Query Evaluation in CROQUE*
– Calculus and Algebra Coincide –

Torsten Grust[1], Joachim Kröger[2], Dieter Gluche[1], Andreas Heuer[2], and Marc H. Scholl[1]

[1] University of Konstanz
Database Research Group
P.O. Box D188, D-78457 Konstanz, Germany
e-mail: ⟨*firstname*⟩.⟨*lastname*⟩@uni-konstanz.de
[2] University of Rostock
Database Research Group
D-18051 Rostock, Germany
e-mail: {jo,heuer}@informatik.uni-rostock.de

Abstract. With the substantial change of declarative query languages from plain SQL to the so-called "object SQLs", in particular OQL, there has surprisingly been not much change in the way problems of query representation and optimization for such languages are tackled. We identify some of the difficulties pure algebraic approaches experience when facing object models and the operations defined for them. Calculus-style formalisms suite this challenge better, but are said not to be efficiently implementable in the database context.

This paper proposes a hybrid query representation and optimization approach, combining the strengths of a many-sorted query algebra and the monoid comprehension calculus. We show that efficient execution plans beyond nested-loop processing can be derived – not only for σ-π-$\bowtie$ queries – in such a framework. The translation process accounts for queries manipulating bulk-typed values by employing various join methods of the database engine, as well as queries that use aggregation, construction of arbitrary values, and arithmetics.

1 Introduction

The choice of the intermediate representation for a declarative query language such as OQL is guided by the vital characteristics that we demand from such a representation, namely

(a) the ability to capture the concepts (operations and types) of the query language *completely*, and
(b) the suitability to serve as a starting point for a mapping from this representation to the primitives of the underlying persistent storage manager (for efficient execution).

* CROQUE (**C**ost- and **R**ule-based **O**ptimization of object-oriented **Que**ries) is a research project funded by the German Research Association (DFG) under contracts He 1768/5-2 and Scho 554/1-2.

Traditional call interfaces to storage managers and query engines have been designed to be driven by algebraic query plans. Given a tree or DAG representing the (optimized) query expressed in terms of the system's object algebra, an interpreter or intermediate compiler generates calls to the storage manager to perform the algebraic manipulations. In fact, some of these calls implement certain algebra operators within the storage subsystem itself. Among them are selections, $\sigma[p]$, some projections, π, several join types, like $\bowtie_p$ (join), $\ltimes_p$ (semijoin), $\overline{\ltimes}_p$ (antijoin, the complement to $\ltimes$), and $⟗_p$ (two-sided outerjoin) as long as predicate p fulfills simplicity restrictions. All other operations are carried out during a postprocessing phase that mainly takes place in memory.

Algebraic approaches to query representation fit into this scenario. The initial algebraic expression is rewritten so that large subtrees of the query tree can be processed under the responsibility of the storage subsystem. Apparently, algebras fulfill requirement (b) well. However, with the substantial change of declarative query languages from plain SQL to so-called "object SQLs", in particular OQL, there has surprisingly been almost no change in the way the problems of query representation and optimization are tackled. In some sense, query processors did not reflect the shift in paradigm the query languages themselves went through.

Algebras that have been derived from the well-known nested relational algebras (NRAs) still dominate the research work in this evolving field by far. Most of these approaches did not add expressiveness to NRA, though, which made them cover the classic SQL subset of OQL only. Notable exceptions are the efforts of [12] and [13], and their query algebras AQUA and PFL, respectively.

The AQUA data model and algebra has been designed to serve as a "standard" intermediate representation for object query languages. AQUA's operators may operate on scalars as well as values of various bulk types. AQUA gains this generality by defining operators as higher-order functions that themselve take functions as parameters. Among other uses, these function parameters represent value constructors (e.g. set union or list concatenation), thus turning a set-manipulating operator into a list manipulator by appropriate parameterization.

Poulovassilis and Small's functional query language PFL realizes the general application of functions and iteration abstractions (like fold) which are needed to implement OQL. The semantically clean foundation of PFL allowed for the derivation of a large body of equivalences, the basis for the development of a PFL query optimizer.

The issue of efficiently mapping such languages to database query engine programs, however, are not dicussed. It is one of the core work items of our CROQUE project to bridge this gap for a language similar to PFL, the *monoid calculus* [8]. The calculus is expressive enough to represent OQL completely. Translating OQL into the monoid calculus results in uniform program schemes (*monoid comprehensions*) which help to keep the expected size of an optimizer rewrite rule base as well as its search space in tractable bounds.

In [5], Daniel Chan proposed collection comprehensions (over sets, bags, and lists) as an intermediate representation for object SQLs. We map quantification and aggregation to comprehensions too, making the intermediate query forms even more uniform. Further processing phases (mappings $\mathbb{A}_0$ and $\mathbb{A}_1$ to be introduced in Sect. 3) benefit from this simplicity. Chan's execution engine is non-standard in the sense that it does not support joins, but *map* and *reduce* (*fold*). Instead, our query compilation process assumes a richer algebraic query engine as sketched above (with extensions) as its translation target.

We proceed as follows. In Section 2, some light is shed on the consequences of understanding OQL as "a better SQL" only. Section 3 then introduces CROQUE's internal hybrid representation of OQL which, we believe, overcomes the major deficiencies of the pure algebraic query processing strategies. The query compilation process assumes an algebraic query engine as sketched above (with extensions) as its translation target. Section 4 summarizes.

2 Algebraic Representations for OQL

Having the instruments of NRA at hand, there is a tendency to view OQL queries as "advanced" SQL queries, with an explicit emphasis on the efficient translation of *nested subqueries* [11,6,15].

2.1 Nested `select`s

The following `select-from-where` clause is understood as the basic building block of complex queries (let $\overline{e}$ denote $e_1, \ldots, e_n$):

$$\begin{array}{l} \texttt{select struct(}a_1\texttt{:}e_{k_1}, \ldots, a_k\texttt{:}e_{k_k}\texttt{)} \\ \quad \texttt{from } E_1 \texttt{ as } e_1, \ldots, E_n \texttt{ as } e_n \\ \;\texttt{where } p(\overline{e}) \end{array}$$

The obvious algebraic translation would be

$$\pi_{e_{k_1}, \ldots, e_{k_k}}(\sigma[\lambda\overline{e}.p(\overline{e})](E_1 \times \ldots \times E_n))$$

(query optimizers then convert the Cartesian products into real joins if possible).

