index dynamically. Thus the optimizer should be able to introduce index operators into queries even if they are not materialized.

**Example 3:** The extended relation $r_1$ in Figure 2 is a materialized secondary index on extended relation $r$ from Figure 1. It is created by the following view definition:

$$r_1 = \text{index}^r_A(r)$$

Similarly, extended relation $r_2$ is a materialized secondary index on extended relation $r$.

We now illustrate how the standard uses of indices can be formalized as transformations in the tid-algebra. In the following theorems, instance $r$ has scheme $R$ and instance $s$ has scheme $S$.

**Theorem 4:** Let $A$ be an attribute in scheme $R$ other than $rTid$. Then

$$\pi_{rA, rTid}(r) = \text{lookup}_{rTid}(\pi_{rTid}(\text{index}_A(r)), r)$$

A more common use of an index is to efficiently select from a relation. This usage is stated formally in the following theorem:

**Theorem 5:** If $c$ is a condition involving only attribute $rA$, then

$$\sigma_c(\pi_{rA, rTid}(r)) = \text{lookup}_{rTid}(\pi_{rTid}(\sigma_c(\text{index}_A(r))), r)$$

If a selection condition is the conjunction of several subconditions, then we can perform a selection by first intersecting tids and then doing a lookup:

**Theorem 6:** Suppose that condition $c = c_1 \land \cdots \land c_n$, where each $c_i$ mentions no attribute other than $rA$. Then

$$\sigma_c(\pi_{rA, rTid}(r)) = \text{lookup}_{rTid}(\sigma_c(\text{index}_A(r)), r)$$

where $M = \pi_{rTid}(\sigma_c(\text{index}_A(r)))$.

Indices also play an important part in computing joins. For example, for some join conditions, it is possible to compute the join by first joining two indices and then doing a lookup:

**Theorem 7:** Let $c$ be a condition mentioning no attributes except $rA$ and $sB$. Then the following two equations hold:

$$\pi_{rA, sB}(r \land c, s) = \pi_{rA, sB}(\text{index}_A(r) \land \text{index}_B(s))$$

$$\pi_{rA, sB}(r \land c, s) = \text{lookup}_{rTid, sTid}(\pi_{rTid, sTid}(\text{index}_A(r) \land \text{index}_B(s)), r, s)$$

If the join condition is more general, then we may need to join an index with an extended relation:

**Theorem 8:** Let $c$ be a condition mentioning no attributes except $rA$ and attributes of $s$. Then the following two equations hold:

$$\pi_{rA, sB}(r \land c, s) = \pi_{rA, sB}(\text{index}_A(r) \land \text{index}_B(s))$$

$$\pi_{rA, sB}(r \land c, s) = \text{lookup}_{rTid, sTid}(\pi_{rTid, sTid}(\text{index}_A(r) \land \text{index}_B(s)), r, s)$$

6. Join Indices

A join index is a high-level access structure that supports the efficient computation of a join.

**Definition:** A join index for stored extended relation instances $r$ and $s$ on condition $c$ is an operator defined by the following view definition:

$$J_{r, s}(r, s, c) = \pi_{rTid, sTid}(c)$$

In [14], join indices contain surrogates instead of tids. They are called join supports in [4] and indirect joins in [6]. Like secondary indices, join indices can be materialized.

**Example 4:** The extended relation $r_3$ in Figure 2 is a materialized join index, which was defined as follows:

$$r_3 : = J_{r, s}(r, s, "Chair = Faculty")$$

Now we present some theorems governing the use of join indices. Theorem 9 declares that a join can be performed by doing a lookup on a join index.

**Theorem 9:**

$$\pi_{rA, sB}(r \land c, s) = \pi_{rA, sB}(\text{index}_A(r) \land \text{index}_B(s))$$

In general, the join of several extended relations can be performed by first joining their join indices and then doing a lookup. For example, Theorem 10 states the case for three extended relations:

**Theorem 10:** Let the schemes of $r_1$, $r_2$, and $r_3$ be $R_1$, $R_2$, and $R_3$, respectively.
respectively.

\[ \text{lookup}_{\text{rTid}, \text{rTid}, \text{rTid}}(J(r, t, c_1), r) \]

\[ J(r, t, c_2)(x_1, r_3) \]

\[ \square \]

Theorem 11 declares that under some conditions a join of two extended relations can be computed by joining an index and a join index.

