DEMB: Cache-Aware Scheduling for Distributed Query Processing *

Junyong Lee[1], Youngmoon Eom[1], Alan Sussman[2], and Beomseok Nam[1]

[1] School of Electrical and Computer Engineering
Ulsan National Institute of Science and Technology,
Ulsan, 689-798, Republic of Korea
[2] Dept. of Computer Science
University of Maryland
College Park, MD 20742, USA

Abstract. Leveraging data in distributed caches for large scale query processing applications is becoming more important, given current trends toward building large scalable distributed systems by connecting multiple heterogeneous less powerful machines rather than purchasing expensive homogeneous and very powerful machines. As more servers are added to such clusters, more memory is available for caching data objects across the distributed machines. However the cached objects are dispersed and traditional query scheduling policies that take into account only load balancing do not effectively utilize the increased cache space. We propose a new multi-dimensional range query scheduling policy for distributed query processing frameworks, called *DEMB*, that employs a probability distribution estimation derived from recent queries. DEMB accounts for both load balancing and the availability of distributed cached objects to both improve the cache hit rate for queries and thereby decrease query turnaround time and throughput. We experimentally demonstrate that DEMB produces better query plans and lower query response times than other query scheduling policies.

1 Introduction

As requirements for computing power increases and high-speed networks become more widespread, cluster computing is rapidly and widely accepted in various disciplines. For high performance data intensive computing applications, a large number of distributed and parallel query processing middleware systems have been developed and employed to solve large, complex scientific problems [14, 8, 15].

In distributed and parallel query processing systems, load balancing plays an important role to maximize overall system throughput by spreading the workload evenly across multiple servers. Besides load balancing, cache hit rate is another critical performance factor that must be accounted for to improve system throughput. However, prior distributed query scheduling policies do not take into consideration load balancing and cache hit rate at the same time. As more servers are added to the distributed query processing system, the amount of cache space available in the distributed servers

* This research was supported by the 1.100027.01 Research Fund of the UNIST(Ulsan National Institute of Science and Technology) and 2.110147.01 National Research Foundation of Korea.

increases linearly. But traditional scheduling policies such as round-robin or load-based scheduling policies do not consider cached objects that may be available in the distributed cache infrastructure, so get little benefit from the larger cache space. Hence we need more intelligent query scheduling policies in order to leverage the availability of a large number of cached objects in a distributed environment.

It is not an easy task to obtain both good load balancing and a high cache hit rate in modern heterogeneous cluster systems, especially if query patterns are also heterogeneous. Our earlier work [12] has shown that either a high cache hit rate with poor load balancing or good load balancing with low cache hit rate fails to maximize system throughput. Another difficult challenge in designing a distributed query scheduling policy is how to make a query scheduler know or predict the availability of cached objects in remote servers. Note that the query scheduling decisions are usually made in a remote front-end server, and that cached objects are evicted dynamically from remote servers, potentially at a high rate (as fast as new data objects are produced). Obviously it is not easy to keep track of cached objects in remote caches. Even if a query scheduler can keep track of them, the amount of information needed in the scheduler will become substantial as the number of servers increases, potentially making the scheduler a performance bottleneck. In order to make the scheduler scalable, the query scheduling policy must be designed to be lightweight.

In our previous work [12] we have proposed a statistical prediction-based query scheduling policy, called DEMA (Distributed Exponential Moving Average) that clusters queries on the fly to increase cache hit rate as well as to balance the query load across servers. The DEMA scheduling policy partitions a set of user queries and evenly distributes them across parallel servers to achieve load balance. DEMA has been shown to produce high cache hit rates since it preserves query locality by grouping similar queries and assigning them to the same server. If the distribution of queries through the query space is static, or slowly changes over time, the DEMA scheduling policy achieves good load balance, however if the query distribution changes rapidly or is extremely skewed, DEMA may suffer from load imbalance as we will discuss further in Section 3.1.

In this paper, we propose an alternative scheduling policy - DEMB (Distributed Exponential Moving Boundary) that overcomes the drawbacks of the DEMA scheduling policy by equally partitioning recent queries using a probability density estimation. Our experimental results show that the new scheduling policy outperforms prior query scheduling policies by a large amount and that it improves upon the DEMA scheduling policy, decreasing response time by up to 50%.

The rest of the paper is organized as follows. In Section 2 we discuss other research related to distributed query scheduling policies. In Section 3 we review the DEMA scheduling algorithm and its drawbacks. In Section 4 we introduce our new query scheduling algorithm, and discuss experimental results from simulations in Section 5. In Section 6 we conclude and discuss future work.

2 Related Work

Load-balancing problems have been extensively investigated in many different fields. Godfrey et. al [5] proposed an algorithm for load balancing in heterogeneous and dynamic peer-to-peer systems. Catalyurek et. al [3] investigated how to dynamically restore balance in parallel scientific computing applications where the computational structure of the applications change over time. Vydyanathan et. al [17] proposed scheduling algorithms that determine what tasks should be run concurrently and how many processors should be allocated to each task. Zhang et. al [20] and Wolf et al. [18] proposed scheduling policies that dynamically distribute incoming requests for clustered web servers. WRR (Weighted Round Robin) [7] is a commonly used, simple but enhanced load balancing scheduling policy which assigns a weight to each queue (server) according to the current status of its load, and serves each queue in proportion to the weights. However, none of these scheduling policies were designed to take into account a distributed cache infrastructure, but only consider the heterogeneity of user requests and the dynamic system load.

LARD (Locality-Aware Request Distribution) [1, 13] is a locality-aware scheduling policy designed to serve web server clusters, and considers the cache contents of back-end servers. The LARD scheduling policy causes identical user requests to be handled by the same server unless that server is heavily loaded. If a server is too heavily loaded, subsequent user requests will be serviced by another idle server in order to improve load balance. The underlying idea is to improve overall system throughput by processing queries directly rather than waiting in a busy server for long time even if that server has a cached response. LARD shares the goal of improving both load balance and cache hit ratio with our scheduling policies, DEMA and DEMB, but LARD transfers workload only when a specific server is too heavily loaded while our scheduling policies actively predict future workload balance and take actions beforehand to achieve better load balancing.

In relational database systems and high performance scientific data processing middleware systems, exploiting similarity of concurrent queries has been studied extensively. That work has shown that heuristic approaches can help to reuse previously computed results from cache and generate good scheduling plans, resulting in improved system throughput as well as reducing query response times [8, 12]. Zhang et al. [19] evaluated the benefits of reusing cached results in a distributed cache framework in a Grid computing environment. In that simulation study, it was shown that high cache hit rates do not always yield high system throughput due to load imbalance problems. We solve the problem with scheduling policies that consider both cache hit rates and load balancing.

