Adaptive Distributed Query Processing

Yongluan Zhou

Department of Computer Science
National University of Singapore
zhouyong@comp.nus.edu.sg

Abstract

For a large-scale distributed query engine, which supports long running queries over federated data sources, it is hard to obtain statistics of data sources, servers and other resources. In addition, the characteristics of data sources and servers are changing at runtime. A traditional distributed query optimizer or centralized adaptive techniques is inadequate in this situation. In this paper, we introduce a new highly scalable distributed query processing mechanism called SwAP (Scalable & Adaptable query Processor). SwAP can quickly learn and adapt to the fluctuations of the selectivities of operations, the workload of servers, as well as the connection speed without any statistics, and accordingly change the operation order of a distributed query.

1 Introduction

In a large scale distributed system, it is often very difficult to find an optimal plan for a query. This is because a query processor may not have accurate statistics of the participating relations stored at other nodes and the selectivities of operations. Moreover, the workload and network bandwidth of the processing servers may change during runtime. This problem is particularly severe in systems supporting continuous queries, which will run for a long time.

There have been a lot of works on adaptive query processing that address this problem [2]. Most of them are mainly focused on centralized processing environments. However, in many cases, data sources are geographically distributed and the query engine is inherently distributed. Furthermore, it is obvious that the number of queries and size of data a single server can handle are limited. Therefore, there is a need to design a distributed query engine that adapts to the changing environment during runtime.

The ultimate aim of the PhD thesis is to build up the infrastructure for a highly adaptive distributed query processing engine that should have the following features:

- The system is highly distributed. It supports both continuous queries and ad-hoc queries over federated data sources.
- The system can adaptively approach to an optimal processing plan with little or without statistics. It can also adapt to the fluctuations of environment mentioned above.
- Based on the adaptivity mechanism, the system can support QoS management for user queries.

To date, we have worked out a new distributed query processor, called *SwAP* (Scalable & Adaptable query Processor). The system builds on and goes beyond a straightforward adaptation of Eddies [1], a centralized adaptive query processing mechanism, to reorder operations of a distributed query plan at runtime. The reordering of operations is realized by dynamically changing the orders in which tuples are routed through the processing sites according to fluctuations in the selectivities and cost of operations, as well as the workload and connection speeds of servers. As a result, SwAP can lead to an optimal plan.

SwAP harnesses both the *horizontal* and *vertical* parallelism between processing sites. For both types of parallelism, there is an eddy at each site providing adaptivity for operations running locally. Vertical parallelism offers greater opportunity for adaptivity. In particular, we propose a new mechanism for vertical parallelism to learn the selectivity, workload and connection speed of the processing servers, and then accordingly adapt the orders in which tuples are routed through the servers. There are two key components in this scheme: *Remote Meta-Operator* (RMO) and *Virtual Tuples*. Each RMO represents the operations running on one remote site. Sending tuples to a RMO means sending a tuple to that remote site for processing. Virtual Tuples are sent back by the remote site for gathering statistics about operations running on the remote site, e.g. tickets in the lottery routing scheme. We also proposed a variant of the SteM [5] to reduce

the overhead when dealing with the intermediate tuples from a remote site. Furthermore, this mechanism can continuously adapt to the runtime changes of the characteristics of the servers mentioned above.

We first review related work in Section 2. Details of the design of SwAP are given in Section 3. Finally, we conclude in Section 4 with our agenda for future work.

2 Related work

Due to the space limit, we only reviewed the most related work here. Eddy [1] is actually an iterator interposed between operators and source data. Operators are continuously fetching tuples via eddy and may return the result tuples. By routing tuples through operators in different orders (under a routing scheme), eddy is able to adaptively change the order of operations during runtime without generating a query plan. The authors also introduced the back-pressure effect and lottery routing scheme, that enable eddy to adaptively observe the operator behavior (cost and selectivity) and thus route tuples through operators in an order approaching the optimal plan. The idea of back-pressure effect is that a high cost operator consumes tuples more slowly and thus forces the eddy to route tuples to lower cost operators. Under the lottery routing scheme, each operator is assigned a number of tickets. When two operators vie for a tuple, the operator with more tickets will "win" the tuple. An operator gets a ticket when a tuple is routed to it and looses a ticket when it returns a tuple to the eddy. Thus the number of tickets can be used to estimate the selectivity of an operator.

