Incremental Maintenance of Materialized OQL Views

M. Akhtar Ali
Dept. of Computer Science
University of Manchester
Manchester M13 9PL, UK
akhtaram@cs.man.ac.uk

Alvaro A.A. Fernandes
Dept. of Computer Science
University of Manchester
Manchester M13 9PL, UK
alvaro@cs.man.ac.uk

Norman W. Paton
Dept. of Computer Science
University of Manchester
Manchester M13 9PL, UK
norm@cs.man.ac.uk

ABSTRACT

The importance of materialized views has grown significantly with the advent of data warehousing and OLAP technology. This increases the relevance of solutions to the problem of incrementally maintaining materialized views. So far, most work on this problem has been confined to relational settings. Proposals that apply to object databases have either used non-standard models or fallen short of providing a comprehensive framework. This paper contributes a solution to the incremental view maintenance problem for a large class of views expressed in OQL, the query language of the ODMG standard for object databases. The solution applies to immediate update propagation, and works for any update operation on views defined over a substantial subset of ODMG types. The approach presented has been fully implemented and preliminary performance results are reported.

1. INTRODUCTION

A view definition facility is valuable for many database applications. It has been known for some time that, in certain situations, it is more profitable to materialize a view than to compute its extent every time the view is used. The problem then arises of propagating to the materialized view (MV) any changes made to the entities over which the MV is defined. It can be very costly to re-materialize the entire MV every time a change has been made that might affect it. For this reason, it is generally desirable to propagate the changes incrementally, i.e., to compute the changes needed in the MV and to effect these only, rather than materializing the entire view again. Interest in MVs has increased as a result of a growing awareness of the important role that they might play in data warehousing contexts. This importance is even greater if one considers distributed architectures, in which network bottlenecks are likely to make the efficiency gains accruing from view materialization even more desirable. Finally, view materialization becomes almost unavoidable in information integration contexts, in which it may be infeasible in practice to compute again and again a view defined on multiple, often remote, data sources. Our work is motivated by the need to manage data replication in the context of a scientific data warehousing project focusing on genomic data [15] in which case the challenges posed by the complex structure of the data and the need to model complex behaviour make the object-oriented approach the best option by far.

The paper is structured as follows. Section 2 provides a brief introduction to the technical background. Section 3 describes a solution to the IVM problem using an algebraic approach. Section 4 presents some preliminary performance results for the implemented solution. Section 5 describes briefly the research context for the contributions presented in the paper and contrasts these with work by other researchers. Section 6 concludes and points to future work.

DOLAP '2000 Washington, D.C, USA

2. TECHNICAL BACKGROUND

This section introduces examples of ODL and OQL material used in the paper and briefly describes the query processing notions used in the solution, specifically those implemented in OPTGEN [7, 8] and λ-DB [9].

The ODMG standard for object databases [5] includes an object definition language (ODL) and an object query language (OQL). Some familiarity with this model is assumed in the paper. ODL is used to declare a schema which defines the valid application types. Figure 1 is an example ODL schema.

```
class person
   ( extent Persons ){
      attribute string name;
      attribute unsigned short age;
      attribute string address;};
class department
   ( extent Departments ){
      attribute string name;
      attribute string address;
      relationship set<employee> employees
         inverse employee::dept;};
class employee extends person
   ( extent Employees ) {
      attribute unsigned long salary;
      attribute set<person> dependents;
      relationship department dept
         inverse department::employees;
      attribute employee manager;};
```

Figure 1: Example ODL Schema

Figure 2 depicts an example OQL query over Figure 1 and a view defined using the query. In OQL, a view is defined by the define/as construct, which assigns a name to an OQL query and specifies which variables appearing in the latter provide data values to the view.

```
select struct(E:e.name, D:d.name)
from e in Employees, d in Departments
where e.dept = d;

define worksFor() as
   select struct(E:e.name, D:d.name)
   from e in Employees, d in Departments
   where e.dept = d;
```

Figure 2: OQL Query and View

The λ-DB system [9] implements a monoid comprehension calculus [8] and a monoid algebra over the ODMG object model (not including arrays and dictionaries). OQL queries in λ-DB are first translated into expressions in the calculus, normalized and then translated into monoid algebraic expressions. These can be optimized before they are mapped to physical execution plans. An execution plan is then translated into C++ and compiled. The resulting executable, when run, evaluates the original OQL query. λ-DB adopts a query processing approach closely resembling that used in mainstream DBMSs but acts essentially as translator into source code that exploits the functionality provided by the SHORE object DBMS [4]. The solution presented in this paper manipulates the algebraic expressions that λ-DB generates for OQL queries. For details on the monoid algebra see [7, 8].

```
reduce(bag,
  join(bag,
    get(bag, Employees, e, and()),
    get(bag, Departments, d, and()),
    and(eq(e.dept, d)),
    none),
  X1,
  struct(bind(E, e.name), bind(D, d.name)),
  and())

reduce(bag,
  nest(bag,
    unnest(bag,
      get(bag, Employees, e, and()),
      c,
      project(e, dependents),
      and(),
      true),
    X2,
    project(c, name),
    vars(e),
    and()),
  X1,
  struct(bind(E, e.name), bind(Dep, X2)),
  and())
```

