



ELSEVIER

Available at

[www.ElsevierComputerScience.com](http://www.ElsevierComputerScience.com)

POWERED BY SCIENCE @ DIRECT®

Data & Knowledge Engineering 47 (2003) 131–166

**DATA &  
KNOWLEDGE  
ENGINEERING**

[www.elsevier.com/locate/datak](http://www.elsevier.com/locate/datak)

## MOVIE: An incremental maintenance system for materialized object views

M. Akhtar Ali <sup>a</sup>, Alvaro A.A. Fernandes <sup>b,\*</sup>, Norman W. Paton <sup>b</sup>

<sup>a</sup> *School of Informatics, University of Northumbria, Newcastle Upon Tyne NE1 8ST, UK*

<sup>b</sup> *Department of Computer Science, University of Manchester, Oxford Road, Manchester M13 9PL, UK*

Received 21 August 2002; received in revised form 13 November 2002; accepted 29 January 2003

---

### Abstract

View materialization is an important technique for high performance query processing, data integration and replication. Solutions to the problem of incrementally maintaining materialized views are very relevant. So far, most work on this problem has been confined to relational settings and solutions have not been comprehensively evaluated. This paper describes MOVIE, a complete, implemented and evaluated solution to the problem of incrementally maintaining materialized OQL views in ODMG-compliant object databases. The evaluation throws light into how the effectiveness of incremental maintenance is affected by issues such as database size, and the complexity and selectivity of views.

© 2003 Elsevier B.V. All rights reserved.

*Keywords:* Object database; Materialized view; Incremental view maintenance; Design, implementation and performance evaluation

---

### 1. Introduction

This section briefly describes the context in which the problem addressed by the paper is particularly relevant, the motivations for the particular solution that the paper reports and, broadly, the contributions that distinguish this solution from previous work on the problem.

---

\* Corresponding author.

*E-mail address:* [a.fernandes@cs.man.ac.uk](mailto:a.fernandes@cs.man.ac.uk) (A.A.A. Fernandes).

A view definition facility is valuable for many database applications. 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 costly to re-materialize the entire view every time a change has been made that might affect it.

For this reason, it is often desirable to propagate the changes incrementally, i.e., to compute the changes needed in the MV and to effect only those changes, 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. If one considers distributed architectures in general, in which replication is a useful strategy for addressing resilience issues and network bottlenecks can be expected, the efficiency and resilience gains accruing from view materialization may be even more important.

Certain aspects of the incremental maintenance of MVs have been studied in detail for relational database management systems (DBMSs) as well as in a deductive setting. Work on an object-based setting is significantly scarcer. Also, equal attention has not always been given to the various sub-problems. Most previous work concentrates on techniques to ascertain the relevance of update events and to compute the corresponding changes when such relevance is characterized. This activity has yielded many proposals for techniques, architectures and algorithms for maintaining the consistency of MVs with the base data which determined their extent. Some of these proposals have been implemented, but many of the techniques have never been subject to evaluations in experimental or operational contexts. Thus, there are, currently, surprisingly few completely described solutions, few solutions other than in relational or deductive settings, and few empirical performance analyses. As a result, understanding of the challenges raised by each of the sub-problems involved in incrementally maintaining MVs and by the change from a relational or deductive to an object-based setting can be greatly improved. Likewise, understanding the circumstances in which MVs are beneficial (or not) can be seen to lag behind research on incremental maintenance techniques.

This paper describes in detail a complete, implemented and evaluated solution to the IVM problem in object databases. The solution holds for a challenging, useful, and modern setting, viz., that of ODMG-compliant object databases.

The main contributions of the paper can be summarized as follows:

- (1) a solution to the problem of incrementally maintaining (an expressive subset of) materialized OQL views defined over expressive ODMG-compliant schemas with respect to any update operation in the ODMG language bindings, and
- (2) an experimental evaluation of the implemented system, thereby throwing light on the circumstances in which view materialization is likely to yield performance benefits.

The remainder of the paper is structured as follows. Sections 2 and 3, respectively, precisely define the problem and its solution space. Section 4 briefly presents the technical background underlying the contributions of the paper. Then, Section 5 describes in detail the MOVIE system, and Section 6 the evaluation of its performance. Section 7 discusses and contrasts work that is related to the contributions of the paper. Finally, Section 8 draws some conclusions and points to work still to be carried out in light of the contributions made here.

## 2. Problem definition

This section defines in some detail the problem addressed by the paper. The detail is needed not only because in the relevant literature not much attention has been dedicated to this task, but also to characterize more precisely in what sense MOVIE can be said to be a more comprehensive solution than typically found in the literature. The processes of view materialization are first characterized. Then, the materialized view maintenance (MVM) problem is defined in detail. Section 3 characterizes the corresponding solution space.

Hereafter, **bold** is used for syntactic entities (i.e., expressions in a formal language), and *bold italic* for their denotations (i.e., their corresponding semantics entities). If the expression is from an external, user-level language, they are written in typewriter font. If from a language internal to the system, then sans-serif font is used. For example, if the extent of a user-defined class is identified by `CompositeParts`, then *CompositeParts* refers to the set of instances comprising that extent. Also, a query  $q = \text{select } c \text{ from } c \text{ in } \text{CompositeParts}$  might be mapped internally to a comprehension such as  $\text{comp}(q) = \{c \mid c \leftarrow \text{CompositeParts}\}$ , and  $q$  refers to the outcome of evaluating  $q$ .

### 2.1. Materialization

Let  $\text{DL}_{\mathcal{M}}$ ,  $\text{QL}_{\mathcal{M}}$ , and  $\text{ML}_{\mathcal{M}}$  denote, respectively, the definition, query and manipulation languages associated with a data model  $\mathcal{M}$ . In this paper, unless otherwise stated,  $\mathcal{M} = \text{ODMG}$ , and the subscript is, therefore, often dropped. A valid schema  $s \in \text{DL}$  denotes a set  $\mathcal{S}$  of conforming database states. Given  $s, s' \in \mathcal{S}$ , a valid update  $u \in \text{ML}$  over a database state  $s$  denotes an update event  $u$  that effects a transition from  $s$  to  $s'$ . A valid query  $q \in \text{QL}$  over a database state  $s$  denotes a well-typed (under  $s$ ) bag  $q$  of records. One says that, given  $s$ ,  $q$  evaluates to  $q$ . Note that the denotation of a query is parameterized to (i.e., may vary depending on) a particular database state.

The process of defining a view  $v = (n, q)$  within a schema  $s \in \text{DL}$  is denoted by  $\text{define}(n, q)$ . The process causes  $n$  to become, in  $s$ , a name for the denotation of the query  $q \in \text{QL}$ . Once  $\text{define}(n, q)$  succeeds,  $n$  can be used thereafter as a shorthand for the query  $q$  that  $n$  has been bound to. In other words, in  $\text{DL}$ ,  $\text{QL}$  and  $\text{ML}$  expressions, the denotation of  $n$  is the denotation of its associated query  $q$ . One says that, given  $s$  and a binding of  $n$  to  $q$ ,  $n$  evaluates to  $q$ . In the remainder of the paper, a view  $v = (n, q)$  is often referred to by  $v$  or  $n$  or  $q$ , depending on the context and provided that no confusion arises.

The process of materializing a view  $v = (n, q)$  over the database state  $s$  is denoted by  $\text{materialize}(n, s)$ . Intuitively, if a view  $n$  is materialized, then its denotation  $n$  is bound to a database state (typically, that which was current when the materialization process took place). The process causes a transition from  $s$  to a new state  $s'$  that, with respect to  $s$ , is enriched with a new type  $n$  whose attributes are those defined by  $q$  and whose extent, denoted by  $n_s$ , is the result of evaluating  $q$  over  $s$ .

When a view  $v = (n, q)$  is materialized,  $n$  is no longer necessarily equivalent to  $q$ . While a non-materialized view is consistent irrespective the database state (because its denotation is dynamically computed on an as-needed basis), an MV whose extent is  $n_s$ , may be rendered inconsistent by an update event that causes a transition in the database to a state  $s' \neq s$ .

For so long as  $\mathbf{n}_{s_i}$  is consistent in  $s_j$ ,  $j > i$ , view materialization is economical in principle, because the evaluation of  $\mathbb{QL}$  and  $\mathbb{ML}$  expressions that refer to the view need only (at most) scan the MV extent  $\mathbf{n}_{s_i}$  as opposed to having to evaluate the query  $\mathbf{q}$  over  $s_j$ . If an update event causes a transition in the database to a state  $s'$  such that  $\mathbf{n}_s$  is rendered inconsistent in  $s'$ , then, before any  $\mathbb{QL}$  or  $\mathbb{ML}$  expression that mentions  $\mathbf{n}$  can be evaluated over  $s'$ , the MV must be rendered consistent again in  $s'$ . The latter can be achieved either by re-materializing the entire extent of the MV or by effecting such updates to  $\mathbf{n}_s$  as necessary to make it indistinguishable from what it would be if the MV was re-materialized. The second alternative is known as *incremental maintenance*.

Whether view materialization is economical in practice depends on characteristics of the applications defined over a particular database. These include whether queries are more frequent than updates, whether the updates that do occur are likely to cause many MVs to be rendered inconsistent, and whether it is costly to maintain an MV (either through re-materialization or through incremental maintenance). Section 6 explores these issues experimentally.

## 2.2. Materialized view maintenance

Given a query  $\mathbf{q} \in \mathbb{QL}$ , assume that  $\text{define}(\mathbf{n}, \mathbf{q})$  has succeeded and so has  $\text{materialize}(\mathbf{n}, s_j)$ . Then:

- (1) [*Update event monitoring*] if an update  $\mathbf{u} \in \mathbb{ML}$  is applied to some  $s_j \in \mathbb{DL}$ ,  $j \geq i$ , thereby causing a transition in the database state to  $s_{j+1}$ , and
- (2) [*Update event relevance checking*] if  $\mathbf{n}_{s_i} \neq \mathbf{n}_{s_{j+1}}$  (i.e.,  $\mathbf{u}$  has rendered  $\mathbf{n}$  inconsistent in the current database state  $s_{j+1}$ ), then one must choose:
  - (a) [*Update event consumption policy*] when to take action to render  $\mathbf{n}$  consistent in  $s_{j+1}$ , and
  - (b) [*MV maintenance policy*] which action to take as a means of achieving that goal, e.g., whether to equate  $\text{maintain}(\mathbf{n}, \mathbf{u}, s_{j+1})$  with
    - (i) [*Re-materialization*] the function  $\text{materialize}(\mathbf{n}, s_{j+1})$ , which re-materializes  $\mathbf{n}$  at  $s_{j+1}$ , thereby restoring consistency, or with
    - (ii) [*Incremental maintenance*] a function  $\text{increment}(\mathbf{n}, \mathbf{u})$ , which maps  $\mathbf{n}_{s_i}$  into  $\mathbf{n}_{s_{j+1}}$ , with similar effect to re-materialization.

The MVM problem is characterized by (1)–(2a) plus (2b(i)) or (2b(ii)). The incremental view maintenance (IVM) problem is, more specifically, characterized by the choice of (2b(ii)) as the maintenance policy.

As far as maintenance policies are concerned, re-materialization requires no extra software infrastructure over that already available for view materialization, but is typically costly. The costs involved are often significant because the goal is to restore consistency, and it is likely that most elements in the re-materialized view extent are not changed by the update event.

Incremental maintenance requires extra software infrastructure but is typically less costly than materialization because it may be so designed as to consider only those elements in the MV extent that were affected by the database state transition being considered.

### 3. Solution space

Given the problem definition in Section 2, this section describes dimensions of the solution space that are particularly pertinent to characterizing the contributions of this paper. The level of detail makes it possible for Section 7 to situate solutions and gauge their differentiating characteristics more clearly. The dimensions cluster around linguistic and environmental issues (or, alternatively, around static, compile-time issues and dynamic, run-time ones).

The first dimension is *the definition language dimension*. Solutions differ depending on the types (and hence, schemas and database states) over which views can be defined. Intuitively, the richer the type system the more varied the semantic issues arising (e.g., handling of duplicates, or of orderings) in a context in which restoring consistency is the main objective. A non-exhaustive list of values in this dimension, partially (and informally) ordered from least to most expressive, could be as follows: [*first-normal-form (1NF) relational, nested relational, object-based*]. Historically, most of the work on the IVM problem has been carried out in a 1NF relational setting (the deductive setting being structurally equivalent to it). The object-based case is significantly more complex and has been considered much less often in the IVM literature.

