From the modeling of parallel relational query processing

to query optimization and simulation

LIONEL BRUNÉ, HARALD KOSCH and WOLFGANG WÖHNER
LIP–CNRS, ENS Lyon, 48 allée d’Itaïë
69364 Lyon, France

Received May 12, 1997
Revised July 4, 1997
Communicated by G.L. Rejnas

ABSTRACT

This paper presents a novel theoretical model for representing parallel relational
query processing. It is based on two components. First, a scheme graph, called
DPL graph, describes all possible execution dependencies between operators, including
communication and run-time control mechanisms. Second, a timed high-level Petri net
is used for modeling the data- and control flow of DPL graphs. Our model provides the
framework for implementing a parallel query optimizer which is able to access sub-search
spaces not yet considered. Furthermore, based on the DPL graphs and its related timed
high-level Petri net, a simulation environment has been designed for testing run-time
control strategies as well as query optimization methods.

Keywords: Parallel databases, theoretical representation model for query processing,
query optimization, simulation

1. Introduction

With the emergence of decision support systems, relational queries tend to be-
come more and more complex. For instance, the April 97 results of Teradata for
the TPC-D benchmark [1] (database size 1 TB, 20 processors system) showed an
average response times of 7126 seconds (i.e. about 120 minutes).

This complexity renders the classical query representation models obsolete be-
cause they do not provide a sufficiently accurate view of the actual parallel execu-
tion. In that context, this paper presents a novel theoretical model for representing
parallel relational query processing, allowing a very precise description of the im-
plementation techniques used in parallel databases. This model is based on two
components. First, a scheme graph called DPL graph is introduced, which repre-
sents the “scenario” of the parallel execution. Based on an analysis of the execution
dependencies between operators, DPL graphs allow a very accurate representation
of most parallelization strategies. Furthermore special operators model communication
and run-time control. Second, a timed high-level Petri net, the TB-nets, are used for modeling the dynamics, i.e. the data and control flows, of DPL graphs.

a The initials stand for three kind of dependency edges : the data, precedence and loop dependency.

b Timed Basic nets [2], developed at the Politecnico di Milano.
DPL graphs, completed by the TB-nets description provide the framework for implementing a parallel query optimizer based on DPL graphs. This query optimizer accesses sub-search spaces, containing low-cost processing strategies, not yet considered in this context. In particular an original class of execution strategies is introduced: the serialized bushy trees. Furthermore, a simulation environment for testing run-time control strategies as well as query optimization methods is designed.

This paper is organized as follows. Section 2 proposes a brief overview of previous works. Section 3 introduces the basic concept of DPL graphs. Section 4 presents how the DPL graphs have been used for implementing our parallel query optimizer. Section 5 describes how TB-nets can model the data and control flows in DPL graphs. Section 6 shows how the flexibility of the DPL graphs and their related TB-nets have been used for designing a parallel query processing simulation environment. Finally, section 7 concludes this paper and points out future developments.

2. Related work

Query parallelization strategies are usually represented by so-called query processing trees (PT) [3]: The leaves of a PT represent the base relations that participate in the query and intermediate nodes model operations. These latter receive their input relations via the incoming edges and send the result relation through the outgoing edge to the next operator.

The PT has been chosen as a model for representing parallel query execution since it allows expressing different kinds of parallelism. Thus, inter-operator parallelism: operators lying on different paths of a query processing tree can be executed concurrently. Intra-operator parallelism: each relational operator can be decomposed into several sub-operators to be executed on different partitions of the same relation. Finally pipeline parallelism: two nodes lying on the same edge are in a producer-consumer relationship.

In order to express fine-grain parallelism, most works [4,5,3] introduce low-level implementations into the existing PT model. The resulting representation will furthermore be called D graphs (D stands for the Data dependencies between operators).

To the best of our knowledge, previous work [4,6,5,7,3,8] consider only data dependency between operators and suffer from the fact that they do not provide appropriate representations of two important parallelization strategies. Thus, they do not take into account a processing strategy in which operators are ordered without reference to a data stream. Furthermore, none of these models deal correctly with the algorithms based on bucket processing. This technique is applied when relation partitions cannot fit into the main memory of the processors. Instead of working on the whole partitions, these algorithms splits the relation partitions in memory-resident buckets [9].
As far as we know, only two major works have proposed to model the control- and data flows associated with an execution scenario. In the Bubba project [10], a control protocol is associated with every operator. This protocol, compiled into the execution scenario, specifies the run-time execution of the operator. However, this framework only allows the representation of simple data-dependency between operators. For instance, pipeline parallelism cannot be considered. The modeling of the control schema in DBS3 [11] is based on the propagation of termination messages and triggers of operators. Two kinds of triggers are considered, sequential and pipeline triggers. Although this framework is more sophisticated than Bubba’s, it suffers from the fact that it is completely static, i.e. it does not integrate dynamic control schemes.

