Autonomic Query Parallelization using Non-dedicated Computers: An Evaluation of Adaptivity Options

Norman W. Paton
School of Computer Science,
University of Manchester
Oxford Rd, Manchester M13 9PL, UK
Email: norm@cs.man.ac.uk

Vijayshankar Raman, Garret Swart,
Inderpal Narang
IBM Almaden Research Centre,
650 Harry Road, San Jose CA 95120, USA
Email: (ravijay,gswart,narang)@us.ibm.com

***Abstract*— Writing parallel programs that can take advantage of non-dedicated processors is much more difficult than writing such programs for networks of dedicated processors. In a non-dedicated environment such programs must use autonomic techniques to respond to the unpredictable load fluctuations that characterize the computational model. In the area of adaptive query processing (AQP), several techniques have been proposed for dynamically redistributing processor assignments throughout a computation to take account of varying resource capabilities, but we know of no previous study that compares their performance. This paper presents an simulation based evaluation of these autonomic parallelization techniques in a uniform environment and compares on how well they improve the scaleability of the computation. The existing strategies are compared with a new approach inspired by work on distributed hash tables. The evaluation identifies situations in which each strategy may be used effectively and it should be avoided.**

I. Introduction

The cost of dedicating specific computing resources to specific operations has always been high and it is now becoming prohibitive, not because the hardware is becoming more expensive, but because the cost of managing the hardware and assigning it functions has. The lack of dedicated processing resources means that new autonomic parallelization algorithms are needed; the old approach of dividing up a problem at the start and keeping that division until the end just doesn't work well any more. Instead, algorithms have to be structured so that they can make use of any number of processors at each stage and so that processors becoming unavailable during a computation is no longer an error condition, but a normal condition.

Some problems admit simple solutions in this context, such as large scale embarrassingly parallel problems. Effort has more recently been focusing on solving problems that don't fit the embarrassing model, including on problems in query processing. Parallel query processing is now well established, with most major database vendors providing parallel versions of their products. Such systems have been shown to scale well to large numbers of parallel nodes, but are normally deployed on dedicated hardware and software infrastructures. More recently, however, internet (e.g. [4], [10]) and grid (e.g. [1], [12], [15]) query processing has sought to evaluate fragments of a query on diverse distributed computational resources identified at query runtime, benefiting from pipelined and/or partitioned parallelism to improve performance.

There are currently several different proposals for taking advantage of non-dedicated resources to support scaleable query evaluation, but these technologies have not yet been widely deployed in practice. One open issue is how best to take account of the fact that using non-dedicated resources generally involves working with partial or out-of date information on machine performance, loads or data properties. Such uncertainty means that query optimizers may make decisions on the basis of incorrect information, or that their plausible decisions may rapidly be overtaken by events. This has encouraged a number of researchers to investigate the use of adaptive query processing (AQP) techniques in distributed settings (e.g. [11], [23]).

The potential of partitioned parallelism for providing scaleable query processing over non-dedicated resources, has led to several proposals for adaptivity specifically aimed at maintaining load balance (e.g. [7], [19], [16]). However, these proposals were developed and evaluated in very different architectural contexts (service-based grids, stream query processors, and local area networks, respectively), and thus it is not straightforward from the original papers to ascertain how the approaches perform relative to each other. This paper compares these three approaches to balancing load for parallel stateful operators, and considers a further additional approach proposed here. The study explores how the proposals perform in a consistent setting, thus enabling direct comparison, over a wider range of conditions than were studied in the original papers.

The paper is structured as follows. Section II describes the adaptivity approaches evaluated. The evaluations are based on simulations of the different approaches; Section III describes the simulator and the experimental setup. Section IV describes the experiments conducted and discusses the results. Section V reviews the lessons learned.

II. Adaptive Load Balancing

In AQP for load balancing, the problem to be solved is as follows, illustrated for the case of a query involving a single join. The result of a query $A \bowtie B$ is represented as the union of the results of a collection of plan fragments $F_i = A_i \bowtie B_i$, for $i = 1...P$, where P is the level of parallelism. Each of the fragments F_i is executed on a different computational node. The tuples in each A_i and B_i are usually identified by applying a hash function on the columns to be compared in the join, thereby ensuring that each F_i contains in A_i all the tuples that match tuples in B_i. The time taken to complete the evaluation of the join is $max(evaluation\_time(F_i))$, for $i = 1...P$, so any delay in the completion of a fragment leads to a delay in the completion of the join. As such, the load balancing problem involves trying to make $evaluation\_time(F_i)$ as consistent as possible, by matching the amount of work to be done on each node to the capabilities of the node.

Load balancing is particularly challenging for stateful operators such as hash-join, as maintaining a balanced load involves ensuring that the portion of the hash table on each node reflects the required work distribution, and that changes in the work distribution to maintain load balance are preceded by corresponding changes to the hash table on each node. Therefore, assuming that tuples from A have been used to build the hash table and that the probe phase is underway, if a node N_i begins to perform less well, for example because another job has been allocated to it, maintaining load balance involves allocating some of the hash table for A from N_i to other nodes, and sending future tuples from B to the appropriate node. As the hash table for A on node N_i may be large, rebalancing the work may involve a significant overhead; the trade-offs associated with dynamically balancing load for stateful operators are studied in Section IV. [1].

A popular approach to implementing parallel query evaluation uses one of the flavors of the exchange operator [8] to distribute tuples between parallel plan fragments. Each exchange operator reads tuples from its child, and redirects each tuple to a single parent on the basis of a hash function applied to the join attributes. As an even distribution of tuples over query fragments may lead to load imbalance, exchange may deliberately send tuples to parent nodes unevenly, using a *distribution policy* to capture the preferred proportion of tuples to be sent to each of the parents. Such versions of exchange are used in several of the AQP strategies described below. Figure 1 illustrates a parallel plan for $A \bowtie B$; the dotted lines delimit plan partitions, which are allocated to different nodes. As such, the join is run in parallel across two nodes. Each line between a pair of filled circles represents communication between an exchange producer and an exchange consumer. The exchange producers on the scan nodes share a distribution policy, which ensures that tuples with matching join attributes are sent to the same machine for further processing.

