An Experimental Performance Evaluation of Incremental Materialized View Maintenance in Object Databases

M. Akhtar Ali[1], Norman W. Paton[2], and Alvaro A.A. Fernandes[2]

[1] School of Computing and Mathematics, University of Northumbria
Newcastle Newcastle Upon Tyne NE1 8ST, UK
akhtar.ali@unn.ac.uk

[2] Department of Computer Science, University of Manchester,
Oxford Road, Manchester M13 9PL, UK.
{norm,alvaro}@cs.man.ac.uk

Abstract. The development of techniques for supporting incremental maintenance of materialized views has been an active research area for over twenty years. However, although there has been much research on methods and algorithms, there are surprisingly few systematic studies on the performance of different approaches. As a result, understanding of the circumstances in which materialized views are beneficial (or not) can be seen to lag behind research on incremental maintenance techniques. This paper presents the results of an experimental performance analysis carried out in a system that incrementally maintains OQL views in an ODMG compliant object database. The results indicate how the effectiveness of incremental maintenance is affected by issues such as database size, and the complexity and selectivity of views.

1 Introduction

Incremental view maintenance is the propagation to a materialised view of updates to base data that affect the value of the view. Proposals for solutions to the incremental view maintenance (IVM) problem are numerous [7], and individual solutions differ in the data model, view language, data to which access is required, and timeliness of update propagation. The variety of different techniques available and the intrinsic complexity of the trade-offs involved in deciding when to materialize has promoted the selection of views for materialization to a research topic in its own right [4,10]. However, although materialized view selection is crucially dependent on an understanding of the cost issues associated with incremental maintenance, there has been surprisingly little work seeking systematically to evaluate the performance of proposals for IVM systems.

This paper presents the results of an experimental performance evaluation of the MOVIE (Materialized Object-oriented VIEws) system [1], which provides an algebraic solution to the incremental maintenance of OQL [3] views. The performance evaluation has made use of the OO7 [2] schema and data generation procedure. The paper is structured as follows. Section 2 provides an overview of the MOVIE system including some of the technical background that underpins it. Section 3 describes the schema, queries, views and updates used in the experiments. Section 4 presents the experiments conducted in the performance evaluation. Finally, Section 5 concludes and points to future work.

2 Incremental Maintenance of OQL Views

2.1 Technical Background

The ODMG standard for object databases includes an object model (OM), an object definition language (ODL) and an object query language (OQL). Familiarity with the ODMG model and OQL is assumed to the level that is required to interpret schema diagrams and OQL queries.

The λ-DB system [6] implements a monoid algebra over the ODMG object model (not including arrays and dictionaries). OQL queries in λ-DB are translated into algebraic expressions that are optimized before they are mapped to physical execution plans. An execution plan is then translated into C++ and compiled. The resulting executable, when run, evaluates the original OQL query.

2.2 An Overview of MOVIE

The MOVIE system is an implementation of our solution to the IVM problem for OQL views, and is described more fully in [1]. The solution assumes the availability of the update event, the changes made to the database (hereafter referred to as the *delta)*, the MV definition, and the current materialized state of the view. In order to achieve incremental maintainability for all update operations, we also assume the availability of references to the base objects that contribute data to the MV, and of the base extents required for materializing it. Our solution works for MVs that refer to any ODMG type (excluding array and dictionary) and that are definable using the reduce, join, get, nest and unnest bulk operators of the algebra proposed in [6] (excluding self joins). It is valid for any update operation in the ODMG standard (e.g., `new()` and `delete()` on objects, `insert_element()` on collections, etc.). In terms of practicality, our solution yields incremental maintenance plans (IMPs) at the algebraic level and, to the best of our knowledge, for object databases, is the first one to do so. This makes it easier to integrate our solution into the kind of query processing frameworks that mainstream database management systems (DBMSs) rely on (e.g. [5]).

When an MV is defined, its definition is traversed to identify the kinds of update events that might require propagation of changes to the MV. For each kind of event, an algebraic IMP is constructed that can compute the required changes to the MV. The core of our solution is, therefore, the generation of an IMP that is appropriate for each kind of update event. In our approach, two kinds of IMP (described below) suffice to compute the changes required in the MV as a result of any update operation in the ODMG standard. Immediately after an update event takes place which implies the need to update an MV, the corresponding delta (comprising the old and the new state of affected objects) is made available and the associated IMP (which uses the delta) is evaluated to compute the changes needed.

