18th IEEE International Conference on Control Applications
Part of 2009 IEEE Multi-conference on Systems and Control
Saint Petersburg, Russia, July 8-10, 2009

An Efficient Load Balancing LQR controller in Parallel Database Queries under Random Perturbations

Anastasios Gounaris, Christos A. Yfoulis and Norman W. Paton

Abstract— **This work investigates the problem of dynamic, intra-query load balancing in parallel database queries across heterogeneous nodes in a way that takes into account the inherent cost of adaptations and thus avoids both over-reacting and deciding when to adapt in a completely heuristic manner. The latter may lead to serious performance degradation in several cases, such as periodic and random imbalances. We follow a control theoretical approach to this problem; more specifically, we propose a multiple-input multiple-output feedback linear quadratic regulation (LQR) controller, which captures the tradeoff between reaching a balanced state and the cost inherent in such adaptations. Our approach, apart from benefitting from and being characterized by a solid theoretical foundation, exhibits better performance than state-of-the-art heuristics in realistic situations, as verified by thorough evaluation.**

I. Introduction

Parallel processing of a single query, either over static databases or data streams, involves splitting a query graph into several subgraphs so that these graphs can run on different machines, e.g., in a *pipelined* fashion when one subgraph feeds data to another subgraph. Moreover, each of these subgraphs may be able to be instantiated several times, with each of the resulting instances operating on a different subset of data (or data *partition*).

However, to fully exploit the potential of parallelism, work needs to be assigned to machines in a way that reflects their capabilities, which is challenging in an environment with heterogeneous and potentially autonomous, non-dedicated resources. Key characteristics of a typical such environment include the following:

- There exist unpredictable fluctuations in the load of available machines. A consequence of this fact is that it is not efficient to divide a task into several partitions and to stick with this division until completion.
- The information about the machine characteristics that describe performance capacity and loads is typically incomplete and/or inaccurate at compile time; thus a load balancer with responsibility for efficient work assignment should rely mostly on runtime feedback.
- The instances of subgraphs are stateful. A consequence of this fact is that the cost of workload re-assignments, which typically depends on the size of the state, is not negligible in general.

Anastasios Gounaris is with the Computer Science Department, Aristotle University of Thessaloniki, Thessaloniki, Greece. E-mail: gounaria@csd.auth.gr

Christos A. Yfoulis is with the Automation department, ATEI of Thessaloniki, Thessaloniki, Greece. E-mail: cyfoulis@teithe.gr

Norman W. Paton is with the School of Computer Science, University of Manchester, Manchester, UK. E-mail: npaton@manchester.ac.uk

Several authors have tried to address the challenges that result from the aforementioned characteristics. One of the most notable examples is the Flux approach [1], which introduces a new operator that monitors the execution speed and the idle time of each participating machine at runtime, and adjusts the workload allocation accordingly, with a view to equalizing machine utilization. Additional heuristics are applied to smooth the workload allocation changes. In the Flux operator, state typically consists of several state partitions defined by a hash function, and each node is allowed to either transmit or receive a single state partition during the same balancing step so that over-reacting is avoided. In addition, Flux tries to guarantee that the time spent enforcing adaptivity decisions (i.e., moving state from one machine to another as a result of a workload reallocation) does not exceed the time of query processing; this is done by keeping the same workload allocation for a period that is at least equal to the time spent carrying out the adaptation that brought it about. Note that this does not guarantee that the overall time will be reduced by adaptive balancing; in other words, execution is not improved in all cases. Indeed, Flux, apart from being rather sensitive to its parameters, such as the size of the window used for collecting execution statistics, may adapt in a non-beneficial manner in response to transient and periodic imbalances, as it may keep shifting the state partitions.

Several attempts have been made to improve the behavior in these situations. In [2], some extensions to the Flux approach are described. An enhancement of the Flux decision making criterion, which is investigated in [2], enacts adaptations only when the accumulated delay due to the use of the current workload allocation strategy is greater than the cost of changing to the strategy that would have been best over some period. This improves slightly on the snapshot-based original Flux in unstable environments, and, in essence, behaves like an integral controller, in that it adapts in a way that takes account of the average error over a period. However, the efficiency relies heavily on the window size chosen.

