Incremental Updates for Materialized OQL Views

Dieter Gluche, Torsten Grust, Christof Mainberger, and Marc H. Scholl

University of Konstanz
Department of Mathematics and Computer Science
D-78457 Konstanz, Germany
e-mail: ⟨*firstname*⟩.⟨*lastname*⟩`@uni-konstanz.de`

Abstract. This work discusses the CROQUE approach to the maintenance problem for materialized views. In a CROQUE database, application-specified collections (type extents or classes) themselves need not be materialized. In exchange, the system maintains (redundant) views of the application data that help to minimize query response time. We understand views as functions of database objects and examine algebraic properties of these functions, in particular linearity, to derive incremental update plans. It turns out that it is feasible to employ ODMG OQL as a view definition language – instead of inventing a specialized one – in such an environment, since the majority of its clauses represent linear functions.

1 Introduction and Motivation

Whether a view can be maintained incrementally depends on the types of the involved database objects, their update operations, and the properties of the functions that the view computation process applies to these objects.

The view maintenance problem has been studied in different contexts. It is currently investigated in data warehousing applications, for example. In our case, it arised as a topic in the physical database design approach taken by the CROQUE project[1]. The overall objective of CROQUE is to implement an ODMG-style object database engine with a flexible storage manager and a corresponding query and update transformer and optimizer.

In CROQUE, the physical database level is solely made up of materialized views of the conceptual application data. During the lifetime of the database, it is the system's task to maintain the materialized views of the application data. The physical database designer has the freedom to (redundantly) materialize those parts of the database which are most frequently accessed in a given query mix. Additional views may be created upon changes of the transaction load. Likewise, views may be dropped. Obviously, this freedom is constrained: it must be guaranteed that the complete application data may be reassembled from the

[1] CROQUE is a joint research project of the Universities of Rostock and Konstanz, funded by the German Research Association, DFG.

materialized views at any time. Cut short, CROQUE pursues an implementation of the old ANSI/X3/SPARC 3-schema architecture: a clean separation between logical (the conceptual application data) and physcial (the materialized views) database structures [1].

This approach to physical design may very well have the consequence that application-specified collections (e.g. type extents or classes) themselves are *not* materialized in the database. In exchange, the system maintains various views of the application data that are tailored to minimize query response time.

Updates as well as queries, however, are specified with respect to the application base data, since, in general, users need not (actually: should not) be aware of the views the system maintains for them (physical data independence). It would be clearly inefficient to reassemble the application data from the materialized views (which may be a costly operation), carry out the user update on this transient representation of the base data, and then make the updates persistent by refreshing all affected views. This has been called the *recomputation approach* to materialized views maintenance.

The present paper discusses efficient methods for the *incremental* update of materialized views *without* the need to access the base data the views originated from. Its results make CROQUE's materialized view approach to physical database design feasible.

The problem of deciding on optimal storage structures (based on materialized views) when facing a given transaction load is an interesting and non-trivial optimization problem, but is not the focus of this paper.

This work provides a framework in which view definitions and view updates are understood as mappings between algebraic domains that represent collection type constructors such as sets, bags, and lists. The algebraic properties of these mappings, in particular *linearity*, are then used to generate incremental update plans.

Unlike in other approaches, incremental update operations are not restricted to collection types. Updates to primitive types (adding to/subtracting from an integer) are treated uniformly. The approach is extensible in the sense that user-defined types and functions that fulfill certain algebraic properties can be uniformly incorporated; it is natural in the sense that the results scale down to the well-known view maintenance rules for (nested) relational algebras, if we restrict types and functions to (nested) sets or bags of tuples and the standard relational algebra operators, respectively.

We proceed by shortly reviewing related previous work and then introduce our understanding of database types and their inherent update operations in Sect. 3. This leads to the derivation of the linearity requirement in Sect. 4, which the materialized functions have to satisfy, if we want to propagate updates incrementally. In Sect. 5, we then back up the claim that this requirement is not as rigorous as it may seem: todays object query languages (OQL, in particular) are appropriate candidates for view materialization languages that allow for incremental updates. Section 6 finally completes the picture by extending the previous discussion to incremental deletions, making this work a comprehensive approach to the view update problem. We conclude in Sect. 7.

2 Related Work

This work was on one hand strongly influenced by recent work on the analysis of view definition languages, and on the other hand by the various efforts that investigated sophisticated view maintenance techniques.

In earlier work [17,16] we examined various degrees of freedom for physical design for nested relational data models.

An algorithm that recomputes relational σ-π-$\bowtie$-views only partially is proposed in [3]. Conditions are given to detect irrelevant updates. However, the work does not consider queries beyond σ-π-$\bowtie$-queries. Gupta, Mumick, and Subrahmanian showed in [14,13] how to maintain views defined using a deductive query language. An incremental view maintenance algorithm was proposed for non-recursive views, that models insertions and deletions to bags by a multiplicity counting mechanism. [12] examines how to maintain outer join and σ-π-$\bowtie$ views without accessing the base relations. Neither additional collection constructors, nor further update operations are considered.

Griffin and Libkin [9] present a framework for incremental view maintenance, minimizing the update costs by computing minimal differences between the materialized data and the data to be added resp. deleted. However, the authors derive view update plans that presuppose the presence of the base relations from which the view was derived. In an environment like CROQUE, where these base data has to be reassembled (probably at a high cost) first, such update strategies are not applicable without fundamental adaption.

Quass [15] extended work done by Griffin and Libkin in [9], by considering a number of SQL aggregation functions (e.g. `count`, `max`). Quass' work presents transformation rules that allow for update propagation in presence of this particular set of aggregation functions.

In contrast to these, we do not restrict the view definition language but rather examine algebraic properties of the functions to be materialized. The various update operations that are sensible for a variety of database types are treated uniformly in our work.

The model of database types as well as the corresponding query language we adopt, has been affected by recent work on generic calculi and algebras for collection types. This branch of research started out with the adoption of list comprehensions as a database query language [18] and has been generalized to arbitrary monoids more recently [4,7,8,10].

