Improving List Comprehension Database Queries

Phil Trinder
Oxford University [*]

Philip Wadler
Glasgow University

Abstract

The task of increasing the efficiency of database queries has recieved considerable attention. In this paper we describe the improvement of queries expressed as list comprehensions in a lazy functional language. The database literature identifies four algebraic and two implementation-based improvement strategies. For each strategy we show an equivalent improvement for queries expressed as list comprehensions. This means that well-developed database algorithms that improve queries using several of these strategies can be emulated to improve comprehension queries. We are also able to improve queries which require greater power than that provided by the relational algebra. Most of the improvements entail transforming a simple, inefficient query into a more complex, but more efficient form. We illustrate each improvement using examples drawn from the database literature.

1 Introduction

The functional programming community is often accused of being too inward looking. Functional languages are used to implement other functional languages, and few real-world programs have been written in them. There are exceptions to this, for example work by Luk [16] and Wray [24]. In this paper we present an application of functional programming techniques to the real world problem of improving database queries.

The relational calculus is a query formalism which underlies several database systems [9, 13, 27]. A query language is said to be relationally *complete*, i.e. adequately expressive, if it is at least as powerful as the calculus. Any relational calculus query can be translated into an equivalent list comprehension. Indeed, comprehension notation has greater power than the relational calculus as, unlike the calculus, it can express computation and recursion. Also, list comprehension queries can be cleanly integrated with a functional database described elsewhere [1]. These results are reported in [21, 22].

[*]Currently visiting at the Computing Science Department, Glasgow University, Glasgow G12 8QQ, Scotland. Email: trinder@uk.ac.glasgow.cs

Improvements in conventional relational systems fall into two categories, algebraic and implementation-based. Algebraic improvements are the result of transforming a query into a more efficient form using the relational algebra. Implementation-based improvements are the result of using information about how the data is stored. This information may include the size of the relations, what indices exist and the nature of those indices.

To improve conventional queries Ullman [25] identifies four algebraic and two physical implementation ideas. For each of these ideas we show the equivalent improvement for list comprehension queries. This means that well-developed database algorithms that use these improvements can be emulated to improve comprehension queries. An example of a query being improved by emulating an existing algorithm can be found in [22].

There is a class of useful queries which cannot be expressed in the relational algebra. These entail computation or recursion. In [22] the transformations presented here are used to improve two queries that involve both computation and recursion.

Most of the improvements entail transforming a simple, inefficient query into a more complex, but more efficient form. We present the transformations we have developed and illustrate each improvement by example. Only a few transformations are required, but those described are used several times.

Using list processing to compute database queries is not a new idea. Buneman et al have built a commercial database system using lists [7]. Breuer, Nikhil and Poulovassilis have all described functional database systems and also recommend list comprehensions as a powerful notation for expressing queries [6, 17, 19]. Transformation methods are well developed in both the functional programming community [5, 8, 10] and the database community [15, 20, 26].

Freytag [12] has shown how to transform functional notations to improve a query evaluation plan. The query evaluation plan is the output of an algebraic relational optimiser, so this is at a lower-level than our transformations, which perform improvements equivalent to the algebraic improvements. He is able to minimise the number of traversals of the data, and the number of conditional expressions.

The remainder of this paper is structured as follows. Section 2 introduces list comprehensions. Section 3 outlines the assumptions under which the queries are improved. Section 4 describes the algebraic improvements. Section 5 describes the improvements gained using information about the implementation. Section 6 concludes.

2 List Comprehensions

For brevity list comprehension, or ZF notation is not formally described, merely illustrated by example. A full description is given in [23]. A set comprehension describing the set of squares of all the odd numbers in a set A can be written

$$\{square\ x \mid x \in A \wedge odd\ x\}$$

and has a corresponding list comprehension

$$[square\ x \mid x \leftarrow A;\ odd\ x].$$

This can be read as 'the list of squares of x such that x is drawn from A and x is odd'. The syntax of comprehensions can be sketched as

$$\begin{aligned} comp &::= [e \mid q] \\ q &::= q; q \mid f \mid p \leftarrow e \end{aligned}$$

Here e is an expression in the language. The expressions to the right of the vertical bar are called qualifiers. Qualifiers are either filters or generators. Filters, f, are boolean-valued expressions that specify the conditions the variables must satisfy to be included in the result, e.g. $odd\ x$.

