Extending Iterators for Advanced Query Execution

Florian Waas

CWI
Kruislaan 413
1098 SJ Amsterdam
The Netherlands
flw@cwi.nl

Abstract

Today's commercial relational database systems use tree-shaped execution plans. The evaluation techniques for these plan are well understood and have been refined over the last decade. However, for queries that contain disjunctive predicates, using the more general class of direct acyclic graphs and splitting data streams can be beneficial. Unfortunately, the iterator based evaluation techniques used for tree-shaped plans do not apply to this case. Iterators implement a breadth first search providing full encapsulation where operators communicate by answered requests in synchronous manner.

In this paper we develop an extension of the conventional iterator based evaluation technique. We introduce request handles that add context information to the data requests which allows for arbitrary plan topologies including cycles. The original problem of evaluating plans with operators that split data streams can then be solved by mere rewriting of the execution plan.

Keywords: *Query evaluation, Iterators, Disjunctive predicates*

1 Introduction

Query execution is the last in the chain of tasks in the query evaluation process which executes the relational algebra expression the query optimizer generated. Since the relational algebra is a functional approach, the evaluation of any expression in this algebra can be structured as a tree. The nodes depict the operators and the edges express data dependencies.

The expressions are evaluated with a kind of depth-first search known as iterator concept [Gra93]. Iterators can be viewed as nested function calls where an operator requests data from its predecessors which in turn may request data from their predecessors and so on. The concept of iterators emerged as the *de facto* standard for essentially two reasons: its resource efficiency and the strict encapsulation which guarantees a high degree of extensibility.

In the traditional relational setting, the output of an operator consists always of *qualifying* data. Consider for instance a filter that implements a restriction. All data the filter passes on to the next operator has to fulfill this restriction.

As Kemper et. al. showed in [KMPS94] and [SPMK95], in the context of disjunctive queries, it can be very beneficial to also consider the data that does *not* qualify. Because, in some situations data that qualifies at one operator can probably bypass other operators whereas data that does not qualify in the first place needs to pass additional filters. The resulting evaluation plans are no longer tree-shaped. More general, operators may split the data stream and unify it at a later point in time again. However, as they observed, this new class of plans cannot be evaluated with the iterator model as the principle that *every* data request is answered with either a data item or, if no further data is to be processed, with a special token, does no longer apply.

We will develop a solution to this problem which adds two new aspects to the iterator concept preserving the advantages of the original approach. First, all data requests are identified by a request handle so that each operator knows who requested the data and has therefore the possibility to respond in different ways. Thus, operators may have any number of inputs or outputs to allow for arbitrary plan topologies. And secondly requests may be answered with a special token that indicates that no data is available at the moment. Those two extensions provide a flexible framework in which the problems imposed by trees other than tree-shaped can be solved by mere rewriting of the plan.

```
class Iterator
{
    ...
    void open();
    DataUnit next();
    void close();
}
```

Figure 1. Iterator interface.

2 Iterators

The concept of iterators is widely used in both commercial databases like DB2 or SQLServer, and research prototypes [Gra90]. In this section, we outline the principles, following Graefe's approach, briefly. For a more detailed description, we refer the interested reader to [Gra93] and the standard literature on database system implementation.

Every operator, i.e. node of an evaluation plan can be abstracted with an interface consisting of three components as mentioned above. Figure 1 shows a C++ style like notation. The roles are as follows:

open. Operators may need to set up internal structures like memory buffers etc. All these initializations are done in the call to the *open* routine.

The operator propagates the *open* to its children which in turn pass it on to their predecessors recursively. That is, after calling the *open* of the root operator all necessary structures of the query plan get initialized.

next. This procedure implements the actual algebraic operator for a single unit of data (`DataUnit`). In a sequential plan this typically is a single tuple—in parallel systems where the passing on may involve additional communication costs, a larger granularity can be chosen, e.g. pages (see below).

