Efficient Load Balancing in Partitioned Queries Under Random Perturbations

ANASTASIOS GOUNARIS
Aristotle University of Thessaloniki, Greece
and
CHRISTOS A. YFOULIS
Alexander Technological Educational Institute of Thessaloniki, Greece
and
NORMAN W. PATON
University of Manchester, UK

This work investigates a particular instance of the problem of designing efficient adaptive systems, under the condition that each adaptation decision incurs some non negligible cost when enacted. More specifically, we deal with 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.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed databases*

Additional Key Words and Phrases: adaptation cost, load balancing, LQR, partitioned queries, control theory

1. INTRODUCTION

A broad range of adaptive systems aim at optimizing some objective function while operating in a rapidly and unpredictably evolving environment. One of the most challenging issues in that scenario is the planning of adaptations in such a way that the adaptive system reacts sensibly in the sense that it both avoids unstable conditions, under which no progress in execution can be made due to continuous adaptations, and considers the adaptation overhead during planning. The latter is important when the functions of identifying a response to a change or enacting such a response incur non-negligible cost. In this work, we deal with a particular instance of this problem, which is encountered in parallel processing of database queries using non-dedicated resources.

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*) [DeWitt 1992].

However, to fully exploit the potential of parallelism, work needs to be assigned to machines in a way that reflects their capabilities. This type of load balancing, which can be also regarded as a constrained optimization problem, is challenging in an environment with heterogeneous and potentially autonomous, non-dedicated resources. Key characteristics of a typical such environment include the following:

—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.

—There exist unpredictable fluctuations in the load of available machines. A consequence of this fact is that, in the general case, it is not efficient to stick with any decision on the partitioning of a task until completion.

—The instances of subgraphs may be stateful, i.e., they may need to hold some state in main memory (or on disk) in order to be able to evaluate the data assigned to them. This is the case when the subgraphs comprise join and grouping operators, which are some of the most commonly used operators in real queries. A consequence of the fact that the instances of subgraphs are stateful is that part of the state must be moved from one machine to another over the network every time a workload re-assignment takes place, which entails that the cost of performing adaptations is not negligible.

The approach presented in this work is novel in the way the combination of the three afore-mentioned characteristics are tackled, and is founded on applied control theory; the problem under investigation falls into the broader vision of developing autonomic, self-managing solutions for data management [Lightstone et al. 2003]. 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 [Hellerstein et al. 2004]. Nevertheless, their application to computing systems is rendered problematic because of issues such as effective modelling of the system and its dynamics, and overhead times in enforcing adaptations [Diao et al. 2005].

In control systems, the main part is the controller, which receives system measurements, and provides a system configuration that impacts on system performance. There are several orthogonal dimensions across which controllers can be characterized and compared. A controller can be either single-input or multiple-input, and similarly, single-output or multiple-output. The controller inputs (not to be confused with the feedback collected) are adjustable configurations of key system parameters, such as proportion of tuples or memory allocated to a machine, whereas the outputs are measurable properties of the system, such as response time, CPU utilization, and so on. When the measured outputs impact on the controller input, then the controller is termed as feedback or closed-loop, otherwise it is called feed-forward or open-loop. The former does not require the development of accurate complex models and can tolerate relatively high model inaccuracies, thus it is more practical for use in volatile environments. The controllers can be either continuous-time or discrete-time; mostly all controllers of computing systems belong to the latter category due to the nature of computing systems. Finally, controllers can be adaptive themselves in the case where runtime measurements can lead to changes in their own design [Åström and Wittenmark 1995; Dumont and Huzmezan 2002].

To address the problem of balancing the load of a partitioned query across multiple heterogeneous machines, we must employ an adaptivity mechanism that is aware of the potentially significant costs incurred by the adaptations. Otherwise, these overheads may outweigh any benefits. To this end, we employ 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 in our load balancing problem) along with the cost of deviations from the ideal state, in a unified cost function. This property allows LQR controllers to avoid continuous, non-beneficial adaptations. In this work, the focus is on presenting how such a controller can be applied to parallel database queries and examining its behavior, rather than on the control theoretical technical details of the controller's design, which have appeared in [Gounaris et al. 2009]. We also believe that this work is of broader interest since, both the load balancing problem for database queries is representative of load balancing problems in other areas, and the issues encountered with respect to the adaptation cost exist in several other autonomic data management scenarios.

The contributions of this work are summarized as follows:

—it discusses the applicability and the configuration methodology of a control theoretical approach to load balancing in (stateful) parallel database queries based on LQR, which is inherently suitable for adaptations that incur some cost;

—it provides evidence that the resulting mechanism is stable, effective and capable of reaching a balanced state in short times; and

—it compares its performance against state-of-the art load balancing proposals. More specifically, our solution is thoroughly evaluated against the adaptive load balancing methodology of Flux [Shah et al. 2003], and its overhead is investigated.

The remainder of this article is structured as follows. Section 2 discusses related work. The presentation of our technique, along with a more detailed problem description and a brief introduction to control theory, is in Section 3. In Sections 4 and 5, we investigate the stability of the approach proposed and its efficiency and effectiveness, respectively. Finally, Section 6 concludes the article. The technical details and the theoretical proofs of the controllability (i.e., the system's capability of reaching any desirable state with the appropriate input within a bounded time period) and stabilizability of the system, which are essential for those interested in the blending of control theory and autonomic computing systems, are presented in the appendix.

2. RELATED WORK

Typically, load balancing in databases is either addressed just before execution (e.g., [Rahm and Marek 1995; Mehta and DeWitt 1995]) or, for specific operators, at a single point during execution (e.g., [Wolf et al. 1993; Lu and Tan 1992]). More adaptive solutions fall into the area of adaptive query processing [Deshpande et al. 2007], which is a relatively new field dealing with self-optimizing execution of queries. However, most of the proposals in this area so far do not consider parallel or distributed execution, thus overlooking runtime load balancing problems.

One of the most notable examples, which tries to address the challenges that have motivated our work, is the Flux approach [Shah et al. 2003]. Flux introduces a new query 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 time 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 [Paton et al. 2009], some extensions to the Flux approach are described. More specifically, a change to the Flux algorithm is proposed, to carry out replication during the probe phase in operators involving hash tables. In other words, the adaptivity decisions are taken in the same way but the operator state is not moved as in Flux, rather it is replicated at the expense of higher memory usage. This may reduce the number of future state movements, and, consequently, it performs significantly better when many adaptations are needed during execution, assuming that there are no memory limitations. As such, it may be quite successful in some cases, but is suboptimal where the build phase is long compared with the probe phase, or where memory is not abundant. Another proposal is the use of a dynamic hash table approach in which all hash table inserts and probes take place twice. However, this seems not to be a winner, since it is characterized by a significant response time overhead where no load balancing is required, and by creating considerable amounts of extra work that reduces throughput in a loaded environment.

In addition, an enhancement of the Flux decision making criterion is investigated in [Paton et al. 2009], in which, Flux 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.

Several other dynamic flavors of the load balancing problem in database queries have been examined, as well. For example, in [Balazinska et al. 2004], each node acts selfishly and autonomously based on bilateral contracts, whereas in [Xing et al. 2005], load balancing is achieved through the redistribution of operators rather than redistribution of data. In a multi-query environment, the problem changes significantly and load balancing can be performed at the granularity of individual queries (e.g., [Gedik and Liu 2003]).

In investigating the use of techniques from control theory for directing decisions in adaptive load balancing for stateful database operators, the results in this article can be related to recent and ongoing research on *decision making in autonomic systems*, *decision making in adaptive query processing* and *applications of control theory to computer systems*.

In terms of *decision making in autonomic systems*, the autonomic computing community investigates techniques for supporting self-configuration, self-optimization, self-healing and self-protection [Kephart and Chess 2003]. All such activities involve control loops in which the behaviour of an autonomic system is changed at runtime in the light of feedback. A recent classification identifies control loops that are directed by *action*, *goal* and *utility* policies [Kephart and Das 2007]: an *action* policy typically takes the form `if` *Condition* `then` *Action*, and thus makes explicit how a system should respond to a given state; a *goal* policy has a fixed objective, and it is the role of the controller to identify an action that moves the system towards a desirable state; and a *utility* policy provides a function (a *utility function*) that computes the relative values of alternative states, and a collection of actions that can change the state of a system – it is then an optimization problem to identify which actions yield the greatest utility. In this work, we can be considered to be implementing a *goal* policy; the goal is to balance load, and control theoretic techniques determine the action that is taken to meet the goal.

Typically, *decision making in adaptive query processing* has involved the use of *action* policies. As such, an adaptive system monitors some aspect(s) of the progress of a query, the condition diagnoses a problem with the current evaluation strategy, and the action revises some aspect of query evaluation. As an example, *POP* [Markl et al. 2004] monitors operator cardinalities, and the condition checks whether the cardinalities encountered at runtime are within the range for which the current plan was considered by the optimizer to be optimal. If not, then the *Action* calls the optimizer to generate a new plan given the runtime statistics and materialized intermediate results. In several adaptive query processors, action policies are implemented explicitly using a trigger-style syntax [Ives et al. 1999; Ng et al. 1999], although for the most part condition testing and action implementation are implemented through changes to the query evaluator (for example, in POP, conditions are monitored in a *CHECK* operator inserted in the execution plan that itself initializes reoptimization if required [Markl et al. 2004]). Action policies have also been deployed for adaptive load balancing. For example, in DITN [Raman et al. 2005], a query consists of a collection of parallel fragments, $f_i \in Q$, and the condition is a test to establish if the completion time of any f_i is more than twice the completion time of the first completed fragment. If so, the action creates and executes a redundant clone f_i' of f_i that is in a race with f_i to generate data that can contribute to the answer of the query. As already mentioned, in Flux [Shah et al. 2003], the condition is to establish the presence of load imbalance, and the action updates the distribution policy and relocates hash table state in a way that reflects the new distribution policy. However, action policies often embed heuristics that influence what adaptations can take place and when. For example, Flux employs heuristics that constrain the maximum and minimum changes that can be made to a distribution policy during an adaptation, and the frequency with which adaptations can take place. This reflects the fact that writing action policies is often quite involved, and that heuristics and thresholds can have a significant impact on the behavior of an algorithm. This work seeks to provide a firm foundation for decision making based on control theory.