Generators have the form $p \leftarrow e$, where p is a pattern which introduces one or more new variables and e is a list-valued expression which indicates the range of values they may take on, e.g. $x \leftarrow A$. For database queries the most common pattern is a tuple of variables $(a_0, .., a_n)$, and we use $\overline{a}$ to denote these.

The following database query examines a list *orders* and returns the order numbers of orders placed by Brooks,

$$\begin{aligned} &[order\_no \mid (order\_no, name, item, qty) \leftarrow orders; \\ &\qquad name = \text{`}Brooks, B.\text{'}]. \end{aligned}$$

Our final example constructs a join of two binary relations,

$$[(a,b,d) \mid (a,b) \leftarrow AB;\ (c,d) \leftarrow CD;\ b = c].$$

If $AB = [(\text{`a'}, 1), (\text{`b'}, 4)]$ and $CD = [(1, \text{`z'}), (3, \text{`x'}), (4, \text{`y'})]$, then the comprehension's value is

$$[(\text{`a'}, 1, \text{`z'}), (\text{`b'}, 4, \text{`y'})].$$

Our final comment on comprehensions is to note that they are already optimal in the sense that they perform the minimum number of *cons* operations required to build the result list [23].

3 Improvement Environment

Queries are improved under certain assumptions about the environment in which they will be evaluated. Because this work spans the database and programming language worlds, assumptions are made about both.

A programming language assumption is that the queries are evaluated under a lazy regime. We also assume that the lists processed by the comprehensions represent relations which have been retrieved from secondary, or permanent storage.

The remaining assumptions concern the underlying database. They are typical of those found in conventional improvers. Further, the functional database we describe in [22] provides the functionality we assume.

We assume that permanent storage is provided by disks. Disks store data as *blocks* or convenient-sized chunks. The *size* of a list is the number of blocks in it. Each disk access retrieves a block. Thus traversing a list will require a number of disk accesses proportional to its size.

The number of disk accesses required to evaluate a query is the cost metric. This is because the time required for an access is typically three or four orders of magnitude greater than the time to execute a machine instruction. Hence we are prepared to perform considerable additional computation if it will reduce the number of accesses required.

We assume that the implementation supports caching. This is an efficiency technique where some memory, called a *cache*, is allocated to retain values recently retrieved from disk. If a program needs a value which is in the cache, it can be read from memory, and no disk access is required.

We assume that the implementation supports indices. An index on a relation typically consists of a sorted tree of values. The tree can be searched for a value in time proportional to the logarithm of the size of the relation. We assume that information such as the size of the relations and the nature of the indices is available.

We assume that the order of the tuples in the queries result is not significant. This is consistent with the relational model, and means that we sometimes use *bag equality* on lists. Two lists, l_0 and l_1, are bag equal, i.e. $l_0 \cong l_1$, if they contain the same elements, although possibly in different orders.

Because disk access is central to this cost model, not all of the transformations we present in the following sections will improve comprehensions that do not perform disk accesses. Finally, we note that the conventional improvements we are emulating are not guaranteed to be optimal for all possible instances of the database.

4 Algebraic Improvements

This section contains subsections describing each of the conventional algebraic improvements and how an equivalent list comprehension improvement is obtained. The description of conventional improvement techniques is closely based on that given by Ullman [25].

Most of the improvements are obtained using transformations which are analogous to identities in the relational algebra. The transformations we describe are examples of classes of transformations. We do not describe all members of these classes.

4.1 Selections

Performing selection as early as possible is the most important improvement. It reduces the size of the intermediate results by discarding tuples which are not required. Ullman illustrates this with a query which prints the A components of those tuples in the AB and CD relations that have the same values in B and C, and have a D value of 99. This may be expressed in the relational algebra as $\Pi_A(\sigma_{B=C \wedge D=99}(AB \times CD))$. It can also be written as the comprehension

$$[a \mid (a,b) \leftarrow AB;\ (c,d) \leftarrow CD;\ b=c;\ d=99].$$

To produce queries that perform selections earlier we introduce the following transformation.