[1]We note that load balancing for stateless operators, such as calls to external operations, is more straightforward than for stateful operators, in that there is no need to ensure that the state of the operator is appropriately located before changing the flow of data through parallel partitions, as discussed in [7].

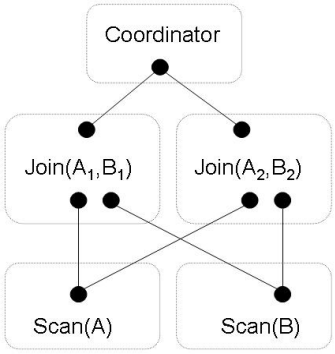


Fig. 1. Example parallel execution plan.

The adaptive strategies are described based on the specific approaches they take to: (i) *monitoring* – the collection of information about the progress of a query or the environment; (ii) *assessment* – the analysis performed on the monitoring information to identify a problem that suggests that adaptation may be necessary; and (iii) *response* – the reaction that is taken with a view to addressing the problem that has been detected.

Four adaptivity strategies are compared, and are characterized below by the nature of their response. We avoid using the original names of the proposals in the paper, as in several cases we have modified the proposal in a way that eases comparison. For example, changes include using adaptivity techniques in different architectural contexts, and either changing or providing missing values for properties such as the thresholds at which adaptive behavior is initiated. The simulation results are are not especially sensitive to small changes in these thresholds, principally because increasing the frequency of adaptation reduces the cost of making an inappropriate adaptation, effects that tend to cancel each other out.

a) Adapt-1: Sibling-based data redistribution:: in this approach, when a load imbalance is detected, portions of the hash table are relocated from more highly loaded to less highly loaded nodes. The reallocation is carried out by transferring data between sibling partitions. In the original proposal [19], the reallocation is carried out using an operator known as Flux.

- *Monitoring:* query partitions are monitored to identify the amount of contention for the computational resource on each node and the rate at which data is being processed by the partition. The contention on a node is 0.5 if it is being used half the time, and 2 if 2 jobs are seeking to make full use of the resource at the current time.
- *Assessment:* distribution policies are computed based on the current level of contention in relation to the amount of work allocated. That is, for a partition p on node n, $proposed\_distribution(p, n) = rate(p)/contention(n)$, where $rate(p)$ is the current rate at which p is processing tuples, and

$contention(n)$ is the level of contention on node n[2]. The $proposed\_distribution$ values are normalized to sum to 1, and where the proposed distribution differs from the current distribution by greater than a threshold (0.05 in the experiments), a response is scheduled.

- *Response:* Each table is considered to consist of a number of fragments (50 in the experiments) which are the unit of redistribution; the number of fragments must be larger than the parallelism level, but each fragment should also be large enough to represent a potentially worthwhile adaptation. Partitions are sorted into two lists: *producers* and *consumers*, based on how much work they need to lose or gain, respectively, to conform to the $proposed\_distribution$. Fragments are then transferred from each partition in the *producer* list to the corresponding partition in the *consumer* list until the change required is smaller than a fragment. The resulting data distribution gives rise to a new *distribution policy* for upstream exchange operators. Thus the load is rebalanced by (i) redistributing operator state among the siblings in a way that reflects their throughput; and (ii) updating the *distribution vector*s of the upstream exchange operators so that subsequent tuple routing reflects the revised data distribution.

b) Adapt-2: Cache-based data redistribution:: in this approach, when load imbalance is detected, the portion of the hash table allocated to each node is modified by sending table fragments from caches on upstream exchanges [7]. The caches are maintained by a fault-tolerance technique that allows a query to recover from node failures, as described in [20]. In essence, each time an exchange sends data from one machine to another a cache at the producer stores this data until it has been fully processed by the consumer.

- *Monitoring/Assessment:* as in *Adapt-1*.
- *Response:* as *Adapt-1*, except that fragments that are added to hash tables on specific nodes are assigned from caches in upstream exchanges rather than from sibling partitions.

The principal performance differences between *Adapt-2* and *Adapt-1* are: in *Adapt-1*, when work is to be transferred from a poorly performing node, this node must be used as the source of operator state that is to be relocated elsewhere, whereas in *Adapt-2* the state is obtained from (potentially different) upstream nodes; and *Adapt-2* has the overhead of maintaining a cache.

To limit the frequency of re-adapting, we adopt a heuristic from Flux [19] in both *Adapt-1* and *Adapt-2*, as follows: when an adaptation has taken place, if it took time *t* to relocate data for use in the hash joins, then no further adaptation is allowed until a further time *t* has passed.

c) Adapt-3: Redundant data maintenance:: in this approach, hash tables are replicated on nodes in such a way that both hash table building and probing avoids the use of heavily loaded nodes, adapting a technique originally proposed for use with distributed hash tables [21]. This technique has not been used for adaptive query processing before. Each hash table bucket is randomly assigned to three nodes (i.e. each value that can be produced by the hash function applied to the join attribute is associated with a bucket, and each such bucket is associated with three computational nodes). At hash table build or probe time, whenever a tuple is to be hashed, it is sent to the two most lightly loaded of the three candidate nodes for the relevant bucket.

When the hash table is being constructed, this behavior distributes the hash table over the three nodes in a way that guarantees that every tuple can be found on any two of the three nodes associated with the relevant bucket. The tuples that hash to a bucket are allocated to the nodes associated with that bucket in a ratio that reflects the loads on the respective machines when the hash table was being built. Each stored tuple has an extra field that records the node of its replica tuple.

When the hash table is being probed, each tuple is sent to two of the three machines associated with its bucket. The two probes are designated as *primary* and *secondary* – the primary probe is that to the most lightly loaded of the three candidate nodes, and the secondary probe is that to the second most lightly loaded of the candidates. Where the probe matches tuples, the join algorithm generates a result from the primary probe unless the matching tuple is stored only on the other two nodes.

The dynamic behavior of the algorithm ensures that work is allocated to the most lightly loaded nodes both at probe time and at build time.

- *Monitoring:* any change to the level of contention on a node is monitored.
- *Assessment:* no additional assessment is conducted.
- *Response:* the rank order of nodes used by the algorithm is obtained by sorting the nodes on their levels of contention.

