Chapter 9
Adaptive Query Processing in Distributed Settings

Anastasios Gounaris, Efthymia Tsamoura, and Yannis Manolopoulos

Abstract. In this survey chapter, we discuss adaptive query processing (AdQP) techniques for distributed environments. We also investigate the issues involved in extending AdQP techniques originally proposed for single-node processing so that they become applicable to multi-node environments as well. In order to make it easier for the reader to understand the similarities among the various proposals, we adopt a common framework, which decomposes the adaptivity loop into the monitoring, analysis, planning and actuation (or execution) phase. The main distributed AdQP techniques developed so far tend to differ significantly from their centralized counterparts, both in their objectives and in their focus. The objectives in distributed AdQP are more tailored to distributed settings, whereas more attention is paid to issues relating to the adaptivity cost, which is significant, especially when operators and data are moved over the network.

9.1 Introduction

The capability of database management systems to efficiently process queries, which are expressed as declarative statements, has played a major role in their success over the last decades. Efficiency is guaranteed due to several sophisticated optimization techniques, which heavily rely on the existence of metadata information about the data to be processed, such as the distribution of values and the selectivity of the relational operators. Since the late 1970s and the introduction of System R [58], static optimization of query plans and subsequent execution has been the main choice for database system developers. However, when the metadata required are not available or accurate at compile time, or when they change during execution,

Anastasios Gounaris · Efthymia Tsamoura · Yannis Manolopoulos
Aristotle University of Thessaloniki, Thessaloniki, Greece
e-mail: {gounaria,etsamour,manolopo}@csd.auth.gr

the query processor needs to revise the current execution plan on the fly. In this case, query processing is called adaptive.

In adaptive query processing (AdQP), there is a feedback loop, similar to the one appearing in autonomic systems, according to which the query processor monitors its execution properties and its execution environment, analyzes this feedback, and possibly reacts to any changes identified with a view to ensuring that either the current execution plan is the most beneficial or a modification of the current plan can be found that is expected to result in better performance.

Although AdQP is particularly relevant to wide area settings, in which query statistics are more likely to be limited or potentially inaccurate, and the computational properties, such as the processing capacity of hosting machines, are volatile, most AdQP proposals have focused either on completely centralized query processing or on centralized processing of data retrieved or stemming from remote sources and data streams, respectively. In such settings, there is typically a single physical machine used for query execution, which is predefined, and thus the focus is mostly on adapting to changing properties of the data processed, e.g., cardinalities of intermediate results and operator selectivities. This is, of course, of high importance for distributed query processing (DQP), as crucial information about the data may be missing at compile time. However, of equal significance are adaptations to changing properties of a potentially arbitrary set of resources that DQP may employ and of their communication links. Currently, AdQP with respect to changing resources is not addressed as satisfactorily as with respect to changing data properties.

In this survey chapter, we systematically discuss AdQP techniques that are tailored to distributed settings both with respect to the data sources and the processing nodes. We also investigate the issues involved in extending AdQP techniques originally proposed for single-node processing so that they become applicable to multi-node environments as well. In order to make it easier for the reader to understand the similarities among the various proposals, we adopt a common framework, which decomposes the adaptivity loop into its constituent phases mentioned above, i.e, monitoring, analysis, planning and actuation phase. The later corresponds to the phase, in which the adaptivity decisions are executed by the system.

Structure. The structure of this chapter is as follows. In the remainder of this section we briefly discuss preliminary concepts of distributed query processing and optimization (Section 9.1.1), and related work (Section 9.1.2). In Section 9.2, we present the framework that forms the basis of our analysis. The next section contains a short review of traditional AdQP for centralized settings and explains the reasons why such techniques cannot be applied to wide-area environments in a straightforward manner. The discussion of the AdQP techniques for distributed settings, which is the core part of this chapter, is in Sections 9.4-9.6. Existing work in distributed AdQP techniques can be classified in three broad categories. Techniques that do not rely on the existence of traditional query plans fall into the first category, which is examined in Section 9.4. The second category comprises approaches that perform load management at the operator level (Section 9.5), whereas, in Section 9.6, we discuss distributed AdQP techniques where the adaptivity occurs at a higher level.

The final section (Section 9.7) contains an assessment of the current status in the area, along with directions for future work, and concludes the chapter.

9.1.1 Distributed Query Processing Basics

Distributed query processing consists of the same three main phases as its centralized counterpart, namely parsing (or translation), optimization and execution. During parsing, the initial query statement, which is expressed in a declarative language such as SQL, is translated into an internal representation, which is the same for both centralized and distributed queries [42].

Query optimization is commonly divided into two parts, query rewriting and cost-based plan selection. Query rewriting is typically carried out without any statistical information about the data and independently of any previous physical design choices (e.g., data locations, existence of indices) apart from the information about data fragments. In distributed queries over non-replicated fragmented data, the relevant data fragments are identified during this procedure as well [51]. Secondly, cost-based optimization is performed. The search strategy typically follows a dynamic programming approach for both centralized and distributed queries [43, 46] provided that the query is not very complex in terms of the number of different choices that need to be examined; in the latter case the plan space is reduced with the help of heuristics. Traditional cost based optimization is capable of leading to excellent performance when there are few correlations between the attributes, adequate statistics exist and the environment is stable.

The optimized plan is subsequently passed on to the query execution engine, which is responsible for controlling the data flow through the operators and implementing the operators. Although in both traditional disk-based queries and continuous queries over data streams the operators are typically those that are defined by the relational algebra (or their modifications [69]), the execution engine may differ significantly. In disk-based queries, the pull-based iterator model of execution is preferable [31], according to which each operator adheres to a common interface that allows pipelining of data, while explicitly defining the execution order of query operators and ensuring avoidance of flooding the system with intermediate results in case of a bottleneck. On the other hand, continuous queries over data streams may need to operate in a push-based mode [8]. The main difference between the push and pull model of execution lies in the fact that, in the push model, the processing is triggered by the arrival of new data items. This property may give rise to issues that are not encountered in pull-based systems, which have full control on the production rate of intermediate results. For example, push-based query processors may need to resort to approximation techniques when data arrival rates exceed the maximum rate in which the system can process data. We do not deal with approximation issues in this chapter; we refer the interested reader to Chapter 7 of this book. However, it is worth mentioning that AdQP in push-based systems considers additional issues, such as adaptations to the data arrival rates.

In conventional static query processing, the three phases of query processing occur sequentially, whereas, in AdQP, query execution is interleaved with query optimization in the context of a single query with a view to coping with the unpredictability of the environment, and evolving or inaccurate statistics. According to a looser definition of AdQP, the feedback collected from the query execution of previous queries impacts on the optimization of future queries (e.g, [61]); we do not deal with such flavors here. Note that the need for on-the-fly re-optimizations is mitigated with the help of advanced optimization methodologies, such as robust optimization (e.g., [3, 12]). Also, other topics related to AdQP are discussed in Chapter 10 (on combining search queries and AdQP).

9.1.2 Related Work

A number of surveys on AdQP have been made available [35, 26, 4, 18]. However, none of them focuses on distributed queries over distributed resources, although the work in [18] is closer in spirit to this chapter in the sense that it adopts the same describing framework. Static DQP is described in [51, 42], whereas the work in [31] discusses query processing issues in detail.

9.2 A Framework for Analysis of AdQP

AdQP can be deemed as the main means of self-optimization in query processing, and, as such, it relates to autonomic computing. According to the most commonly used autonomic framework, which is introduced in [41], at the conceptual level, autonomic managers consist of four parts, namely monitoring, analysis, planning and execution, whereas they interface with managed elements through sensors and effectors. In line with this decomposition, a systematic discussion about distributed AdQP distinguishes between monitoring, analysis, planning and execution. Note that these parts need not necessarily correspond to distinct implemented components at the physical level.

