Probabilistic Adaptive Load Balancing for Parallel Queries

Daniel M. Yellin #1, Jorge Buenabad-Chávez *2, Norman W. Paton +3

#IBM Israel Software Lab
Jerusalem Tech Park, Malcha, Building 8, 3rd floor, Jerusalem, Israel 96951
1dmy@us.ibm.com

*Departmento de Computación, CINVESTAV-IPN
Av. Inst. Pol. Nal. 2508
D.F, México 07360
2jbuenabad@cs.cinvestav.mx

+School of Computer Science, University of Manchester
Oxford Rd, Manchester M13 9PL, UK
3npaton@cs.man.ac.uk

***Abstract*— In the context of adaptive query processing (AQP), several techniques have been proposed for dynamically adapting/redistributing processor load assignments throughout a computation to take account of varying resource capabilities. The effectiveness of these techniques depends heavily on *when* and *to what* they adapt processor load assignments, particularly in the presence of varying load imbalance.**

This paper presents a probabilistic approach to decide when and to what to adapt processor load assignments. Using a simulation based evaluation, it is compared to two other approaches already reported. These two approaches are simpler on their decision making than the probabilistic approach, but the latter performs better under several scenarios of load imbalance.

I. Introduction

Parallel query processing is now well established, with most major database vendors providing parallel versions of their products and several distributed query processors exploiting partitioned or pipelined parallelism. Partitioned parallelism has the potential to provide scaleable query processing, but as it performs at the speed of the slowest participant, it has prompted several proposals for adaptivity specifically aimed at load balance (e.g., [1], [2], [3]). These proposals differ from most work on dynamic load balancing for parallel databases (e.g., [4], [5]) in changing load balance within, rather than between, queries. An evaluation and comparison of these approaches can be found in [6], [7].

We assume an underlying adaptive strategy as described by the Flux query operator [3]. This strategy monitors the effectiveness of the current distribution policy (how the data is distributed across a set of machines), and if it determines that for better performance it must change this policy, it stops query processing while the state of the query operators is migrated from heavily loaded machines to less loaded machines. All but one of the strategies reported in [6], [7] stop query processing while adaptation takes place.

Most approaches for deciding when to adapt use only current machine load levels. In this paper we explore a new probabilistic approach that uses past performance (machine load levels) over a period of time to *predict* future performance. This prediction is then used to decide when to adapt. To evaluate this new approach, we compare it to two other approaches. One is the original Flux approach, which adapts when the imbalance level becomes higher than a threshold value; i.e., when the proposed distribution based upon current workloads differs from the current distribution by more than a threshold value. The second approach, inspired by a competitive algorithm [8], adapts when the cost of query processing using the current distribution exceeds the cost of adapting to the proposed workload. In contrast, the probabilistic approach reported here adapts when, according to previous performance history, the *expected cost* of processing in the future using the current distribution will be greater than the expected cost of processing with the proposed distribution, including the cost of adapting the system. This paper analyses and compares these three decision making approaches through a simulation based evaluation. Overall, the probabilistic approach is more stable in the presence of varying imbalance. It bears some similarity to the use of control theory within self-managing databases reported in [9], in that information on previous performance history is used to decide when to adapt, but how such information is used is different.

Section II presents the problem of adapting for load balance in query processing and the Flux adaptivity strategy with its original decision making approach. Section III presents the decision making approach based on a competitive algorithm. Section IV presents the probabilistic decision making approach. Section V presents the experiments conducted and discusses the results. We conclude in Section VI.

II. Adaptive Query Processing for load balancing

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

Load balancing is particularly challenging for stateful operators, such as hash-join, because maintaining a balanced load involves ensuring that (i) the portion of the hash table on each node reflects the required work distribution, and (ii) changes in the work distribution to maintain load balance are preceded by corresponding changes to the hash table on each node. In what follows, we refer to the period during which a hash table is being built as the *build phase* and the subsequent period when it is being accessed as the *probe phase*. Assume that tuples from A have been used to build the hash table and that the probe phase involving B is underway. If a node N_i begins to perform less well (for example, because another job has been allocated to it), maintaining load balance involves reallocating part of the hash table from N_i to other nodes, and ensuring that future tuples from B are sent to the appropriate node. As the hash table for A on node N_i may be large, these movements of state may involve a significant overhead.