The second dimension is *the manipulation language dimension*. Solutions differ depending on the update events that can be responded to (and hence, on the data manipulation statements whose effects the solution can accommodate). Intuitively, the richer the data manipulation language the more varied the semantic issues arising (e.g., handling full-fledged updates, as opposed to append-only state transitions). Most solutions assume an ability to respond at the level of an atomic update event. The type system that the solution applies to determines the primitive operations from which one defines the observable behaviour of the type instances. Thus, the possible atomic update events that a solution might choose to address are determined by such primitive operations. Values in this dimension could be subsets of the following non-exhaustive set: {*insert, delete, modify, implicit consequences*}. By implicit consequences is meant, e.g., the enforcement of referential integrity (as is the case, in the ODMG setting, for updates on relationship attributes reflecting on their inverses, and vice-versa). This list might be extended by consideration, of operations on collections of instances, as needed in the ODMG setting and as supported by MOVIE. The object-based case is significantly more complex and has been considered much less often in the MVM literature.

The third dimension is *the view language dimension*. Solutions differ depending on the queries that can be bound to view names. Intuitively, the richer the view definition language the more varied the semantic issues arising (e.g., handling of negation, or of recursion). The overwhelming majority of work on the IVM problem has concentrated on studying how different levels of expressiveness in the view definitions lead to compromise in other dimensions of the solution space. Solutions have been concerned mainly with the most challenging points in the solution space that more expressive view definitions give rise to. To cite but a few: incrementally maintaining views involving aggregation, negation (in a deductive setting, which corresponds, in a relational setting, to set difference), or recursion has proved quite challenging. Values in this dimension could be sub-lists of the following non-exhaustive list of values, partially (and informally) ordered from least to most expressive, could be as follows: [*select-project-join (SPJ), unnest ( $\mu$ ), nest ( $\Gamma$ ),*

*sub-queries* (SQ), *aggregation* ( $\mathfrak{A}$ ), *duplicates* ( $\mathcal{D}$ ), *union* ( $\cup$ ), *intersection* ( $\cap$ ) *difference* ( $-$ )]. The object-based case in the ODMG setting is arguably not the most complex as far as the expressiveness of the view definition language is concerned: the deductive case is, and has been considered more often in the IVM literature.

The fourth dimension is *the event processing dimension*. In addition to differences related to the database languages that underlie them, solutions will also differ regarding the strategy they adopt for processing update events. One possibility is to react to the update event immediately. In this case, restoring consistency is done at the finest possible granularity with respect to any event that may have rendered the MV inconsistent. An alternative is to defer the reaction to the update event, typically until the end of the transaction in which the event took place. Thus, a non-exhaustive list of values in this dimension, often taken as alternatives to one another, could be as follows: [*immediate, deferred*]. Intuitively, an immediate reaction simplifies matters greatly. A deferred reaction might have to grapple with complex chains of consequences and with compound (possibly mutually compensating) events. This complexity may, nevertheless, be a price worth paying because the accumulating overhead of reacting to individual update events may be higher than that of reacting at a coarser granularity, e.g., at transaction commit only.

The fifth, and final, dimension is *the environmental information dimension*. The final dimension used to characterize the solution space for the IVM problem concerns how much data must be available in the environment for the solution to work. Intuitively, the less data that is needed the better. Values in this dimension could be sub-lists of the following non-exhaustive list of values partially (and informally) ordered from least to most expressive: [*materialized view extent, update delta, view definitions, base relations, auxiliary views*]. By *update delta* is meant the pre- and post-update state of tuples or objects involved in the update. Views that can be maintained without the need to access the base relations from which the materialized data originally came are called *self-maintainable*. Self-maintainability is, generally, a desirable property, but in some cases, access to base relations cannot be avoided by any solution to the IVM problem. There is a trade-off between the amount of data required and the update events that can be accommodated by each solution. Relying on more data allows more update events to be handled.

#### 4. Technical background

As previously indicated, MOVIE is a solution to the IVM problem in an object-based setting, with the data model  $\mathcal{M} = \text{ODMG}$ . This section introduces (via examples) the subsets of the object definition language (ODL) and of the object query language (OQL) that constitute the bindings for, respectively, the  $\mathbb{DL}$  and  $\mathbb{QL}$  supported in MOVIE. The  $\mathbb{ML}$  supported coincides exactly with the set of operations in the ODMG language bindings that are applicable to types definable in the  $\mathbb{DL}$  supported in MOVIE. However, note that the formulation of the solution is in terms of update event types, where an event type abstracts away from details specific to any method invocation and captures only those aspects of the update that bear on the IVM problem. Finally, the section briefly describes the query processing notions used in the solution, specifically those based on the monoid comprehension framework described in [11] and implemented in OPTGEN [9] and  $\lambda$ -DB [12].

#### 4.1. ODMG ODL and OQL

Some familiarity with the ODMG standard for object databases [7] is assumed in the paper. The ODMG ODL is used to declare a schema which defines the valid application types. The example database schema used in this paper is based on the OO7 Benchmark [5]. The ODL representation of the OO7 database schema is given in Fig. 1, which is an example of the fragment of ODL supported by MOVIE.

```

module OO7DB {
  class designObj {
    attribute long id;
    attribute string type;
    attribute long buildDate; };
  enum AssemblyType {Complex, Base};
  class atomicPart extends DesignObj (extent AtomicParts key id) {
    attribute long x;
    attribute long y;
    attribute long docId;
    relationship bag<connection> to inverse connection::from;
    relationship bag<connection> from inverse connection::to;
    attribute compositePart partOf; };
  class connection (extent Connections) {
    attribute string type;
    attribute long length;
    relationship atomicPart from inverse atomicPart::to;
    relationship atomicPart to inverse atomicPart::from; };
  class compositePart extends designObj (extent CompositeParts key id) {
    attribute bag<atomicPart> parts;
    attribute atomicPart rootPart;
    relationship document documentation inverse Document::part;
    relationship bag<baseAssembly> usedInPriv inverse baseAssembly::componentsPriv;
    relationship bag<baseAssembly> usedInShar inverse baseAssembly::componentsShar; };
  class document (extent Documents keys title, id) {
    attribute string title;
    attribute long id;
    attribute string Text;
    relationship compositePart part inverse compositePart::documentation; };
  class manual (extent Manuals) {
    attribute string title;
    attribute long id;
    attribute string Text;
    relationship moduleT Mod inverse moduleT::man; };
  class assembly extends designObj (extent Assemblies) {
    attribute AssemblyType asmType;
    relationship complexAssembly superAssembly inverse complexAssembly::subAssemblies;
    relationship moduleT oModule inverse moduleT::assemblies; };
  class complexAssembly extends assembly (extent ComplexAssemblies) {
    relationship bag<assembly> subAssemblies inverse assembly::superAssembly; };
  class baseAssembly extends assembly (extent BaseAssemblies key id) {
    relationship bag<compositePart> componentsPriv inverse compositePart::usedInPriv;
    relationship bag<compositePart> componentsShar inverse compositePart::usedInShar; };
  class moduleT extends designObj (extent Modules key id) {
    attribute complexAssembly designRoot;
    relationship manual man inverse manual::Mod;
    relationship bag<assembly> assemblies inverse assembly::oModule; };
};

```

Fig. 1. OO7 database schema in ODL.

```

define dbSizeView() as
select struct(ComPartId:c.id,
             CType:c.type,
             DocId:d.id)
from   c in CompositeParts,
       d in Documents
where  c.documentation = d

define comAtomBuildDates() as
select struct(ComPartId:c.id,
             ComBDate:c.buildDate,
             AtomBDates:
             (select a.buildDate
              from a in c.parts))
from   c in CompositeParts

```

Fig. 2. Example OQL views.

The ODMG OQL is designed to resemble SQL92, with the extensions one might expect given the constructs defined by the structural and behavioural properties of its underlying type system, viz., complex objects, object identity, path expressions, among others. OQL is not computationally complete (e.g., it cannot express recursion nor does it have explicit update operations). 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. Fig. 2 depicts example OQL views (over the schema in Fig. 1) that MOVIE can maintain. To the left of Fig. 2 is an example of a view, involving a join, which builds structures containing the `id` and `type` of `compositePart` and `id` of document objects. The view to the right of Fig. 2 contains a subquery and involves nesting and unnesting of collection values.

#### 4.2. Monoid-based ODMG query processing

The solution described in this paper builds on the monoid-based query processing framework developed in [11]. In particular, the algorithms that embody the IVM solution implemented in MOVIE operate on expressions in the Fegaras–Maier monoid algebra. The authors' practical experience of implementing systems using the monoid approach has been reported for parallel databases in [28] and for a deductive extension to OQL in [27].

A comprehensive description of a mapping of OQL 1.2 into monoid comprehensions is given in [15]. These can be normalized as described in [11]. Normalized comprehensions are then mapped to the monoid algebra described here using the examples in Fig. 3. For details on the monoid algebra see [11].

For the view to the left of Fig. 2, the corresponding algebraic expression is given to the left of Fig. 3. The algebraic operators occurring in it are `reduce`, `join`, and `get`. The monoid used by all of them is `bag`. The indentation in Fig. 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 its monoid, a `get` operator specifies the extent it scans (e.g., `CompositeParts`), a variable ranging over the objects in that extent (e.g., `c`), and a retrieval condition (e.g., `and()`, in this case a vacuously true one). As well as its monoid, the `join` operator specifies its two input arguments (e.g., the values of `get` expressions), the join condition (e.g., `and(eq(c.documentation, 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 result returned (e.g., `struct(ComPartId=c.id, CType=c.type, DocId=d.id)`) and an internal variable (e.g., `X1`) that can be used to refer to its instances. Such a structure consists of `bind`

<pre> reduce(bag,   join(bag,     get(bag, CompositeParts, c, and()),     get(bag, Documents, d, and()),     and(eq(c.documentation, d)),     none),   X1,   struct(ComPartId=c.id, CType=c.type,     DocId=d.id),   and()) </pre>	<pre> reduce(bag,   nest(bag,     unnest(bag,       get(bag, CompositeParts, c, and()),       a,       project(c, parts),       and(), true),     X2,     project(a, buildDate),     vars(c), and()),   X1,   struct(ComPartId=c.id, ComBDate=c.buildDate,     AtomBDates=X2),   and()) </pre>
--	--

Fig. 3. Algebraic forms of views in Fig. 2.

expressions (e.g., `ComPartId=c.id`) in which a user-defined attribute name (e.g., `ComPartId`) is bound to the value requested via a user-defined expression (e.g., `c.id`).

The algebraic expression to the left in Fig. 3 is not very different from a relational one. In contrast, the one to the right (corresponding to the OQL view to the right in Fig. 2) makes use of `nest` and `unnest` operators. When applied to an object with a collection-valued attribute, the `unnest` operator pairs up the object with each element in the collection that is the value of the relevant attribute and projects out all such formed pairs. The expression `a in c.parts` in the sub-query 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 view to the right in Fig. 2, the `nest` operator is used in Fig. 3 to construct a collection, denoted by `X2`, of type `bag(long)`, which is then assigned to the user-defined variable `AtomBDates`.

The  $\lambda$ -DB system [12] implements the above monoid-based approach [11] to query processing over the ODMG object model (not including arrays and dictionaries).  $\lambda$ -DB adopts a query processing approach closely resembling that used in mainstream DBMSs but acts essentially as a translator into source code that exploits the functionality provided by the SHORE object DBMS [6]. An execution plan is then translated into SHORE C++ and compiled. The resulting executable, when run, evaluates the original OQL query. The solution described in detail in Section 5 manipulates algebraic expressions (such as those in Fig. 3) that  $\lambda$ -DB generates for OQL queries.

## 5. The MOVIE solution

This section presents the first major contribution of this paper, viz., a solution to the problem of incrementally maintaining (an expressive subset of) materialized OQL views defined over expressive ODMG-compliant schemas with respect to any update operation in the ODMG language bindings. It starts with an overview of the solution that adds detail regarding the point in the solution space of the IVM problem occupied by MOVIE. Then, the main algorithms comprising the MOVIE approach to IVM are presented. The section closes with more detail on how MOVIE is implemented over  $\lambda$ -DB.

5.1. An overview

The positioning of the MOVIE solution, given in Section 5.1.1, is based on the dimensions discussed in Section 3. The processes that take place when an MV is defined are shown in the data flow diagram in Fig. 4, where shaded boxes denote components that are assumed to exist, clear boxes denote new components, or extensions to existing ones, required by MOVIE, and the rounded-corner box collects processes that take place at update propagation time as discussed in Section 5.2.3.

With respect to the extension box labelled materialize, all that is required is to extend the evaluator with a sub-component to make persistent the instances returned as the result of executing a query. It is assumed 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 Fig. 4 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 are described in Sections 5.1.2 and 5.1.3. Section 5.2.1 presents the algorithms that define the functionality of the box labelled specify events to monitor. The functionality of the box labelled generate maintenance plan is described in Section 5.2.2.

