Practical Adaptation to Changing Resources in Grid Query Processing

Anastasios Gounaris, Norman W. Paton,
Rizos Sakellariou, Alvaro A.A. Fernandes
Univ. of Manchester, Manchester M13 9PL, UK
{gounaris,norm,rizos,alvaro}@cs.man.ac.uk

Jim Smith, Paul Watson
Univ. of Newcastle-Upon-Tyne
Newcastle upon Tyne, NE1 7RU, UK
{jim.smith,paul.watson}@ncl.ac.uk

1. Introduction

Grid computational resources, as well as being heterogeneous, may also exhibit unpredictable, volatile behaviour. Therefore, query processing on the Grid needs to be adaptive in order to cope with evolving resource characteristics, such as machine load and availability. To address this challenge in a Grid environment, the non-adaptive OGSA-DQP [1] system described in [1] has been enhanced with adaptive capabilities.

In most cases, the physical machines that are made accessible via Grid middleware to participate in the wide area processing of a typically long-running distributed query, are not dedicated. As such, many other tasks may be running simultaneously, aggravating the execution imbalance problems that naturally arise from resource heterogeneity in terms of CPU speeds, network bandwidth and maximum amount of memory available. Dynamic rebalancing by redistributing workload (i.e., the tuples waiting to be processed), based on runtime measurements of machine throughput, is a promising way to improve the performance of query processing. Indeed, it may be beneficial either to rebalance the load over machines that are already being used to evaluate a query, or to change the machines associated with a query at runtime. For example, a part of a query execution plan may form a bottleneck, and, to address this problem, more machines may need to be allocated. In addition, non-dedicated resources are prone to failures, and it is often better to recover from such failures in a targeted way than to restart processing the query from scratch.

Taking into consideration the above characteristics of, and problems associated with, a Grid environment, OGSA-DQP has been extended to support both adaptive workload partitioning and resource allocation in response to changes in resource performance and in the resource pool, respectively. This demo focuses on the following adaptivity features:

- **dynamic workload distribution**, to ensure that each machine receives a fair amount of workload according to its runtime behaviour;
- **dynamic resource allocation**, to benefit from new resources becoming available after the query plan has started executing; and
- **failure recovery**, especially in cases where the failed node stores intermediate state that needs to be recreated to guarantee the correctness of the final results.

[1] Publicly available from http://www.ogsadai.org.uk/dqp

2. Adaptations in a Grid context

In OGSA-DQP, query evaluation is implemented as a collection of interacting services, where multiple evaluation services on different grid nodes evaluate different plan fragments, potentially using partitioned parallelism, as described in [1]. In the adaptive extensions to OGSA-DQP, the services have been extended to support notification of monitoring information, and to include messages for adapting query evaluation along the lines described in Section 1.

Figure 1 shows an example parallel query plan fragment, which retrieves data from *node1* and *node2*, and joins them on *node3* and *node4* to benefit from partitioned parallelism. The *exchange* operators [3] are responsible for data communication and tuple routing. To achieve this, when they act as data producers, they hold a distribution vector *dv*, which describes the portion of the workload each of their consumers will receive. Furthermore, *exchanges* have been extended with recovery logs for each outgoing edge, which cache, at any point, all the tuples sent upwards in the query plan that have not completed their processing in their parent nodes, as described in [4]. The remainder of this section describes how the mechanisms above are built on to implement adaptive strategies. Three such strategies are demonstrated: recovery from machine failure, dynamic workload balancing, and dynamic resource allocation.

2.1. Failure Recovery

Failure recovery involves detecting failures, finding replacement machines and ensuring that execution can continue after a recovery period. This phase of the demo focuses on the last aspect in its most challenging form: when

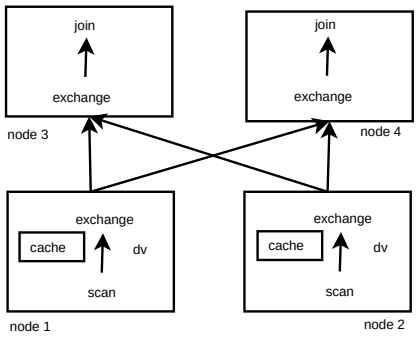

Figure 1. An example parallel query plan.

the failed machine holds state. It shows how the system incurs low overhead when there is no real failure, and an acceptable overhead when there is indeed a failure [4].