Monitoring involves the collection of measurements produced by the sensors. In the context of query processing, the types of measurements include data statistics (e.g., cardinalities of intermediate results), operator characteristics (e.g., selectivities) and resource properties (e.g., machine CPU load). The feedback collected is processed during the analysis phase with a view to diagnosing whether there is an issue with the current execution plan. If this is the case, then an adaptation is planned, which can be thought of as an additional query plan along with operations that ensure final result correctness. Execution is concerned with the actuation of the planned adaptations. Planned adaptations are executed either immediately in simple scenarios, or, in more complex cases (e.g., when internal state of some operators must be modified first), after certain procedures have been followed.

In this chapter, we follow the approach in [18] and we provide a summary of the measurements collected, and the analysis, planning and actuation procedures that are encapsulated in each of the main AdQP techniques presented. Note that these aspects of AdQP may be arbitrarily interleaved with query processing. For example, in some techniques, measurements' collection occurs after query processing has been suspended (e.g., due to materialization points), whereas other techniques continuously generate monitoring information during query execution. Also, analysis and planning may be tightly connected, since, in some cases the analysis of the feedback is done in a way that identifies better execution plans as well. Due to this fact, we prefer to examine analysis along with planning. Note that other variants of this framework, such as the one in [27], may regard planning and execution as a single response phase, whereas, during monitoring, preliminary analysis may be performed to filter uninteresting feedback.

9.3 AdQP in Centralized Settings

The role of this section is twofold. Firstly, it provides a short review of the main techniques employed in centralized AdQP, which is thoroughly investigated in surveys such as the one in [18]. Secondly, it discusses the feasibility of applying such techniques in distributed settings.

9.3.1 Overview of Techniques

In broad terms, the objective of conventional AdQP is to take actions in light of new information becoming available during query execution in order to achieve better query response time or more efficient CPU utilization. Although AdQP can be applied to plans consisting of any type of operators, there exist operators that naturally lend themselves to adaptivity in the sense that they facilitate plan changes at runtime. Such operators include symmetric hash joins and the proposals that build on top of them (e.g, XJoins [63]), multi-way pipelined joins (e.g., [65]) and eddies [2]. All the operators mentioned above can be complemented with additional operators that encapsulate autonomic aspects within their design; for example, certain operator implementations provide built-in support to adapt to the amount of the memory available (e.g., [52]).

Eddies constitute one of the most radical adaptive techniques on the grounds that they do not require explicit decisions on the ordering of commutative and associative operators (e.g., selections and joins). This results in a much simpler query optimization phase. Eddies have been proposed in order to enable fine-grained adaptivity capabilities during query execution; actually, they allow each tuple to follow a different route through the operators. More specifically, in eddies, the order of commutative and associative operators is not fixed and adaptations are performed by simply changing the routing order. To this end, several routing policies have been proposed. The eddy operator is responsible for taking the routing decisions according to the policy adopted and monitoring information produced by the execution of tuples. As an example, assume that a long-running query over three relations of equal size is processed with the help of two binary joins. At the beginning, the selectivity of the first join is much lower than the selectivity of the other and the eddy will route most of the tuples to the most selective one. Later, the second join becomes more selective (e.g., due to the existence of a time-dependent join attribute); the eddy will be capable of swapping the order of join execution. Any static optimization decision on the join ordering would fail to construct a good plan in such a scenario where a different ordering yields better results in different time periods during query execution.

Hybrid approaches that combine eddies with more traditional optimization have been proposed as well. For example, in [49] and [48], a methodology is proposed where multiple plans exist and the incoming tuples are routed to these plans with the help of an eddy. Each such plan is designed for a particular subset of data with distinct statistical properties. In general, several extensions to the original eddy operator have been made (e.g., [56], [16], [10], [14]).

Another notable centralized AdQP technique has been proposed in [6], which adaptively reorders filtering operators. This proposal takes into account the correlation of predicates and can be used to enhance eddies routing policies. It has also been extended to join queries [7]. In general, join queries are treated in a different manner depending on whether they are fully pipelined and whether adaptations impact on the state that is internally built within operators because of previous routing decisions. Non-pipelined join queries were among the first types of queries for which AdQP techniques have been proposed. Such techniques are typically based on the existence of materialization points and the insertion of checkpoints, where statistics are collected and the rest of the adaptivity loop phases may take place if significant deviations from the expected values are detected (e.g., [40]). Two specific types of generalizations of these works are referred to as progressive optimization [47] and proactive optimization [5], respectively. Adaptive routing history-dependent pipelined execution of join queries is one of the most challenging areas in AdQP, where proposals exist that either use conventional query plans (e.g., [39]) or eddies (e.g., [16]).

9.3.2 On Applying Conventional AdQP Techniques in Distributed Settings

Undoubtedly, distributed AdQP techniques can benefit from the adaptive techniques proposed for a centralized environment. In DQP, each participating site receives a sub-query, which can be executed in an adaptive manner with the help of the techniques described previously. However, these adaptations, which are restricted to sub-queries only, are not related to each other, and, as such, they are not guaranteed to improve the global efficiency of the execution. For example, suppose that a set of operators in a query plan are sent to multiple machines simultaneously according to the partitioned parallelism paradigm [19]. The execution of a query operator in a plan may benefit from partitioned parallelism when this operator is instantiated several times across different machines with each instance processing a distinct data partition. An eddy running on each of those machines could be very effective in detecting the most beneficial operator order at runtime; nevertheless, nothing can be done if the workload allocated to each of these machines is not proportional to their actual capacity.

In general, when AdQP techniques that were originally proposed for single-node queries are applied to full DQP, their efficiency is expected to degrade significantly due to the following reasons.

- Firstly, adaptations may impact on the state built within operators, as explained in [16, 73]. State movements in DQP incur non-negligible cost due to data transmission over the network. If this cost is not taken into account during the planning phase, then, the associated overhead may outweigh any benefits. Centralized AdQP techniques that manipulate the operator state in order to improve performance do not consider such costs, whereas, if state movement is avoided, then the adaptivity effects may be limited [16]. This situation calls for new AdQP techniques tailored to distributed settings.
- Secondly, several of the AdQP techniques mentioned above involve a final stitch-up (or clean-up) phase, which is essential for result correctness (e.g., [39]). As with state movement, when such a phase is applied to distributed plans, then additional overhead is incurred, which needs to be carefully assessed before proceeding with adaptations.
- Thirdly, direct applications of centralized AdQP techniques result in techniques in which there is a single adaptivity controller responsible for all the adaptivity issues. Obviously, this may become a bottleneck if the number of participating machines and/or the volume of the feedback collected is high. Scalable solutions may need to follow more decentralized approaches, which has not been examined in single-node settings.
- Finally, the optimization criteria may be different, since issues, such as load balancing, economic cost, energy efficiency are more likely to arise in DQP. These issues are closely related to load allocation across multiple machines, which is an aspect that does not exist in centralized environments.

Overall, the focus of distributed AdQP is different due to the fact that overhead and scalability issues are more involved, while load management is performed at a different level. The techniques described in the sequel address some of these issues.

9.4 AdQP for Distributed Settings: Extensions to Eddies

The original eddies implementation in [2] and its variants mentioned in Section 9.3 cannot be applied to a distributed setting in a straightforward manner. This is due to the fact that the eddy architecture is inherently centralized in the sense that all tuples

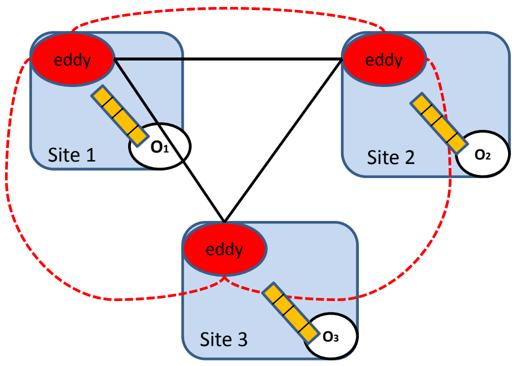

Fig. 9.1 Example of a distributed eddy architecture. The dashed lines show the statistics flow among the eddy operators, while the solid lines show the tuple flow.

must be returned to a central eddy; obviously, this paradigm leads to a single-point bottleneck, unnecessary network traffic and delays when operators are distributed. This section presents solutions to these problems.

