 Processing Read-Only Queries Over Views With Generalization

David Goldhirsch
Laura Yedwab

Computer Corporation of America
4 Cambridge Center
Cambridge, Mass. 02142

Abstract. The traditional Query Modification approach to query processing is inappropriate for views involving generalization. We use a combination of modification and materialization for queries over such views. Furthermore, by choosing modification or materialization as part of global optimization, we permit more optimization than would be provided by a purely modifying approach.

Introduction

Generalization is an abstraction that groups conceptually related objects into a "generic" object [SMITH]. For example, students and instructors can be generalized as people.

We can devise views that represent generalization [KATZ], [DAYAL]. However, as will be seen, the standard approach of Query Modification [STONEBRAKER] will not correctly handle all queries over such views. Furthermore, even where it works, query modification may not be the best way to handle such queries.

We have developed a query processing architecture that correctly and efficiently handles these queries.

Although we developed this approach in terms of the Functional Data Model and the language DAPLEX [SHIPMAN], our observations and algorithm apply to any system supporting virtual generalization.

A Simple Data Model and Language

Consider a database of entities. Each entity is of a type that specifies the name and range of its attributes. An attribute value can be a string, integer, a reference to another entity (perhaps of a different type) or a set of any of these.

Figure 1 is a schema for such a database.

Using the notation of [SHIPMAN], queries over such a database can be expressed, procedurally, with statements that create new entities, print attribute values and iterate through sets of entities (perhaps in a specified order).

For example, the following query prints the name of each cat followed by the name (alphabetically) of each of its "friends":

for each A in ANIMAL
  where ANIMAL-SPECIES (A) = "Cat"
  loop
    print (ANIMAL-NAME (A));
    for each F in ANIMAL
      where F is in ANIMAL-FRIENDS (A)
        ordered by ANIMAL-NAME (F)
      loop
        print (ANIMAL-NAME (F));
      end loop;
    end loop;
end loop;

Expressions can contain aggregations. These are built-in operations over a set of values. A query to print the median INSTRUCTOR-SALARY taken over the set of INSTRUCTORS would be: "print (median (INSTRUCTOR-SALARY (INSTRUCTOR)))".

Views with Generalized Entities

Our system supports two kinds of virtual entities. Simple types are derived from one underlying entity type. For each underlying entity (subject to selection), the attributes of that entity are used to create one of the virtual entities.

As described in [DAYAL] and [GOLDHIRSCH],
The idea is that, for each iteration over a generalization (of u underlying types), the query is broken up into \(2^u - 1\) separate queries -- one for each component of the derivation. If the resulting queries refer to other generalizations, the algorithm is applied recursively.

Figure 5a contains a query that prints (without ordering) the EARNINGS of each PERSON. After modification, using the derivation of Figure 3, we get the three queries of Figure 5b.

The modification approach of [KATZ] and [DAYAL] does NOT correctly handle queries containing ordered iterations and/or aggregations over generalizations.

To see why ordering isn't correctly modified, imagine adding an "ordered by PERSON_NAME (P)" clause to the loop in Figure 5a. The clause would appear, referring to STUDENT and/or INSTRUCTOR, in EACH of the queries of Figure 5b. This produces three independently ordered lists, rather than the single, ordered list printed by the query in Figure 5a.

To see why aggregation fails, observe that there is no way to "decompose" a median over the three components of PERSONS. That is, no combination of medians over STUDENTS and INSTRUCTORS can be used to modify the query "print (median PERSON_EARNINGS PERSON))".

Some aggregations CAN be handled by modification. For example, the count of PERSONs is the sum of: the count of PERSONS derived from STUDENTS, the count of PERSONs derived from INSTRUCTORS and the count of PERSONS derived from both a STUDENT and an INSTRUCTOR.

**Modification is Not Always Desirable**

The "obvious" solution might be to materialize generalizations that are being aggregated and/or ordered, and to use modification for all other virtual loops.

An assumption implicit in [STONEBRAKER], [KATZ] and [DAYAL] is that, given a choice, modification is the preferred method. This is true for SIMPLE types.

However, because it drastically alters a query's architecture, modification for generalizations can produce inefficient queries.

As an example, consider the query in Figure 6a. Observe that the loops over PERSON and DWELLING can each be handled either by modification or materialization.

An optimization goal might be to minimize the number of joins [RIESI needed to process the nested loops in this query. In this example, we can show that it takes more joins to run the modified queries than it would to materialize PERSON and DWELLING (running the query as is).