In our approach, the control and data flow is modeled with the help of Timed Basic Petri (TB) nets. This representation is more compact and is based on a well-founded theoretical background. The notion of conditioned transitions firing allows the adaption of the control flow to the actual run-time context. Furthermore, the introduction of a timestamp to the token flow enables the introduction of a simulation measure. Thus, we were able to design an simulation tool for parallel query processing.

3. DPL graphs

This section introduces the complete notion of DPL graphs. It is based on the study of the optimization of a sample query \( R_1 \bowtie R_2 \bowtie R_3 \) executed on a shared nothing system. Relations are supposed to be already partitioned on the join attribute over all the disks. The intermediate result relation \( R_2 \bowtie R_3 \) must be repartitioned. The two join operators are implemented using hash-based algorithms, whereby the two hash tables are built on the base relations \( R_2 \) and \( R_1 \).

3.1. Operator vertices

The vertices within a DPL graph represent various operators which can be divided into three different categories. First, the basic operators are atomic operators working on relation partitions. Basic operators are graphically represented by circles whose inscriptions detail their functionality (e.g. build hash table). Second, the communication operators implement data redistribution. They are graphically represented as boxes whose inscriptions state the kind of repartition to be done. Example: suppose the join \( R_1 \bowtie R_2 \) is to be executed using a hash based method. If the input relation \( R_1 \) is not distributed on the same processors as \( R_2 \) is, at least one of the two relations must be repartitioned. This can be specified using a communication operator any-to-any repartition. Third, the control operators are used to control the query processing, their inscriptions state the kind of control to be performed. They are graphically represented by lozenges. Example: The control operator choose [12] decides dynamically between several alternative execution strategies (see fig. 5).
All operators are enriched with annotations depending on the characteristics of a parallel environment. Those annotations specify for example the method how the stored data is accessed and the kind of data dependency that exists between relational operators (sequential or pipelined). Additionally, the different types and degrees of parallelism, namely pipeline and data parallelism, can be referred to.

3.2. From D graphs to DPL graphs

D graphs cannot model all possible parallel execution strategies. To illustrate this, assume that the available memory is limited and that the hash-table on \( R_1 \) can be built only when the \( R_2 \bowtie R_3 \) has been terminated. Such a situation can not be represented using D graphs (fig. 1, left scheme\(^c\)). Indeed no data dependency exists between the probe hash table \( R_3 \) and the build hash table \( R_1 \). Therefore, these two operators should be executed in parallel.

Representing such an execution strategy requires the introduction of an original **precedence dependency**, stating that an operator must be terminated before another operator can start, though no data dependency is involved. Graphs including precedence dependencies, graphically represented as a double directed edge, will be called **DP graphs** (for data and precedence dependencies). Thus, the DP graph of fig. 1, right scheme, indicates that the build hash table \( R_1 \) can start only when the join \( R_2 \bowtie R_3 \) (probe hash table \( R_3 \) operator) terminated.

Let us consider now a more realistic processing situation in which the relation partitions do not fit into the processors main memory. As told before, the common

\(^c\)In order to keep the figures readable, only the data dependencies between operators (sequential or pipelined) are mentioned. Other annotations are hidden.
way is to work on portions of partitions, called buckets [9]. Taking a look at the simplest bucket based hash join, the Grace hash join [9]. In the split phase, a first hash function is applied to the tuples in order to determine the processors they must be assigned to; then a second hash function is applied to determine the number of buckets. In the local join phase, the relations buckets to be joined are successively loaded into main memory and a classical hash-join is applied.