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

Different forms of IMPs are generated depending on the event type and the category of the view[1], as follows. We note that similar categories of plan are present in other IVM systems, and thus that lessons learned from the experiments described below are potentially relevant to other proposals.

1. **Planting a delta** – For some kinds of updates (e.g. `onInsertTo`), the constructed IMP computes the changes required to `v` by evaluating the view 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. As well as generating the additional data for the view, the incremental plan also projects the OIDs of the objects represented in the view.
2. **Joining a delta with materialized OIDs** – For other kinds of updates (e.g. `onDeleteTo`), the constructed IMP 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. This only applies to ReduceGetJoin views.

3 Experimental Views, Queries, and Updates

Databases: The databases used in our experiments are based on the OO7 Benchmark [2]. We did not use the three OO7 database size configurations, but rather generated databases of different sizes by varying the number of modules (the top level class in OO7). The actual parameters given to the OO7 data generation program, plus some information on the resulting database, are given in Figure 1 (where the database sizes correspond to four, eight, twelve, sixteen and twenty times the size of the small database in the OO7 benchmark).

Testbed: The experiments reported in this paper were carried out on a PC with the following hardware and software: Intel Pentium Pro processor, 200 MHz, 256KB cache, 128MB RAM, 4GB SCSI Hard Disk (where the system software and 128MB of swap space reside); RedHat Linux 6.0 Kernel 2.2.5-22, SHORE 1.1.1 and λ-DB 0.5.

[1] A complete list of event types and view categories, the latter characterised by the algebraic operators that are used to evaluate the view, is given in [1].

	Database Size				
	db1	db2	db3	db4	db5
fan-out	6	6	6	6	6
modules	4	8	12	16	20
objects	288,380	576,760	865,140	1,153,520	1,441,900
size (MB)	45	90	133	177	222

Fig. 1. Databases Used in the Evaluation

Views: The views we use are given in Figure 2. Note that for each MV v, we denote the corresponding non-materialized view by $vVir$. For example, to the view `complexView1` (in Figure 2.b) there corresponds a view named `complexView1Vir` that is otherwise identical to `complexView1` but is not materialized. We have defined the views in Figure 2 for the purpose of investigating the following hypotheses:

1. The selectivity of the view influences the performance of incremental view materialization.
2. The structural complexity of the view influences the performance of incremental view materialization.
3. The benefits of incremental view materialization increase with an increase in database size.

Selectivity: 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 (to different degrees) the input data.

In the experiments, a template view (shown in Figure 2.a) is defined that retrieves the `id` and `type` of `compositePart` objects using different predicates to filter the input. A set of five views with different selectivities is derived by instantiating the template, as follows. Each view differs in the predicate ϕ_i in the `where` clause. Each predicate is obtained by instantiating the template ϕ_i with `c.id` $<=k_i$ where $k_1 = 1200$ to $k_5 = 6000$ in increments of 1200, thereby yielding five different views with selectivities from 0.2 to 1.0 in increments of 0.2. In experiments involving 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.

Complexity: With regard to the structural complexity of the view, we take it 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, we define four views (in Figures 2.b to 2.e) with increasing complexity in order to perform experiments over the materialized and virtual versions of the views.

Database Size: With regard to database size, the number of modules varies uniformly across the five databases. We are interested in finding the impact of

```
define viewSel_i() as
select struct(
              ComPartId:c.id,
              CType:c.type)
from   c in CompositeParts
where  φ_i
```

(a) Selectivity Dependent (on ϕ_i) Views

```
define complexView1() as
select struct(
              ComPartId:c.id,
              CType:c.type)
from   c in CompositeParts
where  c.id <= 200
```

(b) `complexView1`: A Reduce-Get View

```
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 <= 200 and
       c.buildDate > a.buildDate
```

(c) `complexView2` = `complexView1` plus Unnest

```
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<=200 and
       c.buildDate>a.buildDate and
       c.buildDate>b.buildDate
```

(d) `complexView3` = `complexView2` plus Unnest

```
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<=200 and
       c.buildDate>a.buildDate and
       c.buildDate>b.buildDate and
       c.documentation=d
```

(e) `complexView4` = `complexView3` plus Join

```
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
```

(f) `dbSizeView`: A Size-Sensitive View

Fig. 2. View Templates and Views Used in the Evaluation

```
Update:delete a compositePart object
Event: onDeleteTo CompositeParts
```

