On the Complexity of Nested Relational Operations

Marc H. Scholl
Institute of Information Systems, ETH Zurich
CH-8092 Zurich, Switzerland
e-mail: scholl@inf.ethz.ch

1 Introduction

Designing and implementing a query optimizer and query processor for a nested relational database system (DBMS) requires a number of steps like deciding on operations supported on nested relations (i.e. the semantics of a query language (QL)), defining a syntax (i.e. a concrete QL for the system), investigating equivalences among expressions of the QL, devising optimization strategies or rules, and finding efficient evaluation techniques for single operations of the QL and combinations thereof. While these activities are rather theoretical in nature, they obviously have tremendous impacts on the architecture and implementation of the corresponding DBMS. We have presented powerful operations on nested relations in an algebraic syntax in [SS86], investigated equivalence rules in this algebra in [Sch86, SPS87], and described the implementation of an efficient query evaluation algorithm in [DPS86, PSS+87]. Here we focus on the following problem: typical query evaluation techniques distinguish between "simple" and "complex" operations in the QL (like selections and projections vs. joins or sorting in the relational setting). The simple operations are executed "on the fly" in a rather low level of the DBMS architecture while more complex ones are processed on top of that layer. Classical examples of such architecures are System R (RDS on top of RSS [ABC+76, CAB+81]), Ingres (MVQP on top of OVQP [SWKH76]), LOLEPOPs in Starburst are a more recent example of simple operations [Loh88].

The idea behind such a dichotomy in query processing strategies is that (combinations of) simple operations can be computed while scanning data, hence "on the fly", we will call them "*single scan operations*" in the sequel. Intuitively, we read in blocks from the disk, inspect tuples sequentially and compute the result of such operations without touching data more than once. Formally, these operations have linear complexity in the size of the involved relations. Other operations require inspecting single data elements more than once (like sorting, or joins with nested loops). The exact classification of operations into these two classes is crucial as to avoid unnecessary scans of the input data, particularly as this usually means reading in data from secondary storage.

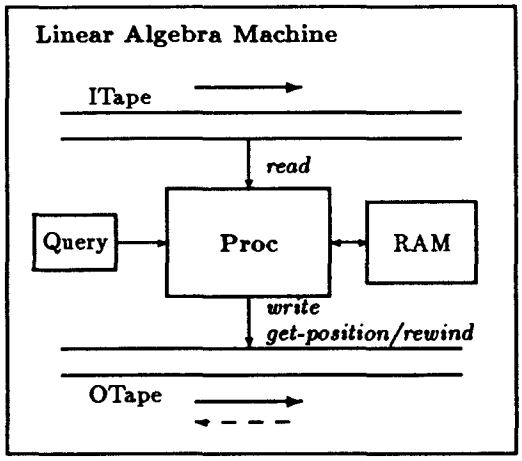
Among the relational operations, clearly selection is a single scan operation: while scanning tuples of the input relation sequentially, we can test the selection condition on each single tuple and decide whether it belongs to the result or not. Projections are only linear if we ignore duplicate elimination. Obviously, joins are not linear in general. Therefore, we find selections and projections in the lower level query processors (RSS, OVQP) and joins (together with the other non-linear operations) in the higher level query processors (RDS, MVQP) in typical relational DBMS architectures. In the flat relational context, the distinction seems to be quite evident. However, when switching to nested relations with corresponding new styles of operations things need a little more elaboration. Selection conditions may involve set comparisons, projections and selections may involve complex operations (like joins) on deeper levels of the nesting hierarchy[1]. New operations are added: nesting and unnesting. Deciding whether such operations are single scan processible or not is non-trivial in some cases.