3 Updating Materialized Views

A materialized view in the context of this work is to be understood as a precomputed, physically stored query, whose results we may reference by a name we assign to it, say v.

Definition 1 (Materialized View). Let $f(\overline{v})$ be an expression computing a value of type τ, then $v \leftarrowtail f(\overline{v})$ denotes the ***materialized view*** v; $\overline{v}$ denotes the vector of database variables (e.g. entry points) that f references in order to compute v.

The field of applications where one can benefit from materialized query results is wide. Query optimizers may detect that (sub-)queries need not be evaluated but can be replaced by comparably cheap accesses to an already precomputed view. Resources that are not randomly available, e.g. off-line or costly data sources, may be accessed once and the resulting data may then be materialized for further reference. In general, we gain the chance to spend query evaluation efforts (for view materialization) when we have the time and sources handy, but save resources during subsequent references to the materialized view.

However, user updates alter base data that a materialized query $v \leftarrowtail f(\overline{v})$ might reference. It is the focus of this paper to derive efficient methods that allow v to reflect these changes without recomputing and materializing $f(\overline{v})$ from scratch (which might be impossible due to lack of resources anyway, and possibly involves the costly reconstruction of base data). Changing a variable's value in $\overline{v}$ leaves us with the problem of maintaining v, too. Function f above is not constrained to be an expression of some specific view definition language, but may be an arbitrary program written in the query language of the database system, which we introduce in Sect. 5.

Updates to variables in $\overline{v}$ may be propagated by recomputing v. If $\mathit{upd}(\overline{v})$ denotes the base data's state after an update, $v := f(\mathit{upd}(\overline{v}))$ expresses just this naive recomputation.

On the other hand, recomputation is often unnecessary and wasted effort. Since updates to base data are specified in an *incremental* manner, which is especially true for bulk data types (e.g. through `insert into` or `delete from` clauses), there is the chance to propagate just these incremental changes to the materialized queries and keep them up to date without the need to reassemble $\overline{v}$ first. An incremental update refers to the status quo of the data to be altered in contrast to overwriting. The data types themselves implicitly provide the meaningful incremental update operations we can apply. Insertion and deletion of elements is the appropriate operation for bulk types such as *sets* and *bags* (*multi-sets*), while appending or removing prefixes and suffixes applies well to *lists*.

Work on query evaluation issues in CROQUE [10] already adopted a view of database types and values which reflects the nature of complex values being incrementally built from simpler values using the type's inherent operations. For example, any finite set may be constructed by using the empty set as a start, and then adding singleton sets containing the elements needed in a step by step manner using set union: $\{1, 2, 3\} = \{\} \cup \{1\} \cup \{2\} \cup \{3\}$. While the order does not matter for sets, lists demand special attention in this case.

Recently, work on query language expressiveness and optimization has adopted the notion of (collection) *monoids* to model this constructive approach to database types [4,6,5].

Definition 2 (Monoid). Let τ be a data type, $merge[\mathcal{M}] :: \tau \times \tau \to \tau$ a binary function, and $zero[\mathcal{M}]$ a distinguished element of type τ (read the symbol :: as *type of*). The triple $\mathcal{M} = (\tau, merge[\mathcal{M}], zero[\mathcal{M}])$ is called a ***monoid*** if $merge[\mathcal{M}]$ is associative with (left and right) identity $zero[\mathcal{M}]$.

In addition, we call a monoid $\mathcal{M}$ commutative resp. idempotent, whenever $merge[\mathcal{M}]$ is. This is already sufficient to capture primitive data types and typical query operations on these. Consider the monoid $sum = (int, +, 0)$ which can express the aggregation operation *sum* of many query languages. Any finite sum is made up of succesive applications of $+$ to two elements of type *int*. We see $+$ as a natural incremental update operation on integers. Further examples of monoids are $max = (int, \max_2, -\infty)$ and $some = (bool, \vee, false)$.

Definition 3 (Collection Monoid). Let C be a collection type constructor, $\mathcal{C} = (C\langle\tau\rangle,\ merge[\mathcal{C}],\ zero[\mathcal{C}])$ be a monoid, and $unit[\mathcal{C}] :: \tau \to C\langle\tau\rangle$ a function that delivers a singleton collection. $\mathcal{C} = (C\langle\tau\rangle, merge[\mathcal{C}], zero[\mathcal{C}], unit[\mathcal{C}])$ is called a ***collection monoid***.

For convenience, first assume $C = set$. We obtain the set monoid $(set\langle\tau\rangle, \cup, \{\}, \lambda e.\{e\})$, i.e. exactly the algebraic structure we will use to reason about set values and incremental changes to these values (insertions via $\cup$ in this case). The bag $(bag\langle\tau\rangle, \uplus, \{\!\{\}\!\}, \lambda e.\{\!\{e\}\!\})$ and list monoids $(list\langle\tau\rangle, +\!\!+, [], \lambda e.[e])$, are derived in just the same manner ($\uplus$ and $+\!\!+$ denote the additive bag union and list concatenation, respectively). Note, that τ itself may be an arbitrary type as well, which gives rise to a data model similar to complex object models known today. In what follows we will treat monoids like database types, if this does not lead to confusion.

The monoid approach treats primitive and bulk types uniformly (consider the *unit* function of monoids over atomic types to be the identity). We will benefit from this uniformity in the sequel, making the presentation of the approach concise and extensible in the sense that any type constructor that can be understood as a monoid can be incorporated right away.

In the light of the above definitions the incremental view maintenance problem now has the application of *merge* operations, i.e. constructive user updates, to base data as its starting point. We extend this approach to handle destructive updates after we have provided the basic ideas. More generally, a user update request $\overline{v} := upd(\overline{v})$ is now specified by means of *merge* operations applied to $\overline{v}$. Since we know the properties of *merge*, we may derive characteristics (e.g. idempotence) of *upd* as well. If it would be feasible to put down the computation carried out by f – the view's defining function – in the same manner, i.e. we describe f's effect on the level of monoid element manipulations, we could also make assertions about $f \circ upd$.

