The VLDB Journal manuscript No.
(will be inserted by the editor)

Norman W. Paton · Jorge Buenabad-Chavez · Mengsong Chen · Vijayshankar Raman · Garret Swart · Inderpal Narang · Daniel M Yellin · Alvaro A.A. Fernandes

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

Received: date / Revised version: date

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 prevail in the computational environment. In adaptive query processing (AQP), several techniques have been proposed for dynamically redistributing processor load 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 a simulation-based evaluation of these autonomic parallelization techniques in a uniform environment and compares how well they improve the performance of the computation. Four published strategies are compared with a new algorithm that seeks to overcome some weaknesses identified in the existing approaches. In addition, we explore the use of techniques from online algorithms to provide a firm foundation for determining when to adapt in two of the existing algorithms. The evaluations identify situations in which each strategy may be used effectively and in which it should be avoided.

Norman W. Paton
School of Computer Science,
University of Manchester
Oxford Rd, Manchester M13 9PL, UK
E-mail: npaton@manchester.ac.uk

Jorge Buenabad-Chavez
Department of Computer Science
CINVESTAV-IPN
Av. IPN No. 2508 Col. San Pedro Zacatenco
Mexico, D.F. 07360. MEXICO
E-mail: jbuenabad@cs.cinvestav.mx

Mengsong Chen
School of Computer Science,
University of Manchester
Oxford Rd, Manchester M13 9PL, UK
E-mail: chenm@cs.man.ac.uk

Vijayshankar Raman
IBM Almaden Research Center,
650 Harry Road, San Jose CA 95120, USA
E-mail: ravijay@us.ibm.com

Garret Swart
IBM Almaden Research Center,
650 Harry Road, San Jose CA 95120, USA
E-mail: gswart@us.ibm.com

Inderpal Narang
IBM Almaden Research Center,
650 Harry Road, San Jose CA 95120, USA
E-mail: narang@us.ibm.com

Daniel M. Yellin
IBM T.J. Watson Research Labs,
P.O. Box 704, Yorktown Heights, NY 10598, USA
E-mail: dmy@us.ibm.com

Alvaro A.A. Fernandes
School of Computer Science,
University of Manchester
Oxford Rd, Manchester M13 9PL, UK
E-mail: alvaro@cs.man.ac.uk

1 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, for example, 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. In such problems, the macro problem can be split into micro problems of arbitrary computational size and small description size, each micro problem can be solved independently, and the answers to the micro problems

can be combined trivially into a solution for the macro problem. These problems can be solved optimally by balancing the granularity overhead, the computing time lost in making the micro problem smaller, and the resources wasted when a processor is snatched in the middle of solving a micro problem. Effort has more recently been focusing on solving problems that don't fit the embarrassing model, including problems in query processing. For example, parallel query processing [8] 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.

By contrast, distributed query processing (DQP) [19] most commonly accesses existing data sources *in situ*, but makes limited use of parallel query processing techniques. Indeed, most DQP systems follow the wrapper-mediator model, with query processing divided between the mediator and the wrappers of the accessed resources (e.g. [6,18]). More recently, however, internet (e.g. [4, 16]) and grid (e.g. [33,1,21,25]) query processing has sought to evaluate fragments of a query on computational resources identified at query runtime, benefiting from pipelined and/or partitioned parallelism to improve performance. However, such query processors typically have limited information about the nodes being used and little opportunity to influence what other work they carry out. Several different proposals have been put forward that seek to take 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. [10,17,36,39]).

The potential of partitioned parallelism for providing scaleable query processing over non-dedicated resources, combined with the fact that load imbalance causes a task that makes use of partitioned parallelism to perform at the speed of the slowest participant, has led to several proposals for adaptivity specifically aimed at maintaining load balance (e.g. [13,32,28]). These approaches, which are designed for use in open and unpredictable environments, differ from most work on dynamic load balancing for parallel databases (e.g. [27,7]) in changing load balance within, rather than between, queries. 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 the above approaches to balancing load for parallel stateful operators, and considers two additional approaches. The study explores how the proposals perform in the same setting, thus enabling direct comparison over a wider range of conditions than were studied in the original papers.

The contributions of this paper are: (i) the proposal of a novel approach to adaptive load balancing, based on incremental replication of operator state; (ii) an evaluation of existing approaches and the new approach to load balancing for queries evaluated using partitioned parallelism; (iii) an exploration of the use of techniques from online algorithms [38] to determine when to adapt; and (iv) a characterization of the trade-offs that affect the design of AQP systems. The paper is structured as follows. Section 2 provides the technical context for the material that follows, by describing the adaptivity approaches evaluated. The evaluations are based on simulations of the different approaches; Section 3 describes the simulator and the experimental setup. Section 4 describes the experiments conducted on the new and existing techniques and discusses the results. Section 5 describes and evaluates the use of techniques from online algorithms to inform decision making in two of the techniques. Section 6 reviews the lessons learned. This paper extends the results in [26] by including a new adaptive algorithm, presenting the results of several additional experiments, and exploring the application of techniques from online algorithms in adaptive load balancing.

2 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 delays the completion of the join as a whole. As such, load balancing aims to make the values for $evaluation\_time(F_i)$ 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, because maintaining a balanced load involves ensuring that (i) the portion of the hash table on each node reflects the required work distribution, and (ii) changes in the work distribution to maintain load balance are preceded by corresponding

changes to the hash table on each node. In what follows, we refer to the period during which a hash table is being built by a join operator on the join attribute(s) of its left operand as the *build phase*, and the subsequent period when it is being looked up for each of the tuples in its right operand as the *probe phase*. Assume that tuples from A have been used to build the hash table and that the probe phase involving B 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 reallocating part of the hash table from N_i to other nodes, and ensuring that future tuples from B are sent to the appropriate node. As the hash table for A on node N_i may be large, these movements of state may involve a significant overhead; the trade-offs associated with dynamically balancing load for stateful operators are studied in Section 4. 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 [13].

A popular approach to implementing parallel query evaluation uses one of the flavors of the *exchange* operator [14] to distribute tuples between parallel plan fragments. Each *exchange* operator has *producer* and *consumer* components, such that each producer is configured to send data to one or more consumers. In essence, each exchange operator reads tuples from its children, 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 plan 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 arrow between a pair of partitions represents communication between an exchange producer and an exchange consumer. The exchange producers on the scan nodes use the same hash function and share a distribution policy, which ensures that tuples with matching join attributes are sent to the same machine for further processing. The distribution policy is represented in the figure by the proportion of data sent to each parent node (*0.8* and *0.2*, respectively).

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 about the environment; (ii) *assessment* – the analysis performed on the monitoring information to identify a problem that suggests that adaptation may be beneficial; and (iii) *response* – the reaction that is taken with a view to addressing the problem that has been detected.

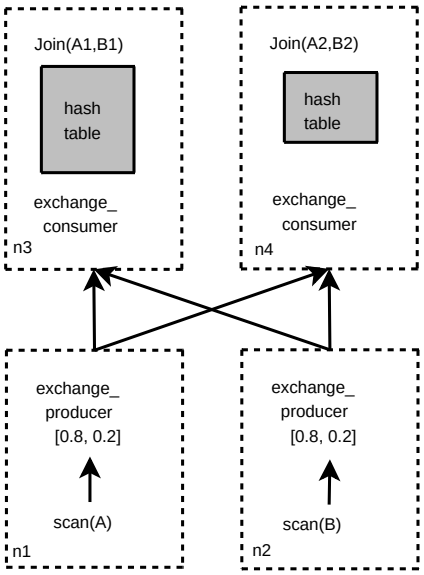


Fig. 1 Example parallel execution plan.

Five adaptivity strategies are compared, and are characterized in Table 1 and 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.

Adapt-1: Sibling-based data redistribution: when a load imbalance is detected, this approach relocates portions of the hash table from more highly loaded to less highly loaded nodes. An example of before and after states for an adaptation using *Adapt-1* is given in Figure 2. The example shows query evaluation taking place on four nodes, *n1* to *n4*, for the query $A \bowtie B$. The distribution policy of the *exchange_producer* indicates the relative sizes (in terms of numbers of tuples) of the subsets of A and B that should be sent to each of the parent nodes. In the example, in the initial state on the left, 80% (denoted by 0.8) of the data should be sent to node *n3* and 20% (denoted by 0.2) of the data should be sent to *n4* for joining. If this allocation is found to be distributing work between the nodes in a way that leads to load imbalance, then the distribution policy is updated, and hash table state is transferred between the sibling join fragments. In the figure, the new distribution policy sends 40% of the data to *n3* and *60%* of the data to *n4*, as a result of which part of the existing hash table containing the operator's state has had to be moved from *n3* to *n4*.

In the original paper [32], the reallocation is carried out using an operator known as Flux. This behavior can be broken down into monitoring, assessment and response components as follows:

Strategy	Source	Summary of Adaptation
Adapt-1	[32]	Redistribute operator state between sibling join nodes.
Adapt-2	[13]	Redistribute operator state from caches in exchanges.
Adapt-3	this paper	Replicate operator state on demand.
Adapt-4	[26]	Maintain redundant operator state based on runtime load.
Adapt-5	[28]	Run redundant copies of late fragments.

Table 1 Summary of key properties of algorithms.

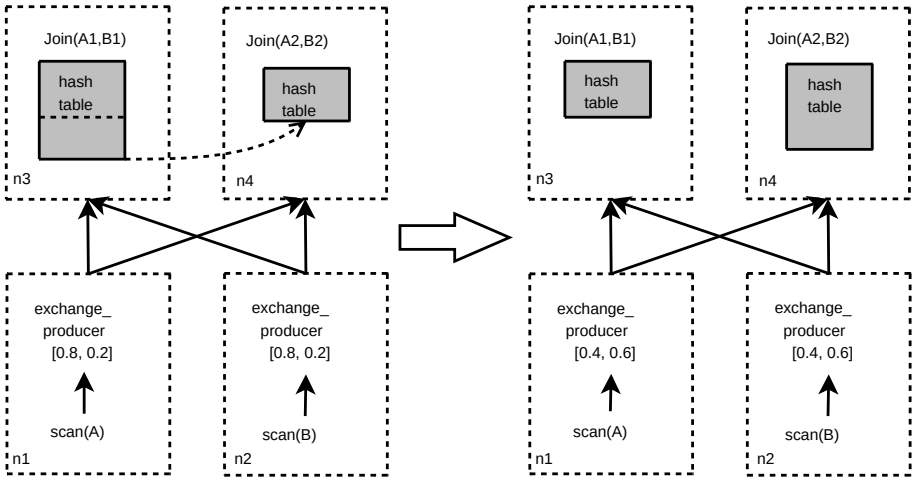


Fig. 2 An example of an adaptation in Adapt-1, where changes in the distribution policies of exchange producers lead to reallocation of hash table data from *n3* to *n4*.