In terms of *applications of control theory to computer systems*, a nice overview of existing proposals and a summary of techniques appears in [Hellerstein et al. 2004]. Control theoretical solutions with a view to achieving self-managing behavior have been incorporated into commercial systems [Lightstone et al. 2007], 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. The usage of LQR in a database environment has been proposed in [Diao et al. 2004; Diao et al. 2005] with the aim of adjusting the sizes of memory pools in a database system. At a higher level, our work adopts a similar framework; however it differs from [Diao et al. 2004] by not relying on offline profiling for the controller configuration, which results in a completely different mode of operation, according to which the controller is continuously readjusted. In contrast with our work, in [Diao et al. 2004], the controller is non-adaptive, and operates in a less volatile setting; moreover, in [Diao et al. 2005], it is assumed that each memory pool can be balanced independently, which is not the case in workload balancing.

To the best of our knowledge there is no prior control theoretic work that deals with balancing the execution of partitioned query tasks in volatile settings; however an interesting approach to enforcing desired utilization set points under a range of dynamic workloads with the help of a controller appears in [Fu et al. 2006], where the methodology adopted is based on diffusive load balancing. In a different setting, cost-aware load balancing has been investigated in [Birdwell et al. 2006], as well. 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

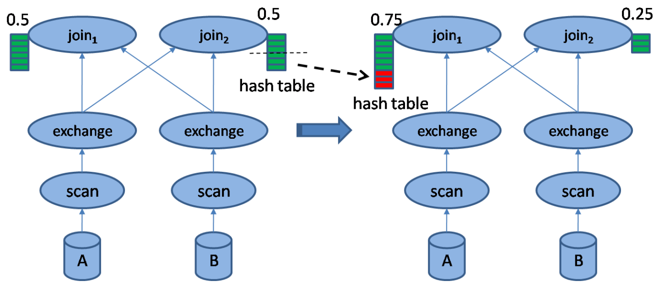


Fig. 1. An example of balancing stateful operators.

data transmission from service-based databases [Gounaris et al. 2008b; 2008a].

3. THE LQR CONTROLLER

3.1 Description of the Load Balancing Problem

Consider two relations with two attributes each, *A(X,Y)* and *B(Y,Z)*, which are joined remotely using a hash join; the hash table is built on $A.Y$. The hash table must be temporarily stored as query state on each participating machine, and subsequently, is probed by tuples from B one by one. A complete query graph includes also scan operators that are responsible for extracting the data from the stored relations, and exchange operators that are responsible for distributing data to the participating nodes. Let us assume that the hash join operator is parallelised over two physical nodes, and that these two nodes are capable of processing tuples from B at the same speed. Then, in a balanced execution, the two nodes should receive and process the same amount of workload measured in tuples. 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. This involves two things: firstly, the B tuples are not distributed equally to the two joins from that point on, and secondly, the corresponding A tuples from the hash table on the second machine must move to the first one. State movements can be enabled through simple extensions to the operator interface (e.g., as in [Shah et al. 2003]); also, note that during such movements the execution is suspended to avoid the generation of erroneous results.

A trivial solution to 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, where the numbers next to the hash tables denote the proportion of the assigned workload). 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, especially when the load conditions change frequently.

The role of the dynamic load balancer is, at the end of each adaptivity step, to re-distribute the workload in such a way that the following is minimized:

$$max(y_i(k+1) + c_i(k+1)), \quad i = 1 \dots P \tag{1}$$

where P is the number of nodes participating in query execution, $y_i(k)$ defines the expected value for the completion time of the ith node given the workload allocation in the kth step, and $c_i(k+1)$ denotes the cost (overhead) to reach a correct state regarding the decisions in the kth step. The constraint is that the workload proportions assigned must be non-negative numbers, no greater than 1, and their sum must be 1.

To the best of our knowledge, there is no practical methodology to solve Eq. (1) analytically. However, 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 $y_i(k) = \frac{1}{P}\sum_{i=1}^{P} y_i(k)$.

3.2 Control theory basics

Before proceeding to the description of our solution, we briefly explain the required control theory concepts, terms, models and design techniques; more details can be found in textbooks such as [Hellerstein et al. 2004; Franklin et al. 1998].

The main task in control theory is to achieve desired goals related to the performance of a target system, which are referred to as the *control objectives.* A typical case is regulating the main characteristics of interest (such as response time or CPU utilization) to desired values. The key idea in *feedback* control is to use measurements of the quantities to be regulated (which have to be measurable and are termed *measured outputs*) to determine the *control inputs*, which are adjustable configurations of key system parameters, such as the proportion of tuples or memory allocated to a machine. The fact that the measured outputs are used to adjust the control inputs which then influence the outputs, results in a circular flow of information suggesting the term *feedback* or *closed-loop* control.

A typical feedback control system consisting of a single feedback loop is shown in Figure 2. For illustration purposes, a single input-single output (termed *SISO*) control system is shown, although in general multiple input-multiple output (*MIMO* or *multivariable*) systems are commonly found, especially in computing systems. The *block diagram* in Figure 2 is a standard tool for showing all the key elements of a feedback control system and their input-output relationships, as well as the main signals of interest and the flow of information. The *target system* is the computing system to be controlled (e.g. the parallel query processing system in our work). The *reference input* (sometimes also termed the *desired output* or the *setpoint*) is the desired value of a system's measured output (e.g. expected completion times for a node in our setting). The *transducer* (also usually termed *sensor*) captures effects such as unit conversions and delays, and transforms the measured output so that it can be compared with the reference input. The difference between the reference input and the measured output is the *control error.* The *controller* adjusts the setting of a control input to the target system with a view to causing its measured output to become equal to the reference input. The controller computes values of the control input based on current and past values of the control error.

The systematic construction of a controller requires a model of the input-output relationship of the target system, usually termed the *system model.* The main reason for the success of *automatic* (feedback) control is that the aforementioned problem of regulating a system's output to a desired value specified by the refer-

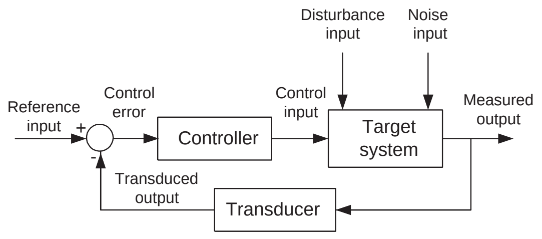


Fig. 2. Block diagram of a feedback control system.

ence input can be solved by a properly designed controller, even in the presence of disturbances and limited knowledge of the target system's behavior (model *uncertainties*). This is attributed to the use of feedback. An alternative is *open-loop* or *feedforward control* where the controller uses the reference input (or sometimes even the disturbance input, when it is measurable) to adjust the control input. Measurements of the control output are avoided and design complexity is reduced. However, such a scheme is rarely used in practice as it can be successful only in unchanging or predictable operating environments where an accurate system model can be built. This is not the case in computing systems due to the volatility of the environments, which impose serious difficulties to the development of accurate complex models.

To model a system, a transfer function describing how the input is transformed into the output may be used. An alternative to the transfer function models is to use *state-space models* for describing system dynamics. System dynamics can be described in terms of variables other than the control inputs and the measured outputs. These need not be measurable, but they are *internal* variables that allow a complete description of a system. The set of variables used in a state-space model is called the *state vector*. Transfer function models are effective at modeling SISO systems, while state-space models provide a scalable approach to modeling MIMO systems.

The design problem is to select the feedback gains that yield the desired controller properties. There are two main design techniques for state-space models, i.e. *pole placement* and *linear quadratic regulation (LQR)*. The first determines the poles needed to achieve the desired closed-loop properties and then computes the feedback gains required. The second approach, LQR, employs an optimization technique that parameterizes the trade-off between control errors and control effort, and as such, is more suitable for our problem.

3.3 Design methodology of the LQR controller

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. Full technical details are given in the appendix and in [Gounaris et al. 2009].

The first stage in the application of an LQR-based adaptivity technique is to build a state-space model, which involves the definition of the control inputs, system variables and measured outputs. In our setting, the output values are the expected completion times for each node and the inputs, i.e. our manipulated variables, are the workload allocations at each step. The state variables contain the control error of each machine, which is the difference between the current measured output of a machine and the average measured output between all machines. Due to the unpredictability of machine load and the time-varying average measured output, the state variables also contain the accumulated error to this point for each machine. The accumulated error contains the sum of the values of the errors in all steps. To simplify the design, we assume that the expected completion time depends solely on the control input, i.e., the workload allocation, and the machine load. Also we assume that the expected completion time of one machine does not depend on the processing speed of another. The actual processing speed of each machine can be monitored at runtime.

There are two main approaches to meeting the constraint regarding the workload proportion, which directly impacts on the number of nodes for which the workload allocation is explicitly decided by the controller: (a) to normalize the controller decisions so that the constraint is met; or (b) to allow the controller to specify the workload allocation only for $P-1$ nodes, and the allocation of the last node to be the difference between 1 and the sum of the $P-1$ workload allocations. The first approach results in a nonlinear control system which is difficult to analyze and to assure stability, convergence and robustness properties; however it is included in the present work for completeness. The second approach to constraint satisfaction is much more amenable to rigorous analysis, and it can be shown that the system is fully controllable. Furthermore, 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; proofs for these properties are delegated to the appendix.

By designing an appropriate stabilizing controller using the state space model described, which has disturbance rejection properties due to the accumulated error state variables (integrators), we can guarantee that the errors of all P states converge to zero. In addition to load disturbances and reference trajectory disturbances (the reference point of each output is the average of all outputs), the controller can also tolerate modeling inaccuracies.

