EM-KDE: A Locality-Aware Job Scheduling Policy with Distributed Semantic Caches

Youngmoon Eom [a] Deukyeon Hwang [a] Junyong Lee [a]
Jonghwan Moon [a] Minho Shin [b] Beomseok Nam [a]

[a] *School of Electrical and Computer Engineering, UNIST Supercomputing Center, Ulsan National Institute of Science and Technology, Ulsan, Republic of Korea*

[b] *Dept. of Computer Engineering, Myongji University, Yongin, Gyonggido, Republic of Korea*

Keyword Locality-aware scheduling; Distributed semantic cache; Distributed scheduling; Parallel multi-dimensional range query;

Abstract

In modern query processing systems, the caching facilities are distributed and scale with the number of servers. To maximize the overall system throughput, the distributed system should balance the query loads among servers *and* also leverage cached results. In particular, leveraging distributed cached data is becoming more important as many systems are being built by connecting many small heterogeneous machines rather than relying on a few high-performance workstations. Although many query scheduling policies exist such as round-robin and load-monitoring, they are not sophisticated enough to both balance the load and leverage cached results. In this paper, we propose distributed query scheduling policies that take into account the dynamic contents of distributed caching infrastructure and employ statistical prediction methods into query scheduling policy.

We employ the kernel density estimation derived from recent queries and the well-known *exponential moving average* (EMA) in order to predict the query distribution in a multi-dimensional problem space that dynamically changes. Based on the estimated query distribution, the front-end scheduler assigns incoming queries so that query workloads are balanced and cached results are reused. Our experiments show that the proposed query scheduling policy outperforms existing policies in terms of both load balancing and cache hit ratio.

[1] This work was supported by the National Research Foundation of Korea(NRF) funded by the Ministry of Education, Science and Technology (2011-001475) and KEIT of Korea

1 Introduction

Load-balancing has been extensively investigated in various fields in the past including multiprocessor systems, computer networks, and distributed systems. For load balance, various scheduling algorithms have been introduced; one of the simplest algorithms is the round robin scheduling, and more intelligent load balancing algorithms were proposed, taking additional performance factors into account, such as the current system load, heterogeneous computational power, and network connection of the servers.

In many computational domains, including scientific and business applications, the application dataset has been growing in its size. Moreover, recent computing trend is to analyze massive volumes of data and identify certain patterns. Hence many modern applications spend a large amount of execution time on I/O and manipulation of the data. The fundamental challenge for improving performance of such data-intensive applications is managing massive amounts of data, and reducing data movement and I/O.

In order to reduce the I/O on the large datasets, distributed data analysis frameworks place huge demands on cluster-wide memory capabilities, but the size of the memory in a cluster is not often big enough to hold all the datasets, making *in-memory* computing impossible. However, the caching facilities scale with the number of distributed servers, and leveraging the large distributed caches plays an important role in improving the overall system throughput as many large scale systems are being built by connecting small machines.

In distributed environments, orchestrating a large number of distributed caches for high cache-hit ratio is difficult. The traditional query scheduling policies such as load-monitoring [1] are not sophisticated enough to consider cached results in distributed servers; they only consider load balance. Without high cache-hit ratio, the distributed caches will be underutilized, leading to slow query responses. On the contrary, scheduling policies that are solely based on data reuse will fail to balance the system load. If a certain server has very hot cached items, that single server will be flooded with a majority of queries while the other servers are all idle. In order to maximize the system throughput by achieving load balancing as well as exploiting cached query results, it is required to employ query scheduling policies that are more intelligent than the traditional round-robin and load-monitoring scheduling policies.

In this paper, we propose novel distributed query scheduling policies for multi-dimensional scientific data-analysis applications. The proposed distributed query scheduling policies make query scheduling decisions by interpreting the queries as multi-dimensional points, and cluster them so that similar queries cluster to-

funded by the IT R&D program MKE/KEIT (No. 10041608).

gether for high cache-hit ratio. The proposed scheduling policies also balance the load among the servers by leveling the cluster sizes in the caches. Our clustering algorithms differ from the well-known clustering algorithms such as k-means or BIRCH [30] in that the complexity of the proposed query scheduling algorithms is very light weight and independent of the number of cached data items since the query scheduling decisions should be made dynamically at run time for the incoming stream queries. Moreover, the goal of the well known clustering methods is to minimize the distance of each object to the belonged cluster, while the distributed query scheduling policies try to balance the number of assigned objects in each cluster while increasing the data locality.

To evaluate the performance of the proposed scheduling policies, we implemented the scheduling policies on top of a component-based distributed query processing framework. Scientific data analysis application developers can implement the interface of user-defined operators to process scientific queries on top of the framework. We conducted extensive experimental studies using the framework and show the proposed query scheduling policies significantly outperform the conventional scheduling policies in terms of both load balancing and cache hit ratio.

The rest of the paper is organized as follows: In section 2, we discuss other research efforts related to cache-aware query scheduling and query optimization. In section 3, we describe the architecture of our distributed query processing framework. In section 4, we discuss BEMA (Balanced Exponential Moving Average) scheduling policy [15] and analyze its load balancing behavior. In section 5, we propose a novel scheduling policy - EM-KDE (Exponential Moving - Kernel Density Estimation) that improves load balancing. In Section 6 we present an extensive experimental evaluation, where we examine the performance impact of different scheduling policies, measuring both query execution and waiting time, as well as load balancing. Finally we conclude in section 7.

2 Related Work

The scheduling problem that minimizes the makespan of multiple jobs in parallel systems is a well known NP-hard optimization problem. This led to a very large number of heuristic scheduling algorithms that range from low level process scheduling algorithms on multiprocessor systems to high level job scheduling algorithms in cluster, cloud, and grid environment [4, 5, 6, 8, 21, 26, 27].

Catalyurek et al. [6] investigated how to dynamically restore the balance in parallel scientific computing applications where the computational structure of the applications change over time. Vydyanathan et al. [24] proposed a scheduling algorithm that determines which tasks should be run concurrently and how many processors should be allocated to each task. Their scheduling algorithm tries to minimize the makespan of mixed parallel applications by improving the data locality of user tasks.

Zhang et al. [29] and Wolf et al. [25] proposed scheduling policies that dynamically distribute incoming requests for clustered web servers. WRR (Weighted Round Robin) proposed by Katevenis et al. [9] is a commonly used, simple but enhanced load balancing scheduling policy, which assigns a weight to each queue (server) according to the current load, and serves each queue in proportion to the weight. 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) [2, 18] 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 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, but LARD transfers workload only when a specific server is heavily loaded while our scheduling policies actively predict future workload balance and take actions beforehand to achieve better load balancing.

DEMA (Distributed Exponential Moving Average) [16] is a locality-aware scheduling policy for multidimensional scientific data analysis applications. DEMA scheduling policy partitions the problem space into as many Voronoi cells as the back-end servers, and all the queries that belong to the same cell are assigned to the associated back-end server. The DEMA scheduling policy adjusts the Voronoi cell boundaries so that the similar number of queries are likely to fall in each cell, and the queries that belong to a cell maintains locality to each other. Similar to our work, DEMA achieves load balancing as well as exploits cached results in a distributed caching infrastructure. BEMA(Balanced Exponential Moving Average) [15] is an improved version of DEMA that is also a locality-aware scheduling policy. We compare the performance of BEMA against our proposed scheduling policy in this work.

In relational database systems and high performance scientific data processing middleware systems, exploiting similarity of concurrent queries has been studied extensively. Prior 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 and query response time [10, 16]. Zhang et al. [28] evaluated the benefits of reusing cached results in a distributed cache framework, and they showed that high cache hit rates do not always yield high system throughput without load balancing. We aim to design scheduling policies that achieve both high cache-hit ratio and load balancing at the same time.

In order to support data-intensive scientific applications, a large number of distributed query processing middleware systems have been developed including MOCHA [20], DataCutter [10], Polar* [22], ADR [10], and Active Proxy-G [10]. 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 is similar to our distributed query processing framework, but it employs meta-data directory services that monitor performance of back-end application servers. Using the collected performance metrics, the front-end scheduler determines where to assign incoming queries considering load balancing. Our framework is different from Active Proxy-G in that we do not monitor the status of the back-end servers, but statistically estimate the best query assignment based on the observed query distribution to maximize the reuse of cached objects while balancing the load as well.

MapReduce is a distributed data processing framework developed by Google, which schedules user jobs to increase data locality [7]. The locality-aware scheduling algorithms of MapReduce have been studied in recent literature [21, 26, 27]. Zaharia et al. [26] proposed a scheduling policy that delays a job if it can not be scheduled on a server that has needed data. ILA (Interference and Locality-Aware) scheduling algorithm proposed by Bu et al. [5] improved Zaharia's delay scheduling so that it adaptively adjusts the delay time proportional to the input file size, and considers the interference between virtual machines. Quincy is another scheduling algorithm that considers both fairness and data locality [8]. The data locality, fairness, and delay penalty are modeled in a min-cost network flow model. Their works are different from ours in that we consider the locality of volatile cached data items rather than data location in distributed file systems.

3 Distributed and Parallel Query Processing Framework and Distributed Semantic Caching

Many scientific data analysis applications have common features in the overall query processing workflow although they process different types of raw data sets. Since scientific datasets are commonly represented in a multi-dimensional space, the scientific datasets are often accessed via multi-dimensional range queries.

