Dynamic Aspects of Query Processing in Parallel Database Systems*

Johann K. Obermaier  Florian Waas

Humboldt-Universität zu Berlin, Institut für Informatik
Unter den Linden 6, 10099 Berlin, Germany
(lastname)@dbis.informatik.hu-berlin.de

Abstract
This paper reports ongoing work in developing a query representation method for the implementation of dynamic and parallel query processing in database systems. We present Stream Processing based Query Representation — a concept derived from software engineering — as powerful approach, which covers query representation from high-level query declaration to low-level procedural (parallel) execution plans. Hence, it is an ideal basis to extend the query processing component at any level, e.g. with new optimization techniques or dynamic execution methods.

1 Introduction

Query evaluation in parallel database systems (PDBS) is quite different from evaluation in sequential systems. Exploitation of parallel systems requires additional tasks and concepts like inter-process communication, scheduling, load balancing, and parallel implementations of algebraic operators [DG92]. So far, much work has been performed on very special issues of query optimization (cf. [LOT94]). Many of the proposed solutions have been proven to be efficient under more or less restrictive assumptions but cannot be controlled by optimizers under realistic assumptions. Thus, query optimization in PDBS is still an open issue [HFV96].

A major reason for this dilemma is the strict separation of optimization and execution in consecutive phases. The optimizer generates an query evaluation plan (QEP) for the execution engine. To simplify the optimization task it is broken up into orthogonal phases $P_i$ ($i = 1, ..., n$), e.g. algebraic transformations, scheduling, etc. In general, each phase $P_i$ has it’s own representation $R_i$ for the query, ranging from SQL to query evaluation plan. An optimization step is a mapping either from $R_i$ to $R_i$, i.e. an in-phase optimization step of phase $P_i$, or from $R_i$

*This work was supported by the German Research Council under contract DFG Fr 1142/1-1.
to \( R_{t+1} \), i.e. a \textit{transition} from phase \( P_t \) to phase \( P_{t+1} \). Obviously, multiple in-phase optimization steps of one phase are commutative and each step can be applied multiple. But in each phase only one transition step can be applied and it is irreversible. The decision which step to apply next is based on cost estimations. So, the quality of the optimization result depends extremely on the accurateness of the cost prediction.

The answer to this problem is to give up the strict separation of optimization and execution by pushing optimization steps into the execution phase. Some first rudiments of this dynamic optimization strategy have already been proposed for sequential systems [GW89, INSS92]. The advantage is that during query running time many parameters can be determined exact or at least more precise than in advance, i.e. load situations in parallel systems. As long as optimization is done in consecutive phases each query representation must only cover information for the current and successive phases. With dynamic optimization the representation in a phase must cover information for all phases, because there is no fix order of the phases any more. The key to this is a single query representation \( R \) for all phases \( P_t \), which always contains all information.

In this paper, we propose such a general query representation called \textit{Stream Processing based Query Representation (SPQR)}. It is based on concepts in software engineering [BDD+92]: Streams and stream processing operations. SPQR keeps all optimization information, which can be access at any time. Optimization steps can be applied not only in a predefined order. So, SPQR allows reorganizing and deferring dynamic optimization.

The remainder of this paper is organized as follows: In Section 2, we give an insight to query processing in PDBS, including the principal techniques for dynamic processing. In Section 3, we present the basic concepts of stream processing. Due to lack of space we define things in an informal manner. In Section 4, we introduce a classification of processing stages in PDBS and define our new methodology, called SPQR. In Section 5, we discuss the application of SPQR and modelling of dynamic processing. We give a brief overview of related work and make
suggestions for future work in Section 6.

2 Query Processing in PDBS

Here, we give a short sketch of the issue of parallel query processing. Due to the lack of space we not consider issues of sequential query processing. In parallel query processing we have to deal with two additional features: (1) extensions of the set of algebraic primitives with new kinds of operators, e.g. communication and partitioning operators, parallel implementation of operators; (2) parallel scheduling, i.e. parallelizing queries and assigning and reorganization of subtask to processing units depending on load, data location, etc. Optimization in PDBS is still an open problem, because the complexity increased dramatically because of the additional parallel scheduling [TL91, HFV96]. Hence, with static optimization, i.e. optimization before of query running time, errors in cost estimations can not be avoided. The resulting QEP is suboptimal. To overcome this drawbacks optimization decisions must be deferred to query running time. There are three basic approaches for dynamic optimization:

Reoptimization/Rescheduling The optimizer generates a query evaluation plan, which can be reorganized dynamically, i.e. transition optimization steps must be reversible. To realize this we need (1) to define conditions when to reoptimize, i.e. the difference between the actual and the assumed performance exceed a given threshold, (2) the knowledge to find an improved plan, i.e. the plan must contain enough information to derive a better plan, and (3) a component of the execution engine which controls activation and execution of the reoptimization.

