Locality-Aware Fair Scheduling in the Distributed Query Processing Framework

Youngmoon Eom

Department of Computer Engineering

Graduate School of UNIST

Locality-Aware Fair Scheduling in the Distributed Query Processing Framework

A thesis
submitted to the Graduate School of UNIST
in partial fulfillment of the
requirements for the degree of
Master of Science

Youngmoon Eom

12. 15. 2014

Approved by

[illegible]

Advisor
Beomseok Nam

Locality-Aware Fair Scheduling in the Distributed Query Processing Framework

Youngmoon Eom

This certifies that the thesis/dissertation of Youngmoon Eom is approved.

12. 15. 2014

[illegible]

Advisor: Beomseok Nam

[illegible]

Young-ri Choi: Thesis committee Member #1

[illegible]

Woongki Baek: Thesis committee Member #2

Abstract

Utilizing caching facilities in modern query processing systems is getting more important as the capacity of main memory is having been greatly increasing. Especially in the data intensive applications, caching effect gives significant performance gain avoiding disk I/O which is highly expensive than memory access. Therefore data must be carefully distributed across back-end application servers to get advantages from caching as much as possible.

On the other hand, load balance across back-end application servers is another concern the scheduler must consider. Serious load imbalance may result in poor performance even if the cache hit ratio is high. And the fact that scheduling decision which raises cache hit ratio sometimes results in load imbalance even makes it harder to make scheduling decision. Therefore we should find a scheduling algorithm which balances trade-off between load balance and cache hit ratio successfully.

To consider both cache hit and load balance, we propose two semantic caching mechanisms DEMB and EM-KDE which successfully balance the load while keeping high cache hit ratio by analyzing and predicting trend of query arrival patterns.

Another concern discussed in this paper is the environment with multiple front-end schedulers. Each scheduler can have different query arrival pattern from users. To reflect those differences of query arrival pattern from each front-end scheduler, we compare 3 algorithms which aggregate the query arrival pattern information from each front-end scheduler and evaluate them.

To increase cache hit ratio in semantic caching scheduling further, migrating contents of cache to nearby server is proposed. We can increase cache hit count if data can be dynamically migrated to the server where the subsequent data requests supposed to be forwarded. Several migrating policies and their pros and cons will be discussed later.

Finally, we introduce a MapReduce framework called Eclipse which takes full advantages from semantic caching scheduling algorithm mentioned above. We show that Eclipse outperforms other MapReduce frameworks in most evaluations.

Contents

List of Figures

I. Introduction

The data we are dealing with grows exponentially, leading to era of so-called Big Data. Since Big Data cannot be simply processed with traditional way of computing, a number of distributed query processing techniques have emerged. Cloud computing, cluster computing, grid computing, MapReduce are examples of them.

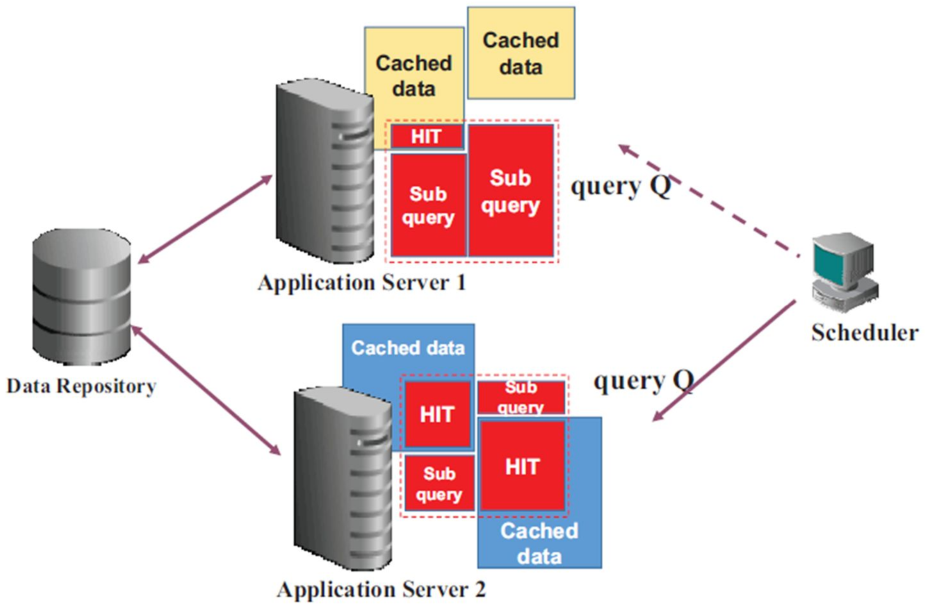


Figure 1. Architecture of Distributed Query Processing Framework

The architecture of distributed query processing framework is shown in Figure 1. The clients submit their queries to the front-end scheduler and queries are scheduled to dedicated back-end application servers. If the data requested by a query is not in the cache memory of back-end application server, the data should be read from data repository which takes relatively long time. If a subsequent query requests same data, the data can be accessed directly from the cache memory saving time to read remotely from data repository. Although the data described in the figure is represented as two-dimensional data, the data can be simply one-dimensional data or it can be easily extended to high dimensions.

There are many aspects to consider when we design distributed query processing system. One important factor among them is the utilization of cache. Since the caching facilities are distributed over multiple back-end application servers, we get different cache hit ratio from different data placement policies. Therefore data must be carefully distributed as caching effect gives significant performance gain avoiding disk I/O which is highly expensive than memory access.

After a query is processed in a back-end application server, the data consumed by the query is stored in the cache memory of the server. If the same data is requested by subsequent queries, the data will be read from the cache memory unless it has been evicted out by data replacement policy of the cache memory. In this condition, similar queries in terms of data locality should be scheduled to the same server for higher probability of cache hit.

We can utilize cache memory of back-end application servers more efficiently with semantic caching. As shown in the Fig 1, data requested by a range query can be divided into smaller sub-queries. And each sub-query can be candidate of cache hit. In this way, we can expect partial cache hit even if entire data requested by the range query is not in the cache memory.

Another important factor for the performance of distributed query processing system is load balance of back-end application servers. Serious load imbalance may result in poor performance even if the cache hit ratio is high. For example, distributing thousands of same queries evenly will give better performance in terms of query wait execution time than scheduling every query to one back-end application server just to increase the cache hit count.

If we know which queries will be submitted in the future, we can find an optimal scheduling which gives the best load balance and the cache hit ratio leading the shortest query wait execution time. But there is no efficient algorithm to find such scheduling since it is known to be an NP-hard problem. Furthermore, it is meaningless to schedule after we get all query requests especially if the framework is an interactive system. Therefore we need a prediction mechanism for the future query arrival pattern. Without prediction, it is highly likely to get either bad load balance or low cache hit ratio as popular data might be cached in few back-end application servers so that we need to sacrifice either cache miss penalty for load balance, or load balance for cache hit.