A popular approach to implementing parallel query evaluation uses one of the flavors of the *exchange* operator [10] to distribute tuples between parallel plan fragments. Each *exchange* operator has *producer* and *consumer* components, such that each producer is configured to send data to one or more consumers. In essence, each exchange operator reads tuples from its children, and redirects each tuple to a single parent on the basis of a hash function applied to the join attributes. As an even distribution of tuples over query fragments may lead to load imbalance, *exchange* may deliberately send tuples to parent nodes unevenly, using a *distribution policy* to capture the preferred proportion of tuples to be sent to each of the parents.

Sibling-based data redistribution in Flux: when a load imbalance is detected, Flux relocates portions of the hash table from more highly loaded to less highly loaded nodes. An example of before and after states for an adaptation using *Flux* is given in Figure 1. The example shows query evaluation taking place on four nodes, *n1* to *n4*, for the query $A \bowtie B$. The distribution policy of the *exchange_producer* indicates the relative sizes (in terms of numbers of tuples) of the fragments of A and B that should be sent to each of the parent nodes. In the example, in the initial state on the left, 80% of the data should be sent to node *n3* and 20% of the data should be sent to *n4* for joining. If this allocation is found to be distributing work between the nodes in a way that leads to load imbalance, then the distribution policy is updated, and hash table state is transferred between the sibling join partitions. In the figure, the new distribution policy sends 40% of the data to *n3* and *60%* of the data to *n4*, as a result of which part of the existing hash table containing the operator's state has had to be moved from *n3* to *n4*. In the original paper [3], the reallocation is carried out using an operator known as Flux. This behaviour can be broken down into monitoring, assessment and response components as follows:

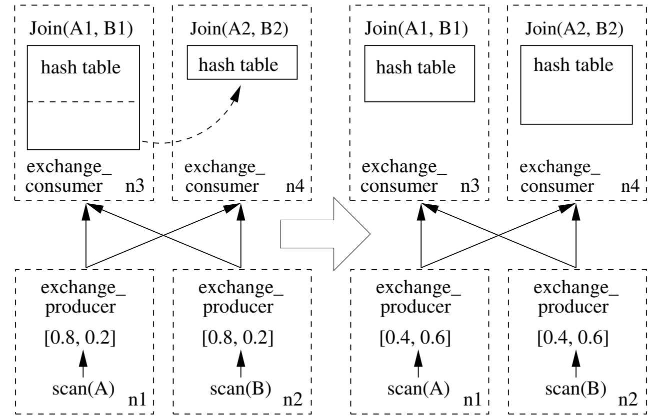

Fig. 1. An example of an adaptation in Flux.

Monitoring: query partitions are monitored to identify the amount of contention for the computational resource on each node and the rate at which data is being processed by the partition. The contention on a node is 0.5 if it is being used half the time, and 2 if 2 jobs are seeking to make full use of the resource at the current time.

Assessment: distribution policies are computed based on the current level of contention in relation to the amount of work allocated. For a partition on node n:

$$proposed_distribution(n) = \\ \quad \textbf{if } contention(n) < 1 \textbf{ then } 1 \\ \quad \textbf{else } 1/contention(n)$$

where $contention(n)$ is the level of contention on node n. In essence, given the level of contention for a node, the definition assumes that if the contention is less than 1, then the complete resource on that node is available for query processing; in contrast, if the contention is greater than 1, then the assumption is that the query will be able to access the resource with equal rights to the other sources of contention[1].

[1]Different heuristics can be used for deriving a $proposed_distribution$. For example, OGSA-DQP [1] uses $rate(p)/contention(n)$, where $rate(p)$ is the current rate at which the partition p is processing tuples, whereas Flux [3] uses the contention directly. The approach described in the text performed better in the experiments than these approaches.

The $proposed_distribution$ values are normalised to sum to 1, and where the proposed distribution differs from the current distribution by more than a threshold (0.05 in the experiments), a response is scheduled.