Qualifier Interchange allows us to swap any two qualifiers q and q', if they don't refer to variables bound in each other. Using $\cong$ to denote bag equality, it may be stated

$$\begin{aligned} & [e \mid q_0;\ q;\ q';\ q_1] \\ \cong\ & \\ & [e \mid q_0;\ q';\ q;\ q_1]. \end{aligned}$$

Rewriting programs so that selection occurs as soon as possible is a well-known program transformation strategy called *filter promotion* [10]. Qualifier interchange is a generalisation of filter promotion as it allows us to change the order of generation as well as the order of filtration. This generality is also reflected by the fact that qualifier interchange is analogous to several relational algebra identities. These are the identities governing the commuting of products and selections. □

In our example the filters are '$b=c$' and '$d=99$'. We cannot promote '$b=c$' over '$(c,d) \leftarrow CD$' because '$b=c$' refers to c which is bound in '$(c,d) \leftarrow CD$'. We can, however, interchange '$d=99$' and '$b=c$', giving

$$[a \mid (a,b) \leftarrow AB;\ (c,d) \leftarrow CD;\ d=99;\ b=c].$$

Now, as '$(a,b) \leftarrow AB$' doesn't bind c or d, we can promote '$(c,d) \leftarrow CD;\ d=99$' over it, giving

$$[a \mid (c,d) \leftarrow CD;\ d=99;\ (a,b) \leftarrow AB;\ b=c].$$

This is considerably more efficient than the original query. If the size of AB and CD is n, then the time complexity of the original query is $O(n^2)$. Usually the number of tuples with d value 99 is much smaller than n. If we assume that it is a small constant, i.e. independant of n, the new query is $O(n)$.

4.2 Converting Product into Join

If we combine certain selections with a prior cartesian product to make a join, performance is improved. This is because the cost of a cartesian product of two relations of size n is of $O(n^2)$, whereas the cost of a join, such as a natural join, is of $O(n \log n)$. Before we can convert the product in our example query we must manipulate it into a suitable form, for which we use the following transformation.

Filter Hiding. If f_a is a filter involving only variables in A, then

$$\begin{aligned} & [e \mid q_0;\ \overline{a} \leftarrow A;\ f_a;\ q_1] \\ =\ & \\ & [e \mid q_0;\ \overline{a} \leftarrow A';\ q_1] \\ & \quad \text{where} \\ & \qquad A' = [\overline{a} \mid \overline{a} \leftarrow A;\ f_a]. \end{aligned}$$

Even although the tuples satisfying f_a are drawn from both A and A', lazy evaluation ensures that they are only read from secondary storage once. This is because, when a demand is made for an element of A', the demand is propagated immediately to a demand for an element of A satsfying f_a. The element of A may need to be retrieved from secondary storage, but once this is done, it can be passed directly to the expression demanding an element of A'. □

Applying filter hiding to '$(c,d) \leftarrow CD;\ d=99$' in our example query we obtain

$$\begin{aligned} & [a \mid (c,d) \leftarrow CD';\ (a,b) \leftarrow AB;\ b=c] \\ & \quad \text{where} \\ & \qquad CD' = [(c,d) \mid (c,d) \leftarrow CD;\ d=99]. \end{aligned}$$

In this instance, filter hiding has neither improved nor degraded the query. We have, however, manipulated it into a form in which the cartesian product can be converted into a join. The transformation to do this is described next.

Product Elimination converts a cartesian product followed by an equality test into a natural join. It is the most common member of a class of transformations, which generate the different relational joins.