Similar to the *open*, the *next* call is propagated and input data is demanded from the child operators. The *next* call is always answered with *qualifying* data or the *End-Of-Stream* token, a special instance of `DataUnit` indicating that no more data is available. Depending on the algebraic nature of the operator not all of its inputs are handled the same way but for instance all data of one child is processed before any data from the next child is requested. Also the case that all input data has to be processed before any output is produced is possible, e.g. when sorting or processing aggregates.

close. The *close* call is the counterpart of the *open*. Temporary data structures necessary for a proper functioning of the *next* are released and resources are returned to the operating system's resource pool. As with all other operations, the *close* is recursively propagated throughout the query plan.

The iterator concept has proven a very robust implementation of relational algebraic operators. Its main advantages are the easily achieved extensibility with respect to new operators as well as to different implementations for one operator. However, most notable is the implicit resource management: all data is generated on *demand* (*next* call), i.e. only when needed for the next processing steps, so, no resources are occupied longer than necessary. Due to this fact the iterator concept is referred to as *demand-driven evaluation paradigm*. We will use both names synonymously in the remainder of the paper.

3 Non-Deterministic Data Availability

From the disjunctive queries mentioned above we derive a more general model that consists of an operator which splits data according to a predicate into several disjoint sets (see Fig. 2a, SPLIT), that is, branches in the execution plan. Finally after the data is assigned to a certain branch and processed the separate results are collected (COLLECT). The branches may contain an arbitrary number of operators. However, at most one of the branches may be empty, i.e. contain no operator, as with the previous example. To have more than one empty branch is not useful—therefore we exclude this case form further considerations. We call a situation where the activation of a partial query plan depends on predicate evaluation at run time, *non-deterministic data availability*. Similar situations also occur in parallel and distributed databases [Gra96, Waa99].

For the moment let us assume that every branch contains only one single operator and each operator outputs all of the data it consumes. This corresponds to restrictions all data fulfills.

the first request is sent to the top most operator. Then it is passed to the predecessor and so on. In a tree-shaped graph it does not matter which side of a union is evaluated first, that is, one of the sides may be preferred from the resource allocation point of view but both ways work. However, with non-deterministic data availability the request has to anticipate the outcome of the SPLIT. Since this is impossible due to the encapsulation of the predicate in SPLIT, the following may happen: The request is sent to the COLLECT (cf. 2b (1)) which in turn sends it to one of its predecessors—no matter the particular decision mechanism used for to decide which predecessor to call. Requests are indicated by gray, responses by black arrows. Let us assume without loss of generality it is sent to left as indicated in Figure 2b by step (2). The request is propagated to the SPLIT and data is requested from the bottom most operator (4). An answer is obtained and the control flow returns to the SPLIT.

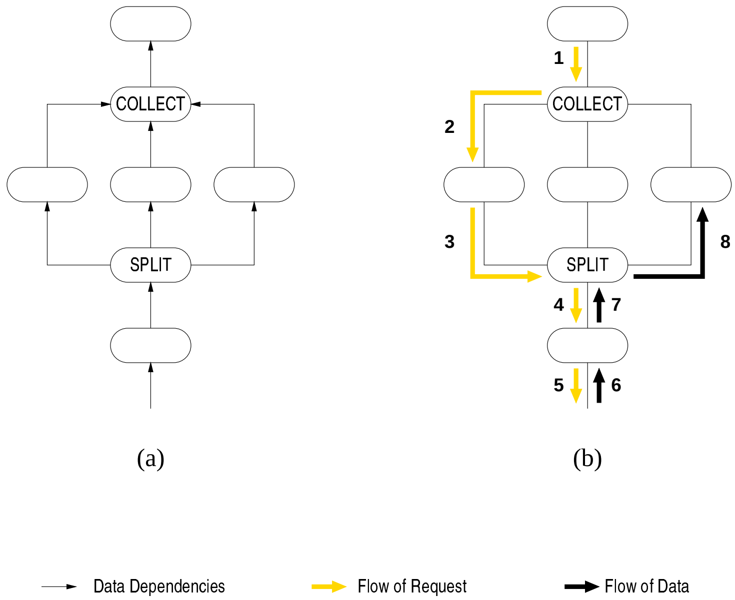

Figure 2. General model for non-deterministic data availability.