Response: Each table is considered to consist of a number of fragments (50 in the experiments) which are the unit of redistribution; the number of fragments must be larger than the parallelism level, but each fragment should also be large enough to represent a potentially worthwhile adaptation. Partitions are sorted into two lists: *over_utilised* and *under_utilised*, based on how much work they need to lose or gain, respectively, to conform to the $proposed_distribution$. Fragments are then transferred from each partition in the *over_utilised* list to the corresponding partition in the *under_utilised* list until the change required is smaller than a fragment. The redistribution steps are modelled as taking place in parallel. The resulting data distribution gives rise to a new *distribution policy* for upstream exchange operators. Thus the load is rebalanced by (i) redistributing operator state among the siblings in a way that reflects their throughput; and (ii) updating the *distribution policies* of the upstream exchange operators so that subsequent tuple routing reflects the revised data distribution. To limit the frequency of re-adapting, *Flux* follows this heuristic: when an adaptation has taken place, if it took time *t* to relocate data for use in the hash joins, then no further adaptation is allowed until a further time *t* has passed.

Experiments on the sensitivity of *Flux* to the number of fragments in a table and the threshold that defines the minimum change in distribution policy are reported in [6].

III. Adaptivity based on a competitive algorithm

Flux may perform costly adaptations in response to transient imbalances. In essence, this is because it adapts on the basis of a snapshot of an environment, and adapts to the state of the system when the snapshot is taken. In unstable environments, adaptations will thus be suitable for a very short time. In [6] we used techniques of competitive algorithms with a view to improving the decision as to *when* and *to what* to change the distribution policy in *Flux*. We modified the 3-competitive Delta algorithm originally designed to dynamically select between implementations of components [8], such as web services, to develop adaptable distributed applications.

The revised version of Delta is listed in Figure 2. The idea behind this decision making approach is that adaptation takes place only when the accumulated extra cost of query processing under imbalance reaches the cost of adapting. In the algorithm, an adaptation takes place when *accum_delay* becomes equal to or greater than *switch_cost*; *accum_delay* is the accumulated *extra* cost incurred by the current distribution policy, and *switch_cost* is the cost that would be incurred in adapting from the current distribution policy to a new suggested distribution policy.

The suggested distribution is referred to as the *preferred distribution policy* ($preferred_dp$); it is an average computed over a time period represented by $1..count$,

```
count = 0;
preferred_dp_{n in 1..length(proposed_distribution)}[n]= 0;
current_dp = ... ;
TimeToSwitch = False;
while (not TimeToSwitch)
    count = count + 1;
    Process next portion of query;
    Compute proposed_distribution_{count};
    preferred_dp_{n in 1..length(proposed_distribution)}[n]=
        1/count * sum_{t=1}^{count} (proposed_distribution_t[n]);
    delay = accum_delay(current_dp, preferred_dp, count);
    if delay >= switch_cost
        TimeToSwitch = True;
    endif
endwhile
Switch to use preferred_dp;
```

Fig. 2. Revised Delta.

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

Fig. 3. Computing the accumulated delay.

where each entry in $1..count$ represents a moment during query evaluation, either since start or since last adaptation, when the collection of the monitoring information required to compute a $proposed_distribution$ took place. (The $proposed_distribution$ is computed the same way as for *Flux* described in the previous section.) The distribution policy $current_dp$ is that currently being used[2]. As the $preferred_dp$ is the average of previous proposed distributions, it can be computed efficiently and incrementally. During operation evaluation, $preferred_dp$ is updated at each monitoring point (every 0.1s in the simulation) to take account of the $proposed_distribution$ at that point, and used to compute both the $switch_cost$ and the accumulated delay resulting from the use of the current distribution policy over the period $1..count$.

The accumulated delay is computed as described in Figure 3, which estimates the level of the delay that will result from continuing with *current_dp* compared with changing to *preferred_dp*. In essence, the value of the *i*th entry in *current_dp* represents the fraction of the work that is currently assigned to the *i*th node. By contrast, the value of the *i*th

[2]The notation vector_i[j] (see Figure 2), refers to the jth element of the vector_i recorded at time i.

entry in *preferred_dp* represents the fraction of the work that would have been assigned to that node using the strategy represented by *preferred_dp*. Where *current_dp[i]* is greater than *preferred_dp[i]*, more work was assigned to a node than it was able to process, and thus the node is contributing to sub-optimal performance from the operator. When this is the case, the difference between the distribution policies for the *i*th node represents the portion of the work that the node was unable to evaluate using the current distribution policy.

