Cluster Computing manuscript No.
(will be inserted by the editor)

Multi-dimensional Multiple Query Scheduling with Distributed Semantic Caching Framework

Youngmoon Eom · Jinwoong Kim · Beomseok Nam

Received: date / Accepted: date

Abstract It is becoming more important to leverage a large number of distributed cache memory seamlessly in modern large scale systems. Several previous studies showed that traditional scheduling policies often fail to exhibit high cache hit ratio and to achieve good system load balance with large scale distributed caching facilities. To maximize the system throughput, distributed caching facilities should balance the workloads and leverage cached data at the same time. In this work, we present a distributed job processing framework that yields high cache hit ratio while achieving balanced system load. Our framework employs a scheduling policy - *DEMA* that considers both cache hit ratio and system load and it supports geographically distributed multiple job schedulers. We show collaborative task scheduling and the data migration can even further improve the performance by increasing the cache hit ratio while achieving good load balance. Our experiments show that the proposed job scheduling policies outperform legacy load-based job scheduling policy in terms of job response time, load balancing, and cache hit ratio.

Keywords Semantic Caching · Multi-dimensional Query · Multiple Query Optimization · Data Migration · Distributed Query Scheduling

This research was supported by NRF (National Research Foundation of Korea) grant NRF-2014R1A1A2058843.

Y. Eom, J. Kim, B. Nam
School of Electrical and Computer Engineering, Ulsan National Institute of Science and Technology, Ulsan, South Korea
E-mail: youngmoon01@unist.ac.kr

J. Kim
E-mail: jwkim@unist.ac.kr

B. Nam (corresponding author)
E-mail: bsnam@unist.ac.kr

1 Introduction

Multi-dimensional data processing has been the subject of extensive research in various fields including scientific applications, database systems, computer graphics, geographic information systems, etc. A large portion of real world datasets are multi-dimensional, and enormous amount of multi-dimensional datasets are generated and analyzed by modern computer systems.

Efficient processing of such large multi-dimensional dataset is a major challenge in many disciplines, and distributed and parallel query processing systems have been successfully used for scientific applications to solve such large scale complicated data analysis problems because substantial performance gains can be obtained by exploiting data and computation parallelism.

When dealing with computationally intensive scientific applications, load balancing plays an important role in reducing query response time and improving system throughput via task parallelism. In addition to the load balancing, data intensive applications are known to benefit from reusing intermediate cached results if multiple queries can exploit sub-expression commonality [11].

In distributed query processing systems, the size of distributed caches scales with the number of distributed servers. Leveraging the large distributed caches plays an important role in improving overall system throughput but it is not an easy problem to reuse cached objects while maintaining load balance. Load-based scheduling policies do not consider cached objects in distributed caching system and get little benefits from reusing them. To take the advantages of large scale distributed caching infrastructure, more intelligent query scheduling policies are required. However, a cache-aware query scheduling policy does not always imply high system throughput because it may hurt system load balance. Suppose a scheduler knows a remote server has some popular cached objects, so it forwards most subsequent queries to the remote server just to improve cache hit ratio, but it will flood the server with a large number of queries while all the other servers are idle. Although in general cache hit ratio helps reducing query response time, if a server is flooded with too many waiting queries, the waiting time can be much higher than the benefit of cache hit.

In this paper, we study a distributed query scheduling policy - *DEMA* that makes query scheduling decisions based on *spatial locality* of multi-dimensional queries so that similar queries get clustered together in order to improve cache hit ratio and the number of queries in each cluster becomes balanced. Our proposed spatial clustering algorithm is different from the well known spatial clustering algorithms used in machine learning in a sense that the query scheduling decisions must be made at run time, i.e., the overhead of runtime scheduling algorithms should be minimal. DEMA scheduling policies take into account the dynamic contents of distributed caching infrastructure in order to exploit sub-expression commonality of cached results and employ statistical prediction methods to balance load across distributed servers.

In order to manage the distributed caching infrastructure in a seamless fashion and to facilitate collaborative scheduling over geographically dispersed data repositories and application servers, our distributed job processing framework employs multiple front-end servers that periodically synchronize workload statistics to make better job scheduling decisions. A salient feature of the cloud computing is that it enables collaborative research and facilitates access to remote computing resources. In the context of distributed cloud environment, multiple front-end schedulers are likely to reduce job response time as they accelerate scheduling decisions.

Our framework also employs an autonomic data migration policy. Because DEMA scheduling policy makes scheduling decision based on spatial locality, if the distribution of clustered cached objects changes and some cached objects do not belong to its cluster any more, our framework migrates the cached object to a remote server where the cached object can be reused. Our extensive experimental studies show our proposed job scheduling policy with multiple job schedulers and the data migration policy significantly improves the query processing throughput and outperforms legacy load based scheduling policy by a large margin.

The rest of the paper is organized as follows: Section 2 describes other research works related to cache-aware query scheduling policies and query optimization in distributed systems. In section 3, we describe overall architecture of our distributed query processing framework. Section 4 introduces DEMA scheduling policy and how the scheduling policy achieves both load balance and high cache hit ratio. In section 5, we discuss collaborative scheduling with multiple front-end schedulers. And we propose the data migration policy for DEMA scheduling algorithem in section 6. In section 7 we evaluate our scheduling policy against other existing scheduling policies. And we conclude the paper in section 8.

2 Related Work

Load-balancing problems have been extensively investigated in many different fields. Godfrey et. al [6] proposed an algorithm for load balancing in heterogeneous and dynamic peer-to-peer systems. Catalyurek et. al [5] investigated how to dynamically restore balance in parallel scientific computing applications where the computational structure of the applications change over time. Vydyanathan et. al [16] proposed scheduling algorithms that determine what tasks should be run concurrently and how many processors should be allocated to each task. Zhang et al. [19] and Wolf et al. [17] 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) [2,12] 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 jobs 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 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 reuse previously computed results from cache and generate good scheduling plans, resulting in improved system throughput as well as reducing query response times [8,11]. Zhang et al. [18] 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 [4], Polar* [15], ADR [9], and Active Proxy-G [1]. 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 [11]. 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 and Parallel Query Processing Framework

Figure 1 shows how a scheduler in our distributed multiple query processing framework makes a scheduling decision for a given query. The goal of this framework is parallel processing of multiple queries, load balancing, and high ratio of cached data reuse via distributed semantic caching system. The front-end scheduler interacts with clients for receiving clients' queries and forwards the query to one of the back-end application servers. Each back-end application server independently runs on a cluster node and has an access to large-scale back-end storage devices or networked databases.

When a query is submitted to the front-end scheduler, it determines which back-end application server should process the query. Most schedulers in distributed

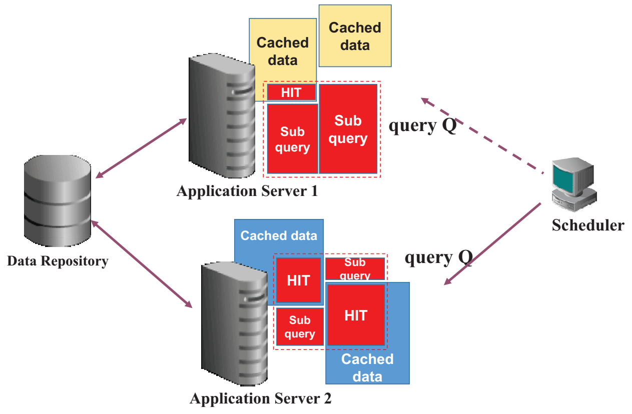


Fig. 1 *Architecture of Distributed Query Processing Framework*

query processing middlewares employ a monitoring service that periodically collects performance metrics from back-end application servers. Based on the performance metrics, the schedulers make scheduling decisions to balance the system load across back-end applications.

Our distributed query processing framework is a component-based distributed job processing system that is currently under development. Our framework provides a programming model that scientific data analysis application developers can use to implement user-defined functions to process multi-dimensional scientific datasets on top of our framework. The programming model in our framework includes APIs that search and read raw datasets from back-end storage devices using multi-dimensional metadata of the dataset, interfaces that application developers can implement for application specific user-defined operations, and APIs that store and search intermediate query results.

When a query is forwarded to an application server, the server executes an application-specific user-defined operator for the query. The user-defined operator searches for cached objects in its semantic cache that can be reused to either completely or partially answer the query. Our framework first finds a cached data object that has the largest overlap with the query. If the reusable cached object doesn't cover the whole range of the query, then it partitions the query into several sub-queries for the partial region that was not in the cache as in Figure 1, and repeats the same process for the sub-queries recursively. If no cached object is found in the cache, it reads raw dataset from storage systems and process the sub-query from scratch.