In the sequel, we discuss the very basic property f has to satisfy so that the update's effects can be applied to the view's status quo, $f(\overline{v})$, instead of initiating recomputation, i.e. applying $f \circ upd$ to $\overline{v}$. We will then show that query languages that basically describe monoid mappings are as powerful as today's object query languages, say ODMG's OQL, and – more important in this context – possess the just mentioned property. Without losing the ability to derive incremental update strategies, a complete query language can therefore serve as a language in which we may define materialized views.

4 Monoid Homomorphisms

For the sake of readability, let the infix operators $\oplus$ and $\otimes$ denote the *merge* functions of two (not necessarily different) monoids, $\mathcal{M} = (\tau_{\mathcal{M}}, \oplus, \mathit{zero}[\mathcal{M}])$ and $\mathcal{N} = (\tau_{\mathcal{N}}, \otimes, \mathit{zero}[\mathcal{N}])$, respectively. Additionally, let $v \leftarrowtail f(C)$ be a materialized view of type $\tau_{\mathcal{N}}$ that refers to a collection $C :: \tau_{\mathcal{M}}$. Consequently, f is of type $\tau_{\mathcal{M}} \to \tau_{\mathcal{N}}$.

Suppose a user update alters the value of C. The complete update specification then is $C := \mathit{merge}[\mathcal{M}](C, \triangle C) = C \oplus \triangle C$, if $\triangle C$ denotes the data to "*merge*" with C. Merging is type-dependent and may actually mean the insertion of elements into a set (if $\tau_{\mathcal{M}} = \mathit{set}\langle\tau\rangle$) or simply incrementing an integer (if $\mathcal{M} = \mathit{sum}$).

View v's value after the update is given by $f(C \oplus \triangle C)$, which performs the *merge* first and then recomputes the materialization's result. An incremental update plan would take advantage of the effort spent in materializing v up to now, that is, use $f(C)$. Assume we could assert a fundamental algebraic property of f, namely being a *homomorphism*:

Definition 4 (Monoid Homomorphism). Let both $\mathcal{M} = (\tau_{\mathcal{M}}, \oplus, \mathit{zero}[\mathcal{M}])$ and $\mathcal{N} = (\tau_{\mathcal{N}}, \otimes, \mathit{zero}[\mathcal{N}])$ denote monoids, and $f :: \tau_{\mathcal{M}} \to \tau_{\mathcal{N}}$ a function. f is a ***monoid homomorphism from*** $\mathcal{M}$ ***to*** $\mathcal{N}$, denoted $f \in \hom^{\mathcal{M}\to\mathcal{N}}$, iff

$$\begin{aligned} f(\mathit{zero}[\mathcal{M}]) &= \mathit{zero}[\mathcal{N}] \\ f(e_1 \oplus e_2) &= f(e_1) \otimes f(e_2) \ . \end{aligned}$$

Then we have

$$f(C \oplus \triangle C) = f(C) \otimes f(\triangle C)$$

which immediately gives rise to an incremental update plan as in

$$\begin{aligned} v &:= f(C \oplus \triangle C) \\ &= f(C) \otimes f(\triangle C) \\ &= v \otimes f(\triangle C) \ . \end{aligned}$$

Since the size of the increment $\triangle C$ is typically small compared to C (assuming $\mathcal{M}$ to be a collection monoid), recomputation will dominate the cost for the application of f to just $\triangle C$ by far. Observe, that the value of C needs not to be known. The property of f being a homomorphism (or equivalently, being *linear* in its parameter) is the key to incremental update propagation.

Example 5. Consider the materialized view $v \hookleftarrow \texttt{count}(C)$, where C is of type $bag\langle\tau\rangle$ for some τ and `count` denotes the OQL counting function, which we model using the *sum* monoid mentioned earlier. The update inserts the elements e_1, e_2 into C, so we have $\triangle C = \{\!\{e_1, e_2\}\!\}$. The update would then proceed as follows:

$$\begin{aligned} v &:= \texttt{count}(C \uplus \{\!\{e_1, e_2\}\!\}) \\ &= \texttt{count}(C) + \texttt{count}(\{\!\{e_1, e_2\}\!\}) \\ &= v + 2\ . \end{aligned}$$

Propagating the update to v simply results in an addition of an integer. The above transformation is correct, since `count` is a homomorphism from the monoid *bag* to *int*. □

To be strict, the above example already applied a sequence of two updates to C, namely $(C \uplus \{\!\{e_1\}\!\}) \uplus \{\!\{e_2\}\!\}$. This poses no problem thanks to the homomorphic nature of f (`count` in this case), since we have $f((C \uplus \{\!\{e_1\}\!\}) \uplus \{\!\{e_2\}\!\}) = f(C) + f(\{\!\{e_1\}\!\}) + f(\{\!\{e_2\}\!\})$. There is no real difference in discussing single element or bulk updates.

Example 6. Computing the bounding box of a graphical object is a common task of geographical information systems. The bounding box operation is a homomorphism with respect to the combination of two objects into one. Consider the materalized view $v \hookleftarrow \texttt{bbox}(obj_1)$. The update $obj_1 := \texttt{combine}(obj_1, obj_2)$ can be propagated incrementally into v:

$$\begin{aligned} v &:= \texttt{bbox}(\texttt{combine}(obj_1, obj_2)) \\ &= \texttt{bbox}_2(\texttt{bbox}(obj_1), \texttt{bbox}(obj_2)) \\ &= \texttt{bbox}_2(v, \texttt{bbox}(obj_2)) \end{aligned}$$

where $\texttt{bbox}_2$ computes the bounding box of two boxes, which is efficiently implementable. Note that $\texttt{bbox}_2$ is idempotent which has impacts if we consider deletions of graphical objects as a valid update operation (cf. Sect. 6). □

What about view definitions depending on more than one database variable at the same time? The just mentioned considerations remain valid as long as f is linear in the parameters that are affected by an update. We do not have to impose restrictions on f's behaviour with respect to unaffected parameters. This weakens our demands on f being a homomorphism; being linear (in some parameters) is obviously sufficient. Let us take down the following.