Recalling our example in section 2, a natural join takes two relations of arity r and s and constructs a new relation of arity $r+s-1$, i.e. with one of the identical columns eliminated. To reflect this in the following definition, b_j represents the eliminated column, and we write $\overline{ab}$ for $(a_0, .., a_n, b_0, .., b_{j-1}, b_{j+1}, .., b_m)$. Any reference to the eliminated column must also be relaced by a reference to the identical column, a_i. The substitution of a_i for b_j is written $e[a_i/b_j]$. Product elimintation can then be stated

$$\begin{array}{l} \quad [e \mid q_0;\ \overline{a} \leftarrow A;\ \overline{b} \leftarrow B;\ a_i = b_j;\ q_1] \\ = \\ \quad [e[a_i/b_j] \mid q_0;\ \overline{ab} \leftarrow AB;\ q_1[a_i/b_j]] \\ \qquad \text{where} \\ \qquad\qquad AB = jmerge_{ij}\ (sort_i\ A)\ (sort_j\ B). \end{array}$$

The $jmerge_{ij}$ function is defined as

$$\begin{array}{l} jmerge_{ij}\ []\ B = [] \\ jmerge_{ij}\ A\ [] = [] \\ jmerge_{ij}\ (\overline{a} : A)(\overline{b} : B) \\ \quad = \overline{ab} : jmerge_{ij}\ (\overline{a} : A)\ B, \text{if } a_i = b_j \\ \quad = jmerge_{ij}\ (\overline{a} : A)\ B, \text{if } a_i > b_j \\ \quad = jmerge_{ij}\ A\ (\overline{b} : B), \text{if } a_i < b_j. \end{array}$$

The transformation introduces a sort-merge to compute the join. Alternative algorithms could also be used. The $sort_i$ functions sort on ith component of the relation. To resolve the typing problems raised by such joins we rely upon some existing solution. □

Applying it to our example we obtain

$$\begin{array}{l} [a \mid (c,d,a) \leftarrow CDAB] \\ \quad \text{where} \\ \qquad CDAB = jmerge_{12}\ (sort_1\ CD')\ (sort_2\ AB) \\ \qquad CD' = [(c,d) \mid (c,d) \leftarrow CD;\ d = 99]. \end{array}$$

This has complexity of $O(n \log n)$, because AB is still of size n. The desirability of performing this transformation on our example depends on the ratio between $\log n$ and the number of tuples in CD having a d value of 99. Also note how the transformations have taken a simple query and produced a more efficient, but more complex query.

4.3 Combination of Unary Operations

In a naive processor, we may traverse a relation for each selection or projection encountered in a query. Efficiency is improved if a sequence of selections and projections can be evaluated in a single pass over a relation. Buneman, Frankel and Nikhil have shown that lazy evaluation causes this to occur automatically in functional query languages [7].

To illustrate the automatic combination, consider one of the examples from section 2,

$$\begin{array}{l} [order\_no \mid (order\_no, name, item, qty) \leftarrow orders; \\ \qquad name = \text{`}Brooks, B.\text{'}]. \end{array}$$

This performs a select on name, and a project on order number. Demand for the next order number is propagated to a demand for a order tuple with name Brooks. The next tuple in *orders* is obtained, possibly from secondary storage. If the name is Brooks, the corresponding order number can be returned immediately. If the name is not Brooks, we need to examine the next tuple in orders. The significant point is that the tuple is retrieved only once. The name test and the projection onto order number both occur while the tuple is in primary storage.

4.4 Common Subexpressions

It is clearly advantageous to compute only once a result which will be used many times. This is in fact what happens in a functional language with list comprehensions. If e_x is an expression referring to x more than once, and A is the relation produced by some complex computation, then $[e_x \mid x \leftarrow A]$ retains those parts of A which have been realised for some reference to x for as long as there is a reference to them. This is called sharing [18].