Since reading raw dataset and generating intermediate query output is a very expensive operation for large scale scientific data analysis application, if possible, the scheduler needs to make scheduling decisions to maximize cached data reuse in distributed caching facility. For example, in Figure 1, a given query Q better be forwarded to application server 2 since its cached multi-dimensional data objects overlap a larger portion of the query than application server 2. However, if a scheduler makes scheduling decisions only based on cache hit ratio, it may cause system load imbalance and hurt overall system throughput. For example, if a very few application servers have popular cached data objects, the application servers will be flooded with most of incoming queries, and it hurts system load balance and as a result system throughput will be poor.

4 Predicting Cached Data Objects in Distributed Semantic Caches

DEMA (Distributed Exponential Moving Average) that we propose is a multiple query scheduling policy that improves system throughput and reduces the query response time by taking into account of both load balance and cache hit ratio at the same time using a statistical prediction method. As distributed caching system dynamically changes its cached contents at very fast rate, DEMA scheduling policy makes real time scheduling decisions based on aggregated statistics of historical queries.

4.1 Background: Exponential Moving Average (EMA)

Our DEMA scheduling policy aggregates statistics of past queries using exponential moving average method. 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 in stock price and trading volume predictions. EMA formula computes a weighted average of all historical data by assigning exponentially more weight to recent data. As our target applications process multi-dimensional range query, we compute multi-dimensional exponential moving average point as follows.

Let p_t be the 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 EMA_t can be defined incrementally by

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

and it can be expanded as

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

Since our framework is supposed to run long enough to amortize the initial EMA error, we simply chose $k = 1$, i.e., $EMA_1 = p_1$ where p_1 is the first observed

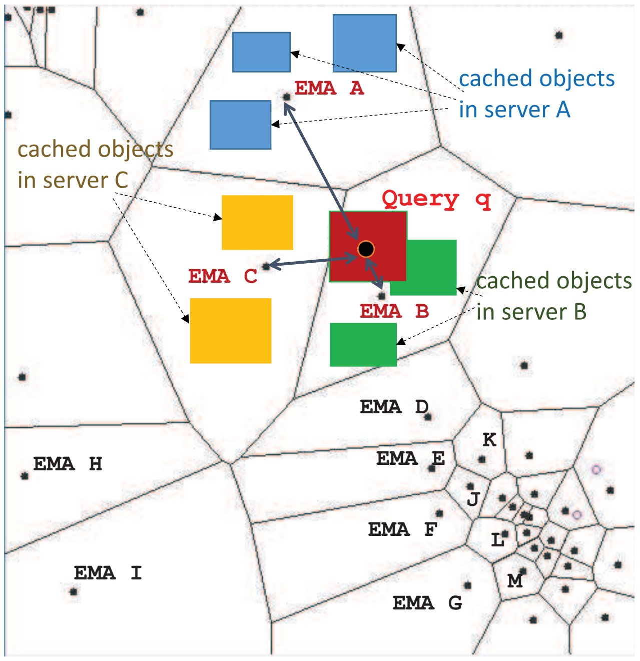


Fig. 2 *DEMA scheduler calculates the Euclidean distance between EMA points and an input data needed by an incoming job, and assigns the job to the server B whose EMA point is closest.*

data. Equation 1 is used for the following EMAs. Note that EMA value can be a multi-dimensional element depending on the data domain of the application.

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).

4.2 Distributed Exponential Moving Average (DEMA)

The *Distributed Exponential Moving Average (DEMA)* scheduling policy employs as many EMA values as the back-end application servers to determine which server is to be assigned for each query and how the assignment adapts to the dynamic change of the distribution of input datasets. As our target applications process multi-dimensional datasets, cached objects in each back-end application server can be represented by multi-dimensional coordinates. As shown in Figure 2, DEMA scheduling policy computes a representative multi-dimensional point (*EMA point*) for the list of cached objects in each application server's cache. When a new query is submitted to a scheduler, the scheduler computes the distance between the query and EMA points. In the example shown in the Figure 2 the query is closer to server 1's cached objects. Hence the query should be forwarded to server 1, then the application server can reuse some of the cached objects in order to process the new query.

Our DEMA scheduling algorithm works in the following three steps.

1. Initialization. For simplicity, let a unit circle represent the *query space* where the value starts from zero and increases in a clockwise direction up to one which meets at zero. We assume that each input data is mapped to a point in multi-dimensional problem space using its center. Suppose there are n back-end application servers that process user jobs and one scheduling server that assigns each incoming job to an application server, as shown in Figure 1. In the beginning, the scheduling server creates n EMA variables $EMA_1, EMA_2, \ldots,$ EMA_n, with EMA_i for the ith application server, and then assigns the first n input data to $EMA_1, EMA_2, \ldots,$ EMA_n, respectively.

2. Assignment. Let q denote the center point of an input data requested by an incoming job, ranging from 0 to 1, and let q also represent the job itself. Upon receiving a job q, the scheduling server finds the EMA point EMA_{i^*} that is closest to q (in terms of their Euclidean distance), and then assigns q to server-i^*. For example, in Figure 2 an input data q is closest to EMA point B, hence an incoming job that needs the input data q is assigned to server B.

3. Update. Once q is assigned to server-i^*, the scheduling server updates the EMA point for the server by

$$EMA_{i^*} = \alpha q + (1 - \alpha) EMA_{i^*}. \tag{3}$$

Complexity. In step 2, the scheduling server needs to find the closest EMA point to a given input data. This algorithmic problem, known as Nearest Neighbor Search, can be solved in $O(\log n)$ using space partitioning data structure such as *k-d tree* or *R-tree.* [1]

4.3 Improving Cache Hit

Note that an application server keeps only certain number of recent input data in its cache; for example, it may employ Least Recently Used (LRU) policy. Let L denote the current number of cached input data in the cache. Ideally, it is desired that the EMA point reflects only those in the cache. However, EMA reflects all the past data including those that were expunged from the cache, but with less weight for older ones. Therefore, we may want to choose the decay factor α such that the weight sum of the expunged input data is managed below a threshold. Formally, given a threshold $\epsilon \in (0, 1)$, we want

$$(1 - \alpha)^L < \epsilon. \tag{4}$$

Since L depends on the size of cached objects, we use the average number of input data in the cache instead, denoted by L^*. Therefore, we can choose the smallest α that meets

$$\alpha > 1 - \epsilon^{L^*}. \tag{5}$$

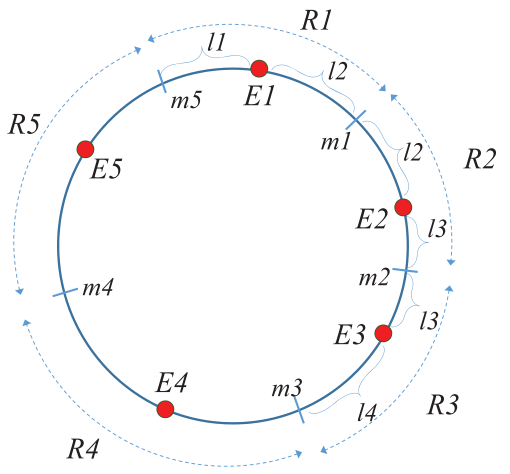
In this way, the EMA value of an application server reflects what are current in the cache, and assigning incoming jobs that are close to the EMA, only jobs that need similar input data (i.e., close to each other in problem space) are assigned to the application server. This clustering effect promotes the cache hit ratio in the application server.

4.4 Load Balancing of DEMA

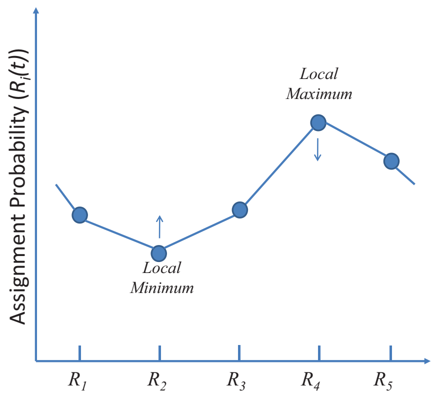
Another important feature that distributed multiple query scheduling policies guarantee is *load balancing*; each job can be assigned to one of the application servers with equal probability, leveling the workloads of the application servers. To that end, DEMA scheduling algorithm gradually moves the EMA points so that they follow the distribution of requested input data. If a certain region of problem space becomes hot, the EMA points move to the hot spot by EMA equation. If input data requests are uniformly distributed, then DEMA algorithm tries to keep the sizes of Voronoi regions as similar as possible. DEMA algorithm achieves this goal by evenly distributing EMA points throughout the problem space.