Note that the OQL `select-from-where` block delivers a *bag* (multiset) as its result type, which already demands a non-standard interpretation of the former set-based algebraic operators, especially π and $\times$ in this case. Now, nested subqueries may occur in the `where`-clause (which has been allowed since the early days of SQL and has been extensively studied, e.g. in [11] and a whole series of follow-up papers) and the `from`-clause. Nested subqueries in the `select`-clause have been investigated more recently. Consider

```
select struct(a1:e1, a2:select f(e1, e2)
                        from E2 as e2
                        where p'(e1, e2))
  from E1 as e1
 where p(e1)
```

The above form immediately leads to a nested projection if we apply the above algebraic translation scheme. Execution of such plans often results in inefficient nested loop processing. Consequently, recent work proposed non-standard join operators such as *binary grouping* (Γ, [6]), *hierarchical joins* [14], or the *nestjoin* (Δ, [15]), which we will look upon more closely in the following. Actually, the above query is implemented by a single nestjoin which performs a join over predicate p' and a grouping with regard to function f simultaneously:

$$(\sigma[\lambda e_1.p(e_1)](E_1)) \underset{\lambda e_1.\lambda e_2.f(e_1,e_2);\lambda e_1.\lambda e_2.p'(e_1,e_2)\to a_2}{\Delta} E_2$$

The nestjoin allows for unnesting the subquery; operand E_2 appears at the top-level, giving us several alternatives of implementing this query efficiently.

In $R\,\Delta_{f,p\to M}S$, for each $r \in R$, predicate p tests for each $s \in S$ if s is member of the group (named M in the result) belonging to r, and, if so, adds the result of $f(r,s)$ to the group. As a simple example from the relational domain, consider

R

#	A
a	1
b	2
c	3

S

#	B
a	10
a	20
c	30

$R \underset{\lambda r.\lambda s.(r.A+s.B),\lambda r.\lambda s.(r.\#=s.\#)\to M}{\Delta} S$

#	A	M
a	1	$\{11, 21\}$
b	2	$\{\}$
c	3	$\{33\}$

In contrast to the equivalent *join-group-map* sequence there is no need to introduce *null* values: empty groups are represented as empty collections which are part of the data model.

2.2 Non-linear Translations

We already end up with significantly more complex translations if we take the freedom to construct arbitrary values in the `select`-clause from the variables we declared in the `from`-clause, like OQL allows us to:

```
select struct(x:e.x, y:e.y, M:set(e.x, e.y))
  from E as e
```

First, note that this query deals with bags and sets simultaneously, which is quite typical for OQL queries but in no way common for proposed algebraic approaches. Let us ignore this fact for the sake of this example.

We cannot translate this query by a standard projection (note that we use the same values, e.g. $e.x$ and $e.y$, more than once to build the complex result) or nesting (set M includes values of different columns of E). Rather, a translation would be

$$E \underset{\mathrm{id};\lambda e_1.\lambda e_2.(e_1.x=e_2 \vee e_1.y=e_2)\to M}{\Delta} (\pi_x(E) \cup \pi_y(E))$$

(where $\mathrm{id} = \lambda x.x$ denotes the identity mapping). This translation is *non-linear* in the sense that expression E has to be copied and further processed every time we reference a variable that ranges over it (here e) to construct our complex value. This is an inherent problem of all algebraic approaches, being variable-less formalisms by nature. The translation is likely to be inefficient. A formalism that incorporates variables, like an appropriate calculus, would allow for a direct translation. Consider

$$\mathit{bag}\{\langle x = e.x, y = e.y, M = \mathit{set}\{e.x, e.y\}\rangle \mid e \leftarrow E\}$$

which denotes a *bag comprehension*, an expression of the *monoid calculus* we introduce shortly. Variable e gets sequentially bound to the elements of E and can be used in the head of the comprehension to construct values which are then collected to make up the resulting bag. More importantly, the representation is linear.

If we abstract from the problem of value construction to the general application of functions under the variable bindings generated by the `from`- and `where`-clauses, we obtain a more general view of the basic `select`-block in OQL:

$$\begin{array}{l} \texttt{select}\ f(\overline{e}) \\ \quad \texttt{from}\ E_1\ \texttt{as}\ e_1, \ldots, E_n\ \texttt{as}\ e_n \\ \ \ \texttt{where}\ p(\overline{e}) \end{array}$$

Algebras have been enriched by a very powerful operator, namely $\mathbf{map}[f]$, to account for this generalization. $\mathrm{map}[f](E)$ applies function f to every element of E, making

$$\mathrm{map}[f](\sigma[\lambda\overline{e}.p(\overline{e})](E_1 \times \ldots \times E_n))$$

a translation of the above `select`. Note that f may be quite complex, in particular an expression of the enhanced NRA itself.

2.3 Queries with Aggregates

If we consider more than just `select-from-where` (i.e. π-σ-$\bowtie$ respective map-σ-$\bowtie$) queries, we again have to enrich algebras. As an example, consider query Q_1:

$$\begin{array}{l} Q_1 = \texttt{sum(select}\ x \\ \qquad\qquad\quad \texttt{from}\ \texttt{flatten(}E\texttt{)}\ \texttt{as}\ x \\ \qquad\qquad\quad \texttt{where}\ x\ \texttt{> 0)} \end{array}$$

The `sum` aggregate – or operator `+` in general – has no algebraic counterpart yet. We could resort to treating such operators as "black boxes" that are just moved around during the query rewriting process. An optimizer that knows about `+`'s properties (like being commutative, associative, and having identity 0), however, has the choice of *folding* the aggregation into the program evaluating the `select-from-where` block, thus reducing the need for extra summation loops. This applies to aggregations in general, covering OQL's `count`, `max/min`, `avg`, `exists`, and `forall` (where the latter two aggregate boolean values).

The incorporation of general aggregations has been realized by means of **fold** operators of various sorts [1,2,17,7,13]. $\mathrm{fold}[\oplus; f](E)$ applies function f to every element of E and accumulates the function results via the binary operator $\oplus$. $\mathtt{sum}(E)$ is thus implemented by $\mathrm{fold}[+; \mathrm{id}](E)$. The uses of fold are manifold. We will use folding of boolean values to implement quantification as well. Consider $\mathrm{fold}[\vee; \lambda e.e > 10](E)$ which implements the predicate $\exists e \in E : e > 10$.

The use of aggregation, operations on scalars like arithmetics, and multiple collection types in one query are not captured by the algebras proposed in the vast majority of work on query algebras for object SQLs. Being strict, we would have to say that these algebras miss requirement (a) and therefore fail to be complete representations of OQL-like languages at all. To summarize, let us list the features we identified as crucial for an adequate OQL representation that fulfills both requirements:

(1) We clearly need **specialized join operators** (such as Δ) which can cope with certain nesting cases.
(2) We need a means to express general **function application** (map), the **construction of arbitrary values** in particular.
(3) There have to be ways to reason about **aggregation** and quantification (which we can look at as aggregation over boolean values), i.e. fold.
(4) The **treatment of the diverse collection types** (sets, bags, lists) should be uniform to avoid special operators like *list-set-join* or *bag-projection*. Incorporation of new collection types should be possible seamlessly.