Figure 3: Algebraic Form of Figs. 2 and 4

For the view in Figure 2, the corresponding algebraic expression is given at the top of Figure 3, in which the algebraic operators reduce, join, and get occur. The monoid used by all of them is bag. The indentation in Figure 3 indicates tree levels. Thus, the root reduce node has one join node as child, and this, in turn, has two get nodes as children. As well as the monoid, a get operator specifies the extent it scans (e.g., Employees), a variable ranging over the objects in that extent (e.g., e), and a retrieval condition (e.g., and(), in this case a vacuously true one). As well as the monoid, the join operator specifies its two input arguments (e.g., the values of get expressions), the join condition (e.g., and(eq(e.dept, d))), and a specification of the kind of join to be performed (e.g., a natural join is denoted by the keyword none). Besides the monoid and the retrieval condition (vacuously true in the example), the reduce operator defines a structure for the results returned (e.g., struct(bind(E, e.name), bind(D, d.name))) and an internal variable (e.g., X1) that can be used to refer to its instances. Such a structure consists of bind expressions (e.g., bind(E, e.name)) in which a user-defined attribute name (e.g., E) is bound to the value requested via a user-defined expression (e.g., e.name).

The algebraic expression at the top in Figure 3 is not very different from a relational one. In contrast, the bottom one (corresponding to the OQL query in Figure 4) makes use of nest and unnest operators. The unnest operator iterates over a collection of objects and then for each object flattens a collection projected from the object into its component elements. The expression `c in e.dependents` in the subquery maps to an unnest operator. The nest operator is essentially the inverse of unnest: it constructs a new collection out of the elements supplied. For example, for the query in Figure 4, the nest operator is used in Figure 3 to construct a collection of type bag<string>, which is then assigned to the user-defined variable Dep.

```
select struct(E:e.name, Dep:(
      select c.name
      from c in e.dependents
)) from e in Employees
```

Figure 4: Nest and Unnest Query

3. IVM IN ODMG DATABASES

The solution to the incremental view maintenance (IVM) problem presented here assumes, as do all other solutions, the availability of the update event, the changes made to the database (hereafter referred to as the *delta)*, the MV definition, and the current materialized state of the view. In order to achieve the goal of incremental maintainability for all update operations, we also assume the availability of references to the base objects that contribute data to the MV, and of the base extents required for materializing it (although there is no need to recourse to base objects and extents for certain kinds of update). Our solution works for MVs that refer to any ODL-definable type (excluding array and dictionary collections) and that are definable using the reduce, join, get, nest and unnest bulk operators of the monoid algebra proposed in [7, 8] (excluding self joins). It is valid for any update operation in the ODMG standard (e.g., `new()` and `delete()` on objects, `insert_element()` on collections, etc.). In terms of practicality, our solution yields incremental maintenance plans (IMPs) at the algebraic level and, to the best of our knowledge, for object-based solutions, is the first one to do so. This makes it easier to integrate our solution into the kind of query processing frameworks that mainstream database management systems (DBMSs) rely on.

When an MV is defined (and subsequently compiled, evaluated and materialized), its definition is traversed to characterize the kinds of update events that might require propagation of changes to the MV. Such events include derived ones, that arise in the ODMG context as a result of the database engine being required to enforce referential integrity constraints declared by means of the `relationship/inverse` construction. For each kind of event, an algebraic IMP is constructed that can compute the required changes to the MV. The core of our solution is, therefore, the generation of an IMP that is appropriate for each kind of update event. In our approach, two kinds of IMP suffice to compute the changes required in the MV as a result of any update operation in the ODMG standard. Immediately after an update event takes place which implies the need to update an MV, the corresponding delta (comprising the old and the new state of affected objects) is made available and the associated IMP (which uses the delta) is evaluated to compute the changes needed.

The comprehensive nature of our solution with respect to update operations requires that the OIDs of objects that contribute data to the MV are also materialized. This is achieved, at view compilation time, by generating from an MV definition `v` another view definition, which we refer to as *OIDs_for_*`v`, which is itself compiled, evaluated and materialized too. Thus, the OIDs of objects that contribute data for an instance of `v` are associated with the OID of that instance in *OIDs_for_*`v`. This space overhead (which naturally induces some time overhead) is present in our solution for it to be exhaustive over the set of operations in the ODMG standard.

If `v` contains the keyword `distinct` then *OIDs_for_*`v` removes it. Thus, given an instance o of `v`, *OIDs_for_*`v` contains instances $o_1, \dots, o_n$ for each of the n distinct derivations of o. If `v` contains the keyword `order by` on an attribute a then so does *OIDs_for_*`v`. Thus, the extent of *OIDs_for_*`v` shares the ordering on a with the extent of `v`. The top part of Figure 5 shows *OIDs_for_*`worksFor`, with generated strings in italics. In turn, the generation of an IMP needs to start from an algebraic query tree in which denotations are available for the OIDs of contributing objects. Thus, rather than using the algebraic query tree for `v`, the algorithms start from the algebraic query tree resulting from the compilation of a second view definition derived from `v`, which we refer to as *OID_projecting_*`v`. It is identical to `v` except that it also includes those attributes in *OIDs_for_*`v` that originate from extents occurring in the `from` clause of `v`. If `v` does not contain the keyword `distinct` then *OID_projecting_*`v` includes it, thereby imposing a set semantics on the latter. The bottom part of Figure 5 shows *OID_projecting_*`worksFor`, again with generated strings in italics. Note that, in contrast with `v` and *OIDs_for_*`v`, *OID_projecting_*`v` is never materialized.