In order to support data-intensive scientific applications, a large number of distributed query processing middleware systems have been developed including MOCHA [14], DataCutter [8], Polar* [15], ADR [8], and Active Proxy-G [8]. Active Proxy-G is a component-based distributed query processing grid middleware that employs user-defined operators that application developers can implement. Active Proxy-G employs meta-data directory services that monitor performance of back-end application servers and a distributed cache indexes. Using the collected performance metrics and the distributed cache indexes, the front-end scheduler determines where to assign incoming

queries considering how to maximize reuse of cached objects [12]. The index-based scheduling policies cause higher communication overhead on the front-end scheduler, and the cache index may not predict contents of the cache accurately if there are a large number of queries waiting to execute in the back-end application servers.

3 Distributed Query Processing Scheduling Algorithms

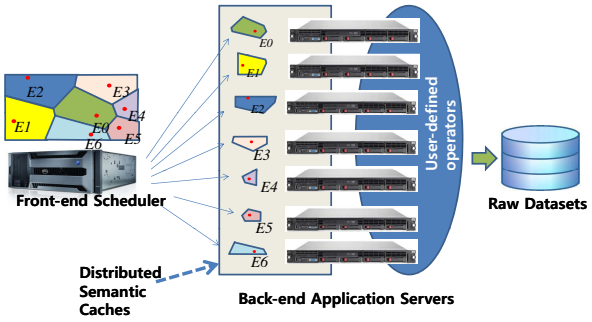


Fig. 1. *Distributed Query Processing Framework with Distributed Semantic Caches*

Figure 1 shows the architecture of Active Proxy-G, a distributed and parallel query processing middleware for scientific data analysis applications. The front-end server runs a query scheduler that determines which back-end application server will process a given query. The homogeneous back-end servers retrieve raw datasets from networked storage systems and process incoming queries on cluster nodes. If the back-end servers are heterogeneous, the scheduling algorithms can be modified with appropriate weight factors.

3.1 DEMA Scheduling Policy

Exponential moving average (EMA) is a well-known statistical method to obtain long-term trends and smooth out short-term fluctuations, which is commonly used to predict stock prices and trading volumes [4]. In general, EMA computes a weighted average of all observed data by assigning exponentially more weight to recent data. The formula that calculates the EMA at time t is given by

$$EMA_t = \alpha \cdot data_t + (1 - \alpha) \cdot EMA_{t-1} \quad (1)$$

where α is the weight factor, which determines the degree of weight decrease over time.

The *Distributed Exponential Moving Average (DEMA)* scheduling policy [12] estimates the cache contents of each back-end server using an exponential moving average (EMA) of past queries executed at that server. In the context of a multidimensional

range query scheduling policy, we use the multidimensional center point of the query as $data_t$ in Equation 1.

Given that application servers replace old cache entries and the DEMA scheduling policy gives less weight to older query entries, the *smoothing factor* $\alpha \in (0, 1)$ must be chosen so that it reflects the degree of staleness used to expunge old data. This implies that α should be adjusted based on the size of the cache space. For example, α should be 1 if a cache can contain only a single query result and α should be close to 0 if the size of the cache is large enough to store all the past query results. However, the number of cached objects in a server can be hard to predict when the sizes of cached objects can vary widely.

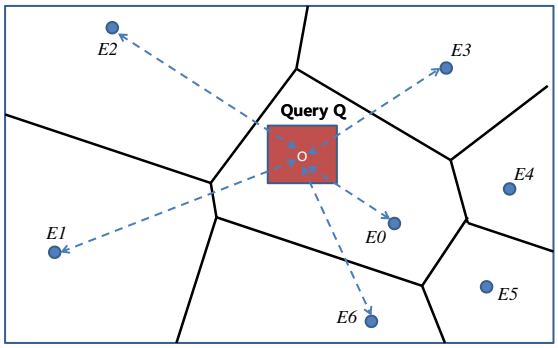


Fig. 2. *The DEMA scheduler calculates the Euclidean distance between EMA points and an incoming query, and assigns the query to the server (0) whose EMA point is closest.*

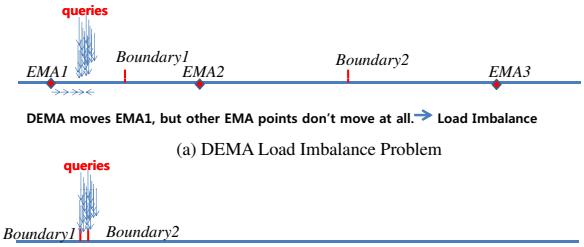
If we can keep track of both the number of current cache entries in each back-end server (k) and the last $k \cdot N$ query center points in the front-end scheduling server (where N is the number of back-end application servers), we can alternatively employ a simple moving average (SMA) instead of EMA, which takes the average of the past k query center points. SMA eliminates the weight-sum error and correctly represent the cache contents of remote back-end application servers. However, SMA does not quickly reflect the moving trend of arriving queries. Furthermore, it causes some overhead in the front-end server to keep track of the last $k \cdot N$ query center points.

In the DEMA query scheduling policy the front-end query scheduler server has one multi-dimensional EMA point for each back-end application server. For the 2 dimensional image, each query specifies ranges in the x and y dimensions, as shown in Figure 2. For an incoming query, the front-end server calculates the Euclidean distance between the center of the multidimensional range query and the EMA points of the N application servers, and chooses an application server whose current EMA point is the closest to that of the incoming query. Clustering similar range queries increases the probability of overlap between multi-dimensional range queries, and increases the cache hit rate at each back-end server. In Figure 2, the given 2 dimensional range query

will be forwarded to server 0 since the EMA point of server 0 is closer to the query than any other EMA point. This strategy is called the *Voronoi assignment model*, where every multidimensional point is assigned to the nearest cell point. The query assignment regions induced from the DEMA query assignment form a *Voronoi diagram* [2].

After the query Q is assigned to the selected application server, the EMA point for that application server is updated as $EMA_{s^*} = \alpha Q + (1 - \alpha)EMA_{s^*}$ (EMA_s represents the EMA point of the sth server). For every incoming query, one EMA point moves in the direction of that query, but we do not need to calculate the changing bisectors of the EMA points since DEMA scheduling algorithm compares the Euclidean distance between EMA points and a given query. The complexity of the DEMA scheduling algorithm is $O(N)$ where N is the number of back-end application servers. So the DEMA scheduling policy is very light-weight.

3.2 Load Balancing



(a) DEMA Load Imbalance Problem

(b) DEMB moves all the boundaries according to new query distribution

Fig. 3. *DEMA Load Imbalance Problem*

The DEMA scheduling algorithm clusters similar queries together so that it can take advantage of cache hits. In addition to a high cache hit rate, the DEMA scheduling algorithm balances query workload across multiple back-end application servers by moving EMA points to hot spots.