2.4 The Monoid Comprehension Calculus

CROQUE's approach incorporates the monoid comprehension calculus into its query representation to achieve these fundamental goals. The monoid calculus actually does fulfill the features (2)–(4), thus meeting requirement (a) from Sect. 1. However, calculi are said to be not efficiently implementable in the database context, since the calculus expressions inherently describe nested loop processing strategies. Section 3 will present a *hybrid approach*, comprising an algebra and the monoid calculus, which shows that query execution plans beyond nested loop processing are in fact derivable in this framework.

This will finally provide us with a framework that meets both characteristics we identified in the Introduction.

We will introduce the ideas of the **monoid comprehension calculus** only briefly here, [16,18,3,8] give comprehensive surveys. The calculus can be looked at as a generalization of the well-known relational calculus. It is more general in the sense that we gain the ability to manipulate not only sets, but any type that possesses the properties of a monoid. A monoid $\mathcal{M}$ is an abstract algebraic structure equipped with an associative binary operation $\mathit{merge}[\mathcal{M}]$, having the distinguished element $\mathit{zero}[\mathcal{M}]$ as its identity. Monoids that represent collections additionally specify the function $\mathit{unit}[\mathcal{M}]$ that constructs a singleton value of that collection type. Within the monoid framework, we now represent the bulk types *set*, *bag*, and *list* as triples $(\mathit{zero}[\mathcal{M}], \mathit{unit}[\mathcal{M}], \mathit{merge}[\mathcal{M}])$, resulting in $(\{\}, \{e\}, \cup)$, $(\{\!\{\}\!\}, \{\!\{e\}\!\}, \uplus)$, and $([], [e], +\!\!+)$ respectively. The properties of $\mathit{merge}[\mathcal{M}]$ (being idempotent, commutative, or both), in fact, make the three collection types different. Our primary focus is the manipulation of bulk data, but the diverse types of aggregation and arithmetic operations we encounter during the translation of OQL fit into this framework as well. Examples of such non-collection monoids are $\mathit{sum} = (0, \mathrm{id}, +)$, $\mathit{some} = (\mathit{false}, \mathrm{id}, \vee)$, and $\mathit{all} = (\mathit{true}, \mathrm{id}, \wedge)$. While *sum* serves the obvious purpose, *some* and *all* implement existential and universal quantification. Similar monoid instances are *max*, *min*, and *prod*.

Queries are represented as a subclass of special monoid mappings, **monoid homomorphisms**, which we write down as monoid comprehensions, the corresponding generalization of set comprehensions or ZF-expressions. The homomorphisms provide us with a semantically clean way to reason about queries that involve multiple collection and scalar types since the range and destination monoids of these mappings may not be the same [8,10].

In the **monoid comprehension** $\mathcal{M}\{f \mid q_1, \ldots, q_n\}$ the **qualifiers** q_i are either **generators** of the form $e \leftarrow E$ or **predicates** p. A generator q_i sequentially binds variable e to the elements of its range E; e is bound in $q_{i+1}, \ldots, q_n$ and f. The binding of e is propagated until a predicate evaluates to *false* under this binding. Function f, the **head**, is evaluated under all bindings that pass, and the function results are then accumulated via $\mathit{merge}[\mathcal{M}]$ to make up the final result. Comprehensions nest to arbitrary depth. Abstracting monoid element operations this way gives us the ability to reason about equivalences and rewriting rules independent of particular collection types. E.g.

$$\mathcal{M}\{f \mid \overline{q_1}, e \leftarrow \mathit{zero}[\mathcal{N}], \overline{q_2}\} \equiv \mathit{zero}[\mathcal{M}]$$

(where the $\overline{q_i}$ denote sequences of qualifiers) holds for any instantiation of monoids $\mathcal{M}$ and $\mathcal{N}$, making the expected size of an optimizer rewrite rule base more manageable.

Examples of comprehensions are the following mapping from lists to bags

$$\mathit{bag}\{e \mid e \leftarrow [1,2,3,1], e \neq 2\} = \{\!\{1,1,3\}\!\}$$

or

$$set\{\langle e_1, e_2\rangle \mid e_1 \leftarrow E_1, e_2 \leftarrow E_2, some\{e_1 = c \mid c \leftarrow e_2.M\}\}$$

which computes the join of E_1 and E_2 with regard to predicate $e_1 \in e_2.M$.

Note that for E of type $\mathcal{N}$ and $\oplus$ denoting $merge[\mathcal{M}]$ of some monoid $\mathcal{M}$, the comprehension $\mathcal{M}\{e_1 \mid e \leftarrow E, e_1 \leftarrow f(e)\}$ is equivalent to $\mathrm{fold}[\oplus; f](E)$, i.e. the operation we identified as crucial for our query evaluation purposes:

$$\begin{aligned}
\mathrm{fold}[\oplus; f](zero[\mathcal{N}]) &= zero[\mathcal{M}] \\
\mathrm{fold}[\oplus; f](unit[\mathcal{N}](e)) &= f(e) \\
\mathrm{fold}[\oplus; f](merge[\mathcal{N}](e_1, e_2)) &= f(e_1) \oplus \mathrm{fold}[\oplus; f](e_2)
\end{aligned}$$

These three equations reflect the way of how values are constructed by the calculus, namely by the application of $zero[\mathcal{M}]$, $unit[\mathcal{M}]$, and $merge[\mathcal{M}]$ exclusively.

In fact, one can show that fold's expressiveness is sufficient to capture OQL. In [10] we actually gave a complete mapping of OQL to the monoid comprehension calculus. Summing things up, what we gain with the monoid calculus is (1) the necessary expressive power to capture OQL-like languages, (2) a uniform and extensible way of manipulating values of diverse collection types, (3) a uniform treatment of arithmetics, quantification, and aggregation, and (4) a formalism that can handle variables and bindings like the source query language can.