Typically, the load balancing problem in a single query environment is transformed to the problem of making the execution times of parallel subtasks equal, as their maximum defines the overall execution time. The spirit of Flux is the same, although the adaptivity steps are not based on a corresponding balancing function, but on a heuristic, as mentioned previously. Such approaches essentially adopt a definition of

balanced execution, which does not take into account the inherent overhead for enforcing the balancing decisions. This limitation, which is particularly felt in unpredictably volatile imbalances, is addressed in this work.

The approach we follow is founded on applied control theory; the problem under investigation falls into the broader vision of developing autonomic, self-managing solutions for data management [3]. In principle, autonomic computing can benefit a lot from control theory techniques, which are well-established in engineering fields and are typically accompanied by theoretical investigations of properties such as stability, accuracy, and settling time [4]. Nevertheless, their application to computing systems is rendered problematic because of issues such as effective modeling of the system and its dynamics, and overhead times in enforcing adaptations [5]. Control theoretical solutions with a view to achieving self-managing behavior have been incorporated into commercial systems [6], although it is acknowledged that factors such as volatile loads and the difficulty in constructing realistic models that also capture the cost of adaptations, are prohibitive for the application of control theory to database systems.

To address the problem of balancing the load of a partitioned query across multiple heterogeneous machines, we design an adaptive multiple-input, multiple-output (MIMO), discrete-time, feedback linear quadratic regulation (LQR) controller. In general, LQR controllers can encapsulate the cost to enforce a response (i.e., the cost to move state from one machine to another) along with the cost of deviations from the ideal state in a unified cost function. As such, they seem promising. To the best of the authors' knowledge, there is a single example of a design for an LQR controller for database systems, namely to adjust the memory pool sizes [9] in a way that is more tolerant to noise than other optimization techniques [10]. There is no prior control theoretical work that deals with balancing the execution of partitioned query tasks in volatile settings. In a different setting, cost-aware load balancing has been investigated in [7]. In this work, the existence of a detailed mathematical model of the system is assumed, and the main contribution is, when deciding on the workload distribution, to take into consideration the number of in-transit tasks due to previous adaptivity actions. In our environment, all data transfers are completed before resuming query execution. Finally, control theory has recently been employed to optimize data transmission from service-based databases [8].

The contributions of this work are summarized as follows: (i) it introduces an LQR control theoretical approach to load balancing in (stateful) parallel database queries, which is inherently suitable for adaptations that incur some cost; (ii) it presents a detailed methodology as to how such a controller mechanism can be configured; (iii) it shows that the resulting mechanism is stable, effective and capable of reaching a balanced state in short times; and (iv) it compares its performance against state-of-the art load balancing proposals.

The remainder of this article is structured as follows. Section II describes the load balancing problem formally. The presentation of our LQR controller is in Section III, whereas in Section IV we investigate the efficiency and effectiveness of the approach. Section V concludes the article.

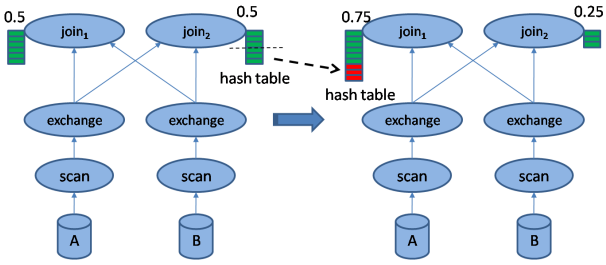

Fig. 1. An example of balancing stateful operators.

II. Description of the Load Balancing Problem

Consider two relations, *A* and *B*, which are joined remotely using a hash join; the hash table is built on *A*. Let us assume that the hash join operator is parallelised over two physical nodes, and that these two nodes are capable of processing tuples at the same speed. Then, in a balanced execution, the two nodes should receive and process the same amount of workload. However, if, during execution, the first machine becomes three times as fast as the second machine, then the workload distribution should change to reflect that. The problem is that a workload distribution that is proportional to the nodes' execution speed can yield the lowest response times only if the operators are stateless. In stateful operators, like hash joins, which create internal state in the form of hash tables, any workload re-allocation triggers state movements, which incur some cost (see Fig. 1). Consequently, a more efficient load balancer should take into account this cost when deciding on workload re-allocations with a view to reducing the query execution time.