Ideally, we want to assign the same number of queries to back-end application servers with a uniform distribution. In DEMA, the probability of assigning a new query to a specific server depends on the query probability distribution and the size of the Voronoi hyper-rectangular cell (e.g., a range in a 1-dimensional line or area for a 2D space).

DEMA balances the server loads by trying to keep the region size inversely proportional to the probability that queries fall inside their region. For the 2D uniform distribution case, as shown in Figure 2, queries that fall inside the Voronoi region of server B's EMA ($Vor(B)$) are assigned to server B. We denote the Voronoi cell of server A's EMA as $Vor(A)$. The probability that a query arrives in a specific Voronoi cell $Vor(A)$ is proportional to the size of the $Vor(A)$. Thus, more queries are likely to land in larger cells than smaller cells for a uniform query distribution.

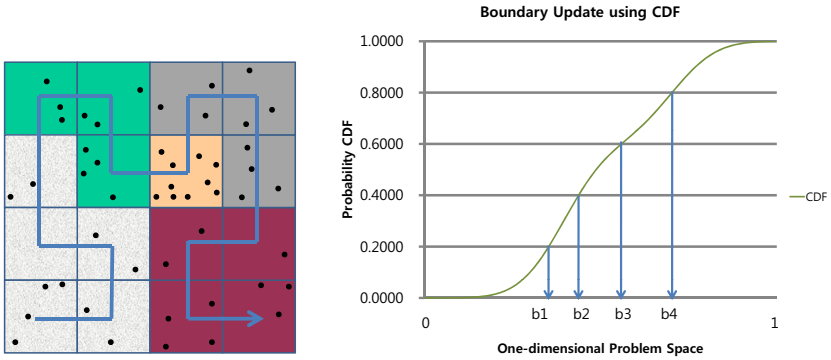
One important property of the DEMA scheduling policy is that an EMA point tends to move to the center of its Voronoi cell if queries arrive with a uniform distribution. In Figure 2, EMA point $E0$ is located in the lower right corner of $Vor(A)$. Since more queries will arrive in the larger part of the cell (i.e. the upper left part of $Vor(A)$), Equation 1 is likely to move the EMA point $E0$ to the upper left part of $Vor(A)$ with higher probability than to the lower right part, which will tend to move the $E0$ toward the center of the cell. That will result in moving the bisectors of $E2$ and $E0$ and the bisector of $E0$ and $E5$ to the left as well. Eventually, this property makes the size of $Vor(E2)$ decrease and the size of $Vor(E5)$ increase. As the size of large Voronoi cells becomes smaller and the size of small Voronoi cells becomes larger, the sizes of the Voronoi cells are likely to converge and effectively the number of queries processed by each back-end application servers will be similar. A similar argument can be made for higher dimensional query spaces.

A normal distribution is another commonly occurring family of statistical probability distributions. It is known that the sum of normally distributed random variables is also normally distributed [6]. If we assume that each historical query point $data_t$ is an individual random variable, the weighted sum EMA_t also should have a normal distribution. Hence the DEMA scheduling policy approximately balances the number of processed queries across multiple back-end application servers even when the queries arrive with a normal distribution.

However the DEMA scheduling policy may suffer from temporary load imbalance if the query distribution changes quickly, with query hot spots moving through the query space randomly. Let us look at an extreme case. Suppose queries have arrived with a uniform distribution, and suddenly all the subsequent queries land in a very small region that is covered by a single Voronoi cell, as illustrated in Figure 3. In such an extreme case, only one corresponding EMA point will move around the new hot spot and the single server will have to process all the queries without any help from other servers. Although this may not commonly occur, we have observed that the DEMA scheduling policy suffers from failing to adjust its EMA points promptly. If a new query distribution spans multiple Voronoi cells, the EMA points slowly adjust their boundaries based on the new query distribution. However the time to adjust depends on the standard deviation of the query distribution. In the next section, we propose an alternative query scheduling policy that solves this load imbalance problem.

4 DEMB: Distributed Exponential Moving Boundary

To overcome the described weakness of the DEMA scheduling policy, we have devised a new scheduling policy called Distributed Exponential Moving Boundary (DEMB) that estimates the query probability density function using histograms from recent queries. Using the query probability density function, the DEMB scheduling policy chooses the boundaries for each server so that each has equal cumulative distribution value. The DEMB scheduling policy adjusts the boundaries of all the servers together, unlike DEMA, so that it can provide good load balancing even for dynamically changing and skewed query distributions.



(a) Mapping cache contents on Hilbert curve

(b) Calculating Boundaries using CDF

Fig. 4. *Calculating DEMB boundaries on Hilbert curve*

In the DEMB scheduling policy, the front-end scheduler manages a queue that stores a predefined number (window size, WS) of recent queries, periodically enumerates those queries, and finds the boundaries for each server by assigning an equal number of queries to each back-end application server.

One challenge in the DEMB scheduling policy is to enumerate the recent multi-dimensional queries and partition them into sub-spaces that have equal probability. In order to map the multi-dimensional problem space onto a one dimensional line, we employ a Hilbert space filling curve [11]. A Hilbert curve is a continuous fractal space-filling curve, which is known to maintain good mapping locality (clustering), i.e. it converts nearby multi-dimensional points to close one-dimensional values.

Using the transformed one dimensional queries, we estimate the cumulative probability density function in the one-dimensional space and partition the space into N sub-ranges so that each one has the same probability as other sub-ranges. The front-end scheduler uses the sub-ranges for scheduling subsequent queries. When a query is submitted, the front-end scheduler converts the center of the query to its corresponding one-dimensional point on the Hilbert curve, determines which sub-range includes the point, and assigns the query to the back-end server that owns the sub-range.

Assigning nearby queries in one-dimensional sub-ranges takes advantage of the Hilbert curve properties. As shown in Figure 4(a), the one dimensional boundaries on the Hilbert curve cluster two dimensional queries so that they have good spatial locality. The DEMB scheduling policy achieves good load balance as well as a high cache hit rate since the scheduler assigns similar numbers of nearby queries on the Hilbert curve to each back-end server.

The DEMB scheduling policy is presented in Algorithm 1. When a new query is submitted to the scheduler, it is inserted into the queue and the oldest query in the queue is evicted, which can change the query probability density function. However if we update the boundaries of each server for every incoming query, that may cause too much overhead in the front-end scheduler. Instead, we employ a sliding window approach,

where the scheduler waits for a predefined number of queries (**update interval**, UI) to arrive before updating the boundaries. Note that the update interval does not have to be equal to the window size. As the update interval is increased, the window size also has to be increased. Otherwise some queries may not be counted when calculating the query probability distribution.