Mapping the source language, OQL, to the monoid calculus as the target formalism is actually a straightforward process. The OQL `from`-clause translates into a sequence of generators, predicates in the `where`-clause introduce filters, while the `select`-clause corresponds to a comprehension's head, i.e. function application. Let $\mathbb{T}$ be the mapping from OQL to the monoid calculus. The core of $\mathbb{T}$ would then read

$$\mathbb{T}\left(\begin{array}{l}\texttt{select } f(\overline{e}) \\ \quad \texttt{from } E_1 \texttt{ as } e_1, \ldots, E_n \texttt{ as } e_n \\ \texttt{where } p(\overline{e})\end{array}\right) = bag\{\mathbb{T}(f(\overline{e})) \mid \begin{array}{l} e_1 \leftarrow \mathbb{T}(E_1), \\ \ldots, \\ e_n \leftarrow \mathbb{T}(E_n), \mathbb{T}(p(\overline{e}))\}\end{array}$$

This meets the OQL semantics, including the case of one or more E_i being empty collections: the overall result will be empty. [10] extends $\mathbb{T}$ to capture the complete specification of OQL 1.2 [4]. The problems we experienced with the construction of complex values, especially non-linear translations caused by the use of variables, do not affect the calculus, as we have mentioned before. Consider

$$\begin{array}{l} Q_2 = \texttt{select distinct } e \\ \qquad\quad \texttt{from } E \texttt{ as } e \\ \qquad\quad \texttt{where exists } x \texttt{ in } M\texttt{: } N \texttt{ <= set}(e, x) \end{array}$$

(where `<=` denotes the set inclusion predicate) and its image under $\mathbb{T}$

$$\mathbb{T}(Q_2) = set\{e \mid e \leftarrow E, some\{N \subseteq set\{e, x\} \mid x \leftarrow M\}\}$$

However, how are we supposed to find an efficient algebraic execution plan for a nested comprehension like this?

3 A Hybrid Translation and Optimization Schema for OQL

We just observed that monoid comprehensions basically correspond to nested folds when translated into an extended nested algebra. While this is true, this observation may lead to the impression that one is stuck with nested loops when it comes to finding execution plans for comprehensions: $\mathrm{fold}[\oplus; f]$ accesses each element of the argument collection (possibly even in order, if applied to a list), applies function f and then merges the result via $\oplus$. There is no obvious strategy for the efficient evaluation of monoid calculus expressions like the strategy for NRA we sketched in the Introduction. We miss a mapping of nested comprehensions to the primitives of query engines which are tailored to evaluate terms of a particular algebra.

On the other hand there are classes of OQL queries that clearly are of algebraic nature, e.g., compute joins, perform selections, simple projections, grouping, and nesting, and we are better off to translate these queries into algebraic plans directly. The large body of knowledge of optimization methods in this field should clearly be reused if possible.

The apparent potential of the monoid calculus and the latter observation motivated the *hybrid approach* to OQL representation and optimization that we pursue in the CROQUE project:

- Translate the source OQL query into a mix of monoid calculus and algebra, which we will introduce below. If we are able to detect subqueries that can be computed by algebraic terms at this point, we choose an algebraic representation for these subqueries. To ensure a clean semantics for such mixed representations we will define the algebra operators by means of monoid calculus terms. In addition, these definitions will provide a definite semantics for the operators when applied to the different collection types. We obtain a many-sorted query algebra this way.
- Perform rewriting on this representation. The whole body of rule-based algebraic rewriting techniques applies. We additionally may apply calculus rewrite rules which above all allow for simple unnesting of complex subqueries. Comprehension normalization (which basically is accomplished by applying the rewriting rules of Fig. 3) actually implements numerous unnesting techniques found in the literature. Calculus expressions may be translated into algebra and vice versa. The rewriting process follows the heuristic of translating calculus into equivalent algebraic terms.
- As a final step, convert remaining calculus terms into folds. We will show shortly how this can be done. It is a requirement for the underlying query execution engine to provide facilities for the evaluation of fold.

In the remainder, we will present the strategy sketched above. Let us continue by introducing the query algebra to the depth needed.

3.1 A Monoid-Aware Algebra

The following introduces some operators of the algebra by putting the operators' definitions down to the equivalent monoid comprehensions as mentioned above. We gain several advantages with this approach: (1) a semantically clean interaction of the algebra with the calculus is guaranteed. Calculus terms may play the role of selection or join predicates, may represent functions (e.g., as they are needed as parameters for map and fold), or may be arguments to algebra operators in general. (2) The generic definitions hold for any monoid instantiation, as long as well-definedness conditions are obeyed. The algebra will operate on multiple collection types, not only sets. (3) We uncover the control structure of the operators. Unfolding and loop fusion becomes possible.

Due to space restrictions, we merely present the basic ideas here, but the design of the operators follows a common principle; [9] contains a comprehensive list of all operators.

In the following, read symbol :: as *has type*. Let *int*, *bool*, and *string* be atomic types. $\langle a_1 :: \alpha_1, \ldots, a_n :: \alpha_n \rangle$ is a record with field tags a_i and respective field types α_i; $\otimes$ concatenates records. With $\mathcal{M}$ being a collection monoid (*set*, *bag*, *list*), and α being some type, $\alpha\mathcal{M}$ is the type denoting a $\mathcal{M}$-collection of α-members, e.g. *int list*.

The core of the algebra is $\text{fold}[\oplus; f](E)$, which applies f to the elements of E. It then uses $\oplus$, the *merge* operation of some monoid $\mathcal{M}$, to collect the result. In particular, if we have $E :: \alpha\mathcal{M}, f :: \alpha \to \beta\mathcal{M}, \oplus :: \beta\mathcal{M} \times \beta\mathcal{M} \to \beta\mathcal{M}$, we define

$$\text{fold}[\oplus; \lambda e.f(e)](E) = \mathcal{M}\{e' \mid e \leftarrow E, e' \leftarrow f(e)\}$$

The result's type is $\beta\mathcal{M}$.

The semijoin $E_1 \ltimes_p E_2$ is a join variant that delivers only those left operand objects that have at least one join partner with respect to the predicate p. Implementation can be efficient in terms of space (buffers or files that hold the intermediate result only carry E_1 objects) and time (as soon as any join partner is found for an E_1 object, it belongs to the result; no other E_2 object has to be touched). Let $E_1 :: \alpha\mathcal{M}, E_2 :: \beta\mathcal{N}, p :: \alpha \times \beta \to \text{bool}$, then we define

$$E_1 \underset{\lambda e_1.\lambda e_2.p(e_1,e_2)}{\ltimes} E_2 = \mathcal{M}\{e_1 \mid e_1 \leftarrow E_1, \mathit{some}\{p(e_1, e_2) \mid e_2 \leftarrow E_2\}\}$$

The aforementioned nestjoin, $E_1 \,\Delta_{f;p\to A} E_2$ is a non-standard join between E_1 and E_2 in the sense that for each E_1 object we create a group A of E_2 objects that match the join predicate p. On creation of this group we apply f. Types are as follows: $E_1 :: \alpha\mathcal{M}, E_2 :: \beta\mathcal{N}, f :: \alpha \times \beta \to \gamma, p :: \alpha \times \beta \to \text{bool}$.