We can also use *let* or *where* expressions to preserve common subexpressions, even between comprehensions. For example, consider the improvement of a query which computes the difference between two projections of a join. Using $\bowtie$ to denote join, this can be specified in the algebra as $\Pi_i(A \bowtie B)$ —$\Pi_j(A \bowtie B)$. Writing — for list difference, we might express this as

$$\begin{array}{l} [a_i \mid \overline{a} \leftarrow A;\ \overline{b} \leftarrow B;\ a_k = b_l]\ — \\ [a_j \mid \overline{a} \leftarrow A;\ \overline{b} \leftarrow B;\ a_k = b_l]. \end{array}$$

Using product elimination we obtain

$$\begin{array}{l} ([a_i \mid \overline{ab} \leftarrow AB] \text{ where } AB = jmerge_{kl}\ (sort_k\ A)(sort_l\ B)) \\ — \\ ([a_j \mid \overline{ab} \leftarrow AB] \text{ where } AB = jmerge_{kl}\ (sort_k\ A)(sort_l\ B)). \end{array}$$

If we assume that the result of the join is of size n, then it is also reasonable to assume that computing the difference between the projections on the join costs $n \log n$.

Under these assumptions the cost of the above query is $3n \log n$, as we perform the join twice before computing the difference. We can eliminate the redundant join by using the definitions of where and substitution, to obtain

$$[a_i \mid \overline{ab} \leftarrow AB] - [a_j \mid \overline{ab} \leftarrow AB]$$
$$\text{where}$$
$$AB = jmerge_{kl}\ (sort_k\ A)(sort_l\ B)).$$

The left-hand comprehension costs $n \log n$ accesses as it constructs and consumes the join simultaneously. The right-hand comprehension need only traverse the result of the join, which costs n accesses. Finally, the difference also costs $n \log n$, giving a total cost of $2n \log n + n$ accesses.

5 Implementation-based Improvements

Improvements based on implementation information are particularly useful for improving queries that are more complex than those permitted by the relational algebra. The extra-relational parts of these queries are not amenable to relational algebra transformations. The only related work the authors know of is by Freytag [12], who shows how to improve aggregate queries.

5.1 Preprocessing Files

The most important file processing ideas are sorting and the creation of indices. The product elimination transformation illustrated the introduction of sorting. We have described elsewhere a functional database system which supports indices [22]. A transformation which allows an index to be used is presented below.

Index introduction. If e' is an expression, and there is an index $jindex_A$ on an attribute a_j, then

$$\begin{aligned} & [e \mid q_0;\ \overline{a} \leftarrow A;\ a_j = e';\ q_1] \\ = \\ & [e \mid q_0;\ \overline{a} \leftarrow jindex_A\ e';\ q_1].\ \square \end{aligned}$$

In our example from subsection 4.1,

$$[a \mid (c,d) \leftarrow CD;\ d = 99;\ (a,b) \leftarrow AB;\ b = c]$$

we can apply index indroduction to '$(c,d) \leftarrow CD;\ d = 99$', to obtain

$$[a \mid (c,d) \leftarrow dindex_{CD}\ 99;\ (a,b) \leftarrow AB;\ b = c].$$

A second application gives

$$[a \mid (c,d) \leftarrow dindex_{CD}\ 99;\ (a,b) \leftarrow bindex_{AB}\ c].$$

Another file processing example Ullman gives is an efficiency improvement for cartesian products. With caching, efficiency is improved by choosing the smaller relation to vary more slowly, or be in the 'outer loop'. This is because, if the records of the smaller relation are cached, then, as they are combined with each record in the larger relation, they are used more often.

To make the improvement, we simply promote the smaller relations generator using qualifier interchange. This is another example of the general power of qualifier interchange. If L is the larger relation, S is the smaller, and we write $\bar{ls}$ for $(l_0, .. , l_n, s_0, .. , s_m)$, then

$$[\bar{ls} \mid \bar{l} \leftarrow L;\ \bar{s} \leftarrow S]$$

becomes

$$[\bar{ls} \mid \bar{s} \leftarrow S;\ \bar{l} \leftarrow L].$$

5.2 Evaluating Options

It is often possible to compute a result in more than one way, either by reordering operations or by treating the operands of a binary operator differently. Time spent evaluating these options is usually much less than the time spent evaluating the query in an inferior way. Usually the cost of a large number of alternatives is considered, and the best of these is selected. As an example Ullman presents the options evaluated for simple selections in System R [2]. These have the form

SELECT A_1, ... A_n
FROM R
WHERE P_1 AND ... P_n.

The equivalent list comprehension form is $[\overline{a} \mid \overline{a} \leftarrow R;\ P_1;\ ...\ P_n]$.