In the DEMB scheduling policy, the window size WS is an important performance factor to estimate the current query probability density. As the window size becomes larger, the recent probability density function can be better estimated. However, if the query distribution changes dynamically, a large window size causes the scheduler to use a large number of old queries to estimate the query probability density function. As a result, the scheduler will not adapt to rapid changes in the query distribution in a timely manner. On the other hand, a small window size can cause a large error in the current query probability distribution estimate due to an insufficient number of sample queries. Moreover, if the window size is smaller than the size of the distributed caches, the query probability density estimation may not correctly reflect all the cached objects in the back-end servers, in addition to the moving trend of query distribution. Therefore choosing an optimal window size under various conditions is one of the most important factors when applying the DEMB scheduling policy to the distributed query processing framework.

In most systems the size of the distributed caches will be much larger than the front-end server's queue size WS. Instead of increasing the window size in order to reduce the error in the probability distribution estimate, we can calculate the moving average of the past query probability distributions, as for the DEMA scheduling policy.

After updating the query probability distribution, we choose the sub-range of each server so that each has equal probability. In Figure 4(b), the problem space is divided into 5 sub-ranges where 20% of the queries are expected for each one. However it is not practical to estimate the real probability distribution function because that requires a large amount of memory to store the histograms. Instead, we assume the query probability density function is a continuous smooth curve. Then we can make the probability density estimation process simpler. The scheduler will determine the boundaries of each server using the WS most recent queries, and apply the following equation to calculate the moving average for each boundary.

$$BOUNDARY[i]_t = \alpha \cdot CUR\_BOUND[i]_t + (1 - \alpha) \cdot BOUNDARY[i]_{t-1}$$

The weight factor α is another important performance parameter in the DEMB scheduling policy, as for DEMA. As described in Section 3.1, $alpha$ determines how earlier boundaries for each server will be considered for the current query probability distribution. However unlike the DEMA scheduling policy, the DEMB scheduling policy has two parameters to control how fast the old queries decay. One is α, and the other is the window size (WS). A large window size (WS) can be used to give more weight to the old queries instead of a low α value.

Now we show an example to see how the DEMB scheduling policy works. Suppose there are 10 back-end servers ($N = 10$), the window size is 500 queries ($WS = 500$), and the boundary update interval is 100 queries ($UI = 100$). The front-end scheduler will replace the oldest query in the queue with the newest incoming query every

Algorithm 1
DEMB Algorithm

procedure
$ScheduleDEMB(Queryq)$

1: INPUT: a client query Q
2: $MinDistance \leftarrow MaxNum$
3: $distance \leftarrow HilbertDistance(query)$
4: **for** $s = 0 \rightarrow N - 1$ **do**
5: **if** BOUNDARY[s] is not initialized **then**
6: forward query Q to server s.
7: $BOUNDARY[s] \leftarrow HilbertDistance(Q)$
8: return
9: **else**
10: **if** $s = 0 \bigwedge distance \leq BOUNDARY[0]$ **then**
11: $selectedServer \leftarrow 0$
12: **else if** $BOUNDARY[s-1] < distance \bigwedge distance \leq BOUNDARY[s]$ **then**
13: $selectedServer \leftarrow s$
14: **end if**
15: **end if**
16: **end for**
17: forward query Q to $SelectedServer$.
18: **if** $QueryQueue.size() < WindowSize$ **then**
19: $QueryQueue.enqueue(Q)$
20: **if** $QueryQueue.size()\%N = 0$ **then**
21: $UPDATE(QueryQueue)$
22: **end if**
23: **else**
24: $QueryQueue.dequeue()$
25: $QueryQueue.enqueue(Q)$
26: **if** $intervalCount = UpdateInerval$ **then**
27: $UPDATE(QueryQueue)$
28: $intervalCount \leftarrow 0$
29: **end if**
30: **end if**
31: $intervalCount \leftarrow intervalCount + 1$

end procedure

procedure
$UPDATE(Queue\ QueryQueue)$

1: INPUT: a queue that stores recent queries $QueryQueue$
2: $LOAD \leftarrow QueryQueue.getSize()/N$
3: **for** $i = 1 \rightarrow N - 1$ **do**
4: $CUR\_BOUND[i] \leftarrow (HilbertDistance(LOAD \times i) + HilbertDistance(LOAD \times i + 1))/2$
5: **end for**
6: **for** $i = 1 \rightarrow N - 1$ **do**
7: $BOUNDARY[i] \leftarrow alpha * CUR\_BOUND[i] + (1 - alpha) * BOUNDARY[i]$
8: **end for**

end procedure

time a new query arrives. The incoming queries will be forwarded to one of the back-end servers using the boundaries of each server ($BOUNDARY[i]$). When the 100th query arrives, the scheduler recalculates the boundary for each server using the past 500 queries. Since there are 10 back-end servers, each server should process 50 queries to achieve perfect load balance. Hence, the new boundary between the 1st server and the 2nd server should be the middle point of the 50th query and the 51st query in the Hilbert curve ordering. Likewise, the new boundary ($CUR\_BOUND[i]$) between the ith server and the $i+1$th server should be the middle point of the $50 * i$th query and the $50 \cdot i + 1$th query. After calculating the new boundaries ($CUR\_BOUND[i]$) for the back-end servers, we compute the moving average for each boundary. If the query distribution has changed, $BOUNDARY[i]$ would move to $CUR\_BOUND[i]$. In this way, the DEMB scheduling policy makes the boundaries move according to the new query distribution.

The cost of the DEMB scheduling policy is determined by the number of servers and the level of recursion for the Hilbert curve, i.e. $O(N \cdot HilbertLevel)$. The complexity of the Hilbert curve is determined by the level of recursion in the Hilbert curve, which is usually a very small number. With a higher level of recursion, a Hilbert space filling curve can map a larger number of points onto it. For example, for a level 15 Hilbert curve about 1 billion (4^{15}) points can be mapped to distinct one-dimensional values. In our experiments, we set the level of the Hilbert curve to 15, and employed at most 50 servers. The cost of DEMB scheduling is very low, but is somewhat higher than that of DEMA, which is $O(N)$.