Let us assume that the problem space is a 2-dimensional plane and the input data requests follow a uniform distribution. Figure 2 is a snapshot of the Voronoi diagram where a dot represents an EMA value of an application server (e.g., EMA B is the EMA value of server B), and a line segment represents the set of Euclidean middle points between two EMA points. In DEMA scheduling, an incoming query that falls in a Voronoi cell of an EMA point makes the EMA point slide toward the input data point, where the amount of the movement is affected by the smoothing factor α (the more α, the more shift). In Figure 2, the job that needs the input data q will be forwarded to server B since the EMA point of server B is closer to the query than any other EMA points. It is called the *Voronoi assignment model* where we assign every multidimensional point to the nearest cell point.

[1] In our implementation, however, we performed a linear search that takes $O(n)$ assuming that n is small enough, for example, up to 40.



(a) Problem Space with five EMA values at time t



(b) Load Balancing by Convex Optimization

Fig. 3 *DEMA Load Balancing in One-dimensional Space*

The query assignment regions induced from the DEMA query assignment form a *Voronoi diagram* [3].

In this case, EMA B moves a little toward q, moving all the border lines of the cell a little toward the same direction. An important observation is that each EMA has tendency to move toward the center of its cell. For example, EMA E in the figure is located at the right corner of the cell. Thus more queries will arrive to the left of the EMA in the cell than to the right. Therefore, the EMA E is more likely to slide to the left rather than to the right. This movement will also move the border to the left, making the Voronoi region of EMA I smaller and EMA J and K larger. This property hints at the mechanism of load balancing; DEMA scheduling algorithm has a tendency to make a larger cell (EMA I) smaller and a smaller cell (EMA K) larger.

4.5 Proof of DEMA Load Balancing

In this section, we provide a mathematical exposition of why DEMA balances the workload of application servers when the distribution of input data follows a uniform distribution. For brevity, we assume a 1-dimensional problem space in the proof. However, one can apply the same proof method described below to any higher dimension without difficulty.

Suppose we are given a 1-dimensional problem space as a unit circle (see Figure 3(a)) with five EMA values $\{E_1, E_2, \ldots, E_5\}$ representing application servers $\{S_1, S_2, \ldots, S_5\}$, respectively. Let us denote the middle point of E_i and E_{i+1} by Mi. Then, the Voronoi cell region of $S_i(i > 0)$ is represented by a range $[m_{i-1}, m_i)$ denoted by R_i. We use R_i for both the range and the length of the range interchangeably. In DEMA, all the jobs that need input data whose center falls in R_i are assigned to server S_i. Since the total size of the problem space is one and uniform query distribution is assumed, R_i also represents the probability that a query is assigned to server S_i. Note that $\sum_i R_i = 1$.

In fact, we can view the whole DEMA system as a Time Series; each E_i (thus R_i) is a time series $E_i(t)$ (thus $R_i(t)$) that changes over a discrete time $t = 1, 2, 3, \ldots$ and the transition from t to $t + 1$ happens when a job that needs data point $q(t)$ is assigned to server i. In other words, given the current state $\langle E_1(t), \ldots, E_5(t)\rangle$, the next job that needs a data point $q(t)$ triggers a state transition to $\langle E_1(t+1), \ldots, E_5(t+1)\rangle$. Since the transition only depends on the previous state, the whole DEMA system is also a Markov Chain with discrete time on a general state space [13]. As described below, however we take a simpler approach than a Markov chain analysis to prove load balancing.

Figure 3(b) depicts the sizes of the ranges for all the servers corresponding to Figure 3(a) at time t. In the figure, we note that $R_2(t)$ is a local minimum and $R_4(t)$ is a local maximum, that is, $R_2(t) < R_1(t) \wedge R_2(t) < R_3(t)$, and $R_4(t) > R_3(t)$, $R_4(t) > R_5(t)$. To balance the workloads of the application servers, DEMA algorithm tries to make the assignment probability of all the servers as similar as possible. It does so by, in the next step, decreasing all the local maxima and increasing all the local minima. Formally, we want

$$E[R_2(t+1)] \ > \ R_2(t) \tag{6}$$

and

$$E[R_4(t+1)] \ < \ R_4(t) \tag{7}$$

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

Theorem 1 *For a local maximum $R_i(t)$ at time t, we have*

$$E[R_i(t+1)] \;<\; R_i(t) \tag{8}$$

and for a local minimum $R_i(t)$ at time t, we have

$$E[R_i(t+1)] \;>\; R_i(t). \tag{9}$$

Proof Due to similarity, we only prove Equation 9. In this proof, we directly use the example in Figure 3(a). However, one can easily change the proof to a general case. In the following, when there is no confusion, we omit the time index t for simplicity (e.g., $E_2(t) = E_2$). In Figure 3(a) l_i is the distance from m_{i-1} to E_i and the distance from E_{i-1} to m_{i-1}. Without loss of generality, we assume that $m_5 = m_0 = 0$. Note that we have a local minimum $R_2(t)$; $R_2(t) < R_1(t)$ and $R_2(t) < R_3(t)$, or equivalently

$$\ell_3 < \ell_1 \text{ and} \tag{10}$$
$$\ell_2 < \ell_4. \tag{11}$$

Suppose a new job that needs input data $x \in [0, 1)$ arrives. If x falls outside $\overline{m_5 m_3}$, none of E_1, E_2, and E_3 moves, thus we get $R_2(t+1) = R_2(t)$. If $x \in [0, m_1]$ then $E_1(t+1) = \alpha x + (1-\alpha)E_1(t)$, thus we get

$$m_1(t+1) = \frac{1}{2}(E_1(t+1) + E_2(t+1)) \tag{12}$$
$$= \frac{1}{2}(\alpha x + (1-\alpha)E_1(t) + E_2(t)) \tag{13}$$

If $x \in [m_1, m_2]$, E_2 moves but the length of R_2 remains the same, i.e., $R_2(t+1) = R_2(t)$. When $x \in [m_2, m_3]$, then $E_3(t+1) = \alpha x + (1-\alpha)E_3(t)$, thus we get

$$m_2(t+1) = \frac{1}{2}(E_2(t+1) + E_3(t+1)) \tag{14}$$
$$= \frac{1}{2}(E_2(t) + \alpha x + (1-\alpha)E_3(t)). \tag{15}$$

Now let us compute $E[R_2(t+1)]$. Since input data x follows a uniform distribution in $[0, 1)$, the probability density function of x is $f(x) = 1$, and the cumulative density function is $F(x) = x$. By definition,

$$\begin{aligned} E[R_2(t+1)] = &\int_0^{m_1} m_2 - \frac{1}{2}(\alpha x + (1-\alpha)E_1 + E_2)\,\mathrm{d}x \\ &+ \int_{m_1}^{m_2} R_2 dx \\ &+ \int_{m_2}^{m_3} \frac{1}{2}(E_2 + \alpha x + (1-\alpha)E_3) - m_1\,\mathrm{d}x \\ &+ \int_{m_3}^{1} R_2\,\mathrm{d}x \end{aligned} \tag{16}$$

Since $E_1 = \ell_1, E_2 = \ell_1 + 2\ell_2, E_3 = \ell_1 + 2\ell_2 + 2\ell_3$ and $m_i = E_i + \ell_{i+1}$ for $i = 1, 2, 3$, the average change of R_2 becomes

$$\begin{aligned} E[R_2(t+1)] - R_2(t) &= E[R_2(t+1)] - (\ell_2 + \ell_3) \\ &= \frac{\alpha}{4}(\ell_1^2 - \ell_3^2 + \ell_4^2 - \ell_2^2). \end{aligned} \tag{17}$$

Because of Equation 10 and 11, we have

$$E[R_2(t+1)] - R_2(t) > 0. \tag{18}$$

Therefore, on average, the assignment probability of a local minimum server is more likely to increase at the next query.

It is possible that decreasing a local maximum can make a neighboring interval a new local maximum (and the same applies to local minimum). Nevertheless, subsequent jobs are likely to decrease those new local maxima (or increase new local minima) as shown above. However, this only happens when the assignment probabilities of the local maxima and neighboring EMAs are close to each other, and in that case it ls likely that the overall balance was already achieved.