Given a query, a fault is artificially injected. This part of the demo illustrates the capability of the system to start-up a replacement evaluator, recreate on it any state that was previously on the failed machine, and rewire the data flows from and to the other evaluators so that the evaluation can continue, without any work done up to this point being thrown away. For example, suppose that *node4* in Fig. 1 fails and is replaced by a new node, *node5*. In such a scenario, *node1* and *node2* update their *dv*s to include the new machine and discard the old one, and they recreate the intermediate state that was initially on the failed machine from their local cache (recovery log).

2.2. Dynamic Workload Balancing

In adaptive OGSA-DQP, the query engine produces monitoring events that include information about the actual processing cost of each operator instance in the query plan and about the actual communication cost of each data block for each network connection. The update frequency is configurable. Based upon such events, an optimal workload distribution vector is computed for each producer *exchange* and is compared against the current one. If the two vectors differ by more than a pre-specified amount, and the query is not close to completion, the vectors are updated. If the imbalanced part of the query plan holds state, that state is redistributed according to the revised distribution vector from the cache upstream. In this way some work is migrated from slow machines to machines that could handle their load, without the former being responsible for such a task, which could add further load [2].

The demo query examined retrieves data from a store to which an expensive predicate is applied by calling a remote Web Service (WS). That is, compared with the query plan in Fig. 1, there is a single *scan*, and WS calls instead of *joins*. The call to the WS is the dominating cost for this query and is parallelised across multiple machines. It is assumed that the system does not know the different machines characteristics at compile time, and thus treats the participating machines as if they had identical capabilities. In this phase of the demo, it is shown how evaluators send updates on their processing costs and how the system can route an amount of data to each machine that is compatible with its runtime behaviour by continuously updating the *dv*s, thus improving query response time.

2.3. Dynamic Resource Allocation

There can be many different flavours of dynamic resource allocation. In this demo, a simple example is examined as follows. The system monitors query plan behaviour, and tries to find which part of the query plan is a performance bottleneck, to further parallelise it on a new machine. The response consists of the following steps: informing the evaluators that interact with the new machine, installing the new evaluator, and updating the distribution vectors of the machines sending data to the new evaluator so that the latter can receive its share of the workload. If state recreation is required, this is done from the cache upstream, as in the cases of failure recovery and workload balancing.

The scenario is the same as in the previous case. However, in this phase of the demo, the machine registry contains machines that can be used in the execution but have not been selected at compile time. The novel features of this demo include the capability of an instance of the execution engine, firstly, to spawn a new instance remotely, and secondly, to identify at runtime where the bottleneck is in the query plan, and to increase the degree of partitioned parallelism at this part.

References

[1] M. N. Alpdemir, A. Mukherjee, N. W. Paton, P. Watson, A. A. A. Fernandes, A. Gounaris, and J. Smith. Service-based distributed querying on the grid. In *Proc. of ICSOC*, volume 2910 of *LNCS*, pages 467–482. Springer, 2003.

[2] A. Gounaris, J. Smith, N. W. Paton, R. Sakellariou, A. A. A.Fernandes, and P. Watson. Adapting to changing resource performance in grid query processing. In *Int. Workshop on Data Management in Grids*, volume 3836 of *LNCS*. Springer, 2005.

[3] G. Graefe. Encapsulation of parallelism in the Volcano query processing system. In *Proc. of ACM SIGMOD*, pages 102–111, 1990.

[4] J. Smith and P. Watson. Fault-tolerance in distributed query processing. In *Proc. of IDEAS*. IEEE Computer Society, 2005.