For example, if *current_dp[i]* is *0.5* and *preferred_dp[i]* is *0.2*, then the total portion of work assigned to the node that it cannot process is *0.3*. We call this difference the *inappropriate assignment level*. The *delay fraction* (*df* in Figure 3) denotes the amount of delay to be incurred by the *i*th node if evaluation continues using *current_dp*. It can be estimated by dividing the *inappropriate assignment level* for node *i* by *preferred_dp[i]*. For the example, this gives (0.3/0.2) = 1.5. That is, the delay resulting from using *current_dp* instead of *preferred_dp* is an additional 1.5 times what it had taken using *preferred_dp*. Thus the result of *accum_delay* is the product of the delay fraction and the *period* for which inappropriate load balance has been in place. The *period* is the number of time units measured by count.

The *switch_cost* in Figure 2 between configurations is variable, as it depends on the amount of data that needs to be relocated for use in the hash joins. Since many unnecessary adaptations can occur when *switch_cost* is relatively low, we retain the minimum threshold value on adaptation size from Section II in *Flux*. Only when this threshold value holds the condition *delay* >= *switch_cost* is tested.

In summary, Revised Delta introduces two fundamental changes to Flux. First, it uses a more sophisticated technique for deciding when to adapt – it must be that the extra accumulated cost incurred by the current distribution policy (in contrast to the preferred distribution policy) is greater than the adaptation cost. Second, instead of using the current workload as the basis for what to adapt to, it uses the average workload, as measured over the period of time since start or last adaptation.

IV. Probabilistic Delta

The Revised Delta algorithm of the last section responds to the consequences of sustained imbalance by accumulating evidence that change is necessary. The change to *preferred_dp* (i.e., the average of previously monitored optimal distributions) often performs well in practise [6]. However, Revised Delta suffers from a flaw in that it only uses historical data to determine when to adapt. It does not use any prediction of future workloads. For instance, consider the case when one machine suddenly experiences a burst of additional workload, but slowly returns to its prior stable state. Revised Delta may, in this case, adapt to a new distribution policy just when the workload is about to resume its prior stable state. This is because Revised Delta averages prior workloads to compute *preferred_dp*, and when Revised Delta sees that *preferred_dp* would have been more efficient than the current workload policy over the monitored period, it switches to *preferred_dp*. If the current monitored workload of a machine is just slightly more than its former stable state, it still may influence *preferred_dp* enough to put it over this threshold, and cause an adaptation.

The probabilistic algorithm of this section, which we call Probabilistic Delta, adapts when the *expected cost* of switching from distribution *current_dp* to *preferred_dp* and processing the remainder of the query using distribution *preferred_dp* is less than the *expected cost* of processing the remainder of the query by continuing to use distribution *current_dp*. Our intuition is that Probabilistic Delta should perform better because it bases its decisions on a probabilistic model of how much gain will be achieved by switching distributions[3].

We compute the expected cost as follows. Let $cost(used_dp, optimal_dp, t)$ be the number of time units it takes to process a query using the distribution $used_dp$ to achieve the same amount of work that the optimal distribution $optimal_dp$ achieves in t time units. Let $prob(d, t)$ be the probability that during the next t time units the observed optimal distribution will be d. The expected cost of $current_dp$ and of $preferred_dp$ are computed thus:

```
// Expected cost of continuing to use current_dp for
// t time units
proc EC_NoChange(current_dp, preferred_dp, t)
return
  cost(current_dp, current_dp, t) * prob(current_dp, t) +
  cost(current_dp, preferred_dp, t) * prob(preferred_dp, t) ;

// Expected cost of changing to preferred_dp and using it for
// t time units
proc EC_Change(preferred_dp, current_dp, t)
return
  cost(preferred_dp, preferred_dp, t) * prob(preferred_dp, t) +
  cost(preferred_dp, current_dp, t) * prob(current_dp, t) +
  switch_cost ;
```

The expected cost of continuing to use $current_dp$ ($EC_NoChange$) has to consider two possibilities: in the future the optimal distribution will be $current_dp$ or it will be $preferred_dp$. The first term, $cost(current_dp, current_dp, t) * prob(current_dp, t)$ is the expected cost of using $current_dp$ given the probability that $current_dp$ is the optimal distribution over the next t time units. The second term, $cost(current_dp, preferred_dp, t) * prob(preferred_dp, t)$ is the expected cost of using $current_dp$ given the probability that $preferred_dp$ is the optimal distribution over this same duration. The expected cost of changing to $preferred_dp$ (EC_Change) is computed in a similar fashion, except we