5.1.1. Positioning the MOVIE solution

The solution to the IVM problem presented here assumes, as do all other solutions, the availability of the update event, the *update 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, the availability of both the references to the base objects that contribute data to the MV and of the base extents required for materializing it is also assumed (although there is no need to

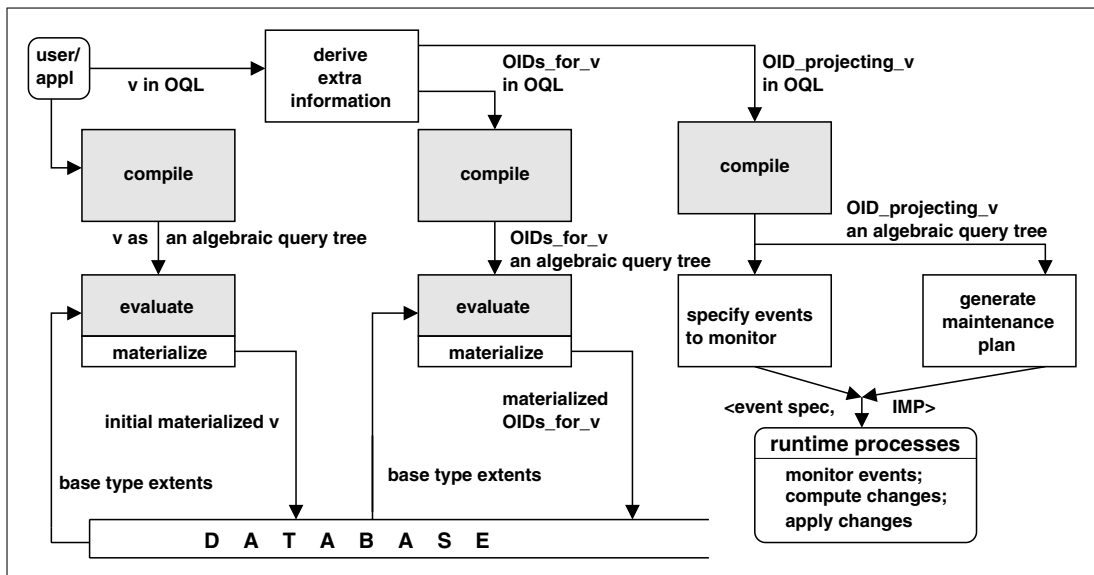


Fig. 4. Processes at view compilation time.

<b>Definition Language</b>	<i>object-based</i>
<b>Manipulation Language</b>	<i>insert, delete, and modify (on both extents and collection-valued properties), implicit consequences</i>
<b>View Language</b>	<i>RGJ, <math>\mu</math>, <math>\Gamma</math>, SQ</i>
<b>Event Processing</b>	<i>immediate</i>
<b>Environmental Information</b>	<i>materialized view extent, update delta, view definitions, base relations, auxiliary views</i>

Fig. 5. MOVIE dimensions in the IVM solution space.

recourse to base objects and extents for certain kinds of update). The MOVIE solution is valid for MVs that refer to any ODL-definable type (excluding array and dictionary collections) and that are definable using the reduce, get, join, nest and unnest bulk operators of the monoid algebra (excluding self-joins). Note that the monoid algebraic operators reduce, get, join roughly correspond, respectively, to the relational algebraic operators project, select, join. As a result, an SPJ relational query is later referred to as a reduce-get-join (RGJ) query, for example. The MOVIE solution 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, the MOVIE solution yields incremental maintenance plans (IMPs) at the algebraic level and is, to the best of the authors' knowledge, for object-based solutions, the first one to do so. This makes it easier to integrate the MOVIE solution into the kind of query processing frameworks that mainstream DBMSs rely on.

The point occupied by MOVIE in the solution space described in Section 3 is characterized in Fig. 5. Note that in Fig. 5, *insert* denotes `new()` on objects and `insert_element()` on collections. Similarly, *delete* denotes `delete()` on objects and `delete_element()` on collections.

### 5.1.2. Materialization in MOVIE

When an MV is defined, its definition is traversed to characterize the kinds of update events that might render the MV inconsistent with the database. These include derived events that arise in the ODMG model as a result of the requirement to enforce referential integrity constraints declared with the `relationship/inverse` construction. Depending on the event type, one or more algebraic IMPs are constructed that, when evaluated, compute the required changes to the MV. A crucial component in the MOVIE solution is, therefore, the generation of IMPs that are appropriate for each kind of update event. In MOVIE, 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. Once the changes needed have been obtained, MOVIE applies them to the relevant MV extent.

The comprehensive nature of the MOVIE 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 is referred to as **OIDS\_for\_v**, which is itself compiled, evaluated and materialized. Thus, the OIDs of

<pre> define OIDs_for_dbSizeView()   select distinct <i>struct(CO:c, DO:d, VO:v)</i>   from e in CompositeParts,        d in Documents,        v in <i>DBSizeView</i>   where c.documentation = d         and v.ComPartId = c.id         and v.CType = c.type         and v.DocId = d.id </pre>	<pre> define OID_projecting_dbSizeView()   select <i>distinct</i> <i>struct(CO:c, DO:d,</i>                     <i>ComPartId:c.id,</i>                     <i>CType:c.type,</i>                     <i>DocId:d.id)</i>   from c in CompositeParts,        d in Documents   where c.documentation = d </pre>
---	---

Fig. 6. Derived views for Fig. 2.

objects that contribute data for an instance of  $v$  are associated with the OID of that instance in **OIDs\_for\_v**.

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  $v$ . Consider the `dbSizeView()` view to the left of Fig. 2. The left part of Fig. 6 shows `OIDs_for_dbSizeView`, with generated strings in italics. The right part of Fig. 6 is explained shortly.

### 5.1.3. Incremental maintenance in MOVIE

In MOVIE, the generation of an IMP starts 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 is referred 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 right part of Fig. 6 shows `OID_projecting_dbSizeView`, again with generated strings in italics. In contrast with  $v$  and **OIDs\_for\_v** (i.e., these views are materialized and maintained incrementally), **OID\_projecting\_v** is never materialized: only the definition (compiled into algebraic form) is needed as an input to the generation of the IMPs.

After extra information is derived, 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.

**5.1.3.1. 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 on *dbSizeView* (defined in Fig. 2) of inserting a new `compositePart` object into *CompositeParts*. Let the delta corresponding to the insertion of a new object into the extent of the class `compositePart` be denoted by  $\Delta(\text{onInsertToExtent}, \text{CompositeParts})$ . The delta, in this case, can be thought of as simply containing the new `compositePart`'s OID. Note that the delta contains a representation of the state transition. Below, let  $\Delta_{\text{old}}(\text{onInsertToExtent}, \text{CompositeParts})$

<pre> reduce(set,   join(set,     get(set,       Δ(onInsertToExtent, <i>CompositeParts</i>),       c,       and()),     get(set, Documents, d, and()),     and(eq(c.documentation, d)),     none),   X1,   struct(CO=c, DO=d, ComPartId=c.id,     CType=c.type, DocId=d.id),   and()) </pre>	<pre> reduce(set,   join(set,     get(set, OIDs_for_dbSizeView, mat_oids, and()),     get(set,       Δ(onModifyAttribute, <i>CompositePart.type</i>),       δ,       and()),     and(eq(mat_oids.CO, δ)),     none),   X1,   struct(mat_obj=mat_oids.VO, CType=δ.type),   and()) </pre>
--	---

Fig. 7. IMPs for Fig. 2: onInsertToExtent and onModifyAttribute.

denote the state before the transition took place, and  $\Delta_{\text{new}}(\text{onInsertToExtent}, \mathbf{CompositeParts})$  the state after (the arguments may come to be omitted if they are easily inferable from the context or if no specific event is being referred to). For insertions only the new state matters for incremental MV maintenance. Correspondingly, for deletions only the old state matters. However, for modification, both the old and new state are necessary, so the representation of the delta benefits from being conceptualized as a pair. The IMP generated by MOVIE (shown in algebraic form to the left of Fig. 7) is different from the evaluation plan derived for `OID_projecting_dbSizeView` only in ranging over  $\Delta(\text{onInsertToExtent}, \mathbf{CompositeParts})$  rather than over  $\mathbf{CompositeParts}$ . When this IMP is evaluated, the result is a set of instances of `OID_projecting_dbSizeView` which is then input to the algorithm that applies the changes to the MV (and to `OIDs_for_dbSizeView`, if required).

**5.1.3.2. 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 type of a `compositePart` object. In this case (shown in algebraic form to the right of Fig. 7), the information captured in the delta is not enough to identify which object(s) in the MV might be affected because the delta only refers to the updated `compositePart` object: there is no handle in the delta to the instances in the MV which have data that was contributed by the affected `compositePart` object. In this case, the IMP will need to join `OIDs_for_dbSizeView` 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_dbSizeView`, if required).

## 5.2. The main MOVIE algorithms

This section presents the core algorithms used in MOVIE to solve the IVM problem at the point in the solution space characterized in Section 5.1.1.

In Section 2.2, given bindings for the linguistic dimensions in the solution space, the MVM problem was characterized as comprising: (1) a choice of policy for update event consumption, (2)

a choice of policy for MV maintenance, (3) an implementation of update event monitoring, (4) an implementation of update event relevance checking, and (5) an implementation of a function increment to map the extent of the MV as it was before the update event into the MV extent that would be obtained were the view to be re-materialized in the database state after the update event.

MOVIE adopts an *immediate* update event consumption policy ((1) above) and an *incremental maintenance* policy ((2) above). The remainder of this section starts by describing an algorithm that associates to each MV the set of update events that must be monitored for the MV to be incrementally maintained. It then describes an algorithm to choose an IMP that computes the changes needed to the MV for each update event type. Finally, it describes how the computed changes are applied to the MV. MOVIE addresses the problem of monitoring and checking the relevance of update events ((3) and (4) above) by generating, from the definitions associated with the MV, the set of update event types that must be reacted to. MOVIE addresses the problem of restoring the consistency of an MV with respect to an update event that compromised that consistency ((5) above) in two stages: firstly, between the two kinds of IMP described in Section 5.1.3, MOVIE chooses the appropriate one for the update event it is reacting to; secondly, it applies the changes obtained by evaluating the IMP, thereby restoring the consistency of the MV.

Some assumptions underpinning the remainder of this section are now made explicit. All the algorithms operate on algebraic expressions represented as trees such as depicted in Fig. 3. These are an internal form of OQL expressions such as those in Fig. 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 Fig. 3 as a source of examples, some of the categories of interest are operator names (e.g., join), extent names (e.g., Documents), object names (e.g., d), attribute names (e.g., compositePart.buildDate, atomicPart.buildDate), and internal names (e.g., X1). Another category of interest is collection names (e.g., parts in Fig. 3). Notice that when a name is not globally unique, it is assumed that context information is concatenated to it so as to make it unique, and that the outcome of this concatenation is construed as a token. Functional notation is used to denote metadata that might be relevant, e.g., given an attribute name *a*, the type of which *a* is an attribute is denoted by `typeWhereDefined(a)`, and one 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 the inverse relationship is denoted as `inverseOf(a)`. In addition, given a collection attribute *c*, `typeOfElementIn(c)` denotes the type of the elements in the collection.

### 5.2.1. Specifying events to monitor

This subsection describes the algorithm used in MOVIE to identify, given an MV that is to be incrementally maintained, which update events need to be monitored by the system.