Modifying the loop over PERSON results in 3 queries each over DWELLING, the components of PERSON (STUDENT and INSTRUCTOR) and CAT. Each query has 4 loops and represents 3 joins.

For each query, applying modification to eliminate the DWELLING loop produces 3 queries each over STUDENT, INSTRUCTOR, CAT and the components of DWELLING. This is a total of \(3 \times 3 = 9\) queries. Each query contains 5 loops and represents 4 joins. The total number of joins is \(9 \times 4 = 36\).

Assume we use only modification for a query of \(L\) loops. Assume that \(g\) of these loops range over generalizations of \(u\) underlying types. The result is \((2^u - 1)^L\) queries each with \((L + g(u-1))\) loops. This represents a total of

\[
(2^u - 1)^L(L + g(u-1) - 1) + 0(2^u)
\]

joins.

On the other hand, using a "worst case" strategy (see [GOLDFIRSH]), we can materialize the PERSONs and DWELLINGS each using 6 joins. The query itself contains 3 loops and represents 2 joins. So, to materialize PERSON, materialize DWELLING and then run the original query over the materialized entities, we need \((6 + 6) - 1 = 14\) joins.

In general, assuming a worst case strategy, we can handle a query of \(L\) loops (of which \(g\) are generalizations of \(u\) underlying types) using

\[
g(2^u - 1)(u-1) + L - 1 = 0(g2^u)\]

joins.

As the number of generalizations in a query
grows, the number of joins after modification grows exponentially. The number of joins for materialization grows only linearly.

The point is not that materialization is therefore "better" than modification, or vice versa, but that there are cases where each might be better than the other. Arbitrarily preferring one method can significantly detract from the optimization of the query.

A QUERY PROCESSING ARCHITECTURE

We need a processing scheme that sensibly chooses modification or materialization for each loop (over a generalization) in a query.

The traditional plan works in three steps: (1) view-map the query into one or more queries that refer only to existing entities, (2) optimize, or produce a "strategy" for running the resulting query (queries) and (3) execute the query (queries) according to the strategy.

This architecture is inappropriate since (a) the view-mapper cannot understand the impact of changing the architecture of the query -- nor can it calculate the cost of any particular materialization, and (b) an ordinary optimizer would not know the view-mapping algorithm.

We choose to make step (2) a "hybrid" of optimization and deferred modification. As an illustration, consider the processing of the query in Figure 6a.

The view-mapper proceeds, iteration by iteration, to resolve references to virtual entities. Say, for example, it first recognizes CAT as a SIMPLE virtual type. It modifies the CAT loop, creating an appropriately restricted loop over ANIMAL.

Figure 6b shows the query after the modification of the CAT loop.

Notice that "A is in PERSON-CATS (P)" is, at this stage of the view-mapping, type-inconsistent: the range of PERSON-CATS is the set of CATS, not the set of ANIMALS.

The view-mapper now sees the loops over generalizations DWELLING and PERSON. With each of these, it associates a copy of the corresponding BOJ+ derivation.

To avoid the type inconsistencies, we view-map these copies. In particular, the definition of PERSON_CAT (in PERSON's BOJ+) refers to virtual type CAT. Since CAT is a SIMPLE type, the definition is handled by modification. For example, where a PERSON is both a STUDENT and a INSTRUCTOR, the definition becomes:

PERSON_CATS :=
(A in ANIMAL where ANIMAL_SPECIES(A) = "Cat"
and
(ANIMAL_NAME(A) is in
STUDENT_ANIMAL_NAMES(S)
or
ANIMAL_NAME(A) is in
INSTRUCTOR_ANIMAL_NAMES(I)))

After the view-mapping of PERSON's derivation, the range of PERSON_CATS has effectively been changed from CATS to ANIMALS. The view-mapped query is now type-consistent.

The query optimizer now considers all (or many) of the different processing strategies for the query. This includes choosing materialization or modification for each loop over a generalization.

To make this choice, the optimizer first decides which loops must be handled by materialization (none, in this example). For each remaining loop, it uses the Deferred Modifier to compare the (global) effects of using modification with the cost of materializing the generalization.

In this example, if nested loops are processed by joins, the optimizer might decide to materialize both PERSONs and DWELLINGs.