Figure 1 shows the architecture of our distributed and parallel query processing middleware for scientific data analysis applications. This architecture aims to build an efficient and scalable query processing system by achieving load balancing, parallel sub-query processing, and high cache hit ratio in a distributed semantic caching infrastructure [1]. The front-end server interacts with clients for receiving queries. When a query is submitted, the front-end server forwards the query to one

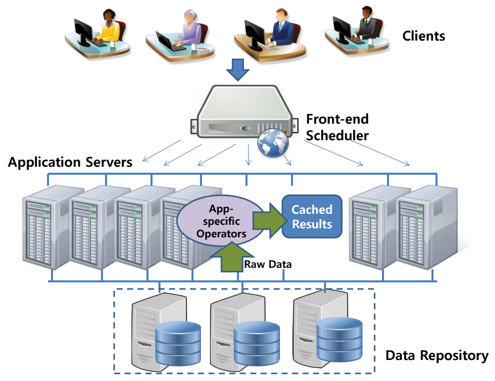


Fig. 1. *Architecture of Distributed Query Processing Framework*

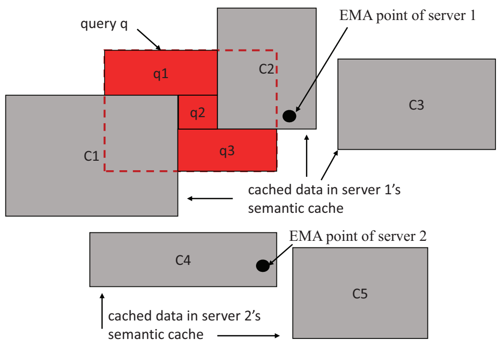

query q
EMA point of server 1
q1
C2
q2
C3
C1
q3
cached data in server 1's semantic cache
EMA point of server 2
C4
C5
cached data in server 2's semantic cache

Fig. 2. *Semantic Caching for Two-Dimensional Datasets and Query*

of the back-end application servers using its scheduling policy and the back-end server becomes responsible for processing the query and returning the result to the client.

When a back-end application server receives a query, the server spawns a query thread to execute the query. The query thread searches for cached objects in its own semantic cache that can be reused to either completely or partially answer a query. Since cached objects are generated by multi-dimensional range queries, cached objects are tagged by the queries' metadata - multi-dimensional coordinates. As shown in Figure 2, query thread searches for complete or partial objects for a given query using the tagged multi-dimensional coordinates. A query thread tries to find a cached object that has the largest overlap with the query. If the reusable cached object doesn't cover the whole range of the query, then the query thread generates sub-queries for the partial regions that are not covered by the cached data objects and repeats the same process for the sub-queries ($q1, q2, q3$ in the example). If a sub-query does not find any reusable cached object in the cache, it reads raw data from data repository and generates the query result from scratch by executing the user-defined operators. Our semantic caching algorithm is detailed in Algorithm 1.

Using this framework, application developers can customize the framework for their own applications, specifying how to retrieve raw datasets from storage sys-

Algorithm 1.
Query Processing with Semantic Cache

```
1: procedure PROCESSQUERY($Q$)                    ▷ $Q$ is a multi-dimensional query
2:     result ← null;
3:     $reusableData \leftarrow cacheHashMap.findLargestOverlap(Q.coord)$
4:     if $reusableData$ is not $null$ then
5:         $subQueries \leftarrow generateSubQueries(Q.coord, reusableData.coord)$
6:         for $s = 1$ to $subQueries.size$ do
7:             $output \leftarrow processQuery(subQueries[i])$;
8:             $result \leftarrow combine(result, output)$;
9:         end for
10:    else
11:        $result \leftarrow executeUserDefinedOperator(Q)$;
12:    end if
13:    $cacheHashMap.addAndReplace(Q.coord, result)$;   ▷ LRU replacement
14:    return $result$;
15: end procedure
```

tems, how to process incoming queries, and how to search, tag, and find query range overlapping amount using the tagged semantic metadata of cached objects.

4 Multiple Scientific Query Scheduling Policy from Geometric Perspective

In this section, as a background, we discuss an existing query scheduling policy called BEMA (Balanced Exponential Moving Average) [15] that is shown to outperform DEMA (Distributed Exponential Moving Average) [16] and identify its limitations.

4.1 Exponential Moving Average (EMA)

Exponential moving average (EMA) is a well-known statistical method to predict long-term trends and smooth out short-term fluctuations; applications are found in finance such as predicting stock prices and trading volumes [12]. Given a series of values, EMA computes the weighted average of all the past values with exponentially more weights on recent values. The EMA method can be employed to compute the trend of cached data in a distributed semantic cache infrastructure.

Let p_t be the multi-dimensional coordinate of a cached object at time $t > 0$ and EMA_t be the computed average at time t after adding p_t into the cache. Given the *smoothing factor* $\alpha \in (0, 1)$ and the previous average EMA_{t-1}, the next average

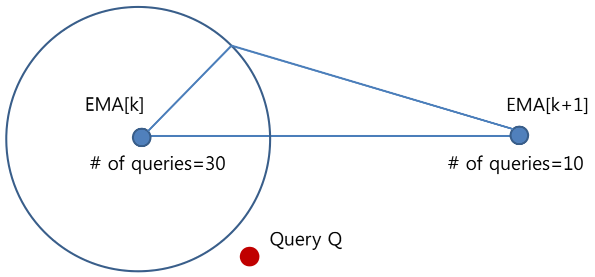


Fig. 3. *BEMA scheduler assigns the query to the server whose Apollonius circle encloses the query.*

EMA_t can be defined by

$$EMA_t = \alpha \cdot p_t + (1 - \alpha) \cdot EMA_{t-1}, \tag{1}$$

and it can be expanded as

$$EMA_t = \alpha p_t + \alpha(1 - \alpha)p_{t-1} + \alpha(1 - \alpha)^2 p_{t-2} + \cdots \tag{2}$$

The smoothing factor α determines the degree of weighing decay towards the past. For example, α close to 1 drastically decreases the weight on the past data (short-tailed) and α close to 0 gradually decreases (long-tailed). Note that EMA value can be a multi-dimensional element depending on the data domain of the application.

4.2 Balanced Exponential Moving Average (BEMA)

In distributed query processing systems, the front-end server often keeps track of the number of recently assigned queries to each back-end server for load balancing purpose. In order to promote the cache-hit ratio, the front-end server even needs to know the cache content of each back-end server. Such information can be obtained from the back-end servers by direct communication. This communication, however, can pose serious overhead on the front-end server as well as the back-end servers when the number of back-end servers is large and the number of cached data is enormous.

To achieve both the load balancing and cache-hit without communication overhead, the *Balanced Exponential Moving Average (BEMA)* scheduling policy employs the EMA method to approximate the trend of cached data in the back-end servers. Specifically, the front-end server maintains an exponential moving average (EMA) of query center points assigned to each back-end server. For example, in Figure 2, the front-end server stores two EMA points instead of five cached data objects in server 1 and 2.

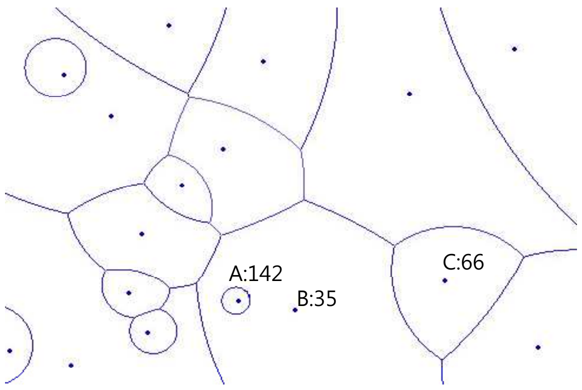


Fig. 4. *BEMA scheduler calculates the weighted Euclidean distance between EMA points and an incoming query, and assigns the query to the server B whose EMA point is closest.*

In BEMA scheduling policy, a query is assigned to the server whose EMA is the closest to the center of the query in terms of *weighted normalized Euclidean distance*. In the example shown in Figure 2, a query q is assigned to server 1 since the query's center point is closer to EMA of server 1 than EMA of server 2. In order to enforce each dimension has the same importance, we normalize the multi-dimensional problem space before we compute the Euclidean distance so that each normalized coordinate is within 0 and 1.

The weighted Euclidean distance from a server is the distance weighted proportional to the current load of the server. The rationale is that we want to assign less queries to the sever with more current load. Figure 3 illustrates an example of a 2-dimensional query assignment when server k's load has three times the server $(k + 1)$'s. The *Apollonius circle* in the figure represents points that have equal weighted-Euclidean-distances to both the servers. In this example, the query Q is assigned to server $(k+1)$ as it is closer to that server. With the Apollonius circle assignment model, BEMA scheduling policy makes scheduling decisions based on a weighted Voronoi diagram illustrated in Figure 4, which divides the plane into cells with respect to weighted Euclidean distance. Each cell is associated to a back-end server, i.e., all the queries falling in a cell are assigned to the corresponding server.

The front-end server can obtain the current load of the back-end server either by monitoring the number of recently assigned queries to the server, or by periodically collecting performance metrics from the back-end server.