After extra information is derived as described, the events to be monitored are identified and their corresponding IMPs generated. Different forms of IMPs are generated depending on the update and the properties of the view, as follows:

1. **Planting a delta** – For some kinds of updates, the constructed IMP computes the changes required to `v` by evaluating *OID_projecting_*`v` over the delta to the affected base extent, rather than over the base extent itself, while accessing all other base extents referenced in the MV. For example, consider the effect of inserting a new employee into the `Employees` extent on the materialized state of `worksFor` (in Figure 2). The delta corresponding to the insertion will simply contain the new employee. The IMP generated by our solution (shown in algebraic form at the top of Figure 6) is different from the evaluation plan derived for *OID_projecting_*`worksFor` only in ranging over the delta to `Employees` rather than over `Employees` itself (but accessing `Departments` as well). When this IMP is evaluated, the result is instances of *OID_projecting_*`worksFor` which are then input to the algorithm that applies the changes to the MV (and to *OIDs_for_*`worksFor` if required).

2. **Joining a delta with materialized OIDs** – For other kinds of updates, the IMP constructed by the system joins *OIDs_for_*`v` with the delta in order to identify MV objects that are affected by the update. The idea is to avoid access to base extents whenever possible. For example, consider the effect of modifying the name of an employee. In this case (shown in algebraic form at the bottom of Figure 6), the information in the delta is not enough to identify which object in the MV might be affected because the delta only refers to the updated employee: there is no handle in the delta to the instances in the MV which have data that was contributed by the affected employee. In this case, the IMP will need to join *OIDs_for_*`worksFor` with the delta (on the OID of the object in the delta). When this IMP is evaluated, the result is input to an algorithm that applies the changes to the MV (and to *OIDs_for_*`worksFor` if required).

```
define OIDs_for_worksFor()
   select distinct struct(EO:e, DO:d, VO:v)
   from e in Employees, d in Departments,
         v in WorksFor
   where e.dept = d
         and v.E = e.name and v.D = d.name

define OID_projecting_worksFor()
   select distinct
         struct(EO:e, DO:d, E:e.name, D:d.name)
   from e in Employees, d in Departments
   where e.dept = d
```

Figure 5: Derived Views for Fig. 2

Some assumptions underpinning the remainder of this section are now made explicit. All the algorithms operate either on algebraic expressions represented as trees or on OQL expressions that comprise the query part of view definitions (e.g., those in Figure 2). The algorithms assume that, given an expression, a straightforward tokenizer can identify, for each grammatical category of interest, the set of all tokens of a given category that occur in that expression. Using Figure 3 as a source of examples, some of the categories of interest in the algebraic expression shown are operator names (e.g., join), extent names (e.g., Departments), object names (e.g., e), attribute names (e.g., employee.name, department.name), and internal names (e.g., X1). Another category of interest is collection names (e.g., `dependents` in Figure 1). Notice that when a name is not globally unique we assume that context information is concatenated to make it so, and that this concatenation is

```
reduce(set,
  join(set,
    get(set, Δ_{onInsert Employees}, e, and()),
    get(set, Departments, d, and()),
    and(eq(e.dept, d)),
    none),
  X1,
  struct(bind(EO,e),bind(DO,d),bind(E,e.name),bind(D,d.name)),
  and())

reduce(set,
  join(set,
    get(set, OIDs_for_worksFor, mat_oids, and()),
    get(set, Δ_{onModifyAttr employee.name}, δ, and()),
    and(eq(mat_oids.EO, δ)),
    none),
  X1,
  struct(bind(mat_obj,mat_oids.VO),bind(E,δ.name)),
  and())
```

Figure 6: IMPs for Fig. 2: insert, modify

construed as a token. Functional notation is used to denote metadata that might be relevant, e.g., given an attribute name a we denote the type of which a is an attribute as $typeWhereDefined(a)$, and we might go on and denote the extent name associated with that type by writing $extentNameOf(typeWhereDefined(a))$. This is in contrast with $typeOf(e)$, denoting the domain from which values for an expression e are drawn. If a is a relationship name, then we denote the inverse relationship as $inverseOf(a)$. In addition, given a collection attribute c we denote the type of the elements in the collection as $typeOfElementIn(c)$.

Figure 7 shows the processes that take place in the wake of the definition of an MV. Shaded boxes denote components that are assumed to exist and clear boxes denote new components or extensions to existing ones required for IVM. This section describes each of the clear boxes in Figure 7. The oval collects processes that take place at update propagation time.

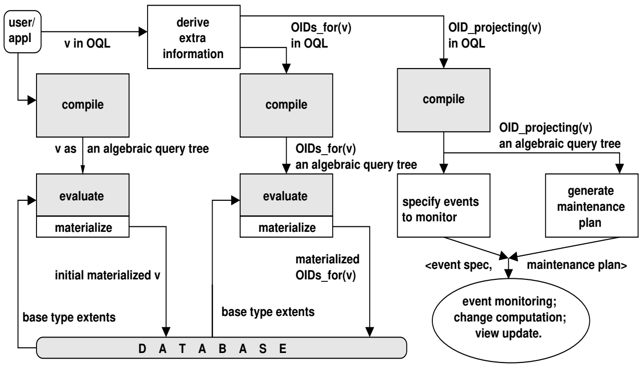

Figure 7: Processes at View Compilation Time

With respect to the extension box labelled materialize, all that is required is to extend the evaluator with a subcomponent to make persistent the instances returned as the result of executing a query. We assume that the materialization process also encompasses the setting up of types for storing the MV-specific data and metadata required by IVM processes.