There are many concerns for the prediction of future query arrival pattern to get better load balance and cache hit ratio. One difficult problem is that query arrival pattern highly depends on time. In other words, query arrival pattern shows different trends at each time period involving either gradual changes, rapid changes or both.

If we choose to adapt quickly to current query arrival pattern, we will get better load balance as scheduler actively balance the load according to the current pattern. However, since many of previously cached data are not matched to the current arriving queries, cache cannot be fully utilized.

On the other hand, we will get higher cache hit ratio if we gradually adapt to query arrival pattern trends. However, we might get bad load balance because the scheduling decision is little behind the current query arrival trend, failing to correctly estimate popular data and unpopular data.

Therefore, scheduling which raises cache hit ratio sometimes breaks the load balance, and scheduling which balance the load sometimes lower the cache hit ratio. As an extreme case, the cache hit ratio of Round Robin scheduling algorithm is generally very low although it shows a perfect load balance.

Considering aspects explained above, DEMB (Distributed Exponential Moving Boundaries) scheduling algorithm is proposed to get good load balance and high cache hit ratio which introduces the Hilbert space filling curve to represent multi-dimensional problem space into one-dimensional problem space. Using a simple doubly linked list, DEMB successfully manages query histogram in a well-partitioned way so that load is well-balanced while high cache hit ratio is achieved. DEMB is also good at adapting to a big change on a query arrival pattern. In case of previous scheduling policies like DEMA (Distributed Exponential Moving Averages), once trained with a specific query arrival pattern, it takes quite long time to adapt to new query arrival pattern if it shows completely different distribution. Contrary to case of DEMA, DEMB can adjust its scheduling according to the current query arrival pattern very rapidly using fixed set of most recent queries as its histogram.

Although DEMB scheduling policy shows good load balance and cache hit ratio, considerable scheduling overhead can be incurred if window size of the query histogram is set to too large value for some reason. For that problem, we propose EM-KDE (Exponential Moving-Kernel Density Estimation) scheduling algorithm as the improved version of DEMB scheduling algorithm. The EM-KDE scheduling uses an array with fixed size rather than a doubly linked list as in DEMB scheduling policy. Using fixed number of histogram bins, EM-KDE can avoid overhead from having large window size of query histogram and it successfully approximates kernel density estimation of the recent query arrival pattern. EM-KDE scheduling performs as well as DEMB without heavy overhead even with large window size of query histogram.

Another concern discussed in this paper is the environment with multiple front-end schedulers. If back-end servers are distributed to multiple sites and the sites are physically distant from each other, it is difficult to manage scheduling queries to all back-end servers at a single front-end scheduler due to the heavy network overhead. To solve the problem, multiple front-end schedulers can be used to get query requests from clients of each site in a distributed manner and schedule queries from them to back-end application servers independently from other remote site.

In this scenario, scheduling independently from other remote site will break the data locality if the query arrival pattern of each site is significantly different, resulting the decreased cache hit ratio. To prevent this problem, we synchronize the scheduling information of each site periodically to maintain the data locality. Two synchronization methods (Averaging, Weighted Averaging) are introduced in this paper. And their performances are compared with a default scheduling in the later section.

Both DEMB and EM-KDE focus on scheduling queries to maintain high cache hit ratio while balancing the load. We can increase cache hit ratio further by migrating the data in a back-end application server to another server. During scheduling is being adjusted to the current query arrival pattern, scheduling of a query can be changed to different server. However, the cached data requested by the query remain in the previously scheduled server. Therefore we cannot utilize the cached data even if the data is in the cache memory of the cluster when the scheduling of a query is changed as a result of adaptation to current query arrival pattern.

To mitigate such a chance of cache miss, we propose two kinds of migration policy for higher cache hit ratio. One is the pull mode migration policy which enables a back-end server to search for the data requested by a query in its neighbor server. The other policy is the push mode migration policy in which a cached data can be migrated to a neighbor server during the look up operation. The evaluation of each migration mode policy will be shown in the later section.

As our last work in this paper, a MapReduce framework called Eclipse is designed and implemented to take full advantages from semantic caching scheduling algorithm explained above. Using the one-dimensional index for each input data, the map tasks are scheduled according to the EM-KDE scheduling policy and the data is cached into the main memory of the scheduled server. Since the intermediate data is also tagged with its key, intermediate data also can be represented in a one-dimensional index. This feature derives the proactive shuffling which eliminates the need for the sort during the shuffle phase. As shuffle phase is sometimes main bottleneck in existing MapReduce frameworks, proactive shuffling gives a large performance gain. Also the intermediate caching reduces the time to process map tasks significantly with some applications. We show that Eclipse outperforms other MapReduce frameworks in most evaluations.

II. Background & Related Work

Zhang et al. and Wolf et al. proposed scheduling policies that dynamically distribute incoming requests for clustered web servers. WRR(Weighted Round Robin) proposed by Katevenis et al. 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 servers 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) 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) is a locality-aware scheduling policy for multi-dimensional 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) is an improved version of DEMA that is also a locality-aware scheduling policy.

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. Zhang et al. 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, DataCutter, Polar, ADR and Active Proxy-G. 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. 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.

As the demand for large-scale data analysis frameworks grew in high performance computing community in the late '90s, several distributed and parallel data analysis frameworks such as Active Data Repository that supports map/reduce programming paradigm and DataCutter that supports generic DAG workflows, were developed for large scale scientific datasets. A few years later, the growing demand for large-scale data processing application from the industry made Google develop a similar MapReduce framework. Since then, there has been a great amount of efforts to extend and improve the MapReduce framework for various data-intensive applications.

Spark shares the same goal with our framework in that it reuses a working set of data across multiple parallel operations. Resilient Distributed Datasets (RDDs) in Spark are read-only in-memory data objects that can be reused for multiple MapReduce tasks. Spark addresses the conflict between job scheduling fairness with data locality by delaying a job for a small amount of time if the job cannot launch a local task. Our Eclipse job scheduling is different from Spark in the aspect of managing the cached data objects, i.e., Spark requires programmers to explicitly control the placement of RDDs while Eclipse autonomously places the cached data objects based on system load and job distribution. Dcache is another MapReduce framework where a central cache manager uses its best efforts to reuse the cached results of previous jobs. Compared to Dcache, Eclipse is more scalable as it does not have a central directory that keeps the list of cached data objects.