Definition 7 (Linearity). As usual, let $\mathcal{M} = (\tau_\mathcal{M}, \oplus, zero[\mathcal{M}])$ and $\mathcal{N} = (\tau_\mathcal{N}, \otimes, zero[\mathcal{N}])$ denote monoids, and $f :: \tau_1 \times \ldots \times \tau_{i-1} \times \tau_\mathcal{M} \times \tau_{i+1} \times \ldots \times \tau_n \to \tau_\mathcal{N}$ a function. f is ***linear in its i-th parameter***, denoted $f \in lin_i^{\mathcal{M}\to\mathcal{N}}$, iff

$$\begin{aligned} f(\ldots, v_{i-1}, zero[\mathcal{M}], v_{i+1}, \ldots) &= zero[\mathcal{N}] \\ f(\ldots, v_{i-1}, x \oplus y, v_{i+1}, \ldots) &= f(\ldots, v_{i-1}, x, v_{i+1}, \ldots) \otimes \\ &\quad\ f(\ldots, v_{i-1}, y, v_{i+1}, \ldots)\ . \end{aligned}$$

For $n = 1$, $lin_1^{\mathcal{M} \to \mathcal{N}}$ is just the set of homomorphic mappings from $\mathcal{M}$ to $\mathcal{N}$, $\hom^{\mathcal{M} \to \mathcal{N}}$. In what follows, we may view some non-linear functions like, e.g., the self join $f(E) = E \bowtie E$ as $f(E_1, E_2) = E_1 \bowtie E_2$ for which we have $f \in lin_1^{\mathcal{M} \to \mathcal{N}}$ and $f \in lin_2^{\mathcal{M} \to \mathcal{N}}$.

Well-known view maintenance results of approaches [2,14] that use relational algebras as their view definition languages are covered by the above: expressions of the relational algebra act as homomorphic mappings from the monoid of sets (or bags) of tuples into the very same monoid. Consider the following:

Example 8. Let f define a typical σ-π-$\bowtie$-view as in $v \leftarrow\!\!\prec f(C_1, C_2, a, p, q) = \pi_a \sigma_p (C_1 \bowtie_q C_2)$, where the C_i are of types $bag\langle \tau_i \rangle$, respectively. p and q denote predicates, while a is the attribute list of the final projection. f is linear in C_1 and C_2, since $\bowtie$, σ, π are linear with respect to their collection-typed arguments, and a composition of linear functions is linear as well. This makes f an element of $lin_1^{bag \to bag}$ and $lin_2^{bag \to bag}$.

We issue the updates $C_1 := C_1 \uplus \triangle C_1$ and $C_2 := C_2 \uplus \triangle C_2$, and have

$$\begin{aligned}
v &:= \pi_a \sigma_p ((C_1 \uplus \triangle C_1) \bowtie_q (C_2 \uplus \triangle C_2)) \\
&= \pi_a \sigma_p (C_1 \bowtie_q C_2) \uplus \pi_a \sigma_p (\triangle C_1 \bowtie_q C_2) \uplus \pi_a \sigma_p (C_1 \bowtie_q \triangle C_2) \uplus \pi_a \sigma_p (\triangle C_1 \bowtie_q \triangle C_2) \\
&= v \\
&\quad \uplus \pi_a \sigma_p (\triangle C_1 \bowtie_q C_2) \\
&\quad \uplus \pi_a \sigma_p ((C_1 \uplus \triangle C_1) \bowtie_q \triangle C_2) \quad .
\end{aligned}$$

□

We have just seen that the classical relational algebra operators fall into the class of linear monoid mappings. However, does the monoid homomorphism approach appropriately scale up to query languages for complex objects?

5 OQL as a Materialized View Definition Language

Remember that the physical level of a CROQUE database consists of a collection of queries against the conceptual schema. These materialized views make up the base data from which the system answers subsequent user queries.

The CROQUE project committed itself to use OQL as its materialized view definition language (MVDL) rather than inventing a specialized language for that purpose. CROQUE's benefits are twofold:

1. the process of mapping logical to physical queries is easily accomplished by view replacement (which may even happen on the OQL level), and
2. a large subset of OQL may be translated into linear mappings between monoids.

The latter makes OQL a suitable candidate for a language to define materialized views that may be incrementally updated in our model: for a view definition $v \leftarrow\!\!\prec f(\overline{v})$, we may code f using OQL. We shed some light on this now.

How can one see that a reasonable subset of OQL indeed defines linear mappings? [11,10] gave a translation of OQL into an algebra whose core is **fold**, a common iterator abstraction in functional programming languages. We borrow from this field and look at fold's definition in the monoid context. Let $\mathcal{M} = (\tau_{\mathcal{M}}, \oplus, \mathit{zero}[\mathcal{M}], \mathit{unit}[\mathcal{M}])$ and $\mathcal{N} = (\tau_{\mathcal{N}}, \otimes, \mathit{zero}[\mathcal{N}], \mathit{unit}[\mathcal{N}])$ be monoids, and $f :: \tau_{\mathcal{M}} \to \tau_{\mathcal{N}}$. The following three equations define fold:

$$\mathrm{fold}[\mathcal{N}; f](\mathit{zero}[\mathcal{M}]) = \mathit{zero}[\mathcal{N}] \tag{1}$$

$$\mathrm{fold}[\mathcal{N}; f](\mathit{unit}[\mathcal{M}](e)) = f(e) \tag{2}$$

$$\mathrm{fold}[\mathcal{N}; f](e_1 \oplus e_2) = \mathrm{fold}[\mathcal{N}; f](e_1) \otimes \mathrm{fold}[\mathcal{N}; f](e_2) \enspace . \tag{3}$$

In order to make the above well-defined, $\otimes$ has to be commutative resp. idempotent whenever $\oplus$ is. Note that $\mathrm{fold}[\mathcal{N}; f] \in \hom^{\mathcal{M} \to \mathcal{N}}$, as one can see from (1) and (3). Fold's recursion scheme is sufficiently general to be able to compute all linear OQL clauses that represent linear functions. Due to space constraints we can merely provide examples here. [11] gives a comprehensive treatment.