The box labelled derive extra information in Figure 7 is a simple syntactic mapping to generate, from the definition of an MV `v`, the two additional view definitions, viz., $OIDs\_for\_$`v` and $OID\_projecting\_$`v`, whose meaning and purpose was described above. Note that `v` and $OIDs\_for\_$`v` are materialized (for use when IMPs are evaluated), but $OID\_projecting\_$`v` is not materialized at all and is only used in algebraic form for the construction of an appropriate IMP for each kind of event with which IMPs are paired.

Figure 8 presents the algorithms that define the functionality of the box labelled specify events to monitor in Figure 7. Abbreviations are used as follows: ***Ins*** for ***onInsertTo***, ***Del*** for ***onDeleteTo***, ***Mod*** for ***onModifyAttr***, ***InsEl*** for ***onInsertElementIn***, ***DelEl*** for ***onDeleteElementIn***, and ***ModEl*** for ***onModifyAttrInElement***.

Note that the generation of event specifications is designed to generalize, and cover, all update operations in the ODMG standard, including those on collection types other than dictionary and arrays (which are not currently defined in the monoid algebra of [7, 8]). In Figure 8, by a 'scan in a view' is meant an expression of the form '*var* `in` *VScope*' or '*VScope* `as` *var*'. In Figure 8, by a 'path-expression in a view' is meant a period-separated list of names, each of which is either a *var* introduced in a scan or an attribute name introduced in the schema definition. Relationship names and the names of their corresponding inverse relationships are also referred to as attribute names in Figure 8 and elsewhere in the paper. The function $attributesProjected(C, Q)$ is only informally defined here. It returns the set of attribute names projected from a type C in a query Q. Figure 9 shows (as arrow targets) the event specifications returned by **specifyEventsToMonitor** for the view at the bottom of Figure 5.

The arrow sources are the syntactic features in the view definition that induce the event specifications in the target. To illustrate the execution of the algorithms in Figure 8, notice that **specifyEventsToMonitor** comprises the following stages: first, updates to extents scanned in the view need to be monitored — this is the purpose of the first iteration (over scans in the view); and second, updates to attributes (possibly collection ones, and possibly relationships and their inverses) mentioned in the view need to be monitored — this is the purpose of the second iteration (which, for each path expression in the view definition, invokes **specifyEventsInPathExpr** to traverse the path expression recursively). Basically, scans induce the need to monitor inserts and deletes on extents or collections, attribute names induce the need to monitor modifications and, if their type is not primitive, updates to instances of the type they are defined on.

Recall that there are two types of IMP for an MV `v` in our approach: one that plants a delta in $OID\_projecting\_$`v` and the other that joins the delta with $OIDs\_for\_$`v`. The box labelled generate maintenance plan in Figure 7 implements a function that, depending on the event type and the MV type, decides whether to use one type of IMP or the other. Figure 10 defines the condition under which the MV type is derived from the algebraic expression representing the MV. In this table, '$\surd$' means that the operator in that row appears in the algebraic expression where as '$\times$' means that it does not. For example, the MV in Figure 2 is a ReduceGetJoin view because its algebraic form, shown in Figure 3, contains the reduce, get and join operators. Where a view can be classified in more than one type, the rightmost

```
function specifyEventsToMonitor(MVD: OQLQuery)
  → set[eventSpecification] ≡
  E: set[eventSpecification] := ∅; arg: string;
  foreach scan s in MVD do
    if  (VScope(s) is an extent name) then
      E := E ∪ {Ins VScope(s), Del VScope(s)};
    fi ; od ;
  foreach path-expr p in MVD do
    E := E ∪ specifyEventsInPathExpr(p, MVD); od ;
return E; where
function specifyEventsInPathExpr(p: path-expr, Q: OQLQuery)
  → set[eventSpecification] ≡
  E: set[eventSpecification] := ∅; Attrs: set[string] := ∅; arg: string;
  cases of p such that
  when (p is empty) return E;                  /* base case */
  when (head(p) is a var)   /* done, by iterating over scans */
    E := E ∪ specifyEventsInPathExpr(rest(p), Q);
  else    /* so, head(p) is an attribute name, let's handle it */
    if  (typeOf(head(p)) is not a collection type) then
        if  (typeOf(head(p)) is not primitive) then
              /* head(p) names a single-valued relationship */
            E := E ∪ { Mod head(p) };
            arg := extentNameOf(typeWhereDefined(head(p)));
            E := E ∪ { Ins arg, Del arg };
            arg := inverseOf(head(p));
            if  (arg is not a collection type) then
                E := E ∪ { Mod arg };
              else        /* inverse is of a collection type) */
                E := E ∪ { InsEl arg, DelEl arg }; fi ;
          else
            E := E ∪ { Mod head(p) }; fi ;
      else              /* typeOf(head(p)) is a collection type */
        E := E ∪ {InsEl p, DelEl p};
        arg := typeOfElementIn(head(p));
        Attrs := attributesProjected(arg, Q);
      /* now, changes to attributes from collection elements */
        foreach attribute a in Attrs do
          E := E ∪ { ModEl a(p) }; od ;
        if  (arg is not primitive) then
          /* head(p) names a collection-valued relationship */
            arg := extentNameOf(arg);
            E := E ∪ { Ins arg, Del arg };
            arg := inverseOf(head(p));
            if  (arg is not a collection type) then
                E := E ∪ { Mod arg };
              else        /* inverse is of a collection type) */
                E := E ∪ { InsEl arg, DelEl arg }; fi ;
 /* head(p) does not name a relationship and has been handled */
          else  continue  fi ;
    fi ;
    E := E ∪ specifyEventsInPathExpr(rest(p), Q);
  esac; return E; □
```