(a) U1: A Deletion Update Event

```
Update:assign "type111" to the type
       attribute of the compositePart
       object with id = 500
Event: onModifyAttr compositePart.type
```

(b) U2: An Attribute Modification Event

```
Update:insert a new object into
       the Documents extent
Event: onInsertTo Documents
```

(c) U3: An Insertion Update Event

Fig. 3. Update Events Used in the Evaluation

database size on the performance of incremental view materialization. A size-sensitive view, `dbSizeView` (in Figure 2.e), provides a means of showing the impact of database size on the performance of materialized views. 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 10000 `compositePart` and `document` objects. The cardinality of the result of `dbSizeView` increases from 2000 in the case of `db1` to 10000 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 derived or materialized views. In Section 4, these are referred to as MQ_i and VQ_i, $1 \leq i \leq 6$, when they are evaluated over the materialized and the virtual views, respectively.

Updates: The updates that are used in experiments involving propagation are given in Figure 3.

To exemplify the two kinds of IMP that are generated by MOVIE, consider the `dbSizeView` in Figure 2.f. For the update event U2 in Figure 3.b to be incrementally propagated, changes are computed by joining a delta with materialized OIDs. This results in the algebraic query plan shown in Figure 4.a. In contrast, for the update event U3 in Figure 3.c to be incrementally propagated, changes are computed by planting a delta. This results in the algebraic query plan shown in Figure 4.b. The nature of these plans is discussed in more detail in [1], and the mapping from OQL to the object algebra is described in [6].

```
reduce(set,
    join(set,
        get(set, $\Delta_{\text{onModifyAttr compositePart.type}}$,$\delta$,and()),
        get(set,OIDs_for_dbSizeView,mat_oids,and()),
        and(eq(mat_oids.DO, $\delta$)),
        none),
    X1,
    struct(mat_obj=mat_oids.VO, CType=$\delta$.type),
    and())
                                    (a)
reduce(set,
    join(set,
        get(set, $\Delta_{\text{onInsertTo Documents}}$,d,and()),
        get(set,CompositeParts,c,and()),
        and(eq(c.documentation, d)),
        none),
    X1,
    struct(ComPartId=c.id), CType=c.type, DocId=d.id),
    and())
                                    (b)
```

Fig. 4. Computing Changes to `dbSizeView` Following (a) U2 and (b) U3

4 Experiments and Discussion

The performance of incremental view materialization 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.

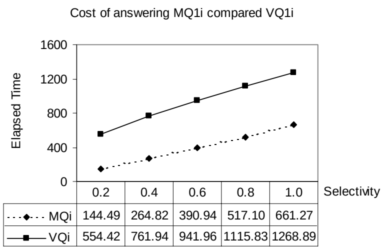
Three experiments are designed to test each of the hypotheses in Section 3 exploring the above three kinds of cost. The experiments reported were run in cold states so that one experiment is not favoured against another due to object caching. After each individual experiment, the database server was 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. The values reported for relative cost are rounded to the nearest integer value.

4.1 Experiment 1: Varying Selectivity

This experiment explores the impact of view predicate selectivity on the performance of IVM. We break the experiment into parts, each corresponding to one of the hypotheses under scrutiny.

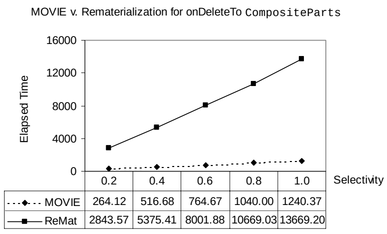
Query Answering: MQ1i and VQ1i. The elapsed times taken to answer MQ1i and VQ1i, $1 \leq i \leq 5$, using the predicates defined with `viewSel_`i are measured and compared. The results are plotted in Figure 5.a.

They 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 evaluating VQ1i always requires scanning of the complete extent of `CompositeParts`, 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.



	0.2	0.4	0.6	0.8	1.0
MQi	144.49	264.82	390.94	517.10	661.27
VQi	554.42	761.94	941.96	1115.83	1268.89

(a) Query Answering



	0.2	0.4	0.6	0.8	1.0
MOVIE	264.12	516.68	764.67	1040.00	1240.37
ReMat	2843.57	5375.41	8001.88	10669.03	13669.20

(b) View Maintenance

Fig. 5. Results of Experiment 1: Varying Selectivity

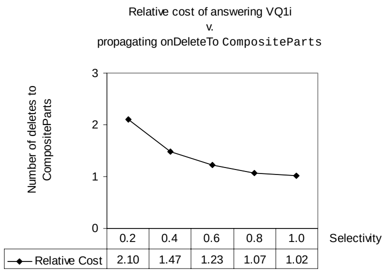
View Maintenance: Propagating U1 to `viewSel_i`. We measure the elapsed time taken to propagate update U1 (in Figure 3.a) – which requires maintenance of `viewSel_`i – using MOVIE and using rematerialization. The results are plotted in Figure 5.b.