In such a state model, the control input vector (denoted as $\mathbf{u}(k)$) is a linear function of the system's state, i.e., the workload decisions are linearly related to error states:

$$\mathbf{u}(k) = -\mathbf{K}\,\mathbf{x}(k)\ ,\ \mathbf{x}(k) = \begin{bmatrix} \mathbf{e}(k) \\ \mathbf{e_I}(k) \end{bmatrix} \tag{2}$$

where $\mathbf{x}(k)$ is the state vector, which consists of the error vector $\mathbf{e}(k)$ and the accumulated error vector $\mathbf{e_I}(k)$. The LQR framework is responsible for the specification of the $\mathbf{K}$ matrix, and more importantly, for ensuring that this matrix efficiently implements the tradeoff between quick convergence to the optimal workload allocation and the penalty for this convergence. To enable that, the cost function (to be minimized) employed by LQR is in the following form

$$\mathcal{J} = \frac{1}{2} \sum_{k=1}^{\infty} \left[\mathbf{x}^{\mathbf{T}}(k)\, \mathbf{Q}\, \mathbf{x}(k) \;+\; \mathbf{u}^{\mathbf{T}}(k)\, \mathbf{R}\, \mathbf{u}(k) \right] \tag{3}$$

where, in our controller, the matrices $\mathbf{Q}$ and $\mathbf{R}$ are set in a way that the requirement for quick convergence is quantified by the square of state variables multiplied by the weights in the $\mathbf{Q}$ matrix, and the convergence penalty is quantified by the square of the control input multiplied by the weights in the $\mathbf{R}$ matrix.

The dimensions of the $\mathbf{Q}$ and $\mathbf{R}$ matrices are $2N \times 2N$ and $N \times N$, respectively. N denotes the number of controlled nodes, and depends on the way the constraint is enforced; when the first approach to constraint satisfaction is employed, it is equal to P, whereas, when the second one is preferred, it is equal to $P - 1$.

To summarize, the problem of developing a load balancer that considers the overhead of its decisions, has been now 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 Eq. (1).

An important step in the configuration is to 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, and to find 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, $\mathbf{Q}$ is in the following form

$$\mathbf{Q} = \begin{bmatrix} a\,\mathbf{I}_{N,N} & \mathbf{0}_{N,N} \\ \mathbf{0}_{N,N} & b\,\mathbf{I}_{N,N} \end{bmatrix}, \quad a, b \geq 0 \tag{4}$$

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 have to 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. The diagonal elements of $\mathbf{R}, r_i, i = 1 \ldots N$, are of a scalar value c, which is a tuning parameter; the higher its value, the more a change in the workload allocation is penalized.

As mentioned previously, the new workload allocation in each step (i.e., the control input) is found by multiplying $\mathbf{K}$ with the state vector (see Eq. (2)). It is outside the scope of this work to present how $\mathbf{K}$ can be specified analytically. Further details can be found in any control theory textbook, see e.g., [Franklin et al. 1998]. In practice, toolkits, such as the Control System Toolbox of $MATLAB^{TM}$, provide an automated technique for its estimation. However, it should be noted that our controller has *adaptive* features, since, at each step, the LQR problem is solved and new gains $\mathbf{K}(k)$ in Eq. (2) are specified, in light of the updated processing speeds of the participating nodes.

In this work, we consider only weighting matrices that remain fixed throughout the query execution, as is the typical case with LQR controllers. In Sections 4 and 5.2.1, we further discuss how the value of their elements can be defined following some simple heuristics. 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.

4. STABILITY AND CONVERGENCE PROPERTIES

In this section, the effectiveness of our proposal is investigated. We examine the properties of stability, accuracy, convergence speed, and overshoot [Diao et al. 2005] with regard to the way the constraints are satisfied and the number of nodes that are directly controlled, and the values of the $\mathbf{Q}$ and $\mathbf{R}$ matrices. Note that we evaluate the efficiency of the system in terms of performance and the capability to take adaptation cost into account in the next section, which includes the comparison of our approach with other proposals. This section also aims at providing clear insights into how the LQR controller operates and hints for the parametrization of $\mathbf{Q}$ and $\mathbf{R}$, with the help of two simple examples, simulated in $MATLAB^{TM}$. The theoretical discussion of the system controllability and stabilizability, which is closely related to the way the constraints are satisfied, is deferred to the Appendix.

4.1 Example with just two machines

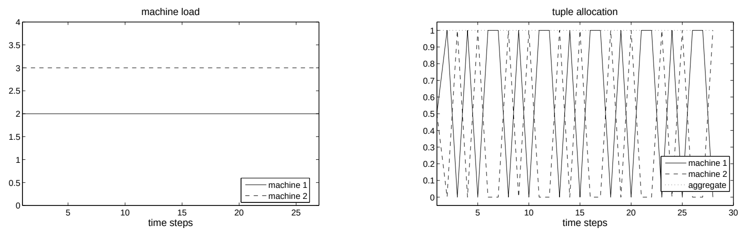


Fig. 3. Left: the (static) load of the two machines. Right: the controller's tuple allocation decisions when the controller decides on the workload allocation of all nodes explicitly.

In the first example, there are two machines that evaluate two instances of the same partitioned operator. The two machines have equal computational capacity, but their CPU load during execution differs (e.g., due to external jobs running).

Unfortunately, in this setting, stability cannot always be guaranteed. Consider for example Fig. 3, where the initial allocation regards the two machines as of equal capacity but in reality, one is more loaded that the other. When the first approach to the fulfillment of the workload constraint is adopted, according to which the controller decides on the workload allocation of all nodes explicitly, the system oscillates between the upper and lower limits in an unstable manner. This behavior is observed regardless of the choice of the weighting matrices $\mathbf{Q}, \mathbf{R}$. This experiment suggests that this type of controller is inappropriate for our problem.

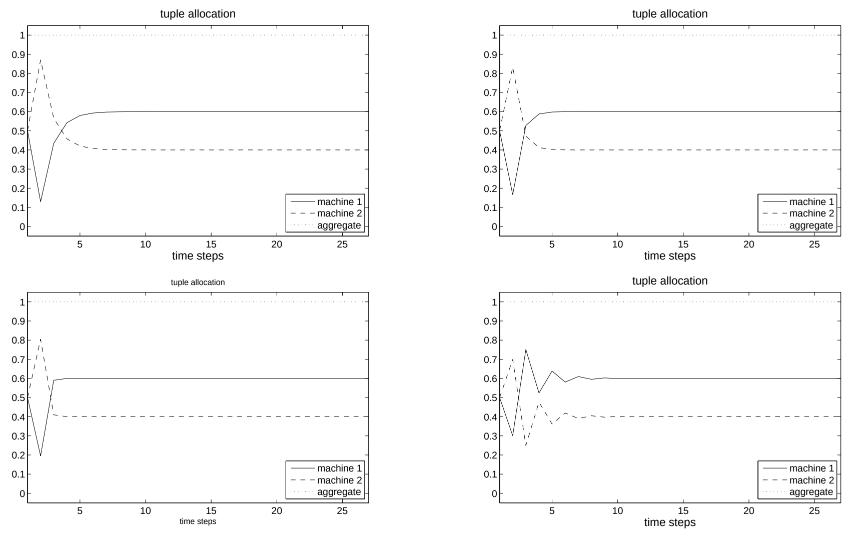


Fig. 4. The tuple allocation decisions when $\mathbf{Q} = diag([1, 1])$ and the $\mathbf{R}$ matrix is (from top right to bottom left): a) $\mathbf{R} = [10]$; b) $\mathbf{R} = [5]$; c) $\mathbf{R} = [3]$; d) $\mathbf{R} = [0.1]$.

A remedy is to use the second controller proposed in the previous section, where the constraint is enforced by specifying the workload proportion of the last node indirectly. When the second approach to constraint satisfaction is employed, the behavior of the system is as expected (Fig. 4). The parameters of the $\mathbf{Q}$ and $\mathbf{R}$

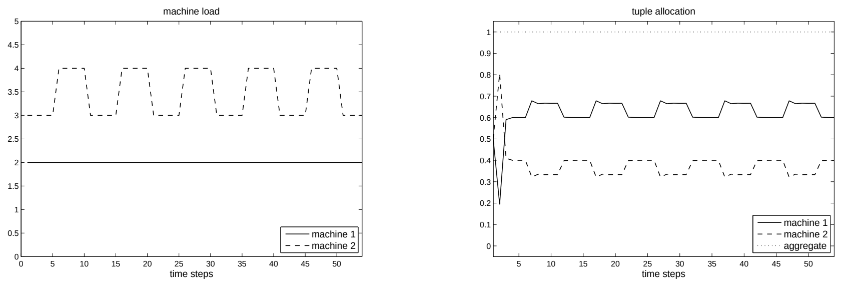


Fig. 5. Left: the (periodic) load of the two machines. Right: the tuple allocation when $\mathbf{Q} = diag([1, 1]$, $\mathbf{R} = [3]$.