We present a model of query processing that explicates our intuition about single scan operations. Using this model we analyse the operations individually and consider *expressions*, i.e. compositions of operations. In designing such a model we have to be very careful, as can be seen from the following simple example. Our intuitive understanding is, of course, that join is no single scan operation, however, consider the following algorithm:

```
begin
for each r ∈ R do
   read(r) into Temp
for each s ∈ S do
   { add Temp⋈s to the result }
end.
```

Obviously, the algorithm computes the correct result without "touching" (i.e. reading in) stored tuples more than once. However, we used intermediate storage space in this example, and accessed it multiply. So, what is the point? Should we prohibit any intermediate storage? The answer is negative: we will need *some* additional storage. Consider a nested relational selection (on the famous Departments and Employees

[1] We use our terminology of a nested relational algebra [SS86]. However, all investigations and results apply for other QLs too, provided they support the types of operations we consider.



(a) LAM-Architecture

```
while not { End of input }
do read (d(d#, dn, dl))
   temp := get-position (OTape)
   write (d(d#, dn, dl))
   found := false
   while not { End of subrelation }
   do read (e(e#, en, s))
      if e(en) = 'Smith'
        then found := true
      fi
      write (e(e#, en, s))
   od
   if not found
      then rewind (OTape,temp)
   fi
od
mark_end-of-tape (OTape)
```

(b) A sample program

Figure 1: The abstract linear algebra machine

–nested– relation) asking for departments that have an employee named 'Smith'. The employees working for a particular department are stored as a 'subrelation'. Hence, assuming a hierarchical pre-order scan of nested tuples we have to somewhere store department information (e.g. name and location) until we finished the (subordinate scan) of all employees. Only then can we decide whether the department matches the selection condition. So, as we want to consider such a nested selection a single scan operation, we in fact need some intermediate storage. Thus, to avoid the above problem with the join operation, we have to limit the size of additional storage used in an algorithm in order to consider it single scan.

We present our abstract model of a nested relational query processing engine for linear operations in section 2 and give our results concerning linearity of operations in section 3.

2 An Abstract Algebra Machine

Input data may be read only once, thus we use an *input (Turing) tape* that is automatically moved forward while reading data into the processor's memory. Tuples of the input (nested) relation[2] are stored in a pre-order linearization on the input tape. The units of transfer between the tape and the processor memory are so-called "atomic fragments", i.e. all atomic values of a single level of the nesting hierarchy are read in one step. E.g. in our department and employees example, there would be one fragment for each department followed by the fragments describing the individual employees (of this department), followed by the next department and so on[3]. Data is read in and used for computing results by the *processor*, which eventually writes result tuples to an *output (Turing) tape*. The ouput tape cannot be read, however, we allow rewinding the output tape back to a position that has formerly been recorded in the processors memory. This allows for dropping (department) tuples that have been considered candidate matches to a query but finally do not qualify, for instance (see the nested selection example above). Once rewinded, the content of the output tape behind the new position is lost, hence the output tape behaves like a stack with "push" and "pop" but no "top" operation.

The *processor* can issue "read" commands to the input tape, "write", "get position", and "rewind" operations to the output tape, and checks conditions on the input. It gets the query to perform on the input in some convenient form, e.g. as an operator tree. Obviously, we need at least enough memory to hold the query tree and the data units transfered by one read operation on the input tape. The sizes of these are determined by the query or the schema[4], it is important to notice that the size is independent from the extension of the database! We already mentioned that we spend some more memory to 'remember' output tape positions, the state of subconditions in complex selections and some other information. Again, we do not allow the size of that additional memory to depend on the *extension* of the relation (it may well depend on its *intention*, i.e. schema). The algebra machine can be thought of as shown in figure 1(a), where we included the set of operations valid on the two tapes. Our criterion for an operation to be single scan processible is

[2] only monadic operators will be linear, see below

[3] this exactly matches the implementation in DASDBS[DGW85, DPS86]

[4] the maximum lengths of all atomic attributes of the fragments, an information that is usually found in the DB catalogs

Definition 1 *An operation on nested relations is called* single scan processible *(linear), iff it can be computed by the abstract algebra machine with an amount of memory independent from the state of the database.*

As an example, we give a "program" which run on the algebra machine computes the nested selection mentioned above. On $Dept(dno, dname, dloc, Empl(eno, ename, sal))$ the query, written in our algebra is:

$$\sigma[\ \sigma[ename = \text{'Smith'}](Empl) \neq \emptyset\ ](Dept)$$

(notice that *all* employees are obtained from qualifying departments). A possible computation of the result using the algebra machine is shown in figure reffig-mach(b). Here we used two local variables (i.e. additional memory to store them): 'found' is a boolean flag set when the nested selection hits, and 'temp' is used to store a position of the output tape, viz. the one before the current output tuple. If a department fails the selection, we remove it by simply rewinding the tape and overwriting with either the next department or an end-of-tape mark.