The monitoring, assessment and response components here essentially update state that is used by the intrinsically adaptive algorithm to determine the routing of tuples to different nodes. This algorithm is able to react at a finer grain than *Adapt-1* and *Adapt-2*, because there is no need to relocate hash table state in response to load imbalances. However, this capability is obtained at the cost of maintaining replicas of all hash table entries, and doubling the overall number of builds and probes. As a result, this approach will be slower than *Adapt-1* and *Adapt-2* where no load imbalance occurs.

In relation to the wider AQP literature, *Adapt-3* essentially adapts continuously through query evaluation, a property shared with several other adaptive operators, such as Eddies [2] and XJoin [22]. By contrast, *Adapt-1*, *Adapt-2* and *Adapt-4* have a standard evaluation phase, which is interrupted by adaptation, after which standard evaluation continues, a characteristic shared with several other adaptive techniques, such as POP [14] and Rio [3].

[2]Different heuristics can be used for deriving a $proposed\_distribution$; [7] uses the approach described, whereas Flux [19] uses the contention directly.

d) Adapt-4: Redundant fragment evaluation:: in this approach, when a plan fragment F_i is slow to start producing tuples, a redundant copy of F_i, F_i' is created on a lightly loaded node; whichever of F_i and F_i' is first to produce data is then used as the preferred source of results [16].

- *Monitoring:* the time of completion of plan fragments is monitored.
- *Assessment:* whenever a plan fragment completes, the running of redundant plan fragments is considered. In the experiments, each time a plan fragment completes all incomplete fragments are considered as candidates for redundant evaluation by the response component. This is more aggressive at scheduling redundant evaluations than the original paper [16], as less aggressive strategies react too slowly to compete with *Adapt-1* to *Adapt-3* in most cases.
- *Response:* in the experiments, if nodes are available that are not already running part of the same query, and that have a level of contention of less than 1, then a candidate partition is chosen at random for redundant execution on each of the available nodes. In the experiments, suitable nodes were always made available. The running of such redundant fragments can, of course, slow down the existing fragments by increasing the load on shared resources such as the network or disks.

Because partitions in execution plans based on exchange tend to be closely integrated, with exchanges redistributing tuples before and after each join in a multi-join query, *Adapt-4* executes plans with less tightly coupled partitions. The resulting advantage is that individual query partitions are easier to allocate redundantly to other available nodes. The disadvantage is that the allocation of data to joins is less targeted than when exchanges are used, and thus additional work is carried out, as detailed in [16].

The approach is as follows. In parallel query processing with exchange, a hash function is used in tuple distribution to ensure that matching tuples are always sent to the same node. Thus, assuming a perfect hash function and no skew, for $A \bowtie B$ and a level of parallelism *P*, $|A|/P$ tuples will be joined with $|B|/P$ tuples on each node. If, instead of redistributing using a hash function, tuples are allocated to specific nodes in the order in which the data is read, then in $A \bowtie B$, it is necessary to ensure that every tuple from A is matched with every tuple in B on some node. If there are 4 nodes, this can be done, for example, by joining $|A|/2$ A tuples with $|A|/2$ B tuples on each of the 4 nodes, or by joining $|A|/4$ A tuples with every B tuple on every node. The most appropriate allocation strategy depends on the relative sizes of A and B, as discussed more fully in [16]. As such, *Adapt-4*, like *Adapt-2* and *Adapt-3*, accepts some overhead to facilitate the chosen adaptivity strategy.

This approach can be considered to combine some measure of fault tolerance with adaptation, in that a query may complete even when nodes running partitions fail. The other approaches do not provide this capability in themselves, although fault tolerance schemes have been proposed in association with *Adapt-1* [18] and *Adapt-2* [20].

Description	Value	Unit
Time to probe hash table	1e-7	s
Time to insert into hash table	1e-5	s
Time to add a value to fixed-size buffer	1e-6	s
Time to map a tuple to/from disk/network format	1e-6	s
CPU time to send/receive network message	1e-5	s
Size of a disk page	2048	bytes
Seek time/Latency of a disk	5e-3	s
Transfer time of a disk page	1e-4	s
Size of a network packet	1024	bytes
Network latency	7e-6	s
Network bandwidth	1000	Mb/s
Size of the exchange producer/consumer cache	50000	tuples
Size of the disk cache for workloads	50	Mb

TABLE I
COST MODEL PARAMETERS.

III. SIMULATION MODEL

The paper compares the adaptivity strategies from Section II by simulating query performance. We use simulation because this allows multiple strategies to be compared in a controlled manner with manageable development costs; the authors have previously participated in the implementation and evaluation of strategies *Adapt-2* and *Adapt-4* (as reported in [7], [16], respectively). However, such implementation activities involve substantial development effort, yield results that are difficult to compare due to the use of diverse software stacks, and restrict experiments to the specific hardware infrastructures available. We see the simulation studies as complementing and extending existing evaluations of the adaptive systems, by allowing comparisons of techniques that had previously been studied largely in isolation.

A. Modeling Query Evaluation

The simulation depends on a model of the cost of performing primitive tasks. A cost model consists of a collection of cost functions and parameters. The parameters characterize the environment in which the tasks are taking place and the tasks themselves. The properties used in the cost model in the simulations are described in Table I; these parameters are ball park numbers obtained from the execution times of micro-benchmark queries. The cost functions used in the simulations are based on those in [17].

The simulator emulates the behavior of an iterator-based query evaluator [9]. As such, each operator implements an *open()*, *next()* and *close()* interface, in which a single call to open initializes the operator, each call to *next()* returns the next tuple in the result of the operator, and a single call to *close()* reclaims the resources being held by the operator.

In this setting, the top level loop of the simulator, in which each iteration represents the passing of a unit of time, asks the root operator of a plan for the number of tuples it can return in the available time; this number is determined by how rapidly the operator itself can process data and the rate at which its

children can supply data. The operator computes, using the cost model, the number of tuples it can potentially consume from its operands, and then asks each of these operands for that number of tuples. The child operator then indicates how many tuples it can return up to the number requested, based on its ability to process tuples and the speed of its children. Each operator, in carrying out work, adds to the level of contention that exists for the resources that it uses, and computes the amount of work that it is able to do in a time unit taking into account the contention for the resources it must use.