We then have

$$E_1 \underset{\substack{\lambda e_1.\lambda e_2.f(e_1,e_2);\\ \lambda e_1.\lambda e_2.p(e_1,e_2)\to A}}{\Delta} E_2 = \mathcal{M}\{e_1 \otimes \langle A = \mathcal{N}\{f(e_1,e_2) \mid e_2 \leftarrow E_2, \\ p(e_1,e_2)\}\rangle \mid e_1 \leftarrow E_1\}$$

The result is of type $\alpha \times \langle A :: \gamma\mathcal{N}\rangle\mathcal{M}$. Δ is valuable when translating queries with nesting in the `select` clause, as well as for the evaluation of complex predicates between `select` blocks. Efficient techniques for the evaluation of grouping, e.g. sorting or indices, apply to the implementation of Δ, too.

3.2 Execution Strategies for Monoid Comprehensions

Achieving good execution plans for the monoid calculus is not as obvious as for algebras. Provided we believe the calculus to be a valuable approach to object SQL query processing, we face the problem of how to find good QEPs for potentially deeply nested comprehension expressions. In what follows we will present a mapping, $\mathbb{A}$, from the calculus to a hybrid representation employing the monoid-aware nested algebra, which we assume to be the target language of the query execution engine.

Successive applications of $\mathbb{A}$ will replace monoid calculus terms by algebraic equivalents. The intermediate results of $\mathbb{A}$ are subject to algebraic and calculus-based rewriting, to the results of which $\mathbb{A}$ is applied again. Since any calculus term at least has its canonical fold representation we are guaranteed to have a way out of this process. The rewriting finally yields a term of the target algebra we introduced, i.e. a program for our query engine. The degrees of freedom in this process define the dimensions of the search space along which we look for the optimal representation of our query: calculus $\to$ calculus, calculus $\leftrightarrow$ algebra, and the classical algebra $\to$ algebra.

After introducing the ideas of $\mathbb{A}$ we will present the derivation of execution plans in this framework by example, including the problematic queries Q_1 and Q_2 from Sect. 2. Let us start with a first basic version $\mathbb{A}_0$ of the mapping and improve $\mathbb{A}$ as we go.

$\mathbb{A}_0$ – Comprehensions Implemented by fold. The initial mapping $\mathbb{A}_0$ puts the comprehensions down to their nature of being monoid homomorphisms, which, in turn, are implemented by the algebra's fold operator (see Fig. 1). $\mathbb{A}_0$ is defined via structural recursion over the list of qualifiers. Constants (Rule 1) and database variables (2), e.g. globally named collections, provide the base cases for the recursion. Rules (3)–(5) reduce the qualifier list from its head on. Rule (4) introduces a fold over a generator's range. Predicates are handled via a conditonal (5) that replaces non-qualifying elements by $\mathcal{M}$'s identity *zero*[$\mathcal{M}$], which does not contribute to the result.

Note that it would be sufficient for $\mathbb{A}_0$'s target query engine to implement fold, the respective monoid operations, and a conditional to execute the resulting plans. While this at least provides a complete mapping from

$$
\begin{array}{rll}
\mathbb{A}_0(c) & \to c & (1)\\
\mathbb{A}_0(v) & \to v & (2)\\
\mathbb{A}_0(\mathcal{M}\{f \mid \}) & \to \mathit{unit}[\mathcal{M}](\mathbb{A}_0(f)) & (3)\\
\mathbb{A}_0(\mathcal{M}\{f \mid e \leftarrow E, \overline{q}\}) & \to \mathrm{fold}[\mathit{merge}[\mathcal{M}]; \lambda e.\mathbb{A}_0(\mathcal{M}\{f \mid \overline{q}\})](\mathbb{A}_0(E)) & (4)\\
\mathbb{A}_0(\mathcal{M}\{f \mid p, \overline{q}\}) & \to \text{if } \mathbb{A}_0(p) \text{ then } \mathbb{A}_0(\mathcal{M}\{f \mid \overline{q}\}) \text{ else } \mathit{zero}[\mathcal{M}] & (5)
\end{array}
$$

Fig. 1. Mapping $\mathbb{A}_0$. Implement comprehensions by nested folds.

comprehensions to the algebra, we always end up with nested applications of fold because of (4), i.e. nested-loop processing. Consider the result of $\mathbb{A}_0$'s application to $\mathbb{T}(Q_2)$ in which the inner fold implementing the existential quantification is executed for each $e \in E$:

$$
\begin{aligned}
&(\mathbb{A}_0 \circ \mathbb{T})(Q_2)\\
&\quad = \mathbb{A}_0(\mathit{set}\{e \mid e \leftarrow E, \mathit{some}\{N \subseteq \mathit{set}\{e,x\} \mid x \leftarrow M\}\})\\
&\quad \underset{(4)}{=} \mathrm{fold}[\cup; \lambda e.\mathbb{A}_0(\mathit{set}\{e \mid \mathit{some}\{N \subseteq \mathit{set}\{e,x\} \mid x \leftarrow M\}\})](\mathbb{A}_0(E))\\
&\quad \underset{(5)}{=} \mathrm{fold}[\cup; \lambda e.\text{if } \mathbb{A}_0(\mathit{some}\{N \subseteq \mathit{set}\{e,x\} \mid x \leftarrow M\}) \text{ then } \mathbb{A}_0(\mathit{set}\{e \mid \})\\
&\qquad\qquad \text{else } \{\}](E)\\
&\quad \underset{(4)}{=} \mathrm{fold}[\cup; \lambda e.\text{if } \mathrm{fold}[\vee; \lambda x.\mathbb{A}_0(\mathit{some}\{N \subseteq \mathit{set}\{e,x\} \mid \})](M) \text{ then } \{\mathbb{A}_0(e)\}\\
&\qquad\qquad \text{else } \{\}](E)\\
&\quad \underset{(3)}{=} \mathrm{fold}[\cup; \lambda e.\text{if } \mathrm{fold}[\vee; \lambda x.N \subseteq \{e,x\}](M) \text{ then } \{e\} \text{ else } \{\}](E)
\end{aligned}
$$

$\mathbb{A}_0$ fails to employ the more efficient algorithms of the query engine.

However, since the query engine's operators have been put down to monoid calculus terms, we have the possibility to incorporate join strategies into the mapping.