If the SPLIT assigns the data to the left branch the data is forwarded to the callers successively and the control flow returns to COLLECT etc. However, if the data is assigned to the middle or right branch by the SPLIT the processing breaks as the function call down the left branch must be closed first (cf. 2b(8)).

The only solution suggested so far uses a buffer at the SPLIT [CKM+99]. The COLLECT sends a request down on one of the branches and tries to get data that fulfills the particular branch's predicate. All data that is checked and assigned to a different branch is buffered for the time being. If the buffer is full or no further data from SPLIT's predecessor is available, but no data was assign to the calling branch the request is closed with an empty tuple and a new request is sent down from the COLLECT on another branch. However, this technique has the severe drawback that possibly large intermediate results are materialized in the buffer. Moreover, substantial overhead of unnecessary function calls is added. Last and most notably, this technique partly sacrifices extensibility as for instance the nesting of several SPLIT/COLLECT pairs is not possible.

4 Request handles and TNAs

Our solution to the problem consists of two parts. First, we enrich the iterator model so that operators can distinguish different kinds of requests. Then, we restructure the query plan using this new feature.

In order to cope with operators that provide more than just one output stream we extend the generic iterator interface in two ways:

```
DataUnit TNA;

class RequestIterator
{
    ...
    void open(RequestHdl &hdl);
    DataUnit next(RequestHdl &hdl,...);
    void close(RequestHdl &hdl);
}
```

Figure 3. Extended interface.

1. All functions differentiate their callers by *request handles*.[1] This allows individual action for different consumer operators.

2. Besides qualifying tuples and the End-Of-Stream token, the *next* call may also return a special *Temporarily-Not-Available (*TNA*)* token, indicating that no qualifying data is available *at the moment*. Streams that may contain TNAs are called *non-strict*, otherwise *strict*.

In Figure 3, the extended interface is shown. Using the new interface, we are no longer restricted to tree-shaped query

[1] From the technical point of view, request handles are comparable to UNIX file handles.

plans. But to solve the problem of non-deterministic data availability, we also need to transform the query plan. We collapse the SPLIT and COLLECT operators to one single operator called HUB, as shown in Figure 4.

As before, requests are shown as gray arrows, responses black. The numbers illustrate the single phases for a tuple that qualifies for the right operator. After fetching the tuple from the preceding operator (2–5) a request is sent to the right operator (6) which in turn requests the data from the HUB (7,8), the answer is processed and returned to the HUB (9). Finally, the result tuple is passed on to the successor (10).

Using TNA tokens ensures consistent processing. Furthermore, the stream to the successor operator is always strict, i.e. any regular operator can be used as successor. The extension and modification to both evaluation paradigm and query graph adhere to the basic principle of encapsulation providing unrestricted flexibility like the original iterator interface.

Discussion

The introduced request handles together with the concept of TNA tokens allow the general handling of query plans other than tree-shaped with a simple, sound and consistent technique. However, there are some points that need particular consideration when it comes to an efficient implementation.

1. During the assembly of the query plan the additional decision whether to use a strict or a non-strict version of a particular operator needs to be made. Unlike with the bare iterator paradigm, not all combinations are allowed—some may lead to deadlocks. On the other hand, it appeared in all our experiments there always is a deadlock-free variant that can be chosen. This subject requires further research and explicit modeling though.

2. The way we introduced the new evaluation paradigm was to render the principle as lucid as possible. Clearly, we would add many superfluous function calls to the execution, resulting in unnecessary copying of data over the stack, to the execution. But also implementing the bare demand-driven iterators with function calls (*next*) is known to be too expensive a strategy. Therefore, the *next* calls are translated into a navigation on the tree structure without copying any tuples unless necessary as for instance in JOINs when the format of the data changes. We use a similar technique for the extended iterators. TNAs do not get copied through a possibly deep nesting of function calls—consider cases where a branch consists of more than only one operator but the address of the first caller is used to get back in one single step and resume processing without any great delay.

Finally, enriching a conventional query processing system with the new paradigm is easy. The necessary plan transformations can all be done after the query optimization took place and do not interfere with any other stage of the processing.