9.4.1 Techniques

In [62], a distributed eddy architecture is proposed, which does not suffer from the limitations mentioned above. More specifically, in [62], each distributed operator is extended with eddy functionality. Moreover, a distributed eddy reaches routing decisions independently of any other eddies. Each operator places the received tuples in a first-come first-served queue. After a tuple has been processed, it is forwarded to the local eddy mechanism, which decides on the next operator that the tuple may be passed to based on the tuple's execution history and statistics. The operators in the distributed eddy framework learn statistics during execution and exchange such information among them periodically, e.g., after some units of time have passed or after having processed a specific amount of tuples. As in traditional eddies, routing decisions need not take place continuously; on the contrary, they may be applied to blocks of tuples in order to keep the associated overhead low [15]. Figure 9.1 shows an example of a distributed eddy architecture in a shared-nothing cluster of three sites.

The differences between centralized and distributed eddies are not only at the architectural level. The distributed eddies execution paradigm may be employed to minimize the result response time or to maximize the tuple throughput. For both objectives, it can be easily proved that the optimal policy consists of multiple execution plans that are active simultaneously, in the spirit of [17]; note that AdQP techniques typically consider the adaptation of a single execution plan that is active at each time point. However, analytical solutions to this problem are particularly expensive due to the combinatorial number of alternatives, and, in addition, they require the existence of perfect statistical knowledge; assuming the existence of perfect

Table 9.1 Adaptive control in Distributed Eddies [62].

Measurement: The eddy operators exchange statistics periodically regarding (i) the selectivity, (ii) the cost and (iii) the tuple queue length of the operators they are responsible for.
Analysis-Planning: The routing is revised periodically employing routing policies tailored to distributed settings. The routing decisions are made for groups of tuples.
Actuation: The planned decisions take effect immediately; no special treatment is needed since operators' internal state is not considered.

knowledge in a centralized statistics gathering component is not a realistic approach. So, the efficiency of distributed eddies relies on the routing policies. Interestingly, the most effective routing policies are different from those proposed in [2].

More specifically, several new routing policies are introduced in [62], in addition to those proposed for centralized eddies. Basic routing policies for centralized eddies include *back-pressure*, which considers operator load, and *lottery*, which favors the most selective operators. In the *selectivity cost* routing policy in [62], both the selectivity and the cost of the operators are considered in a combined manner. Although the above policy considers two criteria, it does not consider the queuing time spent while a tuple waits for processing in an operator's input queue. Thus tuples are routed to an operator regardless of its current load. In order to overcome this weakness, the *selectivity cost queue-length* policy takes into account the queue lengths of the operators as well. Contrary to the spirit of centralized eddies, the policies mentioned above are deterministic rather than probabilistic; note that this property is orthogonal to adaptivity. Finally, two more policies are proposed that route tuples in a probabilistic way that is proportional to the square of the metrics estimated by the selectivity cost and the selectivity cost queue-length policies, respectively. According to the experiments of the authors of [62], taking the square of the metrics exhibited better performance than taking the metrics alone.

Overall, during the monitoring phase, the raw statistics that need to be collected from each operator include its average tuple queue length, its selectivity and its processing cost. The routing policies effectively plan any adaptations. The actuation cost incurred when an alternative routing is adopted is negligible; the tuples are simply routed according to the new paths. Cases where the adaptation overhead cost is non-negligible, e.g., where operators create and hold internal state, which is affected by adaptations, are not investigated. The high-level summary of distributed eddies in [62] is given in Table 9.1.

The work of Zhou *et al.* in [72] extends the distributed eddies architecture proposed in [62] with SteMs [56]. SteMs add extra opportunities for adaptivity, since, apart from operator ordering, they can also change the join algorithm implementation (e.g., index-based vs. hash join) and the data source access methods (e.g., table scan vs. indexed access based on an attribute's column) at runtime. The local eddy operators utilize the traditional back-pressure and the lottery-based routing

Table 9.2 Adaptive control in [72].

Measurement: The eddy operators exchange statistics periodically. The exchanged statistics include (i) the average tuple queue length of the operators and (ii) the number of output tuples that are generated by the operators for the number of input tuples supplied to them.
Analysis-Planning: The routing is revised periodically. The back-pressure and the lottery based routing policies are employed for batches of tuples.
Actuation: The planned decisions take effect immediately; no special treatment is needed since operators' internal state is not considered.

strategies proposed for centralized settings [2]. In a distributed setting, the former routing strategy is used to accommodate the network transmission speeds and site workload conditions, while the latter reflects the remote operators' selectivity. Note that, as in [62], statistics are exchanged among the remote eddies at periodic time intervals, while routing decisions are made for groups of tuples. The statistics required by an eddy operator in [72] include (i) the average tuple queue length of the operators, and (ii) the number of output tuples that are generated by the operators. As in [62], the overhead incurred when an alternative routing is enforced, is negligible (see Table 9.2). Finally, FREddies is a distributed eddies framework for query optimization over P2P networks [36], which shares the same spirit as [62, 72].

9.4.2 Summary

The proposals described above are an essential step towards the application of eddies in DQP. However, a common characteristic is that the techniques in this category tend to avoid costly adaptations that involve manipulation of operators' internal state in order to diminish the risk of causing performance regression. A side-effect of such a reserved policy is that further opportunities to improve the performance of AdQP may be missed, as shown by successful relevant examples of AdQP techniques in centralized settings (e.g., [16, 21]. Another observation is that more research is needed in order to understand what type of routing policies is more efficient in distributed settings, and what are the benefits of probabilistic versus deterministic routing and of routing policies that are closer in spirit to flow algorithms [17].

9.5 AdQP for Distributed Settings: Operator Load Management

Load management can be performed at several levels; at operator-level load management, the main unit of load is an operator instance. In intra-operator load management, the different operator instances correspond to the same logical operator, which implies that partitioned parallelism is employed and adaptivity is concerned with only a part of the query plan. On the other hand, inter-operator load management deals with adaptations that consider the whole plan, and operator instances may correspond to different logical operators.

9.5.1 Intra-Operator Load Management

Horizontal partitioning is a common approach to scale operators in a shared-nothing cluster [19]. In horizontal partitioning, an operator is divided into multiple instances. Each such instance is placed on a different site and processes different subsets of the input data. The operators on the different sites can work in parallel. Thus the result of the operator is given by aggregating the partial results that were produced by the different operator instances. For example, the result of an equi-join operator $A \bowtie B$ is given by the union of the partial results $A_i \bowtie B_i$, $i = 1, \ldots, P$, where P is the degree of parallelism, i.e., the number of operator instances that work in parallel on different data. A_i and B_i correspond to the subsets of data that are processed at the $i-th$ site and are partitioned according to the join attribute.

9.5.1.1 Background

A straight-forward way to enable query plans to benefit from partitioned parallelism without modifying the operators, such as joins and aggregates, is through the insertion of *exchanges* [30]. The exchange operator is one of the most notable non-intrusive attempts to parallel operator evaluation. The operator does not modify or filter any tuples but aims to distribute tuples across different operator instances. The exchange operator is logically partitioned into two components that may be hosted on different sites. The consumer component resides at a consumer operator instance and waits for tuples coming from the upstream producer operator instances. The producer component encapsulates the routing logic: it is responsible for routing the tuples to the consumer operator instances. The most common routing policies are hash-based, value range-based and round-robin.