MOVIE reacts to the following update event types: `onInsertToExtent`, `onDeleteFromExtent`, `onModifyAttribute`, `onInsertElementToCollection`, `onDeleteElementFromCollection`, and `onModifyAttributeInCollectionElement`. Sometimes (hopefully self-explanatory) abbreviations may be used. These update event types generalize, and cover, all update operations in the ODMG standard, including those on collection types other than dictionary and arrays. From these update event types, MOVIE generates specifications of events that need to be monitored for each MV. The event specifications are generated from the (algebraic form of the) MV definition. Fig. 8 presents

```

function specifyEventsToMonitor(MVD: algebraic.expression) → 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) };
    foreach path p in MVD do
      E := E ∪ specifyEventsInPathExpression(p, MVD);
return E;
where
function specifyEventsInPathExpression(p: path, Q: algebraic.expression)
  → set[eventSpecification] ≡
  E: set[eventSpecification] := ∅;
  Attrs: set[string] := ∅;
  arg: string;
  case of p such that
  when (p is empty) /* base case */
    break;
  when (head(p) is a var) /* done, by iterating over scans */
    E := E ∪ specifyEventsInPathExpression(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) /* head(p) names a single-valued relationship */
          then
            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 };
          else
            E := E ∪ { Mod head(p) };
        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) };
            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 };
              else /* head(p) does not name a relationship and has been handled */
                continue
            E := E ∪ specifyEventsInPathExpression(rest(p), Q);
          esac;
return E;
□

```

Fig. 8. Specifying events to monitor.

the algorithms that, given an MV definition, return the event specifications associated with the MV (and hence, the update events that need to be monitored in order to ensure that it can be kept

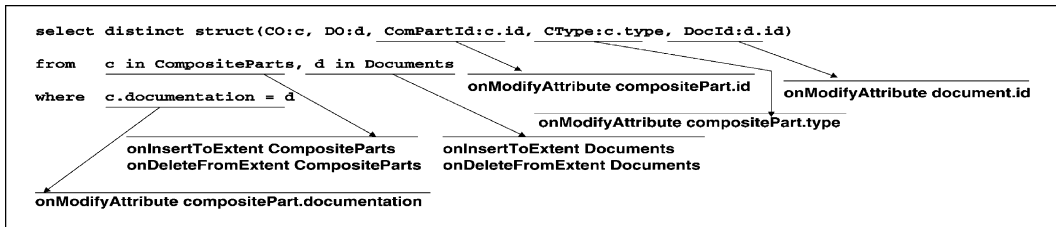


Fig. 9. Event specifications for Fig. 6.

consistent by incremental maintenance). In Fig. 8, by a ‘scan in a view’ is meant an expression of the form ‘*var in VScope*’ or ‘*VScope as var*’. In Fig. 8, by a ‘path 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. In Fig. 8, relationship names and the names of their corresponding inverse relationships are also referred to as attribute names in Fig. 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*.

To illustrate the execution of the algorithms in Fig. 8, consider Fig. 9. It shows (as arrow targets) the event specifications returned by `specifyEventsToMonitor` for the view to the right of Fig. 6. The arrow sources are the syntactic features in the view definition that cause the generation of the event specifications in the target. Now, 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 `specifyEventsInPathExpression` 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.

### 5.2.2. Generating an incremental maintenance plan

This subsection describes the algorithm used in MOVIE to choose an IMP whose evaluation returns the set of changes needed to restore the consistency of an MV with respect to each update event that may affect that consistency.

Recall that there are two types of IMP for an MV *v* in the approach described in this paper: one that plants a delta in `OID_projecting_v` and the other that joins the delta with `OIDs_for_v`. MOVIE chooses an IMP for an MV depending on the algebraic classification of the latter and the type of the update event. The algebraic classification of an MV depends on the algebraic operators that occur in it. Although a view can belong to more than one class (e.g., a ReduceGetJoin view is also a ReduceGet but not vice versa), however, we choose the more complex among the applicable classes. Thus, if `nest` occurs in the algebraic form of an MV then the view class is `Nest`; if `unnest` occurs but `nest` does not, then the class is `Unnest`; and if `join` occurs but not `nest` or `unnest`, then the class is `ReduceGetJoin`, otherwise the class is `ReduceGet`. For example, `dbSizeView` in Fig. 2 is a `ReduceGetJoin` view because its algebraic form, shown to the left in Fig. 3, contains the reduce, get and join operators.

```

define V1() as
  select struct(CID:c.id, CP:c.parts)
  from c in CompositeParts
  (a)

define V2() as
  select struct(CID:c.id, CP:a.id)
  from c in CompositeParts, a in c.parts
  (b)

-----

define V3() as
  select struct(CID:c.id,
              CP:(select a.id
                  from a in c.parts))
  from c in CompositeParts
  (c)

```

Fig. 10. Example views illustrating '(PNU)-C'.

Given the MV type determined as in the preceding paragraph, Fig. 11 determines precisely, for each event type and for each MV type, what kind of IMP is chosen by MOVIE to compute the changes to be applied. In Fig. 11, note that 'CA' abbreviates the phrase 'condition attribute', by which is meant an attribute occurring in a condition expression (e.g., in `dbSizeView` in Fig. 2, `c.documentation` is a CA whereas `c.type` is not). Note also that '(PNU)-C' denotes whether the event affects a *projected-collection* (PC) or *nested-collection* (NC) or *unnested-collection* (UC) in a view definition. Fig. 10 shows example views used for illustrating '(PNU)-C'.

In Fig. 10a, the collection `c.parts` is projected and assigned to the variable `CP`. The update events of interest to the view, among others, are `onInsertToExtent` and `onDeleteFromExtent` on ***compositePart.parts*** that affect the `CP` attribute of objects in the view. The event `onModifyAttributeInCollectionElement` on ***compositePart.parts(atomicPart.id)*** is not relevant, as elements in `CP` are OIDs of `atomicPart` objects and these are not affected by modifications to attributes. In Fig. 10b, the collection `c.parts` is unnested as the `id` attribute from each `atomicPart` object in the collection is projected and assigned to the variable `CP`. The update events of interest to the view, among others, are `onModifyAttribute` on ***atomicPart.id***, `onInsertToExtent` and `onDeleteFromExtent` on ***compositePart.parts*** that affect the `CP` attribute, and the membership of objects in the view. The event `onModifyAttributeInCollectionElement` on ***compositePart.parts(atomicPart.id)*** is not relevant as it is already handled by `onModifyAttribute` on ***atomicPart.id***. In Fig. 10c, the collection `c.parts` is unnested, the `id` attribute from each `atomicPart` object in the collection is projected and then a nested collection (containing the projected `id` attribute values) is assigned to the variable `CP`. The update events of interest to the view, among others, are `onModifyAttributeInCollectionElement` on ***compositePart.parts(atomicPart.id)***, `onInsertElementToCollection` and `onDeleteElementFromCollection` on ***compositePart.parts*** that affect the `CP` attribute of objects in the view. In this case, it can be seen that the view has nested the collection `c.parts`. Although the event `onModifyAttribute` on ***atomicPart.id*** is relevant, the delta associated with it can neither be planted in `OIDs_projecting_V3` nor joined with `OIDs_for_V3`. However, `onModifyAttribute` on ***atomicPart.id*** is monitored and when raised, consequently, `onModifyAttributeInCollectionElement` on ***compositePart.parts(atomicPart.id)*** is also raised and the view is incrementally maintained accordingly.

Case	Event Type	CA?	(PNU)-C?	View Type		
				ReduceGet $\vee$ ReduceGetJoin	Unnest	Nest
1	onInsertToExtent			plant $\Delta_{new}$	plant $\Delta_{new}$	plant $\Delta_{new}$
2	onInsertElement– ToCollection		PC	$\mathbf{vx} \bowtie \Delta_{new}$	$\mathbf{vx} \bowtie \Delta_{new}$	$\mathbf{vx} \bowtie \Delta_{new}$
3	onInsertElement– ToCollection		NC	NA	NA	plant $\Delta_{new}$
4	onInsertElement– ToCollection		UC	NA	plant $\Delta_{new}$	plant $\Delta_{new}$
5	onDeleteFromExtent			$\mathbf{vx} \bowtie \Delta_{old}$	$\mathbf{vx} \bowtie \Delta_{old}$	$\mathbf{vx} \bowtie \Delta_{old}$
6	onDeleteElement– FromCollection		PC	$\mathbf{vx} \bowtie \Delta_{old}$	$\mathbf{vx} \bowtie \Delta_{old}$	$\mathbf{vx} \bowtie \Delta_{old}$
7	onDeleteElement– FromCollection		NC	NA	NA	$\mathbf{vx} \bowtie \Delta_{old}$
8	onDeleteElement– FromCollection		UC	NA	$\mathbf{vx} \bowtie \Delta_{old}$	$\mathbf{vx} \bowtie \Delta_{old}$
9	onModifyAttribute	no		$\mathbf{vx} \bowtie \Delta_{new}$	$\mathbf{vx} \bowtie \Delta_{new}$	$\mathbf{vx} \bowtie \Delta_{new}$
10	onModifyAttributeIn– CollectionElement	no	NC	NA	NA	$\mathbf{vx} \bowtie \Delta_{new}$
11	onModifyAttribute	yes		$\mathbf{vx} \bowtie \Delta_{old}$ plant $\Delta_{new}$	$\mathbf{vx} \bowtie \Delta_{old}$ plant $\Delta_{new}$	$\mathbf{vx} \bowtie \Delta_{old}$ plant $\Delta_{new}$
12	onModifyAttributeIn– CollectionElement	yes	NC	NA	NA	$\mathbf{vx} \bowtie \Delta_{old}$ plant $\Delta_{new}$

Fig. 11. Choosing an incremental maintenance plan.

In Fig. 11, when ‘(PNU)-C’ is NC or UC and the view type is ReduceGet  $\vee$  ReduceGetJoin, the corresponding cells are filled with NA (not-applicable). Similarly, when ‘(PNU)-C’ is NC and the view type is Unnest, the corresponding cells are filled with NA. In Fig. 11, by  $\Delta_{new}$  is meant the data in the delta reflecting the state of the database after the update. Correspondingly, by  $\Delta_{old}$  is meant the data in the delta reflecting the state of the database before the update. For example, for an onInsertToExtent event,  $\Delta_{new}$  is used because the new state of the database is relevant.<sup>1</sup> In contrast, for an onDeleteFromExtent event,  $\Delta_{old}$  is used, as an object once deleted from the database can no longer be referenced, therefore, the old state is essential.

Given an MV  $\mathbf{v}$ , by ‘plant  $\Delta_{new}$ ’ in Fig. 11 is meant that the IMP is generated by planting  $\Delta_{new}$  in the algebraic form of **OID\_projecting\_v**, and by ‘ $\mathbf{vx} \bowtie \Delta_{new}$ ’ is meant that the generated IMP in algebraic form joins  $\Delta_{new}$  to  $\mathbf{vx}$ , where  $\mathbf{vx}$  denotes **OIDs\_for\_v**. Similar remarks apply to the use of  $\Delta_{old}$ . 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.

<sup>1</sup> In some cases, it is possible to avoid accessing base data on onInsertToExtent as discussed in [29]. This is an improvement that will be explored in future work.

An example of (the algebraic form of) an IMP generated by planting a delta was given to the left of Fig. 7, and of one generated by joining a delta with **OIDs\_for\_v**, to the right of Fig. 7. Since the example event types were, respectively, `onInsertToExtent` and `onModifyAttribute` (where the attribute in question is not a CA), and the example MV is a `ReduceGetJoin` view, the examples correspond, respectively, to ‘plant  $\Delta_{\text{new}}$ ’ and ‘ $\mathbf{vx} \bowtie \Delta_{\text{new}}$ ’ in Fig. 11. The intuition behind the entries in Fig. 11 is as follows.

*Insertions.* Where a new object is created that may participate in the view (i.e., after an `onInsertToExtent`), 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—case (1). In case (2), no access to the base data is required as joining the delta to **OIDs\_for\_v** is sufficient to identify the objects in the view affected by `onInsertElementToCollection`, which only causes insertions into a collection-valued attribute in the view objects. In cases (3) and (4), since attribute values are required to be projected from the element inserted by `onInsertElementToCollection`, the view must be evaluated over the delta to identify whether or not the element should contribute to the view, which requires access to base data. To illustrate case (4), consider the Nest view called `comAtomBuildDates` defined to the right of Fig. 2. To the left of Fig. 12 is the algebraic form of `OID_projecting_comAtomBuildDates` derived from `comAtomBuildDates`. Now consider the effect on `comAtomBuildDates` of an insertion into the `parts` collection of a `compositePart` object. This update is monitored as an `onInsertElementToCollection` event. An IMP resulting from this update is generated by planting the delta as shown to the right of Fig. 12.

*Deletions.* Unlike `onInsertToExtent`, `onDeleteFromExtent` does not require access to the base data, as joining the delta to **OIDs\_for\_v** is sufficient to identify the objects in the view affected by the delete—cases (5)–(8). The reason that access to the base data is not required is due to the fact that deletion of objects or elements can only affect the view if they have already contributed data to the view. To find out if the view is affected by such a deletion, the delta is joined with **OIDs\_for\_v**.