**Theorem 11:** Let \( c \) be a condition that mentions no attributes other than \( rA \). Then

\[ \text{lookup}_{\text{rTid}, \text{rTid}, \text{rTid}}(\text{index}_{x}(r), r) \]

\[ J(r, s, c), r, s \]

\[ \square \]

**Example 5:** Consider Figure 1, and the expression

\[ \text{lookup}_{\text{rTid}, \text{rTid}, \text{rTid}}(\text{index}_{x}(r), r) \]

\[ \text{lookup}_{\text{rTid}, \text{rTid}, \text{rTid}}(\text{index}_{x}(s), s) \]

\[ rTid=tTid \]

\[ r, s \]

\[ \square \]

Finally, we can deal with selection conditions on \( r \) and \( s \) similarly to Theorem 6:

**Theorem 12:** Suppose that condition \( c=x_1 \wedge \cdots \wedge x_n \), where each \( x_i \) mentions no attribute other than \( rA \), and that \( d=y_1 \wedge \cdots \wedge y_m \), where each \( y_j \) mentions no attribute other than \( sB \). Then

\[ \text{lookup}_{\text{rTid}, \text{rTid}, \text{rTid}}(M, r, s) \]

where

\[ M=J(r, s, c) \]

\[ \text{lookup}_{\text{rTid}, \text{rTid}, \text{rTid}}(\text{index}_{x}(r), r) \]

\[ \text{lookup}_{\text{rTid}, \text{rTid}, \text{rTid}}(\text{index}_{x}(s), s) \]

\[ \square \]

7. **Support For Traversals**

We saw in the previous section how join indices precompute a single join, that is, in order to compute the join of \( r \) and \( s \) we only need to do a lookup on their join index. However, join indices are less useful for computing several joins. As we saw in Theorem 10, the best we can do is to first take the join of the appropriate join indices and then do a lookup.

There are many applications (such as CAD/CAM) in which we need to be able to evaluate multiple joins using lookups only. We call such joins a traversal of the database. The rapid acceptance of object-oriented databases is largely due to their ability to implement traversals by allowing direct references to tuples. It has even been argued [9] that object-oriented systems are inherently better than relational ones because of this feature. However, note that both object-oriented and relational systems support object identifiers; the only difference is that the identifiers are hidden from the user in the relational model. Consequently, what relational systems really need are access structures which can imitate the direct-reference ability of object-oriented systems, and transformations with which an optimizer can automatically use these access structures. In this section we show how easily the tid-algebra makes this possible.

We wish to define an access structure for a relation \( r \) in which each tuple of \( r \) is "prejoined" with its corresponding tuples for \( s \). To do so, we use the left outer join operator, which is defined as follows:

**Definition:** The left outer join of \( r \) and \( s \) is defined as:

\[ r \bowtie \text{c } s = (r \bowtie s)^c \]

where \( r \) is a set of \( r \) tuples that do not participate in the join, extended to include null values for \( s \)'s attributes.

**Definition:** Let \( r \) and \( s \) be extended relations with \( s \) stored, and let \( c \) be a condition referring only to the attributes in \( r \) and \( s \). Let \( s \)'s scheme be \( R \). Then \( r \)'s join extension with \( s \) on condition \( c \) is given by the following view definition:

\[ r \bowtie s \]

where

\[ \text{lookup}_{\text{rTid}, \text{rTid}, \text{rTid}}(M, r, s) \]

\[ \square \]

**Example 6:**

Consider again the extended relations of Figure 1. The join extensions

\[ r6 = \text{Je}(r, s, \text{"Major=Depr"}) \]

and

\[ r7 = \text{Je}(s, t, \text{"Chair=Faculty"}) \]

are shown in Figure 4. \( \square \)

Note that if a tuple \( t \) in \( r \) refers to more than one tuple in \( s \), then there will be several tuples for \( t \) in \( \text{Je}(r, s, c) \); each tuple will have a different \( tTid \) value.