Practically, the first bucket of the first relation is loaded and a build hash table operator performed. Then, the first bucket of the second relation is loaded and a probe hash table operator applied. After that, the second bucket of the first relation is loaded and the hash table built and so on and so forth. Unfortunately a DP graph cannot model such bucket based join algorithms, because with only one build hash table on bucket operator connected by a sequential data dependency to a probe hash table on bucket operator, the loop phenomenon between these two operators is not represented. Therefore, it is necessary to introduce a new kind of dependency called loop dependency between the two operators, never considered before: A loop dependency is based on a sequence of basic and communication operators lying on a same data dependency path. It indicates that this sequence must be repeated as many times as there are available buckets (or tuples).

Loop dependencies are represented by dotted directed edges (see fig. 2). A DP graph including loop dependencies will be called DPL graph (for data, precedence and loop dependencies).
4. Using DPL graphs for designing a parallel query optimizer

This section analyzes how a parallel query optimizer can take advantage of DPL graphs to improve its optimization effectiveness. The DPL graphs have been integrated in the design of our query optimizer MPO [13].

Let us concentrate on precedence dependencies. Precedence dependencies allow the modeling of parallel processing strategies in which operators are ordered without reference to a data stream. This typically occurs if resources must be optimized, e.g. the available memory. Considering an example three-way join over a relation schema extracted from the TPC-D standard benchmark [14] (Scale Factor 1000)\(^6\). The following table 1 shows the relation definitions and size of the related hash-tables (simplified) of the four base relations after the selections (sel.) and projection (proj.) of the query formulated above.

<table>
<thead>
<tr>
<th>Relation schema</th>
<th>Size in MB after sel./proj.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PART(Partkey, Name, Type)</td>
<td>200</td>
</tr>
<tr>
<td>LINEITEM(Partkey, Orderkey, Linenum, Quantity)</td>
<td>60000</td>
</tr>
<tr>
<td>ORDER(Orderkey, Custkey, Totalprice, Orderdate)</td>
<td>15000</td>
</tr>
<tr>
<td>CUSTOMER(Custkey, Name, Nationkey)</td>
<td>150</td>
</tr>
</tbody>
</table>

Our sample query lists all the customers coming from France (Nationkey = F) and having ordered certain types of part (Type between type1 and type2). The realization of the query consists of a three-way join \(\text{PART} \bowtie \text{LINEITEM} \bowtie \text{ORDER} \bowtie \text{CUSTOMER}\).

Let us assume, that the target machine is based on a shared nothing architecture with 10 processors. The memory size dedicated to the query execution is supposed to be 25 MB on each processor. Furthermore, each join is executed by a hash based method. In order to minimize I/O costs, the hash tables are built on the smaller of the two input relations.

Here, most classical optimizers would only consider linear trees, i.e. one of the two input relations is a base relation, and would discard the possible bushy tree, as shown in fig. 3, left scheme. This is because the memory requirement of the two parallel joins \(\text{CUSTOMER} \bowtie \text{ORDER}\) and \(\text{PART} \bowtie \text{LINEITEM}\) is at least 350 MB (the hash table of the two relations \text{PART} and \text{CUSTOMER} must be held in main memory) and exceeds the total available memory size (250 MB = 10 processors times 25 MB) of the parallel machine. However, this decision excludes the cost-effective bushy tree, implementing a strategy which minimizes the number of processed tuples (about 50% lesser tuples than the best linear tree\(^6\)).

A query optimizer working on DPL graphs can keep the shape of the bushy tree and thus conserve the quality of minimizing the number of processed tuples.

\(^6\)Several Scale Factors can be implemented within the TPC-D benchmark, the Factor 1000 corresponds to a database size of 1 TB.

\(^6\)Such situation is not unusual in complex decision support queries, as many small relations describing the characteristics of a key are joined with very big relations which combine several keys.
This is done by serializing the execution of the two joins, Customer ⋈ Order and Part ⋈ Lineitem, i.e. the two joins are executed one after the other. Thus, the minimal memory requirement reduces to 150 MB, which is clearly below the total available memory size of 250 MB. Such execution scenario is "bushy like" and will furthermore be called serialized bushy trees.

![Bushy tree based on join nodes](image)

Fig. 3. Insertion of a precedence dependency in a bushy tree.

It remains the question either to execute first Customer ⋈ Order or Part ⋈ Lineitem. This decision is involved in the parallel optimization process by computing the costs of both alternatives. For our example, it turned out that the lowest cost strategy executes Customer ⋈ Order before the Part ⋈ Lineitem. Thus, the optimizer chooses to insert a P-edge from the probe hash table Order to the build hash table Part operator and from the probe hash table Lineitem to the any-to-any redistribution of T1. The latter P-edge is required to guarantee that the hash-table on T1 is only built when the memory space occupied by the hash table of Part has been freed.