Figure 8: Specifying Events to Monitor

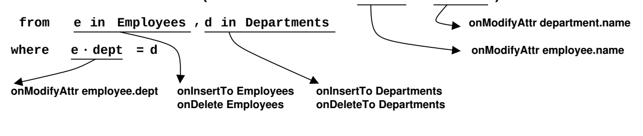

Figure 9: Event Specifications for Fig. 5

(in Figure 10) among these is used.

	ReduceGet	ReduceGetJoin	Unnest	Nest
reduce	√	√	√	√
get	√	√	√	√
join	×	√	√	√
unnest	×	×	√	√
nest	×	×	×	√

Figure 10: Classification by Algebraic Operators

Figure 11 defines precisely, for each event type and for each MV type, what kind of IMP is generated. In Figure 11, note that 'CA' abbreviates the phrase 'condition attribute', by which is meant an attribute occurring in a condition expression (e.g., in Figure 2, e.dept is a CA whereas e.name is not). Note, however, that a CA attribute may appear elsewhere in a query (e.g., in a projection). Also, by Δ_{new} is meant using the data in the delta reflecting the state of the database after the update. Correspondingly, by Δ_{old} is meant using the data in the delta reflecting the state of the database before the update. For example, for an `onInsertTo` event, Δ_{new} is used because the new state of the database is relevant.[1] In contrast, for an `onDeleteTo` event, Δ_{old} is used, as an object once deleted from the database can no longer be referenced, therefore, the old state is essential. Given an MV `v`, by 'plant Δ_{new}' is meant that the IMP is generated by planting Δ_{new} in the algebraic form of *OID_projecting_*`v`, and by '*OIDs_for_*`v` $\bowtie \Delta_{new}$' is meant that the generated IMP in algebraic form joins Δ_{new} to *OIDs_for_*`v`. Similar remarks apply to the use of Δ_{old}. Note also that, in some cells, two IMPs are generated. This happens whenever a CA is modified, in which case there is a need to use the old value of the attribute in one IMP (identical to the one that would have been generated if the attribute were not a CA) to reconstruct which objects made it into the MV on the basis of that value, before using the new value to generate the second IMP. It will be seen later that the output of the two IMPs characterize MV objects to delete and insert, respectively.

An example of (the algebraic form of) an IMP generated by planting a delta was given at the top of Figure 6, and of one generated by joining a delta with *OIDs_for_*`v`, in the bottom of Figure 6. Since the example event types were, respectively, `onInsertTo` and `onModifyAttr` (where the attribute in question is not a CA), and the example MV is a ReduceGetJoin view, the examples correspond, respectively, to 'plant Δ_{new}' and '*OIDs_for_*`v` $\bowtie \Delta_{new}$' in Figure 6. The intuition behind entries in Figure 11 is as follows:

1. ReduceGet: No access is required to base data, as it is sufficient to filter the data provided in the delta (only one extent is referenced in such views, which is the one replaced by the delta). The delta becomes available in the algebraic expression through planting.

2. ReduceGetJoin: Where a new object is created that may participate in the view (e.g., after an `onInsertTo`), the view must be evaluated over the delta to identify whether or not the new object should contribute to the view. This is likely to require access to base data.

[1] In some cases, it is possible to avoid accessing base data on `onInsertTo` as discussed in [16]. This is an improvement that we will explore in future work.

			View Type		
Case	Event Type	CA?	ReduceGet	ReduceGetJoin	Unnest∨Nest
1	`onInsertTo`	–	plant Δ_{new}	plant Δ_{new}	plant Δ_{new}
2	`onInsertElementIn`	–	plant Δ_{new}	$OIDs\_for\_\texttt{v} \bowtie \Delta_{new}$	plant Δ_{new}
3	`onDeleteTo`	–	plant Δ_{old}	$OIDs\_for\_\texttt{v} \bowtie \Delta_{old}$	plant Δ_{old}
4	`onDeleteElementIn`	–	plant Δ_{old}	$OIDs\_for\_\texttt{v} \bowtie \Delta_{old}$	plant Δ_{old}
5	`onModifyAttr`	no	plant Δ_{new}	$OIDs\_for\_\texttt{v} \bowtie \Delta_{new}$	plant Δ_{new}
6	`onModifyAttrInElement`	no	plant Δ_{new}	$OIDs\_for\_\texttt{v} \bowtie \Delta_{new}$	plant Δ_{new}
7	`onModifyAttr`	yes	plant Δ_{old} plant Δ_{new}	$OIDs\_for\_\texttt{v} \bowtie \Delta_{old}$ plant Δ_{new}	plant Δ_{old} plant Δ_{new}
8	`onModifyAttrInElement`	yes	plant Δ_{old} plant Δ_{new}	$OIDs\_for\_\texttt{v} \bowtie \Delta_{old}$ plant Δ_{new}	plant Δ_{old} plant Δ_{new}

Figure 11: Choosing a Maintenance Plan

Several update operations (in rows 2-6 in Figure 11) cannot change which objects contribute to the view, but may affect the data projected in the view. As such, joining the delta to $OIDs\_for\_\texttt{v}$ is sufficient to identify the objects in the view affected by the update. Modifying an attribute that is included in a condition (in rows 7-8) may cause an object that is already in the view to be deleted, or may lead to new objects being added to the view. The former can be identified by joining Δ_{old} with $OIDs\_for\_\texttt{v}$, but the latter requires access to base data through the planting of Δ_{new}.