*Modifications.* Modifying a *none-CA* cannot change the membership of objects that contribute to the view, but may affect the data projected in the view. As such, joining the delta to **OIDs\_for\_v**

<pre> reduce(set,   nest(set,     unnest(set,       get(set, CompositeParts, c, and()),       a,       project(c, parts),       and(), true),     X2,     struct(elO=a, el=a.buildDate),     vars(c), and()),   X1,   struct(ComPartId=c.id,     ComBDate=c.buildDate,     AtomBDates=X2),   and()) </pre>	<pre> reduce(set,   nest(set,     unnest(set,       get(set, <math>\Delta</math>(onInsertElement, <i>compositePart.parts</i>),       c, and()),       a,       project(c, parts),       and(), true),     X2,     struct(elO=a, el=a.buildDate),     vars(c), and()),   X1,   struct(ComPartId=c.id, ComBDate=c.buildDate,     AtomBDates=X2),   and()) </pre>
--	--

Fig. 12. Algebraic form of `OID_projecting_comAtomBuildDates` and an IMP for `onInsertElementToCollection`.

is sufficient to identify the objects in the view affected by the update—cases (9) and (10). Modifying a CA may cause an object that is already in the view to be removed, or may lead to new objects being added to the view. The former can be identified by joining  $\Delta_{\text{old}}$  with **OIDs\_for\_v**, but the latter requires access to base data through the planting of  $\Delta_{\text{new}}$ —cases (11) and (12).

It can be observed from Fig. 11 that out of the 29 entries of choosing an IMP, 19 are of the form  $\mathbf{vx} \bowtie \Delta_{\text{new}}$  (or  $\mathbf{vx} \bowtie \Delta_{\text{old}}$ ). This means that, as a whole, MOVIE achieves *self-maintainability*, in about two-thirds of the cases.

The generation of the IMP(s) denoted by a cell in Fig. 11 is effected by a straightforward algorithm operating on algebraic expressions. In the case of planting a delta, it suffices for the algorithm to rewrite the body of **OID\_projecting\_v** replacing any reference to the updated construct by a reference to the delta corresponding to the update event. In the case of joining the delta with the materialized OIDs, it suffices for the algorithm to rewrite the body of **OID\_projecting\_v** replacing any reference to the updated construct by a join of the delta corresponding to the update event with **OIDs\_for\_v**. Comparing the views in Figs. 2 and 6 with the examples in Fig. 7 gives an idea of the transformations effected by the algorithms.

The result set obtained by evaluating the IMP is then passed on to an algorithm that applies the changes to the MV. If a cell in Fig. 11 is populated by two IMPs, then the effect is that both IMPs are generated and evaluated to yield two result sets, and both result sets are passed on to the algorithm, Section 5.2.3, that effects changes to the MV.

### 5.2.3. Applying the computed changes

Given the result set(s) obtained by evaluating the IMP(s) chosen as described in the previous subsection, MOVIE proceeds to change the MV extent.

Fig. 13 determines precisely, for each event type and for each MV type, how changes are propagated to the MV. Most cells in Fig. 13 should be self-explanatory. For example, `onInsertToExtent` propagates as an insert to the MV extent (denoted by  $\mathbf{VX}$ ), and `onDeleteFromExtent` propagates as a delete from the MV extent. `onInsertElementToCollection` and `onDeleteElementFromCollection` only differ from `onInsertToExtent` and `onDeleteFromExtent`, respectively, in that they target a collection-valued attribute of the MV (denoted by  $\mathbf{AV}$ ).

The first group of cells worth discussing is the one consisting of `onInsertElementToCollection` and `onDeleteElementFromCollection` events for views with an unnested collection. For such events and MVs, separate instances in the extent of the MV will have been created for each element in the collection-valued attribute. So, the target is the extent of the MV for both events, instead of the collection-valued attribute.

The second group of cells worth discussing is the one consisting of attribute modifications. If the event affects a CA, then there will have been two IMPs (one based on the old state and one based on the new state) and the changes computed by the former are propagated as deletes while those computed by the latter are propagated as inserts.

The summary style of presentation adopted in this section is due to the fact that the algorithms are tediously long when presented in pseudocode because of a multitude of straightforward cases that need to be reacted to, one by one. This is a direct consequence of the richness of the ODMG type system and of the expressiveness of the ODMG language bindings. The full details for all of the algorithms used in MOVIE is available in [2].

Case	Event Type	CA?	(PNU)- C?	View Type		
				ReduceGet $\vee$ ReduceGetJoin	Unnest	Nest
1	onInsertToExtent			insert(VX)	insert(VX)	insert(VX)
2	onInsertElement– ToCollection		PC	insert(AV)	insert(AV)	insert(AV)
3	onInsertElement– ToCollection		NC	NA	NA	insert(AV)
4	onInsertElement– ToCollection		UC	NA	insert(VX)	insert(VX)
5	onDeleteFrom– Extent			delete(VX)	delete(VX)	delete(VX)
6	onDeleteElement– FromCollection		PC	delete(AV)	delete(AV)	delete(AV)
7	onDeleteElement– FromCollection		NC	NA	NA	delete(AV)
8	onDeleteElement– FromCollection		UC	NA	delete(VX)	delete(VX)
9	onModifyAttribute	no		modify(AV)	modify(AV)	modify(AV)
10	onModifyAttribute– InCollectionElement	no	NC	NA	NA	modify(AV)
11	onModifyAttr	yes		delete <sub>old</sub> (VX) insert <sub>new</sub> (VX)	delete <sub>old</sub> (VX) insert <sub>new</sub> (VX)	delete <sub>old</sub> (VX) insert <sub>new</sub> (VX)
12	onModifyAttribute– InCollectionElement	yes	NC	NA NA	NA NA	delete <sub>old</sub> (AV) insert <sub>new</sub> (AV)

Fig. 13. Applying the changes.

### 5.3. The MOVIE implementation

The MOVIE system is implemented as an extension to  $\lambda$ -DB.  $\lambda$ -DB implements the ODMG model by source translation onto SHORE C++. MOVIE is, in turn, realized as a source translator that adorns the SHORE C++ source corresponding to a  $\lambda$ -DB application with calls to code that implements the MOVIE IVM solution. The executable obtained from the output of this mapping is a  $\lambda$ -DB application in which MVs can be incrementally maintained as described in this paper. Note, therefore, that MOVIE does not monitor update events in the sense that this is normally understood, e.g., in the literature on active databases [26]. However, there are no semantic implications to this, i.e., the overall effect on MVs is exactly the same.

At view-compilation time, MOVIE (using  $\lambda$ -DB functionality) compiles MV definitions, evaluates the queries that determine their extent and materializes the latter. In addition, event specifications are identified. Then, when  $\lambda$ -DB compiles the application code, update requests are identified. These are then adorned with calls to the IVM code that effects the maintenance properly said. 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 this section, and in particular in Fig. 11) and then propagates those changes to the MV (as described in Fig. 13).

## 6. Performance evaluation

This section presents the second major contribution of this paper, viz., an empirical evaluation of the MOVIE system. To the best of the authors' knowledge, this is the first systematic evaluation of the performance of a comprehensive implemented solution to the IVM problem (c.f. the related work described in Section 7). The paper is thus able to throw light on whether the main assumptions underpinning research on IVM are justifiable. The results indicate how the effectiveness of incremental maintenance is affected by issues such as database size, and the complexity and selectivity of views.

### 6.1. Experimental views, queries and updates

The databases used in the experiments reported here are based on the OO7 Benchmark [5]. The ODL representation of the OO7 database schema is given in Fig. 1. Instead, databases of different sizes were generated by varying the number of `modules` (the top level class in OO7). The actual parameters given to the OO7 data generation program (assuming *fan-out* to be 6), plus some information on the resulting database, are shown in Fig. 14 (where the database sizes correspond to 4, 8, 12, 16 and 20 times the size of the small database in the OO7 benchmark). The experiments reported in this paper were carried out on a PC with the following hardware and software: Intel Pentium II, 350 MHz, 512 KB cache, 256 MB RAM, 12 GB IDE Hard Disk (where the system software and 256 MB of swap space reside); RedHat Linux 6.0 Kernel 2.2.17-14, SHORE 1.1.1 and  $\lambda$ -DB 0.5.

*Views.* The views used in the experiments are shown in Fig. 15. Note that for each MV  $v$ ,  $vVir$  denotes the corresponding non-materialized view. For example, to the view `complexView1` (in Fig. 15c) there corresponds a view named `complexView1Vir` that is otherwise identical to `complexView1` but is not materialized. The views in Fig. 15 were conceived for the purpose of investigating the following hypotheses:

(1) *The selectivity of the view influences the performance of IVM.* Different view selectivities (defined as the ratio of the number of objects selected by a query to the number of input objects) arise as a result of a view containing predicates that filter the input data.

In the experiments, two template views (shown in Fig. 15a and b) are defined that retrieve attribute values from objects using different predicates to filter the input. Two sets of five views with different selectivities are derived by instantiating the template, as follows. Each view differs in the predicate  $\phi_i$  in the `where` clause. Each predicate is obtained by instantiating the template  $\phi_i$  with `c.id <= k_i` where  $k_1 = 1200$  to  $k_5 = 6000$  in increments of 1200, thereby yielding two sets of five different views with selectivities from 0.2 to 1.0 in increments of 0.2. In experiments involving

	Database Size				
	db1	db2	db3	db4	db5
modules	4	8	12	16	20
objects	288,380	576,760	865,140	1,153,520	1,441,900
size (MB)	45	90	134	178	222

Fig. 14. Databases used in the evaluation.

<pre> define selView_i() as select struct(ComPartId:c.id,              CType:c.type) from   c in CompositeParts where  <math>\phi_i</math> </pre>	<pre> define joinSelView_i() as select struct(ComPartId:c.id,              CType:c.type,              DocId:d.id) from   c in CompositeParts,        d in Documents where  c.documentation = d and        <math>\phi_i</math> </pre>
(a) Selectivity Dependent (on $\phi_i$ ) Views (I)	(b) Selectivity Dependent (on $\phi_i$ ) Views (II)
-----	
<pre> define complexView1() as select struct(ComPartId:c.id,              CType:c.type) from   c in CompositeParts where  c.id &lt;= 200 </pre>	<pre> define complexView2() as select struct(ComPartId:c.id,              CType:c.type,              AtomPartId:a.id) from   c in CompositeParts,        a in c.parts where  c.id &lt;= 200 and        c.buildDate &gt; a.buildDate </pre>
(c) complexView1: A Reduce-Get View	(d) complexView2 = complexView1 plus Unnest
-----	
<pre> define complexView3() as select struct(ComPartId:c.id,              CType:c.type,              AtomPartId:a.id,              BaseAssmId:b.id) from   c in CompositeParts,        a in c.parts,        b in c.usedInPriv where  c.id &lt;= 200 and        c.buildDate &gt; a.buildDate and        c.buildDate &gt; b.buildDate </pre>	<pre> define complexView4() as select struct(ComPartId:c.id,              CType:c.type,              AtomPartId:a.id,              BaseAssmId:b.id,              DocId:d.id) from   c in CompositeParts,        a in c.parts,        b in c.usedInPriv,        d in Documents where  c.id &lt;= 200 and        c.buildDate &gt; a.buildDate and        c.buildDate &gt; b.buildDate and        c.documentation=d </pre>
(e) complexView3 = complexView2 plus Unnest	(f) complexView4 = complexView3 plus Join

Fig. 15. View templates and views used in the evaluation.

selectivity, the database for which the view is defined is db3. In this database, there are 6000 compositePart objects. The id of these objects is set sequentially from 1 to 6000, so it is possible to control the selectivity of the view. Note that each compositePart object has an associated document object (i.e., there is a one-to-one relationship between their classes), therefore, by filtering the input based on  $c.id \leq k_i$ , document objects are filtered too, which yields different selectivities.

(2) *The structural complexity of the view influences the performance of IVM.* With regard to the structural complexity of the view, it is assumed to vary with the number and the kind of algebraic operators needed to evaluate the query part of the view. Intuitively, a query that contains two joins is more complex than a query that contains one join. Therefore, four views are defined (in

Update:	delete a <code>compositePart</code> object	
Event:	onDeleteFromExtent <i>CompositeParts</i>	
(a) U1: A Deletion Update Event		
-----		
Update:	assign "type111" to the <code>type</code> attribute of the <code>compositePart</code> object with <code>id = 500</code>	Update: insert a new object into the <code>Documents</code> extent
Event:	onModifyAttribute <i>compositePart.type</i>	Event: onInsertToExtent <i>Documents</i>
(b) U2: An Attribute Modification Event		(c) U3: An Insertion Update Event

Fig. 16. Update events used in the evaluation.

Fig. 15c–f) with increasing complexity in order to perform experiments over the materialized and virtual versions of the views.

(3) *The benefits of IVM increase with an increase in database size.* With regard to database size, the number of modules varies uniformly across the five databases. The interest here is in finding the impact of database size on the performance of IVM. A size-sensitive view, `dbSizeView` (as defined to the left of Fig. 2), provides a means of showing the impact of database size on the performance of MVs. The number of `compositePart` objects per module is 500 and the number of `document` objects is the same. In `db1`, there are 2000 `compositePart` and `document` objects. In contrast, in `db5`, there are 10 000 `compositePart` and `document` objects. The cardinality of the result of `dbSizeView` increases from 2000 in the case of `db1` to 10 000 in the case of `db5`.