Suppose it is our task to sum the elements of the list $[1, 2, 3]$. We can do so by applying $\mathrm{fold}[\mathit{sum}; \mathrm{id}]$ to the list (remember that $\mathit{merge}[\mathit{sum}] = +$, and $\mathit{zero}[\mathit{sum}] = 0$):

$$\begin{aligned}
\mathrm{fold}[\mathit{sum}; \mathrm{id}]([1,2,3]) &= \mathrm{fold}[\mathit{sum}; \mathrm{id}]([1]) + \mathrm{fold}[\mathit{sum}; \mathrm{id}]([2,3]) \\
&= \mathrm{id}(1) + \mathrm{fold}[\mathit{sum}; \mathrm{id}]([2,3]) \\
&= \dots \\
&= 1 + 2 + 3 + \mathrm{fold}[\mathit{sum}; \mathrm{id}]([]) \\
&= 1 + 2 + 3 + 0 = 6 \enspace .
\end{aligned}$$

OQL's core, the `select` f `from` E `where` p block, computes a function linear in E. This may be seen as follows:

$$\texttt{select}\ f\ \texttt{from}\ E\ \texttt{where}\ p = \mathrm{fold}[\mathit{bag}; \lambda x.\mathrm{if}\ p(x)\ \mathrm{then}\ f(x)\ \mathrm{else}\ \mathit{zero}[\mathit{bag}]](E) \enspace .$$

A `from`-clause specifying more than one collection is computed by a nested fold expression. This poses no new problems in our context (keep in mind that we are neither after efficient execution plans for queries nor seek for a general decision procedure for linearity, but rather try to understand the linearity properties of OQL clauses).

The *some* and *all* monoids help to implement quantification as in

$$\texttt{exists}\ x\ \texttt{in}\ E\texttt{:}p = \mathrm{fold}[\mathit{some}; \lambda x.p(x)](E)$$

while *sum*, *max*, etc. allow us to express aggregation

$$\texttt{count(}E\texttt{)} = \mathrm{fold}[\mathit{sum}; \lambda x.1](E) \qquad \text{and} \qquad \texttt{max(}E\texttt{)} = \mathrm{fold}[\mathit{max}; \lambda x.x](E) \enspace .$$

Note that E in `count(`E`)` must not be of type $\mathit{set}\langle\tau\rangle$, since $\cup$ is idempotent while $\mathit{merge}[\mathit{sum}]$ is not, in other words $\texttt{count} \notin \hom^{\mathit{set} \to \mathit{sum}}$. This restriction is crucial and ignoring it may lead to nonsense like

$$1 = \texttt{count(}\{e\}\texttt{)} = \texttt{count(}\{e\} \cup \{e\}\texttt{)} = \texttt{count(}\{e\}\texttt{)} + \texttt{count(}\{e\}\texttt{)} = 1 + 1 = 2 \enspace .$$

Table 1. Linearity of an OQL-defined function f: $(+)$ linear in the specified parameter, $(-)$ non-linear, $(\cdot)$ not a valid OQL clause due to parameter type mismatch.

f	*set*	*bag*	*list*	f	*set*	*bag*	*list*
λE_i.`select` E `from` $E_1, E_2, \ldots$ `where` p	$-$	$+$	$+$	λE_i.`select distinct *` `from` $E_1, E_2, \ldots$ `where` p	$+$	$+$	$+$
λE.`exists` x `in` E:p	$+$	$+$	$+$	λE.`distinct(`E`)`	$\cdot$	$+$	$+$
λE.`forall` x `in` E:p	$+$	$+$	$+$	λE.`flatten(`E`)`	$+$	$+$	$+$
$\lambda E_2.E_1\,\theta$ `some` E_2	$+$	$+$	$+$	λE.`listtoset(`E`)`	$+$	$+$	$+$
$\lambda E_2.E_1\,\theta$ `all` E_2	$+$	$+$	$+$	λE.`element(`E`)`	$+$	$+$	$+$
$\lambda E_1.E_1$ `union` E_2	$+^*$	$+^*$	$\cdot$	λE.`count(`E`)`	$-$	$+$	$+$
$\lambda E_1.E_1$ `intersect` E_2	$+$	$-$	$\cdot$	λE.`sum(`E`)`	$-$	$+$	$+$
$\lambda E_1.E_1$ `except` E_2	$+$	$-$	$\cdot$	λE.`max(`E`)`	$+$	$+$	$+$
$\lambda E_1.E_1$ `+` E_2	$\cdot$	$\cdot$	$+^*$	λE.`min(`E`)`	$+$	$+$	$+$
$\lambda E_2.E_1$ `union` E_2	$+^*$	$+^*$	$\cdot$	λE.`avg(`E`)`	$-$	$+^*$	$+^*$
$\lambda E_2.E_1$ `intersect` E_2	$+$	$-$	$\cdot$	λE.`unique(`E`)`	$+^*$	$+^*$	$+^*$
$\lambda E_2.E_1$ `except` E_2	$+$	$-$	$\cdot$	λE.`exists(`E`)`	$+$	$+$	$+$
$\lambda E_2.E_1$ `+` E_2	$\cdot$	$\cdot$	$+^*$	$\lambda E.x$ `in` E	$+$	$+$	$+$

Table 1 lists OQL functions and marks those with a $+$ sign for which a fold translation similar to the above examples can be found. Any such function is part of the CROQUE MVDL. A view definition made up of (a composition of) these functions fulfills the linearity condition. Such views may be maintained incrementally, as the foregoing sections argued.