- *Monitoring:* query fragments 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 fragment. 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 fragment on node n:

  $proposed\_distribution(n) =$
  if $contention(n) < 1$ **then** 1
  else $1/contention(n)$

  where $contention(n)$ is the level of contention on node n. In essence, given the level of contention for a node, the definition assumes that if the contention is less than 1, then this level of contention comes from the query being evaluated, and that the complete resource on that node is available for query processing; in contrast, if the contention is greater than 1, then the assumption is that the query will be able to access the resource with equal rights to the other sources of contention [1]. The *proposed_distribution* values are normalized to sum to 1, and where the proposed distribution differs from the current distribution by more 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 table fragments must be larger than the parallelism level, but each 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 table fragment. The redistribution steps in this and subsequent strategies (*Adapt-2* and *Adapt-3*) are modeled as taking place in parallel. 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 policies* of the upstream exchange operators so that subsequent tuple routing reflects the revised data distribution.

[1] Different heuristics can be used for deriving a *proposed_distribution*. For example, OGSA-DQP [13] uses $rate(p)/contention(n)$, where $rate(p)$ is the current rate at which the partition p is processing tuples, whereas Flux [32] uses the contention directly. The approach described in the text performed better in the experiments than these approaches.

Experiments on the sensitivity of the strategies to the number of fragments in a table and the threshold that defines the minimum change in distribution policy are described in Appendix A.

Adapt-2: Cache-based data redistribution: when a load imbalance is detected, in this approach the portion of the hash table allocated to each node is modified by sending table fragments from caches on upstream (i.e., ancestor) exchanges [13]. The caches are maintained by a fault-tolerance technique that allows a query to recover from node failures, as described in [34]. In this fault tolerance scheme, each time an exchange sends data from a producer machine to a consumer machine, a cache at the producer stores this data until it has been fully processed by the consumer, at which point it is removed from the cache. If the consumer machine fails, then all unprocessed data associated with the machine can be resent to a dynamically identified replacement. The adaptive technique that makes use of these caches can be captured by monitoring, assessment and response components as follows:

- *Monitoring/Assessment:* as in *Adapt-1*.
- *Response:* as *Adapt-1*, except that table 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 [32] 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.

Adapt-3: Replicate on demand: this approach is a modification to *Adapt-1*; as in *Adapt-1*, the detection of load imbalance may lead to the locations of hash table fragments being updated to support the use of a revised distribution policy. However, unlike *Adapt-1*, table fragments are not *moved* from one node to another (deleting them from their original location), but are *replicated* as required to support the preferred distribution policy. As a result, during the evaluation of a query in an unstable environment, portions of the hash table are replicated, allowing an increasing proportion of the changes between distribution policies to go ahead without the need to relocate data. Because replicating table fragments at hash table build time could lead to a requirement to update numerous replicated fragments, replication takes place only at hash table probe time. Although hash table fragments are replicated, each probe takes place to only one of the replicas, selected on the basis of the load balance, so no duplicates answers are produced. This algorithm can be characterized in terms of its monitoring, assessment and response behavior as follows:

- *Monitoring:* as in *Adapt-1*, except that assessment only takes place if the hash join is in the probe phase.
- *Assessment:* as in *Adapt-1*.
- *Response:* as in *Adapt-1*, each table is considered to consist of a number of fragments, which are the unit of redistribution. 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* computed during *Assessment*. Fragments are then transferred from each partition in the *producer* list to the corresponding partition in the *consumer* list until the change required to move closer to the *proposed_distribution* is smaller than a fragment. Where a replica has previously been created that avoids the need to transfer data from the *producer* to the *consumer*, this replica is used and no hash table data needs to be copied. Where no such replica exists, table fragments are copied from the *producer* to the *consumer* nodes as required to match the *proposed_distribution*, and each such copy is recorded in a replica table.

Adapt-4: 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 [35]. 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 (i.e. the two with the lowest level of contention).

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 exactly 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 as the hash table is being built. Each stored tuple has an extra field that distinguishes 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; this result production rule avoids the production of duplicate answers.

The dynamic behavior of the algorithm ensures that work is allocated to the most lightly loaded nodes, as tuples are being processed, 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 updated by sorting the nodes according to their current 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* to *Adapt-3*, 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* to *Adapt-3* where no load imbalance occurs, and has significant space overheads.

In relation to the wider AQP literature, *Adapt-4* essentially adapts continuously through query evaluation, a property shared with several other adaptive operators, such as Eddies [2] and XJoin [37]. By contrast, *Adapt-1* to *Adapt-3* and *Adapt-5* 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 [24] and Tukwila [17].

Adapt-5: Redundant fragment evaluation: when a plan fragment F_i is slow to start producing tuples, this approach runs a redundant copy of F_i, F_i', on a lightly loaded node; whichever of F_i or F_i' is first to start to produce data is then used as the preferred source of results [28].

- *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 identified as candidates for redundant evaluation by the response component. That is, the assessment component simply identifies all siblings of a completed fragment as candidates for redundant execution. This is a more aggressive policy for scheduling redundant evaluations than the one in the original paper [28], where candidates for redundant execution were only identified when they had taken twice as long as the fastest partition. However, in most cases, this latter policy reacts too slowly to compete with the other adaptivity strategies.
- *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 plan fragment 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.

If this strategy was used directly on query execution plans that used exchange to redistribute data before every join, it would be unlikely to be effective. This is because exchange operators essentially synchronize the evaluation of their parent nodes. For example, a hash join cannot enter its hash table probe phase until the hash table build phase has completed. However, it cannot complete the build phase until the upstream exchange on the build input has completed, and thus all sibling parent join nodes tend to move from the build to probe phases at much the same time. The same sort of argument can be broadened to other operators and landmarks in the evaluation of a query.

To get around this problem, which results from exchanges redistributing tuples before and after each join in a multi-join query, *Adapt-5* executes plans with less tightly coupled partitions. The resulting advantage is that individual query partitions complete at more widely differing times, and 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 [28]. In essence, this additional work results from more tuples that do not match being compared on each node, as the benefit of the hash-based redistribution used in exchange nodes has been lost.

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 [28].

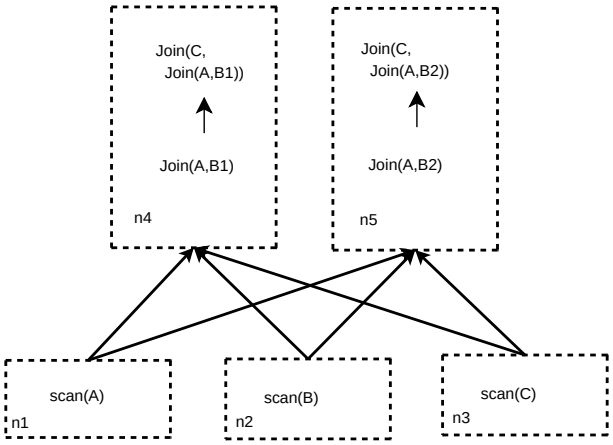


Fig. 3 Example query plan in Adapt-5.

As such, *Adapt-5*, like *Adapt-2* and *Adapt-3*, accepts some overhead to facilitate the chosen adaptivity strategy. An example query plan for two joins, where the joins are parallelized over two nodes, is illustrated in Figure 3.

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* [31] and *Adapt-2* [34].

3 Simulation Model

The paper compares the adaptivity strategies from Section 2 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-5* (as reported in [13,28], 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.

3.1 Modeling Query Evaluation

The simulation depends on a model of the cost of performing primitive tasks. We use a cost model as no individual directly simulated property, such as the amount of network traffic, can be used to compare the effects of the different techniques on response time. For example, evaluations involving *Adapt-4* send the same amount of data when evaluating a query, no matter how much imbalance there is, although, of course, response times are affected by such imbalances. 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 2; 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 [30].

The simulator emulates the behavior of an iterator-based query evaluator [15]. In the iterator model, 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. The iterator model is pull-based, in that data is fetched from a child node to its parent on demand. This means that the scheduling and interaction of nodes within the query evaluator is supported directly by the operation call graph, with any additional complexities encapsulated within the implementations of the operators.

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 2. 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, until such time as either the exchange producer or consumer cache is full. As such, *exchange*, which is typically implemented using multiple threads, can be seen as pushing data across machine boundaries, within an over-

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 cache	10000	tuples
Size of the exchange consumer cache	20000	tuples
Size of the disk cache	50	Mb

Table 2 Cost model parameters.

all evaluation strategy in which results are pulled from the root. The simulator models the filling and emptying of caches on exchange.

Figure 4 illustrates some of the properties tracked by the simulator during the evaluation of an example join query (Q1 from Table 3) in which node 1 stores the data, and node 2 both runs the join and conveys the results to the calling program. The time axis is in tenths of a second. In the parallel algebra, the query is implemented as:

$exchange_3$
$(join(exchange_1(scan(P)), exchange_2(scan(PS))))$.

The query consists of two scans both running on node 1, each feeding into exchange operators which in turn are the operands of the hash join running on node 2, the results of which are passed onto a root exchange. The first graph shows the total number of tuples produced by each operator over time. The first operators to produce data (*scan(P)* and its parent $exchange_1$ which both start producing data at time 0) are slightly out of step in their rate of production because the exchange caches the data it has read from the scan, which is only recorded as output from the exchange when it is extracted by the hash join for insertion into the hash table. This is reflected in the level of contention for the disk accessed by *scan(P)*, which is read at full speed into the exchange cache (which can store the entire table); there is then a gap in disk activity while the remainder of the hash table is populated from the cache. A similar process is followed from time point 27 for the probe phase of the hash join, with the other scan operator (*scan(PS)*) feeding an exchange ($exchange_2$), which in turn fills its cache, after which scanning proceeds at the rate the join consumes the tuples. The output counts for $exchange_2$, $exchange_3$ and the join are superimposed in the graph, as they all produce data at the same rate. This is less than the throughput of the disk, which explains the reduction in the level of contention for the disk after time point 40. The level of contention for the compute resource on which the join runs (*node-2*) is greater than 1 during the probe phase because threads for the two active exchanges ($exchange_2$) and ($exchange_3$) compete with the join for access to the CPU.

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

Table 3 Queries used in experiments.