$\mathbb{A}_1$ – Reduction by Pattern Matching. The extended version $\mathbb{A}_1$ understands the algebra operator definitions as *patterns* to be detected in the comprehension expressions. On a successful match, $\mathbb{A}_1$ replaces the match by its equivalent algebraic term. We derive an initial $\mathbb{A}_1$ by adopting the rules of $\mathbb{A}_0$ and adding the cases of Fig. 2 which improve the processing of predicates and quantifiers. More than one rule may match at a time and any rule may then be selected arbitrarily. However, we listed the rules in the order an optimizer heuristic should try to match, e.g. we prefer to implement universal quantification by $\overline{\ltimes}$ (7) before resorting to a σ with a nested fold as its predicate (10). Note that (10) covers (5); we can therefore drop (5) and the if-then-else conditional completely. Since the $\overline{\ltimes}$ is feasible only if no dependencies (besides the join predicate) between the join partners exist, we check this condition by examining the set of free variables $\mathcal{F}(\cdot)$. Some rules introduce new generator variables and we adapt the comprehension head accordingly, mainly by renaming variables, thus turning f into f' in (8) and

(10). Finally, the comprehension pattern of (11) serves as a catch-all case and implements the application of f via map as discussed in Sect. 2.2.

$$\begin{aligned}
\mathbb{A}_1\,(\mathcal{M}\{f \mid \overline{q}, e \leftarrow E,\; some\{p \mid \overline{q_1}\}\}) &\rightarrow \mathbb{A}_1\,(\mathcal{M}\{f \mid \overline{q}, e \leftarrow \mathbb{A}_1\,(E) \underset{\mathbb{A}_1\,(p)}{\ltimes} \mathbb{A}_1\,(\overline{q_1})\}) \\
&\qquad \text{if } e \notin \mathcal{F}(\overline{q_1}) && (6)\\
\mathbb{A}_1\,(\mathcal{M}\{f \mid \overline{q}, e \leftarrow E,\; all\{p \mid \overline{q_1}\}\}) &\rightarrow \mathbb{A}_1\,(\mathcal{M}\{f \mid \overline{q}, e \leftarrow \mathbb{A}_1\,(E) \underset{\neg\mathbb{A}_1\,(p)}{\overline{\ltimes}} \mathbb{A}_1\,(\overline{q_1})\}) \\
&\qquad \text{if } e \notin \mathcal{F}(\overline{q_1}) && (7)\\
\mathbb{A}_1\,(\mathcal{M}\{f \mid \overline{q}, e_1 \leftarrow E_1,\; e_2 \leftarrow E_2\}) &\rightarrow \mathbb{A}_1\,(\mathcal{M}\{f' \mid \overline{q}, e \leftarrow \mathbb{A}_1\,(E_1) \times \mathbb{A}_1\,(E_2)\}) \\
&\qquad \text{if } e_1 \notin \mathcal{F}(E_2) && (8)\\
\mathbb{A}_1\,(\mathcal{M}\{f \mid \overline{q}, p_1, p_2\}) &\rightarrow \mathbb{A}_1\,(\mathcal{M}\{f \mid \overline{q}, p_1 \wedge p_2\}) && (9)\\
\mathbb{A}_1\,(\mathcal{M}\{f \mid \overline{q}, p\}) &\rightarrow \mathbb{A}_1\,(\mathcal{M}\{f' \mid e \leftarrow \sigma[\mathbb{A}_1\,(p)](\mathbb{A}_1\,(\overline{q}))\}) && (10)\\
\mathbb{A}_1\,(\mathcal{M}\{f \mid \overline{q}\}) &\rightarrow \mathrm{map}[\mathbb{A}_1\,(f)](\mathbb{A}_1\,(\overline{q})) && (11)
\end{aligned}$$

Fig. 2. Mapping $\mathbb{A}_1$. Operator pattern matching.

If we turn back to query Q_2 again, we now rewrite as follows:

$$\begin{aligned}
&\mathbb{A}_1\,(set\{e \mid e \leftarrow E, some\{N \subseteq set\{e, x\} \mid x \leftarrow M\}\}) \\
&\quad \underset{(6)}{=} \mathbb{A}_1\,(set\{e' \mid e' \leftarrow E \underset{\lambda e.\lambda x.\mathbb{A}_1\,(N \subseteq set\{e,x\})}{\ltimes} M\}) \\
&\quad \underset{(4)}{=} \mathrm{fold}[\cup; \lambda e'.\mathbb{A}_1\,(set\{e' \mid \})](E \underset{\lambda e.\lambda x.N \subseteq \{e,x\}}{\ltimes} M) \\
&\quad \underset{(3)}{=} \mathrm{fold}[\cup; \lambda e'.\{e'\}](E \underset{\lambda e.\lambda x.N \subseteq \{e,x\}}{\ltimes} M)
\end{aligned}$$

in which we can drop the outer fold if $E :: \alpha\ set$ (the fold merely implements a type coercion from the semijoin's output type to $\alpha\ set$).

Figure 2 only shows a subset of the rewriting rules, since each algebra operator introduces at least one rule according to its definition. See [9] for the complete $\mathbb{A}_1$ rewrite rule base. An actual optimizer based on our approach will, however, certainly feature a larger set of rules.

Up to now, we have improved the calculus $\mapsto$ algebra mapping. By adding the rules of Fig. 3 we extend the translation scheme once more and introduce the calculus $\mapsto$ calculus (i.e. comprehension level) rewriting dimension. While (12) and (13) allow for selection pushdown and join reordering, respectively, (14) unnests a comprehension (over monoid $\mathcal{N}$). The binding $e \equiv f'$ makes e a synonym for f' in the comprehension's head. This last rule, together with (18) and (15), serves to remove dependencies between potential join partners, thus increasing the chance of introducing efficient joins and abandoning nested loop processing.