Three operators are used in the experiments, namely *scan*, *hash_join* and *exchange*. *scan* simulates the reading of data from the disk and the creation of data structures representing the data read; *hash_join* simulates a main-memory hash join, where the hash table is built for the left input; and *exchange* simulates the movement of tuples between operators on different nodes, as discussed in Section II. In fact, *exchange* departs to some extent from the pull-based approach of the iterator model, in that exchange reads from its children into caches on the producer node as quickly as it can, and transfers data from the producer cache to the consumer cache as quickly as it can, regardless of how many tuples it has been asked for, until such time as the caches are full. As such, the *exchange*, which is typically implemented using multiple threads, can be seen as pushing data across machine boundaries, within an overall evaluation strategy in which results are pulled from the root. The simulator models the filling and emptying of caches on exchange.

B. Experiment Setup

The configurations used are as follows. A single network (modeled as an Ethernet) and type of computer are used throughout, with physical properties as described in Table I. There is assumed to be sufficient main memory in all computers to support the caches described in Table I and to hold join hash tables. In the experiments, most measurements involve a single query, but several explore the evaluation of multiple queries at the same time – we refer to the latter as workloads. All single-query tasks are assumed to run over cold configurations (the caches are assumed to start empty) and all workloads are run over pre-warmed disk caches that are assumed to contain random pages from the database.

The queries used in the experiments are described in Table II, which makes use of the relations from the TPC-H database (www.tpc.org) using scale factor 1 to give a database of approximately 1Gb – we have run experiments on databases with different sizes, but database size does not affect the conclusions of the paper so no such figures are presented. All joins are equijoins on foreign keys. In Q1 and Q2, all attributes of the base tables are included in the query results. In all other queries, in line with common cases and to prevent the shipping of the result from becoming a very significant portion of the cost, scan operators project out tuples that are around 25% of the sizes of the base tables.

The following features characterize the environments in which the experiments are conducted. All of *Adapt-1* to *Adapt-3* are run in a shared nothing configuration – when queries are parallelized over *n* machines, the data in each table is uniformly distributed over all *n* machines, and all joins are run over all *n* machines. Following [16], when evaluating *Adapt-4* we use a shared disk to store the complete database – data is accessed through a separate node representing a Storage Area Network (SAN) with the same computational capabilities as the other machines, but a virtual disk the performance of which improves in proportion to the level of parallelism being considered in the experiment. This allows comparisons to be made across consistent hardware platforms in the experiments, but may be felt to underestimate the performance that can be expected from current SANs.

Name	Query	Result Size (SF 1)	Tuple Size (bytes)
Q1	$P \bowtie PS$	800,000	295
Q2	$S \bowtie PS$	800,000	299
Q3	$O \bowtie L$	6,000,000	53
Q4	$N \bowtie S \bowtie PS$	800,000	102
Q5	$S \bowtie L \bowtie O$	6,000,000	86
Q6	$S \bowtie C \bowtie PS \bowtie O$	1,500,000	144

TABLE II
QUERIES USED IN EXPERIMENTS.

Where there is more than one join, left deep trees are used in evaluation; all joins are equijoins, ordered in a way that minimizes the sizes of the intermediate results produced. In adaptivity experiments, the simulator has monitor, assess and response components as described in Section II. Monitor events are created, and thus potentially responded to, at each clock tick in the simulator; each clock tick represents 0.1s.

The following forms of load imbalance are considered:

1) *Constant:* A consistent external load exists on one or more of the nodes throughout the experiment. Such a situation represents the use of a machine that is less capable than advertised, or a machine with a long-term compute-intensive task to carry out. In the experiments, the *level* of the external load is varied in a controlled manner; the *level* represents the number of external tasks that are seeking to make full-time use of the machine.
2) *Periodic:* The load on one or more of the machines comes and goes during the experiment. In the experiments, the *level*, the *duration* and the *repeat duration* of the external load are varied in a controlled manner; the *duration* of the load indicates for how long each load spike lasts; and the *repeat duration* represents the gap between load spikes.

In this paper, external loads affect only computational resources – only the amount of query evaluation taking place affects levels of contention for disk and network resources. This is because the adaptivity techniques being evaluated seek to overcome load imbalance in stateful operators, and the simulated joins, being main-memory hash joins, suffer from load imbalance principally as a result of changing levels of contention for the computational resources on which they are

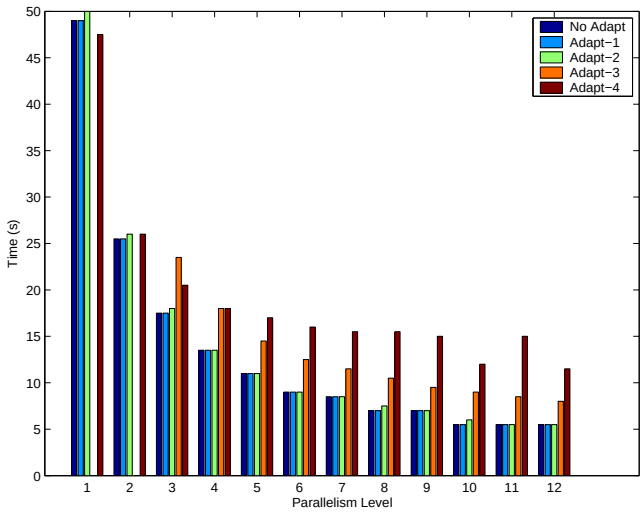


Fig. 2. Experiment 1 – response times for Q1 for different levels of parallelism.

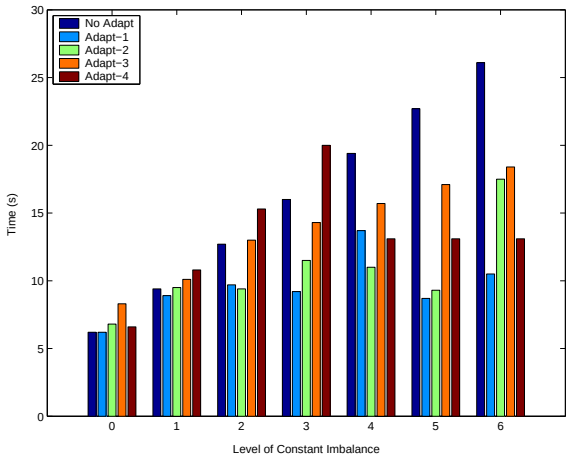


Fig. 3. Experiment 2 – Response times for Q1 for different levels of constant imbalance.

running. The simulation does not model monitoring overheads because several authors have recently reported results on the cost of query monitoring (e.g. [5], [6], [13]), indicating that different approaches can provide suitable information with modest overheads.