To compare the evaluation options the system uses the following information.

- T, the number of tuples in R.
- B, the number of blocks in R.
- I, if there is an index on attribute a_j, the image size I is the number of different values of a_j in R.
- Whether or not an index is clustering. A clustering index is an index on an attribute such that tuples with the same value for that attribute tend to fill blocks entirely.

A predicate of the form '$a_j = c$', where a_j is an attribute and c a constant is said to *match* an index on a_j. Ullmans example query prints the order numbers of any orders for more than 5 pounds of Granola. A list comprehension expressing this is

$$[order\_no \mid (order\_no, name, item, qty) \leftarrow orders; \quad qty \geq 5; \ item = \text{`}Granola\text{'}].$$

The database parameters are that $T = 1000$, $B = 100$, there is a clustering index on '*name*', and a nonclustering indices on '*item*' ($I = 50$) and '*quantity*'. The alternatives in the System R algorithm which are relevant for our storage method are:

1. Get those tuples of R which satisfy a predicate of the form '$a_i = c$' that match a clustering index. Then apply the remaining predicates. This costs B/I block accesses, in our example, if the '*item*' index was clustering, this would be 2 accesses. It cannot be applied as the item index, which is the only index matched by an equality predicate ($item =$'*Granola*'), is not clustering.
2. Use a clustering index on a_i, where '$a_i \ \omega \ c$' is a predicate, and ω is $<, \leq, \geq$ or $>$ to obtain the subset of R that satisfies this predicate, then apply the remaining predicates. This costs $B/2$ block accesses, or 50 block accesses in our example. It cannot be applied as the *quantity* index, which is the only index matched by an inequality predicate ($qty \geq 5$), is not clustering.
3. Use a non-clustering index that matches a predicate '$a_i = c$' to find all of the tuples with a_i value c, and apply the other predicates to these tuples. This costs T/I, or 20 accesses. The item index ($anitemsorders$) fulfills these conditions.
4. Read all of the tuples of R and apply the predicates to each of them. This costs B block accesses, which is 100 in our example. It corresponds to the first comprehension in this section.

In this case we would select option 3. To introduce the index we must first juxtapose the enumeration of orders, '$(order\_no, name, item, qty) \leftarrow orders$', and the selection on the item ordered, '$item =$'*Granola*''. We do this by promoting '$qty \geq 5$' over '$item =$'*Granola*'', using qualifier interchange, giving us

$$[order\_no \mid (order\_no, name, item, qty) \leftarrow orders; \quad item = \text{`}Granola\text{'}; \ qty \geq 5].$$

Index introduction now allows us to use *anitemsorders*, giving

$$[order\_no \mid (order\_no, name, item, qty) \leftarrow anitemsorders \quad \text{`}Granola\text{'}; \ qty \geq 5].$$

6 Conclusion

For each query improvement strategy that Ullman identifies an equivalent list comprehension improvement has been demonstrated. We have noted that database improvement algorithms that use these strategies can be emulated to improve comprehension queries. We have also noted that these transformations can be used to improve queries that require greater power than the relational algebra provides.

Several extensions suggest themselves. We might replace the informal arguments made about when a query requires a disk access by a calculus of programs. Just as we proved the relational completeness of list comprehension notation, a similar completeness property for the improvement transformations is desirable. In other words, we would like to show that, not only can we emulate the major algebraic improvements, we can perform an equivalent transformation for every algebraic identity.

Acknowledgements

We would like to thank David Harper and John Hughes for provoking useful discussions and for their constructive criticisms. We are particularly grateful to David for stimulating our interest in this area.

References

[1] G. Argo, J. Fairbairn, R.J.M. Hughes, E.J. Launchbury and P.W. Trinder, "Implementing Functional Databases," Proceedings of the Workshop on Database Programming Languages, Roscoff, France, September 1987.

[2] M.M. Astrahan et al, " System R: Relational Approach to Database Management," ACM Transactions on Database Systems, vol. 1 n.o. 2, June 1976.

[3] M.P. Atkinson et al, "P.S. Algol Reference Manual 2nd Ed," University of Glasgow Computing Science PPR Report 12, 1985.