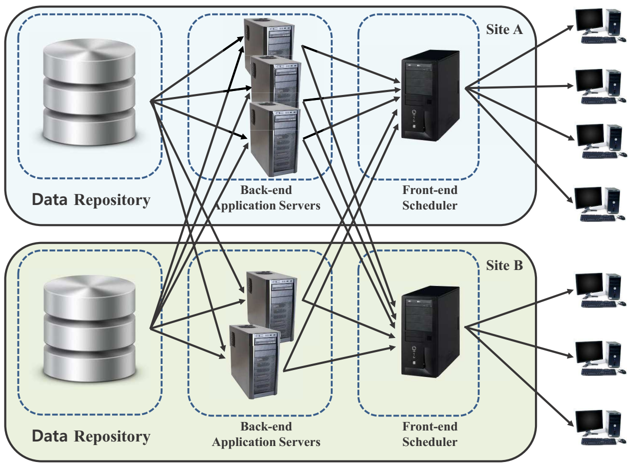


Fig. 4 *Configuration of Multiple Front-end Schedulers for the Geographically Distributed Systems*

5 Collaborative Scheduling in the Cloud

The Cloud environment facilitates collaborative work and allows many clients to execute jobs over geographically distributed computing resources. In such geographically distributed systems, job submission systems had better be near clients to hide network latency when they interact with the system. Thus our distributed query processing framework can be configured to work with multiple schedulers as shown in Figure 4.

With multiple schedulers, a client will submit a job to its closest scheduler, but the scheduler may forward the job to a back-end server which is geographically far remote from the scheduler. all available statistics about

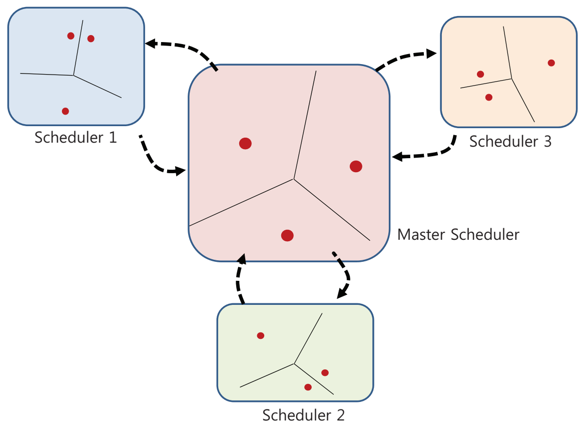


Fig. 5 *Multiple schedulers share EMA points by synchronizing their own EMA points periodically.*

the system such as cached data objects, current system load, network latency between organizations, storage location, etc.

As described in section 4, DEMA scheduling policy makes scheduling decisions based on geometric features of past jobs. One assumption with DEMA scheduling policy is that one schduler has seen all the past jobs, and it predicts which back-end server has what cached data items and how many past jobs were forwarded to which back-end server. With multiple schedulers, this assumption does not hold true any more. One scheduler may receive very different types of jobs from another scheduler. In such cases, EMA points in each schduler can be very far from each other, and it hurts the spatio-temporal locality of the jobs running in each back-end server, and results in poor load balance with low cache hit ratio.

In order to solve the problem, this work compares simple solutions that periodically exchange the EMA points between schedulers, compute the weighted average EMA points of back-end servers, and share the updated EMA points for subsequent incoming jobs as shown in Figure 5. The synchronization process does not have to block processing incoming jobs since unsynchronized EMA points may slow down the job execution time slightly but it might be better to immediately schedule incoming jobs rather than waiting for the synchronization.

Synchronization of EMA Points One of the simple synchronization methods that we study is *averaging method*. In averaging method, each scheduler independently updates its own EMA points, and a master scheduler periodically collects EMA points of other schedulers, computes the average EMA points, and broadcasts the updated EMA points to all other schedulers. The frequency of computing average EMA points should be determined so that network overhead is minimal. But if the frequency is chosen to be too low, EMA points in each scheduler will diverge and the overall system throughput will hurt.

A problem of the averaging method is when the number of scheduled jobs in each schduler varies. For example, if a schduler $S1$ received very few number of jobs while another schduler $S2$ scheduled a large number of recent jobs, the EMA points in $S1$ do not reflect the recent cached data objects in back-end servers. Therefore our improved synchronization method - *weighted averaging method* gives more weight to EMA points of a server which received more queries than other schedulers. The weight is determined by how many jobs were forwarded to each server, i.e., if a back-end server did not receive any job from a scheduler, the weight of EMA point of the back-end server is 0. The weighted average EMA point of each back-end server is calculated as the following equation:

$$EMA_{i^*} = \frac{\sum_{s=1}^{n} N_i[s] \times EMA_{i^*}[s]}{\sum_{s=1}^{n} N_i[s]} \tag{19}$$

where $N_i[s]$ is the number of jobs assigned to server s by scheduler i.

In both synchronization methods, the frequency of synchronization is a critical performance factor that affects the overall system throughput. As we compute the average EMA points more frequently, each scheduler will have more accurate information about cached data objects in back-end servers. If the synchronization interval is very large, each back-end server's buffer cache will be filled with data objects that do not preserve spatio-temporal locality as in round-robin policy.

6 Cached Data Migration

As various incoming queries access different parts of input datasets, EMA points dynamically move to new locations over time. As a result of the movement, some cached data objects may cross the boundaries of Voronoi regions. If the changed Voronoi region does not contain a cached data object any more, the cached data object is not likely to be used by subsequent queries because DEMA scheduler will not forward a query to the server.

Figure 6 shows an example of Voronoi region changes. If the boundary of Voronoi region moves to a lower right direction, some of the cached data objects in server $S2$ will not be covered by the updated $S2$'s region. We will refer to the cached data objects out of current Voronoi region as "misplaced cached data". In the example, a data object $D2$ is a misplaced cached data. Based on

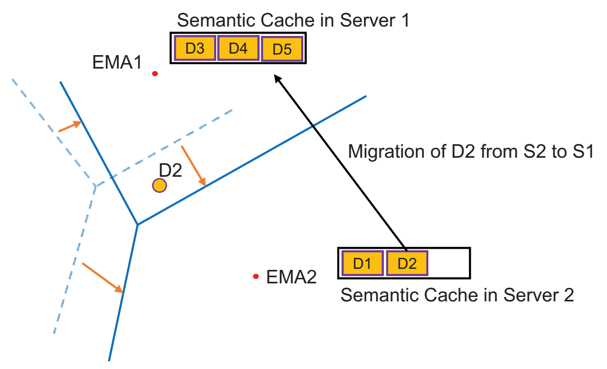

Semantic Cache in Server 1
D3
D4
D5
EMA1
D2
Migration of D2 from S2 to S1
D1
D2
EMA2
Semantic Cache in Server 2

Fig. 6 *An Example of Data Migration: Location of a cached data object D2 is not covered by server S2's region since the boundaries change. If D2 is a frequently accessed data object, it better be moved to server S1.*

the updated boundaries, the scheduler will not forward any incoming query that needs data $D2$ to server $S2$, but instead it will forward it to server $S1$. Although a previously computed query result $D2$ is in a neighbor server $S2$, server $S1$ will read raw datasets from storage systems and process the query from scratch. Since $S2$ is not likely to reuse data object $D2$, $D2$ will be eventually evicted from $S2$'s cache.

In order to manage the distributed semantic buffer cache seamlessly, we propose data migration of cached data objects to improve overall cache hit ratio. With the data migration policy enabled, the misplaced cached data objects are migrated to a remote server whose Voronoi region encloses the cached data objects.

When a front-end server receives a query, it searches for a back-end application server whose EMA point is the closest to the query point. Periodically or when EMA points have moved by a large margin, the front-end server constructs a piggyback message with the server's neighbor EMA points and forwards the query and the neighbor EMA points to the selected back-end application server. With the updated EMA points, back-end application servers can migrate misplaced cached data objects in either pull mode or push mode.

6.1 Pull Mode Migration

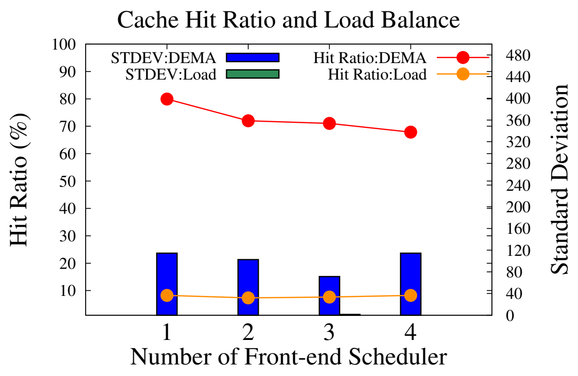
When a back-end application server searches for cached data objects in its own cache but does not find it, pull mode migration policy searches for the cached data objects in its neighbor servers' cache. Since it is very expensive operation to look up neighbor servers' cache for every cache miss, each back-end application server in our framework periodically collects EMA points of its neighbor servers. If a cache miss occurs in a server s, the server checks if the query's point is closer to a neighbor server's historical EMA point rather than its own previous EMA point. If another server n's previous EMA point is closer to the query than its previous EMA point, it means the query would have been assigned to the neighbor server n unless the EMA points moved, and it is highly likely that the neighbor server n has a misplaced cached object for the query. Thus the application server s should check the neighbor server n and pulls the cached data object from n.