The BEMA scheduling policy assigns geometrically proximate queries to the same back-end server in order to increase the cache-hit ratio, and adjusts the geometric boundaries between servers so that server loads are balanced in the long run. Note that the BEMA scheduling policy does not need to compute the boundaries of each cell, but it simply calculates the weighted Euclidean distance from a query to the EMA points of the servers and picks the closest server. The boundary of a server changes when the EMA point of the server or a neighboring server changes, or

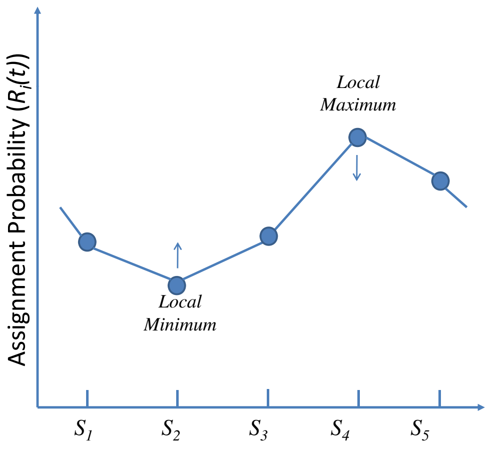


Fig. 5. *Load Balancing by Convex Optimization*

the load of the server or a neighboring server changes. Therefore the complexity of the BEMA scheduling policy is just $O(N)$ where N is the number of back-end servers. Figure 5 illustrates the balancing effect of the BEMA scheduling policy as a convex optimization. The figure shows the query assignment probabilities of server i's at time t (denoted by $R_i(t)$, respectively). The BEMA levels the assignment probabilities by decreasing the local maxima (S_4 in the figure) and increasing the local minima (S_2 in the figure). It does so by making the Voronoi cell of a local maximum more likely to shrink than to expand, and making the Voronoi cell of a local minimum more likely to expand than to shrink. Formally, BEMA maintains that

$$E[R_2(t+1)] \ > \ R_2(t) \tag{3}$$

and

$$E[R_4(t+1)] \ < \ R_4(t) \tag{4}$$

where $E[\cdot]$ is the expectation function. With this property, the system is moving toward a better balanced state.

4.3 Limitations of BEMA

Although the BEMA scheduling policy will balance the server loads in the long run, it fails to quickly adapt to sudden changes in the query distribution. This is mainly because a single query can change boundaries of at most a single server. For example, when a popular query region (called *hot-spot*) suddenly moves to a distant location that is currently covered by only a few servers, BEMA may need a substantial amount of time to move more EMA points toward the hot-spot, and this delay can cause heavy query-processing loads to the hot-spot servers until more servers gather to that area.

In [15], the BEMA scheduling policy is shown to outperform other traditional scheduling policies such as round-robin or load-based scheduling policies and DEMA - a locality aware scheduling policy even for rapidly changing query distribution. However, in this work, we present a scheduling policy that achieves even better load balancing while yielding comparable cache-hit ratio.

5 Multiple Query Scheduling with Exponential Moving Kernel Density Estimation (EM-KDE)

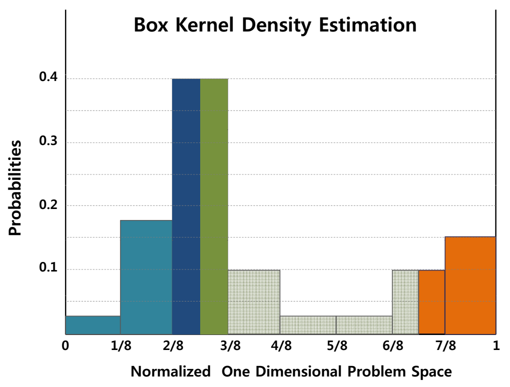
In this section, we propose a novel scheduling policy EM-KDE (Exponential Moving with Kernel Density Estimation). Unlike the BEMA, the EM-KDE quickly adapts to a sudden change of query distribution, while achieving both load balancing and high cache-hit ratio.

We first introduce our KDE-based scheduling method, then address the dimensionality of scientific data analysis queries. Next, we describe the exponential moving method that reflects the recent query trend, and then provide the EM-KDE scheduling algorithm in detail.

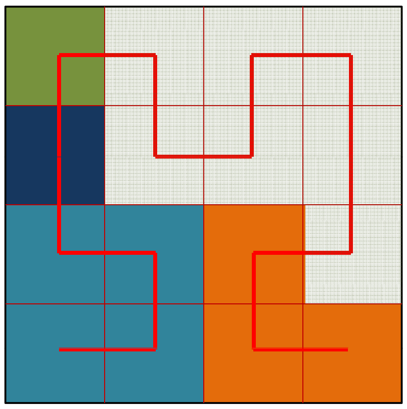
5.1 KDE-based scheduling

In brief, the KDE-based scheduling policy first estimates the probability density function (PDF) of incoming queries, then partitions the query space into equally-probable sub-spaces and associates each sub-space to a back-end server. As a consequence, assigning all the queries falling in a sub-space to the associated back-end server can evenly distribute incoming queries among the servers while keeping the queries to the same server spatially clustered.

To achieve both load balancing and high cache-hit ratio, the front-end scheduler must estimate the current query distribution accurately. In statistics, Kernel Density Estimation (KDE) is commonly used to estimate the PDF of a random variable based on the finite set of samples. When a sample arrives, we accumulate some value on the sample's value and around it in a predefined pattern, called *kernel function*, expecting the final accumulated values at each point to estimate the PDF. Various kernel functions have been proposed: from uniform to Gaussian. Depending on the kernel function, the resulting PDF estimation can look uneven or smooth. For example, uniform kernel function can generate histogram-like PDF with lots of



(a) Simple box kernel density estimation represents the recent query distribution



(b) Two dimensional problem space is partitioned into five sub-spaces along with second order Hilbert curve

Fig. 6. *The one dimensional boundaries on the Hilbert curve are determined by the cumulative probability distribution so that equal number of queries are assigned to each back-end server. The Hilbert curve that preserves spatial locality clusters multi-dimensional queries and increases the probability of cache hit ratio.*

sudden changes. To smooth out the PDF, one can use a smoothing parameter called *bandwidth*, which stretches the kernel function centered at each sample to spread the effect of accumulation to neighboring values. Choosing the optimal bandwidth of the kernel has been studied in many literature [19], and the most commonly used criterion is the mean integrated squared error.

To make the scheduling algorithm as light as possible, we use *box kernel density estimator*, which uses the uniform kernel function. We first partition the query space into a large number of fine-grained histogram bins. Then, for each query, we increase the counter of multiple adjacent k bins at the center of query range by 1/k for a single query, where k is a bandwidth parameter.

After constructing the smoothed probability density estimation, the EM-KDE scheduling policy chooses the boundaries of problem space for each back-end server so that each has equal cumulative distribution value. Unlike the BEMA scheduling policy, the EM-KDE scheduling policy adjusts the boundaries of all the servers together based on the query distribution so that it can provide good load balancing even for dynamic and rapidly changing query distribution.

5.2 Addressing Dimensionality

Scientific data analysis queries typically specify multi-dimensional geometric or geographic range of coordinates: for example, latitude/longitude and time ranges, or 3-D spatial minimum bounding rectangles. To schedule multi-dimensional queries, the EM-KDE scheduling policy enumerates the multi-dimensional queries on a Hilbert space filling curve [14]. A Hilbert curve is a continuous fractal space filling curve, which is known to preserve good spatial locality, i.e., it converts nearby multi-dimensional points into close one-dimensional values. Clustering nearby queries in one-dimensional Hilbert curve takes advantage of the clustering property of the Hilbert curve, and improves the probability of data reuse by assigning nearby queries to the same back-end servers.

Figure 6 illustrates how the EM-KDE scheduling policy partitions the Hilbert space filling curve based on the probability distribution function. The front-end scheduler converts multi-dimensional range queries into one-dimensional Hilbert value, and constructs box kernel density estimation. Using the box kernel density estimation, the scheduler partitions the Hilbert curve so that the cumulative probabilities of the partitioned intervals become equal. The partitioned one-dimensional Hilbert value intervals can be mapped back to the multi-dimensional space as shown in Figure 6(b).

5.3 Exponential Moving KDE

In order to reflect the recent trend in query distribution, we need a mechanism to gradually attenuate the effect of old queries on the PDF estimation. The BEMA scheduling policy employs the EMA to reflect the cache replacement effect in a single server. The EM-KDE scheduling policy, however, uses the EMA for adapting the box kernel density estimation to recent query distribution.

The attenuation parameter α affects the scheduling performance, and its proper value depends on the query dynamics. As α increases, the scheduler becomes more sensitive to the recent query distribution and adapts quickly to new query arrival patterns. On the other hand, as α decreases, the PDF estimation focuses on the long term trend, and becomes immune to temporary changes in query distribution. Generally, a smaller α is expected to have a higher cache-hit ratio since it works as if it does not consider load balancing but makes scheduling decisions with locality and data reuse. So smaller α value is considered to be good when the query arrival pattern is stable over time. But in practical applications the query arrival patterns change dynamically over time, and in such a case α value should be incremented in order to quickly adapt to the new query distribution.