Deferred optimization with alternative plans The optimizer defers optimization steps by generating alternative (sub-)plans. At execution time one of them is chosen [GW89]. To realize this we need (1) to predefine alternative (sub-)plans, (2) an special operator in the QEP which chooses from them. The advantage is that no extern component in needed, but after a (sub-)plan is chosen no further reoptimization of this (sub-)plan is possible.
Deferred optimization with incomplete plans  The optimizer defers optimization steps by generating a incomplete query evaluation plan. For example, in a shared memory parallel system processor assignment can be delegated to the execution or operating system and this optimization step can be omitted. This allows to integrate self optimizing execution strategies, like this proposed in [MOW96]. To realize this we need query representation which can handle incomplete plans.

3 Stream Processing and Stream Processing Operations

In our model, the basic unit of data processing is a single data item. It is comparable to the instance of an abstract data type. It covers both, the pure relational approach where data are always represented as tuples and more complex models like the object oriented. Like in the tuple relational calculus (TRC), the type of a given data item is just a property that can be evaluated within a predicate.

Then, a stream is a finite sequence of data items that may also be empty. Stream processing operations (SPOs) are mappings between different streams, i.e. they read data items from input streams and write items to output streams. For the moment, think of it just as black boxes, consuming and producing data. Stream processing seems very similar to list processing especially when expressed in functional style. Unfortunately, this notation hides the essential difference of both concepts: lists express only logic aspects whereas streams additionally cover temporal order, i.e. data availability. Please note, that streams are sets, neither. Thus, operators like \( \cup, \cap, \times, \ldots \), known from set theory and usually used within the relational algebra must be redefined in terms of stream processing.

Streams are a model for unidirectional channels where data items are sent and received by SPOs at a certain time. Obviously, connecting various SPOs may result in arbitrary networks.
4 Query Representation

Streams and SPOs, introduced so far, express the topology of data flow during execution, or in other words, the producer/consumer relationships. To describe the run time behavior, we must specify the single streams in more detail.

Predicates

Predicates are the key to the required specification. We apply them in two different ways: to characterize (1) data items of a given stream and (2) relationships between subsequent stream elements. The first seems very similar to the tuple relational calculus or, in particular, TRC on streams. In fact, tuple variables used in TRC correspond to the notion of data items in our framework. However, there is a major difference: In TRC the predicates are the query itself. The result relation is constructed with respect to this predicate only and consequently the problem of safety occurs. In our model, predicates are not used for querying but describing the evaluation process of a correct system, i.e. we assume a correct system, specify the output, and hence provide assertions, the following SPOs can rely on.

Obviously, streams can be over-specified by too restrictive predicates. So, specification must always be the minimal set of predicates which the subsequent operator needs as assertions for its task. For instance, if an SPO does not respect sorting on its input streams, we must not formulate any predicate about special orders within these streams.

Refinement

Refinement is the root of dynamic query representation in our framework. Beginning with a declarative form like an SQL statement, the query can be decomposed into subqueries — a common technique in today’s optimizers. But usually, refinement is applied during the optimization process and the intermediate refinement steps are discarded as they are worthless for further processing. In SPQR refinement and its intermediate results stay part of the plan: They contain all necessary information about scheduling.
<table>
<thead>
<tr>
<th>Level</th>
<th>What the SPOs represent</th>
<th>What the Streams represent</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td><strong>Declarative Query</strong></td>
<td>There is only one single SPO that represents the whole query evaluation system.</td>
</tr>
<tr>
<td>4</td>
<td><strong>Machine Scheduling</strong></td>
<td>SPOs of this level specify the evaluation, done by each participating machine; it reflects the shared-nothing level.</td>
</tr>
<tr>
<td>3</td>
<td><strong>Process Scheduling</strong></td>
<td>As it is not useful to have more than one process per processing unit (cf. [Gra90]), SPOs of Level 3 stand for the evaluation task of one process (= processor).</td>
</tr>
<tr>
<td>2</td>
<td><strong>Algebraic Expression</strong></td>
<td>Every SPO is a relational algebraic operator.</td>
</tr>
<tr>
<td>1</td>
<td><strong>Implementation</strong></td>
<td>The tool-box; usually a direct mapping of the previous level’s SPOs into the implementation.</td>
</tr>
</tbody>
</table>

Table 1: Refinement Levels in PDBS

Before discussing in more detail, we define our notion of refinement: A substitution of a single SPO \( f \) with a set \( G \) of SPOs is called refinement if all predicates which hold on \( OUT(f) \) hold on \( OUT(G) \), too. \( (OUT(G) \) denotes all output streams associated with operations \( g \in G \) that are not input stream to any \( g \in G \).