They reveal that the cost of view maintenance increases with an increase in selectivity in both cases but that MOVIE outperforms rematerizalization by around ten times in general. This suggests that, although the maintenance costs increase with an increase in selectivity, IVM can be advantageous independently of view selectivity.

Relative Cost (Query/Update): Answering VQ1i v. Propagating U1. This part of the experiment tries to identify the conditions under which incrementally maintaining MVs is beneficial. We do so by investigating how many updates can be propagated incrementally to an MV in the time taken to answer the corresponding virtual query. We calculate the ratio of the elapsed time taken to answer VQ1i to that taken to propagate update U1 (in Figure 3.a) to `viewSel_`i. The results are plotted in Figure 6.a.

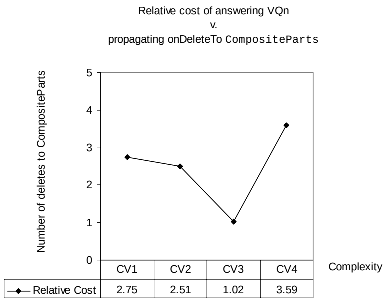
They reveal that, for the view with selectivity of 0.2, two updates can be propagated incrementally for the cost of evaluating a query against the corresponding virtual view. With an increase in the selectivity that ratio reduces to one, suggesting that, for views with high selectivity, if updates are at all frequent, incremental maintenance may not be beneficial.

In summary, the benefits of IVM seem to be greater when views have lower selectivities. This result conforms with general expectations, in that the materialized views with low selectivities involve smaller data sets than those associated with the corresponding virtual views. Space has prevented the inclusion of selectivity experiments on more complex views.



	0.2	0.4	0.6	0.8	1.0
◆ Relative Cost	2.10	1.47	1.23	1.07	1.02

(a) Relative Cost For Expt 1



	CV1	CV2	CV3	CV4
◆ Relative Cost	2.75	2.51	1.02	3.59

(b) Relative Cost For Expt 2

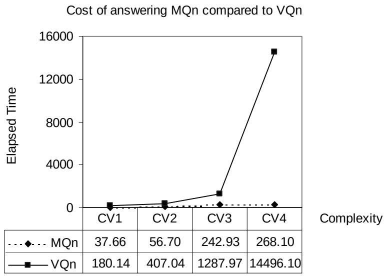
Fig. 6. Relative Query and Update Costs in Experiments 1 and 2

4.2 Experiment 2: Varying Structural Complexity

This experiment seeks to show how the complexity of the views may affect the performance of IVM. Again, we break the experiment into parts, each one corresponding to one of the hypotheses under scrutiny. In experiments involving complexity, the database used is `db3`.

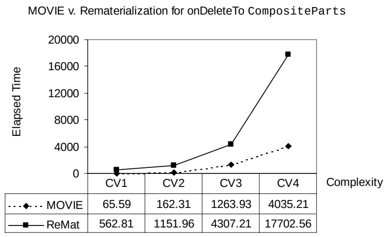
Query Answering MQn and VQn. The elapsed times taken to answer MQn and VQn, $2 \leq n \leq 5$ are measured and compared. The results are plotted in Figure 7.a.

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



	CV1	CV2	CV3	CV4
MQn	37.66	56.70	242.93	268.10
VQn	180.14	407.04	1287.97	14496.10

(a) Query Answering



	CV1	CV2	CV3	CV4
MOVIE	65.59	162.31	1263.93	4035.21
ReMat	562.81	1151.96	4307.21	17702.56

(b) View Maintenance

Fig. 7. Results of Experiment 2: Varying Structural Complexity

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 λ-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 materialisation.

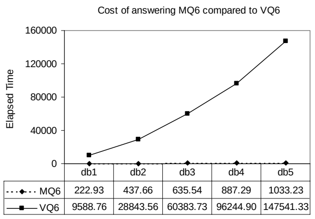
View Maintenance: Propagating U1 to complexView$_i$. We measure the elapsed time taken to propagate update U1 (in Figure 3.a) using MOVIE and using rematerialization. The results are plotted in Figure 7.b.