IV. Evaluating Adaptivity

This section explores the extent to which the adaptivity strategies from Section II are successful at improving query performance in the context of load imbalance. The experiments seek to explore: (i) the extent to which the adaptivity strategies *Adapt-1* to *Adapt-4* are effective at overcoming the consequences of load imbalance; (ii) the kinds of load imbalance that the different adaptivity strategies are most effective at responding to; and (iii) the circumstances under which the costs of adaptation are likely to be greater than the benefits.

Experiment 1: *Overheads of different strategies.* This experiment involves running query *Q1* for all four adaptivity strategies with variable parallelism levels and no external load. As such, there is no systematic imbalance, and the experiment simply measures the overheads associated with the different techniques.

Figure 2 shows the response times for Q1 for parallelism levels 1 to 12 for each of the adaptivity strategies from Section II compared with the case where adaptivity is disabled (*No Adapt*). No figures are given for *Adapt-3* for parallelism levels less than three as it needs at least 3 nodes to support redundant hash table construction. The following observations can be made: (i) As there is no imbalance in the experiments, *Adapt-1* and *Adapt-2* are very close to the *No Adapt* case; (ii) The overheads for maintaining a cache in exchange operators for *Adapt-2* are modest; (iii) The overheads associated with duplicate hash table stores and probes in *Adapt-3* are typically around 30%, although their absolute value reduces with increasing parallelism; (iv) The additional work carried out by *Adapt-4* to ease the running of redundant plan fragments is substantial. At higher levels of parallelism this redundant work is greatest compared with the work carried out in the other approaches, with the *No adapt* case often being about twice as fast as in the presence of the overheads associated with *Adapt-4*. The irregular results for the evaluation strategy associated with *Adapt-4* for larger numbers of processors in Figure 2 are facets of the algorithm used to subdivide the workload; this algorithm selects a workload allocation that minimizes the number of comparisons made by the joins.

The overall story regarding overheads associated with *Adapt-4* bears further comment, however. Although the results in Figure 2 are broadly consistent with figures presented in [16], the overall strategy assumes that input data can be pre-clustered on join attributes, thereby significantly reducing the number of comparisons carried out by many queries while still accommodating the redundant evaluation of query fragments for adaptivity. As such, the overhead associated with *Adapt-4* depends on the contribution that clustering makes to the overall performance of a query. These issues are discussed more fully in [16], and are not considered further here. In essence, as we do not model data as clustered in the experiments, the overheads associated with *Adapt-4* should be taken to be pessimistic. This affects the overall response times for queries using *Adapt-4*, but not the overall behavior of the associated adaptivity strategy, which is the principal concern of this paper.

We note that for the experiments reported here, the network is not a bottleneck, and that using a SAN as a networked virtual disk could be seen as viable for use with any of the adaptivity strategies.

Experiment 2: *Effectiveness of different adaptivity strategies in the context of constant imbalance.* This experiment involves *Q1* being run with a parallelism level of *3*, where an external load is introduced that affects one of the *3* nodes being used to evaluate the join.

Figure 3 illustrates the results for adaptivity strategies *Adapt-1* to *Adapt-4*. The increasing level of imbalance sim-

ulates the effect of having *0* to *6* other jobs competing for the use of one of the compute nodes.

The following observations can be made: (i) Where adaptivity is switched off, performance deteriorates rapidly, reflecting the fact that partitioned parallelism degrades in line with the performance of the least-effective node. Times obtained in the absence of adaptivity are referred to as the base case in what follows. (ii) Both *Adapt-1* and *Adapt-2* consistently improve on the base case, in particular in the context of higher levels of imbalance. We observe that in some cases *Adapt-1* performs better than *Adapt-2*, and that sometimes the reverse holds. This is because the timing and effect of an adaptation in these strategies depends on the precise load at a point in time, and small changes in the timing or nature of an adaptation can lead to significant changes in behavior. For example, both *Adapt-1* and *Adapt-2* can do things like adapt close to the end of the build phase, thereby making a decision on the most suitable load balance that may not be especially effective for the probe phase. In addition, neither *Adapt-1* nor *Adapt-2* assume knowledge of the input cardinalities, and thus could initiate an expensive adaptation close to the end of query evaluation. (iii) *Adapt-3* has overheads in the base case of around 30%, and performs increasingly well as the level of imbalance grows. Given constant imbalance with level of parallelism 3, *Adapt-3* essentially runs the join on the two more lightly loaded nodes throughout. As a result, *Adapt-3* is effective when the reduced level of parallelism compensates for the imbalance in the more parallel version. The cost of individual adaptations is low in the case of *Adapt-3*, so the level of improvement gained with increasing imbalance follows a regular curve. (iv) In *Adapt-4*, although the running of redundant jobs starts with lower levels of constant imbalance, the redundant plan fragment only generates results early enough to be chosen in preference to the original when the level of imbalance is quite large (4 and above in this case). Note that the decision as to whether to use the original fragment or the replacement is made as soon as either starts to produce data, and not when the fragment completes its evaluation. This explains why the response time with level of contention *3* is higher than that for level of contention *4*. With level of contention 3, although a redundant fragment is run, it starts producing data after the original fragment. The redundant fragment would have completed before the original fragment, but is halted because the fragment that is allowed to continue is the one that starts producing data first. Comparing *Adapt-4* in Figure 3 with the other adaptivity strategies, we see that running of redundant plan fragments can be effective, providing comparable performance to *Adapt-1* and *Adapt-2* and better performance than *Adapt-3* for higher levels of imbalance.

Experiment 3: *Effectiveness of different adaptivity strategies in the context of periodic imbalance.* This experiment involves *Q1* being run with a parallelism level of *3*, where an external load is introduced that affects *1* of the *3* nodes, exactly half the time (i.e. *duration* and *repeat duration* are both the same).