3. Unnest ∨ Nest: As no information is materialised on the collections from which unnested members of a view are derived, incremental plans that relate to such views in general require access to base data. In the case of nesting, the materialised data does not in general link directly to the objects from which a nested collection has been constructed, so again it is necessary to access base data in order to compute changes to the view. As such, incremental changes to both unnest and nest views are computed by planting the relevant delta.

For reasons of space, we omit an explicit description of the algorithms denoted by the cells in Figure 11. Comparing the views in Figures 2 and 5 with the examples in Figure 6 gives an idea of the transformations effected by the algorithms.

In our framework, the IVM problem is broken down into processes that take place at view compilation time and processes that run whenever an update takes place. The compilation-time processes have been described in the preceding subsection. In what follows we briefly describe what happens when an update event takes place. Because λ-DB implements the ODMG model by source translation onto SHORE C++, our system does not monitor update events in the sense that this is understood, e.g., in the active databases literature. Instead, we adorn the SHORE C++ code output by the λ-DB compiler with calls to the IVM code. At view-compilation time, as previously described, MV definitions are compiled, evaluated and materialized along with extra information. In addition, event specifications are identified and the IMPs associated with the latter are generated. Then, when λ-DB compiles application code, we identify update requests and adorn application updates with calls to the IVM code. For each update request, whenever there is an MV for which the request is relevant (because it matches the event specifications derived for that MV at view compilation time), the inserted IVM code is such that, when executed, it evaluates the IMP(s) associated with that update to compute the changes (as described in the preceding subsection, and in particular in Figure 11) and then propagates those changes to the MV. The details of how changes are applied to the MVs are omitted due to space restrictions.

4. PRELIMINARY RESULTS

A preliminary performance evaluation has been carried out on the prototype of our solution to the IVM problem, the results of which are briefly described in this section. The experiments reported here are carried out on a PC with the following hardware and software: Intel Pentium Pro processor, 200 MHz, 256KB cache, 128MB RAM, 4GB SCSI Hard Disk (where the system software and 128MB of swap space reside); RedHat Linux 6.0 Kernel 2.2.5-22, SHORE 1.1.1 and λ-DB 0.5. Data was generated at random for the schema in Figure 1 and cardinalities are indicated in Figure 12. A more comprehensive evaluation in which more stringent demands are made on the system is presented in [2].

Performance is evaluated on three aspects: answering queries over materialized and non-materialized views, relating the cost of performing a query over a non-materialized view to the cost of propagating incremental updates to the corresponding MV, and incremental maintenance against rematerialization when the view needs updating. Re-materialization includes deleting the existing materialized extent, recomputing the view extent, and storing the new extent.

The MV used is the variant of `worksFor` given in Figure 13 and referred to as `olderEmpThatWorksFor`.

Figure 12.(a) compares the performance of evaluating queries that retrieve the result of `olderEmpThatWorksFor` in materialized (MQ) and virtual (VQ) forms. These results indicate significant response time gains for querying the materialized version of the view for different database sizes. Figure 12.(b) shows the relative cost of answering VQ and propagating an `onInsertTo` update event incrementally. The idea is to identify under what circumstances MVs may be detrimental to performance in update-intensive applications. For the small database, 31 incremental updates can be performed for the cost of evaluating VQ once. For the medium database, this increases to 54, and to 76 for the large database. This suggests that IVM is more beneficial in larger databases. Figures 12.(c)-(d) show the performance of IVM against rematerialization (ReMat) in response to `onInsertTo` and `onModifyAttr` update events, respectively. In both cases IVM significantly outperforms ReMat. For example, in the case of `onInsertTo`, IVM is 55 times faster than ReMat for the small database, and 132 times faster for the large one. In other words, 55 incremental updates can be performed for