Main Memory MapReduce(M3R) proposed by Shinnar et al. is a MapReduce framework that performs in-memory shuffle by simply storing intermediate results of map tasks in main memory instead of the block device storage. They show in-memory shuffle significantly improves a certain type of applications. However, M3R cannot be used if workloads are large and do not fit in main memory and if applications require resilience since in-memory framework is not fault tolerant. Moreover, it is questionable if MapReduce is the right programming paradigm for their target application - sparse matrix vector multiplication.

DryadInc is an incremental computation framework that allows computations to reuse partial results of the computations from previous runs. The incremental computation and semantic caching approaches for data reuse have been extensively studied in computer systems, database systems, and programming languages community for the past decades. Tiwari et al. proposed MapReuse delta engine for in-memory MapReduce framework that detects input data similarity and reuses available cached intermediate results and computes only for a new portion of input data. They show the reuse of intermediate results significantly improves job execution time. ReStore is another framework that stores intermediate results generated by map tasks in HDFS so that they can be reused by future submitted jobs. Their works are similar to Eclipse but the conventional fair job scheduling policies they used do not guarantee balancing the workloads especially if requested intermediate results are available only in a small number of overloaded servers.

MRShare framework proposed by Nykiel et al. merges a batch of MapReduce queries into a single query so that it takes the benefits of sharing input and output data across multiple queries. The multiple queries optimization problem has been studied for the past decades, and it has been proved to be an NP problem. Nevertheless, extensive research has been conducted to minimize query processing time through data and computation reuse using heuristics or probabilistic efforts. These works that reorder the execution of submitted jobs are complementary to our design.

One of the well-known drawbacks of MapReduce frameworks is that it does not directly support iterative data analysis applications, and another drawback is that it does not support complicated data workflows. Dryad is more generic than MapReduce since it defines computations as directed acyclic graphs but it does not support iterative or recursive computation. In high performance computing community, distributed processing frameworks that support generic DAG workflow have been studied for large scale scientific datasets in the past years. Ajira is another distributed and parallel processing framework that implements MapReduce programming model and supports a generic workflow for streaming data processing. Eclipse efficiently supports iterative data analysis algorithm such as PageRank, but it currently does not support generic DAG workflow. We plan to extend Eclipse so that generic workflow computations can be deployed on top of our proposed seamless distributed semantic caching layer.

III. Distributed Exponential Moving Boundary

3.1. DEMA Scheduling Policy

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

$$EMA_t = alpha*data_t + (1 - alpha)*EMA_{t-1}$$

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

The Distributed Exponential Moving Average (DEMA) scheduling policy estimates the cache contents of each back-end server using an exponential moving average (EMA) of past queries executed at that server. In the context of a multi-dimensional range query scheduling policy, we use the multi-dimensional center point of the query as $data_t$ in Equation 1.

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

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

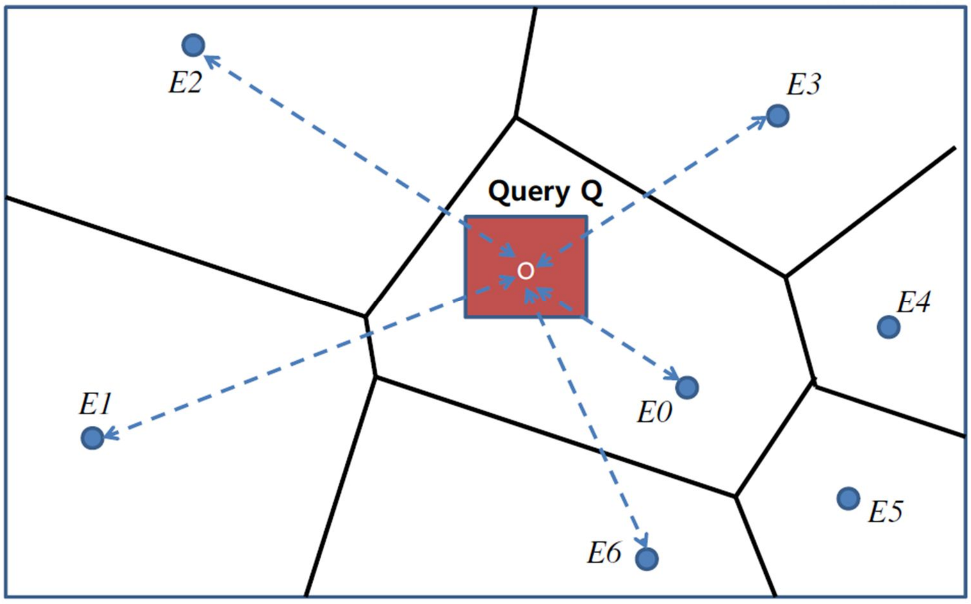
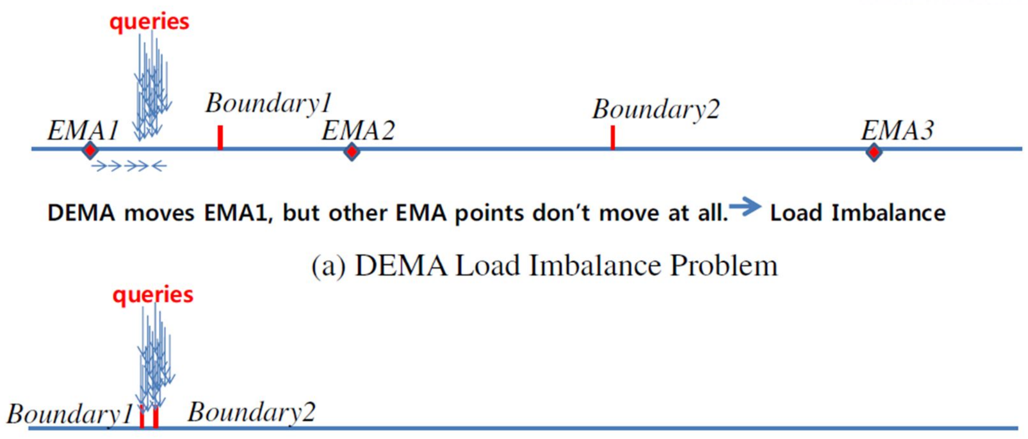


Figure 2. The DEMA scheduler calculates the Euclidean distance between EMA points and an incoming query, and assigns the query to the server (0) whose EMA point is closest.