Different from the software engineering processes, where each model requires its own granularity of refinement, in PDBS we identify about five constant levels of representing a query (cf. Table 1).

The number of levels depends on the particular system. For example, the introduction of a new level between level 2 and 3 makes sense if there are constructions beyond the process description that cannot be mapped to the implementation directly. On the other hand, if every machine can handle only one process because of its hardware configuration, it may be useful to equate level 3 with level 4.

In software engineering, refinement converts declarative specification to functional and fi-
nally to imperative programming. Bringing this forward to SPQR would be, specifying every operator down to the according implementation. On the other hand, in context of query processing there is only a small set of standard operators which we call tool-box. So, we can omit detailed — but unnecessary — constructions and map components of level 2 straight forward to tool-box elements.

**Representation**

Now, we assemble the introduced concepts: In SPQR a query representation is a sequence of stream processing systems $S_t(i = 1, \ldots, n)$, with $S_{i+1}$ is a refinement of $S_i$. The depth of the refining hierarchy, i.e. the number of subsequent refinements, depends on the architecture of the respective database system.

![Diagram](image)

Figure 1: Representation Tree

Obviously, every refining step yields in general more than one substitution. Thus, many alternatives can be derived from the same first stream processing system $S_1$. These alternative representations form a representation tree as shown in Figure 1: As we use the level classification as mentioned above, the tree is of depth 5. A sample representation is depicted by the path of nodes $a,b,c,d,e$ (possible alternatives are indicated by thin lines).

5 **Application of SPQR**

The previous sections described how to model query execution with streams. Thus, SPQR as introduced so far can be used by conventional optimizers as query representation — even if they only support multi-phase optimization.
In this section, we show how to implement the three ways of dynamic optimization we presented in Section 2.

Reoptimization/Rescheduling
The requirement we must fulfill is to provide all information that is necessary to change plan. SPQR suffices as it provides random access to all orthogonal results of the optimization process. Refinement is an efficient method to derive alternative plans. In [OW96] we explain this approach more in detail and introduce the antagonistic operation, called reverse refinement.

Deferred optimization with alternative plans
Introducing alternative plans is to implement a switch operator. In SPQR we model this situation by integrating all required additional functionality in a new algebraic primitive (cf. choose-plan operator [GW89]). The alternative subplans are specified like static plans we described above.

Deferred optimization with incomplete plans
This new way of query representation is the actual strength of SPQR: Arbitrary optimization steps can be applied on the current representation without any respect to a certain order. This includes the omission of critical optimization steps that cannot be decided before running time. SPQR enables the optimizer to delegate this decisions to external components like the operating system of the execution engine or external optimizers.

As a very simple example, think of the decision what join method to use within a given query plan. The optimizer composes a query representation including all levels except the lowest level, the tool-box. This last refinement step is done by the execution engine during running time. This example seems similar to alternative plans but please note that we do not specify alternatives. That is, the engine may complete the refinement even with implementations, the optimizer do not know of.

In [MOW96] we investigated the delegation of processor scheduling in context of shared-everything architectures, a task, that can be done most efficiently by the OS and benefits from
direct hardware support. In terms of SPQR, level 3 is omitted, but all other levels are specified completely. Of course, this demands the engine to support a special interface and delegate the jobs to the respective units. As results show, this strategy is superior to all scheduling methods, that are determined by the optimizer in advance.

6 Conclusion

Query representation in existing parallel database systems is restricted to the implementation level and does not support dynamic query processing sufficiently. The probably most known representative is the parallel database programming language PFAD [HDV88], a successor the sequential language FAD [DV92] with message passing constructs added. The Papyrus Execution Language allows a general control of the dataflow [CS92]. In the proposed language SVP [PSV92] parallelism is understood as strictly divide and conquer on sets. However, all this concepts reflect query processing only on a very low or very high abstraction level. None of them include dynamic query processing sufficiently. To cover the aspect of dynamic parallelism we need a query representation which can express not only the query execution plan but also the resource scheduling of it.

In this paper, we presented SPQR as a query representation methodology for dynamic query processing in parallel (object-)relational database systems, which fulfills this demands. Beyond this, SPQR allows to express incomplete specifications of some query aspects if these can be completed easier by an external component. An example of such a component, which is not part of the optimizer, is the automatic process scheduling mechanism of modern operating systems in shared-memory environments. SPQR provides an uniform query language, which covers representation from the high-level query declaration to low-level procedural execution plans. Hence, it is an ideal basis to extend the query processing at any level, e.g. with new optimization strategies or execution methods. So far, we implemented an execution engine based on SPQR. In the future, we want to build an adaptable and extensible distributed
optimizer, where optimizer components are connected by a software optimizer bus, based on SPQR as a communication language.

References