$$
\begin{array}{rll}
\mathbb{A}_1\,(\mathcal{M}\{f \mid \overline{q}, e \leftarrow E, p\}) \rightarrow & \mathbb{A}_1\,(\mathcal{M}\{f \mid \overline{q}, p, e \leftarrow E\}) & \\
& \text{if } e \notin F(p) & (12) \\
\mathbb{A}_1\,(\mathcal{M}\{f \mid \overline{q}, e_1 \leftarrow E_1, e_2 \leftarrow E_2\}) \rightarrow & \mathbb{A}_1\,(\mathcal{M}\{f \mid \overline{q}, e_2 \leftarrow E_2, & \\
& \qquad e_1 \leftarrow E_1\}) & \\
& \text{if } e_1 \notin F(E_2) & (13) \\
\mathbb{A}_1\,(\mathcal{M}\{f \mid \overline{q}, e \leftarrow \mathcal{N}\{f' \mid \overline{q_1}\}\}) \rightarrow & \mathbb{A}_1\,(\mathcal{M}\{f \mid \overline{q}, \overline{q_1}, e \equiv f'\}) & \\
& \text{if } \mathcal{N} \preceq \mathcal{M} & (14) \\
\mathbb{A}_1\,(\mathcal{M}\{f \mid \overline{q}, \mathit{some}\{p \mid \overline{q_1}\}, \overline{q_2}\}) \rightarrow & \mathbb{A}_1\,(\mathcal{M}\{f \mid \overline{q}, \overline{q_1}, p, \overline{q_2}\}) & \\
& \text{if } \mathit{merge}[\mathcal{M}] \text{ is idempotent} & (15) \\
\mathbb{A}_1\,(\mathcal{M}\{f \mid \overline{q}, e \leftarrow \mathit{zero}[\mathcal{N}], \overline{q_1}\}) \rightarrow & \mathit{zero}[\mathcal{M}] & (16) \\
\mathbb{A}_1\,(\mathcal{M}\{f \mid \overline{q}, e \leftarrow \mathit{unit}[\mathcal{N}](e'), \overline{q_1}\}) \rightarrow & \mathbb{A}_1\,(\mathcal{M}\{f \mid \overline{q}, e \equiv e', \overline{q_1}\}) & (17) \\
\mathbb{A}_1\,(\mathcal{M}\{f \mid \overline{q}, e \leftarrow \mathit{merge}[\mathcal{N}](E_1, E_2), \overline{q_1}\}) \rightarrow & \mathit{merge}[\mathcal{M}] & \\
& \quad (\mathbb{A}_1\,(\mathcal{M}\{f \mid \overline{q}, e \leftarrow E_1, \overline{q_1}\}), & \\
& \quad\; \mathbb{A}_1\,(\mathcal{M}\{f \mid \overline{q}, e \leftarrow E_2, \overline{q_1}\})) & \\
& \text{if } \mathcal{M} \text{ is commutative} & (18)
\end{array}
$$

Fig. 3. $\mathbb{A}_1$ extended. Comprehension level rewriting.

By translating algebra operators, aggregations, quantifiers, and predicates into their comprehension equivalents, we uncover their control structure. E.g., we may view $N \subseteq M$ as some atomic set predicate for which we expect a corresponding query engine procedure, but as well may carry out the subset test explicitly: $\mathit{all}\{\mathit{some}\{y = z \mid z \leftarrow M\} \mid y \leftarrow N\}$. This opens possibilities for join pattern detection or loop fusion. The latter is not feasible if we view the operators only as black box functions that fulfill certain abstract algebraic properties. The translation of Q_2 backs up this claim. Uncovering $\subseteq$ makes an antijoin applicable and we are actually able to deduce a pure algebraic plan:

$$
\begin{array}{l}
\mathbb{A}_1\,(\mathit{set}\{e \mid e \leftarrow E, \mathit{some}\{\mathit{all}\{\mathit{some}\{y = z \mid z \leftarrow \{e, x\}\} \mid y \leftarrow N\} \mid x \leftarrow M\}\}) \\
\quad \underset{(15)}{=} \mathbb{A}_1\,(\mathit{set}\{e \mid e \leftarrow E, x \leftarrow M, \mathit{all}\{\mathit{some}\{y = z \mid z \leftarrow \{e, x\}\} \mid y \leftarrow N\}\}) \\
\quad \underset{(18)}{=} \mathbb{A}_1\,(\mathit{set}\{e \mid e \leftarrow E, x \leftarrow M, \mathit{all}\{y = e \vee y = x \mid y \leftarrow N\}\}) \\
\quad \underset{(8)}{=} \mathbb{A}_1\,(\mathit{set}\{e \mid e' \leftarrow E \times M, e \equiv e'_E, x \equiv e'_M, \mathit{all}\{y = e \vee y = x \mid y \leftarrow N\}\}) \\
\quad \underset{(7)}{=} \mathbb{A}_1\,(\mathit{set}\{e''_E \mid e'' \leftarrow (E \times M) \underset{\lambda e'.\lambda y.y \neq e'_E \wedge y \neq e'_M}{\overline{\ltimes}} N\}) \\
\quad \underset{(11)}{=} \pi_E((E \times M) \underset{\lambda e'.\lambda y.y \neq e'_E \wedge y \neq e'_M}{\overline{\ltimes}} N)
\end{array}
$$

In the above, a subscripted variable e_M denotes e's value projected onto the fields $l_1, \ldots, l_n$ if $M :: \langle l_1 :: \tau_1, \ldots, l_n :: \tau_n \rangle\ C$ for any collection constructor C (record projection).

To conclude the example rewritings, let us turn back to query Q_1 and track the rewriting steps that an optimizer equipped with the complete rule base could undertake. Recall, that Q_1 flattened E, a bag of collections, and then added all elements being greater than zero to obtain the final result:

$$
\begin{aligned}
&(\mathbb{A}_1 \circ \mathbb{T}) \begin{pmatrix} \texttt{sum(select}\ x \\ \quad \texttt{from}\ \texttt{flatten(}E\texttt{)}\ \texttt{as}\ x \\ \quad \texttt{where}\ x\ \texttt{>}\ \texttt{0)} \end{pmatrix} \\
&\underset{(\mathbb{T})}{=} \mathbb{A}_1 (\mathit{sum}\{e \mid e \leftarrow \mathit{bag}\{\mathbb{A}_1(x) \mid x \leftarrow \mathbb{A}_1(\texttt{flatten(}E\texttt{)}, \mathbb{A}_1(x\ \texttt{>}\ \texttt{0)}\}\}) \\
&\underset{(\texttt{flatten})}{=} \mathbb{A}_1 (\mathit{sum}\{e \mid e \leftarrow \mathit{bag}\{x \mid x \leftarrow \mathit{bag}\{z \mid z' \leftarrow E, z \leftarrow z'\}, x > 0\}\}) \\
&\underset{(10)}{=} \mathbb{A}_1 (\mathit{sum}\{e \mid e \leftarrow \sigma[\lambda x.x > 0](\mathit{bag}\{z \mid z' \leftarrow E, z \leftarrow z'\})\}) \\
&\underset{(\mathbb{A}_0)}{=} \mathbb{A}_1 (\mathit{sum}\{e \mid e \leftarrow \sigma[\lambda x.x > 0](\mathrm{fold}[\uplus; \mathrm{id}](E))\}) \\
&\underset{(\mathbb{A}_0)}{=} \mathrm{fold}[+; \mathrm{id}](\sigma[\lambda x.x > 0](\mathrm{fold}[\uplus; \mathrm{id}](E)))
\end{aligned}
$$

In principle, the resulting program performs three loops to accomplish its task. Loop fusion techniques, however, allow us to transform the query into a program that only needs two loops. Instead of flattening, each collection in E is summed up, mapping elements less than or equal to zero to $+$'s identity 0, giving a bag of intermediate sums. A second outer loop then sums up: $\mathrm{fold}[+; \lambda e.\mathrm{fold}[+; \lambda x.\mathrm{if}\ x > 0\ \mathrm{then}\ x\ \mathrm{else}\ 0](e)](E)$. This superior alternative may directly be derived by $\mathbb{A}_0$, i.e. actually is element of our optimizer's solution space.