A join extension can itself be extended, so that it contains attributes referring to several relations. For example, \( \text{Je}(\text{Je}(r, p, c), q, c) \) extends \( r \) to contain the attributes \( pTid \) and \( qTid \). The following theorem asserts that the order of the extension does not matter.
Theorem 13:

\[ JE(r, p, c) = JE(JE(r, q, c), p, c) \]

Thus we can generalize our notation to include the idea of a multiway join extension, as follows:

**Definition:** The join extension of \( r \) with respect to \( (t_1, c_1), (t_2, c_2), \ldots, (t_n, c_n) \) is denoted

\[ JE(r, t_1, c_1, t_2, c_2, \ldots, t_n, c_n) \]

is defined to be

\[ JE(JE(r, t_1, c_1), t_2, c_2, \ldots, t_n, c_n) \]

Note that this definition is a generalization of the previous one. Also note that the join extension of \( r \) with respect to the empty set, i.e., \( JE(r) \), is the same as \( r \). We can thus treat ordinary relations as special cases of join extensions.

Theorem 14 states that we can use a join extension of \( r \) in place of \( r \), provided that we are not interested in the values of their tid-attributes.

**Theorem 14:** Let \( r' \) be a join extension of \( r \). Then

\[ \pi_{t1 \& \& \& t2}(r') = \pi_{t1 \& \& \& t2}(r') \]

Users of relational systems do not access the tid-attributes of relations. Consequently, an optimizer can replace any reference to an ordinary relation by a reference to a join extension of the corresponding extended relation. If a join extension \( r' = JE(r, \ldots) \) is materialized, the relation \( r \) need not be materialized: it can be recovered from \( r' \). We say that \( r \) is implemented by \( r' \). Because ordinary relations correspond to null join extensions, we can without loss of generality assume that each stored instance \( r \) has some join extension \( r' \) as an implementation.

The difference at the user level between a relational and an object-oriented system is the following: in a relational system the user sees relation \( r \), whereas an object-oriented system the user sees its implementation \( r' \). However, because the optimizer knows how to transform queries involving \( r \) to those involving \( r' \), relational queries can execute just as efficiently as object-oriented ones. The following theorems illustrate these transformations.

**Theorem 15:** Let \( r \) and \( s \) have join extensions \( r' \) and \( s' \). Let \( Z = R \times S \sim (tTd, sTd) \). If \( r' \) is a join extension of \( r \) with respect to \( (t, c) \), then

\[ \pi_{c:z}(r \times s) = \pi_{c:z}(\text{lookup}, ts(c, r', s')) \]

In fact, when join extensions refer to other join extensions, we can execute several joins using only lookups. Theorem 16 states the case for two joins.

**Theorem 16:** Let \( R, S, \) and \( T \) be schemes for \( r, s, \) and \( t \) respectively. Let \( Z = R \times S \sim (tTd, sTd, tTd) \). If \( r' \) is a join extension of \( r, s \), \( r' \) is a join extension with respect to \( (r', c), \) and \( r' \) is a join extension with respect to \( (t', c), \) then

\[ \pi_{c:z}(r \times s \times t) = \pi_{c:z}(\text{lookup}, ts(c, r', s', t')) \]

The generalization to arbitrary numbers of joins is straightforward. In general, join extensions can be used to perform **any** of joins using only lookups.

8. Summary

In this paper we have introduced extended relations, and have shown how high-level access structures can be represented by extended relations. We have presented an algebra of extended relations, namely the tid-algebra. Theorems of this algebra correspond to transformations which can be used by a query optimizer. Sections 5 and 6 showed how these transformations can encode access path selection strategies. Section 7 showed that they can also be used to encode different implementations of relations. The result is that high-level relational queries can be optimized to be as efficient as those for object-oriented languages. Moreover, this optimization is algebraic, and can be part of a rule-based query optimizer.

Although we have shown how to use high-level access structures for data retrieval, we have not described what happens to them in the presence of updates. Because high-level access structures are materialized views, we intend to explore techniques of refreshing a materialized view, that is, updating it to reconcile it with updated base relations (or other objects). The refreshing of materialized views can be specified as a set of term rewrite rules in the tid-algebra. We hope to explore the formulation of differential refresh as an optimization problem, and the implementation of a view refresher using an optimizer generator [10].

Another research area is to apply the tid-algebra to more general storage structures for relations, as in [1]. We have adopted a model of optimization in which modules based on algebraic equalities and modules based on program transformation techniques are integrated into a single rule-based optimizer [10]. We are interested in exploring the different kinds of algebras needed for the different levels of query optimization. Our conjecture is that the ability to structure major optimizer components according to different algebras will lead to a modular design significantly more flexible and powerful than current optimizer designs.

References