*Queries.* The queries used in experiments for contrasting the time efficiency of answering queries against materialized and against virtual views are simple select statements over the virtual or materialized views. In Section 6.2, these queries are referred to as  $MQ_i$  and  $VQ_i$ ,  $1 \leq i \leq 7$ , when they are evaluated over the materialized and the virtual views, respectively.

*Updates.* The updates that are used in experiments involving propagation are shown in Fig. 16.

## 6.2. Experiments and discussion

The performance of IVM can be evaluated by considering:

- (1) The cost of query answering over the MV and over its virtual equivalent.
- (2) The cost of incrementally maintaining the MV by update propagation.
- (3) The cost of incrementally maintaining the MV in comparison with the cost of answering a query over its virtual equivalent.

Three experiments are designed to test each of the hypotheses in Section 6.1 exploring the above three kinds of cost. The experiments reported were run in cold states so that one experiment is not favoured against another due to object caching. After each individual experiment, the database server was shut down and restarted for the next experiment, and the Linux cache was flushed. Each individual experiment was run three times and the average taken.

### 6.2.1. Experiment 1: varying selectivity

This experiment explores the impact of view predicate selectivity on the performance of IVM. The experiment is broken into parts, each corresponding to one of the kind of costs specified in Section 6.2.

#### 6.2.1.1. Query answering

- *MQ1i and VQ1i*. The elapsed times taken to answer MQ1i and VQ1i,  $1 \leq i \leq 5$ , using the predicates defined with `selView_i` are measured and compared. The results, plotted in Fig. 17a, reveal that the cost of answering both MQ1i and VQ1i increases with an increase in selectivity. However, the relative cost of evaluating VQ1i compared with MQ1i reduces with increases in selectivity. This is consistent with expectations, because VQ1i is evaluated using an index-based scan over the extent of `CompositeParts` (which is efficient compared to a complete scan of the extent), whereas the number of objects scanned by MQ1i increases with selectivity. With selectivity 1.0, answering the query over the MQ1i is less costly than over the VQ1i because, although the queries retrieve the same number of objects, the view objects are smaller than `compositePart` objects.
- *MQ2i and VQ2i*. The elapsed times taken to answer MQ2i and VQ2i,  $1 \leq i \leq 5$ , using the predicates defined with `joinSelView_i` are measured and compared. The results, plotted in Fig. 17c, reveal that the cost of answering both MQ2i and VQ2i increases linearly with an increase in selectivity. In the case of VQ2i, the increase in the elapsed time is linear due the fact that, for the join algorithm, the size of the `Documents` extent remains constant, whereas the size of the predicated `CompositeParts` extent increases linearly with an increase in selectivity. In the case of MQ2i, the increase is due to a constant increase in the number of objects scanned by MQ2i, which increases with selectivity.

The gap between the elapsed time of VQ2i compared to MQ2i also shrinks with an increase in selectivity. With selectivity 0.2, the elapsed time of answering VQ2i is 139 times that for MQ2i. With selectivity 1.0, this reduces to 75 times, which shows that the benefit of answering queries over MVs is greater in the case of lower selectivities.

#### 6.2.1.2. View maintenance

- *Propagating U1 to selView\_i*. The elapsed time taken to propagate update U1 (in Fig. 16a)—which requires maintenance of `selView_i`—was measured using MOVIE and using re-materialization. The results are plotted in Fig. 17b. They reveal that the cost of view maintenance increases with an increase in selectivity in both cases but that MOVIE outperforms re-materialization by around 20 times in general. This suggests that, although the maintenance costs increase with an increase in selectivity, IVM can be advantageous independently of view selectivity.
- *Propagating U2 to joinSelView\_i*. The elapsed time taken to propagate update U2 (in Fig. 16b)—which requires maintenance of `joinSelView_i`—was measured using MOVIE and using re-materialization. The results are plotted in Fig. 17d. They reveal that the cost of view maintenance increases with an increase in selectivity in both cases but that MOVIE outperforms re-materialization by around 90 times in general. Comparing this with the results of that of Propagating U1 to `selView_i` shows that IVM can be more advantageous when views have

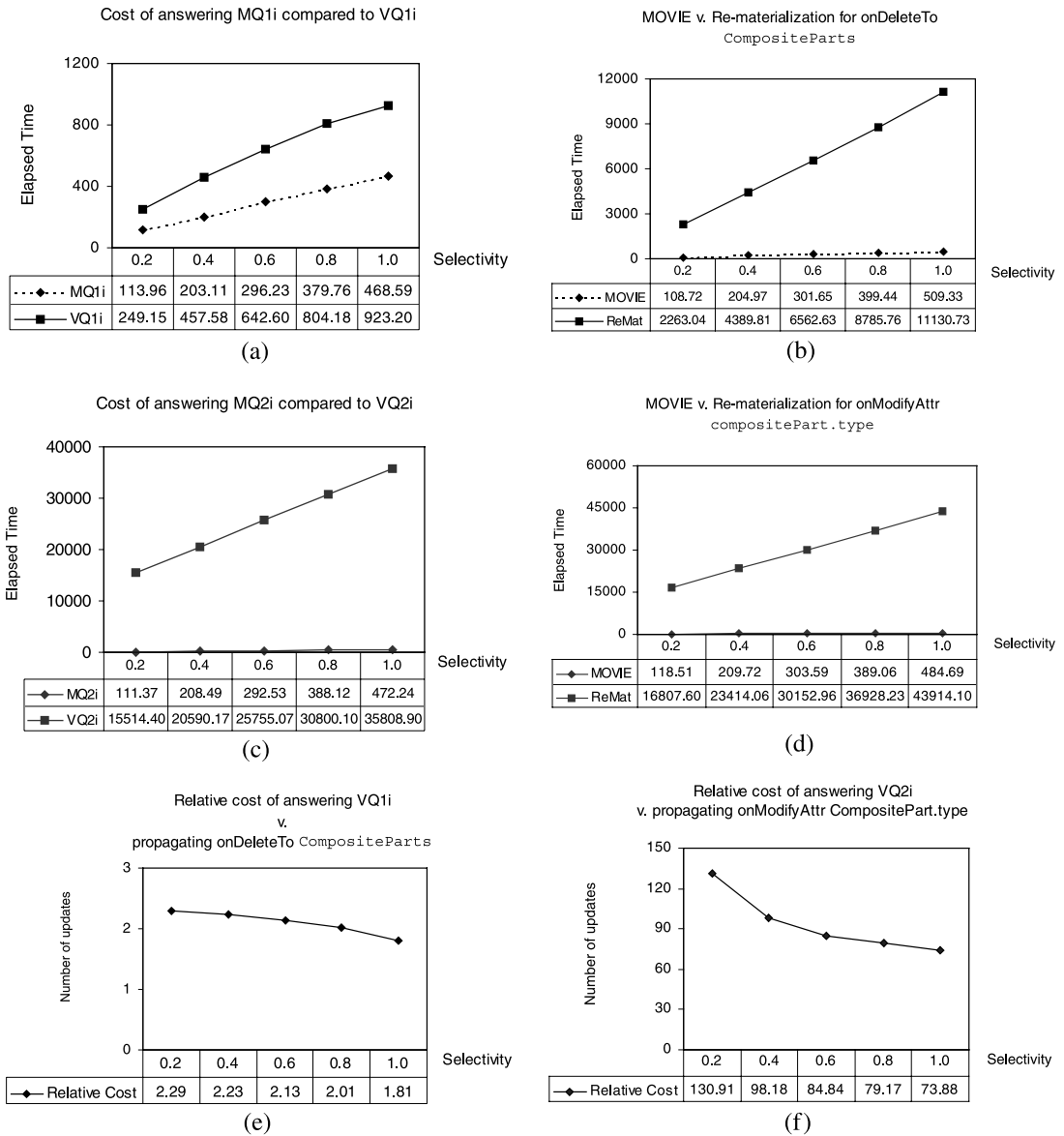


Fig. 17. Results of Experiment 1: query answering and view maintenance (I).

joins because the cost of joins is generally a significant factor in the overall cost of re-materialization.

6.2.1.3. *Relative cost (query/update)*. This part of the experiment tries to identify the conditions under which incrementally maintaining MVs is beneficial. This is done by investigating how many updates can be propagated incrementally to an MV in the time taken to answer the corresponding VQ.

- *Answering VQ1i v. Propagating U1.* The ratio of the elapsed time taken to answer VQ1i to that taken to propagate update U1 (in Fig. 16a) to `selView_i` is calculated. The results, plotted in Fig. 17e, reveal that, for the view with selectivity of 0.2, just over two deletes from extents can be propagated incrementally for the cost of evaluating a query against the corresponding virtual view. With an increase in the selectivity that ratio reduces slightly to just below two, suggesting that, for views with high selectivity, if deletes from extents are at all frequent, incremental maintenance may not be beneficial.
- *Answering VQ2i v. Propagating U2.* The ratio of the elapsed time taken to answer VQ2i to that taken to propagate update U2 (in Fig. 16b) to `joinSelView_i` is calculated. The results, plotted in Fig. 17f, reveal that, for the view with selectivity of 0.2, 130 attribute modifications can be propagated incrementally for the cost of evaluating a query against the corresponding virtual view. With an increase in the selectivity that ratio reduces to 73, suggesting that, for views with joins, incremental maintenance remains beneficial even when attribute modifications are frequent irrespective of view selectivity.

In summary, the benefits of IVM seem to be greater when views have low selectivities or involve joins. This result conforms with general expectations, in that the MVs with low selectivities involve smaller data sets than those associated with the corresponding virtual views and that the cost of joins is dominant factor in query answering as well as re-materialization.

### 6.2.2. Experiment 2: varying structural complexity

This experiment seeks to show how the complexity of the views may affect the performance of IVM. Again, the experiment is broken into parts, each one corresponding to one of the kind of costs specified in Section 6.2. In experiments involving complexity, the database used is db3.

**6.2.2.1. Query answering MQn and VQn.** The elapsed times taken to answer MQn and VQn,  $3 \leq n \leq 6$  are measured and compared. The results are plotted in Fig. 18a. They reveal that, in the case of MVs, because the cardinalities increase from `complexView1` to `complexView3`, the cost of answering queries over the MV increases significantly. However, since the cardinality of `complexView3` is the same as that of `complexView4` that cost increases by very little. This indicates that for access to MVs the dominant factor is cardinality. In the case of the virtual views, from `complexView1` to `complexView3` the cost increases but not very much as a consequence of the relatively low cost of unnesting. From `complexView3` to `complexView4` there is a jump in the cost because `complexView4` requires a join. This jump may be partly the result of the expensive algorithm (viz., Block Nested Loop) that  $\lambda$ -DB selects to join `CompositeParts` and `Documents`. If a more efficient algorithm were used, the jump would likely be less pronounced. In any case, the cost of answering queries over MVs increases with an increase in cardinality (rather than complexity), and thus the relative performance of MVs compared with their virtual counterparts depends very much on the selectivity of the view. Views with low selectivity are likely to be promising candidates for materialization.