3 Results

Now we look at the individual operations on nested relations and combinations of them and state whether they are single pass processible, under what conditions they are, or not.

Theorem 1 *The dyadic operations of the algebra (union, difference, product) and the nest operation are not single scan processible. So are the derived ones like intersection, join, division, etc.*

We do not give the detailed proofs here, they can be found in [Sch88]. Union might be considered a linear operation if we omit duplicate elimination as with projections. Otherwise it requires sorting like difference, intersection and nesting do, hence they are not linear. As input relations for a product can only be read one after the other from the input tape, we would need storage to hold the first relation when scanning the second, thus product is not linear.

Now, the interesting cases are (nested) selections and projections, with their variety of nested subexpressions, and unnesting. Let us begin with projections. In our algebra, projection is used to eliminate some columns of tables, but moreover, we use projection as one of the entry points for nested expressions. For instance, we can project (from the department relation) the employee subrelation as it is stored, but also the result of an arbitrary algebraic expression applied to it. This way we can obtain a new subrelation holding e.g. only the employees named Smith, or the join of the employee subrelation with a children relation, and the like[5]. Therefore, not all projections can be linear. This is the reason why we use the term single pass rather than the classical single table operation. One possible restriction is that only *linear* expressions may be nested inside a projection, but we need others. Consider the fact that more than one expression may be applied to a subrelation to construct several new subrelations. If all of these are linear, they can be computed in parallel while scanning the input, however, as subrelations are written one after the other on the output tape (as on the input), we would have to collect the results of all but the first nested subexpression until the first is written onto the output tape completely. Hence, what can be computed in a single scan is

Theorem 2 *A (nested) projection is linear, iff at most one subexpression is applied to each subrelation and all of these are linear.*

As subrelations come one after the other on the input (and output) tape, we can compute the nested subexpressions one after the other. If they are all linear, we need $\max_i(N_i)$ units of additional memory, where N_i is the requirement of nested subexpression i.

Obviously selections with standard "1NF-style" conditions are linear. However, in the nested case we have set comparisons and nested subexpressions. This means we can select tuples based on the (set) comparison between the results of two nested subexpressions, e.g. departments where the set of all employees named Smith equals (or not equals ...) the empty set, or (equals or not equals or) contains ... the set of employees making more than 50k.[6] Selection in our algebra is a second entry point for nesting of expressions. A first observation is that comparing two arbitrary sets on equality or containment is not a single scan operation (it introduces join complexity). So the only possible set comparisons in linear selections are those between a set (expression or subrelation) and a (set) *constant*, i.e. a set whose value is explicitely given in the query formulation, e.g. the empty set $\emptyset$. Then the space needed to hold the value of one of the two sets is given in the query tree!

Theorem 3 *A (nested) selection is linear, iff it is built according to the following rules:*

- *arbitrary comparisons on atomic attributes,*

[5] $\pi[\ldots, New := \sigma[ename = \ldots](Empl)](Dept)$ or $\pi[\ldots, New := Empl \bowtie Children](Dept)$

[6] we already had the first query, the second one looks like $\sigma[\ \sigma[ename = \ldots](Empl) \supseteq \sigma[sal > \ldots](Empl)\ ](Dept)$

- *element tests ($\in$) on subrelations (or nested subexpressions)*
- *set comparisons between constants and subrelations (or nested subexpressions),*
- *logical conectives of these ($\wedge, \vee, \neg$),*

where all nested subexpressions are linear.