The performance of pull mode migration depends on how accurately each application server identifies a neighbor server that has misplaced cached data objects. Searching all neighbor servers for every cache miss can increase cache hit ratio but it will place too much overhead on the cache look up operation, hence we implemented a push mode migration as described below.

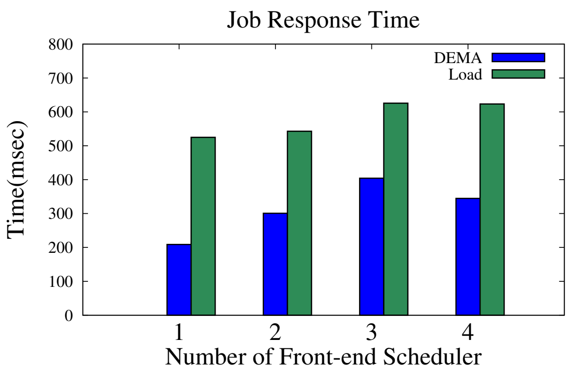
6.2 Push Mode Migration

When a back-end server receives a query, first it searches for its own cached objects in its semantic cache that can be reused. During the look up operation, push mode migration policy investigates whether each cached data object is misplaced or not. If a cached data object is closer to a neighbor EMA point than its own EMA point, the cached data object is considered to be misplaced and we migrate it to a neighbor server. When the neighbor server receives a migrated data object, it determines if the migrated data object should be stored in its cache using its cache replacement policy. Note that our data migration policy does not examine the entire cached data objects since it will cause significant overhead. Instead, we detect misplaced data objects only during cached data look-up operation and transfer them to neighbor servers. I.e., if we use a hash table for cached data look-up, only the cached data objects in an accessed hash bucket will be considered for data migration candidates. This policy migrates some misplaced cached data objects to neighbor servers in a lazy manner, which helps reducing the overhead of cache look-up operation.

As we will show in section 7, our experimental study shows data migration does not always guarantee higher cache hit ratio if the distribution of arriving queries is rather static. With static query distribution, the boundaries of Voronoi regions may fluctuate but not by a large margin. In such cases, cached data objects near boundaries tend to be migrated between two servers repeatedly, and it is possible for them to evict some useful cached data objects continuously in neighbor servers. There can be various solutions to mitigate the negative effect of the continuous eviction problem. One is



(a) Cache Hit Ratio and Load Balance



(b) Job Response Time

Fig. 7 *Performance Comparison Varying Number of Schedulers*

to keep cached data objects unless its center point is not significantly close to a neighbor EMA point. Another option is to replicate the cached data objects on the boundaries by not deleting migrated cached data objects and we mark recently migrated data objects so that the same data objects are not migrated again. In our implementation we chose the second option, which prevents unnecessary repeated LRU cache updates.

7 Experiments

7.1 Query Workload Generator

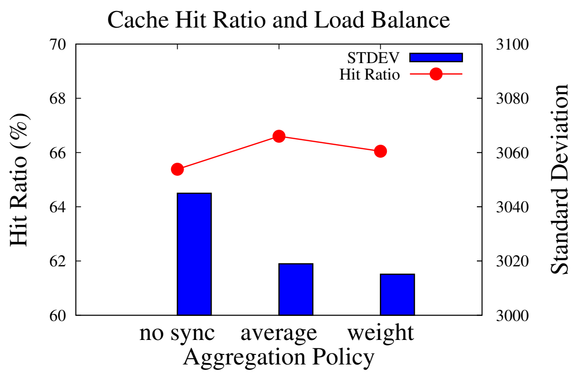
In order to show that the DEMA scheduling policies perform well with various input data distributions in multi-dimensional space, we synthetically generated 2-dimensional queries using uniform, normal, and Zipf's distributions respectively. For all the workloads, the query inter-arrival time distribution was modeled by Poisson process. Since frequently accessed hot data objects in these synthetic workloads do not change over time, we also generated dynamic query workloads that change the requested input data distribution along time dimension. The dynamic query distribution mainly consists of normal distributions, but we made the mean and the standard deviation change every interval of 10,000 queries, i.e., the dynamic query distribution is composed of four successive normal distributions with different mean point and different standard deviation containing 10,000 queries each.

In addition to the synthetic probability distributions, we generated more realistic query workloads using a probability model - Customer Behavior Model Graph (CBMG). In this model, there are a large number of hot multi-dimensional regions of interest, and the first query is selected from one of the regions. Subsequent queries after the first one in the batch may either remain around that point (moving around its neighborhood, temporal movement, or resolution increase or decrease) or move on to a new hot region. This workload model is known to resemble real users' query access patterns and is commonly used to evaluate e-business applications and also used for web site capacity planning [10]. Using the CBMG model, we generated 2-dimensional queries (latitude, longitude) with various transition probabilities, but we only show the results using one of the transition probabilities because results for other transition probabilities were similar to those shown.

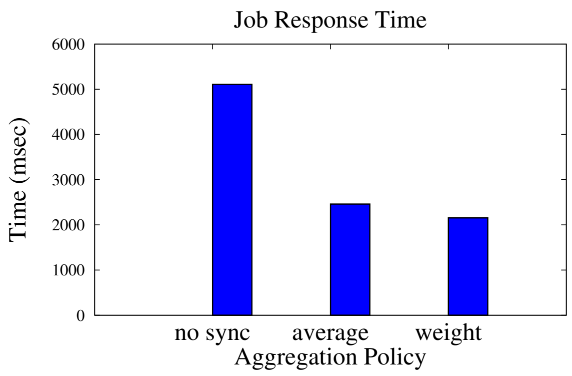
7.2 Multiple Front-end Schedulers Synchronization

For the first set of experiments, we measure the effect of EMA points synchronization with multiple front-end schedulers. In order to measure how periodic synchronization of EMA points adapt to various input data requests, we submit jobs that access different input datasets in various distributions to each scheduler. For the experiments, each scheduler schedules 10,000 jobs that need input dataset tagged with 2-dimensional spatial coordinates, (e.g, latitude and longitude).

In our experiments we employ four homogeneous clusters, each of which consists of a single front-end scheduler and 5 back-end application servers. Note that in our distributed system configuration, a front-end scheduler can assign jobs to any back-end server in remote sites. If the cluster servers are not homogeneous, schedulers should apply weight to the distance between EMA points and input data points so that more powerful servers receive more jobs. As for the network latency between remote sites, our experiments do not consider it but it can be also taken into consideration by applying appropriate weight to the distance between EMA points and input data points.



(a) Cache Hit Ratio and Load Balance



(b) Average Query Response Time

Fig. 8 *The Performance Comparison of Synchronization Policy(5 msec of inter-arrival time)*

7.2.1 Front-end Scheduler Scalability

For the experiments shown in Figure 7, we varied the number of schedulers while the number of back-end application servers is fixed to 20. To measure performance, we use cache hit ratio, load balance, and job response time as performance metrics. Job response time is measured from the moment a job is submitted to a scheduler until the job is finished. Hence the job response time includes the waiting time in a job queue and it depends on the load balance and cache hit ratio. When there is only a single scheduler, the cache hit ratio of DEMA scheduling policy is 80% while the hit ratio of load-based scheduling policy is just 8.25%. As load based scheduling policy solely considers the system load, its standard deviation of scheduled jobs in back-end servers is almost 0 while the standard deviation of DEMA scheduling policy is higher than 50. However, due to very high cache hit ratio, the average job response time of DEMA scheduling policy is about 1.55∼2.51 times faster than that of load-based scheduling policy. Note that as the number of schedulers increases, the cache hit ratio of DEMA scheduling policy decreases slightly because EMA points in each scheduler are likely to diverge with more schedulers. As a result, the job response time of DEMA scheduling policy also increases slightly as the number of scheduler increases, but still it shows significantly lower job response time.

7.2.2 EMA Points Synchronization

Figure 8 and 9 show the job response time, cache hit ratio, and load balancing in terms of the standard deviation of number of processed jobs in each server. Label *average* and *weight* shows the performance of the averaging method and the weighted averaging method respectively that we described in section 5. *nosync* shows the performance of DEMA scheduling policy that does not synchronize the EMA points of multiple schedulers.