Figure 9.2(top) shows an example of the partitioned execution of the hash-join $A \bowtie B$ in a shared-nothing cluster of four sites using an exchange operator. The tuples from the left relation A are used to build the hash-table, while the tuples from B probe the hash-table. As the different operator instances in sites 1 and 2 work in parallel, the time needed to complete the evaluation of the join operator equals the time needed by the slowest operator instance. Consequently, load imbalances can degrade the overall query performance. Load balancing aims to minimize the overall query response time by "fairly" redistributing the processing load among the consumer sites. By fairly is meant that the amount of work to be done on each site must be proportional to the capabilities of the site. In a volatile environment, the capabilities of participating machines or the relative size of the probing partitions may change at runtime. However, modifications to the routing policy so that the partitioning reflects better the current conditions lead to incorrect results, unless

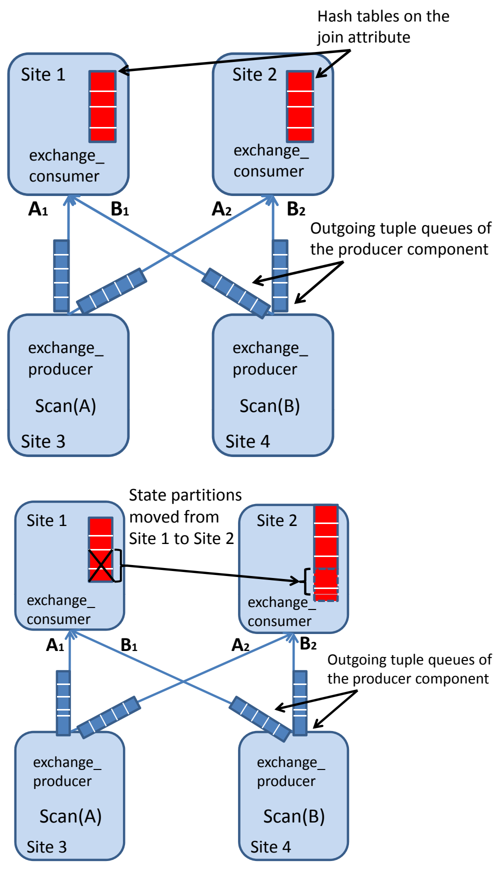

Fig. 9.2 Top: example of executing a hash-join over a shared-nothing cluster using the exchange operator. Bottom: example of state relocation.

these modifications are followed by the relocation of the corresponding buckets in the hash tables. This phenomenon is common to any partitioned operator that builds and maintains internal state during its execution. An example of state relocation is shown in Figure 9.2(bottom), where parts of the hash-table in Site 1 are moved to the hash-table of Site 2 if the routing policy changes at runtime and more tuples are routed to Site 2.

9.5.1.2 The Flux Approach

Flux [59] is an operator that can be deemed as an extension to the proposals of the exchange and *river* [1] mechanisms, accounting for adaptive load balancing of stateful operators, such as windowed equi-joins and group-bys. Two policies are proposed for adaptive load balancing in clusters for settings with ample and limited main memory, respectively. The first policy aims to transfer load (which entails state relocation as well) from an overloaded consumer operator instance to a less loaded consumer operator instance taking into account only the processing speed and idle times of consumers, while the second policy extends the former by considering memory management as well. The goal of the load balancing policy in a cluster with ample main memory is to maximize tuple throughput, through the minimization of utilization imbalances and the number of states moved. On the other hand, the constrained memory load-balancing policy tries to balance memory use across the cluster to avoid or postpone pushing states into disk.

In Flux, instead of having only one state partition per consumer instance, each consumer instance holds multiple "mini"-partitions. This is an effective mechanism for enabling fine-grained balancing [20]. In order to perform load balancing the following functionality is added. First, the consumer components maintain execution statistics tracked over monitoring periods. The maintained statistics differ with respect to the execution environment, i.e., whether the cluster has ample main memory or not. In the first case, the statistics are (i) the number of tuples processed per partition at the consumer side and (ii) the amount of time the consumer operator instance has spent idle, i.e., the amount of time the consumer component, which resides on the corresponding consumer operator instance, waits for input to arrive. From these statistics, the actual utilization of each node is derived. In the second case, i.e., when the aggregate main memory in the cluster is limited, the runtime statistics are (i) the available main memory at each consumer side, and (ii) the size of the partitions, along with indications of whether they are memory-resident or not. Second, the adaptations are coordinated by a global controller. The controller is responsible for collecting the runtime statistics from the consumer components and issuing movement decisions for load balancing.

In both policies mentioned above, load balancing is performed periodically and proceeds in rounds, where each round consists of two phases: a statistics collection phase and a state relocation phase. The duration of the state relocation phase impacts on the length of the next monitoring period with a view to avoiding scenarios where most of the time is spent shifting state partitions around. Also, in order to minimize the number of partition moves, for a given pair of sites, only one state partition is considered for movement, namely the one that reduces the utilization imbalance between the donor site and the receiver, provided that several threshold requirements are met.

The steps that take place when state partitions are relocated from one site to another are roughly the following: quiescing the partition to be moved, transferring the state partitions to the corresponding consumer sites and restarting the partition input stream. During quiescing, the consumer and the producer exchange messages in order to ensure that all in-flight tuples have been processed, and, as such, the consistency of the results is guaranteed. In addition, during the state movement period, each producer component marks the candidate state partition as stalled and buffers the incoming tuples for that state. After the state movement is performed successfully, all buffered tuples are redirected to the consumer operator instances that are selected to be the new hosts and the producer components resume directing tuples.

Table 9.3 Adaptive control in Flux [59].

Measurement: The following statistics are reported periodically from the local consumer components to the central controller: (i) the number of tuples processed per partition at the consumer side, (ii) the amount of time the consumer instance has spent idle, (iii) the available main memory at the consumer side, and (iv) the size of the partitions.
Analysis-Planning: Load balancing is coordinated by a central controller, which detects imbalances based on the measurements received. In case of imbalances, the controller forms pairs of sites that will exchange state partitions based on their utilization (in non memory-constrained clusters) or their excess memory capacity (in memory-constrained clusters) with the help of several thresholds.
Actuation: The steps are the following: (i) stall the input to the state partitions to be relocated, (ii) wait for in-flight tuples to arrive, (iii) transfer the state partitions to the corresponding consumer sites, and (iv) resume processing. The time spent for state movement will be used to define the next monitoring period length.

The policy for load balancing in a memory-constrained environment is similar to the one described above. However, in such an environment, state-movement is guided by the excess memory capacity criterion. The excess memory capacity at a consumer site is defined as the difference between the total memory size of the local states and the total available main memory. Similar to the previous policy, the state partitions selected to be moved are those that reduce the imbalance in excess capacity between pairs of sites. Flux can also perform secondary memory management, in the sense that each consumer site may autonomously decide to push and load state partitions into and from disks, respectively, in a round-robin fashion. The full details of the Flux adaptive load balancing approach can be found in [59]; a summary is presented in Table 9.3. Note that Flux can be complemented with fault tolerance capabilities [60].

9.5.1.3 Improvements on the Flux Approach

Paton *et al.* proposed some modifications to the original Flux operator in [54]. In one of the proposed variants, the execution proceeds as in the original Flux operator but state partitions are replicated instead of simply being moved. This entails higher memory requirements but, at the same time, manages to reduce the number of future state movements. In another variant, which assumes operators building hash tables, each hash-table bucket is randomly assigned to three sites. At hash-table build or probe phase, a tuple is sent to the two most lightly loaded of the three candidate sites that are associated with the bucket that the tuple is hashed to. During the probe phase, if a probe tuple matches a build tuple, the join algorithm generates a result from the probe occurred at the least loaded site, unless the matching (build) tuple is stored only on the other two sites. This variant reduces the adaptation overhead, which is mainly due to state movements, but incurs significant amounts of extra

Table 9.4 Adaptive control in [45].

Measurement: The following statistics are reported periodically from the local sites to the central controller: (i) that available main memory on the site, (ii) the size of partitions and (ii) the corresponding number of tuples that are generated from each partition.
Analysis-Planning: State relocation decisions are triggered by the central controller, when the available main memory imbalances exceed a user-defined threshold, based upon the productivity of each partition and the memory available.
Actuation: Similar to Flux [59] followed by a disk cleanup procedure in order to produce results from the disk resident state partitions.

work. Also, in [54], more advanced mechanisms for the actuation phase are investigated, which aim to cancel planned adaptations when the expected benefit does not outweigh the adaptation overhead.

Liu *et al.* have proposed techniques that deal with load balancing and secondary memory management of partitioned, stateful operators in an integrated manner [45]. As in Flux, the state relocation decisions are guided by a single controller, which periodically collects run-time statistics that are locally monitored at the remote sites during fixed-length time periods and triggers run-time adaptations. The criteria that guide the load balancing process are (i) the available main memory at each site, (ii) the partition sizes and (iii) the number of generated tuples from each partition.