In the DEMA query scheduling policy the front-end query scheduler server has one multi-dimensional EMA point for each back-end application server. For the 2-dimensional image, each query specifies ranges in the x and y dimensions, as shown in Figure 2. For an incoming query, the front-end server calculates the Euclidean distance between the center of the multi-dimensional range query and the EMA points of the N application servers, and chooses an application server whose current EMA point is the closest to that of the incoming query. Clustering similar range queries increases the probability of overlap between multi-dimensional range queries, and increases the cache hit rate at each back-end server. In Figure 2, the given two-dimensional range query will be forwarded to server 0 since the EMA point of server 0 is closer to the query than any other EMA point. This strategy is called the Voronoi assignment model, where every multi-dimensional point is assigned to the nearest cell point. The query assignment regions induced from the DEMA query assignment form a Voronoi diagram.

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


queries
Boundary1
Boundary2
EMA1
EMA2
EMA3
DEMA moves EMA1, but other EMA points don't move at all. Load Imbalance
(a) DEMA Load Imbalance Problem
queries
Boundary1
Boundary2

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

Figure 3. DEMA Load Imbalance Problem

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

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

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

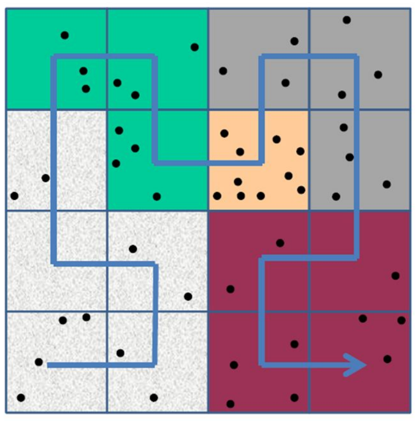
One important property of the DEMA scheduling policy is that an EMA point tends to move to the center of its Voronoi cell if queries arrive with a uniform distribution. In Figure 2, EMA point E0 is located in the lower right corner of Vor(A). Since more queries will arrive in the larger part of the cell(i.e.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

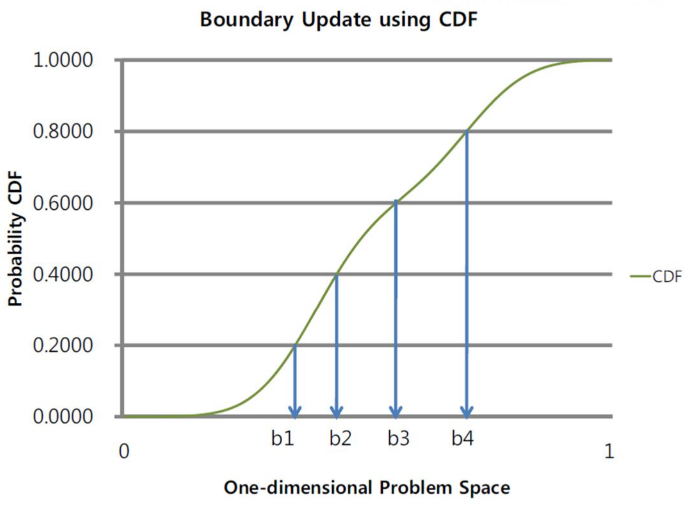
A normal distribution is another commonly occurring family of statistical probability distributions. It is known that the sum of normally distributed random variables is also normally distributed. If we assume that each historical query point $data_t$ is an individual random variable, the weighted sum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

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

3.2. DEMB: Distributed Exponential Moving Boundary

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


(a) Mapping cache contents on Hilbert curve



(b) Calculating Boundaries using CDF

Figure 4. Calculating DEMB boundaries on Hilbert curve

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

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

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

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

Algorithm 1
DEMB Algorithm

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

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

The DEMB scheduling policy is presented in Algorithm 1. When a new query is submitted to the scheduler, it is inserted into the queue and the oldest query in the queue is evicted, which can change the query probability density function. However if we update the boundaries of each server for every incoming query, that may cause too much overhead in the front-end scheduler. Instead, we employ a sliding window approach, where the scheduler waits for a predefined number of queries (update interval, UI) to arrive before updating the boundaries. Note that the update interval does not have to be equal to the window size. As the update interval is increased, the window size also has to be increased. Otherwise some queries may not be counted when calculating the query probability distribution.

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

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

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

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

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

Now we show an example to see how the DEMB scheduling policy works. Suppose there are 10 back-end servers (N = 10), the window size is 500 queries (WS = 500), and the boundary update interval is 100 queries (UI = 100). The front-end scheduler will replace the oldest query in the queue with the newest incoming query every time a new query arrives. The incoming queries will be forwarded to one of the back-end servers using the boundaries of each server (BOUNDARY[i]). When the 100th query arrives, the scheduler recalculates the boundary for each server using the past 500 queries. Since there are 10 back-end servers, each server should process 50 queries to achieve perfect load balance. Hence, the new boundary between the 1st server and the 2nd server should be the middle point of the 50th query and the 51st query in the Hilbert curve ordering. Likewise, the new boundary (CUR_BOUND[i]) between the ith server and the i + 1th server should be the middle point of the 50*ith query and the 50*i + 1th query. After calculating the new boundaries (CUR_BOUND[i]) for the back-end servers, we compute the moving average for each boundary. If the query distribution has changed, BOUNDARY[i] would move to CUR_BOUND[i]. In this way, the DEMB scheduling policy makes the boundaries move according to the new query distribution.

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

IV. Exponential Moving Kernel Density Estimation

4.1. BEMA Scheduling Policy

As an improvement of the DEMA scheduling policy, Balanced Exponential Moving Average (BEMA) scheduling policy is proposed to give weight to each EMA points. Similarly to DEMA, the 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.

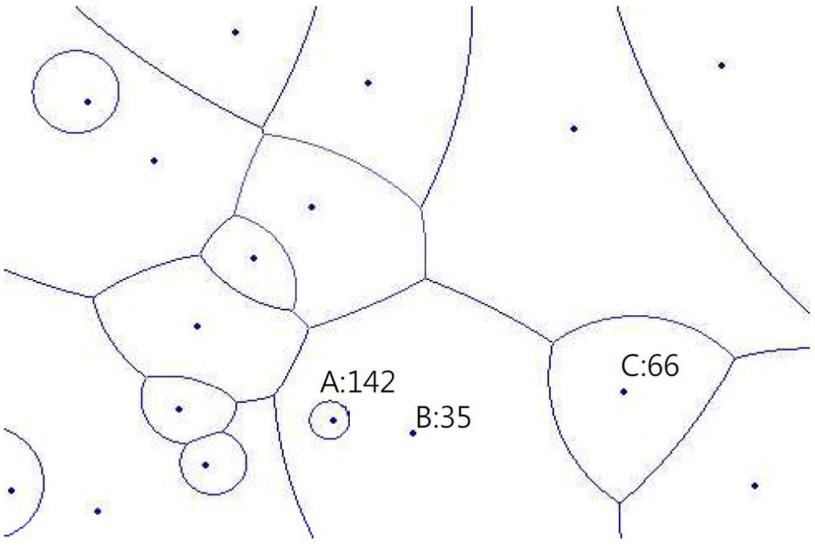


Figure 5. BEMA scheduler calculates the weighted Euclidean distance between EMA points and an incoming query, and assigns the query to the server B whose EMA point is closest.