This example shows that a parallel optimizer limited to data dependencies, by excluding sub-search spaces, risks to miss optimal processing strategies. Oppositely, the representation power of DPL graphs allows one to take into account original execution strategies while keeping the representation model compact.

5. Using TB-nets for modeling the control and data flows in DPL graphs

Timed Petri nets are a powerful and simple theory to model the data flow in distributed and parallel systems [2,15,16]. The reason is that this formalism allows the representation of the control and processing strategies while being independent from an underlying machines architecture. However, to the best of our knowledge,
timed Petri nets have not yet been considered for modeling the control and data flows of parallel relational query processing neither as the base for any simulation tool.

In a timed high-level Petri nets (THLPN) [17], data is associated with tokens and functionality of the operators is associated with transitions. Transition firings are used to model events, i.e. the execution of one operator vertex in the DPL graph. The event occurrence is enabled with the availability of suitable data, represented by tokens in the input places. The event consumes the enabling tokens and produces new data in the output places. Among, the data associated with tokens, a timestamp is used to represent the time at which the token is created. The timestamp associated with the tokens produced by the firing of a transition depends on the timestamp associated with the removed tokens. The dependency is modeled by a time consuming function associated to the transitions. Furthermore, predicates control if a transition can fire or not.

This class of Petri nets is the best adapted to our problem. The reason lies in the fact that both data and functionality aspects can be handled in THLPN, in addition to flat and colored Place/Transition Petri nets [19]. **Time Basic nets (TB-nets)** [18] were chosen as the concrete THLPN model, because they offer both the associations of events and predicates to transitions. Furthermore, a simulation tool of TB-nets has been made free-ware. We make use of this tool to implement a simulation environment for the execution of the DPL graphs (section 6).

5.1. **TB-nets**

**TB-nets** [18] are an extension of the Environment/Relationship (ER) nets, where the tokens are environments on a set of variables identifiers ID and a set of values V, i.e. functions ID → V. Let ENV = VID be the set of possible environments. Furthermore for a given transition t, let t denotes the preset of the transition t, i.e. the set of places connected to t by an arc entering t and t′ be the post set of transition t, i.e. the set of places connected to the transition t by an arc exiting t. The event associated to transitions t is defined as the relationship ([|t|] denote the number of elements in the set t):

\[ a(t) \subseteq ENV^k(t) \times ENV^n(t) \]

where \( k(t) = ||t^e|| \) and \( h(t) = ||t^s|| \)

The projection of \( a(t) \) on \( ENV^k(t) \) is called the predicate of the transition t.

In order to account for pipeline parallelism, the original definition of [18] is slightly modified by introducing two timestamp fields associated to the tokens, instead of one. The first timestamp field represents the simulation time at which the first page is output by the creating transition. The second timestamp field represents the simulation time at which the last page is output by the creating transition. Each transition is associated with time functions which describe the relation between the timestamps of the tokens removed by the firing and those produced by the firing.

---

1Note that time can also be associated to transitions and places. However, these variants are more adapted for stochastic Petri-nets [18].
5.2. Transformation of the operator vertices

The operator vertices of DPL graphs are directly transformed into transitions. The D- or P-edges connecting two operator vertices will be modeled by two transitions, which represent the two operator vertices and a place, which is connected to the two operator vertices.

![Diagram of DPL graph transformation](image)

Fig. 4. Transformation of a sample DPL graph of two basic operators to a TB-net.

Let us consider the example of a DPL graph consisting of two basic operators select and project, connected by D-edges. The project is assumed to start, when the select outputs its first page. The related TB-net is shown in fig. 4. In the graphical representation of a TB-net, the fields associated with tokens are described as structured data. To denote the field $x$ of any token stored in place $p$, the notation $p.x$ is used. As described above, two timestamp fields are required for the token, $p.time1$ (representing the time when the timestamp creating transition put out the first page) and $p.time2$ (representing the time when the timestamp creating transition put out the last page). Their values are modified in the select and project transition.

![Diagram of TB-net control schema](image)

Fig. 5. Left scheme: DPL graph for the choose operator. Right scheme: Corresponding TB-net control schema.