[3]We briefly recall the concept of *mathematical expectation* on which Probabilistic Delta is based. If the probabilities of obtaining the amounts $a_1, a_2, ...,$ or a_k are $p_1, p_2, ...,$ and p_k, where $p_1 + p_2 + ... + p_k = 1$, then the mathematical expectation is: $a_1p_1 + a_2p_2 + ... + a_kp_k$. For example, if we win £10 when a die comes up 1 or 6, and lose £5 when it comes up 2, 3, 4 or 5, our mathematical expectation (the average in the long run) is: $E = 10 \cdot \frac{2}{6} + (-5) \cdot \frac{4}{6} = 0$.

also have to add in the cost of adaptation (i.e., $switch_cost$). Note that $cost(X, X, t) * prob(X, t) = t * prob(X, t)$.

There are several possible ways to compute the probability $prob(preferred_dp, t)$ and $prob(current_dp, t)$. We used a simple estimate for this probability: we look at the actual distribution d that was observed in each previous time unit (within some window), and label d as either belonging to $preferred_dp$ or $current_dp$, depending on whether it is more similar to $preferred_dp$ or $current_dp$, thus:

```
if max(abs(d - current_dp)) <= max(abs(d - preferred_dp))
  label d as belonging to current_dp
else
  label d as belonging to preferred_dp
endif
```

Let n_p be the number of distributions in the window labelled $preferred_dp$, n_c be the number of distributions in the window labelled $current_dp$, and let n_w be the total number of time units in the window. Then $prob(preferred_dp, t) = n_p/n_w$ and $prob(current_dp, t) = n_c/n_w$.

The function $cost(used_dp, optimal_dp, t)$ is this:

```
proc cost(used_dp, optimal_dp, t)
return t + accum_delay(used_dp, optimal_dp, period) ;
```

To understand the computation of this cost function, recall that *accum_delay(used_dp, optimal_dp, t)* computes the proportion of *extra* work the distribution *used_dp* would incur beyond what distribution *optimal_dp* would incur. In the example given previously, if *used_dp[i]* is 0.5 and *optimal_dp[i]* is 0.2, then there is a slowdown factor of $.3/.2 = 1.5$ incurred by *used_dp* (since, on each unit of time, *used_dp* would process only a fraction of the tuples assigned to it in that time unit, it would require an additional 1.5 times what it had taken using *optimal_dp* to complete its processing). Hence, if *optimal_dp* would finish processing in t time units, *used_dp* would finish processing in $t + accum_delay$ time units. In our example, this would be $t + t * 1.5$. While the *delay* is $t * 1.5$, the actual *total cost* – the total number of time units required to do the processing – would be $t + t * 1.5$. Hence the cost function returns the value $t + accum_delay$.

Probabilistic Delta is listed in Figure 4. The first part up to the comment is the same as that of Revised Delta. Probabilistic Delta, like Revised Delta, considers only two options at every decision point: to continue processing based upon the $current_dp$, or to switch to the $preferred_dp$. They differ in determining if and when to make this adaptation.

```
count = 0;
preferred_dp n in 1..length(proposed_distribution) [n]= 0;
current_dp = ... ;
TimeToSwitch = False;
while (not TimeToSwitch)
  count = count + 1;
  Process next portion of query;
  Compute proposed_distribution_count;
  preferred_dp n in 1..length(proposed_distribution) [n]=
     1/count Σ_{t=1}^{count} (proposed_distribution_t[n]);
  // same as Revised Delta up to here.
  ecNoChange = EC_NoChange(current_dp, preferred_dp, count);
  ecChange = EC_Change(preferred_dp, current_dp, count);
  if ecNoChange >= ecChange
    TimeToSwitch = True;
  endif
endwhile
Switch to use preferred_dp;
```

Fig. 4. Probabilistic Delta.