**6.2.2.2. View maintenance: propagating U1 to `complexViewi`.** The elapsed time taken to propagate update U1 (in Fig. 16a) was measured using MOVIE and using re-materialization. The results are plotted in Fig. 18b. Update U1 (i.e., deletion of an instance of `compositePart`) was

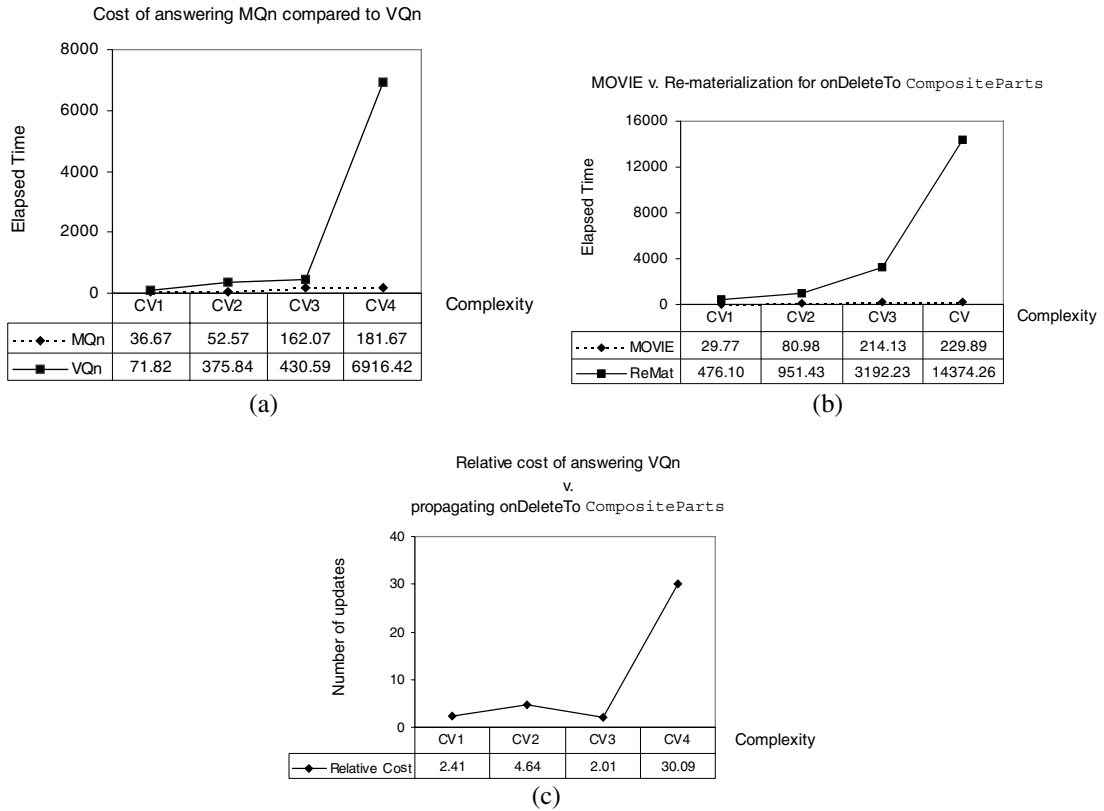


Fig. 18. Results of Experiment 2: query answering and view maintenance (II).

chosen because it affects all of complexView1 to complexView4. The results reveal that, for complexView1 and complexView2, MOVIE is 16 and 11 times faster than re-materialization, respectively. The decrease is due to the fact that U1 requires the deletion of 20 objects from complexView2 as against 1 for complexView1. For complexView3 MOVIE is 15 times faster than re-materialization. The increase in MOVIE performance pertaining to complexView3 (in comparison with complexView2) is due to the fact that the IMPs for incrementally maintaining both complexView2 and complexView3 join a delta with materialized OIDs. Therefore, in case of complexView3 the re-materialization cost is significantly affected by the increase in complexity, whereas the MOVIE cost is affected by the increase in the cardinality of the view extent (i.e., 1840 for complexView3 against 400 for complexView2). For complexView4 MOVIE is 62 times faster than re-materialization. This large increase in the performance of MOVIE from complexView3 to complexView4 is due to two factors. Firstly, re-materialization of complexView4 requires a join for which  $\lambda$ -DB selects an expensive algorithm (viz., Block Nested Loop). Secondly, the cost of increases by very little (i.e., from 214.13 milliseconds, in the case of complexView3, to 229.89 for complexView4).

The elapsed time for re-materializing complexView4 is four times more than that for complexView3. However, MOVIE takes nearly as much time to propagate changes to com-

plexView4 as it takes in the case of complexView3 as the cardinality of both views is the same (i.e., 1840). Similarly, MOVIE takes twice as long to propagate changes to complexView3 as it takes in the case of complexView2 because the cardinality of complexView3 is four times that of complexView2. The bottom line is that, on the one hand, the more complex the view the more expensive it is to maintain by re-materialization; on the other hand, incremental maintenance becomes more expensive with an increase in the cardinality of the view extent.

*6.2.2.3. Relative cost (query/update): answering VQ<sub>n</sub>, 3 ≤ n ≤ 6, v. propagating U1.* In this experiment, the ratio of the elapsed time taken to answer queries over complexView1Vir to complexView4Vir to that taken to propagate update U1 (in Fig. 16a) to the corresponding view is calculated. The results, plotted in Fig. 18c, show that only quite a small number of deletions from extents (1–4) can be propagated in the time taken to evaluate the virtual view in the case of complexView1 to complexView3. In case of complexView4, the ratio increases to 30, mainly because of the expensive join algorithm used for evaluating the query over the virtual view and partly because the cost of incremental maintenance does not increase proportionately. The shape of the graph in Fig. 18c should not be considered to be of general significance. A peculiar point in the graph appears to be the propagation of U1 to complexView3. This is principally because of the high cost of incrementally propagating U1 to complexView3, and this in turn is caused by the fact that a single occurrence of U1 event causes 40 objects to be deleted from the view.

In summary, the experiment has not demonstrated any conclusive pattern relating to increased view complexity. The most striking result, from Fig. 18a and b, is the importance of the cardinality of the view result to the incremental maintenance process.

### 6.2.3. Experiment 3: varying database size

This experiment explores the impact of database size on the performance of IVM. Once again, the experiment is broken into parts that correspond to the hypotheses being tested.

*6.2.3.1. Query answering: MQ7 and VQ7.* Here the elapsed times taken to answer MQ7 and VQ7 are measured and compared. The results are plotted in Fig. 19a. In the case of MQ7, the elapsed time taken to answer a query over dbSizeView increases only slightly with an increase in the database size, reflecting the increase in the size of the MV. The benefit of answering queries over the MV rather than over the virtual counterpart increases as the database increases in size. For example, for the smallest database, it is 38 times faster to use the MV, whereas for the largest database, it is 95 times faster, because the join cost grows non-linearly.

*6.2.3.2. View maintenance: propagating U2 and U3 to dbSizeView.* The elapsed times taken to propagate two different updates: U2 (in Fig. 16b) and U3 (in Fig. 16c)—which require maintenance of dbSizeView—were measured using MOVIE and using re-materialization. The results are plotted in Fig. 19b and c. Fig. 19b shows MOVIE outperforming re-materialization for an update event that modifies an attribute. For update U2, the IMP is generated by joining the delta with the materialized OIDs, i.e., no base data is accessed. The relative performance of MOVIE increases with an increase in the database size because costly joins with base data are avoided altogether.

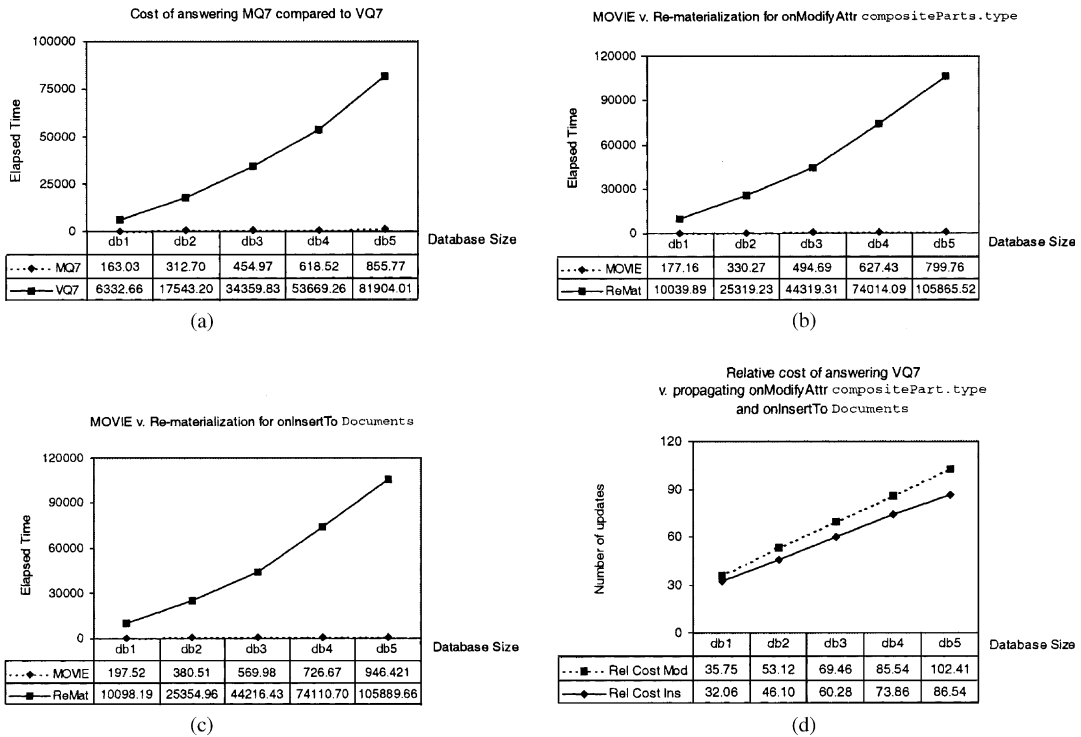


Fig. 19. Results of Experiment 3: varying database size.

Fig. 19c does a similar comparison for an update (viz., U3, which is an insert into a class extent) for which the IMP is generated by planting a delta, i.e., accessing base data. Although planting a delta is generally felt to be more expensive than joining with the materialized OIDs, MOVIE still performs better than re-materialization. In fact, comparing the times in Fig. 19b and c, as expected, joining with the materialized OIDs works out faster than planting the delta. Once again, the benefits of MOVIE increase with an increase in the database size. For example, in the smallest database, it performs 56 times better, whereas in the largest, it is 132 times faster than re-materialization (based on the results in Fig. 19b).

6.2.3.3. *Relative cost (query/update): answering VQ7 v. propagating U2 and U3.* The ratio of the elapsed time taken to answer VQ7 to that taken to propagate updates U2 and U3 (in Fig. 16b and c, respectively) to dbSizeView is calculated. The results, plotted in Fig. 19d, reveal that, for the smallest database, 35 attribute modifications and 32 insertions can be propagated incrementally for the cost of evaluating VQ7. With an increase in database size the ratios continue to grow.

In summary, IVM seems to be more beneficial with larger databases. For the examples, query answering over MVs is always better than over virtual views, no matter the database size, and the difference increases as size increases. The same behaviour is observed if one pitches incremental maintenance against re-materialization. The benefits for both the small and large databases used here indicate that a significant number of update events can be carried out for the cost of answering the query over a non-materialized view.

## 7. Related work

This section discusses previous work on the IVM problem that is closely related to the work reported in this paper. The summary of comparison of related work is given in Table 1.

### 7.1. On relational view maintenance

The incremental maintenance of MVs has been studied in detail for relational DBMSs [3,8,14,16–18,20,25,32]. Ref. [3] provides a differential algorithm for maintaining SPJ views. 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. Refs. [8,14,17,31] support the maintenance of MVs with varying degrees of expressiveness in the view language. However, they all require access to the tables referenced in view definitions. In contrast, in the MOVIE solution, views can be incrementally maintained without accessing base tables for some events. The advantages of the MOVIE solution are not limited to the OO setting i.e., MOVIE algorithms can be adopted into a relational setting (e.g., by storing keys for all the tables referenced in view definitions instead of OIDs).

### 7.2. On object-based view maintenance

Recently, a few proposals have addressed view maintenance in the context of object-based database systems [1,13,22,24,31]. These constitute the work more closely related to the one reported in this paper. Ref. [31] proposes a system called Squirrel 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. Squirrel propagates incremental updates to MVs in response to inserts and deletes only, whereas MOVIE supports all update operations defined by the ODMG. Ref. [13] addresses the IVM problem for a subset of OQL views for a variant of the ODMG object model. 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.

Ref. [22] provides a system called MultiView that is based on a non-standard object model and query language, whose algebraic operators are very close to those of relational algebra. 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 MOVIE avoids accessing base extents for views containing arbitrary number of joins for certain update operations.

Ref. [1] presents algorithms for the incremental maintenance of SP views in the context of a non-standard object model and query language that only support *select-project* views and only handle classical update operations. Finally, [24] proposes a complete solution to the IVM problem in an object-relational (OR) system. The algorithm generates OR-SQL [30] statements, which compute and propagate changes to the MVs. The algorithm is similar to [17], which means that access to base data is required for all update types. In contrast, in MOVIE, for certain update types and certain views, base data is not accessed.