Update U1 (i.e., deletion of an instance of compositeParts) was chosen because it affects all of complexView1 to complexView4. The results reveal that, for complexView1 and complexView2, incremental maintenance is 9 and 7 times faster than rematerialization. However, for complexView3, MOVIE is only 3 times faster. The decrease is due to the fact that U1 requires the deletion of 100 objects from complexView3 (as against 20 for complexView2 and 1 for complexView1).

The elapsed time of rematerializing complexView4 is 4 times more than that for complexView3. Similarly, MOVIE takes almost three times as long to propagate changes to complexView4 as it takes in the case of complexView3. The bottom line is that the more complex the view the more expensive it is to maintain, no matter how it is being maintained.

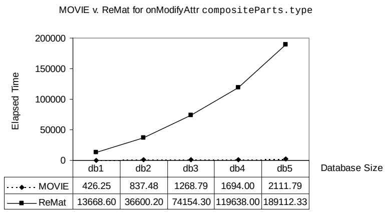
Relative Cost (Query/Update): Answering VQn, $2 \leq n \leq 5$, v. Propagating U1. This part of the experiment tries to identify the conditions under which incrementally maintaining MVs is beneficial by investigating how many updates can be propagated incrementally to an MV in the time it takes to answer the corresponding virtual query. In this experiment, we take the ratio of the elapsed time taken to answer complexViewVir1 to complexViewVir4 to that taken to propagate update U1 (in Figure 3.a) to all of `complexView1` to `complexView4`. The results are plotted in Figure 6.b. The results show that only quite a small number of updates (1 to 4) can be propagated in the time taken to evaluate the virtual view in this experiment. This can be seen as quite a disappointing result for incremental maintenance. The shape of the graph in Figure 6.b should not be considered to be of general significance. The "outlier" appears to be the propagation of U1 to `complexView3`. U1 can only be propagated to `complexView3` once in the time it takes to evaluate the virtual view. This is principally because of the high cost of incrementally propagating U1 to `complexView3`, which in turn is because a single occurrence of U1 causes 100 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 Figure 7.a, is the importance of the cardinality of the result to the incremental maintenance process.



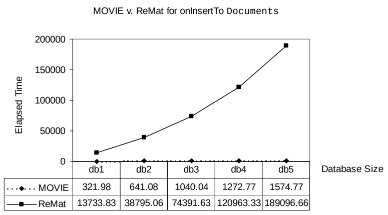
	db1	db2	db3	db4	db5
MQ6	222.93	437.66	635.54	887.29	1033.23
VQ6	9588.76	28843.56	60383.73	96244.90	147541.33

(a) Query Answering



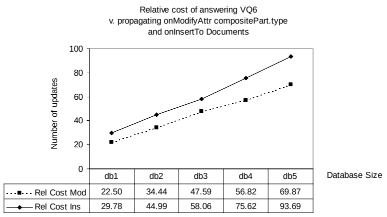
	db1	db2	db3	db4	db5
MOVIE	426.25	837.48	1268.79	1694.00	2111.79
ReMat	13668.60	36600.20	74154.30	119638.00	189112.33

(b) View Maintenance (1)



	db1	db2	db3	db4	db5
MOVIE	321.98	641.08	1040.04	1272.77	1574.77
ReMat	13733.83	38795.06	74391.63	120963.33	189096.66

(c) View Maintenance (2)



	db1	db2	db3	db4	db5
Rel Cost Mod	22.50	34.44	47.59	56.82	69.87
Rel Cost Ins	29.78	44.99	58.06	75.62	93.69

(d) Relative Cost (Query/Update)

Fig. 8. Results of Experiment 3: Varying Database Size

4.3 Experiment 3: Varying Database Size

This experiment explores the impact of database size on the performance of IVM. Once again, we break the experiment into parts that correspond to the hypotheses we are testing.

Query Answering: MQ6 and VQ6. Here the elapsed times taken to answer MQ6 and VQ6 are measured and compared. The results are plotted in Figure 8.a.

In the case of MQ6, the elapsed time taken to answer `dbSizeView` increases only slightly with an increase in the database size, reflecting the increase in the size of the materialized view. 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 43 times faster to use the MV, whereas for the largest database, it is 143 times faster, because the join cost grows non-linearly.