A mark of $+^*$ indicates cases where a function was decided to be part of the MVDL despite being non-linear. In the case of `avg(`E`)`, which is not linear in E, the system instead materializes the pair $\langle$`count(`E`)`, `sum(`E`)`$\rangle$ from which the average is immediate. Maintenance is then implemented using the monoid

$$avg = (int \times int, \lambda(\langle x_1, y_1\rangle, \langle x_2, y_2\rangle).\langle x_1 + x_2, y_1 + y_2\rangle, \langle 0, 0\rangle, \lambda e.\langle e, 1\rangle)$$

which keeps the pair up-to-date (computing the average of an empty collection results in undefined behaviour).

Similar remarks apply to `unique` as well as all cases marked with $-$: at the cost of maintaning additional support data structures (probably more complicated than the simple pair in the `avg` case) it is possible to integrate non-linear functions into CROQUE's update model.

Non-linear functions may thus be used in view definitions. However, the MVDL compiler will warn the physical designer that view updates may be costly due to support data structure maintenance.

6 Deletions

How are we supposed to cope with deletions/subtractions if the sole update primitive the monoid approach provides is the constructive *merge* operation?

In case the involved types are described as non-idempotent monoids, we can extend these monoids to **groups**. The construction we use in the following is similar to the well-known method of attaching the negative integers to the natural numbers: we do not add a "deletion" operation, e.g. a subtraction, but introduce the notion of the **inverse** of for any element.

We follow this idea here, mainly because the homomorphism theories of monoids and groups are closely related.

The approach introduces a new algebraic constructor for a monoid $\mathcal{M}$ (besides *zero*, *unit*, and *merge*), namely $\mathit{inv}[\mathcal{M}](e)$, that designates the inverse for any monoid element e. The property we expect of the inverse for any element e is natural: $\mathit{merge}[\mathcal{M}](e, \mathit{inv}[\mathcal{M}](e)) = \mathit{zero}[\mathcal{M}] = \mathit{merge}[\mathcal{M}](\mathit{inv}[\mathcal{M}](e), e)$.

Besides this property, what are inverse elements like? For any type, we extend the type's domain $\tau_{\mathcal{M}}$. Let us denote the extended domain by $\overline{\tau_{\mathcal{M}}}$. The inverse does *not* undergo any semantic denotation itself, but its sole interpretation is by its effect on a *merge* with other elements. This already satisfies our needs. There are cases, however, where $\tau_{\mathcal{M}} = \overline{\tau_{\mathcal{M}}}$, i.e. the original domain already contains the inverse; take $\tau_{\mathcal{M}} = \mathit{int}$ as an example. In these exceptional cases, the inverse actually has a semantically sensible interpretation, e.g. $\mathit{inv}[\mathit{sum}](e) = -e$ for integers.

Definition 9 (Group Extension of a Monoid). Let $\mathcal{M} = (\tau_{\mathcal{M}}, \oplus, \mathit{zero}[\mathcal{M}])$ be a non-idempotent monoid (see below). The ***group extension of*** $\mathcal{M}$ then is $\overline{\mathcal{M}} = (\overline{\tau_{\mathcal{M}}}, \oplus, \mathit{zero}[\mathcal{M}], \mathit{inv}[\mathcal{M}])$, where

$$\overline{\tau_{\mathcal{M}}} = \{\bigoplus_{i=1}^{n} e_i \mid e_i \in \tau_{\mathcal{M}} \vee e_i \in \{\mathit{inv}[\mathcal{M}](e) \mid e \in \tau_{\mathcal{M}}\}, n \in \mathbb{N}\} \quad .$$

The group's operation $\oplus$ now is of type $\overline{\tau_{\mathcal{M}}} \times \overline{\tau_{\mathcal{M}}} \to \overline{\tau_{\mathcal{M}}}$, but retains its original behaviour if its argument is in $\tau_{\mathcal{M}} \times \tau_{\mathcal{M}}$. It additionally holds, that for all $e \in \tau_{\mathcal{M}} : e \oplus \mathit{inv}[\mathcal{M}](e) = \mathit{zero}[\mathcal{M}] = \mathit{inv}[\mathcal{M}](e) \oplus e$.

The choice of $\mathit{inv}[\mathcal{M}]$ is not arbitrary in the sense that any extension of $\mathcal{M}$ to a group is homomorphic to the "natural embedding" of $\mathcal{M}$ into a group (sometimes called "solution to the universal problem").

A user delete request as in `delete` e `from` C, is now specified as the monoid operation $\mathit{merge}[\mathcal{M}](C, \mathit{inv}[\mathcal{M}](e))$ if $C{::}\tau_{\mathcal{M}}$. As in the constructive case, merging is type-dependent. For $\tau_{\mathcal{M}} = \mathit{int}$ the request would result in $C + (-e) = C - e$.

Example 10. The interpretation we choose for the expression $C \uplus \mathit{inv}[\mathit{bag}](\triangle C)$ is to remove $\triangle C$'s elements (according to their multiplicity) from C. That is $\{\!\{1, 2, 2, 3\}\!\} \uplus \mathit{inv}[\mathit{bag}](\{\!\{1, 2, 3\}\!\}) = \{\!\{2\}\!\}$. If the removal would be impossible, because $\triangle C$ contains some $e \notin C$, the expression $C \uplus \mathit{inv}[\mathit{bag}](\triangle C)$ undergoes no further interpretation; nevertheless it represents an element of $\overline{\tau_{\mathcal{M}}}$ ($\overline{\tau_{\mathcal{M}}} \setminus \tau_{\mathcal{M}}$ to be precise). We come back to this shortly.