The load balancing problem can be formalized as follows. Let P be the degree of intra-operator parallelism of an operator o, and $m_1, m_2, \cdots, m_P$ the P nodes participating in its execution. The workload proportion that each of these nodes receives at the kth adaptivity step is $p_1(k), p_2(k), \cdots, p_P(k)$, with the constraints $\sum_{i=1}^{P} p_i(k) = 1, \forall k$ and $p_i(k) \geq 0, \forall k$. $p_i(k), i = 1 \ldots P$ is defined through a hash function $h_k()$. Each node possesses a certain amount of state $s_i(k), i = 1 \ldots P$, which is needed to evaluate $p_i(k)$. $s_i(k)$ depends on $p_i(k)$. $c_i(k)$ denotes the cost (overhead) to reach state $s_i(k)$ from state $s_i(k-1)$, as a result of a change in $p_i(k)$. The measured output is $y_1(k), y_2(k), \ldots, y_P(k)$ and defines the expected value for the completion time of each of the participating nodes given the workload allocation of the kth adaptivity step. If the query is continuous (e.g., over data streams) the expected completion time refers to the time to complete the evaluation of a fixed aggregate workload. Without loss of generality, we can assume that $y(i)$ strictly monotonically increases with $p(i)$, i.e., assigning more workload to a node leads to an increase in its expected completion time. The role of the load balancer is to minimize the following

$$max(y_i(k+1) + c_i(k+1)), \quad i = 1 \ldots P \qquad (1)$$

and estimate h_{k+1} accordingly.

It can be proved that the workload allocation is always optimal if no further workload re-allocation is needed, i.e., $y_1(k) = y_2(k) = \ldots = y_P(k)$. This condition can be also written as $y_i(k) = \frac{1}{P}\sum_{i=1}^{P} y_i(k)$. To the best of our knowledge, there is no practical methodology to solve Eq. (1) analytically.

III. Design of the LQR controller

Essentially, the load balancing objective defined in Eq. (1) includes a trade-off between (a) reaching the optimal workload allocation, in which the expected completion times are equalized across all participating nodes, and (b) the cost for reaching such an allocation, which is mainly due to state movements. To meet such an objective, we employ a state space model, on top of which we implement a state feedback controller, which is designed with the help of a linear quadratic regulator (LQR). Our LQR controller is capable of accurately finding the controller settings that minimize a cost function, which can capture both the deviations from the optimal state and the cost to reach such a state. In essence, we do not try to postpone adaptations due to the cost they are expected to incur but to modify the response actions so that any adaptations applied are beneficial.

a) The state-space model of the system: In our setting, the output vector $\mathbf{y}$ is a $P \times 1$ vector of the measured output values (i.e. expected completion times for each node) and the input vector $\mathbf{u}(k)$ is a $P \times 1$ vector of inputs, i.e. our manipulated variables which are the workload allocations at the kth step. P is the number of participating nodes mentioned in the previous section.

According to the load balancing requirement, all outputs y_j , $j = 1, \ldots, P$ are equalized to their optimal value, which is their average $\overline{y}(k) = \frac{1}{P}\sum_{i=1}^{P} y_i(k)$. Hence we have to design a tracking controller so that the outputs follow a time-varying reference trajectory $\overline{y}(k)$. Therefore, instead of having a static value or an external signal as the reference input, the reference is specified as a linear combination of measured outputs, i.e. their average. In order to use the LQR regulation controller, this tracking requirement is typically transformed to a regulation problem by considering as state-variables the control errors $e_j = y_j - \overline{y}$. The control error vector is then given by

$$\mathbf{e}(k) = \begin{bmatrix} y_1 \\ y_2 \\ \cdots \\ y_P \end{bmatrix} - \frac{1}{P}\begin{bmatrix} \sum_{i=1}^{P} y_i \\ \sum_{i=1}^{P} y_i \\ \cdots \\ \sum_{i=1}^{P} y_i \end{bmatrix} = (\mathbf{I_{P,P}} - \frac{1}{P}\mathbf{1_{P,P}})\mathbf{y}(k)$$

where $\mathbf{I_{P,P}}$ and $\mathbf{1_{P,P}}$ are the $P \times P$ identity and unary matrices, respectively.