View Maintenance: Propagating U2 and U3 to `dbSizeView`. We measure the elapsed times taken to propagate two different updates: U2 (in Figure 3.b) and U3 (in Figure 3.c), which require maintenance of `dbSizeView`, using MOVIE and using rematerialization. The results are plotted in Figures 8.b and 8.c.

Figure 8.b shows MOVIE outperforming rematerialization 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.

Figure 8.c 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 rematerizalization. In fact, comparing the times in Figure 8.b and Figure 8.c, planting the delta works out faster than joining with the materialized OIDs in this case. Once again, the benefits of MOVIE increase with an increase in the database size. For example, in the smallest database, it performs 43 times better, whereas in the largest, it is 120 times faster than rematerizalization.

Relative Cost (Query/Update): Answering VQ6 v. Propagating U2 and U3. This part of the experiment tries to identify the conditions under which incrementally maintaining MVs is beneficial. We do so by investigating how many updates can be propagated incrementally to an MV in the time taken to answer the corresponding virtual query. We calculate the ratio of the elapsed time taken to answer VQ6 to that taken to propagate updates U2 and U3 (in Figure 3.b and 3.c, respectively) to `dbSizeView`. The results are plotted in Figure 8.d.

They reveal that, for the smallest database, 22 attribute modifications and 30 insertions can be propagated incrementally for the cost of evaluating a query against the corresponding virtual view. 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 rematerizalization. The benefits for both our small and large databases indicate that a significant number of update events can be carried out for the cost of answering the query over a non-materialized view.

5 Conclusions

The recent increase in research on identifying which views should be materialized indicates the importance of studies into the performance of incremental maintenance techniques. The most comprehensive study we can find of the performance of IVM is that of Hanson [8], which is an analytical model of immediate and deferred update propagation in relational databases. We are not aware of any research that has sought to validate the models presented by Hanson experimentally, or that has systematically assessed the performance of view maintenance techniques in relational (or object-oriented) systems. Although analytical models have been proposed in an object-oriented setting (e.g., [9]), we 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, [9] provides a few illustrative examples of the performance of MultiView, but could not be said to have carried out a systematic study.

We believe, therefore, that this paper presents the most thorough experimental performance evaluation of materialized views to date. Experiments have been conducted that assess the performance of query answering and update propagation for different selectivities, query complexities and database sizes. It is clear from the experiments, as would be expected, that queries over materialized views generally run very much faster than those over virtual views. This observation, however, is not what has driven research in IVM. Research in IVM is predicated on two hypothesis:

1. That incremental propagation of updates can be expected to perform significantly better than rematerialization.
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 rematerialization. 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). We would like to thank Leonidas Fegaras for making available to the research community the λ-DB system and for helping us make use of it.

References

1. M. A. Ali, A. A. A. Fernandes, and N. W. Paton. Incremental Maintenance of Materialized OQL Views. In *Proc. 3rd DOLAP*, pages 41–48. ACM Press, 2000.
2. M. Carey, D.J. DeWitt, and J.F. Naughton. The OO7 Benchmark. In *Proc. SIGMOD*, pages 12–21, 1993.
3. R. G. G. Cattell, editor. *The Object Database Standard: ODMG 3.0.* Morgan Kaufmann, 2000.
4. G. K. Y. Chan, Q. Li, and L. Feng. Design and Selection of Materialized Views in a Data Warehousing Environment: A Case Study. In *Proc. DOLAP*, pages 42–47, 1999.
5. S. Cluet and C. Delobel. A General Framework for the Optimization of Object-Oriented Queries. In *Proc. SIGMOD*, pages 383–392, 1992.
6. L. Fegaras and D. Maier. Optimizing Object Queries Using an Effective Calculus. *ACM TODS*, 25(4), 2000.
7. A. Gupta and I. S. Mumick, editors. *Materialized Views: Techniques, Implementations, and Applications.* The MIT Press, 1999. ISBN 0-262-57122-6.
8. Eric N. Hanson. A Performance Analysis of View Materialization Strategies. In ACM Press, editor, *Proc. SIGMOD*, pages 440–453, 1987.
9. H. Kuno and E. Rudensteiner. Incremental Maintenance of Materialized Object-Oriented Views in MultiView: Strategies and Performance Evaluation. *IEEE TKDE*, 10(5):768–792, 1998.
10. C. Zhang and J. Yang. Genetic Algorithm for Materialized View Selection in Data Warehouse Environments. In *Proc. DaWaK*, pages 116–125, 1999.