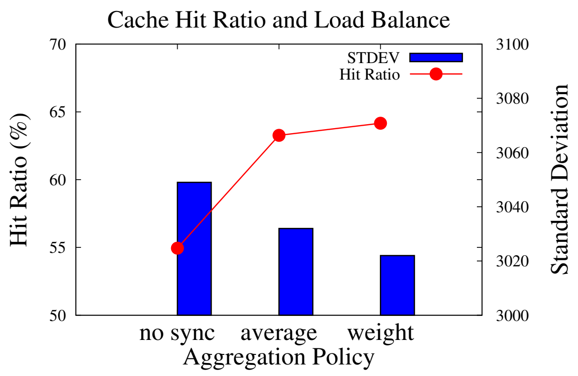
As the ratio of job inter-arrival time and synchronization frequency can affect the effectiveness of synchronizing EMA points, we varied the job inter-arrival time and measured the performance of DEMA scheduling policy. In Figure 8, we fixed the synchronization frequency to 400 msec, and the job inter-arrival time was 5 msec on average.

As expected, the naive *nosync* scheduling policy shows the worst performance because of relatively lower cache hit ratio and higher standard deviation due to the divergence of EMA points across multiple schedulers. The averaging method exhibits the highest cache hit ratio, but weighted averaging method is better in terms of load balance. As a result, weighted averaging method shows the fastest query response time.

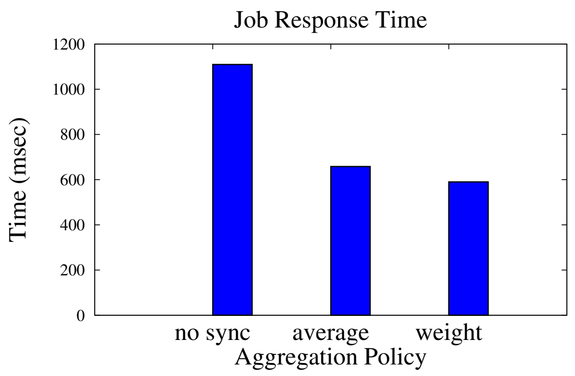
In the experiments shown in Figure 9, the job inter-arrival time was increased to 10 msec while the synchronization frequency was 400 msec. For the same reason, the naive *nosync* scheduling shows the lowest cache hit ratio and the worst load balance resulting in the higest average job response time. The weighted averaging method is again showing best performance in terms of both cache hit ratio and load balance. In both graphs, Figure 8 and Figure 9, the averaging method and weighted averaging method perform significantly better than *nosync* scheduling.

7.2.3 Synchronization Interval

For the experiments shown in Figure 10 and Figure 11 we varied the synchronization interval and measured the performance of the weighted average synchronization methods. As we decrease the synchronization interval, the communication overhead between schedulers increase but the schedulers can make scheduling deci-

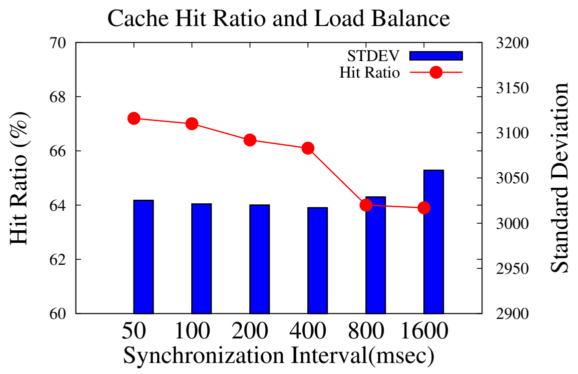


(a) Cache Hit Ratio and Load Balance

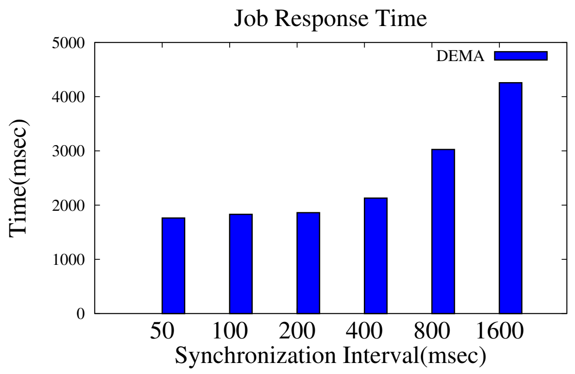


(b) Average Query Response Time

Fig. 9 *The Performance Comparison of Synchronization Policy(10 msec of inter-arrival time)*

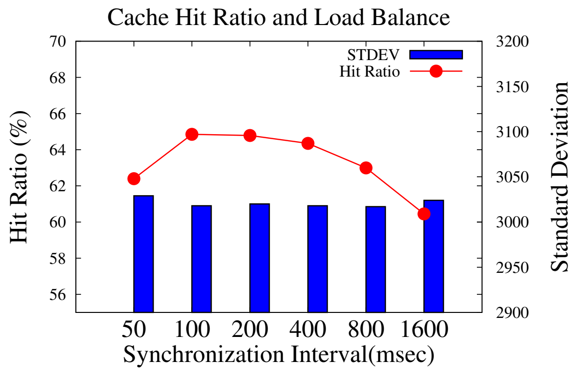


(a) Cache Hit Ratio and Load Balance

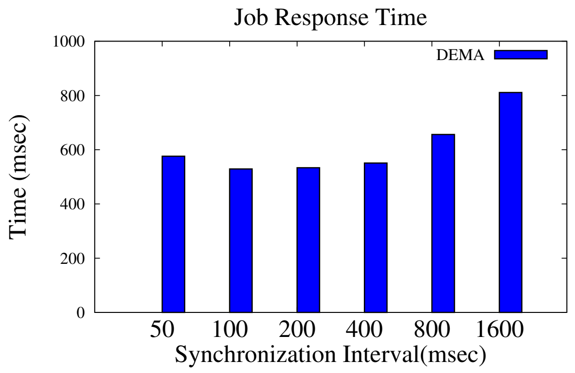


(b) Average Query Response Time

Fig. 10 *The Performance Effect of Synchronization Interval(5 msec of inter-arrival time)*



(a) Cache Hit Ratio and Load Balance



(b) Average Query Response Time

Fig. 11 *The Performance Effect of Synchronization Interval(10 msec of inter-arrival time)*

sions with more accurate prediction of cached data objects.

Fig 10 shows the cache hit ratio, standard deviation of number of processed jobs in each server, and the average job response time with varying the synchronization interval. The job inter-arrival time is fixed to 5 msec. When the synchronization frequency is too high (synchronization per 4 jobs), system load balance is slightly worse than lower synchronization frequency, but its cache hit ratio is highest among others. But as the synchronization interval increases, the cache hit ratio decreases and the load balance also becomes poor. In terms of the job response time, we observe the fastest job response time when synchronization interval is 50. With too high synchronization interval, the response time significantly increases because outdated EMA points hurt cache hit ratio and load balance, but too frequent synchronization also does not help improving job response time due to synchronization overhead.

Fig 11 shows the performance with varying the synchronization interval when the average job response time is 10 msec. Since the job inter-arrival time is doubled compared to the experiments shown in Fig 10, it shows the best performance when the synchronization interval is 100. Similarly, load balancing is not significantly affected by the synchronization interval but the cache hit ratio decreases as the synchronization interval increases.

7.3 Cached Data Migration

To evaluate the performance gain from cached data migration, we measure cache hit ratio and query wait execution time using various workloads. Each workload, composed of 40,000 2-dimensional queries, is generated using uniform, normal, and Zipf's distributions and mixture of those distributions. Workload based on the CBMG is also used to consider the realistic environment.

7.3.1 Scalability

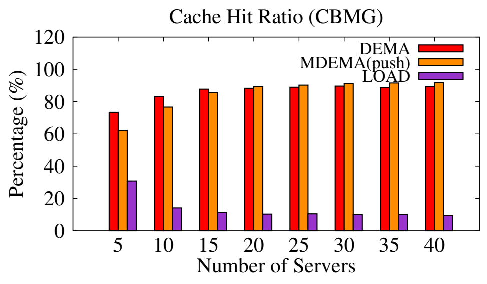
Figure 12 depicts the average query response time of 40,000 queries and cache hit ratio for DEMA scheduling (*DEMA*), DEMA scheduling with data migration (*MDEMA*), and load-based scheduling (*LOAD*), as the number of back-end application servers increases. The query response time is defined as the amount of time from the moment a query is submitted to the scheduler until it completes in a back-end application server, i.e., it includes the waiting time in the application server's queue as well as the actual processing time. The query inter-arrival time was modeled by Poisson Process with average of 1 ms, and each back-end application server has a cache size that can hold up to 200 query results. For the rest of the experiments, we fixed the EMA smoothing factor α to 0.03. The smoothing factor determines how much weight is given to each past query result. When α is 0.03, the recent 200 query results account for 99.7% of the weight in the EMA equation.