V. Experimental Evaluation

The paper compares different criteria for decision making aimed at load balancing by simulating query performance using the simulation technique and cost parameters described in [6], [7]. The experiments use query $P \bowtie PS$, henceforth referred to as *Q1*, where P and PS are from the TPC-H database (www.tpc.org) using scale factor 1. All joins are equijoins on foreign keys. The following forms of load imbalance are considered:

1) *Periodic:* The load on one or more of the machines comes and goes during the experiment. In the experiments, the *level*, the *duration* and the *repeat duration* of the external load are varied in a controlled manner; the *level* represents the number of external tasks that are seeking to make full-time use of the machine; the *duration* of the load indicates for how long each load spike lasts; and the *repeat duration* represents the gap between load spikes.
2) *Poisson:* The arrival rate of jobs follows a Poisson distribution [11]. In the experiments, the *average number of jobs starting per second* is varied in a controlled manner: in each clock tick (0.1 second) a random number of jobs from the Poisson distribution with the given average are simulated as having started. The *duration* of each job started is held constant within an experiment.

We evaluate how successful the approaches for decision making presented in the previous sections are at improving query performance in the context of load imbalance. In what follows, *Flux* will refer to its original decision making approach based on Flux [3] (Section II), *Flux-d* to the approach based on Revised Delta (Section III), and *Flux-pd* to the approach based on Probabilistic Delta (Section IV). In each experiment, the three approaches are compared to the *base case* where adaptivity is disabled, called *No Adapt* in the plots.

Experiment 1: *Effectiveness of the decision making approaches in the context of periodic imbalance.* This experiment involves *Q1* being run with a parallelism level of *3*, where an external load is introduced that affects one of the *3* nodes being used to evaluate the join. Figure 5 shows the results. The increasing level of imbalance simulates the effect of having 0 to 6 other jobs competing for the use of one of the compute

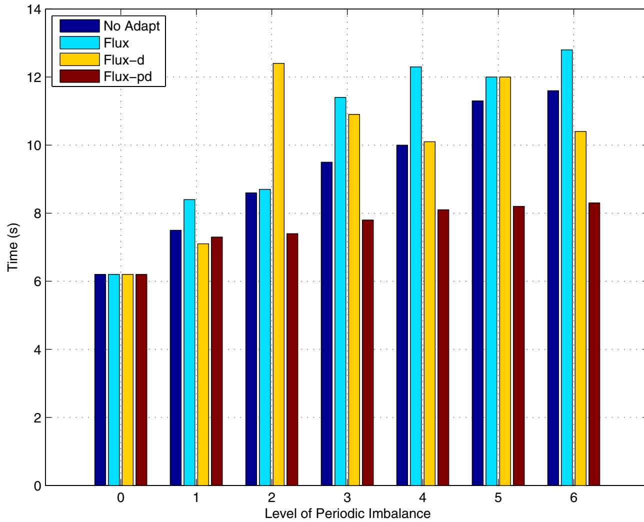

Fig. 5. Experiment 1 – periodic imbalance.

nodes in periodic load spikes of length one second.

The following observations can be made: (i) *Flux* performs less well than Revised Delta and Probabilistic Delta except under imbalance level of *2*, where it performs better than Revised Delta. This is because, under varying imbalance, adapting to snapshots of performance generally involves more costly adaptations than adapting to average levels of imbalance (we discuss this further below). (ii) Probabilistic Delta consistently improves upon the base case, outperforming *Flux* and Revised Delta. This is because the criterion by which Probabilistic Delta decides to adapt is not only the cost of processing under imbalance, as is the case in the other two approaches, but the expected future cost of processing.

We elaborate further with the help of Figure 6. This figure shows, for the level of imbalance *6* in Figure 5, the history of distributions $current_dp$ and $proposed_dp$ in each node running the query under *Flux* and *Flux-d*. For *Flux-pd* (row 3) it shows the history of distributions $current_dp$ and $proposed_dp$ for node 1, but in columns *2* and *3* it depicts the expected cost of using *EC_NoChange* vs using *EC_Change* and *accum_delay* vs *switch_cost*. The x axis is time in all plots; the y axis is the proportion of work assigned and proposed to a node under $current_dp$ and $proposed_dp$. For the plots in columns *2* and *3* for *Flux-pd*, the y axis denotes the cost in time units (0.1s). In each row, the plot on the left (column *1* corresponds to the node being affected by the periodic external load causing the imbalance. Recall that $proposed_dp$ corresponds to the monitored, most convenient distribution at each point in time, and that *Flux* adapts by updating $current_dp$ with the observed-now $proposed_dp$, while Revised Delta and Probabilistic Delta adapt by updating $current_dp$ with $preferred_dp$, which is the average of previous $proposed_dp$s since start or last adaptation.