5 Experiments

5.1 Experimental Setup

The primary objective of the simulation study is to measure the query response time and system load balance of the DEMB scheduling policy with various query distributions. To measure the performance of query scheduling policies, we generated synthetic query workloads using a *spatial data generator*, which is available at [16]. It can generate spatial datasets with normal, uniform, or Zipf distributions. We have generated a set of queries with various distributions with different average points, and combined them so that the distribution and hot spots of queries move to different location unpredictably. We will refer to the randomly mixed distribution as the *dynamic distribution*. We also employed a Customer Behavior Model Graph (CBMG) to generate realistic query workloads [10]. CBMG query workloads have a set of hot spots. CBMG chooses one of the hot spots as its first query, and subsequent queries are modeled using spatial movements, resolution changes, and jumps to other hot spots. The query's transitions are characterized by a probability matrix. In the following experiments, we used the synthetic dynamic query distribution and a CBMG generated query distribution, with each of them containing 40,000 queries, and a cache miss penalty of 400 ms. This penalty is the time to compute a query result from scratch on a back-end server. We are currently implementing a terabyte scale bio-medical image viewer application on top of our distributed query scheduling framework. In this framework, a large image is partitioned into small equal-sized chunks and they are stored across distributed servers.

Obviously the cache miss penalty is dependent on the size of the image chunks. When a cache miss occurs, the back-end server must read raw image file data from disk storage and generate an intermediate compressed image file at the requested resolution. The 400 ms cache miss penalty is set based on this scenario. We also evaluated the scheduling policies with smaller cache miss penalties, but the results were similar to the results described below.

5.2 Experimental Results - DEMB

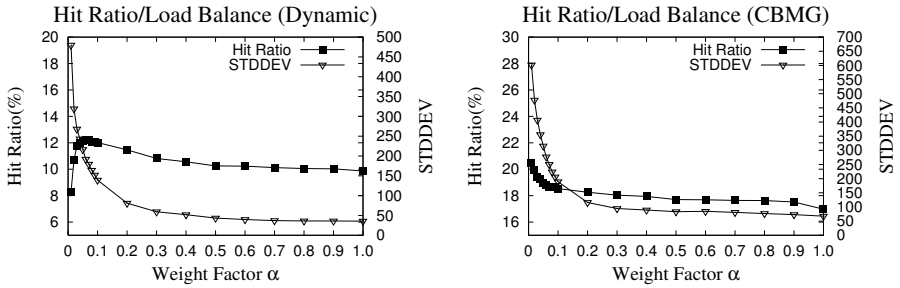
Using various experimental parameters, we measured (1) the query response time, the elapsed time from the moment a query is submitted to the system until it completes, (2) the cache hit rate, which shows how well the assigned queries are clustered, and (3) the standard deviation of the number of processed queries across the back-end servers, to measure load balancing, i.e. lower standard deviation indicates better load balancing. In the following set of experimental studies, we focused on the performance impact of 3 parameters - window size (WS), EMA weight factor (α), and update interval (UI).

Weight Factor: The weight factor α is one of the three parameters that we can control to determine how fast the query scheduler loses information about past queries. The other two parameters are the *update interval* UI and *window size* WS. WS is the number of recent queries that front-end scheduler keeps track of. The update interval UI determines how frequently new boundaries should be calculated in the scheduler with weight factor α. As the weight factor α becomes larger, the scheduler will determine the boundaries using more recent queries. As UI becomes shorter, α will be applied to boundary calculations more frequently. Hence older queries will have less impact on the calculation of boundaries. Also, if WS is small, only a few of the most recent queries will be used to calculate the boundaries.

For the experiments shown in Figure 5 and 6, we employed 50 servers, and fixed the window size to a small number (200), i.e. only 4 recent queries are used to calculate the boundaries of each back-end server. Since the window size is small, we updated the boundaries of each server whenever 10 new queries arrive. These numbers are chosen to minimize the effects of the weight factor α on query response time.

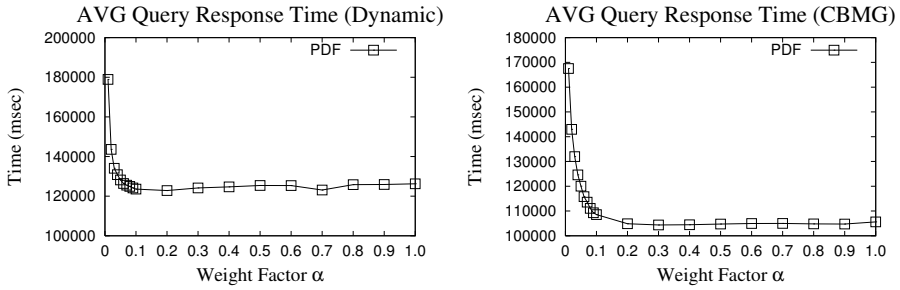
With smaller α, the boundaries move more slowly. If α is 0, the boundaries will not move at all. For both the dynamic query distribution shown in Figure 5(a) and the CBMG distribution shown in Figure 5(b), load balancing (STDDEV) becomes worse as we decrease α because the boundaries of the back-end application servers fail to adjust to the changes in the query distribution.

However the cache hit rate slightly increases from 16.9% to 20.4% as we decrease α for the CBMG query distribution because the CBMG query pattern is rather stationary and a small α value, close to 0, makes the boundaries fixed, which increases the probability of reusing cached objects. However for the dynamic query distribution stationary boundaries decrease the probability of cache reuse, hence the cache hit rate decreases from 12.2% to 8.1% as we decrease α. Figure 6 shows the average query response time determined by the cache hit rate and load balancing. As we decrease α, the query response time increases exponentially since it hurts overall system load balance greatly although it improves the cache hit rate slightly.



(a) Dynamic (b) CBMG

Fig. 5. *Cache Hit Rate and Load Balancing with Varying Weight Factor*



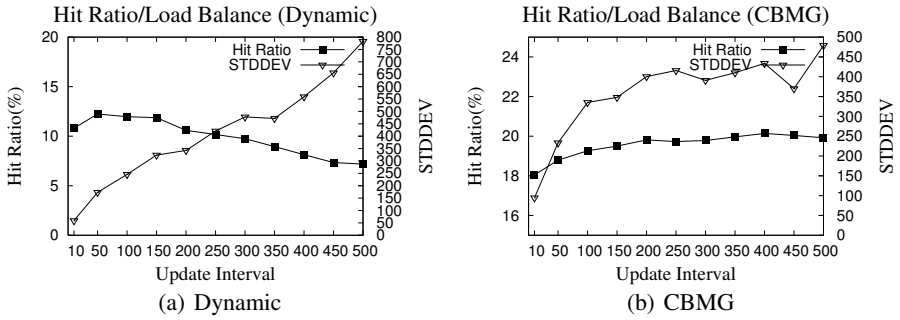
(a) Dynamic (b) CBMG

Fig. 6. *Query Response Time with Varying Weight Factor*

Both leveraging cached results and achieving good load balance are equally important in maximizing overall system throughput. However we note that a very small improvement in cache hit rate might be more effective in improving average query response time than a large standard deviation improvement in certain cases. In Figure 5(a), the load balancing standard deviation is approximately 3 times higher when the weight factor is 0.1 than it is 0.3. But the average query response times shown in Figure 6(a) are similar because the cache hit rate is slightly higher when the weight factor is 0.1.