As the number of servers increases, DEMA scheduling policy improves the cache hit ratio since the total available cache size in distributed caching system increases. However, Load-based scheduling policy does not get any benefit of the increased cache size but its cache hit ratio decreases because frequently requested cached data objects are scattered across more back-end application servers and load-based scheduling policy does not know which remote cache has what cached data objects. Hence the performance gap between load-based scheduling policy and DEMA scheduling policy becomes wider as the number of servers increases. Due to its low cache hit ratio, the average query response time of load-based scheduling policy is order of magnitude higher than that of DEMA scheduling policy.

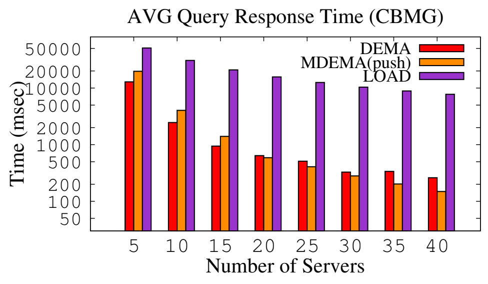
With a small number of servers, the total size of distributed caching system is not enough to hold all frequently accessed data objects, hence the migration of misplaced cached objects evicts other frequently accessed data objects in neighbor servers. It results in lowering overall cache hit ratio in our experiments, thus DEMA scheduling policy that does not migrate misplaced cached objects outperforms MDEMA. However with a larger number of servers, the total size of distributed caching system becomes large enough to hold most of the frequently accessed data objects. If each application server's cache is large enough, some of the cached data objects are less likely to be accessed than others, thus our cache migration policy evicts such less popular data objects and replaces them with more popular data objects from neighbor servers, which are highly likely to be accessed by subsequent queries. Thus with more than 25 back-end application servers, DEMA with data migration policy outperforms DEMA scheduling policy.

7.3.2 Cache Miss Overhead

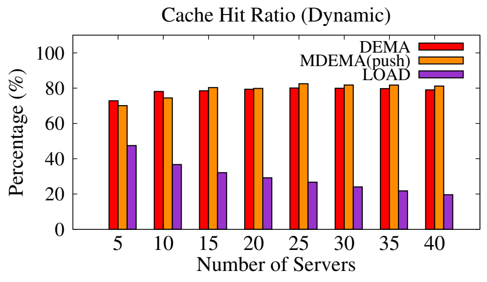
Figure 13 depicts the average query response time for uniform, normal, Zipf's, and dynamic input data distributions with varying the cache miss penalty. If a query needs a larger raw data object and more computation, cache miss causes a higher performance penalty. For the experiments, we fixed the cache size so that it can hold at most 200 query results, and employed 40 back-end application servers. In the Figure 13, we do not present the cache hit ratio and the standard deviation of server loads since the effect of cache miss penalty is not relevant to the cache hit ratio and system load balance. As the cache miss overhead increases, the average query response time linearly increases not only because of the cache miss penalty but also because the waiting time in the queue increases. The waiting time increases because previously scheduled queries suffer from higher cache miss penalty and more queries will be accumulated in the queue. In the experiments, average query response time with load-based scheduling policy is up to 72 times higher than DEMA scheduling policy. When data migration is employed with DEMA scheduling policy the hit ratio improves from 57$\sim$83% to 55$\sim$92%, thus it exhibits up to 40% lower average query response time. Load-based scheduling policy ex-



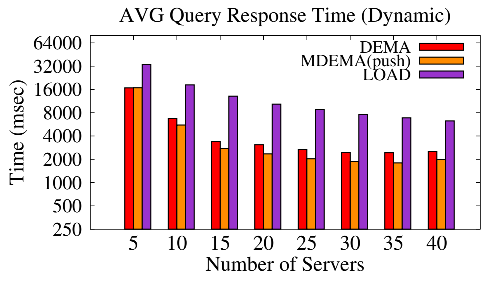
(a) Cache Hit Ratio (Realistic CBMG)



(b) Query Response Time (Realistic CBMG)

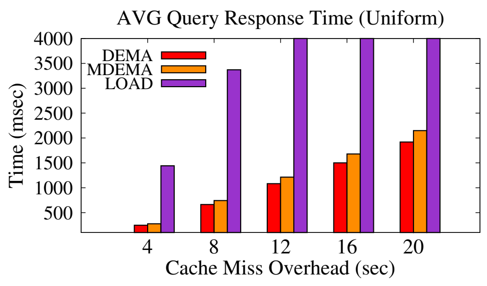


(c) Cache Hit Ratio (Dynamic)

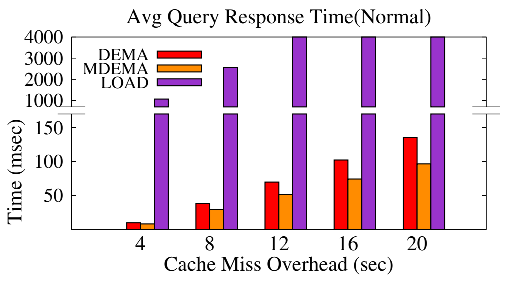


(d) Query Response Time (Dynamic)

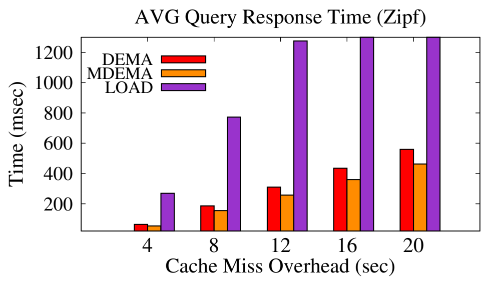
Fig. 12 *Performance Comparison Varying Number of Servers*



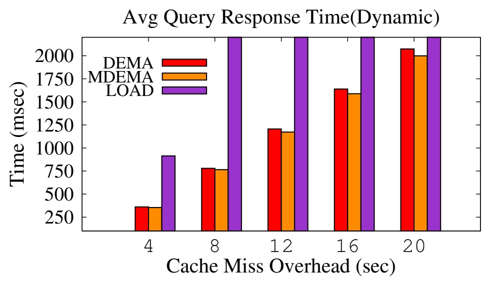
(a) Query Response Time (Uniform Distribution)



(b) Query Response Time (Normal Distribution)



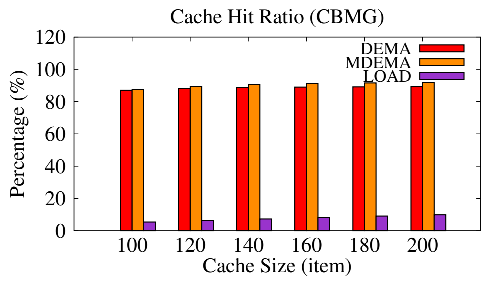
(c) Query Response Time (Zipf Distribution)



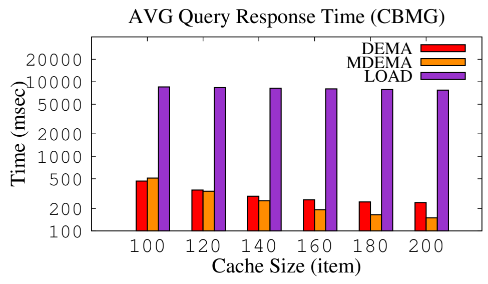
(d) Query Response Time (Dynamic Distribution)

Fig. 13 *The Effect of Cache Miss Overhead*

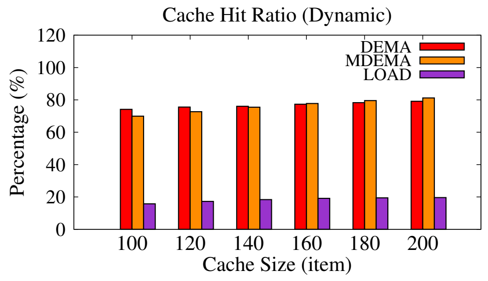
hibits significantly higher average query response time because of low cache hit ratio. Hence note that y-axis in Figure 13(b) is broken into two ranges to show the huge performance gap between load-based scheduling policy and DEMA scheduling policy. The graphs in Figure 13 show the average query response time of DEMA and MDEMA is mostly smaller than the cache miss penalty, i.e, most queries reuse the cached data objects in distributed caching system and they are executed immediately without being accumulated in the waiting queue. But load-based scheduling policy suffers from low cache hit ratio and the cache miss penalty is higher than query inter-arrival time, thus its average query response time is much higher than the query inter-arrival time because the queries wait in the queue for significant amount of time.