Interestingly, we find no restriction on the number of nested subexpressions per subrelation as we had for projections. This is due to the fact that we can compute several linear operations in parallel as long as we do not have to store the results of them (cf. projections above). For selections it is sufficient to keep a boolean flag indicating the current truth-value of every subcondition. We need to remember only one position of the output tape (for the outermost nested selection).

Before we look at the unnest operation, we analyse the use of so-called *dynamic constants* [SS86], i.e. values of higher level attributes inside a deeply nested subexpression. As an example, consider our department relation and assume that we have the employee number of the department's manager (*mno*) as an atomic value. Then $\pi[dname, \pi[ename](\sigma[eno = mno](Empl))](Dept)$ retrieves a list of department names together with (as a subrelation) the names of their managers. The value of *mno* is constant w.r.t. the subrelation *Empl*, thus we could use it in the nested selection condition. We can permit dynamic constants in linear operations provided that the values of these constants do not require storage space depending on the extension. As this is only the case with set (i.e. relation) valued attributes, we have

Theorem 4 *The use of atomic dynamic constants is permitted in linear selections and projections.*

For a given query representation we can determine in advance which attribute values are needed in deeper levels of the nesting hierarchy and save their values in some intermediate storage. The size of that storage depends only on the query and the schema (i.e. the maximal attribute length).

Dynamic constants are also the key to the unnest operation: unnesting means reducing the hierarchical structure of the input relation by repeating attribute values with all subtuples of the unnested subrelation.

Theorem 5 *Unnesting a subrelation S of a relation R is a linear operation, iff S is the only subrelation of R (on that level).*

The values of all atomic attributes of R can be held as dynamic constants for the scan of S and output together with the attributes of S repeatedly. We need additional memory to store one "atomic fragment" of R-tuples. A second subrelation would require extension-dependent storage space.

While selections and projections need extra memory for tape positions and boolean flags for parts of selection conditions, i.e. in an amount primarily determined by the query, dynamic constants and unnesting require space depending on the schema (the domains of the atomic attributes) of the involved relations. Thus, unnesting and dynamic constants are linear operations w.r.t. a slightly more sophisticated notion of linearity.

Finally, we are interested in combinations (compositions) of linear operations. If we use a query language where the model is closed under its operations, we can construct complex expressions by applying more operations to the result of previous operations. In such a sequence of operations, if all operations are linear, is the combination linear too? We know from classical query optimization that given two subsequent operations we can either (i) compute the first one and obtain an intermediate result on which we apply the second operation subsequently, or (ii) compute both of them in a "pipelined" mode, apply the first operation to the first input tuple and feed the result directly to the second operation (see e.g. [JK84]). The second alternative is more efficient in general since we avoid storage of potentially large intermediate results and multiple scans. Therefore, we should be able to combine several linear operations into one linear expression and compute it in pipelining mode.

Theorem 6 *Any composition of linear operations yields a linear operation.*

While we can combine linear operations in an arbitrary way due to the above result, we are interested in a "canonical linear expression" in order to develop guidelines for the implementation of a low-level (i.e. linear) nested relational query processor and for an optimizer. Using algebraic equivalence rules on our nested algebra we can apply transformations similar to the cascading of selections and projections, commutation of projections and selections, and so on [Sch86]. If we only consider (nested) selections and projections, such a canonical form is:

$$\sigma[F_2](\pi[L](\sigma[F_1](R))),$$

where F_1 and L may contain nested linear subexpessions of this type in turn, F_2 is a selection condition which refers to the results of expressions in L (otherwise it could have been commuted with the projection). F_1 and L may have several nested subexpressions on the same subrelation, however, only one of them may occur in L. Typical examples of conditions found in F_2 are of the form "result$\neq \emptyset$", i.e. tuples that do not have any matches in a nested subexpression of L are discarded, like in

$$\sigma[RichEmpl \neq \emptyset](\pi[dname, RichEmpl := \pi[ename](\sigma[sal > 50000](Empl))](Dept))$$

("retrieve a list of department (names) each with a list of employees (names) making more than 50k, drop departments without any such employees"). In addition to the above canonical form we can have unnest operations at any place, however, often it will be sufficient to unnest as a last step:

$$(\mu[\ldots])^* \sigma[F_2](\pi[L](\sigma[F_1](R))),$$

where $(\mu[\ldots])^*$ means a sequence of unnests (each of which applies to the only subrelation).