5.4 EM-KDE algorithm

The detailed algorithm description of EM-KDE is shown in Algorithm 2. Initially, the frequencies of all Hilbert value intervals are set to the same value so that the values add to one. When a query arrives, the current density estimates are faded out

Algorithm 2.
EM-KDE Scheduling Algorithm

```
1: procedure SCHEDULE($Q$)                    ▷ $Q$ is a multi-dimensional query
2:     $hv \leftarrow HilbertValue(Q)$
3:     for $s \leftarrow 0, NumOfAppServers - 1$ do
4:         if $hv <= BOUNDARY[s]$ then
5:             $selectedServer \leftarrow s$
6:             forward query Q to selectedServer.
7:             $BoundaryUpdate(hv)$
8:         end if
9:     end for
10: end procedure
```

by a factor of $(1 - \alpha)$ and the incoming query adds a weight α on a target interval. As a result, the total sum of the counts is always one and it can be considered as a probability density.

The procedure $Schedule$ converts the center point of a multi-dimensional range query into one dimensional Hilbert value - hv, and chooses a back-end server whose Hilbert value sub-range includes the hv value. The scheduler forwards the client query to the back-end server, and calls $BoundaryUpdate$ function to update the frequencies of histogram and recalculate the Hilbert value sub-ranges of back-end servers.

The $BoundaryUpdate()$ function, described in Algorithm 2, interpolates the boundaries of histogram bins in order to further smooth out the blocky simple box kernel density estimation. When the problem space is large but the number of histogram bins is not large enough, the sub-range boundaries of back-end servers are likely to be in between of the histogram intervals. The while loop(line 16) of $BoundaryUpdate()$ function iterates the histogram bins and adds the frequency of each bin to $freqSum$ until it exceeds $freqPerServer$ which is the amount of frequency to be assigned for one back-end server. When the sum of current $freqSum$ and the frequency of next bin (f) becomes greater than $freqPerServer$, we use linear interpolation to calculate the boundary.

The complexity of the EM-KDE scheduling algorithm is $O(n)$ where the n is the number of histogram bins. As a new query arrives, the boundaries of sub-ranges need to be recalculated based on the updated histogram. Updating histogram can be done in constant time, but adjusting the boundaries need to rescan the histogram bins as shown in Algorithm 2. As we increase the number of histogram bins, the EM-KDE will determine the boundaries of each server in a more fine-grained way but it causes more overhead to the scheduler. But when the number of histogram bins is too small the scheduler can make decisions promptly but it may suffer from coarse-grained blocky probability density estimation. In our experimental study, we show that determining 312 boundaries (servers) using 2000 histogram bins takes no

Algorithm 3.
EM-KDE Boundary Update Algorithm

```
1: procedure BoundaryUpdate($hv$)        ▷ $hv$ is an integer Hilbert value of a multi-dimensional data object
2:     $selectedBin \leftarrow \lfloor hv/BinWidth \rfloor$
3:     $freqSum \leftarrow 0$
4:     for $b \leftarrow 0, NumOfBins - 1$ do
5:         if $b >= selectedBin - \lfloor bandwidth/2 \rfloor \&\& b <= selectedBin + \lfloor bandwidth/2 \rfloor$ then
6:             $frequency[b] \leftarrow frequency[b] * (1 - \alpha) + \alpha/(bandwidth + 1)$
7:         else
8:             $frequency[b] \leftarrow frequency[b] * (1 - \alpha)$
9:         end if
10:        $freqSum \leftarrow freqSum + frequency[b]$
11:    end for
12:    $freqPerServer \leftarrow freqSum/NumOfAppServers$
13:    $binWidth \leftarrow MaxHilbertValue/NumOfBins$
14:    $f \leftarrow frequency[0]$
15:    $s, i, boundary, interpolation, freqSum \leftarrow 0$
16:    while $i < NumOfBins$ do
17:        if $(freqSum + f) <= freqPerServer$ then
18:            $freqSum \leftarrow freqSum + f$
19:            $f \leftarrow frequency[++i]$
20:            $binWidth \leftarrow MaxHilbertValue/NumOfBins$
21:            $boundary \leftarrow binWidth * i$
22:        else
23:            $interpolation \leftarrow (freqPerServer - freqSum)/f$
24:            $boundary \leftarrow boundary + interpolation * binWidth$
25:            $BOUNDARY[s++] \leftarrow boundary$
26:            $f \leftarrow f - (freqPerServer - freqSum)$
27:            $binWidth \leftarrow binWidth - interpolation * binWidth$
28:            $freqSum \leftarrow 0$
29:        end if
30:    end while
31: end procedure
```

more than 50 usec. Considering disk I/O latency to read just a single 4 Kbytes page takes often longer than 50 usec, EM-KDE scheduling algorithm is not likely to become a performance bottleneck for data intensive applications. We will discuss the scalability issue of EM-KDE scheduling algorithm in more details in section 6. By any chance, if a scheduler is flooded with an unprecedentedly large number of queries, we can adjust the EM-KDE boundary update interval in order to reduce the scheduling overhead at the cost of sacrificing the probability of reusing cached data objects.

6 Experiments

In this section, we evaluate EM-KDE, BEMA, and round-robin scheduling policies in terms of query response time, cache hit ratio that shows how well the queries are clustered, and the standard deviation of the number of processed queries across the back-end servers to measure load balancing, lower standard deviation indicating better load balancing.

Satellite remote sensing data visualization tool is an example of scientific data analysis applications implemented on top of our distributed query processing framework that can be used for geographical, meteorological, and environmental studies. Systems processing remotely sensed data often provide on-demand access to raw data and user-specified data product generation. It targets datasets composed of remotely sensed AVHRR GAC level 1B (Advanced Very High Resolution Radiometer - Global Area Coverage) orbit data [17]. The raw data is continuously collected by multiple satellites and the volume of data for a single day is about 1 GB. Each sensor reading is associated with a position (longitude and latitude) and the time the reading was recorded. One of the most common access patterns into such satellite datasets is the multi-dimensional range query. Another target scientific data analysis application for our query processing framework is the multi-perspective vision system. The recent advances in computer vision and virtual reality systems develop higher interest in 3D tracking and 3D shape analysis. Multiple cameras shoot a sequence of frames from multiple perspectives and post-processing algorithms develop 3D volumetric representations from an arbitrary viewpoint. The queries of this application specify a 3D region, time, the cameras to use, and the resolution of the outputs.

The performance improvement from the proposed scheduling policies are dependent on the nature of applications. In order to show the magnitude of improvement that can be expected from employing more intelligent scheduling policies, we performed extensive experiments using various workloads, and observed the performance improvement from the scheduling policy mainly depends on the cache miss penalty. If the cache miss penalty is very low, then intelligent scheduling policy that achieves high cache hit ratio and good load balancing is no better than simple load-based or round-robin scheduling policy. As the cache miss penalty for such scientific data analysis applications depends on the range of multi-dimensional queries, we adjusted the query size to make the average query processing time to read raw datasets and process them about 200 ms.

To measure the performance of the scheduling policies, we synthetically generated two types of workloads - *realistic workloads* and *dynamic workloads*. The realistic workloads are generated by *Customer Behavior Model Graph (CBMG)*, which is one of the most commonly used methods that characterizes system workload for E-commerce systems [13]. CBMG models the user behaviors by a transition probability matrix. In order to generate multi-dimensional query workloads, we select a set of hot spots in the problem space. The CBMG workload generator selects one of the hot spots as its first query, and subsequent queries are modeled by the transition probabilities between spatial movements, zoom in or out, and jumps to new hot spots. For dynamic workloads, which exhibit sudden changes in query distribution, we use a *spatial data generator* developed by Yannis Theodoridis, which can generate multi-dimensional data in normal, uniform, or Zipf distributions [23]. We generate a set of different query distributions with various mean and standard deviation, and randomly combine them to make workloads that drastically move hot spots to different locations. Using the dynamic query workloads, we evaluate how well the EM-KDE, BEMA scheduling and Round Robin policies adapt to sudden changes of query arrival pattern. Each workload that we generate has 40,000 2-dimensional queries with various transition probabilities.

Note that the application servers use Least Recently Used (LRU) cache replacement policy and keep only certain number of recent query results in its cache. In our implementation, we used `java.util.LinkedHashMap` class to implement LRU semantic cache. LRU cache is known to be very expensive in real systems, but as our framework targets data intensive applications, the overhead of managing LRU cache is acceptable considering data processing time and the benefit of cache hits are significantly higher than the overhead of managing LRU cache.

Our testbed is 40 nodes Linux cluster machines that have dual Quad-Core Xeon E5506 2.13 GHz CPUs, 12 GB of memory per node, 7000 rpm 250 GB HDD, and they are connected by gigabit switched ethernet. If a target application does not run on multi-cores in parallel, our framework can deploy multiple back-end application servers in a single node. Dual quad core CPUs in 40 nodes allow us to run up to 320 back-end application servers concurrently.

6.1 Experimental Results of EM-KDE

6.1.1 Weight Factor

In the first set of experiments shown in Figure 7 and 8, we evaluate the performance of the EM-KDE scheduling policy using one front-end scheduler and 36 back-end servers with varying the weight factor α from 0.00001 to 0.32768. The weight factor α determines how fast the query scheduler loses information about past queries and thereby how quickly the probability density estimation adapts to the change of query distribution. As we increase α, the boundaries will move faster since it gives more weight to recent queries. With smaller α, the boundaries move more slowly. If α is 0, the boundaries will not move at all. The optimal α makes the boundaries move at the same speed as the query distribution change.