Figure 4(a) illustrates the results for adaptivity strategies

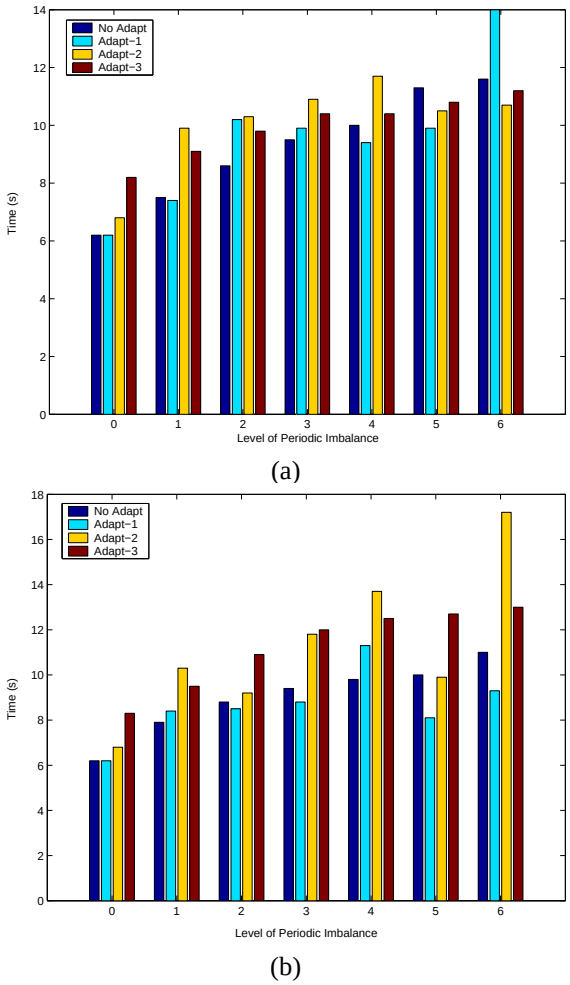
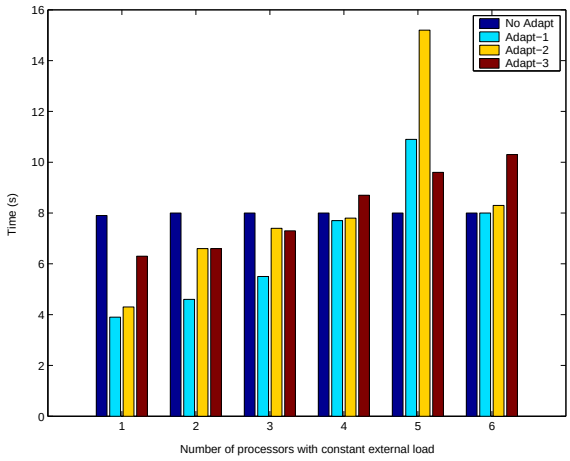
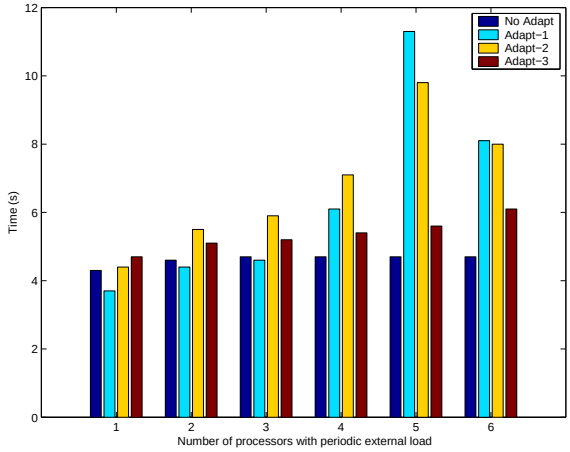


Fig. 4. Experiment 3 – Response times for Q1 for different levels of periodic imbalance: (a) duration and repeat duration = 1s; (b) duration and repeat duration = 5s.

Adapt-1, *Adapt-2* and *Adapt-3*. The increasing level of imbalance simulates the effect of having *0* to *6* other jobs competing for the use of one of the compute nodes in periodic load spikes of length *1* second. Each load spike is short compared with the runtime of the query, and thus might be consistent with, for example, interactive use of the resource in question. The following observations can be made: (i) As the net external load is half that stated (because it is only present half the time) the impact on the base case is less, and thus the challenge set to the load balancing strategies is greater, in that overheads are more likely to counterbalance benefits than in Experiment 2. (ii) Neither *Adapt-1* nor *Adapt-2* perform very well in this case. As the load on each node changes multiple times during the evaluation of the query, costly rebalancing (and thus movement of operator state) is judged to be required quite frequently. However, the new distribution policy will not be ideal if the external load is imposed or removed while rebalancing is taking place, and in either event, the policy is likely to be the appropriate one for at most a short period of time. As a result, these strategies rarely improve significantly on the base



(a)



(b)

Fig. 5. Experiment 4 – Response times for Q1 for different numbers of nodes with (a) constant and (b) periodic external load.

case in this experiment. The net load was never sufficient to cause *Adapt-4* to adapt in this experiment or in Experiment 4, so results for *Adapt-4* are omitted from the graphs.

Figure 4(b) illustrates the same experiment as Figure 4(a), except that *duration* and *repeat duration* have both been increased to *5*s, i.e., the external load changes a small number of times during the evaluation of each query. The observations made for Figure 4(b) essentially apply here too; *Adapt-1* and *Adapt-2* still do harm more often than good, although their overall performance is better than with the more rapid changes in external load.

Experiment 4: *Effectiveness of different adaptivity strategies in the context of variable numbers of machines with external load.* This experiment involves *Q1* being run with a parallelism level of *6*, with the external loads from Experiments 2 and 3 being introduced onto increasing numbers of nodes.