State relocation is triggered by the controller when the available main memory imbalances among the sites exceed a user-defined threshold [45]. In that case, the most productive partitions from the site with the least available memory are moved to the site with the most available memory. Regarding secondary memory management, two different approaches can be followed. One local approach is to push the less productive partitions at each site into disk, when the amount of available memory is less than a user-defined threshold. Another global approach is to find the overall less productive partitions among all the sites and to push them into disk. The steps that take place during state relocation are similar to those in Flux with the addition of a disk cleanup procedure to produce results from the disk resident state partitions. The main characteristics of the integrated approach in [45] are summarized in Table 9.4.

The work in [32] extends Flux by supporting multiway windowed stream joins that are not necessarily equi-joins; moreover it focuses on the combination of load balancing and the so-called diffusion overhead. Load balancing is considered by allocating tuples to the less loaded machines. Diffusion overhead corresponds to tuple replications and intermediate join result transferring, which is needed to ensure correct result generation. Two algorithms are presented, which rely on partial tuple duplication. The first adaptively chooses a master stream, based on which the other streams are transferred, while the second builds upon a greedy solution of the weighted set cover problem [13]. The advantage of both approaches is that the routing is not based on the value of the tuples. Table 9.5 summarizes the main characteristics.

Table 9.5 Adaptive control in [32].

Measurement: The following statistics are reported periodically from the local processing sites and the aggregation site to the controller: (i) the usage and the capacity of the local CPU, memory and bandwidth resources (ii) the results throughput. The controller also monitors (iii) the input data streaming rates.
Analysis-Planning: The controller, apart from dynamically routing input tuples, dynamically selects the master stream and adapts its window segment length.
Actuation: The algorithm provides for special treatment of the intensionally duplicated tuples in order to ensure result correctness.

[66] addresses the same problem as in [32]. In [66], the notion of Pipelined State Partitioning (PSP) is introduced, where the operator states are partitioned into disjoint slices in the time domain, which are subsequently distributed across a cluster. Compared to [32], the approach in [66] does not duplicate any tuples and benefits from pipelined parallelism to a larger extent.

9.5.1.4 Summary

The previous discussion shows that, for the problem of intra-operator load balancing in DQP, several solutions with different functionality have been proposed. These solutions also differ in the trade-off between running overheads (which denote the unnecessary overheads when no adaptations are actually required) and actuation costs, which may accompany the execution of adaptivity decisions. The original Flux proposal is a typical example of an approach with low overhead but potentially high actuation cost, whereas other proposals in [54] mitigate the latter cost at the expense of higher overheads. Regarding the risk of causing performance regression due to costly adaptations, a limitation of the techniques mentioned above is that they do not consider the cost of moving operator state during the planning phase explicitly. The work in [29] fills this gap and revisits the problem of [59] by following a control-theoretical approach, which is capable of incorporating the overhead associated with each adaptation along with the cost of imbalance into the planning phase of the adaptivity loop. Initial results are shown to be promising, when machines experience periodic load variations. This is because the system does not move operator state eagerly, which is proven to be a more efficient approach [28].

9.5.2 Inter-Operator Load Management

While the techniques discussed in the previous section perform load balancing at intra-operator level, the approaches in this section perform load management at inter-operator level. In particular, two representative families of techniques that correspond to different approaches to inter-operator load management are presented. The former, which is exemplified by [70], discusses cooperative load management with a view to achieving load balancing, whereas the latter, exemplified by [9], is concerned with a setting where each node acts both autonomously and selfishly. However, both groups of techniques deal with the problem of dynamically reallocating operators to alternative hosts without performing secondary memory management. Since no secondary memory management is done, the steps that take place during the operator relocation process are the following: (i) stalling the inputs of the operators to be moved and local buffering of the operators' input data, (ii) movement of the operators (along with the data inside their internal states) to their new locations and (iii) restarting of their execution. These steps constitute the typical operator migration procedure.

9.5.2.1 Cooperative Load Management

The work in [70], which is part of the Borealis project, assumes that data streams are processed on several sites, each of which holds some of the operators. Load balancing in [70] is treated as the problem not only of ensuring that the average load (e.g., CPU utilization) of each machine is roughly equal, but also of minimizing the load variance on each site and maximizing the load correlation among the processing sites. The rationale of the approach is described in the following example. Suppose that there exist load measurements of two operators hosted on the same site that have been taken during the last k monitoring periods; these measurements form two time-series. If both time series have a small correlation coefficient ([53]), then, allocating these operators on the same site is a good idea because it means that when one operator is relatively busy, the other is not. By putting these operators on the same site, the load variance of the host is minimized with a view to minimizing end-to-end latency. The average end-to-end latency degrades when highly correlated operators are co-located, since the operators may be simultaneously busy. In addition, if the load time series of two sites have a large correlation coefficient, then, their load levels are naturally balanced. Following the above rationale, the proposal in [70] tries to balance the load of a distributed environment by placing lowly correlated operators (in terms of load) at the same site, while maximizing the load correlation among the processing sites.

Under the proposed load balancing scheme, adaptations are performed periodically. The site and operator loads are locally monitored over fixed-length time periods and are reported to a single controller, which takes load balancing decisions. The load of an operator during a monitoring period is defined as the fraction of the CPU time needed by that operator in order to process incoming tuples (that arrived during that monitoring period) over the length of the monitoring period. The load of a site during this period is defined as the sum of the loads of all its hosted operators. Note that the controller keeps only the site and operator load statistics of the k most recent monitoring periods. Several algorithms are proposed in [70] for load balancing. However, all of them follow the same pattern: given a site pair, they decide which operators to transfer between the sites. The site pairs are decided by

Table 9.6 Adaptive control in Borealis [70].

Measurement: The sites periodically report the site/operator load statistics to the central controller. The controller retains the *k* most recent load statistics for each site/operator.
Analysis-Planning: The variance and the correlation is computed. Operator relocation is triggered periodically by the central controller. The operator redistribution algorithm first detects pairs of sites and then defines the operators to be moved.
Actuation: The re-distribution decisions are enforced through operator migration, which incurs non-negligible cost. This cost is only implicitly taken into account.

the controller based on their average load (the mean value of a site's load time series) following a procedure similar (to an extent) to the one used in Flux[59]. The proposed load balancing algorithms aim to optimize different objectives, i.e., the amount of load (operators) that is moved between a pair of nodes or the quality of the resulting operator mapping. The former technique considers operator migration cost implicitly. A summary of the adaptivity characteristics is given in Table 9.6.

In the context of the same project, a technique for failure recovery, which is equipped with dynamic load balancing characteristics has been proposed [37]. The proposed technique aims to provide low-latency recovery in case of a node's failure using multiple servers for collectively taking over the required actions. In particular the data that is produced and the data that is in the internal states of query fragments is backuped on a selected site. A site's failure is masked by other sites, which host backups and collectively rebuild the latest aggregate state of the failed server. However, new queries that may be submitted for execution or changes in the input streaming rates may change the sites' load and consequently the failure-recovery time. To solve this problem, the proposal in [37] may adaptively relocate the query fragment backups and move them from heavily loaded sites to less loaded ones.

Wang *et al.*, in [67], deal with a problem that is similar to the one in [70]. The distinct feature of this work is that operator placement decisions are based on mechanisms inspired by the physical world. In the physical world, each physical object tries to minimize its energy, whereas its behavior is driven by several types of potentials and the potential energy of an object depends on the location of other objects. In [67], each query operator is considered as a physical object, the potential energy of which reflects its output latency and depends on the site and on the network load conditions. The operator/site load is estimated as in [70], while the network load accounts for the overhead incurred by the network transmissions. As in [70], operator redistribution is performed by a single controller at periodic time intervals. All execution sites monitor the operator/site and network load conditions over fixed-length time periods and send the appropriate statistics to this controller. The controller performs load balancing utilizing heuristics that approximate the optimal solution. In order to minimize the overhead that is incurred during operator movement, only the most loaded operators are considered for redistribution. The fact that network

Table 9.7 Adaptive control in Medusa [9].