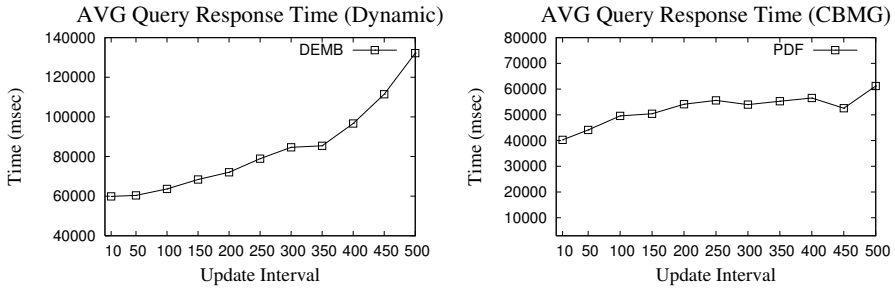
Update Interval: How frequently the scheduler updates the new boundaries is another critical performance factor in the DEMB scheduling policy, since frequent updates will make the boundaries of servers more quickly respond to recent changes in the query distribution. However frequent updates may cause large overheads in the front-end scheduler, and may not be necessary if the query distribution is stationary. Hence the update interval should be chosen considering the trade-off between reducing scheduler overhead and making the scheduling policy responsive to changes in the query distribution.

In the experiments shown in Figures 7 and 8, we measured the performance of the DEMB scheduling policy varying the update interval. In this set of experiments, the α and WS were fixed to 0.3 and 200, respectively. As we decrease the update interval,



(a) Dynamic (b) CBMG

Fig. 7. *Cache Hit Rate and Load Balancing with Varying Update Interval*



(a) Dynamic (b) CBMG

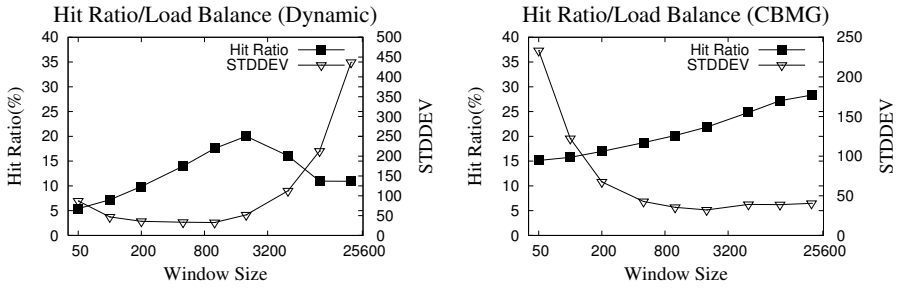
Fig. 8. *Query Response Time with Varying Update Interval*

the boundaries are updated more frequently and they reflect the recent query distribution well. As a result, the DEMB scheduling policy takes good advantage of clustering and load balancing for the dynamic query distribution, as shown in Figure 7(a). As the update interval increases, the boundaries move more slowly and DEMB suffers from poor load balancing and cache misses.

With the stationary CBMG queries shown in Figure 7(b), the cache hit rate does not seem to be affected by the update interval, but the standard deviation increases slightly, although not as much as for the dynamic query distribution. This results indicate that we should update the boundaries frequently as long as that does not cause significant overhead in the scheduler.

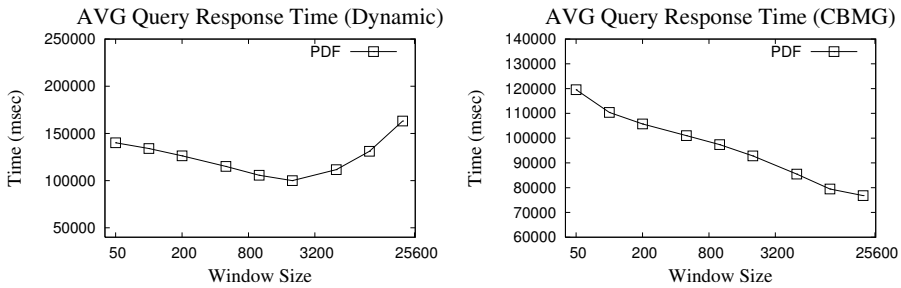
Window Size: The front-end scheduler needs to store the recent queries in a queue so that they can be used to construct query distribution and determine the boundaries of back-end servers. With a larger window size (more queries), the front-end scheduler can estimate query distribution more accurately. Also, a larger WS allows a query to stay in the queue longer, i.e. the same queries will be used more often to construct query histograms and boundaries. On the other hand, a small WS makes the front-end scheduler uses a smaller number of recent queries to determine the boundaries, and the

query distribution estimate is more likely to have large errors. In the experiments shown in Figures 9 and 10, we measured performance with various window sizes. The weight factor α and the update interval UI were both fixed to 1, in order to analyze the effects of only WS.



(a) Dynamic (b) CBMG

Fig. 9. *Cache Hit Rate and Load Balancing with Varying Window Size*



(a) Dynamic (b) CBMG

Fig. 10. *Query Response Time with Varying Window Size*

For the dynamic query distribution, the cache hit rate increases from 5.4% to 20% and the standard deviation decreases slowly as the window size increases up to 1000. That is because the scheduler estimates query distribution more accurately with a larger number of queries. However, if the window size becomes larger than 1000, both the cache hit rate and load balancing suffer from inflexible boundaries. Note that WS also determines how quickly the boundaries change as the query distribution changes. A large window size makes the scheduler consider a large number of old queries so will make the boundaries between servers change slowly. If the query distribution changes rapidly, a smaller window size allows the scheduler to quickly adjust the boundaries. For the CBMG query distribution, both the cache hit rate and load balancing improves as

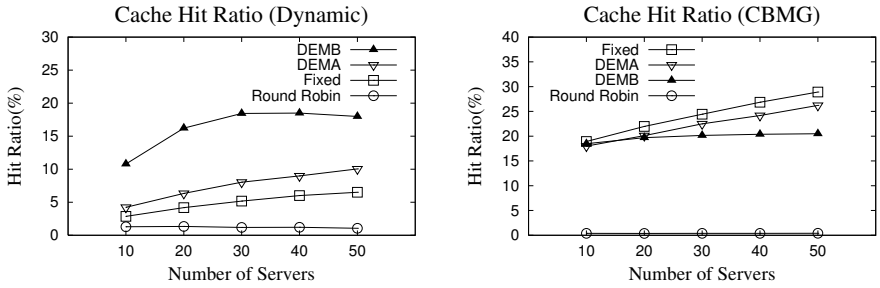
the window size increases. This is because the CBMG queries are relatively stationary over time. Hence a longer record of past queries helps the scheduler to determine better boundaries for future incoming queries.