In our state-space model, inspired by the work in [9], we adopt a *dynamic state feedback* strategy, which is the state-space analog of a PI (proportional integral) controller, that allows tracking of a time-varying reference input and has disturbance rejection properties, due to the presence of an integrator. These properties are absolutely essential in our problem, due to the unpredictability of machine load and the time-varying reference input imposed by the balancing requirement.

The corresponding augmented state vector is $\mathbf{x}(k) = [\mathbf{e}(k) \;\; \mathbf{e_I}(k)]^T$, where the error and the accumulated error are given by

$$\mathbf{e}(k) = (\mathbf{I_{P,P}} - \frac{1}{P}\mathbf{1_{P,P}})\,(\mathbf{y}(k) + \mathbf{d}^m(k)) \tag{2}$$

$$\mathbf{e_I}(k+1) = \mathbf{e_I}(k) + \mathbf{e}(k) \tag{3}$$

The general form of the output equation for the state-space model is as follows.

$$\mathbf{y}(k+1) = \mathbf{A}(k)\,\mathbf{y}(k) + \mathbf{B}(k)\,(\mathbf{u}(k) + \mathbf{d}^c(k)) \tag{4}$$

The vectors $\mathbf{d}^m, \mathbf{d}^c$ in (2),(4), correspond to the *measurement disturbance* and the *control disturbance*. They can accommodate load and reference input changes, measurement noise as well as modeling inaccuracies.

The matrices $\mathbf{A}$ and $\mathbf{B}$ can be found at runtime through system identification or monitoring. The elements of matrix $\mathbf{B}$ correspond to the time units required for each of the participating machines to process a unit of workload, which is the inverse of the processing speed of the machines, and, as such, can capture both changes in the computational capacity, e.g., due to load change, and data skews. Since the expected completion time of one machine does not depend on the processing speed of another, we may assume that $\mathbf{B}$ is diagonal.

Furthermore, the load balancing problem imposes two types of constraints on the control inputs, i.e. $u_i \geq 0$ and $\sum_{i=1}^{P} u_i = 1$, which we have to incorporate into the proposed state-space model. The equality constraint can be satisfied by allowing the controller to specify the allocation only for $P-1$ nodes. For the last machine the allocation will be $u_P(k) = 1 - \sum_{j=1}^{P-1} u_j(k)$. The bound constraints $0 \leq u_i \leq 1$ may be satisfied by careful selection of the LQR parameters, and then it can be also shown that the system is fully controllable. Moreover, the dynamic state feedback strategy adopted assures zero steady state errors and disturbance rejection properties due to the addition of integral action. This holds not only for the first $P-1$ states which are directly controlled, but also for the last P-th state. More details are given in [11].

b) LQR specification: The control input vector $\mathbf{x}(k)$ is

$$\mathbf{u}(k) = -\mathbf{K}\,\mathbf{x}(k)\;,\;\; \mathbf{x}(k) = [\mathbf{e}(k) \;\; \mathbf{e_I}(k)]^T \tag{5}$$

The LQR framework is responsible for the specification of $\mathbf{K}$, and more importantly, for ensuring that $\mathbf{K}$ efficiently implements the tradeoff between quick convergence to the optimal workload allocation and the penalty for this convergence. The cost function to be minimized is $\mathcal{J} = \frac{1}{2}\sum_{k=1}^{\infty}\left[\mathbf{x^T}(k)\,\mathbf{Q}\,\mathbf{x}(k) + \mathbf{u^T}(k)\,\mathbf{R}\,\mathbf{u}(k)\right]$. The dimensions of the $\mathbf{Q}$ and $\mathbf{R}$ matrices are $2N \times 2N$ and $N \times N$, respectively, where $N = P - 1$.

Based on such a cost function, the requirement for quick convergence is quantified by the square of state variables

multiplied by the weights in the $\mathbf{Q}$ matrix. The convergence penalty is quantified by the square of the control input multiplied by the weights in the $\mathbf{R}$ matrix.

The problem of developing a load balancer that considers the overhead of its decisions, is transformed to the problem of defining the (potentially time-varying) $\mathbf{Q}$ and $\mathbf{R}$ matrices. The former captures the requirement for quick convergence to the optimal state, whereas the latter captures the overhead of such a convergence, in line with the objective of (1).