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 server 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 5, 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.

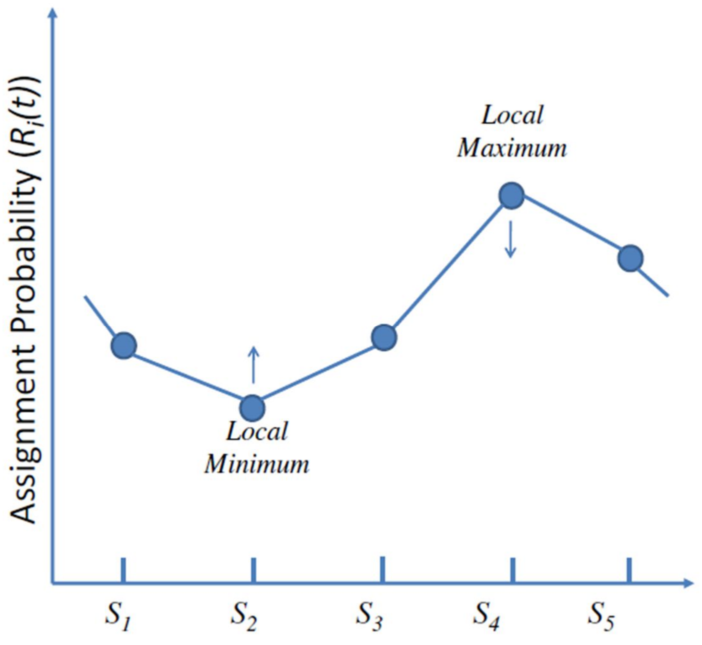


Figure 6. Load Balancing by Convex Optimization

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 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 6 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 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[ ] is the expectation function. With this property, the system is moving toward a better balanced state.

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.

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.

Although DEMB scheduling policy shows good load balance and cache hit ratio, A considerable scheduling overhead can be incurred if window size of the query histogram is set to too large value for some reason. Therefore we need an alternative approach which overcomes limitations of BEMA and DEMB scheduling algorithm.

4.2. Exponential Moving Kernel Density Estimation (EM-KDE)

In this section, we introduce EM-KDE (Exponential Moving with Kernel Density Estimation) scheduling policy. 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, than 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.

4.2.1. 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.

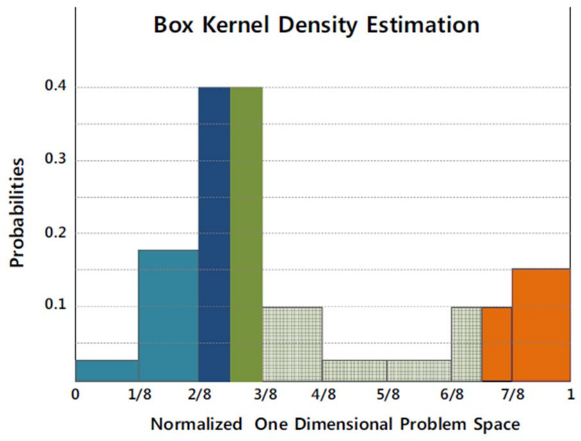
To achieve both load balancing and high cache-hit ratio, the front-end scheduler must estimate the current query distribution accurately. In statistics, Kernel Density Estimation (KDE) is commonly used to estimate the PDF of a random variable based on the finite set of samples. When a sample arrives, 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 lot of 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 literatures, 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.

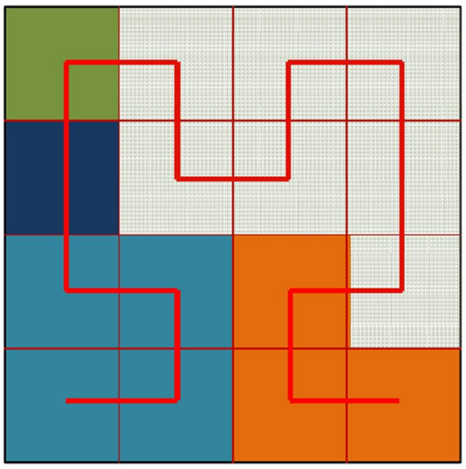
After constructing the smoothed probability density estimation, the EM-KDE scheduling policy chooses the boundaries of problem space for each back-end server so that each has equal cumulative distribution value. Unlike the BEMA scheduling policy, the EM-KDE scheduling policy adjusts the boundaries of 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.

4.2.2. Addressing Dimensionality

Scientific data analysis queries typically specify multi-dimensional geometric or geographic range of coordinates: for example, latitude/longitude and time ranges, or 3D 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. 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.



(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

Figure 7. 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.

Figure 7 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 7(b).

4.2.3. 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 alpha affects the scheduling performance, and its proper value depends on the query dynamics. As alpha increases, the scheduler becomes more sensitive to the recent query distribution and adapts quickly to new query arrival patterns. On the other hand, as alpha decreases, the PDF estimation focuses on the long term trend, and becomes immune to temporary changes in query distribution. Generally, a smaller alpha 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 alpha 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 alpha value should be incremented in order to quickly adapt to the new query distribution.

4.2.4. 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 by a factor of (1 - alpha) and the incoming query adds a weight alpha 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.

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
```

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 sesrver. 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.

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
```

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 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 Evaluation section. 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.