These window size experimental results show that we need to set window size as large as possible unless it hurts the flexibility of the scheduler. Note that the window size should be orders of magnitude larger than the number of back-end servers. Otherwise, the boundaries set from a small number of queries would have very large estimation errors. For example, if the window size is equal to the number of back-end servers, the boundaries will be simply the middle point of the sorted queries, which would make the boundaries jump around the problem space. However a large window size has a large memory footprint in the scheduler, so can cause higher computational overhead in the scheduler, and the same is true for the update interval. In order to reduce the overhead from large window sizes, but to prevent estimation errors from making the boundaries move around too much, the scheduler can decrease the weight factor α instead, which will give higher weight to older past boundaries and smooth out short term estimation errors.

5.3 Comparative Study

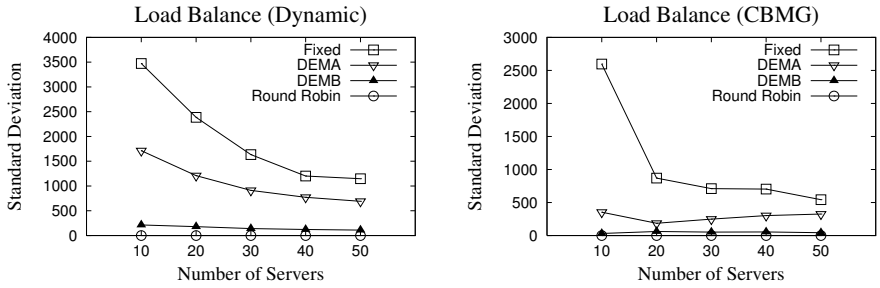
In order to show that the DEMB scheduling policy performs well compared to other scheduling policies, we compared it with three other scheduling policies - round-robin, Fixed, and DEMA. For this set of experiments, we employed 50 back-end application servers and a single front-end server. In order to measure how the other scheduling policies behave under different conditions, we used both the dynamic and CBMG query distributions. The parameters for the DEMB scheduling were set to good values from the previous experiments - the weight factor α is 0.2, the update interval is 500, and the window size is 1000. These numbers are not the best parameter values we obtained from the previous experiments, but we will show that the DEMB scheduling policy shows good performance even without the optimal parameter values. Figures 11, 12, and 13 show the cache hit rate, load balance, and query response time for the different scheduling policies.

The Fixed scheduling policy partitions the problem space into several sub-spaces using the query probability distribution of the initial N queries, similar to the DEMB scheduling policy (where N is the number of servers), and each server processes subsequent queries that lie in its sub-space. But the sub-spaces are not adjusted as queries are processed, unlike DEMB. When the query distribution is stable, Fixed scheduling has a higher cache hit rate than round-robin since it takes advantage of spatial locality in the queries while round-robin does not. Since the Fixed scheduling policy does not change the boundaries of sub-spaces once they are initialized, it obtains a higher cache hit rate than the DEMA or DEMB scheduling policies for certain experimental parameter settings, as shown in Figure 11(b). For the CBMG query distribution, there are 200 fixed hot spots and the servers that own those hot spots and have large cache spaces obtain very high cache hit rates. However, when the query distribution changes dynamically the cache hit rate for the Fixed scheduling policy drops significantly, and is much lower than that of the DEMA and DEMB scheduling policy.



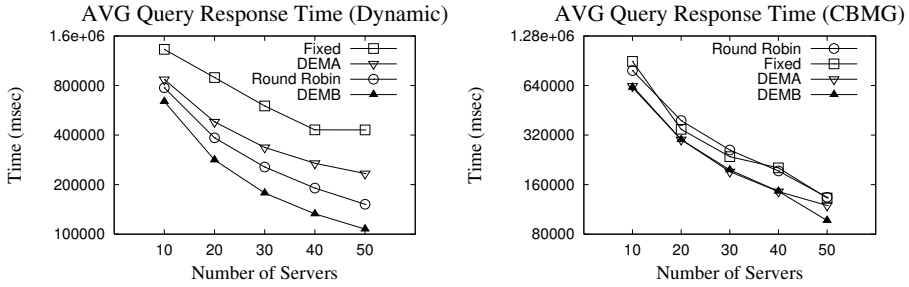
(a) Dynamic (b) CBMG

Fig. 11. *Cache Hit Rate*



(a) Dynamic (b) CBMG

Fig. 12. *Load Balance*



(a) Dynamic (b) CBMG

Fig. 13. *Query Response Time*

Although the Fixed scheduling policy outperforms DEMA and DEMB as measured by cache hit rate when the query distribution is stable, the Fixed policy suffers from serious load imbalance. Some servers process many fewer queries than others, if their subspaces do not contain hot spots. Therefore the standard deviation of the Fixed scheduling is the worst of all the scheduling policies, as shown in Figure 12, and the load

imbalance problem becomes more especially bad for small numbers of servers. Due to its poor load balancing, the average query response time for the Fixed scheduling is higher than for the other scheduling policies, as shown in Figure 13. Note that the query response time is shown on a log scale.

For the dynamic query distribution, the DEMB scheduling policy is superior to the other scheduling policies for all the number of servers in terms of the query response time. DEMB's cache hit rate is about twice that of DEMA - the second best scheduling policy. The DEMA scheduling policy has a lower cache hit rate than DEMB because DEMA slowly responds to rapid changes in the incoming query distribution, as we described in Section 3.1. Note that the DEMA scheduling policy adjusts only a single EMA point per query, while DEMB adjusts all the servers' boundaries at every update interval.

However, for the CBMG query distribution, the quick response to the changed query distribution seems to lower the cache hit rate of DEMB somewhat. As we mentioned earlier, the DEMB scheduling policy is designed to overcome the drawback of the DEMA scheduling policy, which is seen most when the query distribution changes dynamically. In CBMG distribution the query distribution pattern is stable, and both the DEMA and the DEMB scheduling policies perform equally well. In Figure 11(b), the cache hit rate for DEMB is lower than for the DEMA scheduling policy because the boundaries determined by DEMB are likely to move rapidly, since some spatial locality for the queries is lost. When the boundaries are adjusted rapidly based on short term changes from a small number of recent queries, that may improve load balance but it will decrease the cache hit rate when the long term query distribution is stable. Due to the DEMB scheduling policy's fast response time, its load balancing performance is similar to that of the round-robin scheduling policy. The DEMA scheduling policy also balances server load reasonably well for the CBMG query distribution, but for the dynamic query distribution DEMA suffers from load imbalance due to its slow adjustment of EMA points.