In order to design an LQR controller we derive the closed loop difference equation

$$\mathbf{x}(k+1) = \tilde{\mathbf{A}}\,\mathbf{x}(k) \;+\; \tilde{\mathbf{B}}\,\mathbf{u}(k) \tag{6}$$

where $\tilde{\mathbf{A}}$ is a $2N \times 2N$ matrix, and $\tilde{\mathbf{B}}$ is a $2N \times N$ one.

From the combination of (2) and (4) , we can derive that

$$\mathbf{e}(k+1) \quad \cong \quad (\mathbf{I}_{N,N} - \frac{1}{P}\mathbf{1}_{N,N})\,\tilde{\mathbf{B}}(k)\,\mathbf{u}(k)$$

Note also that (3) can be rewritten as

$$\mathbf{e_I}(k+1) = \left[\begin{array}{cc} \mathbf{I}_{N,N} & \mathbf{I}_{N,N} \end{array}\right] \left[\begin{array}{c} \mathbf{e}(k) \\ \mathbf{e_I}(k) \end{array}\right] \tag{7}$$

Inserting the two formulas above in (6), we get

$$\tilde{\mathbf{A}} = \left[\begin{array}{cc} \mathbf{0}_{N,N} & \mathbf{0}_{N,N} \\ \mathbf{I}_{N,N} & \mathbf{I}_{N,N} \end{array}\right], \text{ and} \tag{8}$$

$$\tilde{\mathbf{B}} = \left[\begin{array}{c} \tilde{\mathbf{I}}_{N,N}\,\mathbf{B}_{N,N} \\ \mathbf{0}_{N,N} \end{array}\right], \quad \tilde{\mathbf{I}}_{N,N} = (\mathbf{I}_{N,N} - \frac{1}{P}\mathbf{1}_{N,N}) \tag{9}$$

We next define the weights for the price to pay when some nodes are expected to complete execution later than others, i.e., when they exhibit non-zero error. Also, we specify the weights for the price to pay on the grounds of the accumulated error for some nodes, which can capture periodic phenomena. These weights are stored in the $\mathbf{Q}$ matrix. For simplicity, we make the assumption that the errors are independent of each other, and in this case $\mathbf{Q} = \left[\begin{array}{cc} a\,\mathbf{I}_{N,N} & \mathbf{0}_{N,N} \\ \mathbf{0}_{N,N} & b\,\mathbf{I}_{N,N} \end{array}\right]$, $a, b \geq 0$. The ratio of a and b defines the relative significance of the error and the accumulated error. When $a/b = 1$, then they are equally taken into account.

Finally, we specify the (normalized) weights for the price to pay to enforce a response. This price depends on the size of the state to be transferred. The weights of this step are stored in the $\mathbf{R}$ matrix, and, as previously, we can assume that they are independent of each other, which entails that this matrix is diagonal, too.

Let us assume that the size of state S stored in all machines is known from before and does not change during execution. In addition, we make the assumption that the average bandwidth bw of the network connections from each machine to the others is known, and the state to be transferred is proportional to the workload allocation.

Based on these assumptions, and provided that the first part of the LQR cost function is the square of time units, the diagonal elements of $\mathbf{R}, r_i, i = 1 \ldots N$, are of a scalar value c that is proportional to the square of the division of S by bw. The value of c is a tuning parameter; the higher its value, the more a change in the workload allocation is penalized.

It should be noted that our controller has *adaptive* features, in that a time-varying matrix $\mathbf{B}(k)$ is used in (9), which is updated in every step, and a new LQR optimization problem is solved to specify new gains $\mathbf{K}(k)$ in (5).

An interesting extension of the adaptive properties of our scheme would be the transition to a fully dynamically configured controller, where the weighting matrices of the cost function become time-varying $\mathbf{R}(k), \mathbf{Q}(k)$, and modified at each adaptivity step. E.g. a dynamically updated matrix $\mathbf{R}(k)$ could capture the accurate cost of transferring the state which reflects the current conditions. This could be done through runtime analysis and the design of a switching controller. However, this is a highly non-trivial task which requires stability guarantees, rigorous analysis, and careful consideration of the associated increased overhead, and is left for future work.

IV. Evaluation

This section compares the performance of the LQR controller described in Section III with heuristic control, based on Flux [1] and on techniques described in [2], using simulations of query performance under time-varying imbalance conditions. In this paper, due to space limitations, we present only two representative experiments. Full details may be found in [11]. Before discussing the experiments, useful tips as to how to configure the LQR controller are presented.