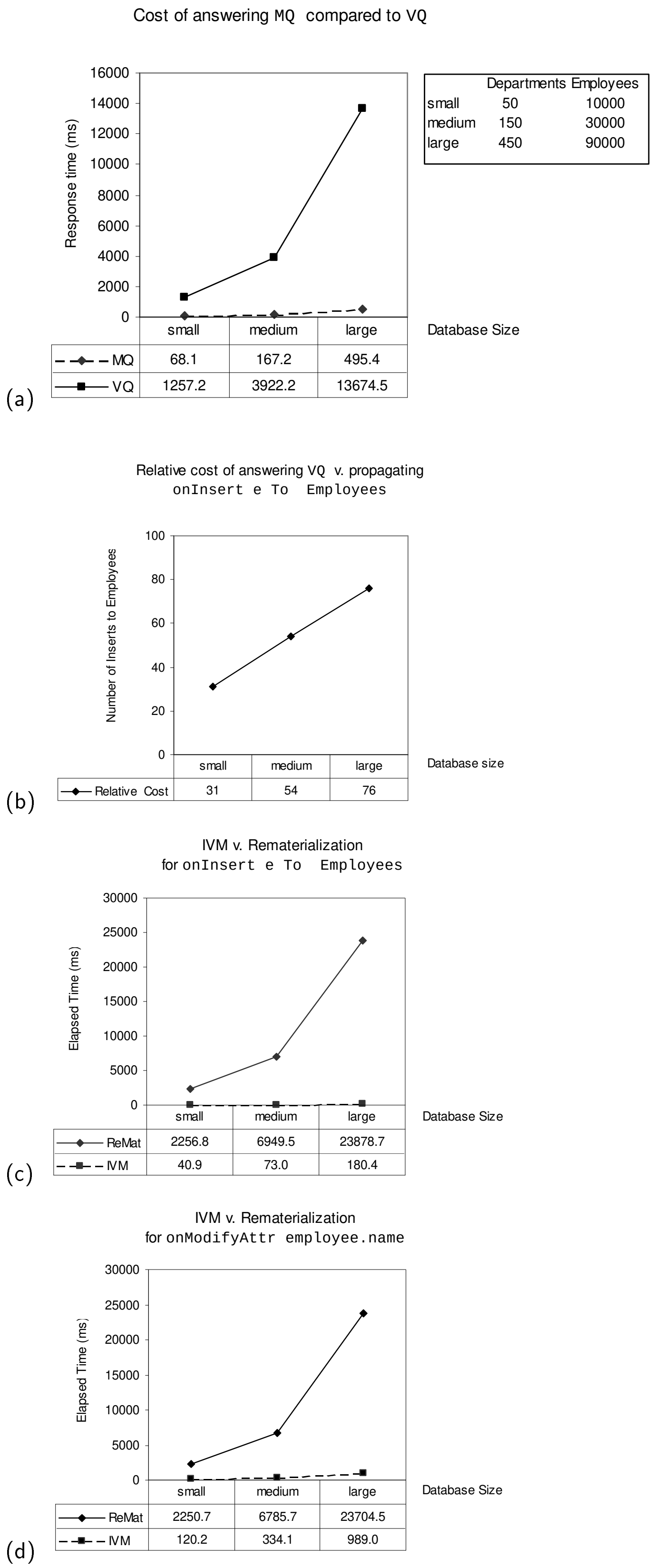

Figure 12: Preliminary Evaluation of the IVM Code

```
define olderEmpThatWorksFor() as
   select struct(E:e.name, D:d.name)
   from e in Employees, d in Departments
   where e.dept = d and e.age >= 39;
```

Figure 13: MV Used for Performance Evaluation

the cost of one re-materialization event in the case of the small database and 132 in the case of the large one. This suggests that the benefit of incrementality increases with an increase in the database size.

5. RELATED WORK

In this section, we discuss work done on the IVM problem that is closely related to the work reported in this paper.

The incremental maintenance of materialized views has been studied in detail for relational DBMSs. Blakeley et al. [3] provide a differential algorithm for maintaining select-project-join (SPJ) views [3]. Our approach of generating an IMP by planting a delta is closely related to this differential algorithm; however, it is also effective in the richer OO setting. Ceri and Widom derive production rules to incrementally update a subset of SQL views [6]. Although their system employs active database technology and key information for all tables referenced in view definitions, it does not avoid access to base tables whereas in our solution views can be incrementally maintained without accessing base tables for some events. The algorithms provided by Gupta et al. [13] incrementally maintain SQL and Datalog views. However, they require access to all base tables for all update operations, which could be more expensive than our approach, since we avoid accessing base data for some updates. Zhuge et al. propose an *eager compensation algorithm* [18] that adapts [3, 13] to data-warehousing environments. Once again, the possibility of avoiding access to base tables is not exploited. In this case, this could be even more expensive because in data-warehousing environments base tables may well reside remotely. Griffin and Libkin [11] extend relational algebra to bags and present an algorithm that derives algebraic expressions to compute changes to views. Although [11] closely resembles [13] in the expressiveness of the language and the amount of information they assume to be available, it provides an algebraic approach to the IVM problem, as we do. Finally, Gupta and Blakeley [12] extend [3] with the possibility of using less information in incrementally maintaining SPJ views. However, they either materialize an extended form of the MV (besides the MV itself) to store the values of non-projected base attributes, or use partial base tables in order to consume less information. In contrast, we store only the OIDs of the contributing objects and avoid accessing base extents whenever possible.