3.2 Experiment Setup

The configurations used are as follows. A single network (modeled as an Ethernet) and type of (single core) computer are used throughout, with physical properties as described in Table 2. There is assumed to be sufficient main memory in all computers to support the caches described in Table 2 and to hold join hash tables; moving to multi-pass hash joins would require some changes to all of *Adapt-1* to *Adapt-4*. All queries are assumed to run over cold configurations (the caches are assumed to start empty). The joins are generally bottlenecks in the queries, and thus the queries are CPU-bound.

The queries used in the experiments are described in Table 3, which makes use of the relations from Table 4; queries make use of 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, 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 tuples in the base tables.

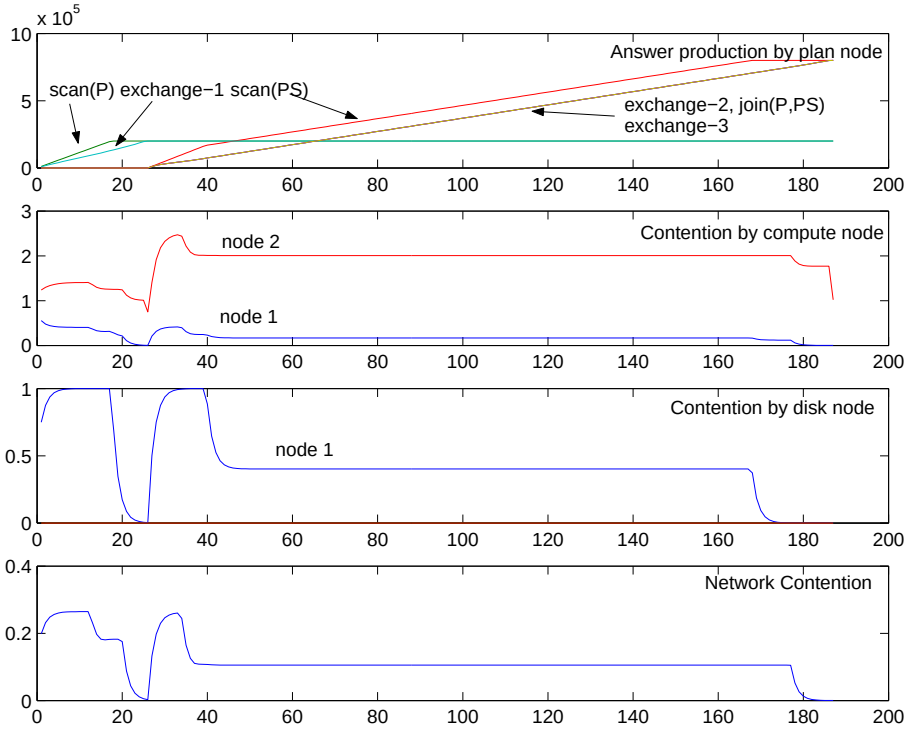


Fig. 4 Inside plan evaluation. In all cases, the horizontal axis is time from the start of evaluation in tenths of a second. In the top graph, the vertical axis is the number of tuples produced. In all other graphs, the vertical axis is the level of contention for the resource.

Name	Short Name	Cardinality (rows)	Tuple Size (bytes)
Supplier	S	10,000	159
Part	P	200,000	155
PartSupp	PS	800,000	144
Orders	O	1,500,000	104
LineItem	L	6,001,215	112
Nation	N	25	128

Table 4 Tables from TPC-H.

The following features characterize the environments in which the experiments are conducted. All of *Adapt-1* to *Adapt-4* 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 [28], when evaluating *Adapt-5* 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 with 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. [28] exploits a SAN to avoid depending on potentially unreliable autonomous nodes for data storage.

Where there is more than one join, left deep trees are used as this reduces the challenge of coordinating adaptations across many joins running at the same time; 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 2. 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.

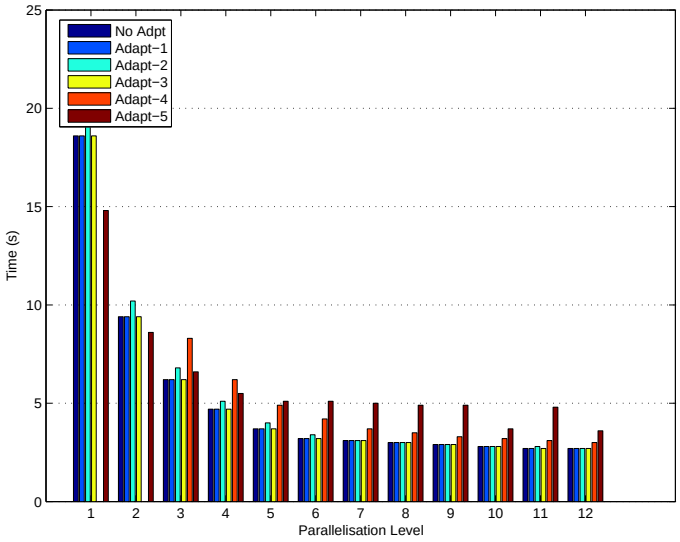


Fig. 5 Experiment 1 – Response times for Q1 for different levels of parallelism.

3. *Poisson:* The arrival rate of jobs follows a Poisson distribution [20]; the Poisson distribution expresses the probability of a number of events occurring in a fixed period of time where these events occur with a known average rate, and are independent of the time since the last event. As such, the Poisson distribution can be used to model things like the number of times a server is accessed in any one hour, when the average number of requests per day is known. In the experiments, the *average number of jobs starting per second* is varied in a controlled manner, and in each clock tick a random number of jobs from the Poisson distribution with the given average are simulated as having started. The *duration* of each job started is held constant within an experiment.

In this paper, all processes are assumed to be running with the same priority, and external loads (i.e. those that do not result from query evaluation) affect only CPU – 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 running. The simulation does not model monitoring overheads; this is because several authors have recently reported results on the cost of query monitoring (e.g. [5,12,22]), indicating that different approaches can provide suitable information with modest overheads.

4 Evaluating the Adaptive Strategies

This section explores the extent to which adaptive strategies from Section 2 are successful at improving query performance in the presence of load imbalance. The experiments seek to determine: (i) the extent to which the strategies *Adapt-1* to *Adapt-5* are effective at overcoming the consequences of load imbalance; (ii) the kinds of load imbalance that the different 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. It may be useful to refer to Table 1 when reading the experiments, as a reminder as to which strategy is which.

Experiment 1: *Overheads of different strategies.* This experiment involves running query *Q1* for all five 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 5 shows the response times for Q1 for parallelism levels 1 to 12 for each of the strategies from Section 2 compared with the case where adaptivity is disabled (*No Adapt*). No figures are given for *Adapt-4* 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*, *Adapt-2* and *Adapt-3* are 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-4* are up to around 30%, although their absolute value reduces with increasing parallelism; (iv) The additional work carried out by *Adapt-5* to ease the running of redundant plan fragments can be substantial, although there is no such overhead for parallelism level of 1. At a parallelism level of 1, *Adapt-5* performs well because it also avoids the overheads of *exchange*. At higher levels of parallelism, however, the amount of redundant work carried out is significant, with the overhead for *Adapt-5* often being around 50% compared with the *No adapt* case. The irregular results for the evaluation strategy associated with *Adapt-5* for larger numbers of processors in Figure 5 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-5* bears further comment, however. Although the results in Figure 5 are broadly consistent with figures presented in [28], 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 plan fragments for adaptivity. As such, the overhead associated with *Adapt-5* depends on

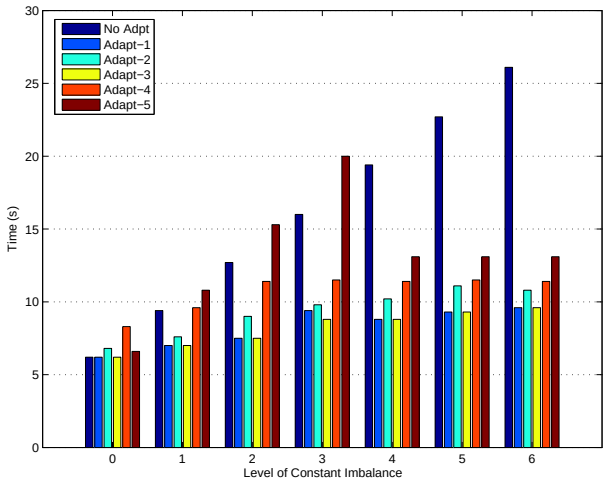


Fig. 6 Experiment 2 – Response times for Q1 running on three nodes for different levels of constant imbalance on one of the nodes.

the contribution that clustering makes to the overall performance of a query. These issues are discussed more fully in [28], and are not considered further here. In essence, as we do not model data as clustered in the experiments, the overheads associated with *Adapt-5* should be taken to be pessimistic. This affects the overall response times for queries using *Adapt-5*, but not the overall behavior of the associated adaptivity strategy, which is the principal concern of this paper.

Experiment 2: *Effectiveness of different 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.

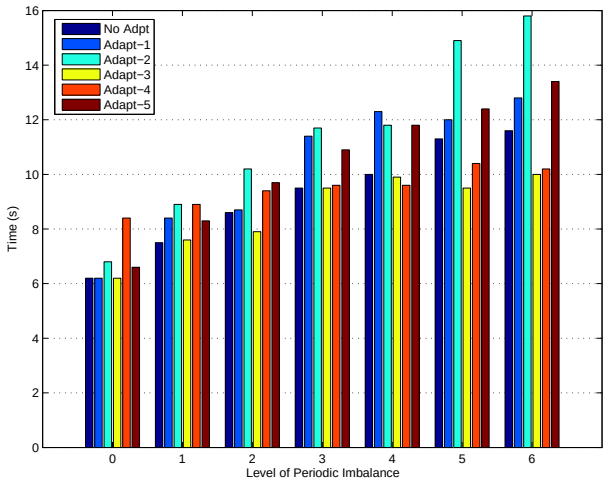
Figure 6 illustrates the results for strategies *Adapt-1* to *Adapt-5*. The increasing level of imbalance simulates the effect of having *0* to *6* other jobs competing for the use of one of the compute nodes.

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[2]. (ii) All of *Adapt-1* to *Adapt-3* consistently improve on the base case, in particular in the context of higher levels of imbalance. We observe that which is the most/least effective varies during the experiment. 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 adapt close to the end of the build phase, thereby selecting a load balance that may not be especially effective for the probe phase. However, because the external load is consistent throughout the experiment, the principal adaptations take place early in the evaluation of the query. At this point, the hash table has only been partially populated, so the cost of adapting early in the evaluation of a query is not high. (iii) *Adapt-4* has overheads in the base case of around 30%, but performs increasingly well as the level of imbalance grows. Given constant imbalance with level of parallelism 3, *Adapt-4* essentially runs the join on the two more lightly loaded nodes throughout. As a result, *Adapt-4* is effective when the reduced level of parallelism compensates for the delays resulting from imbalance in the base case. (iv) In *Adapt-5*, although the running of redundant partitions 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-5* in Figure 6 with the other strategies, we see that running of redundant plan fragments can be effective, providing comparable performance to *Adapt-1* to *Adapt-4* for higher levels of imbalance, although overall it can be seen to be the least effective of the strategies.