Conclusion

We identified the class of single scan processible operations in a nested relational query language. We used the nested algebra of [SS86] in the discussion, however, the results apply to all query languages with similar query facilities. The class of linear operations plays an important role if we

- implement a nested relational DBMS
  where we should try to realize the full functionality of that class within the lowest level of query processing algorithms, the canonical linear expression may be used in defining the interface of such a component,
- design an optimizer
  which should map as many queries as possible to linear expressions, as this helps in reducing processing costs by avoiding duplicate scans.

Concerning the DASDBS prototype, we find a subclass of linear expressions very close to the canonical form at the low-level query processing layer (CRM, see [DPS86, PSS+87]). Algebraic optimization in our nested relational algebra has been studied in [Sch86, SPS87, Sch88], where we found that it is exactly this class of queries that is needed to efficiently support what would be select-project-join queries on an equivalent flat relational database.

References

[ABC+76] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffith, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System R: Relational approach to database management. *ACM Transactions on Database Systems*, 1(2):97–137, June 1976.

[CAB+81] D. D. Chamberlin, M. M. Astrahan, M. W. Blasgen, J. N. Gray, W. F. King, B. G. Lindsay, R. Lorie, J. W. Mehl, T. G. Price, F. Putzolu, P. G. Selinger, M. Schkolnick, D. R. Shultz, I. L. Traiger, B. W. Wade, and R. A. Yost. History and evaluation of System R. *Communications of the ACM*, 24(10):632–646, October 1981.

[DGW85] U. Deppisch, J. Günauer, and G. Walch. Storage structures and addressing techniques for the complex objects of the NF^2 relational model. In A. Blaser and P. Pistor, editors, *Proc. GI Conf. on Database Systems for Office, Engineering, and Scientific Applications*, pages 441–459, Karlsruhe, 1985. IFB 94, Springer. (in German).

[DPS86] U. Deppisch, H.-B. Paul, and H.-J. Schek. A storage system for complex objects. In *Proc. Int. Workshop on Object-Oriented Database Systems*, pages 183–195, Pacific Grove, September 1986.

[JK84] M. Jarke and J. Koch. Query optimization in database systems. *ACM Computing Surveys*, 16(2):111–154, June 1984.

[Loh88] G.M. Lohman. Grammar-like functional rules for representing query optimization alternatives. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 18–27, Chicago, June 1988. ACM, New York.

[PSS+87] H.-B. Paul, H.-J. Schek, M. H. Scholl, G. Weikum, and U. Deppisch. Architecture and implementation of the Darmstadt database kernel system. In *Proc. ACM SIGMOD Conf. on Management of Data*, San Francisco, 1987. ACM, New York.

[Sch86] M. H. Scholl. Theoretical foundation of algebraic optimization utilizing unnormalized relations. In *ICDT '86: Int. Conf. on Database Theory, Rome*, pages 380–396. LNCS 243, Springer, Berlin, Heidelberg, 1986.

[Sch88] M. H. Scholl. *The Nested Relational Model —Efficient Support for a Relational Database Interface—*. PhD thesis, Dept. of Computer Science, Technical University of Darmstadt, 1988. (in German).

[SPS87] M. H. Scholl, H.-B. Paul, and H.-J. Schek. Supporting flat relations by a nested relational kernel. In *Proc. Int. Conf. on Very Large Databases*, pages 137–146, Brighton, September 1987. Morgan Kaufmann, Los Altos, CA.

[SS86] H.-J. Schek and M. H. Scholl. The relational model with relation-valued attributes. *Information Systems*, 11(2):137–147, June 1986.

[SWKH76] M. R. Stonebraker, E. Wong, P. Kreps, and G. Held. The design and implementation of INGRES. *ACM Transactions on Database Systems*, 1(3):189–222, September 1976.