Simulation Model: The simulation uses a cost model consisting of a collection of parameterized cost functions for query operators. The simulator builds upon that used in [2]. Its top level loop, asks the root operator of a plan for the number of tuples it can return in the available time, taking into account the load on the machine to which the operator is allocated. The number returned is determined by how rapidly the operator itself can process data and the rate at which its children can supply data.

The load of each machine, both due to external jobs and query processing tasks, is used to estimate the machine capacity, so that $\tilde{\mathbf{B}}$ is produced at each step (or iteration). More specifically, the computational capacity of each machine is inversely proportional to machine contention, in line with the approach described in [2]. The example operator to be balanced throughout the experiments is a parallel hash join between tables of the TPC-H (http://www.tpc.org) benchmark database with scale factor 1. However, the load balancer presented in this work is independent of any particular operator implementation. The performance metric is the overall query response time, which captures the impact of imbalanced execution that is experienced by the user.

We consider two types of random external job arrival, namely *poisson*, where the rate of job arrival to (some of) the machines evaluating the join in each iteration follows a poisson distribution with a fixed mean value, and *poisson cyclic*, where the average arrival rate follows a poisson

distribution multiplied by a sinusoidal function. As such, the latter type can capture more realistically situations where the machine workload fluctuates between time periods. The machine contention for poisson cyclic workload types is depicted in Fig. 2. In that figure, the average number of jobs starting per second is 1, the average job duration is 1 sec., and the period of the poisson cyclic load is 5 secs; also, the join is parallelized over 3 machines, one of which is perturbed with the external load described. The randomness of the external load, and the inner complexities of query processing result in more realistic, albeit more complex non-smooth load profiles, thus posing a significant challenge to load balancing controllers. In each experiment, the techniques were checked under at least 10 different random imbalances, and the mean performance values were obtained.

LQR tuning policy: An approach to configuring the LQR-based load balancer, which proved to be effective as will be discussed subsequently, is as follows: when the imbalance decreases (e.g., fewer external jobs arrive at the perturbed machines, or the overall number of machines increases while the number of perturbed machines remains the same), it is safe to make the controller more aggressive without risking performance degradation. Similarly, when the imbalance increases, the controller should become less aggressive. A less (resp. more) aggressive behavior is achieved by increasing (resp. decreasing) the weights in the $\mathbf{R}$ matrix, or by decreasing (resp. increasing) the weights that correspond to the accumulated errors in the $\mathbf{Q}$ matrix, or by both these mechanisms. However, in most of our examples, and in order to keep the LQR configuration part as simple as possible, the $\mathbf{Q}$ matrix was set to the identity one. Note that a less (resp. more) aggressive controller requires more (resp. less) steps to converge. In the following experiments, we have experimented with a small number of rather intuitive LQR configurations with a view to illustrating the potential of the LQR approach to this type of load balancing; it is out of the scope of this work to fine tune the LQR controller so that its best performance possible is achieved.

To smooth the differences between successive instances of $\tilde{\mathbf{B}}$, the machine loads are normalized, so that their sum is 1. For both techniques, adaptations of workload distribution are enforced only if the allocation is modified by 5% or more at least on one machine with a view to avoiding overreacting. Each time step, i.e. each controller cycle, is equal to 0.1 secs. Due to the rapidly changing imbalances, it might be the case that the load change on a machine between two consecutive steps is higher than a threshold (set to 5%), which implies that the load has not temporarily converged; in that case, no effort to adapt is made by any balancing mechanism examined, since any such efforts are prone to cause performance degradation. Also, LQR starts enforcing adaptations only after an initial settling period, to avoid oscillations in the starting phase. Finally, as in [2], the original Flux proposal with respect to the mechanism that is responsible for enforcing rebalancing decisions, is modified so that, at each step, as many state chunks are transferred as required to reach a balanced state; limiting this number to one renders the technique too sensitive to the overall number of chunks and the nature of imbalance. LQR re-uses the same mechanism to realize adaptations.