The control operator requires the introduction of predicates. For example consider the choose operator [12] for an alternative execution between an index-scan or a seq-scan access of a relation $R$ (fig. 5, left). The run-time decision between the two alternatives is modeled in the TB-net (fig. 5, right) with the introduction
of a supplement token field `scan` (type integer). It is initialized in the transition related to the `choose` control operator, i.e. `scan` is set to 1, if an `index-scan` is the best strategy for the current execution environment. Based on the token value of `scan`, the predicate associated to the `index-scan` and `seq-scan` transitions decides if the transition fires or not, i.e. `index-scan` will fire if the value of the field `scan` of the input token is 1.

5.3. Transformation of the loop dependency

The control schema for the L-edge requires a complex TB-net construction:

---

Fig. 6. Transformation of an L-edge to a TB-net.

Fig. 6 shows the TB-net construction associated to a sample L-edge. The operators lying on the loop are represented by the transitions noted $t_i, \ldots, t_k$. The L-edge is represented by three places $(p_{\text{NEOS}}, p_{\text{EOS}}, p_{\text{INI}})$ and two transitions $(t_{\text{NEOS}}, t_{\text{EOS}})$. A token must be put in $p_{\text{INI}}$ and $p_i$ and the field EOS (type boolean) must be associated to the tokens. The latter field is initialized by the transition $t_k$, corresponding to the last operator lying on the loop.

In order to express correctly the loop, an event is associated to $t_k$ which sets the value of the field EOS, i.e. it is set to false if buckets are still available. Then if the predicate associated to $t_{\text{EOS}}$ : $p_{\text{EOS}}, \text{EOS} == false$ is fulfilled, the only firing in the net is those of $t_{\text{NEOS}}$. This provides the places $(p_i, p_{\text{INI}}, p_1)$ with a token and reexecutes the loop. If the predicate $p_{\text{EOS}}, \text{EOS} == true$ associated to $t_{\text{EOS}}$ is fulfilled, only the firing of $t_{\text{EOS}}$ is enabled. This provides the place $p_{\text{INI}}$ with a token (in order to come back to the initial state) and fires $t_{k+1}$, which continues the query execution after the loop.
6. Designing a simulation tool for DPL graphs processing

Section 5 has demonstrated how the TB-nets model the data and control flows in DPL graphs. This point is especially interesting because a TB-net free-ware simulator exists: Cabernet [20] (Computer Aided software engineering environment based on ER-NETs).

The transformation of a DPL graph to the TB-net input structure of Cabernet can be easily automated by applying the proposed transformation rules. That means, first the operator vertices of the DPL graph are replaced by transitions (special attention must be paid to the control operators, see above). Second, all edges of the DPL graphs are replaced (the L-edges need the special TB-net construction, as proposed above). Third, the initialization and terminating nets are added. Fourth, the time functions to compute the local execution times of each operator must be implemented.

The proper simulation can be run in a step by step or automatic mode. Running the simulation in the step by step mode allows the redefinition of the simulation environment. Thus load imbalance can be interactively introduced by a modification of the transition timestamp value. The structure of the DPL graph itself can also be easily modified thanks to the graphical representation tool. In the automatic mode, the net executes until it arrives in a stable state, where no more transitions can be fired, or when a deadlock situation occurs. A trace file specifies the sequence of fired transitions and places. Other informations about the simulation run can be shown from the circulated token, e.g. the global simulation times value attached to the token.

7. Conclusion and future work

This paper has described a novel theoretical model for representing parallel relational query processing. In comparison to previous approaches, DPL graphs combined with Tamed Basic Petri (TB) nets allow one, at first, to very accurately represent most of the parallel execution strategies, i.e. the introduction of the precedence dependency allows the modeling of operator orderings without reference to a data stream and the loop dependency the modeling of bucket processing. At second, one can precisely represent the data and control flows of those DPL graphs with the token driven semantics and the time behavior of the TB-nets. This formalism was then utilized to design a powerful simulation environment, based on a Petri-net simulation tool. Furthermore, we demonstrated on an example how a parallel query optimizer can take advantages of DPL graphs and their related serialized bushy trees.

We study the interfacing of our model with a token driven execution environment, developed at the Budapest university. At time this will enable us to propose a complete query processing system.

---

8We have not yet implemented an automatic transformation tool. All tested DPL graphs were transformed by hand.
References