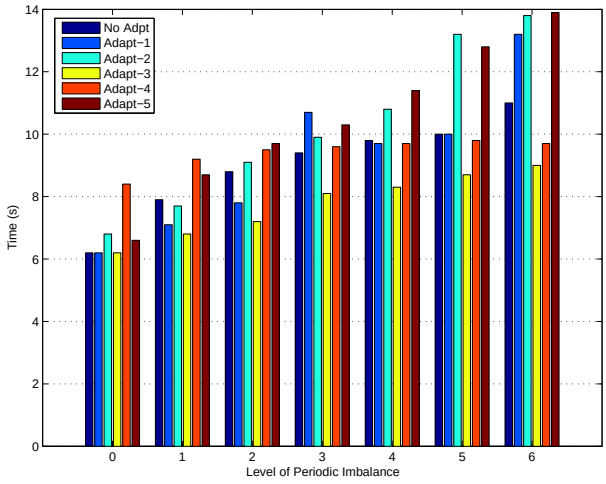
Experiment 3: *Effectiveness of different strategies in the context of periodic imbalance.* This experiment involves *Q1* being run with a parallelism level of *3*, where an external load is introduced that affects *1* of the *3* nodes exactly half the time (i.e. *duration* and *repeat duration* are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

Figure 7(a) illustrates the results for strategies *Adapt-1* to *Adapt-5*. 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)

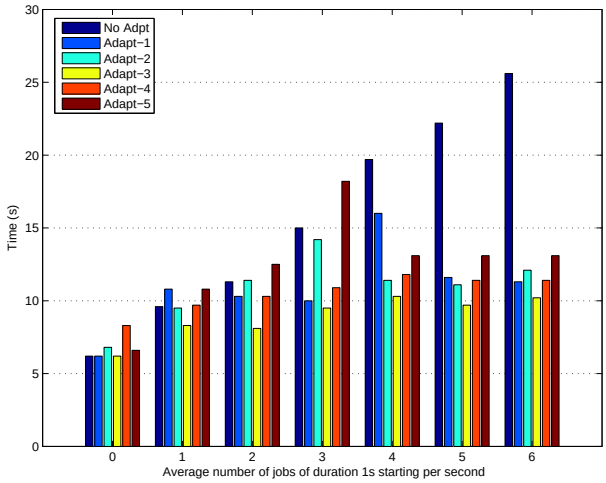
[2] We note that when an external load of level L is imposed in the base case, response times increase by less than a factor of L. This is because the thread in which the join is running, which is the bottleneck, is not the only thread used for query evaluation running on the loaded node (other threads support *scan* and *exchange* operators). As a result, as illustrated in Figure 4, the level of contention for compute resources in each node is greater than 1 in the base case, and thus the effect of adding (say) an external load of level 1 is more to increase the level of contention from 2 to 3 (i.e. by 50%) than from 1 to 2 (i.e. by 100%).



(a)



(b)

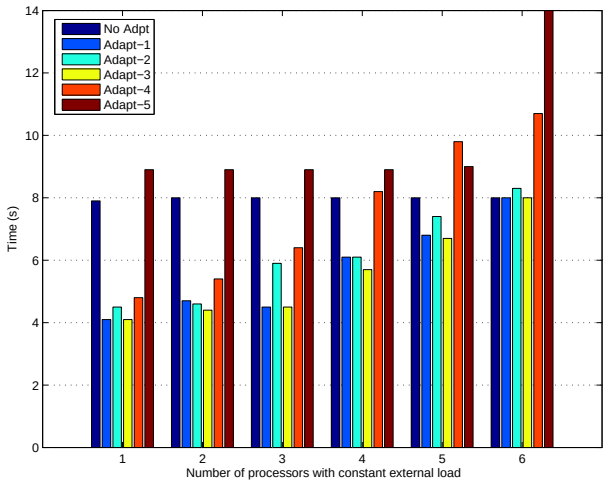


(c)

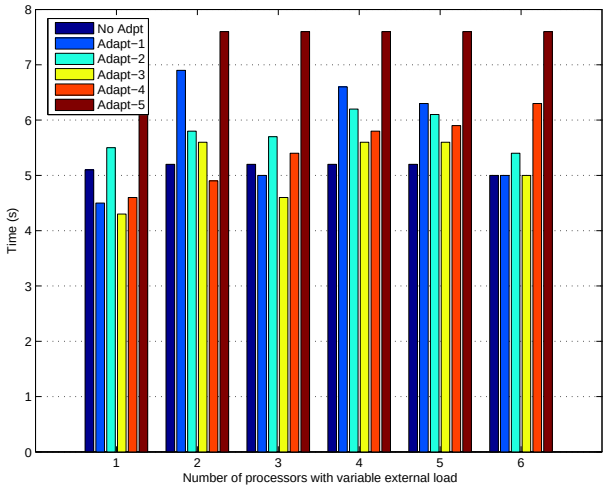
Fig. 7 Experiment 3 – Response times for Q1 running on three nodes for different levels of periodic imbalance on one of the nodes: (a) duration and repeat duration = 1s; (b) duration and repeat duration = 5s; (c) following a Poisson distribution where each job has a duration of 1s.

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 (in this experiment, other than in the base case, these strategies typically adapt *5* or *6* times during the evaluation of a query, with the costs of adaptations typically between *0.2* and *1.5* seconds). For the most part, adaptations take longer where there are higher levels of periodic load, because the levels of imbalance are greater, and thus higher proportions of the operator state need to be relocated. As a result, the portion of each load spike for which a just-completed adaptation is likely to be effective is often quite small. In addition, the heuristic that avoids frequent rebalancing may prevent rebalancing for longer than the duration of a load spike. As a result, these strategies rarely improve significantly on the base case in this experiment. 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) Periodically both *Adapt-1* and *Adapt-2* return significantly poorer results than the base case. This is because they adapt multiple times during a run, with few of the adaptations providing lasting benefit, and subsequent adaptations undoing the effects of previous ones. Furthermore, because the external load changes throughout the experiment, adaptations take place during the hash table probe phase. At this point, the hash table is fully populated, so the cost of adapting later in the evaluation of a query is higher than during the hash table build phase. *Adapt-3* is largely protected from such extreme behavior, as any decision to undo a previous adaptation only requires a change to the distribution policy, with no need to move operator state. (iv) *Adapt-4* copes well with the unstable environment, and provides good performance for higher levels of imbalance. In general, *Adapt-4* provides quite predictable behavior; it essentially runs the join on the two more lightly loaded nodes when there are high levels of imbalance. However, where there is no external imbalance, the random allocation of work to two buckets from three can create some variability between runs, which explains why *Adapt-4* is slightly slower for imbalance level *5* than *6*. (v) The net load imbalance was never sufficient to cause *Adapt-5* to replace a late-completing partition in this experiment.

Figure 7(b) illustrates the same experiment as Figure 7(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 7(a) essentially apply here too; *Adapt-1* and *Adapt-2* do harm as often



(a)



(b)

Fig. 8 Experiment 4 – Response times for Q1 running on six nodes for increasing numbers of nodes with (a) constant and (b) periodic external load.

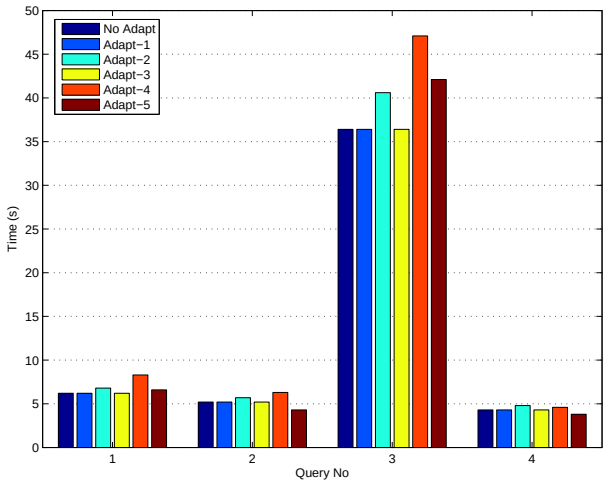
as good, and in several cases commit significant effort to ineffective adaptations. Overall, *Adapt-3* is the most effective strategy in this case.

Figure 7(c) illustrates response times where the *average number of jobs starting per second* is varied, and a random number of jobs belonging to the Poisson distribution is started every clock tick. In essence, this distribution leads to increasingly frequent job start (and end) events, but with few of the substantial changes in load that characterize Figures 7(a) and (b). Overall, the adaptive strategies perform similarly to their position for constant external load in Experiment 2, except that the somewhat less stable environment periodically leads to less than productive adaptations for *Adapt-1* and *Adapt-2*.

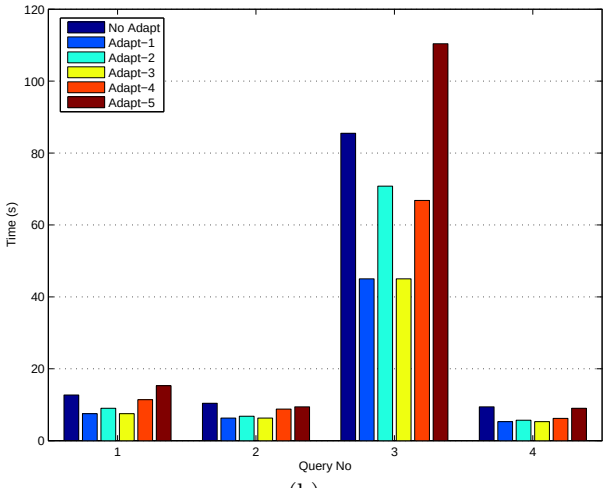
Experiment 4: *Effectiveness of different strategies in the context of variable numbers of machines with external load.* This experiment involves *Q1* being run with a parallelism level of *6*, with the external loads from Experiments 2 and 3 being introduced onto increasing numbers of nodes.