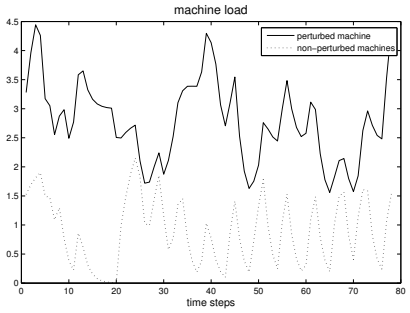

Fig. 2. Typical machine loads when evaluating the example join query and, on one of the machines, a poisson-cyclic load is imposed.

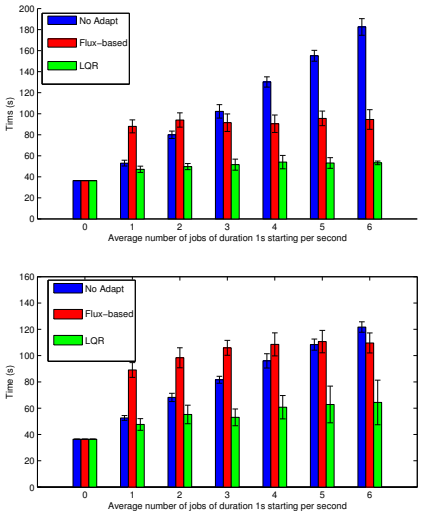

Fig. 3. The average response times of Q1 for a random poisson (top) and poisson cyclic (bottom) imbalance.

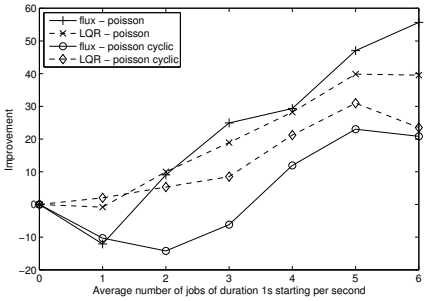

Fig. 4. Percentile improvements over the non-adaptive case for Q2.

Experiment: Performance evaluation in a non network-constrained environment: The first query to be examined is a key/foreign-key join over the *Order* (1.5M tuples) and *LineItem* (6M tuples) tables of TPC-H database; this query will be referred to as *Q1*. In Q1, scans project out 25% of the columns so that communication cost does not dominate. In this experiment, the degree of intra-operator join parallelism is 3, and we compare the improvements of LQR and a Flux-based approach over the non-adaptive case for varying average numbers of external jobs arriving at one of the machines per second. The period of the sinusoidal function

in poisson cyclic imbalances is always set to 5 secs.

The average response time of ten random imbalances for the non-adaptive, Flux and LQR cases are shown in Fig. 3. Two main observations are: (i) both for poisson and poisson cyclic imbalances, LQR performs consistently better than Flux; and (ii) LQR avoids situations where adaptivity yields higher response times, whereas Flux fails to balance the execution in a beneficial manner in a wide range of cases thus causing further performance degradation. For poisson-cyclic imbalance, the average improvement when LQR is employed is 27.8% compared to the non-adaptive case; to the contrary, Flux yields 21% higher response times. For the poisson imbalance, both techniques improve performance; however, the average performance improvements of LQR are more significant (41.9% to 6.3%).

The values of $\mathbf{R}$ that yielded this performance are $0.3 \cdot \mathbf{I_{2,2}}$, whereas $\mathbf{Q}$ is always set to $\mathbf{I_{2,2}}$. As mentioned earlier, finer tuning of this matrix may lead to even higher performance. To check how sensitive LQR is to different values of $\mathbf{R}$, we also experimented with $\mathbf{R} = \mathbf{I_{2,2}}$. The differences in the LQR performance with this configuration are not significant (2-5%). The settling period used was 15 steps for poisson imbalances and 10 steps for the poisson cyclic ones. Small changes in these values did not seem to have a significant impact on performance either.

Fig. 3 also depicts the standard deviation of mean time, which, in general, can be relatively high for LQR. This is due to the fact that occasionally, LQR performs significantly worse than its average performance. As a result, in all experiments conducted, the median response time for LQR is consistently 1-5% lower than the average response time.