Figure 5(a) illustrates the position for a constant load of level *6* being introduced to increasing numbers of machines. The following observations can be made: (i) In the base case, the performance is consistent even though the number of loaded machines increases; this reflects the fact that partitioned parallelism can't improve on the speed of the slowest node, so all that changes in this experiment is the number of slowest nodes, until all nodes are equally slow. (ii) *Adapt-1* and *Adapt-2* perform well when the number of loaded machines is small, and rather better than in Figure 3. This is because in this experiment, the parallelism level is greater (at 6) than that used in Figure 3 (at 3), which means that the hash tables on each node are smaller, which in turn means that the cost of moving fractions of hash tables to counter imbalance is reduced. (iii) *Adapt-1* and *Adapt-2* perform less well as the number of loaded machines increases; this is because, although there continues to be imbalance, encouraging adaptation, the total available resource with which to absorb the imbalance is reduced. (iv) *Adapt-3* also performs well with smaller numbers of highly loaded machines, but rather less well as the number of loaded machines increased. This is because the chances of the "best two from three" nodes hosting a hash table bucket being lightly loaded reduces as the number of lightly loaded nodes reduces. (iv) By the time all six nodes are loaded, there is a similar pattern to that obtained for no imbalance in Figure 3, as in fact there is no imbalance, it's just that all the machines are equally slow.

Figure 5(b) explores a similar space, but with periodic rather than constant load on an increasing number of machines. The periodic load on each machine involves a *duration* and *repeat duration* of *1*s, where the load on the *m*'th machine is introduced after $m-1$ seconds, so that the load builds over time. The principal difference from the case of constant load is that *Adapt-1* and *Adapt-2* are more consistently ineffective as the ratio of loaded to unloaded machines increases, reflecting the fact that although the environment is unstable, it is increasingly difficult to offload resource in a way that pays off.

Experiment 5: *Effectiveness of different adaptivity strategies in a workload composed of multiple queries.*

In this experiment two variants of a workload consisting of all of *Q1* to *Q6* have been run. In the first case the queries have been run simultaneously with a parallelism level of *6* and no external load. The results are given in Table III. This is a challenging setting for workload adaptivity, as overall there are high loads, but there is little imbalance, as all nodes are running the same plans in parallel. The experiment shows that none of the techniques improve overall performance, although the worst deterioration is only around 30%. *Adapt-1* and *Adapt-2* were applied to change the balance *12* and *13* times, respectively, showing that the balance is quite consistent. No adaptation took place with *Adapt-4*.

In Flux [19], rebalancing only takes place when there is a machine with idle time available; as this is never the case in the above example, if the same rule was applied in the experiments, neither *Adapt-1* nor *Adapt-2* would have sought to rebalance load, and their performance would have improved. The experiments have not used this load threshold, on the grounds that in a heterogeneous environment, it may well be beneficial to take work from a slow machine with a level of

Nodes	No Adapt	Adapt-1	Adapt-2	Adapt-3	Adapt-4
6	71.4	85.4	103.8	97.3	91.7
12	71.4	66.3	76.3	93.3	90.1

TABLE III
RESULT TIMES FOR QUERY WORKLOADS.

contention of (say) *2*, to give it to a faster machine with level of contention (say) *2*, even though the recipient has no idle time.

In the second variant, different queries run on different (but overlapping) sets of nodes. The tables used by *Q1* to *Q6* are each allocated randomly to *6* of *12* available machines, and joins take place on the nodes where the data is located (in *Adapt-4*, each query is run on *6* randomly selected nodes). This is a slightly less challenging setting for workload adaptivity, as although overall loads are still high, there is likely to be more imbalance than in the previous scenario because different sets of machines are running different queries. As as result, both *Adapt-1* and *Adapt-2* perform significantly better than for *Workload 1* but neither perform substantially better than the base case. In addition, *Adapt-3* performs somewhat better for *Workload 2* than for *Workload 1*, but its overheads were still too great to improve on the base case. No adaptation took place with *Adapt-4*.

V. CONCLUSIONS

This paper has compared one novel and three existing techniques for balancing load during the evaluation of queries with partitioned parallelism. Overall, the lesson is that, although the techniques are sometimes effective, there are also circumstances in which they either make little difference or make matters worse.

This is not the first paper to identify circumstances in which AQP can reduce rather than improve query performance. For example, [3] describes circumstances in which adaptations responding to incorrect cardinality estimates in optimizers may thrash, with a succession of different adaptations being performed as different properties of an executing plan become clear. In that paper, extending the results from [14], a proposal is made in which bounds are derived within which plans are considered to be effective, and adaptations take place when these bounds are breached. Unfortunately such an approach is not directly applicable to parallelism in non-dedicated resources, as loads may change at any time and without warning.

In essence, AQP strategies differ along the following dimensions: (i) the *overheads* associated with a strategy whether or not it is required; (ii) the *cost* of carrying out an adaptation; (iii) the *stability* of the property to which the adaptation is responding – an adaptation can only be beneficial if the cost of carrying out the adaptation is outweighed by its lasting benefits.

The strategies are compared in terms of *overheads* and *adaptation cost* in Figure 6. In essence, *Adapt-1* and *Adapt-2* have low overheads (because plans essentially evaluate in the normal way in the absence of imbalance), but have high adaptation costs because moving parts of hash tables around in expensive. By contrast, *Adapt-3* has significant overheads because it maintains replicated hash table entries, which in turn allow individual adaptations to be made at minimal costs. The costs and overheads of *Adapt-4* are more difficult to pin down. The overheads take the form of additional work in query plan fragments, which vary from query to query, and the costs are in terms of resources to run redundant fragments, which also increase the total load on the system.

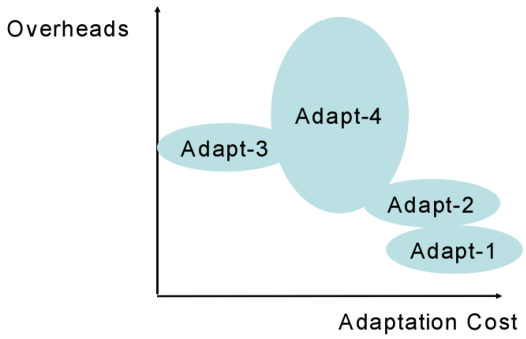


Fig. 6. Trade offs involved in the techniques.