SteMs [5] extend eddies by splitting up the join operator into two *state modules* called SteMs. One SteM is created for each base relation addressed in a query. Tuples arrived are first built into its own SteM and then used to probe the other relations' SteMs to get the join results. By probing SteMs in different orders, the join ordering, join algorithm and the spanning tree (for cyclic queries) can be adapted. SteMs also provide a shared data structure for data from a given table, regardless of the number of access methods. This facilitates the access method adaptation.

A good survey of adaptive query processing can be found in [2]. Among these works, [4] and [3] have addressed the problem of inaccurate or unavailable statistics of query optimization. However, they did not address the problem of inaccurate estimation of workload and connection speed of servers and their fluctuations. In addition, their solution can only re-optimize the remaining part of the query plan after materializing the intermediate results. That means not only increasing the I/O overhead, but also interrupting the pipelined processing of the query. DB2's LEO [7] is an example of another direction of adaptive query processing. It computes adjustments for the statistics while processing queries, and hence it can benefit from the adjustments when optimizing subsequent queries.

Flux [6] is a recent work on introducing adaptivity into parallel query processing. In the scheme, operators are horizontally distributed across a cluster. Flux provides load-balancing by online repartitioning of the data and shipping the states of the corresponding operators. However, only horizontal parallelism is supported. Our scheme can harness both horizontal and vertical parallelism. Clearly, we can also incorporate load-balancing capabilities of Flux into our scheme.

3 Parallelism in SwAP

In SwAP, the operations of a distributed query plan running on multiple sites are continuously reordered to adapt to changing factors. SwAP harnesses both horizontal and vertical parallelism. In both cases, there is an eddy running on each processing site, responsible for dynamically adapting the order of operations running locally. For the case of vertical parallelism, there is a new mechanism for adaptively changing the order of tuples routed through different sites. In this section, we will present the schemes to support the two types of parallelism and an overview of the whole process of query processing in SwAP. Before that, we shall introduce the preparatory phase and discuss how a distributed query plan is generated and set up.

3.1 The preparatory phase

SwAP adapts the operations of a distributed query plan at runtime. Prior to that, a distributed query plan must be generated and set up. This is accomplished by the *preparatory phase*. Given a query, a light-weight optimizer produces a *Distributed Processing Graph* (DPG)[8]. The DPG is an annotated graph that captures the processing sites of operations and the types of parallelism to be employed. The optimizer may produce a spanning tree for a cyclic query if needed. Traditional distributed query optimization techniques can be applied here. We postponed the work of making these decisions adaptive as our future work. For other optimization options, such as choosing the join algorithms and access methods, adaptivity can be achieved in the same way as discussed in [5]. An algorithm is then launched to set up the whole processing plan, including incorporating the *Remote Meta-Operators/Remote Output Operators* (RMO/RO) that represent operations running at remote nodes, and *remote access modules* (RA) that represent remote data sources. We would like to refer interested readers to [8] for further details of the preparatory phase.

3.2 Scheme for horizontal parallelism

In this scheme, different sites are running independently either on different partitions of data or on independent subtrees of a complex query tree, correspond-

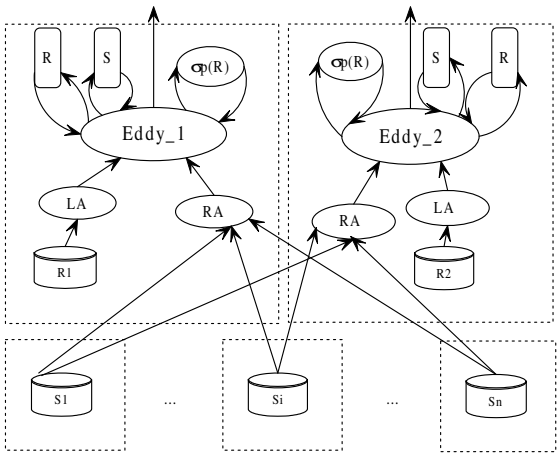

Figure 1: An example of scheme for horizontal parallelism. LA denotes local access module; RA denotes remote access module.