V. Scheduling with Multiple Front-End

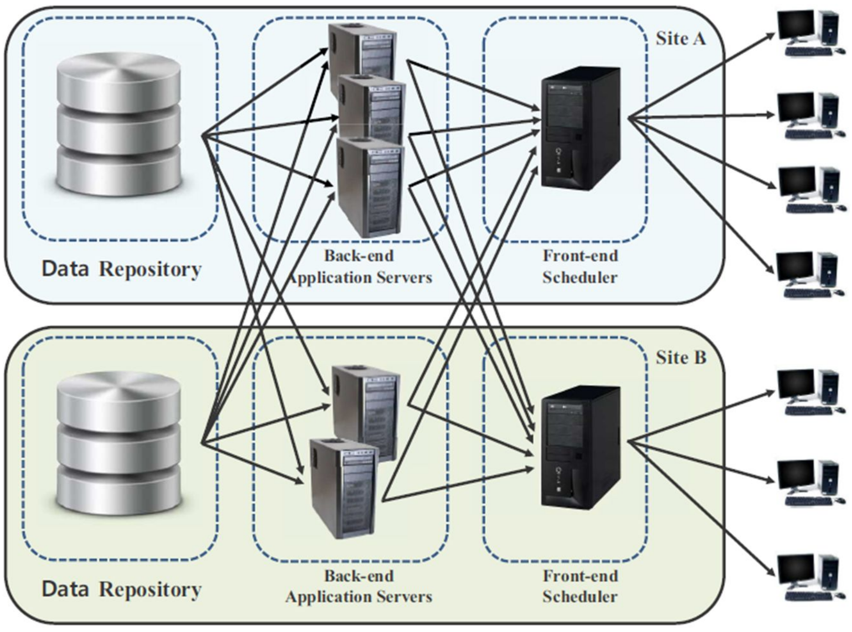


Figure 8. Architecture of Distributed Query Processing Framework with Multiple Front-end Scheduler

5.1. Distributed and parallel query processing framework

Figure 8 is a description of another consideration of our study. 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 8.

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. The scheduler needs to consider all available statistics about the system such as cached data objects, current system load, network latency between organizations, storage location, etc.

As described, DEMA scheduling policy makes scheduling decisions based on geometric features of past jobs. One assumption with DEMA scheduling policy is that one scheduler has seen all the 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 is not true any more. One scheduler may receive very different types of jobs from another scheduler. In such cases, EMA points in each scheduler 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.

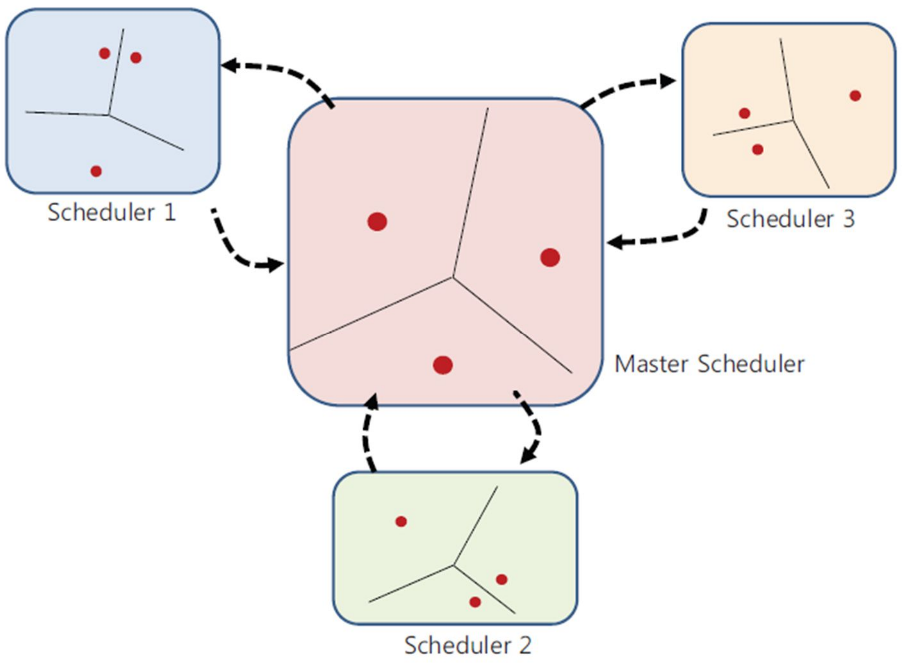


Figure 9. Multiple schedulers share EMA points by synchronizing their own EMA points periodically.

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 9. 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.

5.2. 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 scheduler varies. For example, if a scheduler S1 received very few number of jobs while another scheduler 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:

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

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.

VI. Data Migration of Cached Contents

6.1. Cached data migration

In DEMA scheduling policy, 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.

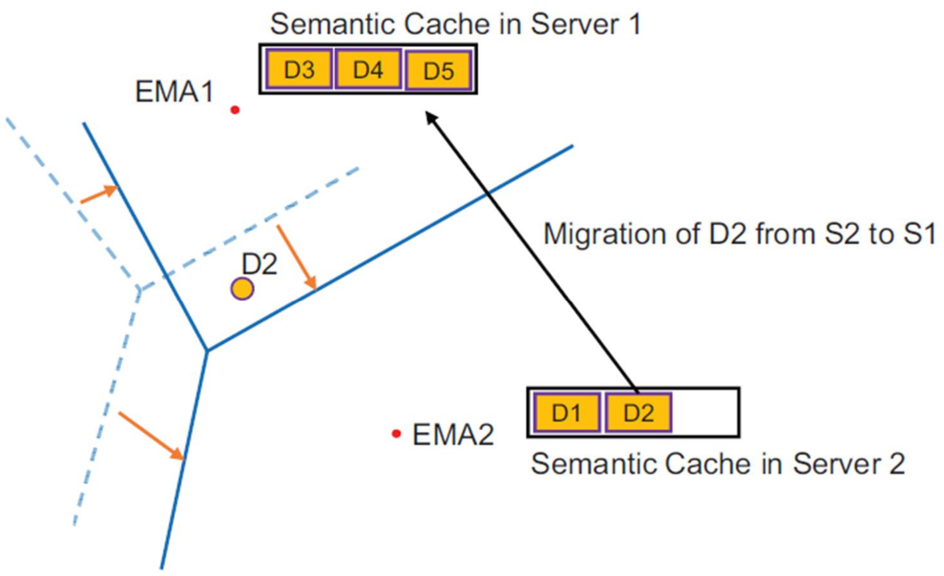

Semantic Cache in Server 1
D3
D4
D5
EMA1
Migration of D2 from S2 to S1
D2
D1
D2
EMA2
Semantic Cache in Server 2

Figure 10. 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.

Figure 10 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 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.

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.

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, 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 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.