Figure 8(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* to *Adapt-3* perform well when the number of loaded machines is small, and rather better than in Experiment 2 (Figure 6). This is because in this experiment, the parallelism level is greater (at 6) than that used in Experiment 2 (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* to *Adapt-3* 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-4* also performs well with smaller numbers of highly loaded machines, but rather less well as the number of loaded machines increases. 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. (v) By the time all six nodes are loaded, there is a similar pattern to that obtained for no imbalance in Experiment 2 (Figure 6), as in fact, there is no imbalance, it's just that all the machines are equally slow! (v) *Adapt-5* adapts, running redundant requests that are subsequently chosen in preference to their counterparts running on loaded machines, where there are 1 to 5 loaded machines. However, the overheads associated with the strategy are such as to provide no overall benefit compared with the *No Adapt* case.

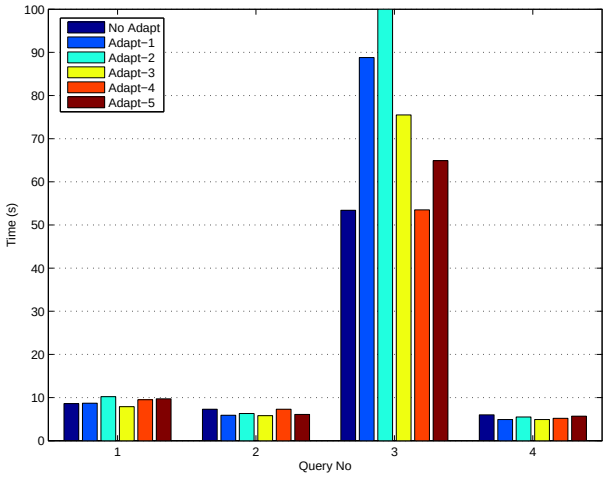
Figure 8(b) explores a similar space, but with periodic rather than constant load on an increasing number of machines. The periodic load on each machine involves a *duration* and *repeat duration* of *1*s, where the load on the mth machine is introduced after $m-1$ seconds, so that the load builds over time. The principal differences from the case of constant load are: (i) *Adapt-1* to *Adapt-3* 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 work to resources in a way that pays off. As in Experiment 3, *Adapt-1* to *Adapt-3* adapt multiple times, and the precise timings of such adaptations relative to load changes affects how much benefit is derived. Thus their response times do not follow a regular curve as the number of loaded machines increases. (ii) *Adapt-5* never successfully adapts, and thus remains slower than the base case throughout.



(a)



(b)



(c)

Fig. 9 Experiment 5 – Response times for Q1 to Q4 with a parallelism level of 3: (a) no external load; (b) constant external load of level 2 on a single machine; and (c) periodic external load of level 2 with duration and repeat duration of 1s.

Experiment 5: *Effectiveness of different strategies for different queries.* This experiment applyies the strategies to several different queries and loads, and assesses the extent to which properties of the query being run affect the benefits of each strategy. The experiment involves *Q1* to *Q4* from Table 3, and both constant and periodic loads selected from those used in Experiments 2 and 3. The results are presented in Figure 9.

It can be observed that while there are minor variations between the queries (for example, the overheads of Adapt-5 compared with the base case differ from query to query), the overall pattern with respect to overheads and the behavior of the strategies is broadly consistent with the findings from the previous experiments. We note that the external load, at level 2, is quite low, which explains the modest improvements in performance from the adaptive strategies relative to the base case in Figure 9(b).

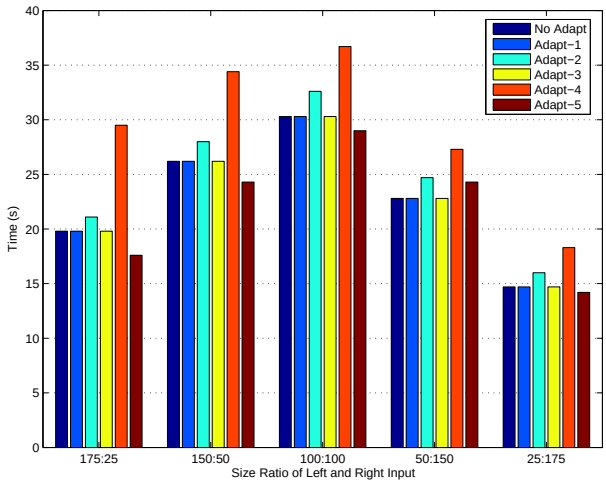
Experiment 6: *Effectiveness of different strategies for different input table size ratios.* This experiment involves applying the strategies to a single join query $A_{r-left} \bowtie B_{r-right}$, where $r-operand$ represents a ratio of the left and right input sizes, where each A-tuple is 155 bytes and each B-tuple is 155 bytes. The results are presented in Figure 10. The size ratios can be interpreted as follows. Tables A and B are taken to have base sizes of 10,000 tuples. The ratios represent the multipliers applied to the base sizes to give the actual table sizes used in the joins. Thus, with a ratio of 175 : 25, the left hand input A_{r-left} has size 175×10000 and the right hand input $B_{r-right}$ has size 25×10000. The join selectivity (the portion of the cartesian product that appears in the result) is constant at 5.0×10^{-6} throughout. These settings mean that the total number of input tuples read is the same in every case, but that the result is larger where the ratios are more similar. As such, the main focus in this experiment is not on absolute response times, but on relative response times.

The following can be observed: (i) Where there is no external load, in Figure 10(a), the total response times for the ratio 175 : 25 is greater than that for 25 : 175 (respectively for 150 : 50 and 50 : 150). This is because the ratios of input sizes lead to a corresponding ratio in the numbers of hash table inserts and probes, and the former are more expensive than the latter, following the cost model parameters in Table 1. (ii) Where there is no external load, the overhead for *Adapt-4* reduces as the relative size of the left hand input reduces. As for point (i), this relates to the relative numbers of hash table inserts and probes; *Adapt-4* carries out redundant hash table inserts and probes on each input tuple, and the overhead resulting from this redundancy is greatest where there are more hash table inserts than probes. (iii) Where there is periodic imbalance, in Figure 10(c), *Adapt-1* to *Adapt-3* generally perform less well where there are more hash table inserts than probes; this is because these strategies

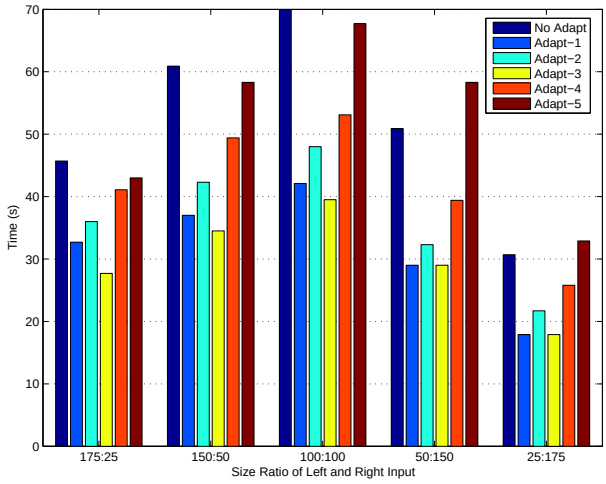
depend on relocating portions of the hash table during load balance, and large hash tables are costly to relocate.

Experiment 7: *Resource usage of different strategies.* Adaptive load balancing seeks to reduce response times by matching the work that needs to be done to the capabilities of the available resources. However, adapting to load imbalance necessarily involves work being done (such as relocating hash table state) in addition to that required for query evaluation, thereby affecting throughput for multiple tasks in loaded environments. This experiment compares the amount of resource used by the adaptive techniques when evaluating *Q1* using constant, periodic and Poisson loads from Experiments 2 and 3. In all cases, the additional amount of work done is reported when adapting to high levels of imbalance, so that significant amounts of adaptation may be required. The results are presented in Figure 11, which shows the amount of work carried out by the different strategies relative to the base case, so *1.0* represents no additional work.

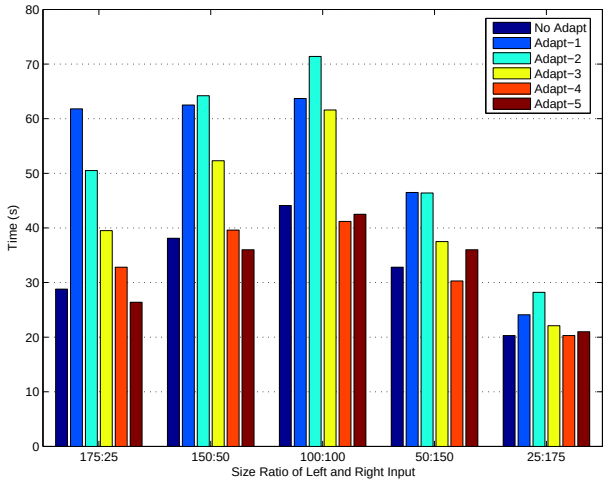
The following can be observed: (i) None of *Adapt-1* to *Adapt-4* create any additional disk traffic. This is to be expected, as the join algorithms in the experiments are main-memory hash joins. (ii) *Adapt-1* to *Adapt-3* handle both constant and Poisson imbalance effectively, by carrying out changes to distribution policies and hash tables early during evaluation when there is little hash table state to relocate. As a result, in Figure 11(a) and (c) the increase in compute and network activity is by small amounts compared to the base case. (iii) *Adapt-1* to *Adapt-3* adapt several times in response to periodic imbalance in Figure 11(b), which leads to additional work being done in a setting that is challenging for these strategies. *Adapt-3*, as well as doing less additional work than *Adapt-1* and *Adapt-2*, has the more beneficial effect on response times. (iv) *Adapt-4* carries out significant amounts of extra work, the levels of which are essentially independent of the level or nature of the imbalance. (v) The additional work carried out by *Adapt-5* is of two forms – that resulting from the extra work required to provide loosely coupled fragments, and that resulting from the running of redundant fragments where there is imbalance. The former is greater than the latter in this experiment. In all of Figure 11(a) to (c), redundant fragments are run, and in both (a) and (c) the redundant fragment runs to completion and is used to compute the result.