For both the dynamic query distribution and the CBMG distribution, load balancing

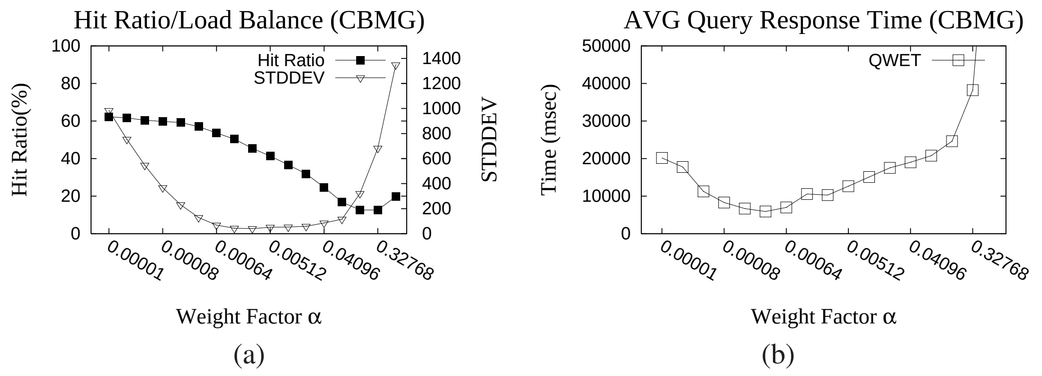


Fig. 7. *Performance of EM-KDE with Varying Weight Factor (Realistic Workload)*

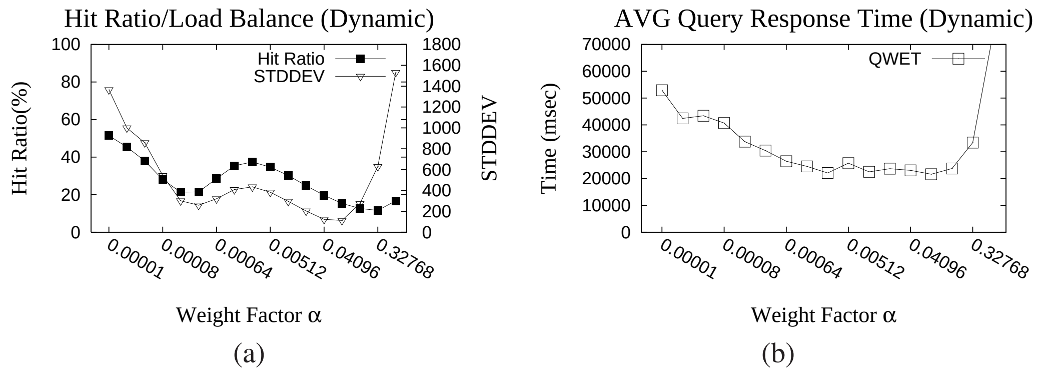


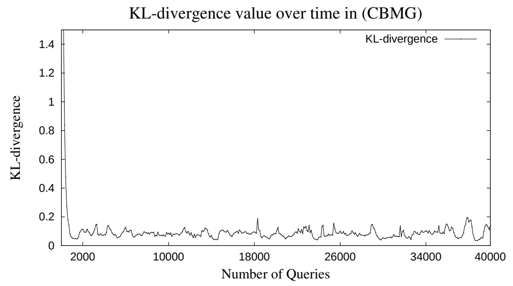
Fig. 8. *Performance of EM-KDE with Varying Weight Factor (Dynamic Workload)*

(STDDEV) becomes worse as α becomes too small because the boundaries of the back-end application servers fail to adjust to the changes in the query distribution. On the other hand, the load balancing is also poor when the α is higher than 0.04.

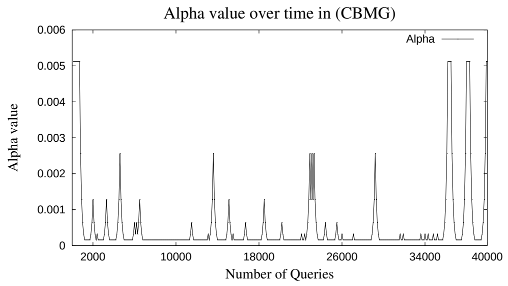
For the realistic workload (CBMG) shown in Figure 7, the cache hit ratio increases as α decreases. This is because the CBMG query pattern is rather stationary and a small α value, close to 0, makes the boundaries fixed, which increases the probability of reusing cached objects. However, when the α is too small, the boundaries move slow and they hurt the load balancing. On the contrary, when the α becomes large, the load balancing also suffers from frequent boundary fluctuation. The Figure 7(b) shows the fast query response time when the cache hit ratio is high and the load balancing is low.

For the dynamic workload shown in Figure 8, the results are similar except that the cache hit ratio and load balancing are good in two different α values. This happens because we synthetically merged four different query workloads that have different query arrival patterns. The best α values for the query workloads are all different and the cache hit ratio and the standard deviation graphs are convoluted. In Figure 8(b), the range of α values for good query response time is wider than the realistic workload, which means the dynamic query workload is less sensitive to α

than stationary query workloads. Also, note that with the dynamic query workload the query arrival pattern suddenly changes at certain intervals, hence the higher α value than the realistic workload shows fast query response time since it gives more weight to recent queries and adapts to the changed query distribution quickly.

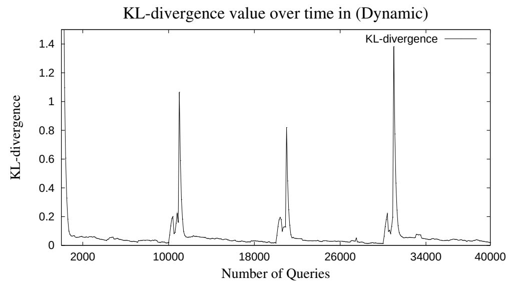


(a) KL divergence over time

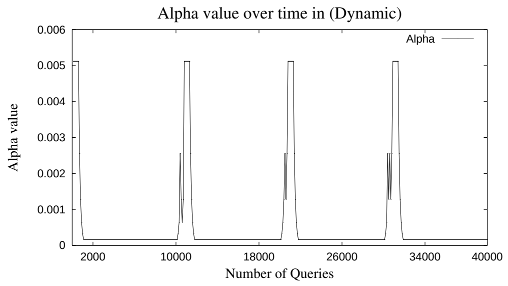


(b) Weight factor over time

Fig. 9. *Weight factor automation (Realistic Workload)*



(a) KL divergence over time



(b) Weight factor over time

Fig. 10. *Weight factor automation (Dynamic Workload)*

In order to automate selecting a good α, we increase or decrease α based on the change of the query distribution. If current query distribution histogram is significantly different from the past query distribution histogram, α value should be increased to move the boundaries faster. When the two query distribution histograms are similar, we decrease α to make the boundaries move slower. The difference between the two query distribution histograms can be measured by *Kullback-Leibler divergence* [11].

Figure 9 and 10 show how the Kullback-Leibler divergence and the selected weight factor value change for the two query workloads. In the synthetically generated dynamic query workloads, query distribution suddenly changes per every 10,000 queries, and the Figures show KL divergence value jumps up at those intervals. In our experiments, we doubled the weight factor when the KL divergence value becomes higher than 0.1, 0.2, 0.3, and so on respectively. Also we divided the weight factor by 2 when the KL divergence value becomes lower than 0.1, 0.2, 0.3, and so on. As shown in the figures, it does not take much time for the automated weight factor selection to adjust the query histogram to the changed query distribution and to make the KL divergence value small enough and stable.

With this automated weight factor adjustment, we obtained the cache hit ratio as high as 76 % for the dynamic query workload and 73 % for the realistic workload. Moreover, the standard deviation of processed queries per server was as low as 595 for dynamic workload and 106 for realistic workload. As a result, the query response time for dynamic query workload is 3092 msec, which is lower than any of the static weight factor alpha value shown in Figure 8(b), and 684 msec for the realistic query workload, which is also lower than any static weight factor value shown in Figure 7(b).