For lists, remember that $+\!\!+$ is not commutative. The interpretation for $L +\!\!+ \mathit{inv}[\mathit{list}](\triangle L)$ and $\mathit{inv}[\mathit{list}](\triangle L) +\!\!+ L$ is to delete the suffix $\triangle L$ resp. the prefix

$\triangle L$ of L. Consider $[1,2,3] +\!\!+\ \mathit{inv}[\mathit{list}]([2,3]) = [1,2,3] +\!\!+\ \mathit{inv}[\mathit{list}]([2] +\!\!+ [3]) = [1,2,3] +\!\!+\ \mathit{inv}[\mathit{list}]([3]) +\!\!+\ \mathit{inv}[\mathit{list}]([2]) = [1]$. Again, if suffix/prefix removal is impossible, the expression is not interpreted any further. $\square$

The key advantage of this approach is that any monoid homomorphism actually defines a unique homomorphism of the corresponding groups, if we can group-embed the range and destination monoids (a simple algebraic fact). We have $f(\mathit{inv}[\mathcal{M}](e)) = \mathit{inv}[\mathcal{N}](f(e))$ for any homomorphic mapping $f :: \mathcal{M} \to \mathcal{N}$, in particular for $\text{fold}[\mathcal{M}; g](\mathit{inv}[\mathcal{M}](e)) = \mathit{inv}[\mathcal{N}](\text{fold}[\mathcal{M}; g](e))$ which completes fold's constructor-wise definition when applied to group elements. Thus, the considerations of the previous sections immediately apply to destructive updates, too.

Example 11. We request the deletion $L_2 +\!\!+\ \mathit{inv}[\mathit{list}]([42])$ while the database maintains the materialized view

$$\begin{aligned} v \hookleftarrow f(L_1, L_2, p) = \texttt{sum(select}\ & e_1 \\ \texttt{from}\ & L_1\ \texttt{as}\ e_1\texttt{,}\ L_2\ \texttt{as}\ e_2 \\ \texttt{where}\ & p(e_1, e_2)\texttt{)} \quad . \end{aligned}$$

The L_i are of type $\mathit{list}\langle\mathit{int}\rangle$, p acts as a filter predicate.

v materializes function

$$f(L_1, L_2, p) = \text{fold}[\mathit{sum}; \lambda e_1.\text{fold}[\mathit{sum}; \lambda e_2.\text{if } p(e_1, e_2) \text{ then } e_1 \text{ else } 0](L_2)](L_1) \quad .$$

Following the considerations of Sect. 4, the update may be propagated, since f is linear in its first two arguments:

$$\begin{aligned} v :=\ & \text{fold}[\mathit{sum}; \lambda e_1.\text{fold}[\mathit{sum}; \lambda e_2.\text{if } p(e_1, e_2) \text{ then } e_1 \text{ else } 0](\ L_2 +\!\!+ \\ & \qquad \mathit{inv}[\mathit{list}]([42]))](L_1) \\ =\ & \text{fold}[\mathit{sum}; \lambda e_1.\text{fold}[\mathit{sum}; \lambda e_2.\text{if } p(e_1, e_2) \text{ then } e_1 \text{ else } 0](L_2) + \\ & \qquad \text{fold}[\mathit{sum}; \lambda e_2.\text{if } p(e_1, e_2) \text{ then } e_1 \text{ else } 0](\mathit{inv}[\mathit{list}]([42]))](L_1) \\ =\ & \text{fold}[\mathit{sum}; \lambda e_1.\text{fold}[\mathit{sum}; \lambda e_2.\text{if } p(e_1, e_2) \text{ then } e_1 \text{ else } 0](L_2) + \\ & \qquad \mathit{inv}[\mathit{sum}]((\lambda e.\text{if } p(e_1, e) \text{ then } e_1 \text{ else } 0)(42))](L_1) \\ (*) \ =\ & \text{fold}[\mathit{sum}; \lambda e_1.\text{fold}[\mathit{sum}; \lambda e_2.\text{if } p(e_1, e_2) \text{ then } e_1 \text{ else } 0](L_2)](L_1) + \\ & \text{fold}[\mathit{sum}; \lambda e_1.-(\lambda e.\text{if } p(e_1, e) \text{ then } e_1 \text{ else } 0)(42)](L_1) \\ =\ & v + \texttt{sum(select}\ -e_1 \\ & \qquad \texttt{from}\ L_1\ \texttt{as}\ e_1 \\ & \qquad \texttt{where}\ p(e_1, 42)\texttt{)} \quad . \end{aligned}$$

The transformation at $(*)$ is correct, because we have

$$\text{fold}[\mathcal{M}; \lambda e.(f_1 \oplus f_2)](E) = \text{fold}[\mathcal{M}; \lambda e.f_1](E) \oplus \text{fold}[\mathcal{M}; \lambda e.f_2](E)$$

for any commutative $\mathit{merge}[\mathcal{M}] = \oplus$. $\square$

We have mentioned in the beginning of this section, that $\mathit{inv}[\mathcal{M}](e)$ itself has no sensible interpretation for some $\mathcal{M}$, e.g. bags or lists. How does one treat view updates that result in expressions containing $\mathit{inv}[\mathcal{M}]$ constructors one cannot get rid of (by means of "normalizing" the resulting expression according to the associativity and possibly commutativity of $\mathit{merge}[\mathcal{M}]$)? Expressions of this kind are the result of updates that tried to delete elements that were not present before the update, respectively tried to remove inexistent suffixes/prefixes (cf. Example 10). Thus, one option is to ignore such updates at all. An alternative way to go about this is to provide an update clause like

$$\texttt{delete from } C \texttt{ where } p$$

at the user-level. This specifies a $\triangle C$, $\triangle C \subseteq C$, which can be removed from C without problems.

Why is it not possible to use the same approach if idempotent monoids, e.g. *set*, *max*, *min*, are involved? Again, the answer is a simple algebraic fact: any idempotent group with a *merge* operation is the trivial group which contains *zero* as its single element, because we have

$$e = e \oplus (e \oplus \mathit{inv}[\mathcal{M}](e)) = (e \oplus e) \oplus \mathit{inv}[\mathcal{M}](e) \underset{\substack{\oplus \\ \text{idempotent}}}{=} e \oplus \mathit{inv}[\mathcal{M}](e) = \mathit{zero}[\mathcal{M}]$$

for all $e \in \tau_{\mathcal{M}}$. Therefore, there is no way to embed or extend an idempotent monoid into a group. The management of deletions in the idempotent case within the monoid approach is hard.