[4] M.P. Atkinson and O.P. Buneman, "Types and Persistence in Database Programming Languages," ACM Computing Surveys, vol. 19 n.o. 2, June 1987, pp. 105-190.

[5] R.S. Bird, "The Promotion and Accumulation Strategies in Transformational Programming," ACM Transactions on Programming Languages and Systems, vol. 6 n.o. 4, October 1984, pp. 487-505.

[6] P.T. Breuer, "A Variety of Functional Programming Sublanguages," Tech. Rep. CUED/F-INFENG/TR26 Engineering Deptartment, University of Cambridge, 1988.

[7] O.P. Buneman, R. Nikhil and R. Frankel, "An Implementation Technique for Database Query Lan-

guages," ACM Transactions on Database Systems, vol. 7 n.o. 2, June 1982, pp. 164-187.

[8] K.L. Clark and J. Darlington, "Algorithm Classification through Synthesis," The Computer Journal, vol. 23 n.o. 1, 1979, pp. 61-65.

[9] E.F. Codd, "Relational Completeness of Database Sublanguages," Database Systems: Courant Computer Science Series Vol 6, Englewood Cliffs, New Jersey, Prentice Hall, 1972.

[10] J. Darlington, "A Synthesis of Several Sorting Programs," Acta Informatica, vol. 11 n.o. 1, January 1978, pp. 1-30.

[11] C.J. Date, An Introduction to Database Systems (3rd Ed). Addison Wesley, 1976.

[12] J.C. Freytag, "Translating Relational Queries into Iterative Programs," Ph.D. Thesis Harvard University, September 1985.

[13] G.D. Held, M.R. Stonebraker and E. Wong, "Ingress a Relational Database System," in Proceedings of the National Computer Conference 44, May 1975.

[14] R.J.M. Hughes, "Lazy Memo Functions", in Proceedings of the 2nd International Conference on Functional Programming Languages and Computer Architecture Nancy, France, September 1985, Springer Verlag LNCS 201, 1985.

[15] W. Kim, "On Optimising an SQL-like Nested Query," ACM Transactions on Database Systems, vol. 7 n.o. 3, September 1982, pp. 443-469.

[16] W. Luk, "Parameterised Design for Regular Processor Arrays," D.Phil Thesis, Oxford University, 1989.

[17] R. Nikhil, "Semantics of Update in a FDBPL," in Proceedings of the Workshop on Database Programming Languages, Roscoff, France, September 1987, pp. 365-383.

[18] S.L. Peyton Jones The Implementation of Functional Programming Languages. Prentice Hall, 1987.

[19] A.P. Poulovassilis, "FDL: An Integration of the Functional Data Model and the Functional Computational Model," Proceedings of the 6th British National Conference on Databases, July 1988, pp.215-236.

[20] J.M. Smith, "Optimising the Performance of a Relational Algebra Database Interface," Communications of the ACM, vol. 18 n.o. 10, October 1975, pp. 568-579.

[21] P.W. Trinder and P.L. Wadler, "List Comprehensions and the Relational Calculus," in Proceedings of the 1988 Glasgow Workshop on Functional Programming, Rothesay, Scotland, August 1988, pp. 115-123.

[22] P.W. Trinder, "Functional Databases," D.Phil Thesis in preparation, Oxford University, 1989.

[23] P.L. Wadler, "List Comprehensions," Chapter 7 of S.L. Peyton Jones The Implementation of Functional Programming Languages. Prentice Hall, 1987.

[24] S.C. Wray, "Implementation and Programming Techniques for Functional Languages," PhD. Thesis, Cambridge University Computing Laboratory TR92, June 1986.

[25] J.D. Ullman Principles of Database Systems, Pitman, 1980.

[26] H.Z. Yang and P.Å. Larson, "Query Transformation for PSJ-queries," in Proceedings of the 13th Int. Conf. on Very Large Databases Brighton, 1987, pp. 245-254.

[27] M.M. Zloof Query By Example. in Proceeding of the National Computer Conference 44, May 1975.