6.1.2 Number of Histogram Bins

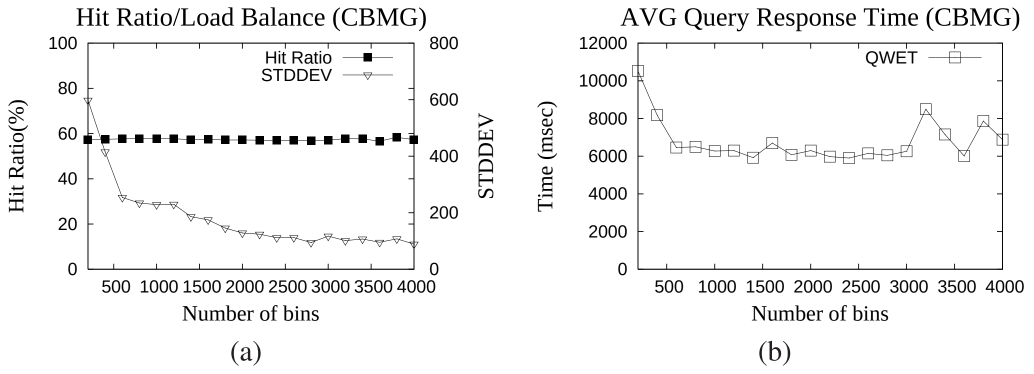


Fig. 11. *Realistic Workload: Number of Bins*

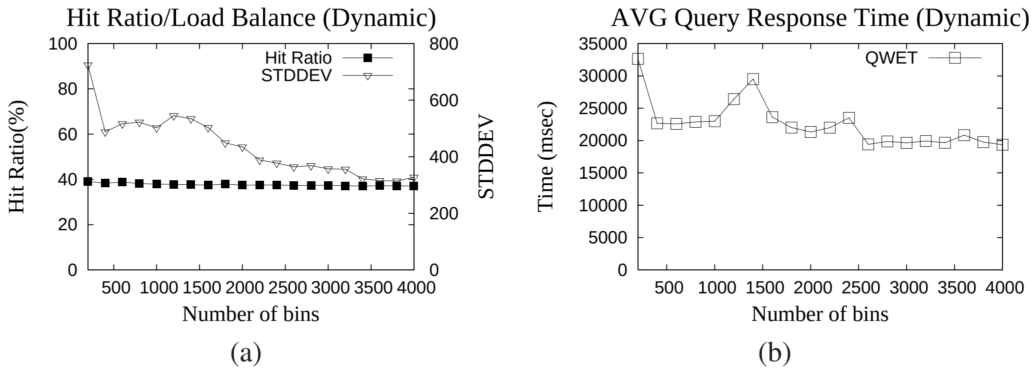
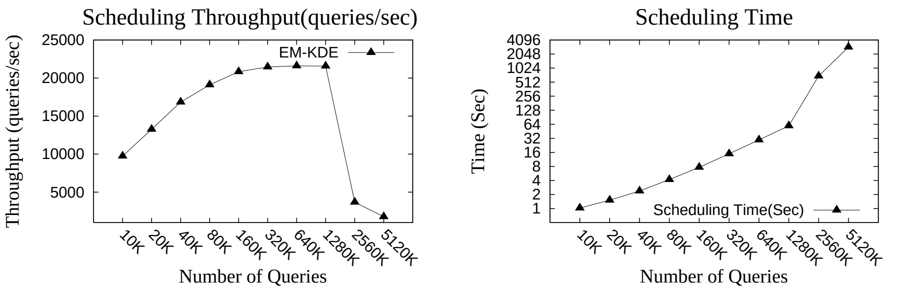


Fig. 12. *Dynamic Workload: Number of Bins*

Since the actual density of query distribution is not known priori, choosing the optimal number of histogram bins to minimize the errors in estimated probability distribution is not an easy problem. As we build the probability distribution with more histogram bins, the histogram will become smoother but the overhead in query scheduling will increase since the complexity of EM-KDE scheduling algorithm is $O(n)$ where n is the number of histogram bins.

Figure 11 and Figure 12 show the cache hit ratio, load balancing, and query response time with varying the number of histogram bins. For the experiments, the number of back-end servers is set to 36, and the average query inter-arrival time is 1 second. In Figure 11, when the number of histogram bins is 200, the cache hit ratio is 57 %, the standard deviation of processed queries is 598, and the average query response time is 10,527 msec. When we increase the number of histogram bins up to 40,000, the cache hit ratio is still 57 %, but the standard deviation decreased to 88 and the average query response time also dropped to 6877 msec. With the dynamic workload shown in Figure 12, the results are similar with the realistic workload. The number of histogram bins does not seem to significantly affect the cache hit ratio because not many hot cached objects are located across the boundary of each server, but load balancing improves slightly because the estimated probability distribution becomes smoother as the number of bins increases. However, in terms of the average query response time, the fine-grained histogram does not seem to help much unless the number of bins is too small. With the given query workloads, 1,000 histogram bins seem to be enough to obtain a smooth probability distribution, but 4,000 histogram bins do not place much scheduling overhead and as a result it does not hurt the average query response time in the experiments. For the rest of the experiments, we fixed the number of histogram bins to 2,000.

6.1.3 Scheduling Throughput with a Batch of Queries



(a) Scheduling Throughput (b) Scheduling Time

Fig. 13. *Scheduling Throughput of a Single Front-end Scheduler*

In order to evaluate the scheduling throughput of our presented distributed query processing framework, we deployed 312 back-end application servers with a single front-end scheduler in our testbed. In order to measure the scheduling throughput of EM-KDE scheduler, we submit a batch of very lightweight queries, each of which accesses a small 1 MBytes of data blocks from disks, with varying the total number of queries from 10,000 to 5 millions. As the queries are very lightweight, the bottleneck is more likely to be in the centralized scheduler rather than distributed back-end application servers.

As shown in Figure 13(a), the scheduling throughput increases up to 21,570 queries/sec with a larger batch of queries when the size of batch queries is less than 1.3 millions. However when more than 2 millions of queries are submitted in a batch, it exceeds the computation capability of the front-end server resulting in thrashing and performance degradation due to disk swapping. Figure 13(b) shows the elapsed job scheduling time per each query for the same experiments. As more queries are scheduled at the same time, job scheduling time increases linearly due to the waiting time in the front-end server's queue. When more than 2 millions of queries are submitted, thrashing occurs and the job execution time sharply increases.

We believe it is not very common to process more than 20,000 queries in a second in large scale scientific data-intensive applications. However, if a more scalable scheduler is needed for a certain type of applications, we may employ multiple front-end schedulers to distribute user queries. With multiple front-end scheduling servers, each EM-KDE scheduler will construct its own EM-KDE boundaries, but its drawback is that each scheduler does not know what queries have been processed by other schedulers. A possible solution to this problem is to let multiple schedulers periodically synchronize multiple EM-KDE boundaries so that they can correctly reflect the global query distribution.

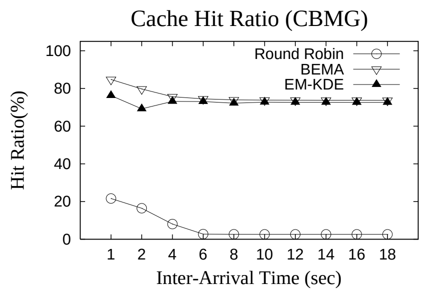
6.2 Performance Comparison

To evaluate the performance of the EM-KDE scheduling policy, we compare the cache-hit ratio and load balance of EM-KDE method with round-robin and BEMA scheduling policies for both realistic and dynamic workloads.

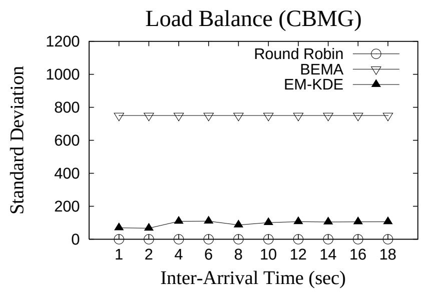
6.2.1 Inter-Arrival Time

Figure 14 and 15 show how the query scheduling policies perform for different concurrent loads. The query arrival was modeled by Poisson process, and the load was controlled by the mean query inter-arrival time. In the experiments the cache size of each back-end server is set to 1 GB and we run 36 back-end application servers. As the mean query inter-arrival time increases, the cache hit ratio of all the three scheduling policies decrease slightly. This is because a back-end server of our framework searches the waiting queries to see if the newly generated cached data item can be reused by any of the waiting queries. If such waiting queries are found, the back-end server fetches the queries immediately from the queue and process them. Because of this *queue jumping* policy, the cache hit ratio increases when the waiting queue has more pending queries. This can lead to unfairness for the rest of waiting queries, but since the queries that reuse cached data items are processed quickly, it will not significantly increase the waiting time of the other queries but help improve system throughput and overall query response time.

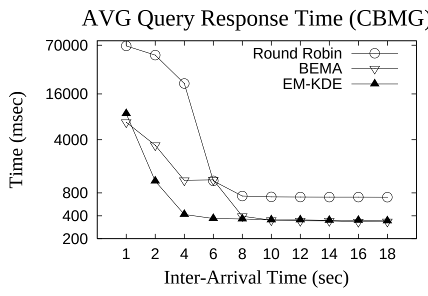
When the inter-arrival time is higher than 6 msec, the waiting queue of the incoming queries become short, and thus the query response time is determined by the query processing time rather than the waiting time. With the realistic workload, the EM-KDE scheduling policy showed the second highest cache hit ratio and good load



(a) Hit Ratio

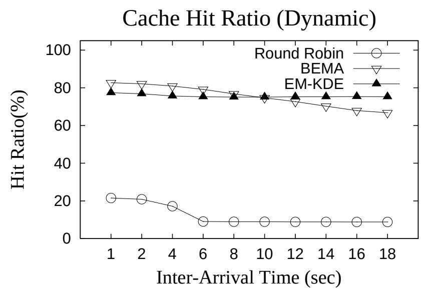


(b) Load Balance

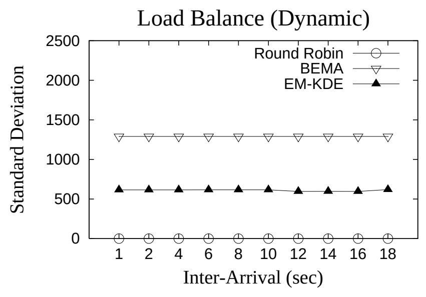


(c) Query Execution Time

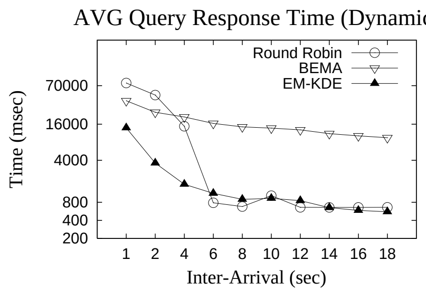
Fig. 14. *Performance Comparison with Varying Inter-arrival Time (Realistic Workload)*



(a) Hit Ratio



(b) Load Balance



(c) Query Execution Time

Fig. 15. *Performance Comparison with Varying Inter-arrival Time (Dynamic Workload)*

balancing, while the BEMA scheduling policy showed the highest cache hit ratio, but its load balancing is much worse than the EM-KDE scheduling policy. As a result, the EM-KDE scheduling policy shows the fastest query response time when the system is heavily loaded. However when the system is not heavily loaded, the waiting time for each query becomes almost ignorable unless load balancing is badly broken.