(a)



(b)

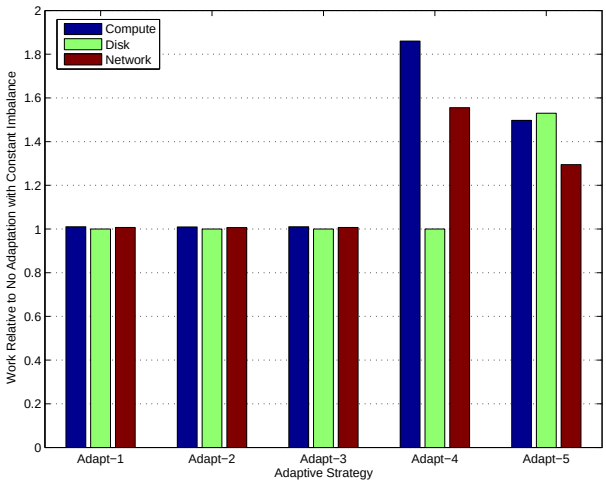


(c)

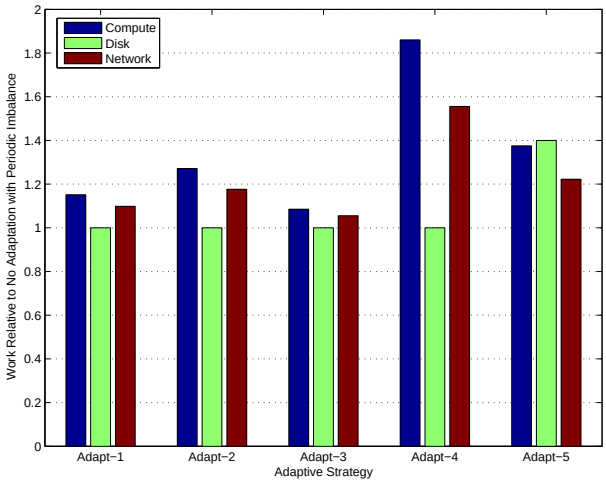
Fig. 10 Experiment 6 – Response times for Q1 with different input size ratios with a parallelism level of 3: (a) no external load; (b) constant external load of level 2; (c) periodic external load of level 2 with duration and repeat duration of 1s.

5 Exploiting Techniques from Online Algorithms

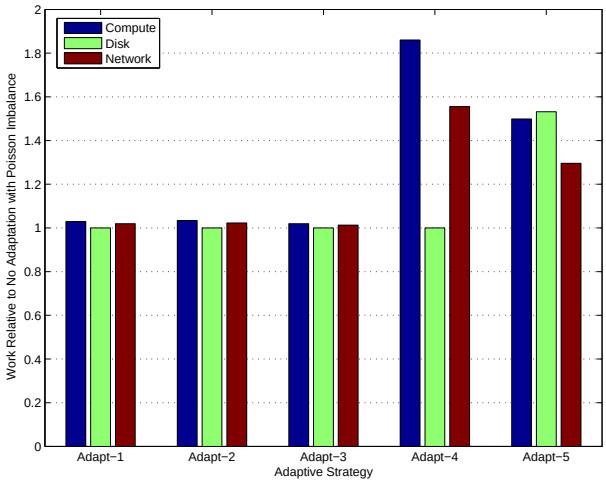
Several of the algorithms evaluated in Section 4, in particular *Adapt-1* and *Adapt-2*, may perform costly adaptations in response to transient imbalances. In essence, this is because the decision to adapt is taken on the basis of a snapshot of an environment for which there is no reliable way of predicting future trends. Many computational systems operate in comparably uncertain settings,



(a)



(b)



(c)

Fig. 11 Experiment 7 – Level of resource usage for adaptive strategies relative to no adaptation. (a) Constant imbalance of level 6, as in Figure 6. (b) Periodic imbalance of level 6 for duration and peak duration = 1s, as in Figure 7(a). (c) Poisson imbalance for 6 new jobs per second, as in Figure 7(c).

and the term *online algorithm* has been coined to refer to a solution to a problem where a sequence of requests is processed as each request arrives without knowledge of future requests [23]. Examples include paging in virtual memory systems [11,29], finding routes in networks, cache memory management [9] and dynamic selection of component implementations [38]. In essence, such systems must make decisions as to which configuration to use in future on the basis of previous performance. The term *competitive algorithm* is used of algorithms that seek to make decisions online that can be characterized by how well they compete with an optimal offline algorithm. One such algorithm is that of [38], which dynamically selects between implementations of components; we have modified this algorithm with a view to improving the decision as to *when* to change the distribution policy in *Adapt-1* and *Adapt-2*, and *what* to change it to.

5.1 Online algorithms for decision making in adaptive systems

In adaptive systems, adaptivity involves some change in system configuration with a view to improving performance. We illustrate this approach to adaptivity with the *adaptive component problem* [38], which was motivated by the increasing use of components, such as web services, to develop distributed applications. In an unpredictable environment, it may be impossible to determine statically which implementation of a component should be used in a given context. Hence *adaptive components* have multiple implementations, each optimized for request sequences with particular properties. Internally, an adaptive component monitors the current workload and adaptively switches to the implementation best suited to the workload. Adaptation takes place when the cost associated with the current configuration is greater than the cost associated with a different configuration by some factor. In [38], the Delta algorithm, determines when to switch between two configurations, and is 3-competitive (i.e., no worse than three times as slow as the optimal offline algorithm). Figure 12 describes a simplified version of Delta that suffices to motivate the approach followed in Section 5.2.

In the figure, the system can operate under two different configurations, *conf1* and *conf2*. Only one configuration can be active at any time, but the cost of processing each request is derived for both configurations, and thus a cost model is required for each configuration. A cost model is also required that can be used to predict the cost of switching between configurations: *switch_cost* is the round-trip switching cost, that is, the cost of changing from *conf1* to *conf2* and back again. A change of configuration occurs when the cost incurred by the active configuration (*conf1* in Figure 12) is greater, by *switch_cost* or more, than the cost incurred by the alternative configuration had it been active.

```
Conf1Cost = 0;
Conf2Cost = 0;
TimeToSwitch = False;
while (not TimeToSwitch)
    Process next request r;
    Conf1Cost = Conf1Cost + Cost(r, conf1);
    Conf2Cost = Conf2Cost + Cost(r, conf2);
    if (Conf1Cost1 - Conf2Cost) >= switch_cost
        TimeToSwitch = True;
    endif
endwhile
SwitchToConf2;
```

Fig. 12 The Delta algorithm; *conf1* is active.

```
start_time = now();
count = 0;
current_dp = ... ;
TimeToSwitch = False;
while (not TimeToSwitch)
    count = count + 1;
    Process next portion of query;
    Compute proposed_distribution_count;
    preferred_dp n in 1..length(proposed_distribution)[n]=
        sum t in 1..count(proposed_distribution_t[n])/count;
    period = now() - start_time;
    delay = accum_delay(current_dp, preferred_dp, period);
    if delay >= switch_cost
        TimeToSwitch = True;
    endif
endwhile
Switch to use preferred_dp;
```

Fig. 13 Revised Delta (from Figure 12) for load imbalance.

```
proc accum_delay(current_dp, preferred_dp, period)
df j in 1..length(current_dp)[j] = 0; % delay fraction
for i in 1..length(current_dp)
    if current_dp[i] > preferred_dp[i] % causing delay
        df[i] = (current_dp[i]-preferred_dp[i])/preferred_dp[i];
    else
        df[i] = 0; % not causing delay
    endif
endfor
return max(df)*period;
```

Fig. 14 Computing the accumulated delay.

5.2 Adapting *Delta* for use with *Adapt-1* and *Adapt-2*

We have taken Delta as a starting point for the design of variants of *Adapt-1* and *Adapt-2*. Both *Adapt-1* and *Adapt-2* use exchange operators to distribute tuples between parallel plan fragments, and revise the distribution policy in the light of runtime performance, as described in Section 2. The idea behind the revised versions of *Adapt-1* and *Adapt-2* is that adaptation takes place only when evidence has been accumulated that the consequences of ongoing imbalance are great enough to motivate incurring the costs associated with adapting. In particular, adaptation takes place when processing the backlog of work assigned to a node is predicted to take at least *switch_cost*, assuming that the throughput on the node continues as before. The *switch_cost* is the cost incurred to relocate hash table data to reflect the revised distribution. The revised version of Delta is listed in Figure 13.

Revised Delta first computes the preferred distribution as the average of the *proposed_distribution*s (as defined in the description of *Adapt-1* in Section 2) since the start of an operation or the last adaptation. The average proposed distribution is referred to as the *preferred distribution policy* ($preferred\_dp$); it is computed over a time period represented by 1..*count*, where each entry in 1..*count* represents a moment during query evaluation where the collection of the monitoring information required to compute a *proposed_distribution* took place.

As the $preferred\_dp$ is the average of previous proposed distributions, it can be computed efficiently and incrementally. During operation evaluation, $preferred\_dp$ is updated at each monitoring point (every 0.1s in the simulation) to take account of the *proposed_distribution* at that point, and used to compute both the *switch_cost* and the accumulated delay resulting from the use of the current distribution policy over the period 1..*count*.

The accumulated delay is computed as described in Figure 14, which estimates the level of the delay that will result from continuing with *current_dp* compared with changing to *preferred_dp*. In essence, the value of the ith entry in *current_dp* represents the fraction of the work that is currently assigned to the ith node. By contrast, the value of the ith entry in *preferred_dp* represents the fraction of the work that would have been assigned to that node using the strategy represented by *preferred_dp*. Where *current_dp[i]* is less than or equal to *preferred_dp[i]*, the node is able to evaluate all the work assigned to it, and thus does not contribute to any delay in the completion of the operator. By contrast, where *current_dp[i]* is greater than *preferred_dp[i]*, more work was assigned to a node than it was able to process, and thus the node is contributing to sub-optimal performance from the operator. Where this is the case, the difference between the distribution policies for the ith node represents the portion of the work that the node was unable to evaluate using the current distribution policy.

For example, if *current_dp[i]* is *0.5* and *preferred_dp[i]* is *0.2*, then the fraction of the total work assigned to the node that it cannot process is *0.3*. This difference is referred to here as the *inappropriate assignment level*. This leaves open the question as to how much delay will be incurred by the ith node if evaluation continues using *current_dp*. This is referred to as the *delay fraction* (*df* in Figure 14), and can be estimated by dividing the *inappropriate assignment level* for node i by *preferred_dp[i]*. For the example, this gives $(0.3/0.2) = 1.5$. That is, the delay resulting from the use of *current_dp* instead of *preferred_dp* is an additional 1.5 times what it would have taken using *preferred_dp*. Thus the result of *accum_delay* is the product of the largest delay fraction and the period for which inappropriate load balance has been in place.

Count	Tuples Processed N1	N2	Total	Backlog N1	N2	*accum_delay*
1	20	50	70	30	0	1.50
2	20	50	140	60	0	3.00
3	20	50	210	90	0	4.50
4	20	50	280	120	0	6.00
5	20	50	350	150	0	7.50
6	20	50	420	180	0	9.00
7	20	50	490	210	0	10.50
8	20	50	560	240	0	12.00
9	20	50	630	270	0	13.50
10	20	50	700	300	0	15.00

Table 5 A sample of accumulated delay for 2-level parallelism with $current\_dp$ = [0.50 0.50] and $preferred\_dp$ = [0.20 0.80]. The workload is equally divided between the two nodes ([0.50 0.50]), but node N1 is not able to cope with its portion of work, assuming that 100 tuples are available for processing for each count.