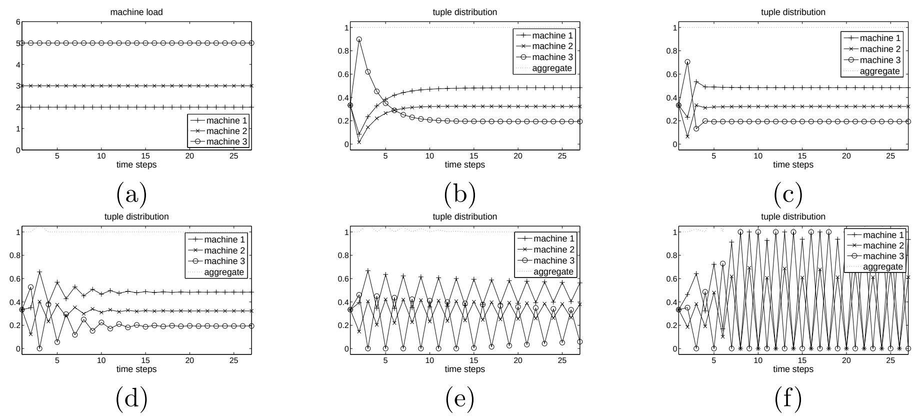


Fig. 6. (a): the (static) load of three machines. (b-f): the tuple allocation decisions when $\mathbf{Q} = diag([1, 1, 1, 1]$ and the $\mathbf{R}$ matrix is: b) $\mathbf{R} = [100, 100]$; c) $\mathbf{R} = [10, 10]$); d) $\mathbf{R} = [3, 3]$; e) $\mathbf{R} = [2, 2]$; f) $\mathbf{R} = [1, 1]$.

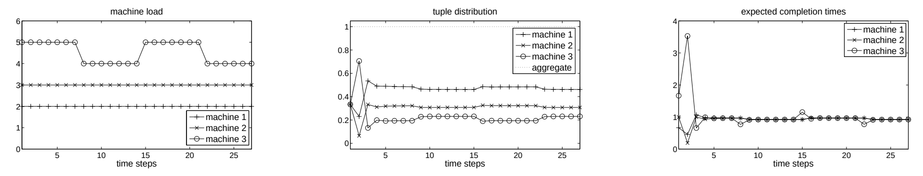


Fig. 7. Left: the (periodic) load of three machines. Middle: the tuple allocation when $\mathbf{Q} = diag([1, 1, 1, 1]), \mathbf{R} = diag([10, 10])$. Right: the expected completion times.

matrices have a significant impact on the speed of convergence and the behavior of the controller, as shown in the figure. If $\mathbf{Q}$ is kept constant, the speed of convergence is inversely proportional to the value of $\mathbf{R}$; i.e., it decreases with a larger value for $\mathbf{R}$, which penalizes the control input more. For small values of $\mathbf{R}$ more aggressive control action is allowed, which induces oscillations due to the initial controller's overreaction. When $\mathbf{R} = [3]$ the system exhibits the best performance: it accurately converges in just 2 steps and enters a totally steady phase after the 3rd step. Another typical adjustment is to apply different weights to the control errors and the accumulated control errors. When the accumulated control errors are more heavily weighted, e.g., when $\mathbf{Q} = diag([0.1, 1])$, a more aggressive controller with larger gains is obtained (not shown in the figure). The opposite happens when smaller weights for the accumulated errors are selected, i.e., a less aggressive controller is obtained leading to a transient response with larger settling time and reduced (or no) oscillations.

LQR controllers can yield nice results in the case of periodic loads, as shown in Fig. 5. In this example the duration of a unit increase of load is 5 steps. The system can converge in 2 steps (and 1 step after it has been informed about the load change). As such, in the example, the system is perfectly balanced 60% of the time. Finally, note that, in all configurations, there is an oscillation in the first few steps; this is due to the initial conditions and, typically, the controller decisions during these steps are not considered. In the experiments in next section, LQR starts enforcing adaptations only after the third step.

4.2 Example with more machines

When three machines participate in the evaluation, the behavior is similar. Consider the example of a static imbalance in Fig. 6. Again, the parameters of the **Q** and **R** matrices seem to have a significant impact on the speed of convergence and the behavior of the controller, as shown in Fig. 6 for varying **R**. The less the control inputs are penalized, the more aggressive the response becomes, since larger control gains are returned by the LQR routine. Hence, it is likely at some point to saturate to the upper (=1) or lower (=0) bounds of our inputs. Hard limit constraints have to be included in our implementation to keep the control inputs inside the designated bounds. Limit cycles are then naturally expected and the unity sum constraint may also be violated. This problematic behavior is shown in the bottom graphs of Fig. 6.

The problem mentioned above can be avoided by careful selection of the **Q** and **R** matrices. This is clearly an application-specific issue that is also dependent on the range of the output values, the machines load, etc. However, what really matters is the relative values of the two parts in the LQR cost function in Eq. (3), i.e., the square of the state variables multiplied by the elements in **Q**, and the square of the workload assignments, multiplied by the elements in **R**. If the first part, which prompts the controller for quick responses, is of the same order of magnitude as the second, which associates a cost to the responses, then the controller is stable, and converges accurately but slowly, as in Fig. 6c,d. However, the speed of convergence increases significantly if the first part is an order of magnitude larger (Fig. 6b). When it becomes even larger (e.g., two orders of magnitude), it may enter instability.

For periodic imbalances, the behavior remains efficient, as shown in Fig. 7 for the choice that yields the best results for a static imbalance in Fig. 6. It is interesting to note that runtime changes of the load of one machine cause significant differences in the expected completion times between machines just for a single step, i.e., the duration of the transient, imbalanced phase is one step (see Fig. 7(middle)). For an arbitrary number of machines, the behavior is similar.

5. EVALUATION

This section compares the performance of the LQR controller described in Section 3 with heuristic control, based on Flux [Shah et al. 2003] and on techniques described in [Paton et al. 2009], using simulations of query performance under time-varying imbalance conditions. More specifically, different experiments are designed 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. More importantly, before discussing the experiments, useful tips as to how to configure the LQR controller are presented.

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

Table I. Cost model parameters.

5.1 Simulation Model

The simulation uses a cost model consisting of a collection of parameterised cost functions based on those validated in [Sampaio et al. 2006]; the parameters were obtained from the execution of micro-benchmark queries, and are given in Table I. The simulator, which extends that used in [Paton et al. 2009], emulates an iterator-based query evaluator [Graefe 1996]; more details of the simulation model are provided in [Paton et al. 2009].

The following operators are used in the experiments: *scan* simulates the reading of data from the disk and the creation of data structures representing the data read; *hash_join* simulates a main-memory hash join; and *exchange* simulates the movement of tuples between operators on different nodes [Graefe 1990]. The *exchange* departs to an extent from the pull-based approach of the iterator model, in that it reads from its children into caches on the producer node as quickly as it can, and transfers data from the producer cache to the consumer cache as quickly as it can, regardless of how many tuples it has been asked for, until the caches are full. The simulator supports both pipelined and partitioned parallelism, the former through simulation of the iterator model using exchanges to enable partially decoupled evaluation of parent and children operators, and the latter by creating multiple instances of an operator on different nodes.

5.2 Experiments

As in Sec. 4, the load of each machine, both due to external jobs and query processing tasks, is used to estimate the machine capacity. More specifically, the computational capacity of each machine is inversely proportional to machine contention, in line with the approach described in [Paton et al. 2009]. The example operator to be balanced throughout the experiments is a parallel hash join between tables of the TPC-H database with scale factor 1. However, the load balancer presented in this work is independent of any particular operator implementation; the only requirements are the operator to create and store internal state that, in the generic case, has to be moved in case of repartitioning and the existence of

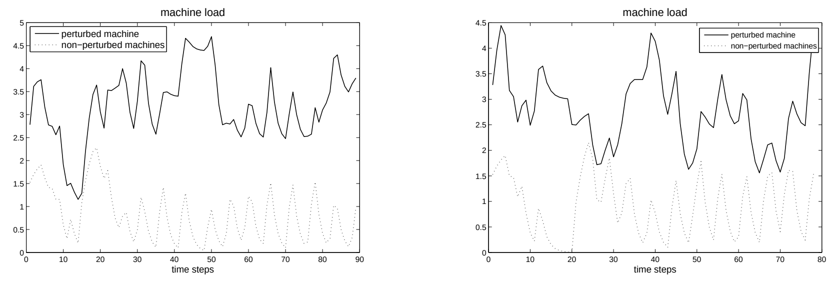
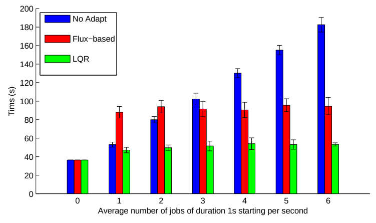


Fig. 8. Typical machine loads when evaluating the example join query and, on one of the machines, a poisson (left) or poisson-cyclic (right) load is imposed.

an interface to enable such state movements. Multiple stateful operators in the same query are balanced by separate controllers, as the controllers operate at the operator level. 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 both these external workload types is depicted in Fig. 8. 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 parallelised over 3 machines, one of which is perturbed with the external load described. Note that under poisson-cyclic imbalances, periodically, the load of a perturbed machine is similar to the load of the non-perturbed ones. 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 and median performance values were obtained.

5.2.1 *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. As discussed in Section 4, a less (resp. more) aggressive behavior is achieved by increasing (resp. decreasing) the weights in the **R** matrix, or by decreasing (resp. increasing) the weights that correspond to the accumulated errors in the **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 **Q** matrix was set to the identity one. Note that a less (resp. more)



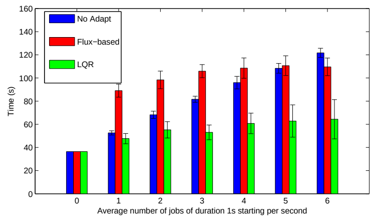


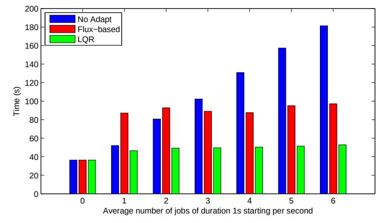
Fig. 9. Experiment 1: The average response times of Q1 for a random poisson (left) and poisson cyclic (right) imbalance.

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, i.e., the configuration of **R** and **Q** that we have employed may be suboptimal.

To smooth the differences between successive instances of the LQR input regarding the machine capacity, the machine loads of Fig. 8 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 of Fig. 8, 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 the initial oscillation shown in the figures in Sec. 4. Finally, as in [Paton et al. 2009], 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.