One could represent sets as bags, performing deletions as mentioned for the non-idempotent case (and getting back to sets using the `distinct` operation). However, this would not fit with the usual semantics e.g. of set subtraction. Obviously, this is a limitation of our approach. Consider the next example featuring OQL's `distinct` clause:

Example 12. $v \hookleftarrow \texttt{distinct(}C\texttt{)}$. v materializes a homomorphism from $\mathit{bag} \rightarrow \mathit{set}$. While insertions to C pose no problem, $v := \texttt{distinct(}C \uplus \triangle C\texttt{)} = v \cup \texttt{distinct(}\triangle C\texttt{)}$, we cannot compute v's new value when we delete an element e of C without reconsulting C. Since v does not contain information about the multiplicity of e in C (due to the idempotence of $\cup$), we cannot decide if e has to be removed from the view, by just examining v (like an incremental update would try to do in order to avoid the possibly costly reassembly of C). $\square$

The problems with deletions in the idempotent case coincide, in some sense, with the difficulties classical work [3,13] experienced in just this area. While the treatment of views with duplicates has been studied in depth [9], incremental maintenance of views using idempotent operators is inherently impossible without further book-keeping efforts.

7 Conclusion and Future Work

Recent approaches to the maintenance problem for materialized functions decided to restrict the set of expressions these functions may be built of as well as their range and destination data types. In contrast, the present work examines simple algebraic properties of the materialized functions, namely linearity in one or more arguments, to decide if incremental update propagation is possible. By modelling database types and their update operations as monoids or groups, this paper employs an abstract view that allows for the immediate incorporation of any type that shares the algebraic properties of a monoid, respectively of any user-defined function that has been asserted to be linear in its arguments.

With the results of this work at hand, CROQUE's materialized view approach to physical database design in OODBMS becomes practicable. Updates on only conceptually present data are transformed into efficient incremental updates on materialized views of this data. OQL turned out to be a suitable MVDL in the CROQUE context, since a large subset of its clauses compute linear functions.

The bounding box operation of Example 6 already gives a flavour of how this approach might perform in non-traditional applications, which, to our knowledge, have not been extensively investigated with respect to materialized views and their maintenance. This is an open issue that we will tackle in the future. Non-standard data types, like text and multimedia data, and the natural operations on these, say, concatenation or filtering, will be closer looked at. We expect the area of application to be wide.

References

1. ANSI/X3/SPARC Study Group on Database Management Systems: Interim Report. FDT Bulletin (ACM SIGMOD), 7(2), 1975.
2. José A. Blakeley, Neil Coburn, and Paul Larson. Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates. In *ACM Transactions on Database Systems*, volume 14, pages 369–400, 1989.
3. José A. Blakeley, Paul Larson, and Frank Tompa. Efficently Updating Materialized Views. In *Proceedings of the ACM SIGMOD Conference*, 1986.
4. Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong. Comprehension Syntax. *ACM SIGMOD Record*, 23:87–96, March 1994.
5. Peter Buneman, Shamim Naqvi, Val Tannen, and Limsoon Wong. Principles of Programming with Complex Objects and Collection Types. *Theoretical Computer Science*, 149(1):3–48, 1995.
6. Leondias Fegaras and David Maier. Towards an Effective Calulus for Object Query Languages. In *ACM SIGMOD International*, 20000 N.W. Walker Road P.O. Box 91000 Portland, OR 97291-1000, 1995. Department of Computer Science and Engineering, Oregon Graduate Institute of Science & Technology.
7. Leonidas Fegaras. A Uniform Calculus for Collection Types. Technical Report 94-030, Oregon Graduate Institute of Science & Technology, 1994.
8. Leonidas Fegaras and David Maier. An Algebraic Framework for Physical OODB Design. Technical report, Oregon Graduate Institute of Science & Technology, 1995.
9. Timothy Griffin and Leonid Libkin. Incremental Maintenance of Views with Duplicates. In *Proceedings of the ACM SIGMOD Conference*, May 1995.
10. Torsten Grust, Joachim Kröger, Dieter Gluche, Andreas Heuer, and Marc H. Scholl. Query Evaluation in CROQUE—Calculus and Algebra Coincide. In Carol Small,

Paul Douglas, Roger Johnson, Peter King, and Nigel Martin, editors, *Proceedings of the 15th British National Conference on Databases (BNCOD15)*, number 1271 in Lecture Notes in Computer Science (LNCS), pages 84–100, London, Birkbeck College, July 1997. Springer Verlag.

11. Torsten Grust and Marc H. Scholl. Translating OQL into Monoid Comprehensions — Stuck with Nested Loops? Technical Report 3a/1996, Department of Mathematics and Computer Science, Database Research Group, University of Konstanz, September 1996.
12. Ashish Gupta, H.V. Jagadish, and Inderpal S. Mumick. Data Integration using Self-Maintainable Views. Technical report, AT&T Bell Laboratories, 1994.
13. Ashish Gupta and Inderpal S. Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Engineering*, 18(2):3–18, June 1995.
14. Ashish Gupta, Inderpal S. Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. In *Proceedings of the ACM SIGMOD Conference*, page 157, Washington, DC, May 1993.
15. Dallan Quass. Maintenance Expressions for Views with Aggregation. In *Proceedings of the Workshop on Materialized Views*, June 1996.
16. Marc H. Scholl. Physical Database Design for an Object-Oriented Database System. In J.C. Freytag, G. Vossen, and D.E. Maier, editors, *Query Processing for Advanced Database Applications*, chapter 14, pages 419–447. Morgan Kaufmann Publishers, 1993.
17. Marc H. Scholl, H.-Bernhard Paul, and Hans-Jörg Schek. Supporting Flat Relations by a Nested Relational Kernel. In *Proceedings of The 13th International Conference on Very Large Data Bases*, pages 137–146, 1987.
18. David A. Watt and Phil Trinder. Towards a Theory of Bulk Types. Technical Report FIDE/91/26, Computer Science Department, University of Glasgow, 1991.