We use the delay fraction of the node with the largest delay fraction because this node will finish last, thereby showing how long the accumulated delay will actually be.

Table 5 illustrates how delay accumulates over time. In the table, we assume that the $current\_dp$ = [0.50 0.50], whereas the more suitable $preferred\_dp$ = [0.20 0.80]. Further, we assume that each *Count* represents a time period t, that *100* tuples are available to be processed in each period t, and that *N2* is able to process *80* tuples in that period. From the $preferred\_dp$, it follows that *N1* can process only *20* of the *50* tuples assigned to it by $current\_dp$ in time t, and thus that it accumulates a backlog of *30* tuples for each t. As *N1* can only process *20* tuples in time t, after a single t it will take $1.5 * t$ to clear its backlog, a delay that will grow linearly while the relative performance of *N1* and *N2* stays the same. Note that $N2$ can process all of the 50 tuples assigned to it by $current\_dp$, with additional spare capacity, so it does not accumulate any backlog.

Another aspect of the Revised Delta algorithm in Figure 13 is that the $switch\_cost$ between configurations is variable, as it depends on the amount of data that needs to be relocated for use in the hash joins ($switch\_cost$ in Delta is fixed). This means that many adaptations are likely to occur when $switch\_cost$ is relatively low even though there is little problem with the current distribution. For this reason we retain the minimum threshold values on adaptation sizes from Section 2 in both *Adapt-1* and *Adapt-2*; this means that the revised versions are guaranteed not to adapt more frequently than their original counterparts. In essence, an additional hurdle has been introduced before adaptation can take place. This hurdle ensures that the problem with the current distribution policy is known to have been sustained to the extent that, if applied to the same amount of data to which it has already been applied, a change to the $preferred\_dp$ will lead to improved response times, even after taking into account the cost of relocating the hash table.

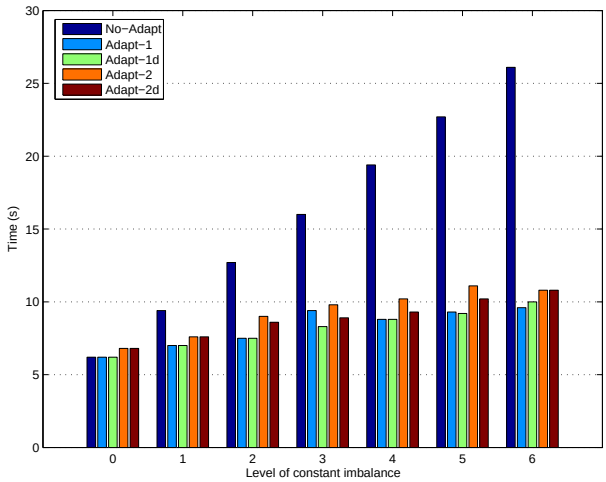


Fig. 15 Experiment 8 – Response times for Q1 for different levels of constant imbalance.

5.3 Experiments

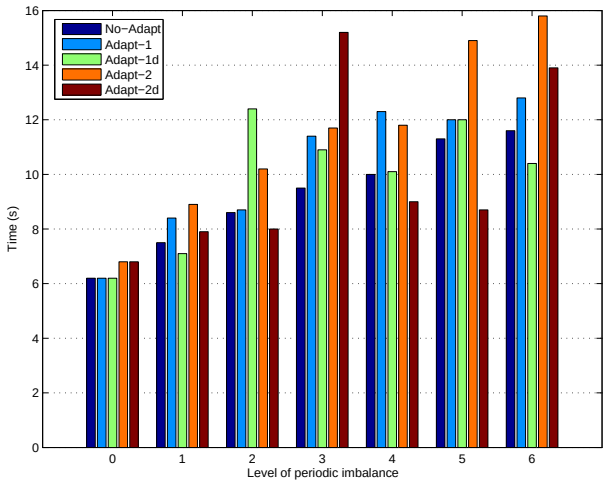
This section compares the performance of *Adapt-1*, *Adapt-2* and the revised versions *Adapt-1d* and *Adapt-2d*.

Experiment 8: *Effectiveness of revised Delta in the context of constant imbalance.* This experiment, like Experiment 2, 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.

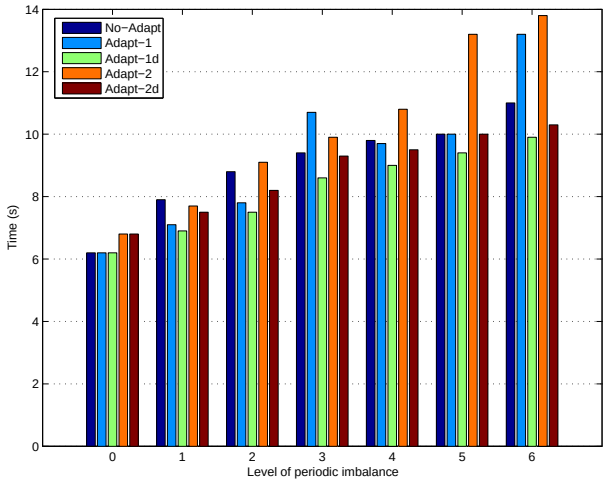
The following can be observed from the results in Figure 15: all adaptive strategies perform well compared with the base case – this is to be expected, as the imbalance to which a reaction is required is consistent and present from the start, enabling adaptation to take place before much operator state has accumulated. This positive result is to be expected; as load imbalance is constant, the snapshots of performance used in *Adapt-1* and *Adapt-2* are generally representative of ongoing behavior. As a result, adapting to average and snapshot imbalance levels leads to similar distribution policies.

Experiment 9: *Effectiveness of revised Delta in the context of periodic imbalance.* This experiment, like Experiment 3, 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), or in which jobs arrive following a Poisson distribution.

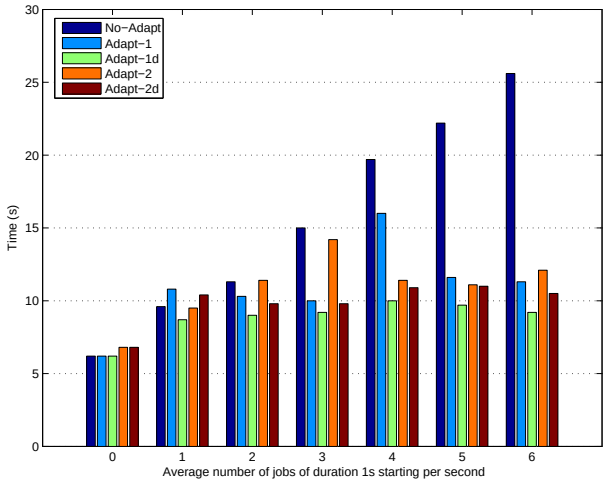
The following can be observed: (i) the rapid changes in load in Figures 16(a) and (b) present a challenging environment for algorithms that perform costly adaptations, and none of the algorithms consistently improve on the base case to a significant degree; (ii) in both Figure 16(a) and (b), the original versions usually perform less well than those using the revised Delta algorithm to decide when to adapt; this is because adapting to average levels of imbalance generally leads to fewer large-



(a)



(b)



(c)

Fig. 16 Experiment 9 – Response times for Q1 for different levels of periodic imbalance: (a) duration and repeat duration = 1s; (b) duration and repeat duration = 5s; (c) following a Poisson distribution.

scale adaptations than in the originals, in which changes in distribution policy tend to be significant and frequent; (iii) in Figure 16 (c), all the adaptivity strategies improve in the base case for higher levels of imbalance, although the versions based on Delta generally out-perform their counterparts, as would be expected – the average imbalance levels that form the basis of the preferred distribution policy should accurately reflect the average loads resulting from the Poisson-based arrival rates for external jobs.

Overall, the revised versions of *Adapt-1* and *Adapt-2* have helped to reduce their weaknesses in unstable settings, but more by avoiding extremes of behavior than by providing fundamental changes in behavior across the board.

6 Conclusions

This paper has compared techniques for balancing load during the evaluation of stateful 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.

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 17. In essence, *Adapt-1* to *Adapt-3* 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 is expensive. However, *Adapt-3* has lower maximum adaptation costs because the use of replication means that the same hash table state will never be moved repeatedly to the same node. By contrast, *Adapt-4* 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-5* 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.

Another property that is relevant to the effectiveness of adaptation is the *stability* of the property to which adaptivity is responding. Some adaptive strategies focus significant attention on ensuring that they adapt only when there is significant evidence that the problem to which a response is being made is great enough to imply that adaptation will be beneficial. For example, both

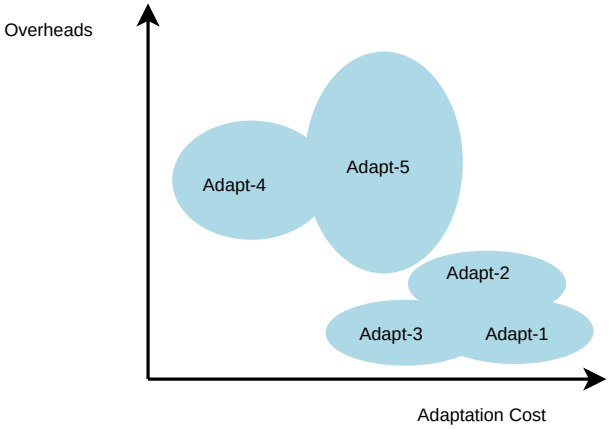


Fig. 17 Trade offs involved in the techniques.

POP [24] and Rio [3] adapt when the selectivity of an operator moves outside a range for which a plan is expected to be effective. As the actual selectivity of a predicate can be computed based on accumulating evidence, the processing of small numbers of tuples with atypical selectivities is unlikely to lead to a proposal to reoptimize – as such, the selectivity is reasonably *stable*. However, even when adapting to changes in predicate selectivities, it is possible for an adaptive strategy to do more harm than good because adaptations react to transient effects; POP has a threshold that limits the number of times it will adapt to limit these potentially harmful consequences.

Unfortunately, load imbalance is not intrinsically stable. Events over which a query processor has no control can lead to substantial changes in 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* to *Adapt-3*), are likely to perform poorly where the load changes rapidly. *Adapt-4*, was designed to overcome this weakness. However, although *Adapt-4* provides dependable performance in unstable environments, it has significant overheads, and thus loses out where there is little imbalance.