VII. Eclipse: The MapReduce implementation

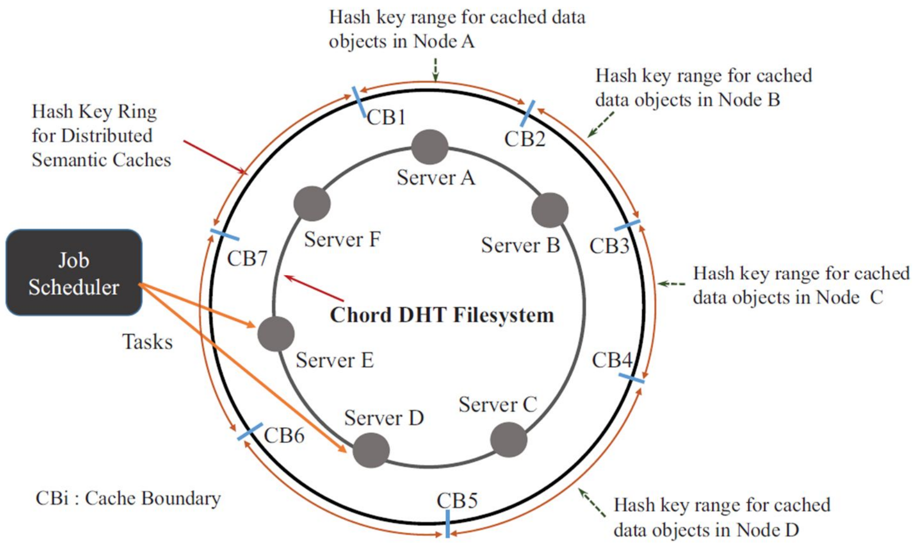

Hash key range for cached data objects in Node A
Hash key range for cached data objects in Node B
Hash Key Ring for Distributed Semantic Caches
CB1
CB2
Server A
Server F
Server B
CB3
CB7
Job Scheduler
Hash key range for cached data objects in Node C
Chord DHT Filesystem
Tasks
Server E
CB4
Server C
Server D
CB6
CBi : Cache Boundary
CB5
Hash key range for cached data objects in Node D

Figure 11. Double-layered data management in Eclipse. The outer layer is the distributed semantic cache layer and the inner layer is the distributed file system layer.

Eclipse is a distributed semantic caching framework that supports distributed and parallel data processing and forms a double-layered ring structure - one for the distributed file system and the other for the distributed semantic cache infrastructure as shown in Figure 11.

7.1. Distributed File System

Eclipse implements a distributed file system on top of Chord distributed hash table(DHT). Input data files are partitioned into fixed-sized data blocks, and each input data block is stored in the Chord DHT file system according to its user-defined hash key. While Eclipse implements the Chord DHT structure in it, Eclipse is not a pure decentralized peer-to-peer system as it has a central file system manager which is responsible for partitioning input data and uploading partitioned input blocks to distributed servers. The file system manager in our current design also maintains a complete routing table to enable a single DHT routing table look-up. However, the central components of Eclipse take minimal responsibility in order to enforce system scalability and fault tolerance of the system.

The DHT routing table of Eclipse in each server can be managed in a decentralized fashion as in original Chord for a large scale cluster. But for a medium-scale cluster(e.g., less than 100 servers) where server failures are not very common and network bandwidth is not contended, managing a complete DHT routing table in each server is a feasible and better option since it provides more reliable performance. The routing table that stores neighbor information including successor and predecessor is updated only when a participating worker server fails or a new worker server joins. Each server exchanges heartbeat messages with direct neighbors to detect system failures, and the central job scheduler and file system manager are notified when a server failure is detected.

If a server fails, the central file system manager reconstructs the lost file blocks. In Eclipse Chord DHT file system, data blocks are replicated over another Chord ring using a different hash function. In order to replicate a data block k times, k different hash functions need to be provided by users. By default, Eclipse does not replicate input data blocks, and it does not allow users to replicate intermediate outputs due to the high overhead of replication. Recent studies of Hadoop cluster in production environment report Hadoop jobs rarely require input data larger than 14 GBytes. For such relatively small input size, the number of cluster servers is likely to be small and a large number of replicas will not provide any benefits but will just result in performance degradation assuming a locality-aware fair job scheduling policy is used, which we will describe later.

7.2. Distributed Semantic Cache

On top of the Chord DHT file system, Eclipse deploys a distributed semantic caching layer, where input data objects are cached in data cache(dCache), and intermediate data objects are cached in intermediate results cache(iCache) respectively. The in-memory caching is recently implemented in the latest HDFS, but the in-memory caching in HDFS caches only input data while our distributed semantic caching layer caches intermediate results as well. Although input data caching works well when popular working sets fit in cluster memory, its performance benefits are limited if the jobs are computation-bounded.

In data-intensive computation environment, it is common that same applications are often submitted with slightly different input data, and it has been shown that exploiting sub-expression commonality across multiple queries significantly reduces the job response time and improves the system throughput.

There exist several prior works that report more than 30 % of MapReduce jobs are repeatedly submitted in production environment. In Eclipse, the intermediate data cache(iCache) enforces avoiding redundant computations by sharing the intermediate results among submitted jobs, thus saves total execution time. In order to reuse intermediate results of previous jobs, Eclipse tags them with semantic metadata - application ID and hash keys of input data blocks and stores them in iCache.

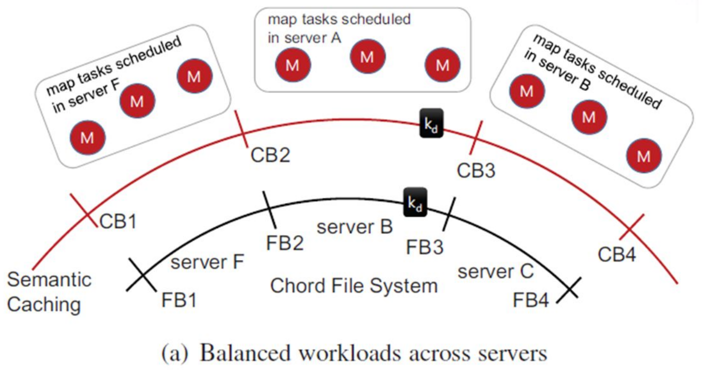
For the incoming jobs, the job scheduler searches for the opportunity of sharing intermediate results by comparing the application ID and the hash keys of input data blocks. If they match and the tagged metadata are verified, the mapper tasks skip computation and reducer tasks can immediately reuse the cached intermediate results.

Unlike the Chord DHT file system, the hash key space of distributed semantic cache layer is managed by a central job scheduler. With a given number of servers, the Eclipse job scheduler partitions the global hash key space of the semantic cache layer and assigns each sub-range to a worker server. When a map task needs to access an input data whose hash key is within the hash key range of a certain worker server, the job scheduler assigns the map task to the worker server.

The hash key ranges of semantic cache layer are dynamically adjusted by the job scheduler to achieve load balance and to increase the probability of reusing cached data objects. Since the hash key ranges of Chord DHT file system are rather static, the hash key ranges of semantic cache layer are not always aligned with the hash key ranges of distributed file system layer.

The central job scheduler receives incoming job requests and constructs a histogram of the hash keys of the recently accessed input data in order to predict the trend of data access patterns, that is, the distribution of hash keys. Based on the distribution of input data accesses, Eclipse job scheduler adjusts the hash key ranges of distributed semantic caches so that equal number of input data blocks are processed across multiple worker servers.