ing to the intra-operator parallelism and bushy parallelism. For both cases, there are an eddy and the required operators running on each processing site. In the first case, each eddy has the same number of operators but on different fragments of the source data. Complete results can be obtained by performing a union operation on the output of all the processing sites. The complete results may be further processed if they are only intermediate results, or be output to the user if they are final results. In the second case, the different subtrees running on different sites can be executed independently and simultaneously. An example of these subtrees are subtrees belonging to different branches of the same node in the query tree. For both cases, each eddy is running independently to other eddies and provides adaptivity of operations running on its own site. Therefore, the mechanism of centralized eddy can be directly applied in this scheme.

Figure 1 is an example of the intra-operator parallelized scheme. In this example, the query is $R \bowtie S$. $R1$ and $R2$ are two fragments of the relation R residing on two different sites. All fragments of S are sent to these two sites to perform the join operation. Eddy_1 and Eddy_2 are running independently to each other and provide adaptivity of operations running on their own sites.

3.3 Scheme for vertical parallelism

In this scheme, a query is split up into several pipelined sub-queries. Each sub-query is assigned to one processing site. Sites are running in a pipelined manner. The output of one site is the input to another site(s). Tuples have to undergo all sub-queries before they are being output as answers. An interesting problem here is that the output of one site may have the choice of being routed through other sites in different order. For example, to evaluate a three-way join $R \bowtie S \bowtie T$, where R, S and T are residing on three different sites and the two joins are to be evaluated in site R and site T respectively, we can first route tuples of S to site R to evaluate $R \bowtie S$ then to site T to perform the join with T to get the final result, or we can route the tuples of S to site T first and then to site R. This is actually the operation ordering problem. A good choice of this order should balance the workload of servers while minimizing the cost of communication and other system resources. We believe this is where traditional query optimization is inadequate as a static query plan that fixes the order in which tuples are routed and hence would be unable to adapt to inaccurate estimations or changes in workload of servers. Our scheme, however, makes the routing decision at runtime and thus can potentially balance the workload of servers, and minimize the communication cost and response time. The decision of choosing which site to output is done by continuously measuring the workload, connection speed and the selectivity of operations of the candidate sites. Moreover, this is done in a distributed manner, i.e., each site is making the decision for its own output. Therefore, our scheme is highly scalable.In the following sub-sections, we first introduce the key components of our scheme and then provide an illustrating example.

3.3.1 Remote meta-operator and virtual tuples

The first key feature is the *Remote Meta-Operator* (RMO). A RMO can be viewed as a local representation of the operations running on a remote site. It is responsible for transmitting intermediate results of the local site to remote sites for further processing, and it also collects statistical information about operations running at remote sites (through the concept of *Virtual Tuples*). For a site that needs to make the decision of choosing a site from n candidate sites to output its intermediate results, we will attach to the eddy n RMOs each corresponding to a remote site. From the view of the eddy, this type of operator is like a regular operator, continuously fetching tuples from the eddy and returning "tuples" to the eddy. However, the "tuples" returned to the eddy are called *Virtual Tuples* and are not typical data tuples. In fact, they do not contain any data, i.e., has zero data length. The operator is also continuously sending tuples to its corresponding remote sites and receiving virtual tuples from them. However, in order to minimize the communication overhead, the remote sites only return the number of *virtual tuples* to be generated to the RMO, and then the RMO will generate the virtual tuples and return them to the local eddy.

For a site that can only transmit its intermediate results to a single remote site, we can attach a *Remote Output* (RO) operator to the eddy rather than a RMO. A RO operator only continuously sends intermediate

results to the corresponding remote site and does not receive virtual tuples from the remote site.

The virtual tuples returned by the RMO can be used to gather statistics of operations running on the remote site. Here we only focused on the lottery routing scheme, where virtual tuples are counted in the lottery routing scheme of the local eddy. Therefore, by using the lottery routing scheme, we can learn the selectivities of processing sites and adaptively change the decision on which site to output the intermediate results. Furthermore, sites with lower workload consume tuples more quickly, while sites with higher workload consume tuples more slowly. Similarly, sites with slower connections to the local site consume tuples more slowly. Under the effect of back-pressure (limited queue size), more tuples are routed to the sites with lower workload and faster connections. Therefore, our scheme can also adapt to the fluctuations of the workload and the connection bandwidth of the processing sites. Moreover, all these decisions are done in a distributed way, i.e. sites are making decisions for their own output. This means that our scheme is highly scalable and is not limited by the number of processing sites.