Figure 13 shows that the average query response time improves as we add more servers to the system. The DEMB scheduling policy outperforms all the other scheduling policies for the dynamic query distribution. For the CBMG query distribution, the DEMB query response time is the lowest in most cases. For 50 servers, DEMB shows the best performance although its cache hit rate is not the highest. This result shows that load balancing plays an important role in large scale systems. However load balancing itself is not the only factor in overall system performance because round-robin does not show good performance.

5.4 Automated Parameter (WS) Adjustment

In the DEMB scheduling policy, the three parameters (weight factor, update interval, and window size) determine how fast the boundaries of each server adjust to the new query distribution. As seen in Figure 8, the update interval (UI) should be set close to 1 to be more responsive, i.e. the boundaries of servers must be updated for every incoming query. The weight factor (α) also determines how much weight is given to recent queries so that the boundaries adjust to the new distribution. However, when the window size (WS) is large enough, the current query distribution already captures

recent changes in query distribution and the weight factor (α) does not affect query response time significantly. If the window size is not much greater than the number of backend servers (within a factor of 2 or 3), the query response time is not as resilient to changes in the weight factor when α is larger than 0.1, as shown in Figure 6. This is because the large window size (WS) and the small update interval (UI) makes a single query counted for multiple (WS) times (such as in a sliding window) to determine the boundaries for each server. Hence, the most effective performance parameter that system administrators can tune for the DEMB scheduling policy is the window size (WS) .

As shown in Figure 10, a larger window size yields lower query response times when the query distribution is stable. But if the query distribution changes rapidly, the window size must be chosen carefully to reduce the query response time. In order to automate selecting a good window size based on changes in the query distribution, we make the scheduler compare the current query distribution histogram with a moving average of the past query distribution histogram using Kullback-Leibler divergence, which is a measure of the difference between two probability distributions [9]. If the two distributions differ significantly the scheduler decreases the window size so that updates to boundaries happen more quickly. When the two distributions are similar, the scheduler gradually makes the window size bigger (but no bigger than a given maximum size), which makes the boundaries more stable over time to achieve a higher cache hit rate. In experiments not shown due to page limitation, we have observed that this automated approach improves query response time significantly.

6 Conclusion and Future Work

In this paper we have described and compared distributed query scheduling policies that take into consideration the dynamic contents of a distributed caching infrastructure with very little overhead ($O(N \times HilbertLevel)$.

In distributed query processing systems where the caching infrastructure scales with the number of servers, both leveraging cached results and achieving good load balance are equally important in maximizing overall system throughput. In order to achieve load balancing as well as to exploit cached query results, such a system must employ more intelligent query scheduling policies than the traditional round-robin or load-monitoring scheduling policies.

In this context, we proposed the DEMB scheduling policy that clusters similar queries to increase the cache hit rate and assigns approximately equal numbers of queries to all servers to achieve good load balancing. We experimentally demonstrate that DEMB produces better query plans that provide much lower query response times than traditional query scheduling policies.

References

1. Aron, M., Sanders, D., Druschel, P., Zwaenepoel, W.: Scalable content-aware request distribution in cluster-basednetwork servers. In: Proceedings of Usenix Annual Technical Conference (2000)

2. de Berg, M., Cheong, O., van Kreveld, M., Overmars, M.: Computational Geometry, Algorithms and Applications. Springer (1998)
3. Catalyurek, U.V., Boman, E.G., Devine, K.D., Bozdag, D., Heaphy, R.T., Riesen, L.A.: A repartitioning hypergraph model for dynamic load balancing. Journal of Parallel and Distributed Computing 69(8), 711–724 (2009)
4. lun Chou, Y.: Statistical Analysis. Holt International (1975)
5. Godfrey, B., Lakshminarayanan, K., Surana, S., Karp, R., Stoica, I.: Load balancing in dynamic structured p2p systems. In: Proceedings of INFOCOM 2004 (2004)
6. Grinstead, C.A., Snell, J.L.: Introduction to Probability. American Mathematical Society (1997)
7. Katevenis, M., Sidiropoulos, S., Courcoubetis, C.: Weighted round-robin cell multiplexing in a general-purpose atm switch chip. IEEE Journal on Selected Areas in Communications 9(8), 1265–1279 (1991)
8. Kim, J.S., Andrade, H., Sussman, A.: Principles for designing data-/compute-intensive distributed applications and middleware systems for heterogeneous environments. Journal of Parallel and Distributed Computing 67(7), 755–771 (2007)
9. Kullback, S., Leibler, R.A.: On information and sufficiency. Annals of Mathematical Statistics 22(1), 79–86 (1951)
10. Menasce, D.A., Almeida, V.A.F.: Scaling for E-Business: Technologies, Models, Performance, and Capacity Planning. Prentice Hall PTR (2000)
11. Moon, B., Jagadish, H.V., Faloutsos, C., Saltz, J.H.: Analysis of the clustering properties of the hilbert space-filling curve. IEEE Transactions on Knowledge and Data Engineering 13(1), 124–141 (2001)
12. Nam, B., Shin, M., Andrade, H., Sussman, A.: Multiple query scheduling for distributed semantic caches. Journal of Parallel and Distributed Computing 70(5), 598–611 (2010)
13. Pai, V., Aron, M., Banga, G., Svendsen, M., Druschel, P., Zwaenepoel, W., Nahum, E.: Locality-aware request distribution in cluster-based network servers. In: Proceedings of ACM ASPLOS (1998)
14. Rodríguez-Martínez, M., Roussopoulos, N.: MOCHA: A self-extensible database middleware system for distributed data sources. In: Proceedings of 2000 ACM SIGMOD
15. Smith, J., Sampaio, S., Watson, P., Paton, N.: The polar parallel object database server. Distributed and Parallel Databases 16(3), 275–319 (2004)
16. Theodoridis, Y.: R-tree Portal, *http://www.rtreeportal.org*
17. Vydyanathan, N., Krishnamoorthy, S., Sabin, G., Catalyurek, U., Kurc, T., Sadayappan, P., Saltz, J.: An integrated approach to locality-conscious processor allocation and scheduling of mixed-parallel applications. IEEE Transactions on Parallel and Distributed Systems 15, 3319–3332 (2009)
18. Wolf, J.L., Yu, P.S.: Load balancing for clustered web farms. ACM SIGMETRICS Performance Evaluation Review 28(4), 11–13 (2001)
19. Zhang, K., Andrade, H., Raschid, L., Sussman, A.: Query planning for the Grid: Adapting to dynamic resource availability. In: Proceedings of the 5th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid). Cardiff, UK (May 2005)
20. Zhang, Q., Riska, A., Sun, W., Smirni, E., Ciardo, G.: Workload-aware load balancing for clustered web servers. IEEE Transactions on Parallel and Distributed Systems 16(3), 219–233 (2005)