Intuitively, LQR performs better for longer-running and continuous queries. But it can also yield performance improvements for smaller queries. Let *Q2* be a query joining *Part* (200K tuples) and *PartSupplier* (800K tuples) on a key/foreign-key bases. The performance improvements are shown in Fig. 4 (corresponding to averages of ten random runs). Again, for poisson cyclic imbalances, LQR performs consistently better than Flux; also applying LQR does not lead to negative improvements as Flux may do. On average, with LQR, the response time is reduced by 22.6% and 15.2% when the imbalance type is poisson and poisson cyclic, respectively. Note that in this experiment, we experimented with only three values for the $\mathbf{R}$ matrix: 0.1, 0.3 and 0.5, with the graph showing the best performing case.

Further experiments are designed in [11] so that concrete insights into the intrinsic characteristics and the behavior of the balancing approaches are provided when attributes such as the size of the relations participating in the join, the number of joins, the load level, the number of the perturbed machines, and the communication bandwidth vary. In addition, the overhead of deploying an LQR controller on a real computer is examined, and the associated cost has been found at the orders of milliseconds measured on a Intel Core2 Duo at 2.2 GHz machine with 3GB of RAM. As such, the overall overhead incurred is low enough not to annul the performance benefits.

V. Conclusions

This work presents a novel approach to balancing parallel query execution over machines with unpredictably time-varying load where not taking into account the cost of balancing decisions leads to suboptimal behavior. The proposal is founded on applied control theory and more specifically on linear quadratic regulation (LQR) optimal control. The evaluation results demonstrate the superiority of LQR and its ability to find a beneficial trade-off between balanced execution and balancing cost. Nevertheless, this work can be extended in various ways. One the most promising direction is the dynamic configuration of the controller (both at an inter- and intra-query level), which implies the development of a switching controller. Designing stable and efficient switching controllers to operate in a rapidly changing environment is a highly non-trivial control theoretical problem.

VI. Acknowledgements

Dr. Yfoulis has been supported by the ATEI grant titled "Adaptive QoS control and optimization of computing systems". This work was partially conducted while A. Gounaris was holding a part-time research position with the University of Manchester.

References

[1] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *Proc. of ICDE*, pages 25–36, 2003.

[2] N. W. Paton, J. Buenabad-Chavez, M. Chen, V. Raman, G. Swart, I. Narang, D. M. Yellin, and A. A. A. Fernandes. Autonomic query parallelization using non-dedicated computers: an evaluation of adaptivity options. *VLDB J.*, Online First, 2008.

[3] S. Lightstone, B. Schiefer, D. Zilio, and J. Kleewein. Autonomic computing for relational databases: the ten-year vision. In *Proc.of the IEEE Workshop Autonomic Computing Principles and Architectures (AUCOPA)*, pages 419–424, 2003.

[4] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[5] Y. Diao, J. L. Hellerstein, S. S. Parekh, R. Griffith, G. E. Kaiser, and D. B. Phung. Self-managing systems: A control theory foundation. In *Proc. of IEEE ECBS 2005*, pages 441–448, 2005.

[6] S. Lightstone, M. Surendra, Y. Diao, S. S. Parekh, J. L. Hellerstein, K. Rose, A. J. Storm, and C. Garcia-Arellano. Control theory: a foundational technique for self managing databases. In *ICDE Workshops*, pages 395–403, 2007.

[7] J. Birdwell, T. Zhong, J. Chiasson, C. Abdallah, and M. Hayat. Resource-constrained load balancing controller for a parallel database. In *Proceedings of the American Control Conference*, 2006.

[8] A. Gounaris, C.A. Yfoulis, R. Sakellariou and M. D. Dikaiakos. A control theoretical approach to self-optimizing block transfer in Web service grids. ACM Transactions on Autonomous and Adaptive Systems, 3(2), 2008.

[9] Y. Diao, J. L. Hellerstein, A. J. Storm, M. Surendra, S. Lightstone, S. S. Parekh, and C. Garcia-Arellano. Incorporating cost of control into the design of a load balancing controller. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 376–387, 2004.

[10] Y. Diao, C. W. Wu, J. L. Hellerstein, A. J. Storm, M. Surendra, S. Lightstone, S. Parekh, C. Garcia-Arellano, M. Carroll, L. Chu, and J. Colaco. Comparative studies of load balancing with control and optimization techniques. In *Proceedings of the American Control Conference*, pages 1484–1490, 2005.

[11] A. Gounaris, C.A. Yfoulis, and N.W. Paton. Efficient Load Balancing in Partitioned Queries Under Random Perturbations. Submitted for publication, 2008.