5.2.2 *Experiment 1: 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 Experiment 1, the degree of intra-operator join parallelism is 3, and we compare the improvements of LQR and a Flux-based approach as implemented in [Paton et al. 2009] 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



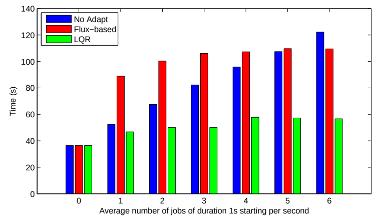


Fig. 10. Experiment 1: The median response times of Q1 for random poisson (left) and poisson cyclic (right) imbalances.

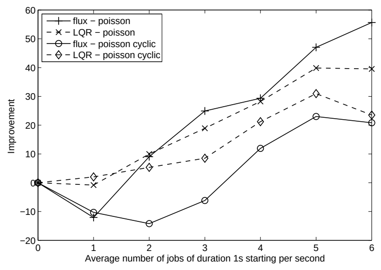


Fig. 11. Experiment 1: Percentile improvements over the non-adaptive case for Q2.

and LQR cases are shown in Fig. 9. 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 $\mathbf{I_{2,2}}$; $\mathbf{I}$ is the identity matrix. 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. 9 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. Fig. 10 shows the median times for Experiment 1.

Intuitively, LQR performs better for longer-running and continuous queries. But

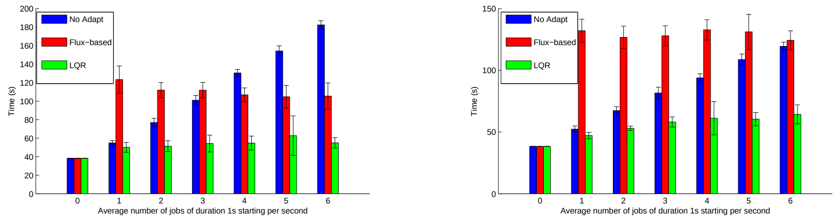


Fig. 12. Experiment 2: The average response times of Q1 for random poisson (left) and poisson cyclic (right) imbalances.

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. 11 (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 **R** matrix: 0.1, 0.3 and 0.5, with the graph showing the best performing case.

5.2.3 *Experiment 2: Performance evaluation in a network-constrained environment.* In this experiment, we apply a ten-fold increase in network latency coupled with a ten-fold decrease in network bandwidth (see Table I) and we repeat Experiment 1. As shown in Fig. 12, the performance of LQR remains essentially the same: LQR reduces the response time on average by 40.2% and 27.8 % for poisson and poisson cyclic imbalances, respectively. However, Flux exhibits much worse performance, yielding increased response times for both imbalance types (e.g., for poisson cyclic imbalances the average degradation is 52%). The configuration of **R** remains the same (i.e., set to $0.3 \cdot \mathbf{I_{2,2}}$): the only difference from the previous experiment is that small increases in the matrix weights improve, instead of aggravating, the performance slightly. This is expected since data movements incur a higher cost in this setting.

5.2.4 *Experiment 3: Varying number of join evaluators.* In this experiment, we repeat the 1st experiment for varying number of machines participating in the join evaluation. On average, under poisson external load, LQR yields 41.9%, 39.7%, 41%, and 39.2% lower average response times compared to the non adaptive case, when the degree of join parallelism is 3 (Experiment 1), 6, 9 and 12, respectively (see Fig. 13). In other words, it exhibits consistent behavior. On the other hand, under the same type of external load, Flux yields 6.3%, 21.8%, 33% and 43.4% performance improvements, respectively. It is interesting to note that in order to achieve this performance, the **R** weights were reduced from 0.3 to 0.1 and to 0.05, for parallelism degrees 9 and 12, respectively, to account for the reduced overall imbalance. This experiment reveals an interesting pattern regarding the behavior of the Flux balancing mechanism: it can yield as significant performance improvements as the LQR controller only if the level of external load on the perturbed

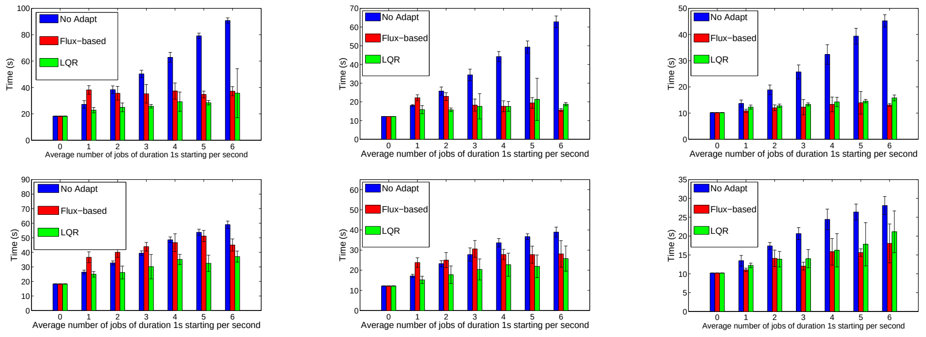


Fig. 13. Experiment 3: Top: The average response times of Q1 for random poisson imbalances when the degree of join partitioned parallelism is 6 (left), 9 (middle) and 12 (right). Bottom: The corresponding average response times for random poisson cyclic imbalances.

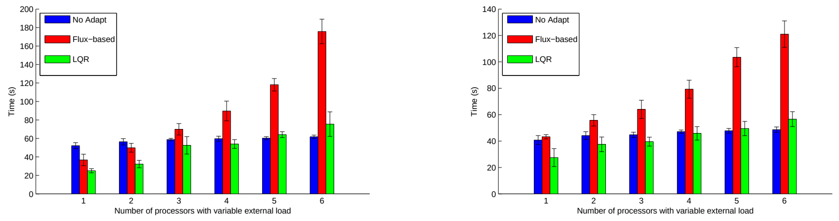


Fig. 14. Experiment 4: The average response times of Q1 for a random poisson (left) and poisson cyclic (right) imbalance.

machine is high or the ratio of non-perturbed machines to the perturbed ones is not small. However, LQR is more robust with respect to these parameters.

The same pattern appears also when the imbalance follows the poisson cyclic distribution: LQR behaves similarly to Experiment 1, whereas the performance of Flux improves with higher degrees of join parallelism, and outperforms LQR when this degree is 12 (Fig. 13). However, configuring LQR efficiently is more intriguing: for small numbers of external jobs arriving at the perturbed machine, the **R** weights are the same as the ones used for poisson imbalance. However, they should be increased (or the initial settling window should be increased) for higher job arrival rates.

5.2.5 *Experiment 4: Varying number of perturbed machines.* In this experiment, the degree of join parallelism is set to 6, the average number of external jobs arriving at the perturbed machines is set to 3, and the number of perturbed machines varies from 1 to 6. Again, the query is Q1. For the two types of imbalance, LQR yields 14.4% and 7% lower response times (on average), whereas Flux fails to yield performance improvements, as depicted in Fig. 14. When no adaptations are allowed, the number of perturbed machines has no impact; the differences in the relevant bars in the figure are solely due to the fact that the imbalances are randomly generated and thus are never the same. When all, or almost all the

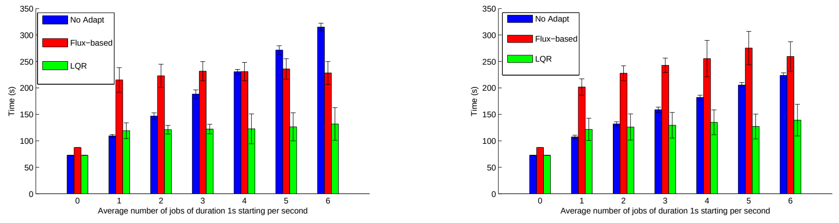


Fig. 15. Experiment 5: The average response times of Q3 for a random poisson (left) and poisson cyclic (right) imbalance.

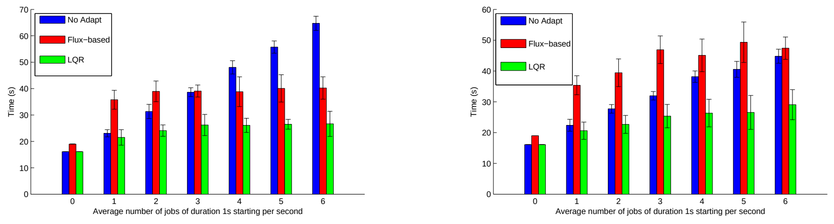


Fig. 16. Experiment 5: The average response times of Q4 for a random poisson (left) and poisson cyclic (right) imbalance.

machines are perturbed, LQR cannot improve performance because it does not avoid non-beneficial data movements; nevertheless the degradation is not as severe as in the Flux case. Since the imbalance increases with the number of perturbed machines, we also increase the **R** weights up to 10 for poisson imbalances and up to 2 for poisson cyclic ones. More specifically, for poisson imbalances, the **R** weights that yield the lower response times are 0.3, 0.3, 1, 2, 5 and 10, for 1 to 6 perturbed machines, respectively. For poisson cyclic imbalances, more moderate increases are needed, especially if they are coupled with an increase of the **Q** elements and an enlargement of the initial settling window: 0.1, 0.3, 1.5, 1.5, 2 and 1.

5.2.6 *Experiment 5: Queries with multiple joins.* In the last experiment configuration, we experiment with two multiple join queries, one 2-join and one 3-join: *Q3*, which joins Q1 with the *Supplier* relation (10K tuples), and *Q4* which joins the *Supplier*, *Customer* (150K tuples), *PartSupplier* (800K tuples) and *Orders* tables. The degree of parallelism for each join is 3, and a separate controller is responsible for each join. As in Experiment 1, LQR can lead to significant performance benefits both for Q3 (see Fig. 15) and Q4 (see Fig. 16). This is not the case though for Flux, which, on average, causes further performance degradations, for both queries and both types of imbalance. Note that the **R** weights and the settling window are a bit larger than those in Experiment 1, up to 1.5 and 20 steps, respectively. This is because the imbalance in one operator due to external jobs is aggravated by the imbalanced execution of other operators in the same pipeline, so that a less aggressive controller is needed.