Recently, a few proposals have addressed view maintenance in the context of object DBMSs [1, 10, 14, 17]. These constitute the work more closely related to ours. Zhou et al. proposes a system called Squirrel [17] that uses a view definition language based on OQL. Although OQL syntax is adopted, the underlying query engine is relational and does not support views containing nest or unnest operations. As well as materialized views, the Squirrel mediators store locally auxiliary information that is generated by intermediate nodes (e.g. projections, joins) in the algebraic representation of the query. An incremental update plan maintains both the view and the auxiliary information. In our system, we do not store results of intermediate nodes. Squirrel propagates incremental updates to MVs in response to inserts and deletes only, whereas we support all update operations defined by the ODMG. Gluche et al. address the IVM problem for a subset of OQL views at the monoid comprehension level in the context of their CROQUE system for a variant of the ODMG object model [10]. The emphasis of this work is on understanding properties of OQL queries with aggregation operations so as to determine under what circumstances views can be materialized incrementally. Thus, they do not provide a directly implementable solution to the IVM problem, but rather establish foundational results that a solution might want to build on. Kuno and Rudensteiner provide a system called MultiView [14] that is based on a non-standard object model and query language, whose algebraic operators are very close to those of relational algebra. MultiView considers views to be references to base objects, therefore it does not follow the mainstream approach of replicating actual data in the view. Such views are less useful in data warehousing and data integration contexts because here replication is part of the purpose and the need to dereference object identities, while greatly simplifying the IVM task, makes it more difficult to reap certain benefits expected of data warehouses. MultiView requires access to all base extents referenced in MVs (not containing joins) in order to propagate updates incrementally for the supported update operations, whereas we avoid accessing base extents for views containing arbitrary number of joins for certain update operations. Finally, Alhajj et al [1] present algorithms for the incremental maintenance of SP views in the context of a non-standard object model and query language whose expressiveness falls short of the needs of the very applications that have provided justification for the move from relational to object-based databases. The emphasis in [1] is on the recording of information that is necessary when performing deferred updates, and the proposal is imprecise with respect to both the view language and the process of update propagation.

6. CONCLUSIONS

The solution presented here has the following salient points in relation to previous work. It is one of few that cover the IVM problem from event specification, to IMP generation, to update propagation; it is the first algebraic approach to the IVM problem in the context of object databases; it works for a wider class of MVs that the only previous work on the IVM problem for OQL views (viz., Squirrel [17]); it also has lower space overheads than Squirrel [17] for the MVs that both solutions cover; it is fully implemented as an extension of λ-DB [9] whereas many IVM proposals give no evidence of associated implementations; it is one of few whose performance has been evaluated over all the steps comprising the IVM problem [2].

We plan to build on the contributions above by providing formal results of the validity of the solution (currently validity is ensured by the empirical testing of the implemented system alone) and extending the subset of OQL that can be used for view definition with self joins as well as aggregation functions, set operations and quantification.

Acknowledgements M. Akhtar Ali gratefully acknowledges the support of the Commonwealth Scholarship Commission in the United Kingdom (Grant PK0257).

7. REFERENCES

[1] R. Alhajj and F. Polat. Incremental View Maintenance in Object Databases. *Data Base for Advances in Information Systems*, 29(3):52–64, 1998.

[2] M. A. Ali, N. W. Paton, and A. A. A. Fernandes. Evaluating An Incremental View Maintenance System. Technical Report (PrePrint Series), Department of Computer Science, University of Manchester, August 2000. http://pevepc13.cs.man.ac.uk/PrePrints/index.htm.

[3] J. A. Blakeley, P.-A. Larson, and F. Tompa. Efficiently Updating Materialized Views. In *Proc. SIGMOD*, pages 61–71, 1986.

[4] M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwilling. Shoring up persistent applications. In *Proc. SIGMOD*, pages 383–394, 1994.

[5] R. G. G. Cattell, editor. *The Object Database Standard: ODMG 3.0.* Morgan Kaufmann, 2000.

[6] S. Ceri and J. Widom. Deriving Production Rules for Incremental View Maintenance. In *Proc. VLDB*, pages 577–589, 1991.

[7] L. Fegaras. Query Unnesting in Object-Oriented Databases. In *Proc. SIGMOD*, pages 49–60, 1998.

[8] L. Fegaras and D. Maier. Towards an Effective Calculus for Object Query Languages. In *Proc. SIGMOD*, pages 47–58, 1995.

[9] L. Fegaras, C. Srinivasan, A. Rajendran, and D. Maier. λ-DB: An ODMG-Based Object-Oriented DBMS. In *Proc. SIGMOD*, page 583, 2000. SIGMOD Record 29(2), June 2000.

[10] D. Gluche, T. Grust, C. Mainberger, and M. Scholl. Incremental Updates for Materialized OQL Views. In *Proc. DOOD*, pages 52–66, 1997.

[11] T. Griffin and L. Libkin. Incremental Maintenance of Views with Duplicates. In *Proc. SIGMOD*, pages 328–339, 1995.

[12] A. Gupta and J. A. Blakeley. Using Partial Information to Update Materialized Views. *Information Systems*, 20(8):641–662, 1995.

[13] A. Gupta, I. Mumick, and V. Subrahmanian. Maintaining Views Incrementally. In *Proc. SIGMOD*, pages 157–166, 1993.

[14] H. Kuno and E. Rudensteiner. Incremental Maintenance of Materialized Object-Oriented Views in MultiView: Strategies and Performance Evaluation. *IEEE TKDE*, 10(5):768–792, 1998.

[15] N. W. Paton, S. A. Khan, A. Hayes, F. Moussouni, A. Brass, K. Eilbeck, C. A. Goble, S. J. Hubbard, and S. G. Oliver. Conceptual Modelling of Genomic Information. *Bioinformatics*, 16:548–557, 2000.

[16] I. Stanoi, D. Agrawal, and A. E. Abbadi. View Derivation Graph with Edge Fitting for Adaptive Data Warehousing. In *Proc. DaWaK*, 2000.

[17] G. Zhou, R. Hull, and R. King. Generating Data Integration Mediators that Use Materialization. *JIIS*, 6(2/3):199–221, 1996.

[18] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View Maintenance in a Warehousing Environment. In *Proc. SIGMOD*, pages 316–327, 1995.