3.3.2 Intermediate tuples

In a pipelined parallelism, a result tuple must have undergone all the sub-queries. To facilitate the routing of tuples, we attached to each tuple a bit vector called *Global Footprint*, where each bit corresponds to one sub-query. Setting a bit in the global footprint means the tuple has undergone the corresponding sub-query. Eddy is based on a tuple's global footprint to determine whether it can be routed through a RMO/RO operator [8].

For a vertically parallelized plan, the intermediate results of a processing site will be transmitted to other sites for further processing. In our scheme, the sites receiving the intermediate tuples from other sites treat them as coming from virtual data sources. Hence, there can be more than one sub-query running on a single site. For example, for the three-way join example $R \bowtie S \bowtie T$ stated at the beginning of this section, under our scheme, there are two joins running on site R which are $R \bowtie S$ and $R \bowtie ST$, where ST is the intermediate joining results of a portion of relation S and the whole relation T. Similarly, there are two queries running on site T. To avoid unnecessary overhead, we use only one SteM for all types of virtual sources containing the same base relation. Building tuples into the SteM is performed by the corresponding access module. The access module knows exactly which fields are used to build the tuples into the SteM. And we also need to add one predicate to each of the other SteMs involved in the join for each type of virtual sources. In this way, all sub-queries require tuples to undergo the same operators and hence there is no need to maintain information for the different sub-queries.

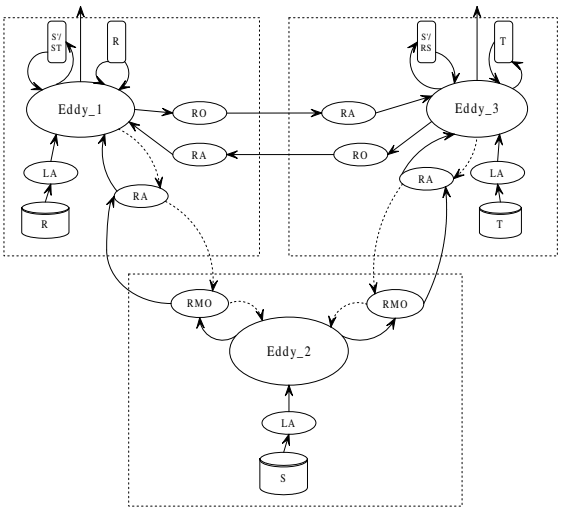

Figure 2: An example of the scheme for vertical parallelism. RA denotes remote access module, LA denotes local access module.

3.3.3 An illustrating example

Figure 2 is an illustrating example of the vertically parallelized case. The query in this example is a three-way join $R \bowtie S \bowtie T$. We assume the three relations are residing on three different sites. The query is split into two joins and they are evaluated in site R and site T respectively. And it is also assumed that each source is located on only one site. Note that these assumptions are made for purpose of illustration; our scheme does not impose such restrictions. On each site, there is an eddy that provides online re-ordering of operations executed locally. For the eddy at site S, there are a local access module to access tuples of relation S and two RMOs to represent the operations executing at the other two sites. For example the left RMO of Eddy_2 represents all the operations executing on site R. The RMOs are continuously fetching tuples from and returning virtual tuples to Eddy_2. The lottery routing scheme [1] can be directly applied here and thus together with back-pressure effect, Eddy_2 can adaptively choose to output intermediate results to the site with more selective operations, lower workload and faster connection speed.