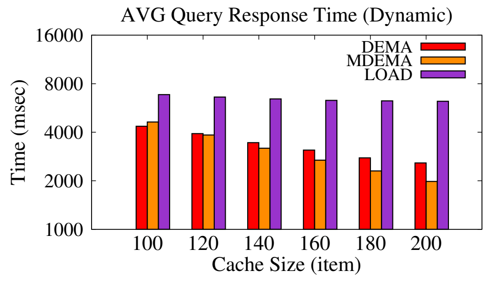
(a) Cache Hit Ratio (Realistic CBMG)



(b) Query Response Time (Realistic CBMG)



(c) Cache Hit Ratio (Dynamic)



(d) Query Response Time (Dynamic)

Fig. 14 *Effect of Cache Size with Realistic CBMG Workload and Dynamic Workload*

7.3.3 Cache Size Effect

Now we consider the impact of cache size to the system. We ran experiments with 40 back-end application servers, and fixed the EMA smoothing factor to 0.03 for DEMA policy. The cache miss penalty was 20 ms. In the graphs shown in Figure 14, Load-based scheduling policy gains the benefit of increased cache size. Note that load-based scheduling policy didn't get benefit of increased number of distributed caches. With non-intelligent scheduling policies such as round-robin or load-based scheduling policy, each application server should have very large cache in order to exhibit high cache hit ratio.

Unlike load-based scheduling policy, DEMA scheduling policy consistently shows very high cache hit ratio ($> 80\%$) even when cache size is small. This is because DEMA scheduling policy considers all cached objects in distributed caching system. Thus even if a single server's cache size is small, the total size of distributed caching system is not small. And DEMA scheduling policy assigns each query to a back-end application server which is highly likely to have the requested data object in its cache even when the cache size is small.

It should be noted that the cache hit ratio is totally dependent on query workloads. If certain query workloads do not have query sub-expression commonality, even DEMA scheduling policy may have low cache hit ratio. However, load-based scheduling policies will have even lower cache hit ratio than DEMA. If cache hit ratio is low, a large number of queries have to be computed from the scratch. In such situations, load balancing plays an important role in decreasing the wait time in the queue. DEMA scheduling policy shows higher standard deviation of the number of processed queries in each server than load-based scheduling policy, but since cache hit does not put significantly high overhead to back-end application server, we counted only the number of missed queries in each server and DEMA scheduling policy exhibited very good load balancing behavior.

In Figure 14, the benefit of migrating misplaced cached objects is not clearly observable when cache sizes are small. Again this is because the migration may evict other frequently accessed data objects in neighbor servers. However, as the cache size increases, data migration improves the cache hit ratio and it results in faster query response time.

8 Conclusion

This paper presents a multiple query scheduling policy for distributed query processing framework that takes into consideration the dynamic contents of distributed caching infrastructure. 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. In order to achieve load balancing as well as to exploit cached query results, it is required to employ more intelligent query scheduling policies than the traditional round-robin and load-based scheduling policies.

We propose *DEMA* for this purpose which makes scheduling decision based on spatial locality of clustered queries. And we discuss how the scheduling policy can be used when multiple schedulers collaborate managing distributed computing resources in the cloud environment. And for better cached data utilization, our framework implemented a cached data object migration module that helps distributed caches cooperate to further improve cache hit ratio.

Our experiments show the distributed job scheduling policy and its simple extension for multiple schedulers and the cached data migration outperform legacy load based scheduling policy in terms of cache hit ratio and load balancing. And the scheduling policy with data migration policy also outperforms legacy scheduling policies by further increasing the cache hit ratio of the framework.

References

[1] Andrade H, Kurc T, Sussman A, Saltz J (2002) Active Proxy-G: Optimizing the query execution process in the Grid. In: Proceedings of the ACM/IEEE SC2002 Conference

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

[3] de Berg M, Cheong O, van Kreveld M, Overmars M (1998) Computational Geometry, Algorithms and Applications. Springer

[4] Beynon MD, Ferreira R, Kurc T, Sussman A, Saltz J (2000) DataCutter: Middleware for filtering very large scientific datasets on archival storage systems. In: Proceedings of the Eighth Goddard Conference on Mass Storage Systems and Technologies/17th IEEE Symposium on Mass Storage Systems, pp 119–133

[5] Catalyurek UV, Boman EG, Devine KD, Bozdag D, Heaphy RT, Riesen LA (2009) A repartitioning hypergraph model for dynamic load balancing. Journal of Parallel and Distributed Computing 69(8):711–724

[6] Godfrey B, Lakshminarayanan K, Surana S, Karp R, Stoica I (2004) Load balancing in dynamic structured p2p systems. In: Proceedings of INFOCOM 2004

[7] Katevenis M, Sidiropoulos S, Courcoubetis C (1991) Weighted round-robin cell multiplexing in a general-purpose atm switch chip. IEEE Journal on Selected Areas in Communications 9(8):1265–1279

[8] Kim JS, Andrade H, Sussman A (2007) Principles for designing data-/compute-intensive distributed applications and middleware systems for heterogeneous environments. Journal of Parallel and Distributed Computing 67(7):755–771

[9] Kurc T, Chang C, Ferreira R, Sussman A, Saltz J (1999) Querying very large multi-dimensional datasets in ADR. In: Proceedings of the ACM/IEEE SC1999 Conference

[10] Menasce DA, Almeida VAF (2000) Scaling for E-Business: Technologies, Models, Performance, and Capacity Planning. Prentice Hall PTR

[11] Nam B, Shin M, Andrade H, Sussman A (2010) Multiple query scheduling for distributed semantic caches. Journal of Parallel and Distributed Computing 70(5):598–611

[12] Pai V, Aron M, Banga G, Svendsen M, Druschel P, Zwaenepoel W, Nahum E (1998) Locality-aware request distribution in cluster-based network servers. In: Proceedings of ACM ASPLOS

[13] Robert CP, Casella G (2005) Monte Carlo Statistical Methods (Springer Texts in Statistics). Springer-Verlag New York, Inc., Secaucus, NJ, USA

[14] Rodríguez-Martínez M, Roussopoulos N (2000) Mocha: A self-extensible database middleware system for distributed data sources. In: Proceedings of 2000 ACM SIGMOD, ACMPRESS, pp 213–224, aCM SIGMOD Record, Vol. 29, No. 2

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

[16] Vydyanathan N, Krishnamoorthy S, Sabin G, Catalyurek U, Kurc T, Sadayappan P, Saltz J (2009) An integrated approach to locality-conscious processor allocation and scheduling of mixed-parallel applications. IEEE Transactions on Parallel and Distributed Systems 15:3319–3332

[17] Wolf JL, Yu PS (2001) Load balancing for clustered web farms. ACM SIGMETRICS Performance Evaluation Review 28(4):11–13

[18] Zhang K, Andrade H, Raschid L, Sussman A (2005a) 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

[19] Zhang Q, Riska A, Sun W, Smirni E, Ciardo G (2005b) Workload-aware load balancing for clustered web servers. IEEE Transactions on Parallel and Distributed Systems 16(3):219–233

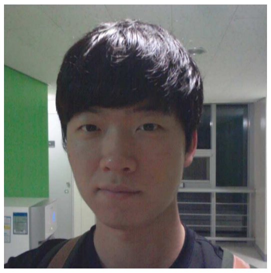


Fig. 15 Youngmoon Eom is a graduate student at Ulsan National Institute of Science and Technology (UNIST), Republic of Korea. He earned his B.S. in Electrical and Computer Engineering in 2013 from UNIST. He was a visiting intern student in computer science deparment at the University of Maryland, College Park in 2011 summer. His research interests are in high performance computing, virtual machine technologies, big data processing platforms, and cloud computing.

Fig. 16 Jinwoong Kim is a graduate student in school of Electrical and Computer Engineeering at UNIST (Ulsan National Institute of Science and Technology) in Korea. He received his B.S. degree in computer science at Chungbuk National University, Korea in 2011. His research interests include parallel algorithms, GPGPU computing, and distributed and parallel systems.

Fig. 17 Beomseok Nam is an assistant professor in school of Electrical and Computer Engineering at UNIST (Ulsan National Institute of Science and Technology) in Korea. Before he joined UNIST, he was a senior member of technical staff at Oracle in Redwood Shores, CA. Nam received his Ph.D. in Computer Science at the University of Maryland, College Park in 2007, and a B.S. and an M.S. degree from Seoul National University, Korea. His research interests include data-intensive computing, database systems, and distributed and parallel high performance computing.