5 Summary

In this paper, we addressed the problem of non-deterministic data availability in query evaluation. We showed that in cases where the activation of parts of a query plan depends on run-time decisions, the demand-driven evaluation paradigm cannot be applied. We developed an extension that enables multiple outputs per operator and gives us the possibility to distinguish the calling operators. Based on this fundamental extension we re-modeled the original query graph and showed how to handle non-deterministic data availability preserving full encapsulation, flexibility and facilitate the exchange of implementations. The concepts presented have been implemented in a query engine prototype and proved a framework that is easy to realize, enables extensibility by its uniform interface, and most notable provides run-time and resource efficient execution.

In case no data stream splitting operators are used, the new technique reduces exactly to the conventional demand-driven evaluation paradigm providing full compatibility.

Our agenda for future research includes a theoretical model that allows us to verify practical experience that we can always find a dead-lock free rewriting of the original query graph.

Furthermore, we see parallel query processing as an interesting area of application where data partitioning SPLIT operators are used to distribute data among different processes [Waa99].

Acknowledgments

Thanks are due to Cesar Galindo-Legaria and his colleagues at Microsoft's SQLServer group for fruitful discussions on the subject as well as to Nikola Dimitrov who helped implement the prototype of the execution engine.

References

[CKM+99] J. Claussen, A. Kemper, G. Moerkotte, K. Peithner, and M. Steinbrunn. Optimization and Evaluation of Disjunctive Queries. *IEEE Trans. on Knowledge and Data Engineering*, 1999. To appear.

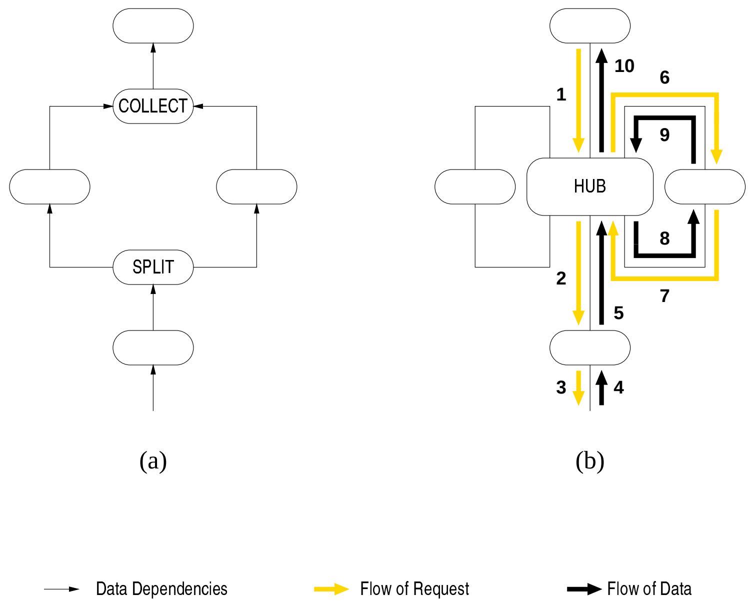

Figure 4. Collapsing SPLIT **and** COLLECT**.**

[Gra90] G. Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 749–764, Atlantic City, NJ, USA, May 1990.

[Gra93] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[Gra96] G. Graefe. Iterators, Schedulers, and Distributed-memory Parallelism. *Software—Practice & Experience*, 26(4):427–452, April 1996.

[KMPS94] A. Kemper, G. Moerkotte, K. Peithner, and M. Steinbrunn. Optimizing Disjunctive Queries and Expensive Predicates. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 336–347, Minneapolis, MN, USA, May 1994.

[SPMK95] M. Steinbrunn, K. Peithner, G. Moerkotte, and A. Kemper. Bypassing Joins in Disjunctive Queries. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 228–238, Zurich, Switzerland, September 1995.

[Waa99] F. Waas. Handling Non-deterministic Data Availability in Parallel Query Execution. In *Int'l. Workshop on Parallel and Distributed Databases*, pages 61–65, Florence, Italy, September 1999.