Measurement: Each site monitors its load.
Analysis-Planning: The operator relocation process is triggered asynchronously when (i) a site becomes overloaded and (ii) another site (not-overloaded) is willing to accept (part of) its load in exchange of payment. The overloaded sites select a maximal set of operators that are more costly to process locally. Each not overloaded site, in turn, continuously accumulates load offers and periodically accepts subsets of offered operators.
Actuation: Typical operator migration, where operator migration overhead is considered small.

conditions are considered helps in mitigating the risk of performing non-beneficial adaptations, which is more likely to occur in [70].

9.5.2.2 Non-cooperative Load Management

The problem of load management in the Medusa project, which is a predecessor of Borealis, is treated under a different perspective, according to which the distributed systems are regarded as computational economies and the participants provide computational resources and accept to host and execute operators from other participants at a specified price [9]. Another difference with the load balancing techniques that are presented so far is that there is no single controller that decides which operators should be transferred to other hosts. In contrast, the hosts decide independently on the amount of load to transfer or accept.

In Medusa, the hosts aim to select an appropriate operator set in order to maximize the difference between the payment they receive and the cost incurred locally when processing this operator set. They negotiate with other hosts the amount of load to transfer or receive and the corresponding payment through contracts. The operator relocation process is not triggered at predefined time periods, but when, firstly, a site becomes overloaded and, secondly, another (not-overloaded) site is willing to accept at least part of the former's site load in exchange of a payment. To this end, the overloaded sites select a maximal set of operators that they are more costly to process locally than to offload, and offer them to another site. Each site continuously accumulates load offers and may periodically accept subsets of offered operators, on the grounds of higher unit-price offers. As such, the negotiation is asynchronous. If a new site accepts some of the offered operators, operator migration takes place. Note that in [9], a negotiation scheme is also proposed for non fixed-price contracts, due to the implications the fixed-price contracts may lead to. Table 9.7 summarizes the main characteristics of this adaptive load management approach.

9.5.2.3 Summary

This section discussed two different approaches to inter-operator load management. For cooperative scenarios, the solutions presented are interesting but still suffer from significant limitations, such as centralized control mechanism and increased risk to cause performance degradation due to the fact that the adaptation costs are not considered during planning explicitly. On the other hand, in non-cooperative scenarios an interesting alternative is proposed, according to which each node decides autonomously. Although the latter approach is more scalable, an issue that merits further investigation is to assess the messaging overhead in both approaches: in decentralized settings, nodes exchange messages in order to reach decisions as part of a negotiation protocol, whereas in centralized settings, nodes transmit monitoring information.

9.5.3 More Generic Solutions

Intra-operator and inter-operator load management techniques can be combined together. An example appears in [27], which considers partitioned pipelined queries running on distributed hosts. Intra-operator load management is responsible for balancing the load across partitioned operator instances in a way that reflects the runtime machine capacities. Inter-operator load management is responsible for detecting bottlenecks in the pipelines and removing them by increasing the degree of partitioned parallelism of the operators that form the bottlenecks. Another interesting feature of the same work is that operator state is not removed from any machine. Moreover, slow machines, for which the proportion of the workload assigned is decreased, do not participate in building operator state at the new sites. The responsibility for state movement rests with operators upstream in the query plan, which hold copies of data mainly for fault tolerance purposes at the expense of higher memory requirements.

A combination of inter- and intra-load management has been proposed for stream processing systems as well. This can be achieved through load sliding and splitting techniques, respectively [11]. Distributed eddies can be leveraged to behave in a similar way, too. However, understanding the interplay between efficient resource allocation and load balancing is a challenging topic because the goals are often conflicting, as explained also in [62].

9.6 AdQP for Distributed Settings: Other Techniques

In this section, we present techniques that deal with the problem of adaptive query optimization in distributed environments where the issues are not investigated at the operator level. In particular, we discuss proposals for adaptive parallelization of queries and web service (WS) calls (e.g., [64], [57], [68]). Additionally, a few works that propose robust algorithms for distributed query optimization are briefly described (e.g., [34], [22]).

In [64], an adaptive technique is proposed that aims to optimize the execution of range queries in a distributed database. The tables are horizontally partitioned and the resulting partitions are replicated at multiple storage hosts. In order to minimize the query response time, the queries are executed in parallel. The hosts that store the data of interest are firstly identified, and then, the identified hosts process the same query using the local data partitions. However, setting the maximum level of parallelism does not necessarily minimizes the result consumption rate, since, if a query is sent to too many storage hosts, results may be returned faster than the client who submitted the query can consume them. Apart from that, things become more complicated when multiple queries run in parallel and need to access the same storage hosts due to disk contention, which may slow down all queries. To solve this problem, Vigfusson *et al.* have proposed an algorithm that can adaptively (i) determine the optimal parallelism level for a single query and (ii) schedule queries to the storage hosts. In order to find the optimal parallelism level for each query, the algorithm randomly selects to modify the number of hosts that process the query in parallel for a short period. If this change results in an increase in the client consumption rate, then the change is adopted. The algorithm also employs a priority-based approach in order to schedule queries to hosts. The work in [57] deals with a similar problem, where adaptive approaches are explored for parallelizing calls to WSs. Also, in [38], substitution of data sources on the fly is supported to tackle data source failures.

Wu *et al.* proposed an adaptive distributed strategy for approximately answering aggregate queries in P2P networks [68]. In particular, data samples are distributed to sites for further processing. At each processing site, local aggregates are computed that are subsequently sent to a coordinator site, which combines them in order to produce the global aggregate value. The proposed strategy adaptively grows the number of processing nodes as the result accuracy increases with a view to minimizing the query response time.

Han *et al.* have proposed an extension to the initial proposal of progressive optimization in [47] to account for distributed environments [34]. This distributed proposal proceeds similarly to its centralized counterpart. Each plan fragment is marked at special points, where the optimality of the overall plan can be validated. The execution sites monitor the cardinalities of the local intermediate results at these special points and send a positive or negative vote for re-optimization to the controller; if the observed cardinality lies in the validity range, then the vote for re-optimization is negative, otherwise it is positive. The controller employs a voting scheme in order to decide whether re-optimization must be triggered or not. Several voting schemes are proposed in [34]. For example, in the *majority voting* scheme, re-optimization is triggered if at least half of the total execution sites vote for re-optimization. On the other hand, in the *maximum voting* scheme, re-optimization is triggered if at least one site sends a positive vote independently of the votes that the other sites send. Another extension to [47] is presented in [22]; the work in [22] focuses on queries that access data from remote data sources.

Finally, some earlier proposals defer resource allocation decisions until more accurate information about data statistics becomes available (e.g., [33, 71, 50]).

9.7 Conclusion and Open Issues

In this chapter, we investigated the state of the art in distributed adaptive query processing. The main techniques developed so far deal with issues such as extensions to eddies (e.g., [62]), intra-operator load balancing (e.g., [59]), inter-operator load balancing (e.g., [70]) and inter-operator load management with selfish hosts (e.g., [9]). These techniques differ significantly from their centralized counterparts, both in their objectives and in their focus. The objectives in distributed AdQP are more tailored to distributed settings, whereas more attention is paid to issues relating to the adaptivity cost, which is significant, especially when operators and data are moved over the network. Nevertheless, most of the techniques consider the increased adaptivity cost in an implicit heuristic-based manner, with the exception of the work in [29]. Apart from the adaptation costs, the overall performance of AdQP techniques needs to be investigated in a more systematic way, since only very few works are accompanied with theoretical guarantees about their behavior [6].

Other issues that have not been adequately addressed include scalability and the interplay between distinct AdQP techniques. Decentralized control in co-operative settings has been discussed in [62], but it is still an open issue how to apply the same approach in broader scenarios. Moreover, the relationship between load balancing and efficient resource allocation should be further explored. Also, in stream environments, load management may include load shedding techniques as well; it is worth conducting research to better understand the relationship between the AdQP techniques presented in this chapter and load shedding methodologies (e.g., [24]). For example, when the data production rate of a streaming data source increases beyond the capacity of the consuming operator, any technique from the following is applicable: to perform load shedding or to move the consumer to a more powerful node or to increase the degree of intra-operator parallelism of the consumer and subsequently perform load balancing. An interesting research issue is to develop hybrid techniques that combine these different approaches with a view to improving efficiency.