Adapt-5 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-5* is only likely to improve on *Adapt-1* to *Adapt-3* where there is high external load or in contexts where *Adapt-1* to *Adapt-3* struggle – for example, when load changes rapidly. However, an advantage of the fact that *Adapt-5* 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 algorithms based on Delta also respond to the consequences of sustained imbalance by accumulating evidence that change is necessary. The variants of *Adapt-1* and *Adapt-2* that adopt this principle were developed

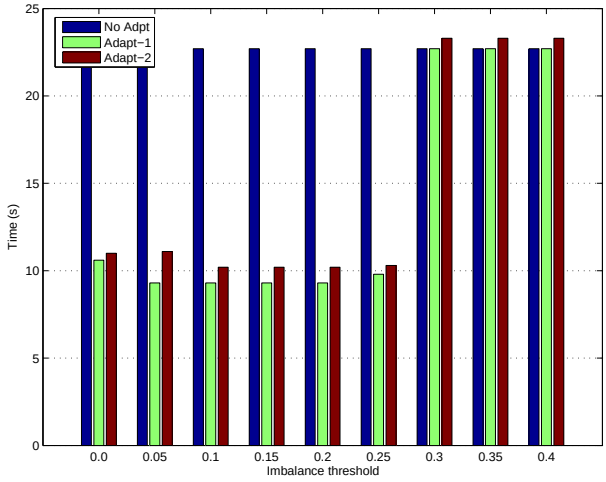
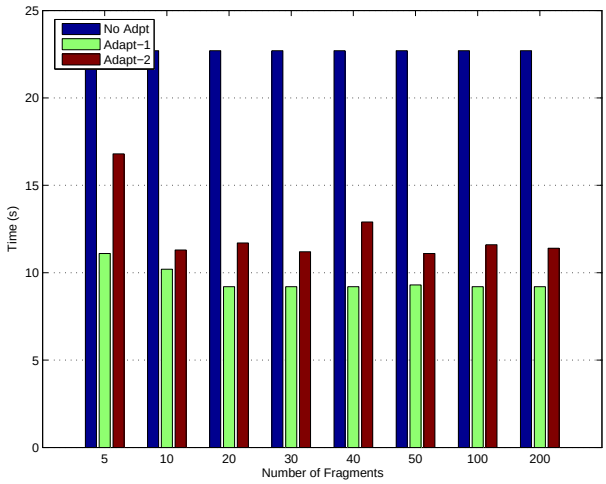


Fig. 18 Sensitivity to Imbalance Threshold – Response times for Q1 running on three nodes with an imbalance level of 5 and varying imbalance threshold: (a) constant imbalance; (b) periodic imbalance with duration and repeat duration = 1s.

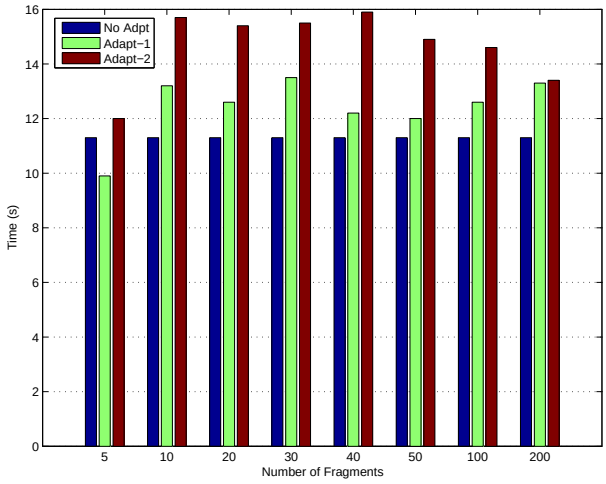
with a view to reducing the consequences of hasty adaptations in unstable environments. Like *Adapt-3*, which was designed for the same purpose, they helped to reduce the effects of worst-case scenarios, without significantly widening the range of circumstances in which *Adapt-1* and *Adapt-2* perform well.

Appendix A: Sensitivity Analysis

The adaptivity strategies *Adapt-1*, *Adapt-2* and *Adapt-3* are associated with parameters that influence their behavior, as described in Section 2. To ensure that the results reported in the paper are not sensitive to small changes in these parameters, tests have been performed that vary these parameters systematically.



(a)



(b)

Fig. 19 Sensitivity to Number of Fragments – Response times for Q1 running on three nodes with an imbalance level of 5 and varying imbalance threshold: (a) constant imbalance; (b) periodic imbalance with duration and repeat duration = 1s.

The imbalance threshold is the smallest change in the distribution policy for which an adaptation is considered (this has been set to *0.05* in the experiments). Figure 18 shows the effect of varying this threshold for *Adapt-1* and *Adapt-2*.

In Figure 18(a), for constant imbalance, the adaptive strategies improve significantly on the *No Adapt* case for a wide range of imbalance thresholds, until the threshold is such as to block adaptation completely. With constant imbalance, in both *Adapt-1* and *Adapt-2*, adaptation takes place a small number of times, and a plausible distribution policy can be identified early in query evaluation. Thus, as long as the threshold is not so large as to prevent the use of this policy, the adaptivity strategies perform well.

In Figure 18(b), for periodic imbalance, the adaptive strategies struggle to cope with the unstable environment, until the threshold is such as to block adaptation completely. With periodic imbalance, adaptation takes place regularly, and higher imbalance thresholds help to reduce the negative consequences of snap decisions to adapt. However, high thresholds can lead to missed opportunities when adaptation is beneficial, as illustrated in Figure 18(a), so the threshold has been kept quite low in the experiments.

The number of fragments indicates the number of groups into which each table is divided, where the groups are the unit of redistribution. Thus the number of table fragments indicates the granularity at which data may be redistributed. The number of table fragments has been set to *50* in the experiments. Figure 19 shows the effect of varying this threshold for *Adapt-1* and *Adapt-2*.

In Figure 19, where there are small numbers of table fragments, this can reduce the numbers of adaptations that take place. This is problematic in the case of constant imbalance in Figure 19(a), and beneficial in the case of variable imbalance in Figure 19(b). With larger numbers of fragments, although specific results vary, the overall behavior of the algorithms remains broadly consistent.

Overall, the sensitivity analysis illustrates that extreme values for the control parameters can restrict the number of adaptations that take place, but that the behavior of the algorithms remains broadly consistent for a range of other values. We note that both *Adapt-1* and *Adapt-2* perform costly adaptations in response to potentially transient effects, and thus that some variation is expected in measurements as parameters are altered, even where this does not follow a regular pattern.

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. S. B. Davidson, J. Crabtree, B. P. Brunk, J. Schug, V. Tannen, G. C. Overton, and C. J. Stoeckert. K2/Kleisli and GUS: Experiments in Integrated Access to Genomic Data Sources. *IBM Systems Journal*, 40(2):512–531, 2001.
7. D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri. Practical skew handling in parallel joins. In *Proc. VLDB*, pages 27–40, 1992.

8. D.J. DeWitt. Parallel Database Systems: The Future of High Performance Database Systems. *Comm ACM*, 35(6):85–98, 1992.
9. S. J. Eggers and R. H. Katz. Evaluating the performance of four cache coherency protocols. In *Proceedings of the 16th International Symposium on Computer Architecture*, pages 2–15, 1989.
10. S. Ewen, H. Kache, V. Markl, and V. Raman. Progressive query optimization for federated queries. In *Proc. 10th EDBT*, pages 847–864. Springer, 2006.
11. A. Fiat et al. Competitive paging algorithms. *J. Algorithms*, 12:685–699, 1991.
12. 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.
13. 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.
14. G. Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. In *Proc. SIGMOD*, pages 102–111, 1990.
15. G. Graefe. Iterators, Schedulers, and Distributed Memory Parallelism. *Software Practice and Experience*, 26(4):427–452, 1996.
16. 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.
17. 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.
18. V. Josifovski, P. Schwarz, L. Haas, and E. Lin. Garlic: A New Flavor of Federated Query Processing for DB2. In *Proc. SIGMOD*, pages 524–532, 2002.
19. D. Kossmann. The State of the Art in Distributed Query Processing. *ACM Computing Surveys*, 32(4):422–469, 2000.
20. D.J. Lilja. *Measuring Computer Performance*. Cambridge University Press, 2000.
21. D.T. Liu and M.J. Franklin. GridDB: A Data-Centric Overlay for Scientific Grids. In *Proc. VLDB*, pages 600–611. Morgan-Kaufmann, 2004.
22. 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.
23. M. S. Manasse, L. A. McGeoch, and D. D. Sleator. Competitive algorithms for on-line problems. In *Proceedings of the twentieth annual ACM symposium on Theory of Computing*, pages 322–333, 1988.
24. 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.
25. 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.
26. N.W Paton, V. Raman, G. Swart, and I. Narang. Autonomic Query Parallelization using Non-dedicated Computers: An Evaluation of Adaptivity Options. In *Proc. 3rd Intl. Conference on Autonomic Computing*, pages 221–230. IEEE Press, 2006.
27. E. Rahm and R. Marek. Analysis of dynamic load balancing strategies for parallel shared nothing database systems. In *Proc. VLDB*, pages 182–193, 1993.
28. V. Raman, W. Han, and I Narang. Parallel querying with non-dedicated computers. In *Proc. VLDB*, pages 61–72, 2005.
29. T. H. Romer, W. H. Ohlrich, A. R. Karlin, and B. N. Bershad. Reducing TLB and Memory Overhead Using Online Superpage Promotion. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 176–187, 1995.
30. 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.
31. M.A. Shah, J.M. Hellerstein, and E.A. Brewer. Highly available fault-tolerant, parallel dataflows. In *Proc. SIGMOD*, pages 827–838, 2004.
32. 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.
33. J. Smith, A. Gounaris, P. Watson, N.W. Paton, A.A.A. Fernandes, and R. Sakellariou. Distributed query processing on the grid. *Intl. J. High Performance Computing Applications*, 17(4):353–368, 2003.
34. J. Smith and P. Watson. Fault-Tolerance in Distributed Query Processing. In *Proc. IDEAS*, pages 329–338. IEEE Press, 2005.
35. 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.
36. F. Tian and D.J. DeWitt. Tuple Routing Strategies for Distributed Eddies. In *Proc VLDB*, pages 333–344, 2004.
37. T. Urhan and M.J. Franklin. XJoin: A Reactively-Scheduled Pipelined Join Operator. *Data Eng. Bulletin*, 23(2):27–33, 2000.
38. D. M. Yellin. Competitive algorithms for the dynamic selection of component implementations. *IBM Systems Journal*, 42(1):85–97, 2003.
39. 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.