CONCLUSIONS

The ability to define and query virtual generalizations is desirable. Our approach for handling queries over virtual entities is significantly different from previous proposals:

1. To handle generalized entities correctly, we use both materialization and modification.
2. To do so efficiently, our optimizer collaborates with the view-mapper to decide whether an entity is to be materialized or modified.

ACKNOWLEDGEMENT

We are grateful to Arvola Chan, Umeshwar Dayal, Steven Fox, Nat Goodman, Steven Huberman, Terry Landers, Dan Ries, Diane and John Smith for their comments and suggestions during the writing of this paper.

STUDENT
student_name: string;
student_expenses: integer;
student_animal_names: set of strings;

INSTRUCTOR
instructor_name: string;
instructor_salary: integer;
instructor_animal_names: set of strings;

ANIMAL
animal_name: string;
animal_species: string;
animal_friends: set of ANIMALS;

Figure 1. A Database Schema.
(CAT_NAME := ANIMAL_NAME (A));
end loop;
end derive;

Figure 2. A View with a SIMPLE derivation.

PERSON
person_name: string;
person_earnings: integer;
person_cats: set of CATS;

DWELLING
dwelling_occupants: set of strings;
dwelling_addr: string;

derive PERSON from
for P in BOJ+ (S in STUDENT, I in INSTRUCTOR)
using join predicate:
"STUDENT_NAME (S) = INSTRUCTOR_NAME (I)"

case P is a STUDENT but not an INSTRUCTOR:
PERSON_NAME:= STUDENT_NAME (S),
person_earnings:= - STUDENT_EXPENSES (S),
person_cats:= (C in CAT where
CAT_NAME (C) is in
STUDENT_ANIMAL_NAMES (S))
case P is an INSTRUCTOR but not a STUDENT:
PERSON_NAME:= INSTRUCTOR_NAME (I),
person_earnings:= INSTRUCTOR_SALARY (I),
person_cats:= (C in CAT where
CAT_NAME (C) is in
INSTRUCTOR_ANIMAL_NAMES (I))
case P is both a STUDENT and an INSTRUCTOR:
PERSON_NAME:= STUDENT_NAME (S),
person_earnings:= INSTRUCTOR_SALARY (I) - STUDENT_EXPENSES (S),
person_cats:= (C in CAT where
CAT_NAME (C) is in
INSTRUCTOR_ANIMAL_NAMES (I)
or
CAT_NAME (C) is in
STUDENT_ANIMAL_NAMES (S))
end loop;
end derive;

Figure 3. A View with GENERALIZED derivations.

for each C in CAT
ordered by CAT_NAME (C)
loop
print (CAT_NAME (C));
end loop;

Figure 4a. A Query over SIMPLE type CAT.

for each A in ANIMAL
where ANIMAL_SPECIES (A) = "Cat"
ordered by ANIMAL_NAME (A)
loop
print (ANIMAL_NAME (A));
end loop;

Figure 4b. After the query has been MODIFIED.

for each P in PERSON loop
print (PERSON_EARNINGS (P));
end loop;

Figure 5a. A Query over GENERAL type PERSON.

for each S in STUDENT where
for every I in INSTRUCTOR:
INSTRUCTOR_NAME (I) # STUDENT_NAME (S)
loop
print (- STUDENT_EXPENSES (S));
end loop;

for each I in INSTRUCTOR where
for every S in STUDENT:
STUDENT_NAME (S) # INSTRUCTOR_NAME (I)
loop
print (INSTRUCTOR_SALARY (I));
end loop;

Figure 5b. The same query, after MODIFICATION.

for each D in DWELLING loop
for each P in PERSON
where PERSON_NAME (P) is in DWELLING_OCCUPANTS (D)
loop
print (PERSON_NAME (P));
for each C in CAT
where C is in PERSON_CATS (P)
ordered by CAT_NAME (C)
loop
print (CAT_NAME (C));
end loop;
end loop;
end loop;

Figure 6a. A Query over several entity types

for each D in DWELLING loop
for each P in PERSON
where PERSON_NAME (P) is in DWELLING_OCCUPANTS (D)
loop
print (PERSON_NAME (P));
for each A in ANIMAL
where ANIMAL_SPECIES (A) = "Cat"
and A is in PERSON_CATS (P)
ordered by ANIMAL_NAME (A)
loop
print (ANIMAL_NAME (A));
end loop;
end loop;
end loop;

Figure 6b. The query after MODIFYING for CATs.
BIBLIOGRAPHY