The results of this experiment provide initial insights into the behavior of our

Experiment	Flux-based		Replication-based		DHT-inspired		LQR	
	mean	median	mean	median	mean	median	mean	median
Exp.1	6.3	7.04	26.7	26.5	28.3	28.2	41.9	43.2
Exp.2	-13	-11	12.6	12.8	27.6	27.5	40.2	42. 5
Exp.3-6 nodes	21.8	20.4	35.9	35.9	35.4	35.1	39.7	41.9
Exp.3-9 nodes	33	32.7	37.2	37.4	37.4	37.3	41	43.3
Exp.3-12 nodes	43.4	45.3	42.3	43.6	39.4	39.2	39.2	39.4
Exp.4	-52	-50	-36	-34	19	19.3	14.4	17.3
Exp.5 - Q3	-22	-23	-6.7	-7.1	7.1	9	28.9	30.7
Exp.5 - Q4	-1.9	-3.5	21.3	21.5	12.4	12.1	31.3	32.6

Table II. The average percentile improvements on the mean and the median response times compared to the non-adaptive case under poisson imbalances.

Experiment	Flux-based		Replication-based		DHT-inspired		LQR	
	mean	median	mean	median	mean	median	mean	median
Exp.1	-21	-22	2.7	1.8	19.6	19.4	27.8	30.8
Exp.2	-52	-52	-29	-31	18.3	18.3	27.8	30.6
Exp.3-6 nodes	-5.7	-7	15.3	15.7	25.7	25.8	22	25
Exp.3-9 nodes	1.9	1.6	14.5	13.3	28.5	27.5	24	25.4
Exp.3-12 nodes	27.2	25.5	30.3	29.7	30.3	30.5	21.8	26.5
Exp.4	-68	-67	-56	-56	12.8	12.7	7	9.8
Exp.5 - Q3	-46	44	-29	-29	-5	-1.6	15.9	20.4
Exp.5 - Q4	-30	-31	0.2	-0.3	-0.5	1	19.4	19.4

Table III. The average percentile improvements on the mean and the median response times compared to the non-adaptive case under poisson cyclic imbalances.

technique in scenarios where multiple operators share resources, e.g., in a multi-query setting. However, the interference of multiple controllers for tasks that share resources requires further investigation.

5.2.7 *Experiment 6: Comparison with other approaches.* Up to this point, we compared the LQR and the Flux balancing mechanisms. In [Paton et al. 2009], two additional techniques were introduced, a replication-based one and another inspired by distributed hash tables (DHTs). As mentioned in Sec. 2, both these proposals are characterized by serious limitations, e.g., the DHT-inspired one suffers from reduced throughput due to duplicate work. In Tables II and III, we present the average percentile improvements over the non-adaptive case for all techniques.

Observing these tables, a conclusion that can be drawn is that when a single machine experiences external load, LQR is the best performing technique, unless the overall number of participating machines is relatively high. In the latter case, the difference between LQR and the other techniques is narrow. Especially for network constrained environments (i.e., Experiment 2) and multiple join queries (i.e., Experiment 5), LQR performs significantly better than any of the other three approaches. When more than one machines is perturbed, LQR is capable of yielding improved response times, in contrast to Flux and replication-based approaches. The DHT-inspired solution performs slightly better in this context, but we believe that the performance difference is not adequate to outweigh its drawbacks in more stable

#machines	average (secs)	stdev
3	0.005201894	0.000484936
6	0.005840807	0.000430175
9	0.007319328	0.000765713
12	0.008871405	0.000724823

Table IV. The cost of applying LQR.

environments.

5.2.8 *Experiment 7: LQR overhead.* At runtime, LQR performs the following tasks: (i) it estimates the **K** vector (Eq. (2)); (ii) it reconfigures the partitioning vector to reflect the new workload distribution (the tuples are not reconfigured); and (iii) it moves state from one machine to another. The second task requires a negligible amount of time, whereas the third one is modelled by the simulator and has been included in all performance results presented. The cost of estimating **K**, which involves also the cost of building all the other model matrices, is strongly correlated with the number of machines to be balanced. As shown in Table IV, this cost is 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. For example, while evaluating Q1, LQR spends, in total, 2 secs approximately per run, even though, in our experiments, where each step was 0.1 secs, LQR may be applied hundreds of times.

6. 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. Although we have considered only partitioned database queries, we believe that our work is of broader interest, as the core problem it deals with is the inherent cost of enacting decisions in adaptive systems.

The proposal is founded on applied control theory and more specifically on linear quadratic regulation (LQR) optimal control. As well as a detailed presentation of the controller design and configuration, the contributions of this work include an empirical and theoretical analysis of the stability and convergence properties, and performance evaluation in comparison with state-of-the-art heuristics. 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. Two of the most promising directions are the dynamic configuration of the controller (both at an inter- and intra-query level), and the investigation of its behavior in multi-query settings. The former implies the dynamic configuration of LQR, which corresponds to the category of switching controllers. Designing stable and efficient switching controllers to operate in a rapidly changing environment is a highly non-trivial control theoretical problem. Another interesting direction is to incorporate the controller in a real system and investigate the actual impact of measurements delays and noise on the behavior of the controller; if this impact is significant, more sophisticated and decentralized solutions (e.g. [Camponogara et al. 2002]), should be examined.

ACKNOWLEDGMENTS

Dr. Yfoulis has been supported by the ATEI grant titled “Adaptive QoS control and optimization of computing systems”.

REFERENCES

Åström, K. J. and Wittenmark, B. 1995. *Adaptive Control.* Addison-Wesley, Reading, MA, USA.

Balazinska, M., Balakrishnan, H., and Stonebraker, M. 2004. Contract-based load management in federated distributed systems. In *NSDI*. 197–210.

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

Camponogara, E., Jia, D., Krogh, B., and Talukdar, S. 2002. Distributed model predictive control. *IEEE Control Systems Magazine 22,* 1, 44–52.

Deshpande, A., Ives, Z. G., and Raman, V. 2007. Adaptive query processing. *Foundations and Trends in Databases 1,* 1, 1–140.

DeWitt, D. 1992. Parallel Database Systems: The Future of High Performance Database Systems. *Comm ACM 35,* 6, 85–98.

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

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

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

Dumont, G. and Huzmezan, M. 2002. Concepts, methods and techniques in adaptive control. In *Proceedings of the American Control Conference*. Boston, MA.

Franklin, G., Powell, J., and Workman, M. 1998. *Digital Control of Dynamic Systems.* Reading, Massachusetts: Addison-Wesley, third ed.

Fu, Y., Wang, H., Lu, C., and Chandra, R. S. 2006. Distributed utilization control for real-time clusters with load balancing. In *RTSS '06: Proceedings of the 27th IEEE International Real-Time Systems Symposium*. 137–146.

Gedik, B. and Liu, L. 2003. Peercq: A decentralized and self-configuring peer-to-peer information monitoring system. In *ICDCS*. 490–499.

Gounaris, A., Yfoulis, C., Sakellariou, R., and Dikaiakos, M. D. 2008a. A control theoretical approach to self-optimizing block transfer in web service grids. *TAAS 3,* 2.

Gounaris, A., Yfoulis, C., Sakellariou, R., and Dikaiakos, M. D. 2008b. Robust runtime optimization of data transfer in queries over web services. In *Proc. of ICDE*.

Gounaris, A., Yfoulis, C. A., and Paton, N. W. 2009. An efficient load balancing LQR controller in parallel databases queries under random perturbations. In *3rd IEEE Multi-conference on Systems and Control (MSC 2009)*.

Graefe, G. 1990. Encapsulation of parallelism in the volcano query processing system. In *Proc. of SIGMOD*, H. Garcia-Molina and H. V. Jagadish, Eds. 102–111.

Graefe, G. 1996. Iterators, Schedulers, and Distributed Memory Parallelism. *Software Practice and Experience 26,* 4, 427–452.

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

Ives, Z., Florescu, D., Friedman, M., Levy, A., and Weld, D. 1999. An Adaptive Query Execution System for Data Integration. In *ACM SIGMOD*. 299–310.

Kephart, J. and Chess, D. 2003. The Vision of Autonomic Computing. *IEEE Computer 36,* 1, 41–50.

Kephart, J. and Das, R. 2007. Achieving self-management via utility functions. *IEEE Internet Computing 11,* 1, 40–48.

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

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

Lu, H. and Tan, K.-L. 1992. Dynamic and load-balanced task-oriented datbase query processing in parallel systems. In *3rd Int. Conf. on Extending Database Technology EDBT*. Springer, 357–372.

Markl, V., Raman, V., Simmen, D., Lohman, G., and Pirahesh, H. 2004. Robust query processing through progressive optimization. In *Proc. SIGMOD*. 659–670.

Mehta, M. and DeWitt, D. J. 1995. Managing intra-operator parallelism in parallel database systems. In *Proc. of 21th Int. Conf. on Very Large Data Bases VLDB*. 382–394.

Ng, K. W., Wang, Z., Muntz, R. R., and Nittel, S. 1999. Dynamic Query Re-Optimization. In *SSDBM*. 264–273.

Paton, N. W., Chávez, J. B., Chen, M., Raman, V., Swart, G., Narang, I., Yellin, D. M., and Fernandes, A. A. A. 2009. Autonomic query parallelization using non-dedicated computers: an evaluation of adaptivity options. *VLDB J. 18,* 1, 119–140.

Rahm, E. and Marek, R. 1995. Dynamic multi-resource load balancing in parallel database systems. In *Proc. of 21th Int. Conf. on Very Large Data Bases VLDB*. 395–406.

