Handling Non-deterministic Data Availability in Parallel Query Execution

Florian Waas

CWI
P.O.Box 94079
1090 GB Amsterdam
flw@cwi.nl

Abstract

The situation of non-deterministic data availability, where it is not known a priori *which of two or more processes will respond first, cannot be handled with standard techniques. The consequence is sub-optimal processing because of inefficient resource allocation and unnecessary delays.*

In this paper we develop an effective solution to the problem by extending the demand-driven evaluation paradigm to the end of using operators with more than just one output stream. We show how inter-process communication and non-deterministic data availability in parallel query processing reduce to cases that can be executed efficiently with the new evaluation paradigm.

1 Introduction

While many concepts fit smoothly in the parallel environment some don't. The one we address here is the question what evaluation paradigm to use for the transport of data throughout the query evaluation plan.

In sequential systems, the *demand-driven* paradigm where data is generated only when needed—keeping the resource usage down and causing almost no overhead—emerged as the *de facto* standard. The problem, however, turns out to be quite different in the parallel case.

The solution proposed in Volcano [1] seems intuitive but proved not general enough to support various kinds of parallelism and the major hardware architectures. The far-reaching changes to the systems introduced in [3] to overcome some of the deficiencies added sizeable overhead which not only slows down the query execution but is also more difficult to allow for in the optimization phase.

In this paper we develop an effective solution to this problem in two steps. First, we show how the demand-driven evaluation paradigm can be extended by *request handles* that allow operators to distinguish their callers. Based on this extension we devise a new evaluation paradigm called *request-driven evaluation paradigm* which, together with a query plan transformation, enables the execution of algebraic operators with more than only one output stream. Secondly, we encapsulate both incoming and outgoing inter-process communication during parallel execution in one *single* relational algebraic operator overcoming the problems of Volcano's *exchange* operator.

2 Iterators

Every relational algebraic expression can be denoted as a tree-shaped evaluation

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

Figure 1. Iterator interface.

graph, called *query plan* in the sequel, with data-flow from the leaves to the root. An operator, i.e. node of the tree can be abstracted with an interface consisting of three components.[1] Figure 1 shows a C++ style like notation. The roles are as follows:

open. Initializes internal structures like memory buffers etc. The operator propagates the *open* to its children which in turn pass it on their predecessors recursively.

next. This procedure implements the actual algebraic operator for a single unit of data (`DataUnit`).

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.

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 any longer than necessary.

[1]For a more detailed description, the interested reader is referred to [2] and the standard literature on database system implementation

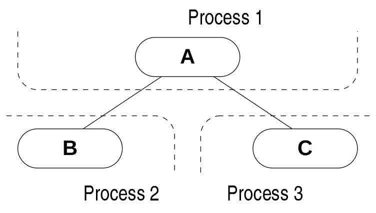

Figure 2. Parallel plan, detail.

3 The Problem

Among other parallelization schemas that have been proposed, cutting an evaluation plan in smaller parts which are assigned to groups of processors afterwards is an important building block for parallel execution [4].

The critical spots for the evaluation are the process boundaries, i.e. the data demands that involve inter-process communication. Clearly, the iterator's *next* call can be accommodated to the special requirements of the inter-process communication medium, e.g. shared-memory communication, RPC etc. However, the recursive proceeding is designed for a single control flow with deterministic data availability. To illustrate the consequences we focus the example in Figure 2.

In a sequential environment, operator A would request data from the children B and C one after another, i.e. repeatedly sending a request and receiving an answer. Now, consider a parallel environment with process boundaries as indicated. For maximal data throughput, all three processes should act as independently of each other as possible. That involves two steps: on the one hand, the request from A should simultaneously go to B and C, and not wait until the answer from the one which has been called first is received. On the other hand, and this is more difficult to overcome, the response data should be collected on a first-come-first-serve basis. This is not only important for operators that can consume data from every input at any time like the UNION, but also operators like JOIN can benefit from it by caching input data locally. However, the desired behavior cannot be achieved with the bare iterator model. Every *next* call is “synchronous” and terminates only when the response data is provided—in other words, the call has to anticipate which of the children will answer first or will waste processing time and resources.

General Model. Consider an operator pair SPLIT/COLLECT which distributes data among an arbitrary number of branches and collects the results. Moreover, the assignment of data to branches is done based on specific properties of the data, e.g. value ranges. This kind of query plan cannot be evaluated with the crude iterator model as the data availability is non-deterministic, i.e. it is unknown which way the tuple will be passed on until the predicate of the SPLIT is evaluated. However, the *next* call from the COLLECT will be propagated to one of the branches prior to this evaluation.

In the following we first focus on this sequential instance of the problem and present a solution by extending the evaluation paradigm. Finally, we develop an approach to transfer the new concepts to the original parallel problem.

4 Request handles and TNAs

In order to cope with operators that provide more than just one output stream we extend the generic iterator interface in two ways (cf. Figure 3):