An additional interesting topic for further research is not merely to combine different query processing techniques, but also to combine AdQP with more generic adaptive techniques in distributed settings. For example, the problem of load balancing has also been studied in the area of P2P networks (e.g., [25, 55, 23]); it is not clear how AdQP behaves when applied to an adaptively managed distributed infrastructure. AdQP in distributed settings may also both benefit from and influence techniques in distributed workflow processing (e.g., [44]). Finally, advanced AdQP techniques should be coupled with techniques that mitigate the need for adaptivity, such as robust initial operator allocation (e.g., [69]).

References

1. Arpaci-Dusseau, R.H.: Run-time Adaptation in River. ACM Trans. Comput. Syst. 21(1), 36–86 (2003)
2. Avnur, R., Hellerstein, J.M.: Eddies: Continuously Adaptive Query Processing. SIGMOD Record 29(2), 261–272 (2000)
3. Babcock, B., Chaudhuri, S.: Towards a Robust Query Optimizer: A Principled and Practical Approach. In: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, pp. 119–130 (2005)
4. Babu, S., Bizarro, P.: Adaptive Query Processing in the Looking Glass. In: Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR), pp. 238–249 (2005)
5. Babu, S., Bizarro, P., DeWitt, D.: Proactive Re-Optimization. In: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, pp. 107–118 (2005)
6. Babu, S., Motwani, R., Munagala, K., Nishizawa, I., Widom, J.: Adaptive Ordering of Pipelined Stream Filters. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 407–418. ACM (2004)
7. Babu, S., Munagala, K., Widom, J., Motwani, R.: Adaptive Caching for Continuous Queries. In: ICDE, pp. 118–129 (2005)
8. Babu, S., Widom, J.: Continuous Queries over Data Streams. SIGMOD Record 30(3), 109–120 (2001)
9. Balazinska, M., Balakrishnan, H., Stonebraker, M.: Contract-based Load Management in Federated Distributed Systems. In: Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation (NSDI), pp. 15–28 (2004)
10. Bizarro, P., Babu, S., DeWitt, D., Widom, J.: Content-based Routing: Different Plans for Different Data. In: Proceedings of the 31st International Conference on Very Large Data Bases (VLDB), pp. 757–768 (2005)
11. Cherniack, M., Balakrishnan, H., Balazinska, M., Carney, D., Çetintemel, U., Xing, Y., Zdonik, S.B.: Scalable Distributed Stream Processing. In: CIDR (2003)
12. Chu, F.C., Halpern, J.Y., Gehrke, J.: Least Expected Cost Query Optimization: What Can We Expect? In: Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, pp. 293–302. ACM (2002)
13. Chvatal, V.: A Greedy Heuristic for the Set-Covering Problem. Mathematics of Operations Research 4(3), 233–235 (1979)
14. Claypool, K.T., Claypool, M.: Teddies: Trained Eddies for Reactive Stream Processing. In: Haritsa, J.R., Kotagiri, R., Pudi, V. (eds.) DASFAA 2008. LNCS, vol. 4947, pp. 220–234. Springer, Heidelberg (2008)
15. Deshpande, A.: An Initial Study of Overheads of Eddies. SIGMOD Record 33(1), 44–49 (2004)
16. Deshpande, A., Hellerstein, J.M.: Lifting the Burden of History from Adaptive Query Processing. In: Proceedings of the 30th International Conference on Very Large Data Bases (VLDB), pp. 948–959 (2004)
17. Deshpande, A., Hellerstein, L.: Flow Algorithms for Parallel Query Optimization. In: ICDE, pp. 754–763 (2008)
18. Deshpande, A., Ives, Z., Raman, V.: Adaptive Query Processing. Foundations and Trends in Databases 1(1), 1–140 (2007)
19. DeWitt, D., Gray, J.: Parallel Database Systems: The Future of High Performance Database Systems. Communications of the ACM 35(6), 85–98 (1992)

20. DeWitt, D.J., Naughton, J.F., Schneider, D.A., Seshadri, S.: Practical Skew Handling in Parallel Joins. In: Proceedings of the 18th International Conference on Very Large Data Bases (VLDB), pp. 27–40 (1992)
21. Eurviriyanukul, K., Paton, N.W., Fernandes, A.A.A., Lynden, S.J.: Adaptive Join Processing in Pipelined Plans. In: EDBT, pp. 183–194 (2010)
22. Ewen, S., Kache, H., Markl, V., Raman, V.: Progressive Query Optimization for Federated Queries. In: Ioannidis, Y., Scholl, M.H., Schmidt, J.W., Matthes, F., Hatzopoulos, M., Böhm, K., Kemper, A., Grust, T., Böhm, C. (eds.) EDBT 2006. LNCS, vol. 3896, pp. 847–864. Springer, Heidelberg (2006)
23. Gedik, B., Liu, L.: PeerCQ: A Decentralized and Self-Configuring Peer-to-Peer Information Monitoring System. In: ICDCS, pp. 490–499 (2003)
24. Gedik, B., Wu, K.L., Yu, P.S., Liu, L.: GrubJoin: An Adaptive, Multi-Way, Windowed Stream Join with Time Correlation-Aware CPU Load Shedding. IEEE Trans. Knowl. Data Eng. 19(10), 1363–1380 (2007)
25. Godfrey, B., Lakshminarayanan, K., Surana, S., Karp, R., Stoica, I.: Load Balancing in Dynamic Structured P2P Systems. In: Proceedings of the 23rd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM), pp. 2253–2262 (2004)
26. Gounaris, A., Paton, N.W., Fernandes, A.A.A., Sakellariou, R.: Adaptive Query Processing: A Survey. In: Eaglestone, B., North, S., Poulovassilis, A. (eds.) BNCOD 2002. LNCS, vol. 2405, pp. 11–25. Springer, Heidelberg (2002)
27. Gounaris, A., Smith, J., Paton, N.W., Sakellariou, R., Fernandes, A.A., Watson, P.: Adaptive Workload Allocation in Query Processing in Autonomous Heterogeneous Environments. Distrib. Parallel Databases 25(3), 125–164 (2009)
28. Gounaris, A., Yfoulis, C.A., Paton, N.W.: Efficient Load Balancing in Partitioned Queries Under Random Perturbations. ACM Transactions on Autonomous and Adaptive Systems (to appear)
29. Gounaris, A., Yfoulis, C.A., Paton, N.W.: An Efficient Load Balancing LQR Controller in Parallel Databases Queries Under Random Perturbations. In: 3rd IEEE Multiconference on Systems and Control, MSC 2009 (2009)
30. Graefe, G.: Encapsulation of Parallelism in the Volcano Query Processing System. In: Garcia-Molina, H., Jagadish, H.V. (eds.) Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, pp. 102–111 (1990)
31. Graefe, G.: Query Evaluation Techniques for Large Databases. ACM Comput. Surv. 25(2), 73–170 (1993)
32. Gu, X., Yu, P., Wang, H.: Adaptive Load Diffusion for Multiway Windowed Stream Joins. In: Proceedings of the 23rd IEEE International Conference on Data Engineering (ICDE), pp. 146–155 (2007)
33. Hameurlain, A., Morvan, F.: CPU and Incremental Memory Allocation in Dynamic Parallelization of SQL Queries. Parallel Computing 28(4), 525–556 (2002)
34. Han, W.S., Ng, J., Markl, V., Kache, H., Kandil, M.: Progressive Optimization in a Shared-Nothing Parallel Database. In: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD), pp. 809–820 (2007)
35. Hellerstein, J.M., Franklin, M.J., Chandrasekaran, S., Deshpande, A., Hildrum, K., Madden, S., Raman, V., Shah, M.A.: Adaptive Query Processing: Technology in Evolution. IEEE Data Eng. Bull. 23(2), 7–18 (2000)
36. Huebsch, R., Jeffery, S.R.: FREddies: DHT-Based Adaptive Query Processing via FedeRated Eddies. Technical Report No. UCB/CSD-4-1339, University of California (2004)
37. Hwang, J.H., Xing, Y., Cetintemel, U., Zdonik, S.: A Cooperative, Self-Configuring High-Availability Solution for Stream Processing. In: Proceedings of the 23rd IEEE International Conference on Data Engineering (ICDE), pp. 176–185 (2007)