Since $+$ is a first-class operator in the monoid framework and its algebraic properties are well known (esp. having identity 0), we can apply loop fusion here. Lifting any type onto the monoid level makes the operations of these types transparent and therefore subject to further analysis and optimization. This is a consequence of the basic requirement (a) we stated at the very beginning: A *complete* representation of the query languages' types and operations in which no "black boxes" remain has to be aspired.

A basic rewriting strategy for an optimizer based on the $\mathbb{A}_1$ rules would try to unnest nested comprehensions (i.e. subqueries) using (14), (15), and (18). The goal is to remove variable dependencies that prevent the introduction of joins. [19,8] discuss a normal form for monoid comprehensions that realizes a minimal number of such dependencies. We move along the calculus $\mapsto$ calculus dimension this way. Then, moves in the calculus $\mapsto$ algebra dimension are carried out in order to introduce joins and the other algebra operators. Rules (6)–(11) implement such moves. This is based on the assumption that the query engine knows about efficient implementations for these operators. All these rewriting steps may be interleaved by algebraic as well as comprehension level rewriting. To obtain a program for the query execution engine, all remaining calculus terms are converted into folds as a final step.

4 Conclusions and Further Work

Facing todays "object SQLs", in particular OQL, this paper proposed to let the monoid play the role sets played in the relational setting. The monoid calculus has been shown to provide the expressiveness that is needed to completely capture OQL. In order to obtain efficient execution plans for algebra-tailored query engines beyond nested-loop processing, we introduced a monoid-aware fold algebra. $\mathbb{A}_1 \circ \mathbb{T}$ implemented a rewrite-based mapping from the source query language OQL to this target algebra, employing non-standard joins. This hybrid approach, combining algebra and calculus, takes advantage of the strengths of both formalisms during the derivation of query engine programs.

Some issues are still open. Most importantly, a rewriting strategy based on $\mathbb{A}_1$ and its successors has to be developed. Moves in the three rewriting dimensions have to be coordinated. Should the algebra be extended to cope with further particular nesting cases, in the spirit of Δ? Which influence does fold have on the construction of a suitable query engine? Additionally, incorporating the calculus into the evaluation framework opens new possibilities to exploit further program transformation techniques, like loop fusion, thus widening the view of database query optimization. Advanced query optimization techniques, like bypass processing for disjunctive queries, have to be adapted to fit the hybrid approach. [10] already took a step in this direction. Our CROQUE project is underway implementing the setup we discussed in this paper. Although based on a subset of rules of $\mathbb{A}_1$ and a limited set of algebra operators only, first tests showed promising translation results for a broad range of OQL queries.

References

1. Francois Bancilhon, Ted Briggs, Setrag Khoshafian, and Patrick Valduriez. FAD, a Powerful and Simple Database Language. In Peter M. Stocker, William Kent, and Peter Hammersley, editors, *Proceedings of the 13th Int'l Conference on Very Large Databases (VLDB)*, pages 97–105. Morgan Kaufmann Publishers, September 1987.
2. Catriel Beeri and Yoram Kornatzky. Algebraic Optimization of Object-Oriented Languages. In Paris C. Kanellakis and Serge Abiteboul, editors, *Proceedings of the 3rd Int'l Conference on Database Theory (ICDT)*, volume 470 of *LNCS*, pages 72–88. Springer Verlag, December 1990.
3. Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong. Comprehension Syntax. In *SIGMOD Record*, volume 23, pages 87–96, March 1994.

4. Rick G. Cattell, editor. *The Object Database Standard: ODMG–93*, chapter 4, Object Query Language (Release 1.2), pages 65–96. Morgan Kaufmann Publishers, 1995. Updates to Release 1.1.
5. Daniel Chan. *Object-oriented Language Design and Processing*. PhD thesis, Computer Science Department, University of Glasgow, 1994.
6. Sophie Cluet and Guido Moerkotte. Nested Queries in Object Bases. In *Proceedings of the 4th Int'l Workshop on Database Programming Languages (DBPL)*, September 1993.
7. Leonidas Fegaras. Efficient Optimization of Iterative Queries. In *Proceedings of the 4th Int'l Workshop on Database Programming Languages (DBPL)*, pages 200–225, September 1993.
8. Leonidas Fegaras and David Maier. Towards an Effective Calculus for Object Query Languages. In *Proceedings of the 1995 ACM SIGMOD Int'l Conference on Management of Data*, May 1995.
9. Torsten Grust, Joachim Kröger, Dieter Gluche, Andreas Heuer, and Marc H. Scholl. Query Evaluation in CROQUE: Calculus and Algebra Coincide. Technical Report in preparation, Department of Mathematics and Computer Science, Database Research Group, University of Konstanz, April 1997.
10. 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.
11. Won Kim. On Optimizing an SQL-like Nested Query. *ACM Transactions on Database Systems*, 7(3):443–469, September 1982.
12. Theodore W. Leung, Gail Mitchell, Bharathi Subramanian, Stanley B. Zdonik, and other. The AQUA Data Model and Algebra. In *Proceedings of the 4th Int'l Workshop on Database Programming Languages (DBPL)*, September 1993.
13. Alexandra Poulovassilis and Carol Small. Investigation of Algebraic Query Optimisation for Database Programming Languages. In *Proceedings of the 20th Int'l Conference on Very Large Databases (VLDB)*, pages 415–426, Santiago, Chile, September 1994.
14. Christian Rich, Arnon Rosenthal, and Marc H. Scholl. Reducing Duplicate Work in Relational Join(s): A Unified Approach. In *Proceedings of the Int'l Conference on Information Systems and Management of Data (CISMOD)*, pages 87–102, Delhi, India, October 1993.
15. Hennie J. Steenhagen, Peter M.G. Apers, Henk M. Blanken, and Rolf A. de By. From Nested-Loop to Join Queries in OODB. In *Proceedings of the 20th Int'l Conference on Very Large Databases (VLDB)*, pages 618–629, Santiago, Chile, September 1994.
16. Phil Trinder. Comprehensions, a Query Notation for DBPLs. In *Proceedings of the Third Int'l Workshop on Database Programming Languages (DBPL)*, pages 55–68, Nafplion, Greece, 1991.
17. Bennet Vance. Towards an Object-Oriented Query Algebra. Technical Report 91–008, Oregon Graduate Institute of Science & Technology, January 1992.
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.
19. Limsoon Wong. Normal Forms and Conservative Properties for Query Languages over Collection Types. In *Proceedings of the 12th ACM Symposium on Principles of Database Systems*, pages 26–36, May 1993.