Unfortunately, load imbalance is not intrinsically stable. Parties over which a query processor has no control can substantially change the load on a resource without warning. Thus techniques with high adaptation costs, such as those that move portions of a hash table around when seeking to restore balance (*Adapt-1* and *Adapt-2*), are likely to perform poorly where the load changes rapidly. *Adapt-3*, was designed to overcome this weakness. However, although *Adapt-3* provides dependable performance in unstable environments, it has significant overheads, and thus is uncompetitive where there is little imbalance.

Adapt-4 has significant overheads in the absence of suitably clustered source data, and adapts only after imbalance has caused complete phases of plan execution to be delayed (e.g., the construction of a hash table). This means that *Adapt-4* is only likely to improve on *Adapt-1* and *Adapt-2* where there is high external load or in contexts where *Adapt-1* and *Adapt-2* struggle – for example, when load changes rapidly. However, an advantage of the fact that *Adapt-4* responds only at certain landmarks in query evaluation is that it responds to the consequences of sustained imbalance rather than the potentially transient presence of an imbalance.

The following activities could usefully be explored to improve the performance of the evaluated techniques:

- *Adapt-1 and Adapt-2:* make use of cardinality information to avoid adapting near the end of plan phases; monitor the frequency and nature of load fluctuations over time, to determine the likelihood that an adaptation will prove worthwhile in a given setting.

- *Adapt-3:* explore the use of higher levels of parallelism for joins in this strategy, where the redundant work takes the form of additional hash table builds and probes – it may be the case that the performance of *Adapt-3* can be significantly improved if it is run with a higher parallelism level to offset the additional hash table probes and builds that are required; monitor the frequency and nature of load fluctuations over time, to determine the likelihood that this strategy should be enabled – *Adapt-3* performs satisfactorily in the sorts of settings in which *Adapt-1* and *Adapt-2* are likely to struggle.
- *Adapt-4:* make more use of information on the progress of plan fragments to avoid launching redundant evaluation when this is unlikely to be required, to bring forward the execution of redundant plan fragments when nodes are progressing very slowly, and to assess more effectively the progress of redundant evaluations.

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. 1st ICSOC*, pages 467–482. Springer, 2003.

[2] R. Avnur and J.M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *ACM SIGMOD*, pages 261–272, 2000.

[3] S. Babu, P. Bizarro, and D. DeWitt. Proactive Re-Optimization. In *Proc. ACM SIGMOD*, pages 107–118, 2005.

[4] R. Braumandl, M. Keidl, A. Kemper, D. Kossmann, A. Kreutz, and S. Seltzsam ans K. Stocker. ObjectGlobe: Ubiquitous Query Processing on the Internet. *VLDB Journal*, 10(1):48–71, 2001.

[5] S. Chaudhuri, V.R. Narasayya, and R. Ramamurthy. Estimating Progress of Long Running SQL Queries. In *Proc. SIGMOD*, pages 803–814, 2004.

[6] A. Gounaris, N.W. Paton, A.A.A. Fernandes, and R. Sakellariou. Self-monitoring query execution for adaptive query processing. *Data Knowl. Eng.*, 51(3):325–348, 2004.

[7] A. Gounaris, J. Smith, N. W. Paton, R. Sakellariou, and A. A. A. Fernandes. Adapting to Changing Resources in Grid Query Processing. In *Proc. 1st International Workshop on Data Management in Grids*, pages 30–44. Springer-Verlag, 2005.

[8] G. Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. In *Proc. SIGMOD*, pages 102–111, 1990.

[9] G. Graefe. Iterators, Schedulers, and Distributed Memory Parallelism. *Software Practice and Experience*, 26(4):427–452, 1996.

[10] R. Huebsch, J.M. Hellerstein, N. Lanham, B. Thau Loo, S. Shenker, and I. Stoica. Querying the internet with pier. In *VLDB*, pages 321–332, 2003.

[11] Z.G. Ives, A.Y. Halevy, and D.S. Weld. Adapting to Source Properties in Data Integration Queries. In *Proc. SIGMOD*, pages 395–406, 2004.

[12] D.T. Liu and M.J. Franklin. GridDB: A Data-Centric Overlay for Scientific Grids. In *Proc. VLDB*, pages 600–611. Morgan-Kaufmann, 2004.

[13] G. Luo, J.F. Naughton, C. Ellmann, and M. Watzke. Toward a progress indicator for database queries. In *Proc. ACM SIGMOD*, pages 791–802, 2004.

[14] V. Markl, V. Raman, D.E. Simmen, G.M. Lohman, and H. Pirahesh. Robust query processing through progressive optimization. In *Proc. ACM SIGMOD*, pages 659–670, 2004.

[15] S. Narayanan, T.M. Kurc, and J. Saltz. Database Support for Data-Driven Scientific Applications in the Grid. *Parallel Processing Letters*, 13(2):245–271, 2003.

[16] V. Raman, W. Han, and I Narang. Parallel querying with non-dedicated computers. In *Proc. VLDB*, pages 61–72, 2005.

[17] S. Sampaio, N.W. Paton, J. Smith, and P. Watson. Measuring and Modelling the Performance of a Parallel ODMG Compliant Object Database Server. *Concurrency: Practice and Experience*, 18(1):63–109, 2006.

[18] M.A. Shah, J.M. Hellerstein, and E.A. Brewer. Highly available fault-tolerant, parallel dataflows. In *Proc. SIGMOD*, pages 827–838, 2004.

[19] M.A. Shah, J.M. Hellerstein, S.Chandrasekaran, and M.J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *Proc. ICDE*, pages 353–364. IEEE Press, 2003.

[20] J. Smith and P. Watson. Fault-Tolerance in Distributed Query Processing. In *Proc. IDEAS*, pages 329–338. IEEE Press, 2005.

[21] G. Swart. Spreading the load using consistent hashing: A preliminary report. In *3rd Int. Symp. on Parallel and Distributed Computing*, pages 169–176. IEEE Press, 2004.

[22] T. Urhan and M.J. Franklin. XJoin: A Reactively-Scheduled Pipelined Join Operator. *Data Eng. Bulletin*, 23(2):27–33, 2000.

[23] Y. Zhou, B.C. Ooi, K-L Tan, and W.H. Tok. An Adaptable Distributed Query Processing Architecture. *Data & Knowledge Engineering*, 53(3):283–309, 2005.