38. Ives, Z.: Efficient Query Processing for Data Integration. Ph.D. thesis. University of Washington (2002)
39. Ives, Z.G., Halevy, A.Y., Weld, D.S.: Adapting to Source Properties in Processing Data Integration Queries. In: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, pp. 395–406 (2004)
40. Kabra, N., DeWitt, D.J.: Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. In: SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, pp. 106–117. ACM Press (1998)
41. Kephart, J.O., Chess, D.M.: The Vision of Autonomic Computing. IEEE Computer 36(1), 41–50 (2003)
42. Kossmann, D.: The State of the Art in Distributed Query Processing. ACM Computing Surveys (CSUR) 32(4), 422–469 (2000)
43. Kossmann, D., Stocker, K.: Iterative Dynamic Programming: a New Class of Query Optimization Algorithms. ACM Trans. Database Syst. 25(1), 43–82 (2000)
44. Lee, K., Paton, N.W., Sakellariou, R., Deelman, E., Fernandes, A.A.A., Mehta, G.: Adaptive Workflow Processing and Execution in Pegasus. Concurrency and Computation: Practice and Experience 21(16), 1965–1981 (2009)
45. Liu, B., Jbantova, M., Rundensteiner, E.A.: Optimizing State-Intensive Non-Blocking Queries Using Run-time Adaptation. In: Proceedings of the 2007 IEEE 23rd International Conference on Data Engineering Workshop (ICDEW), pp. 614–623 (2007)
46. Mackert, L.F., Lohman, G.M.: R* Optimizer Validation and Performance Evaluation for Distributed Queries. In: VLDB 1986 Twelfth International Conference on Very Large Data Bases, pp. 149–159 (1986)
47. Markl, V., Raman, V., Simmen, D., Lohman, G., Pirahesh, H., Cilimdzic, M.: Robust Query Processing through Progressive Optimization. In: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, pp. 659–670 (2004)
48. Nehme, R.V., Rundensteiner, E.A., Bertino, E.: Self-Tuning Query Mesh for Adaptive Multi-Route Query Processing. In: Proceedings of the 12th International Conference on Extending Database Technology (EDBT), pp. 803–814 (2009)
49. Nehme, R.V., Works, K.E., Rundensteiner, E.A., Bertino, E.: Query Mesh: Multi-Route Query Processing Technology. Proceedings of the VLDB Endowment 2(2) (2009)
50. Ozcan, F., Nural, S., Koksal, P., Evrendilek, C., Dogac, A.: Dynamic Query Optimization in Multidatabases. IEEE Data Eng. Bull. 20(3), 38–45 (1997)
51. Ozsu, M., Valduriez, P. (eds.): Principles of Distributed Database Systems, 2nd edn. Prentice-Hall (1999)
52. Pang, H., Carey, M.J., Livny, M.: Memory-Adaptive External Sorting. In: 19th International Conference on Very Large Data Bases, pp. 618–629 (1993)
53. Papoulis, A.: Probability, Random Variables, and Stochastic Processes, 3rd edn.
54. Paton, N.W., Buenabad-Chavez, J., Chen, M., Raman, V., Swart, G., Narang, I., Yellin, D.M., Fernandes, A.A.: Autonomic Query Parallelization using Non-Dedicated Computers: an Evaluation of Adaptivity Options. The VLDB Journal 18(1), 119–140 (2009)
55. Pitoura, T., Ntarmos, N., Triantafillou, P.: Replication, Load Balancing and Efficient Range Query Processing in dHTs. In: Ioannidis, Y., Scholl, M.H., Schmidt, J.W., Matthes, F., Hatzopoulos, M., Böhm, K., Kemper, A., Grust, T., Böhm, C. (eds.) EDBT 2006. LNCS, vol. 3896, pp. 131–148. Springer, Heidelberg (2006)
56. Raman, V., Deshpande, A., Hellerstein, J.M.: Using State Modules for Adaptive Query Processing. In: Proceedings of the IEEE 19th International Conference on Data Engineering (ICDE), pp. 353–364 (2003)
57. Sabesan, M., Risch, T.: Adaptive Parallelization of Queries over Dependent Web Service Calls. In: Proceedings of the 25th IEEE International Conference on Data Engineering (ICDE), pp. 1725–1732 (2009)

58. Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., Price, T.G.: Access Path Selection in a Relational Database Management System. In: Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data. ACM (1979)
59. Shah, M., Hellerstein, J., Chandrasekaran, S., Franklin, M.: Flux: An Adaptive Partitioning Operator for Continuous Query Systems. In: Proceedings of the IEEE 19th International Conference on Data Engineering (ICDE), pp. 25–36 (2003)
60. Shah, M.A., Hellerstein, J.M., Brewer, E.A.: Highly-Available, Fault-Tolerant, Parallel Dataflows. In: Weikum, G., König, A.C., Deßloch, S. (eds.) Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, pp. 827–838. ACM (2004)
61. Stillger, M., Lohman, G.M., Markl, V., Kandil, M.: LEO - DB2's LEarning Optimizer. In: VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, pp. 19–28 (2001)
62. Tian, F., DeWitt, D.J.: Tuple Routing Strategies for Distributed Eddies. In: Proceedings of the 29th International Conference on Very Large Data Bases (VLDB), pp. 333–344 (2003)
63. Urhan, T., Franklin, M.J.: XJoin: A Reactively-Scheduled Pipelined Join Operator. IEEE Data Engineering Bulletin 23(2), 27–33 (2000)
64. Vigfusson, Y., Silberstein, A., Cooper, B.F., Fonseca, R.: Adaptively Parallelizing Distributed Range Queries. Proceedings of the VLDB Endowment 2(1), 682–693 (2009)
65. Viglas, S.D., Naughton, J.F., Burger, J.: Maximizing the Output Rate of Multi-Way Join Queries over Streaming Information Sources. In: Proceedings of the 29th International Conference on Very Large Data Bases (VLDB), pp. 285–296 (2003)
66. Wang, S., Rundensteiner, E.: Scalable Stream Join Processing with Expensive Predicates: Workload Distribution and Adaptation by Time-Slicing. In: Proceedings of the 12th International Conference on Extending Database Technology (EDBT), pp. 299–310 (2009)
67. Wang, W., Sharaf, M.A., Guo, S., Özsu, M.T.: Potential-driven Load Distribution for Distributed Data Stream Processing. In: Proceedings of the 2nd International Workshop on Scalable Stream Processing System (SSPS), pp. 13–22 (2008)
68. Wu, S., Jiang, S., Ooi, B.C., Tan, K.L.: Distributed Online Aggregations. Proceedings of the VLDB Endowment 2(1), 443–454 (2009)
69. Xing, Y., Hwang, J.H., Çetintemel, U., Zdonik, S.: Providing Resiliency to Load Variations in Distributed Stream Processing. In: Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB), pp. 775–786 (2006)
70. Xing, Y., Zdonik, S., Hwang, J.H.: Dynamic Load Distribution in the Borealis Stream Processor. In: Proceedings of the 21st International Conference on Data Engineering (ICDE), pp. 791–802 (2005)
71. Yu, M.J., Sheu, P.C.Y.: Adaptive Join Algorithms in Dynamic Distributed Databases. Distributed and Parallel Databases 5(1), 5–30 (1997)
72. Zhou, Y., Ooi, B.C., Tan, K.L.: Dynamic Load Management for Distributed Continuous Query Systems. In: Proceedings of the 21st International Conference on Data Engineering (ICDE), pp. 322–323 (2005)
73. Zhu, Y., Rundensteiner, E.A., Heineman, G.T.: Dynamic Plan Migration for Continuous Queries over Data Streams. In: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, pp. 431–442 (2004)