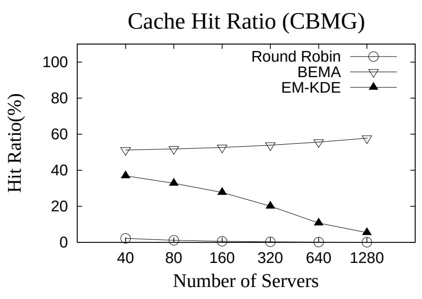
With the dynamic query distribution shown in Figure 15, the BEMA scheduling policy fails to balance the workload; only a few servers receive a large number of queries while the other servers remain idle. Thus, the query response time of the BEMA scheduling policy is much higher than even the round-robin scheduling policy due to its high waiting time. If the system is flooded with a large number of queries, the round-robin scheduling policy also suffers from high waiting time since most of queries result in cache miss. As the query inter-arrival time increases, however, the waiting time decreases and its query response time becomes faster than even the BEMA scheduling policy, which suffers from significant load imbalance problem. Note that the EM-KDE scheduling policy takes the advantages of both high cache hit ratio and good load balancing and outperforms the other two scheduling policies. When inter-arrival time is higher than 10 msec, the waiting queue becomes almost empty and the average query response time of the round-robin scheduling policy becomes close to the actual query execution time without queuing delay. Since the cache hit ratio of the EM-KDE is much higher than that of the round-robin, the performance gap between them will be enlarged when the cache miss penalty of applications is higher.

6.2.2 Scalability

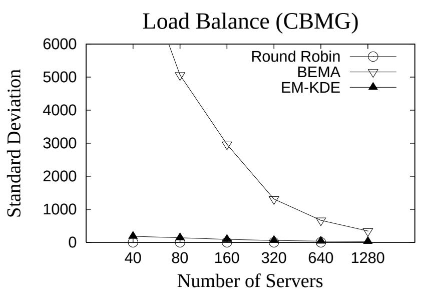
In the experiments shown in Figure 16, 17, and 18, we evaluate the scalability of our distributed query processing framework and scheduling policies.

As the number of back-end servers increases, the average query response time decreases sub-linearly since more back-end application servers can process more queries concurrently. In addition, more back-end servers have larger size of distributed caching infrastructure. In the experiments, the BEMA scheduling uses α value of 0.03 which shows its best performance while the EM-KDE scheduling automatically adjusts its weight factor α for realistic and dynamic workload.

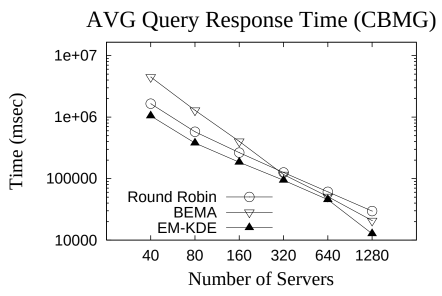
For the experiments shown in Figure 16, we vary the number of application servers ranging from 40 to 1280 and submitted 400,000 two dimensional queries. If more than 320 application servers are deployed in our testbed that has 320 cores in total, multiple back-end application server processes will share a single CPU core. Also, if multiple application servers share a single physical server, disk I/O can become a performance bottleneck that can affect the job execution time. Therefore, in order to evaluate the scalability of our framework, we conducted two types of experiments.



(a) Hit Ratio



(b) Load Balance



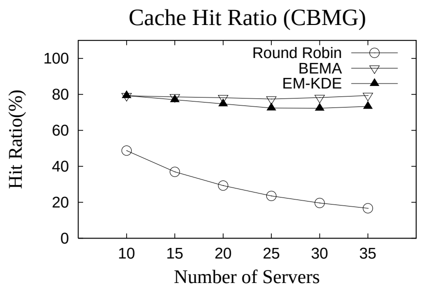
(c) Query Execution Time

Fig. 16. *Performance Comparison with a Large Number of Servers (Realistic Workload)*

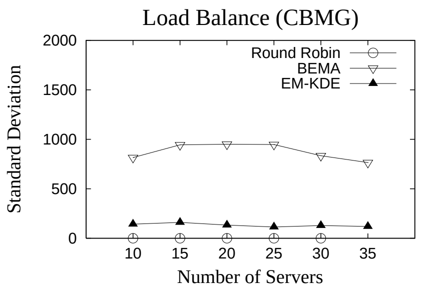
i) One is to submit real data-intensive queries to 40 application servers. ii) The other is to run a large number of application servers in 40 node testbed, but we eliminate disk I/O contention by submitting dummy queries that sleep instead of accessing datasets if cache misses occur. The sleep time was set to 200 msec. 200 msec was the average job execution time that accesses a 20 MB of raw data block if a cache miss occurs in our real experiments.

Throughout the experiments, BEMA scheduling policy shows the highest cache hit ratio, and the cache hit ratio even increases as the number of application servers increases. This is because with a larger number of application servers the total size of distributed semantic caches increases and BEMA scheduling takes the advantage of it since it makes its scheduling decisions solely based on the probability of data reuse. The cache hit ratio of EM-KDE scheduling policy is comparable to BEMA when the number of servers is small. However, as the number of servers increases, the load balancing flavor of EM-KDE scheduling policy hurts the cache hit ratio as EM-KDE scheduling policy assigns equal number of queries to a large number of application servers. As shown in Figure 16(b), EM-KDE outperforms BEMA in terms of load balancing by a large margin. Although BEMA scheduling policy achieves very high cache hit ratio, we observed more than 20,000 queries are waiting in a single application server's queue even after all the other application servers completed their assigned queries. As a result, EM-KDE consistently outperforms the other two scheduling policies in terms of average query execution time.

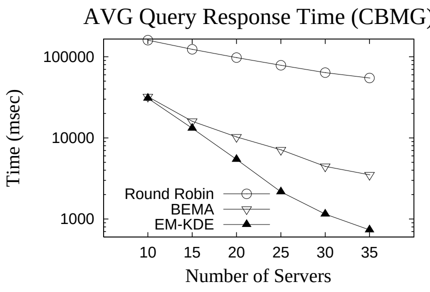
Instead of the large-scale experiments that simulate the cache miss penalty, we also measure the performance of scheduling policies with real queries and real cache miss penalty that accesses raw datasets in disk storage. In order to avoid the resource contention, we run 40 application servers in our 40 node testbed and submit 40,000 queries using realistic and dynamic workloads.



(a) Hit Ratio



(b) Load Balance

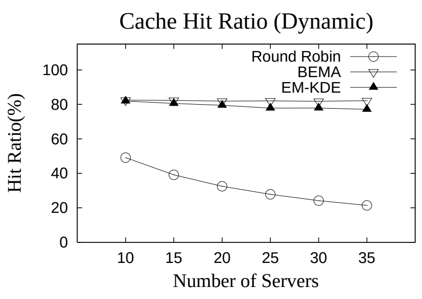


(c) Query Execution Time

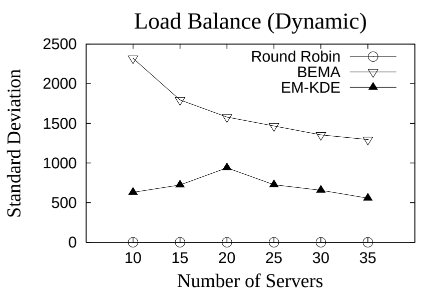
Fig. 17. *Performance Comparison with Varying Number of Servers (Realistic Workload)*

For realistic workloads, as shown in Figure 17, both the EM-KDE scheduling policy and the BEMA scheduling policy show around 80 % cache hit ratio when the number of servers is 10. As we employ more application servers, the cache hit ratio of the EM-KDE scheduling policy decreases slightly since popular data objects are distributed across more application servers. Note that the cache hit ratio of the BEMA scheduling policy does not decrease significantly but it suffers from high load imbalance and shows worse query response time than the EM-KDE scheduling policy. The load balancing performance of the EM-KDE scheduling is not as perfect as the round-robin scheduling policy but it shows quite small standard deviation especially when the query workload is stationary. Also note the BEMA scheduling policy shows faster query response time than the round-robin scheduling policy although its load balancing behavior is very poor.