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

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

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

Wolf, J. L., Yu, P. S., Turek, J., and Dias, D. M. 1993. A parallel hash join algorithm for managing data skew. *IEEE Transactions on Parallel and Distributed Systems 4,* 12, 1355–1371.

Xing, Y., Zdonik, S. B., and Hwang, J.-H. 2005. Dynamic load distribution in the borealis stream processor. In *ICDE*. 791–802.

7. APPENDIX

7.1 Technical details

Here we provide a more formal description of the design of our controller, part of which is in [Gounaris et al. 2009]. According to the load balancing requirement, all outputs y_j , $j = 1, \ldots, P$ (P is the number of machines) 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

$\mathbf{e}(k) = \begin{bmatrix} y_1 \\ y_2 \\ \dots \\ y_P \end{bmatrix} - \frac{1}{P} \begin{bmatrix} \sum_{i=1}^{P} y_i \\ \sum_{i=1}^{P} y_i \\ \dots \\ \sum_{i=1}^{P} y_i \end{bmatrix} = (\mathbf{I}_{\mathbf{P},\mathbf{P}} - \frac{1}{P}\mathbf{1}_{\mathbf{P},\mathbf{P}})\mathbf{y}(k)$ where $\mathbf{I}_{\mathbf{P},\mathbf{P}}$ and $\mathbf{1}_{\mathbf{P},\mathbf{P}}$ are the $P \times P$ identity and unary matrices, respectively. In this formulation matrices and vectors are denoted by boldface uppercase and lowercase letters, respectively. The corresponding state vector $\mathbf{x}(k)$ is augmented with the accumulated error, which contains the sum of the values of the errors in all steps: $\mathbf{x}(k) = \begin{bmatrix} \mathbf{e}(k) \\ \mathbf{e_I}(k) \end{bmatrix}$

Hence, the two types of state variable, i.e., the error and the accumulated error, are specified according to the formulas below.

$$\mathbf{e}(k) = (\mathbf{I}_{\mathbf{P},\mathbf{P}} - \frac{1}{P}\mathbf{1}_{\mathbf{P},\mathbf{P}})\,(\mathbf{y}(k) + \mathbf{d}^m(k)), \quad \mathbf{e_I}(k+1) = \mathbf{e_I}(k) + \mathbf{e}(k) \tag{5}$$

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

$$\mathbf{y}(k+1) = \mathbf{A}(k)\,\mathbf{y}(k) + \mathbf{B}(k)\,(\mathbf{u}(k) + \mathbf{d}^c(k)) \tag{6}$$

$\mathbf{u}(k)$ is the control input vector, which is a linear function of the system's state, i.e. $\mathbf{u}(k) = -\mathbf{K}\,\mathbf{x}(k)$, $\mathbf{x}(k) = \begin{bmatrix} \mathbf{e}(k) \\ \mathbf{e_I}(k) \end{bmatrix}$. $\mathbf{K}$ is specified by the LQR framework.

The vectors $\mathbf{d}^m, \mathbf{d}^c$ in Eq. (5),(6), according to the control theory terminology, correspond to the *measurement disturbance* and the *control disturbance*, which are changes that affect the measured output and the way the control input influences the measured output, respectively. They can accommodate load and reference input changes, measurement noise as well as modeling inaccuracies.

We assume that the expected completion time depends solely on the control input, i.e., the workload allocation, and the machine load. As such, in Eq. (6), the matrix $\mathbf{A}$ can be omitted. Although in general the system dynamics captured in $\mathbf{A}$ may not be negligible, using relatively large control and sample intervals reduces the effects of dynamics and measurement noise. Moreover, in this work we wanted to go beyond techniques relying on detailed off-line profiling, on the basis of representative types of workloads and/or machines, which are difficult to find in a dynamic context. A generic solution is sought, and our experimental results for several different workloads confirm the control performance of our controller and its robustness to unmodeled dynamics. However, offline or online profiling methods could be integrated into the design to improve the accuracy of the controller. The matrix $\mathbf{B}$ is a time-varying matrix that can be found at runtime through system identification or monitoring. It is a $P \times P$ matrix with elements corresponding 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.

As already explained, 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 can incorporate into the proposed state-space model in two ways. The first proposal is to inject an appropriate correction of the initial estimate of $\mathbf{u}$ using artificial disturbance terms $d_i^m(k)$ in Eq. (6) as follows:

$$d_i^m(k) = \begin{cases} -u_i(k), u_i(k) \leq 0 \\ \frac{u_i(k)}{\sum_{j=1}^{P'} u_j}(1 - \sum_{j=1}^{P'} u_j) \ otherwise \end{cases} \tag{7}$$

where P' is the number of nodes for which the initial LQR decision on $\mathbf{u}$ is a positive number. Alternatively, the equality constraint can be satisfied without the need of $\mathbf{d}^m$ by allowing the controller to specify the allocation only for $P-1$ nodes. For the last machine the allocation will be

$$u_P(k) = 1 - \sum_{j=1}^{P-1} u_j(k) \tag{8}$$

The bound constraints $0 \leq u_i \leq 1$ may be satisfied by careful selection of the LQR parameters.

Remember that Eq. (3) gives the cost function (to be minimized) employed by LQR, which is rewritten here: $\mathcal{J} = \frac{1}{2}\sum_{k=1}^{\infty}\left[\mathbf{x}^{\mathbf{T}}(k)\,\mathbf{Q}\,\mathbf{x}(k) + \mathbf{u}^{\mathbf{T}}(k)\,\mathbf{R}\,\mathbf{u}(k)\right]$. To ensure that $\mathcal{J} \geq 0$ it is required that $\mathbf{Q}$ be *positive semidefinite* and that $\mathbf{R}$ be *positive definite*. This is equivalent to the requirement that the eigenvalues of $\mathbf{Q}$ are all either positive or zero, and the eigenvalues of $\mathbf{R}$ are positive. In passing, the eigenvalues of a matrix are the zeros of its characteristic polynomial.

Having defined the state space model, in order to design an LQR controller, we have to derive a difference equation linking the state variables and the control input, according to the form

$$\mathbf{x}(k+1) = \tilde{\mathbf{A}}\,\mathbf{x}(k) + \tilde{\mathbf{B}}\,\mathbf{u}(k) \tag{9}$$

where $\tilde{\mathbf{A}}$ is a $2N \times 2N$ matrix, and $\tilde{\mathbf{B}}$ is a $2N \times N$ one; N is the number of allocations that are explicitly decided by the controller. A challenge in this step is to ensure that the system is *controllable* from the theoretical point of view (or at least *stabilizable*, i.e., all its uncontrollable states are stable) and stable. Proofs of these properties are discussed in the sequel.

From the combination of the first part of Eq.(5) and (6) , we can derive that $\mathbf{e}(k+1) = (\mathbf{I}_{N,N} - \frac{1}{P}\mathbf{1}_{N,N})\,\mathbf{B}(k)\,(\mathbf{u}(k) + \mathbf{d}^c(k)) \cong (\mathbf{I}_{N,N} - \frac{1}{P}\mathbf{1}_{N,N})\,\mathbf{B}(k)\,\mathbf{u}(k)$

Note also that the second part of Eq. (5) can be rewritten as

$$\mathbf{e}_{\mathbf{I}}(k+1) = \begin{bmatrix} \mathbf{I}_{N,N} & \mathbf{I}_{N,N} \end{bmatrix} \begin{bmatrix} \mathbf{e}(k) \\ \mathbf{e}_{\mathbf{I}}(k) \end{bmatrix} \tag{10}$$

Inserting the two formulas above in Eq. (9), we get

$$\tilde{\mathbf{A}} = \begin{bmatrix} \mathbf{0}_{N,N} & \mathbf{0}_{N,N} \\ \mathbf{I}_{N,N} & \mathbf{I}_{N,N} \end{bmatrix} \quad \text{and} \tag{11}$$

$$\tilde{\mathbf{B}} = \begin{bmatrix} \tilde{\mathbf{I}}_{N,N}\,\mathbf{B}_{N,N} \\ \mathbf{0}_{N,N} \end{bmatrix}, \quad \tilde{\mathbf{I}}_{N,N} = (\mathbf{I}_{N,N} - \frac{1}{P}\mathbf{1}_{N,N}) \tag{12}$$

After the $\tilde{\mathbf{A}}, \tilde{\mathbf{B}}, \mathbf{Q}$ and $\mathbf{R}$ matrices have been specified, toolkits, such as the Control System Toolbox of $MATLAB^{TM}$, provide an automated technique for the estimation of $\mathbf{K}$.

7.2 Controllability and stabilizability

A control system is *fully controllable* if all its states can be driven from their initial condition to any desired final value in finite time, by appropriate unconstrained control actions. This property is crucial in control system design, for if some or all of the states that appear in a state-space model of a system are *uncontrollable*, this can have a serious impact on the convergence and stability of the resulting closed-loop system. *Controllability* is directly related to the *stabilizability* of a system; the latter ensures that even if some uncontrolled states exist, these remain bounded.

In a general state-space setting, *stabilizability* of the pair $[\mathbf{A}, \mathbf{B}]$ is the ability to make the closed-loop system asymptotically stable, i.e. ensuring that the closed-loop system matrix $\mathbf{A}_c = \mathbf{A} - \mathbf{B}\,\mathbf{K}$ has all its eigenvalues inside the unit circle, i.e. $0 \leq \|eig(\mathbf{A}_c)\| < 1$. Since there does not exist a control law to drive any *uncontrollable* states –or *modes*, as they are usually called in control terms– to a desired direction, this implies that a system is *stabilizable* if its uncontrollable states, if any, are stable.

Let us now move to our state-space model of the load balancing problem under a dynamic feedback control architecture –Eqs. (5), (6)– and the two control laws proposed in (7), (8). We will first show that the first proposal leads to an *unstabilizable* system due to the presence of an uncontrollable and unstable state. This is a direct consequence of including all nodes in the design and imposing a common reference input equal to their average response.