1. All functions differentiate their callers by *request handles.* 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.*

```
DataUnit TNA;

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

Figure 3. Extended interface.

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. The numbers illustrate the single phases for a tuple that qualifies for the right operator.

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.

5 Making Parallel Query Execution Work

The new evaluation paradigm as introduced above can be used as a powerful tool to tackle the original parallel problem. All we have to do is restructure the original query plan in an appropriate way.

5.1 Query Plan Design

Given a query plan and its scheduling, i.e. the assignment of operators to processors we encapsulate all inter-process communication with two new operators: IN and OUT. In Figure **??**, an example, slightly more complicated as the previous ones, is shown. In the next step, we collapse all INs and OUTs of a process leaving only one

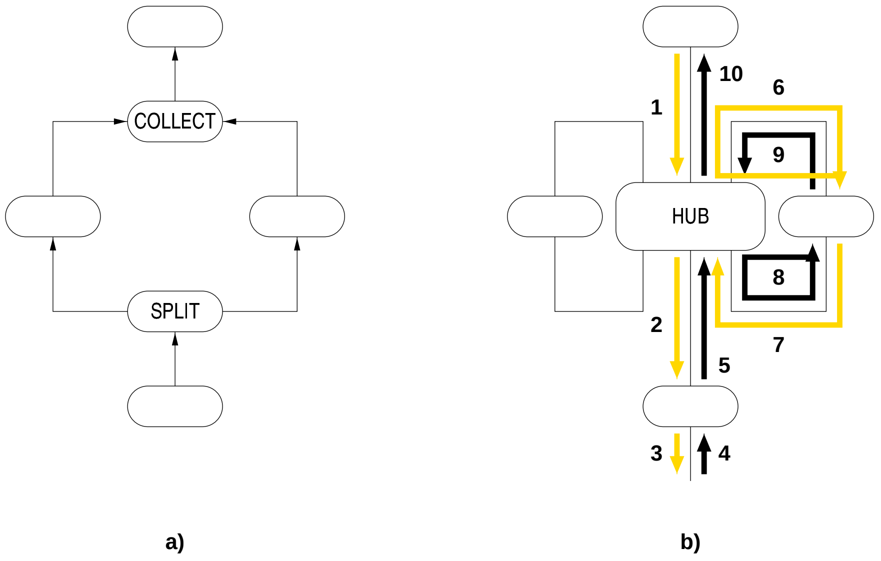

Figure 4. Collapsing SPLIT and COLLECT.

single IN and also one single OUT. At the same time we introduce request handles to identify the streams. The reduced graph corresponds now to the general model for non-deterministic data availability. In the last step we collapse the intermediate IN and OUT operators to a new operator called COMM, which forms the hub in the new plan. An example plan layout is given in Figure 5.

5.2 Processes at Work

After the query plan is restructure and enriched with COMM operators the single parts of the query plan are loaded by the separate processes of the parallel query engine.

The COMM operators are not only distinguished by being the SPLIT/COLLECT hub of the entire query plan of the respective process, they are also extended with inter-process communication means. Moreover, every input and output stream is assigned a buffer within the COMM. After activating the process, the COMM operator sends asynchronous requests to all its producer processes it depends on. Then, a regular request is sent to the process' top-most internal operator, propagated through the local query plan eventually requesting data from the COMM. Like with the general model, the COMM answers these requests in dependency of the caller's request handle.

The input and output buffer ensure maximal process independence. Their sizes are parameters of the query optimization, but for simplicity can also be fixed after calibrating the system.

Once a COMM filled all its output buffers no further internal requests are emitted until the processed data is requested by a consumer process and output buffer capacity becomes available again. By this way an effective self-regulatory mechanism is established that does not need interference from the consumer like the back-pressure concept in Volcano. On the other hand, the COMM does not send requests to other processes unless the local input buffers are empty.

The inter-process communication can now be handled interleaving with the regular processing of the local query graph without problems since all arriving data is stored in the COMM's buffer pool first. Hence, the control-flow within the process does not need to be corrected—all operators that require data that is input to this process simply request it from the COMM.

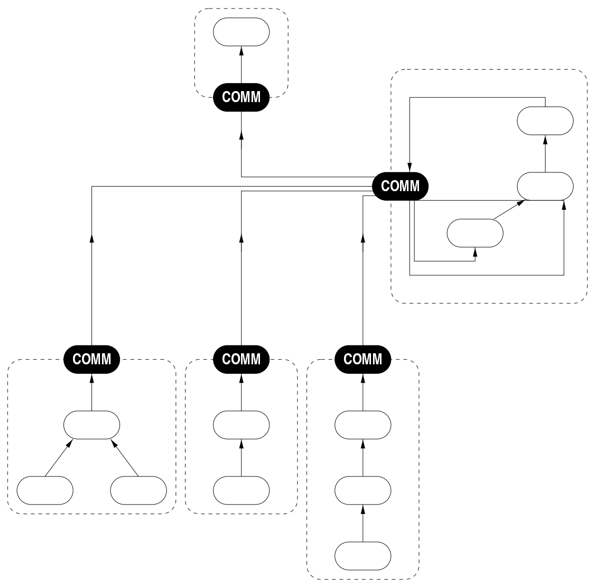

Figure 5. Parallel plan with fully encapsulated inter-process communication.

6 Summary

In this paper, we showed how the demand-driven evaluation paradigm can be extended to suit the advanced requirements of parallel query execution. In contrast to previous work, our new technique not only preserves full encapsulation, flexibility and easy exchangability of implementations for relational algebraic operators, it also offers an elegant solution to the problem of non-deterministic data availability in both sequential and parallel execution.

The query plan layout we proposed makes parallel query execution a simple yet highly efficient task without additional overhead, providing a self-regulatory mechanism of activation and de-activation.

The concepts presented have been implemented in a parallel query engine for shared-nothing workstation clusters and proved a framework that is easy to implement, enables extensibility by its uniform interface, and most notable provides run-time and resource efficient execution.

Acknowledgments

Thanks are due to Nikola Dimitrov who helped implement the prototype of the execution engine.

References

[1] 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.

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

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

[4] W. Hasan and R. Motwani. Optimization Algorithms for Exploiting the Parallelism-Communication Tradeoff in Pipelining Parallelism. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 36–47, Santiago, Chile, Sept. 1994.