In each approach, the three nodes start using $current_dp = 1/3$, so the load is initially distributed equally among them, but their $proposed_dp$ is different because the periodic imbalance is present from the start: node *1*, affected by the imbalance, is proposed less than $1/3$ of the total work, while nodes *2* and *3* (in rows *1* and *2*) are proposed more than $1/3$ (the extra work of node *1* is equally divided and proposed to nodes *2* and *3*). While an adaptation takes place, we have plotted $current_dp$ as being equal to 0; thus each down-up line in $current_dp$ represents an adaptation, and its width is how long the adaptation took. For example, we can see that the first adaptation in each approach was very similar, both as to when and to what to adapt: at about 0.3s each approach pauses to adapt and changes the distribution to about 0.1 for node *1* and above 0.4 for nodes *2* and *3*. During this pause $current_dp$ goes down to 0 and stays there while the system adapts for about 0.2s, resuming execution at 0.5s. Thereafter each approach adapts differently: *Flux* tries to follow the changes in imbalance as closely as possible, Revised Delta less so and Probabilistic Delta still less.

We now concentrate on the plots for nodes *1* in each approach, as the two other plots are complementary. Recall observation (i) above, that adapting to snapshots of performance will generally involve more costly adaptations than adapting to average levels of imbalance. The second adaptation by *Flux* and Revised happens about the same time (about 0.14s), but *Flux* then distributes to node *1* as much work as $proposed_dp$ suggests (above 0.2), while Revised Delta distributes to node *1* the average so far (below 0.2). Hence *Flux* has both to move more operator state and to pause longer (about 1s) than Revised Delta (0.2s). The same pattern can be seen in the following adaptations, where the average imbalance is less than the snapshots of performance.

Another difference between *Flux* and Revised Delta is their flexibility as to when to adapt. *Flux* can only adapt when as much time has passed as it took the last adaptation; so it has to wait for a long time if the last adaptation was costly. Revised Delta has to wait for the cost of adaptation to be less than the accumulated delay; but such cost will be low after an adaptation, and thus Revised Delta can adapt quite soon after adapting if need be. Figure 6 shows that Revised Delta incurred 6 adaptations; the fourth and the sixth were just after the previous adaptation finished. So Revised Delta is more flexible as to when to change, but more conservative as to what to change to.

We now recall observation (ii) above, that Probabilistic Delta takes into account how similarly the system has been performing with regard to the alternatives it has: average distribution $preferred_dp$ or the current distribution ($current_dp$). Figure 6 shows that Probabilistic Delta adapts only once, thereafter ignoring the ups and downs of periodic imbalance. Although this single adaptation is to the same distribution that *Flux* and Revised Delta adapt the first time, because the imbalance is present from the start, Probabilistic Delta sticks to it as long as the system does not show a sustained departure from it, thus avoiding the cost of some adaptations. This is because, as the plot at row *3* and column *2* shows, after the first adaptation the expected cost of using (and changing to) $preferred_dp$ grows increasingly higher than

Fig. 6. Detailed history of distributions and expected costs for experiment 1, imbalance level 6.

the expected cost of continuing to use $current_dp$. For the sake of comparison, the plot at row *3* and column *3* shows $accum_delay$ vs $switch_cost$ under Probabilistic Delta. Those are the metrics used by Revised Delta to decide when to adapt. We see that even though the condition $accum_delay \geq switch_cost$ is satisfied throughout the run, the more refined criterion using expected costs employed by Probabilistic Delta is more adequate for this type of imbalance.

Experiment 2: *Effectiveness of the decision making approaches in the context of Poisson imbalance.* This experiment involves *Q1* being run with a parallelism level of *3*, where one of the *3* nodes is affected by an external load that represents jobs arriving with a Poisson distribution, each job having a duration of 1s. Figure 7 shows the results. Poisson-like imbalance is less demanding than periodic imbalance; all the strategies improve over the base case, more so with higher levels of imbalance. As the external load keeps changing, any change in the distribution policy that reflects the reduced capacity of the node subject to the external load does good overall. Revised Delta performs generally better than *Flux* out of changing to the average distribution. Probabilistic Delta