Table 1  
Summary comparison of proposals for the IVM problem

Proposal	Definition language	View language	Environmental information	Manipulation language	Event processing	Performance evaluation
[3]	Relational	Relational algebra (SPJ)	Base data, MVs	Insert, delete	Immediate?	No
[8]	Relational	SQL (SPJ, SQ, $\cup$ , $\cap$ )	Base data, MVs	Insert, delete, modify	Active	No
[17]	Relational	SQL/datalog (SPJ, $\exists$ , $\varnothing$ , $\cup$ )	Base data, MVs	Insert, delete, modify	Immediate?	No
[32]	Relational	Relational algebra (SPJ, $\varnothing$ )	Base data, MVs	Insert, delete	Immediate	Yes (model based)
[14]	Relational	Relational algebra with bag semantics (SPJ, $\exists$ , $\varnothing$ , $\cup$ )	Base data, MVs	Insert, delete	Immediate?	No
[16]	Relational	SQL (SPJ)	Partial-base data, MVs, extended MVs	Insert, delete	?	No
[31]	Mix of relational and ODMG-93	Extended OQL (SPJ, $\cup$ , -)	Base data, MVs, auxiliary information for intermediate nodes	Insert, delete	Active	No
[13]	OO (variant of ODMG 2)	OQL (SPJ, $\exists$ , $\cup$ , $\cap$ , -)	Base data, MVs	Insert, delete, modify	?	No
[22]	OO (non-standard)	Object algebra (SP, $\cup$ , $\cap$ , -)	Base data, MVs	Insert, delete, modify	Immediate	Yes (experimental)
[1]	OO (non-standard)	SP	Base data, MVs	Insert, delete, modify	Deferred	No
[18]	Relational	SQL (SPJ, $\exists$ , outer-joins)	Base data, MVs, change table	Insert, delete	Deferred	Yes (model based)
[24]	Object relational	OR-SQL (SPJ, $\exists$ )	Base data, MVs	Insert, delete	Active	Yes (experimental)
[25]	Relational	SPJ	Base data, MVs, additional transient or permanent MVs	Insert, delete	Immediate?	Yes (model based)
[23]	Relational	SPJ	Base data, MVs	Insert, delete	Immediate	Yes (experimental)
[20]	Relational	SQL (SPJ)	Base data, MVs	Insert, delete, modify	Active	Yes (model based)
<b>MOVIE</b>	<b>OO (ODMG 3)</b>	<b>OQL (SPJ, <math>\mu</math>, <math>\Gamma</math>, SQ)</b>	<b>Base data, MVs, OIDs of base objects</b>	<b>Insert/delete in extents and in collections, modification to attributes and relationships</b>	<b>Immediate</b>	<b>Yes (experimental)</b>

### 7.3. On performance evaluation of view materialization systems

In [19], a cost model is used to compare the performance of immediate and deferred view materialization algorithms with that of virtual views. The study reveals that the performance of MVs and their virtual correspondents is sensitive to (1) selectivity of the view predicate, (2) probability of updates, (3) the selectivity of the query over the view, and (4) the number of tuples affected by each update. In addition to selectivity, the performance evaluation in this paper also studies the impact of complexity and database size.

In [4], the performance of MVs is compared against the use of join indexes and hybrid-hash joins in virtual views. Their study is based on a cost model and reveals that selectivity, update activity, the probability of update to the joining attributes, and the size of tables and memory have influence on the performance of these methods. In [32], the eager compensation algorithm (ECA) is compared with re-materialization using a cost model. The issue of whether to choose re-materialization or ECA depends on favourable conditions for either of the algorithms. However, their study does not investigate the query/update ratio, which indicates whether it is beneficial to incrementally maintain the view. In [21], the performance study reveals a similar impact of selectivity to the one detected in this paper. They also address query/update issues and show that network traffic for MVs is proportional to the update rate.

The authors of this paper are not aware of any research that has sought to validate experimentally the analytical models presented by Hanson, or that has systematically assessed the performance of view maintenance techniques in relational (or OO) systems. Although analytical models have been proposed in an OO setting (e.g., [22]), they are aware neither of any thorough performance studies based on such models nor of comprehensive empirical studies of views in object databases. Again as an example, [22] provides a few illustrative examples of the performance of MultiView, but cannot be said to be a systematic study.

## 8. Conclusions

The solution presented here has the following salient points in relation to previous work:

- (1) It is one of few that cover the IVM problem from event specification, to IMP generation, to update propagation.
- (2) It is the first algebraic approach to the IVM problem in the context of object databases.
- (3) It works for a wider class of MVs than the only previous work on the IVM problem for OQL views (viz., Squirrel [31]).
- (4) It also has lower space overheads than Squirrel [31] for the MVs that both solutions cover.
- (5) It is fully implemented as an extension of  $\lambda$ -DB [10,12] whereas many IVM proposals give no evidence of associated implementations.
- (6) It is one of few whose performance has been evaluated over all the steps comprising the IVM problem.

In addition, this paper presents the most thorough experimental performance evaluation of MVs to date. Experiments have been conducted that assess the performance of query answering

and update propagation for different selectivities, query complexities and database sizes. Research in IVM is predicated on two hypothesis:

- (1) That incremental propagation of updates can be expected to perform significantly better than re-materialization.
- (2) That incremental propagation of updates can provide improved performance overall for transactions that mix queries and updates, as long as updates are not dominant.

The experiments reported in this paper broadly support (1), with incremental update propagation generally being significantly faster than re-materialization. However, in terms of (2), although some results reported in the paper indicate that many incremental updates can be propagated in the time it takes to evaluate the corresponding virtual view, there are other cases in which the cost of propagating a single update is almost as great as the cost of evaluating the view. Thus this paper can be seen as enforcing the need for judicious selection of views for materialization, with many different issues impacting the decision making process.

## Acknowledgements

M. Akhtar Ali gratefully acknowledges the support of the Commonwealth Scholarship Commission in the United Kingdom (Grant PK0257). The authors would like to thank Leonidas Fegaras for making available to the research community the  $\lambda$ -DB system and for helping them make use of it.

## References

- [1] R. Alhajj, A. Elnagar, Incremental materialization of object-oriented views, *Data and Knowledge Engineering* 29 (2) (1999) 121–145.
- [2] M.A. Ali, *Incremental Maintenance of Materialized Views in Object-Oriented Databases*, Ph.D. thesis, Department of Computer Science, University of Manchester, July 2003.
- [3] J.A. Blakeley, P.-A. Larson, F. Tompa, Efficiently updating materialized views, in: *Proc. SIGMOD*, 1986, pp. 61–71.
- [4] J.A. Blakeley, N.L. Martin, Join index, materialized view, and hybrid-hash join: a performance analysis, in: *Proc. ICDE*, 1990, pp. 256–263.
- [5] M. Carey, D. DeWitt, J. Naughton, The OO7 benchmark, in: *Proc. SIGMOD*, 1993, pp. 12–21.
- [6] 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, M.J. Zwilling, Shoring up persistent applications, in: *Proc. SIGMOD*, 1994, pp. 383–394.
- [7] R.G.G. Cattell (Ed.), *The Object Database Standard: ODMG 3.0*, Morgan Kaufmann, 2000.
- [8] S. Ceri, J. Widom, Deriving production rules for incremental view maintenance, in: *Proc. VLDB*, 1991, pp. 577–589.
- [9] L. Fegaras, Query unnesting in object-oriented databases, in: *Proc. SIGMOD*, 1998, pp. 49–60.
- [10] L. Fegaras,  $\lambda$ -DB Home Page, 1999. Available at <http://lambda.uta.edu/lambda-DB.html>.
- [11] L. Fegaras, D. Maier, Optimizing object queries using an effective calculus, *ACM TODS* 25 (4) (2000).

- [12] L. Fegaras, C. Srinivasan, A. Rajendran, D. Maier,  $\lambda$ -DB: An ODMG-Based Object-Oriented DBMS, in: Proc. SIGMOD, 2000, p. 583.
- [13] D. Gluche, T. Grust, C. Mainberger, M. Scholl, Incremental updates for materialized OQL views, in: Proc. DOOD, 1997, pp. 52–66.
- [14] T. Griffin, L. Libkin, Incremental maintenance of views with duplicates, in: Proc. SIGMOD, 1995, pp. 328–339.
- [15] T. Grust, M.H. Scholl, Translating OQL into Monoid Comprehensions—Stuck with Nested Loops? Technical Report 3a/1996 (Revised), Department of Mathematics and Computer Science, University of Konstanz, 1996.
- [16] A. Gupta, J.A. Blakeley, Using partial information to update materialized views, *Information Systems* 20 (8) (1995) 641–662.
- [17] A. Gupta, I. Mumick, V. Subrahmanian, Maintaining views incrementally, in: Proc. SIGMOD, 1993, pp. 157–166.
- [18] H. Gupta, Selection and Maintenance of Views in a Data Warehouse, Ph.D. thesis, Stanford University, September 1999.
- [19] E.N. Hanson, A performance analysis of view materialization strategies, in: Proc. SIGMOD, 1987, pp. 440–453.
- [20] E.N. Hanson, S. Bodagala, U. Chadaga, Trigger condition testing and view maintenance using optimized discrimination networks, *IEEE Transactions on Knowledge and Data Engineering* 14 (2) (2002) 261–280.
- [21] R. Hull, G. Zhou, Towards the study of performance trade-offs between materialized and virtual integrated views, in: Proc. VIEWS, 1996, pp. 91–102.
- [22] H. Kuno, E. Rudensteiner, Incremental maintenance of materialized object-oriented views in multiview: strategies and performance evaluation, *IEEE TKDE* 10 (5) (1998) 768–792.
- [23] K.Y. Lee, J.H. Son, M.-H. Kim, Efficient incremental view maintenance in data warehouses, in: Proc. CIKM1, 2001, pp. 349–357.
- [24] J. Liu, M. Vincent, M. Mohania, Maintaining views in object-relational databases, in: Proc. CIKM, 2000, pp. 102–109.
- [25] H. Mistry, P. Roy, S. Sudarshan, K. Ramamritham, Materialized view selection and maintenance using multi-query optimization, in: Proc. SIGMOD, 2001, pp. 307–318.
- [26] N.W. Paton (Ed.), *Active Rules in Database Systems*, Springer-Verlag, 1999.
- [27] P.R.F. Sampaio, N.W. Paton, Query processing in DOQL: a deductive database language for the ODMG model, *Data and Knowledge Engineering* 35 (1) (2000) 1–38.
- [28] J. Smith, P. Watson, S. de F.M. Sampaio, N.W. Paton, Polar: an architecture for a parallel ODMG compliant object database, in: Proc. CIKM, 2000, pp. 352–359.
- [29] I. Stanoi, D. Agrawal, A.E. Abbadi, View derivation graph with edge fitting for adaptive data warehousing, in: Proc. DaWaK, 2000, pp. 67–76.
- [30] M. Stonebraker, P. Brawn, *Object-relational DBMSs tracking the next great wave*, Morgan Publishers, 1999.
- [31] G. Zhou, R. Hull, R. King, Generating data integration mediators that use materialization, *JIS* 6 (2/3) (1996) 199–221.
- [32] Y. Zhuge, H. Garcia-Molina, J. Hammer, J. Widom, View maintenance in a warehousing environment, in: Proc. SIGMOD, 1995, pp. 316–327.



**M. Akhtar Ali** obtained his B.Sc. degree in Computer Science and Mathematics, and M.Sc. degree in Computer Science from Peshawar University, Peshawar, Pakistan in 1993 and 1995, respectively. He joined the Army Welfare Trust, Rawalpindi, Pakistan, as a systems analyst in March 1996 and worked on the development of several business information systems. Then he worked as a teaching assistant in the Faculty of Computer Science and Engineering, Ghulam Ishaq Khan Institute of Engineering and Technology, Topi, Pakistan, from September 1996 to August 1997. He was awarded the Commonwealth Scholarship by the Commonwealth Scholarship Commission in June 1997. In September 1997, he joined the Information Management Group at Manchester University as a Ph.D. candidate in object-based database management systems. In August 2000, he joined Northumbria University where he works as a senior lecturer in database systems. He is interested in research on materialized views, data warehousing, object-based database systems and query optimization.



**Alvaro A.A. Fernandes** is a Lecturer in Computer Science at the University of Manchester. He was awarded an M.Sc. in Knowledge-Based Systems by the University of Edinburgh in 1989. From 1990 to 1995, at Heriot-Watt University in Edinburgh, he was a member of the team that designed and implemented the ROCK & ROLL deductive object-oriented database system. He was awarded a Ph.D. in Computer Science in 1995. In 1996 he was appointed for a lectureship at Goldsmiths College (University of London), where he worked on personalization and adaptivity mechanisms. In 1998 he joined the Information Management Group in Manchester. Besides MOVIE, he has worked on Tripod (an ODMG-compliant spatio-historical database system), and on Polar\* (a parallel ODMG-compliant database management system distributed over the Grid). His principal current interests are in distributed query processing in computational grids, spatio-temporal databases and logic-based approaches to knowledge discovery and their application to knowledge management, particularly the integration of deductive and inductive techniques in database settings.



**Norman W. Paton** is a Professor of Computer Science at the University of Manchester. He obtained a B.Sc. in Computing Science from Aberdeen University in 1986, and a Ph.D. from the same institution in 1989. He was a lecturer in Computer Science at Heriot-Watt University, 1989–1995, before moving to Manchester, where he co-leads the Information Management Group. His research interests have principally been in databases, in particular active databases, spatial databases, deductive object-oriented databases and user interfaces to databases. He is currently working on spatio-temporal databases, distributed information systems, Grid data management, and on genome databases.