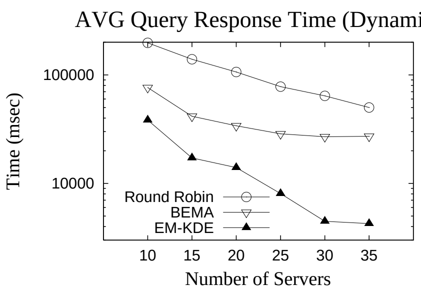
Figure 18 shows the performance with dynamic workloads. As query distribution is not stationary, both BEMA and EM-KDE scheduling policy show higher standard deviation of the number of processed queries per server compared to realistic workloads. Because of the load imbalance, the average query execution times for dynamic workloads are much higher than realistic workloads. But again, EM-KDE consistently outperforms BEMA and round-robin scheduling policy due to high cache hit ratio and good load balancing.



(a) Hit Ratio



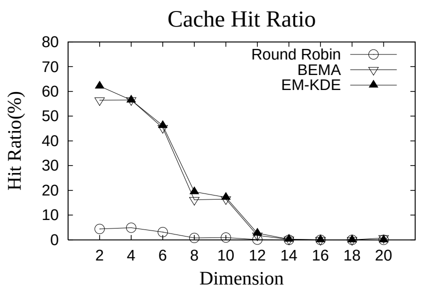
(b) Load Balance



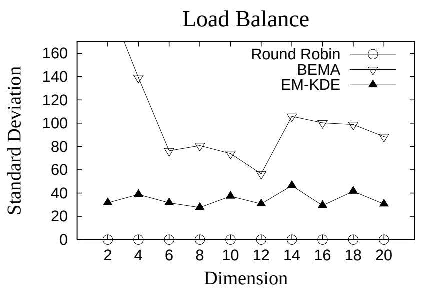
(c) Query Execution Time

Fig. 18. *Performance Comparison with Varying Number of Servers (Dynamic Workload)*

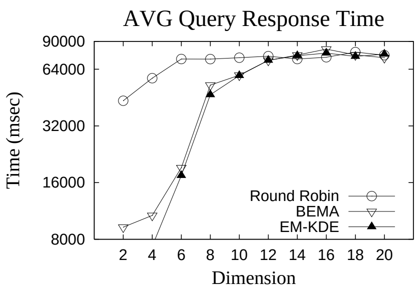
6.2.3 High Dimensional Dataset



(a) Hit Ratio



(b) Load Balance



(c) Query Execution Time

Fig. 19. *Performance Comparison with Varying Number of Dimensions (Uniform Workload)*

In the experiments shown in Figure 19, we present experimental results on high-dimensional synthetic datasets, looking at the effects of the dimensionality on performance. For the experiments, we used 40 back-end application servers, and generated synthetic 40,000 hypercube queries in uniform distribution, with the dimension ranging from 2 to 20. As the number of dimensions increases, the cache hit ratio decreases because the exponentially growing volume makes a query hard to overlap cached data objects. As the cache hit ratio decreases, the query execution time increases and EM-KDE shows similar performance to round-robin scheduling policy when the dimension is higher than 12. This is because of the well known *curse of dimensionality problem* [3].

EM-KDE scheduling policy shows similar cache hit ratio with the BEMA scheduling policy throughout the dimensions. However EM-KDE consistently outperforms BEMA in terms of load balancing. As a result, EM-KDE shows at least 10% faster job execution time against BEMA when the dimension is lower than 10, but the performance gap between scheduling policies drastically decreases as the dimension increases because of the low cache hit ratio, and all three scheduling policies show similar performance.

7 Conclusion and Future Work

In distributed query processing systems where the caching infrastructure is distributed and scales with the number of servers, both leveraging cached results and achieving load balance become equally important to improve the overall system throughput. Conventional scheduling policies that consider load balancing fail to take the advantage of data reuse, while scheduling policies that only consider data reuse may suffer from load imbalance.

In this paper we propose a novel intelligent distributed query scheduling policy that takes into consideration the dynamic contents of a distributed caching infrastructure and load balancing, and balances the two performance factors to achieve high system throughput. We compare the proposed scheduling policy *EM-KDE* against the conventional round-robin scheduling policy and another locality-aware scheduling policy - BEMA. Our experiments show the EM-KDE scheduling policy outperforms the other scheduling policies for realistic workloads and dynamically changing workloads by a large margin.

References

[1] Henrique Andrade, Tahsin Kurc, Alan Sussman, and Joel Saltz. Multiple query optimization for data analysis application clusters of SMPs. In *Proceed-*

ings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid). IEEE Computer Society Press, May 2002.

[2] Mohit Aron, Darren Sanders, Peter Druschel, and Willy Zwaenepoel. Scalable content-aware request distribution in cluster-basednetwork servers. In *Proceedings of Usenix Annual Technical Conference*, 2000.

[3] Richard E. Bellman. *Adaptive Control Processes: A GuidedTour*. Princeton University Press, NJ, 1961.

[4] Tekin Bicer, David Chiu, and Gagan Agrawal. Time and cost sensitive data-intensive computing on hybrid clouds. In *Proceedings of the 12th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*, 2012.

[5] Xiangping Bu, Jia Rao, and Cheng-Zhong Xu. Interference and locality-aware task scheduling for MapReduce applications in virtual clusters. In *Proceedings of the 22nd IEEE International Symposium on High Performance Distributed Computing (HPDC)*, 2013.

[6] Umit V. Catalyurek, Erik G. Boman, Karen D. Devine, Doruk Bozdag, Robert T. Heaphy, and Lee Ann Riesen. A repartitioning hypergraph model for dynamic load balancing. *Journal of Parallel and Distributed Computing*, 69(8):711–724, 2009.

[7] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 4th USENIX conference on Operating Systems Design and Implementation (OSDI)*, 2004.

[8] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, 2009.

[9] Manolis Katevenis, Stefanos Sidiropoulos, and Costas Courcoubetis. Weighted round-robin cell multiplexing in a general-purpose atm switch chip. *IEEE Journal on Selected Areas in Communications*, 9(8):1265–1279, 1991.

[10] Jik-Soo Kim, Henrique Andrade, and Alan Sussman. 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.

[11] Solomon Kullback and Richard A. Leibler. On information and sufficiency. *Annals of Mathematical Statistics*, 22(1):79–86, 1951.

[12] Ya lun Chou. *Statistical Analysis*. Holt International, 1975.

[13] Daniel A. Menasce and Virgilio A. F. Almeida. *Scaling for E-Business: Technologies, Models, Performance, and Capacity Planning*. Prentice Hall PTR, 2000.

[14] Bongki Moon, H. V. Jagadish, Christos Faloutsos, and Joel H. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *IEEE Transactions on Knowledge and Data Engineering*, 13(1):124–141, 2001.

[15] Beomseok Nam, Deukyeon Hwang, Jinwoong Kim, and Minho Shin. High-throughput query scheduling with spatial clustering based on distributed exponential moving average. *Special issue on data intensive eScience, Dis-*

tributed and Parallel Databases, 30(5–6):401–414, 2012.

[16] Beomseok Nam, Minho Shin, Henrique Andrade, and Alan Sussman. Multiple query scheduling for distributed semantic caches. *Journal of Parallel and Distributed Computing*, 70(5):598–611, 2010.

[17] NOAA, National Climatic Data Center. NOAA KLM user's guide section 3.1, September 2000. *http://www2.ncdc.noaa.gov/docs/klm/html/c3/sec3-1.htm*.

[18] Vivek Pai, Mohit Aron, Gaurav Banga, Michael Svendsen, Peter Druschel, Willy Zwaenepoel, and Erich Nahum. Locality-aware request distribution in cluster-based network servers. In *Proceedings of ACM ASPLOS*, 1998.

[19] Vikas C. Raykar and Ramani Duraiswami. Fast optimal bandwidth selection for kernel density estimation. In *SIAM Conference on Data Mining*, 2006.

[20] Manuel Rodríguez-Martínez and Nick Roussopoulos. MOCHA: A self-extensible database middleware system for distributed data sources. In *Proceedings of 2000 ACM SIGMOD*.

[21] Bikash Sharma, Timothy Wood, and Chita R. Das. HybridMR: A hierarchical MapReduce scheduler for hybrid data centers. In *Proceedings of the 33rd International Conference on Distributed Computing Systems (ICDCS)*, 2013.

[22] J. Smith, S. Sampaio, P. Watson, and N.W. Paton. The polar parallel object database server. *Distributed and Parallel Databases*, 16(3):275–319, 2004.

[23] Yannis Theodoridis. R-tree Portal. *http://www.rtreeportal.org*.

[24] N. Vydyanathan, S. Krishnamoorthy, G.M. Sabin, U.V. Catalyurek, T. Kurc, P. Sadayappan, and J.H. Saltz. 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.

[25] Joel L. Wolf and Philip S. Yu. Load balancing for clustered web farms. *ACM SIGMETRICS Performance Evaluation Review*, 28(4):11–13, 2001.

[26] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys)*, 2010.

[27] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving MapReduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX conference on Operating Systems Design and Implementation (OSDI)*, 2008.

[28] Kai Zhang, Henrique Andrade, Louiqa Raschid, and Alan Sussman. 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.

[29] Qi Zhang, Alma Riska, Wei Sun, Evgenia Smirni, and Gianfranco Ciardo. Workload-aware load balancing for clustered web servers. *IEEE Transactions on Parallel and Distributed Systems*, 16(3):219–233, 2005.

[30] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. BIRCH: an efficient data clustering method for very large databases. In *Proceedings of 1996 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1996.