The second proposal, which excludes the last node from the design, does not suffer from uncontrollability anymore, hence stability and performance guarantees for the resulting closed-loop system are provided by a dynamic state-feedback control law. An obvious question is whether these properties also hold for the last node, which has been excluded. We also conclude that the answer to this question is positive.

PROPOSITION 7.1. *The state-space model specified in (9), (11), (12) is* uncontrollable *and* unstabilizable. *It possesses an uncontrollable and unstable state that corresponds to an eigenvalue on the unit circle. The reduced state-space model, obtained when one of the nodes is excluded, is* fully controllable.

PROOF. For the state-space model in (9),(11),(12) we denote $\hat{\mathbf{B}} = \tilde{\mathbf{I}}\,\mathbf{B}$ to obtain

$$\tilde{\mathbf{A}} = \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{I} & \mathbf{I} \end{bmatrix}, \quad \tilde{\mathbf{B}} = \begin{bmatrix} \tilde{\mathbf{I}}\,\mathbf{B} \\ \mathbf{0} \end{bmatrix} = \begin{bmatrix} \hat{\mathbf{B}} \\ \mathbf{0} \end{bmatrix}$$

The system's controllability may be easily checked using the controllability matrix

$$P_c = \begin{bmatrix} \tilde{\mathbf{B}} & \tilde{\mathbf{A}}\tilde{\mathbf{B}} & \tilde{\mathbf{A}}^2\tilde{\mathbf{B}} & \ldots & \tilde{\mathbf{A}}^{N-1}\tilde{\mathbf{B}} \end{bmatrix}$$

Our system is *fully controllable* if and only if the rank of the controllability matrix $P_c : 2N \times N^2$ is equal to the dimension of $\tilde{\mathbf{A}}$, i.e. $rank(P_c) = 2N$ (see e.g. [Franklin et al. 1998]). This is equivalent to the existence of $2N$ linearly independent rows or columns in P_c. Due to the special structure of our $\tilde{\mathbf{A}}$, $\tilde{\mathbf{B}}$ matrices, it is easy to show that $\tilde{\mathbf{A}}^k\,\tilde{\mathbf{B}} = \begin{bmatrix} \mathbf{0} \\ \hat{\mathbf{B}} \end{bmatrix}$, $k \geq 1$. This implies that $P_c = \begin{bmatrix} \hat{\mathbf{B}} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \hat{\mathbf{B}} & \hat{\mathbf{B}} & \dots & \hat{\mathbf{B}} \end{bmatrix}$.

In this structure, we have $2N$ linearly independent columns if and only if the $2N \times 2N$ submatrix P_1 is *non singular*, i.e. $\det(P_1) \neq 0$, where $P_1 = \begin{bmatrix} \hat{\mathbf{B}} & \mathbf{0} \\ \mathbf{0} & \hat{\mathbf{B}} \end{bmatrix}$.

Clearly $\det(P_1) = \det(\hat{\mathbf{B}})^2$, and $\det(\hat{\mathbf{B}}) = \det(\tilde{\mathbf{I}})\,\det(\tilde{\mathbf{B}})$. But $\det(\tilde{\mathbf{B}}) = \prod_{i=1}^{N} b_i$, where b_i are the diagonal entries of the diagonal matrix $\mathbf{B}$, hence for a meaningful load balancing setting, in which the computational loads of all nodes are assumed non-zero, it is straightforward to see that full controllability is equivalent to the condition $\det(\tilde{\mathbf{I}}) \neq 0$. Simple determinant manipulations reveal that $\det(\tilde{\mathbf{I}}_{N\times N}) = 0$, $N = P$ and $\det(\tilde{\mathbf{I}}_{N\times N}) = \frac{1}{P}$, $N = P-1$. Hence the full-order system is uncontrollable and the reduced-order system fully controllable.

Moreover, we have $rank(\tilde{\mathbf{I}}_{P\times P}) = P-1$, thus the matrix loses one degree and our system has a single uncontrollable mode. We finally show that this mode corresponds to an eigenvalue at $z=1$, i.e. on the unit circle. This eigenvalue is unstable, therefore the full-order system is also unstabilizable. To show this we need to calculate the eigenvalues of the closed-loop system matrix $\mathbf{A}_c = \tilde{\mathbf{A}} - \tilde{\mathbf{B}}\,\mathbf{K}$. If $\mathbf{K} = [\mathbf{K}_P \;\; \mathbf{K}_I]$ then $\mathbf{A}_c = \begin{bmatrix} -\tilde{\mathbf{I}}\,\mathbf{B}\,\mathbf{K}_P & -\tilde{\mathbf{I}}\,\mathbf{B}\,\mathbf{K}_I \\ \mathbf{I} & \mathbf{I} \end{bmatrix}$ and further manipulations lead to

$$\det(z\mathbf{I} - \mathbf{A}_c) = \det(z(z-1)\mathbf{I} - \tilde{\mathbf{I}}\mathbf{B}[(z-1)\mathbf{K}_P + \mathbf{K}_I])$$

from which we conclude that for any choice of $\mathbf{K}$, $z=1$ is an eigenvalue of the closed loop matrix , since for $z=1$ we have $\det(z\mathbf{I} - \mathbf{A}_c) = -\det(\tilde{\mathbf{I}}\mathbf{B}\mathbf{K}_I) = -\det(\tilde{\mathbf{I}})\,\det(\mathbf{B})\,\det(\mathbf{K}_I) = 0$. $\square$

PROPOSITION 7.2. *Consider the reduced-order system in (9), (11), (12), in which the last node has been excluded. For this system, assume that by designing an appropriate dynamic state-feedback control law while respecting (8) we obtain an asymptotically stable closed-loop system. Then the response of the last node will also be asymptotically stable.*

PROOF. For $N = P-1$ in Eqs. (9),(11),(12) we obtain the state-space equations of a reduced-order system, where $P-1$ integrators are used for the first $P-1$ nodes, and the last node is excluded from the design. According to (8), its control input u_P is specified as $u_P = 1 - \sum_{j=1}^{P-1} u_j$. This relation can be used to specify the link between the error $e_P(k)$ of the P-th node –for which no care to guarantee asymptotic stability has been taken, since no integrator is used– and the errors of the other nodes $e_j(k)$, $j = 1, 2, \dots, P-1$.

Simple manipulations yield the following equations, where the effect of the variables e_P, y_P of the last node is explicitly shown.

$$\begin{aligned} \mathbf{e}(k) &= (\mathbf{I} - \frac{1}{P}\mathbf{1})\,\mathbf{y}(k) - \frac{1}{P} y_P\,\tilde{\mathbf{1}} = \tilde{\mathbf{I}}\,\mathbf{y}(k) - \frac{1}{P} y_P\,\tilde{\mathbf{1}} \\ e_P(k) &= \frac{P-1}{P} y_P - \frac{1}{P}\tilde{\mathbf{1}}^T\,\mathbf{y} \\ \mathbf{e}(k+1) &= \left[\tilde{\mathbf{I}}\,\mathbf{B} + \frac{1}{P} b_P \mathbf{I}\right] \mathbf{u}(k) - \frac{1}{P} b_P\,\tilde{\mathbf{1}}^T, \quad \text{hence} \end{aligned}$$

$$\begin{bmatrix} \mathbf{e}(k+1) \\ \mathbf{e_I}(k+1) \end{bmatrix} = \tilde{\mathbf{A}} \begin{bmatrix} \mathbf{e}(k) \\ \mathbf{e_I}(k) \end{bmatrix} + \tilde{\mathbf{B}}\,\mathbf{u}(k) + \tilde{\mathbf{d}}(k) \quad \text{where} \tag{13}$$

$$\tilde{\mathbf{A}} = \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{I} & \mathbf{I} \end{bmatrix} \quad , \quad \tilde{\mathbf{B}} = \begin{bmatrix} \tilde{\mathbf{I}}\,\mathbf{B} + \frac{1}{P}\,b_P \mathbf{I} \\ \mathbf{0} \end{bmatrix} \quad , \quad \tilde{\mathbf{d}}(k) = \begin{bmatrix} -\frac{1}{P}\,b_P \tilde{\mathbf{I}} \\ \mathbf{0} \end{bmatrix} \tag{14}$$

and $\tilde{\mathbf{1}}$ is an $N \times 1$ unary vector, b_P is the last diagonal entry of matrix B , i.e. $y_P(k+1) = b_P\, u_P(k)$, and all other matrices have dimensions $N \times N$, $N = P-1$.

Eq. (13) reveals the contribution of the last node to the entries of matrix $\tilde{\mathbf{B}}$. We observe also that the last node gives rise to a time-varying disturbance term $\tilde{\mathbf{d}}$, which depends on the computational load.

We have seen in the previous proposition that the system is stabilizable, hence we can design a stabilizing dynamic state-feedback controller to guarantee in the presence of disturbances that all errors $e_j(k)$, $j = 1, 2, \ldots, P-1$ are asymptotically approaching zero, thus all $P-1$ nodes are balanced. However, this does not guarantee that the system is correctly balanced, unless asymptotic stability for the last node is also shown. Further manipulations reveal that

$$\sum_{i=1}^{P-1} e_i = \tilde{\mathbf{1}}^T \mathbf{e} = (\tilde{\mathbf{1}}^T\,\mathbf{1})\,\mathbf{y} - \frac{1}{P} y_P \tilde{\mathbf{1}}^T \tilde{\mathbf{1}} = \frac{1}{P} \tilde{\mathbf{1}}^T \mathbf{y} - \frac{P-1}{P} y_P$$

However, $\frac{1}{P}\tilde{\mathbf{1}}^T\mathbf{y} - \frac{P-1}{P} y_P$ is already proven that it is equal to $-e_P$, i.e., the last node's error is equal to the negative sum of the rest of the errors, thus it tends asymptotically to zero in a stable design, and load balancing is achieved. □