At site R, there are a local access module to access the relation R, two SteMs for the evaluation of the join operation, two remote access modules to retrieve tuples of relation S and intermediate tuples from site T, and one RO operator to output the intermediate results. The intermediate results from site S and site T are represented as tuples from two separate virtual sources: S' and ST, accessed by two separate access modules. Other than using separate SteMs for the two sources, we use only one SteM and tuples are built into the SteM using the same fields from base

relation S by the access module. There are actually two join operations: $R \bowtie S$ and $R \bowtie ST$ and hence there are two predicates in SteM of R. When the eddy routes a result tuple of $R \bowtie S$ to the RO operator, it will detect that the tuple contains an intermediate result tuple from site S which needs the returning of virtual tuples. Then the eddy will generate a virtual tuple by calling a function of the representation object of virtual source S'. The representation object will accumulate the number of virtual tuples to be sent, and when the number reaches a threshold, it sends the number of virtual tuples to the corresponding RMO. The dotted curves in the figure indicates the flow of virtual tuples. Similar processing is performed at site T.

3.4 Overview

Here we present the whole process of query processing of our scheme. When the system receives a query submitted by the user, it launches a preparatory phase to set up the distributed processing plan. The query is split up into several sub-queries, meanwhile the mode and degree of parallelism as well as the sites to execute the sub-queries are also determined. At each site, there is an eddy and the required operators running to adaptively evaluate the sub-query. If a site needs to adaptively choose a site to output its intermediate results, the system will attach to the eddy the same number of Remote Meta-Operators as the candidate output sites. Otherwise, it simply attaches a Remote Output operator to the eddy, which only outputs results to a single remote site.

4 Conclusions and future work

In this paper, we have presented a novel distributed query processing scheme which can adaptively learn the selectivity, the workload as well as the connection speed of servers. And when these properties change during runtime, our system can also adapt its behavior accordingly to approach an optimal plan. Moreover, in the proposed scheme, all runtime decisions are made in a distributed manner. Hence it is highly scalable. In addition, the proposed scheme is also applicable to parallel query processing. At the time of writing this paper, we have already implemented the prototype of the system and we are under the process of evaluating the system. Results will be reported soon.

The current result is only the first step in our research agenda. There are some problems we are going to consider to complete the PhD thesis. In particular, we will focus on the following. First, the current scheme still needs an pre-optimizer to make the decision of how to split up the query into several subqueries and how to choose sites as processing sites. One can adaptively change the former decision by adaptively merging and splitting operations running on different sites. Mechanisms and policies of merging and splitting of operations are needed to be considered. To make the second decision adaptive, one can try to run the query on the candidate sites at the same time and learn the cost during runtime. The challenge here is how to minimize the duplicates and minimize the communication and system overhead. Second, under the situation of numerous queries running on the system, sharing the computation, storage and network bandwidth across queries is essential. In the current scheme, Remote Meta-Operator and Remote Output are used only for one query. Sharing them across multiple queries can reduce the consumption of network bandwidth and system resources for maintaining multiple connections.

Yet another direction is to enhance the query engine to support QoS management for the user. This includes defining QoS properties and the problem of how the system adapts its behavior during runtime to maintain the QoS requirements. Some possible QoS specifications are: the completion time or the result output rate of the query; the freshness of the data; the number of result tuples; the affordability of the user; the penalty of delaying or halting a query, etc.

References

[1] R. Avnur and J. M. Hellerstein. Eddies: continuously adaptive query processing. In *SIGMOD*, 2000.

[2] J. M. Hellerstein, M. J. Franklin, S. Chandrasekaran, A. Deshpande, K. Hildrum, S. Madden, V. Raman, and M. A. Shah. Adaptive query processing: Technology in evolution. *IEEE Data Engineering Bulletin*, 23(2):7–18, 2000.

[3] Z. G. Ives, D. Florescu, M. Friedman, A. Levy, and D. S. Weld. An adaptive query execution system for data integration. In *SIGMOD*, 1999.

[4] N. Kabra and D. J. DeWitt. Efficient mid-query reoptimization of sub-optimal query execution plans. In *SIGMOD*, 1998.

[5] V. Raman, A. Deshpande, and J. M. Hellerstein. Using state modules for adaptive query processing. In *ICDE*, 2003.

[6] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *ICDE*, 2003.

[7] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. LEO - DB2's LEarning Optimizer. In *VLDB*, pages 19–28, 2001.

[8] Y. Zhou, B. C. Ooi, and K.-L. Tan. SwAP: A scalable and adaptable distributed query processor. Submitted for publication, 2003.