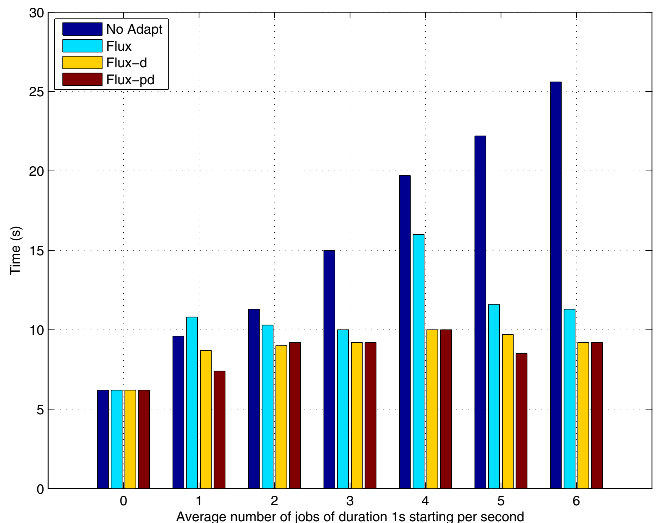

Fig. 7. Experiment 2 – Poisson imbalance.

performs slightly better than Revised Delta in some cases by incurring fewer adaptations.

VI. Conclusions

This paper has presented Probabilistic Delta, a new decision making approach for executing load balancing algorithms during the evaluation of stateful queries with partitioned parallelism. It extends the Revised Delta approach with probabilistic techniques that help achieve good performance under the range of workloads we have tried so far. This is partially achieved because Probabilistic Delta is more resilient to small changes and tends to wait for a sustained departure from a current pattern of use before adapting. However, Probabilistic Delta is sensitive to the window size used for computing expected costs. How to determine an appropriate window size is a topic we are currently investigating.

Revised Delta and Probabilistic Delta have similarity to the use of control theory for self-tuning memory management (STMM) in databases [9], where the size of various memory pools is varied to improve their service time and hence that of the system using a Multiple Input Multiple Output (MIMO) controller. STMM uses a control model, fed with cost-benefit data, to determine how frequently to adapt the size of memory pools, typically between 30 seconds and 10 minutes. This approach is similar to Revised Delta and Probabilistic Delta in that it uses both information on previous performance history and the cost of adaption in order to decide when to adapt. However, in contrast to the cost model used by STMM/MIMO and Revised Delta, the expected cost model used by Probabilistic Delta is intrinsically based upon the prediction of future workloads. More work is needed to compare these approaches.

References

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

[2] V. Raman, W. Han, and I. Narang, "Parallel querying with non-dedicated computers," in *Proc. VLDB*, 2005, pp. 61–72.

[3] M. Shah, J. Hellerstein, S.Chandrasekaran, and M. Franklin, "Flux: An adaptive partitioning operator for continuous query systems," in *Proc. ICDE*. IEEE Press, 2003, pp. 353–364.

[4] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri, "Practical skew handling in parallel joins." in *Proc. VLDB*, 1992, pp. 27–40.

[5] E. Rahm and R. Marek, "Analysis of dynamic load balancing strategies for parallel shared nothing database systems." in *Proc. VLDB*, 1993, pp. 182–193.

[6] N. Paton, J. Buenabad-Chávez, M. Chen, V. Raman, G. Swart, I. Narang, D. Yellin, and A. Fernandes, "Autonomic Query Parallelization using Non-dedicated Computers: An Evaluation of Adaptivity Options," *VLDB Journal*, In press.

[7] N. Paton, V. Raman, G. Swart, and I. Narang, "Autonomic Query Parallelization using Non-dedicated Computers: An Evaluation of Adaptivity Options," in *Proc. 3rd Intl. Conference on Autonomic Computing*. IEEE Press, 2006, pp. 221–230.

[8] D. M. Yellin, "Competitive algorithms for the dynamic selection of component implementations," *IBM Systems Journal*, vol. 42, no. 1, pp. 85–97, 2003.

[9] S. Lightstone, M. Surendra, Y. Diao, S. Parekh, J. Hellerstein, K. Rose, A. Storm, and C. Garica-Arellano, "Control theory: a foundational technique for self managing databases," in *Proc. 2nd Intl. Workshop on Self-Managing Database Systems (SMDB 2007)*, 2007, pp. 395–403.

[10] G. Graefe, "Encapsulation of Parallelism in the Volcano Query Processing System," in *Proc. SIGMOD*, 1990, pp. 102–111.

[11] D. Lilja, *Measuring Computer Performance*. Cambridge University Press, 2000.