Figure 12 shows an example of how Eclipse job scheduler balances the load by adjusting the hash key ranges of distributed semantic cache layer. Suppose a map task needs to access a data block d whose hash key kd falls within the hash key range of a server s's semantic cache([CBs, CBs+1)), i.e., CBs <=kd < CBs+1. Eclipse job scheduler starts a map task in the server s expecting the server s's dCache contains the required input data block. The server s will first check if its iCache has the intermediate results that can be reused for the map task. If not, it searches the input data block in its dCache. If the dCache does not have d, it reads the data block from the Chord DHT file system, and stores it in its dCache. After processing the user-defined map task, the intermediate results will be cached in its iCache.


map tasks scheduled in server F
map tasks scheduled in server A
map tasks scheduled in server B
M
CB1
CB2
CB3
CB4
FB1
FB2
FB3
FB4
server F
server B
server C
Chord File System
Semantic Caching
(a) Balanced workloads across servers

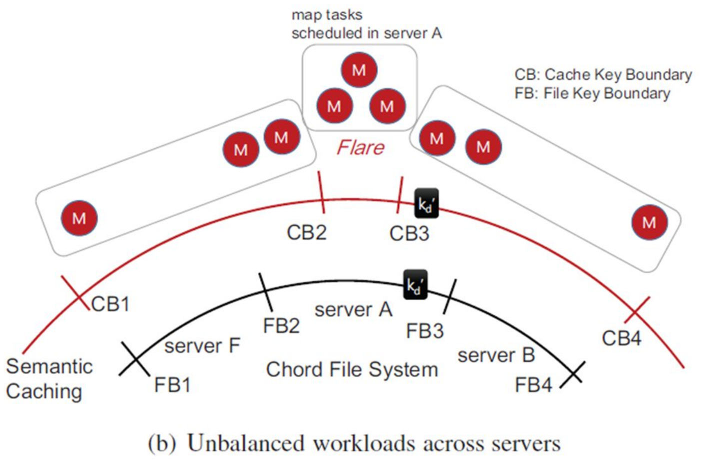
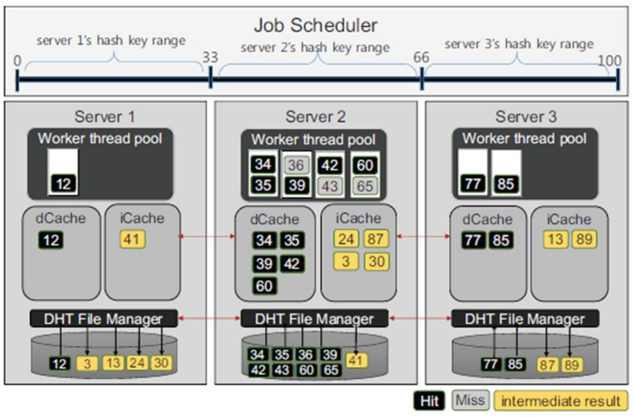


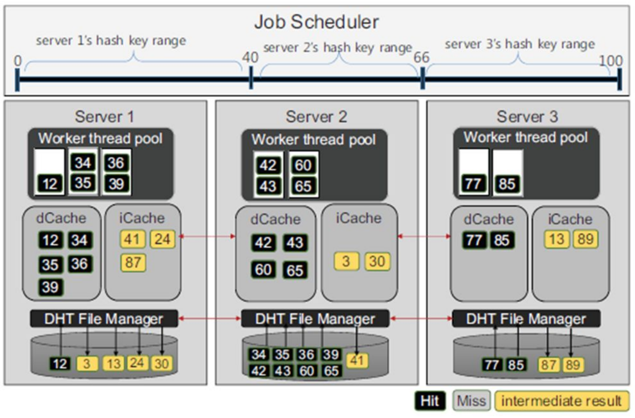
Figure 12. Eclipse achieves load balancing while preserving data locality.

In Figure 12(a), the submitted tasks access uniformly distributed input data blocks, and the job scheduler manages equal-sized hash key ranges, i.e., [CB1, CB2) = [CB2, CB3) = [CB3, CB4).... Thus, Eclipse achieves both good load balance and good data locality. However, if certain input data blocks become popular and load balancing fails as shown in Figure 12(b), the Eclipse job scheduler reduces the hash key range of a busy server([CB2, CB3) in the example) so that some map tasks are scheduled in neighbor servers. In the example shown in Figure 12(b), k'd is stored in server B's local disks but it is cached in server C. We refer to the popular hash key range as flare. In later section, we will describe in more details how Eclipse job scheduler handles the flare problem and achieves both load balancing and data locality. If hash key ranges of semantic cache layer are aligned with the hash key ranges of the Chord DHT file system as shown in Figure 12(a), the cache miss will read input data from local file system. But if they do not match due to flares, cache miss will cause a worker server to look up its DHT routing table and to read input data blocks from remote worker server.

Once the map tasks finish their processing, they will notify the job scheduler that reducers can start their jobs. In addition to storing the intermediate results in iCache, the map tasks store intermediate results in the distributed file systems in parallel. Unlike Hadoop, Eclipse map tasks store the intermediate results in distributed file systems instead of local disk in order to eliminate the shuffling phase as we will describe more in details later.



(a) Unbalanced workloads across servers



(b) Balanced workloads across servers

Figure 13. Functional Components of Eclipse: Load balancing of Eclipse is achieved by adjusting the hash key spaces in job scheduler.

Depending on application types the hash keys of intermediate results can be different from the hash keys of input data. If so, the hash keys of intermediate results can be out of the worker server's hash key range. However, we cache the intermediate results in the iCache of the local worker server as shown in Figure 13. Note that the main purpose of iCache is to avoid redundant processing of the same map task for the same input data, hence the intermediate results better be cached where the map tasks run. If a subsequent map task is going to read input data blocks that were already computed by previous map tasks of the same application, the map task simply reuses the cached results in iCache.

Since the size of iCache is not always large enough to hold the entire intermediate results, the intermediate results must be stored in distributed file system according to their hash keys so that reducers can read all the intermediate results from the distributed file system. The intermediate results stored in distributed file systems are invalidated by time-to-live(TTL) which can be set by applications.

7.3. Job Scheduling Policy in Eclipse

The job scheduler in Eclipse is responsible for allocating worker servers to the incoming MapReduce jobs. When a client submits a job to the job scheduler, it identifies what data files are needed for the job and communicates with a central file system manager to find out where are the partitioned input blocks for the input file. Given the list of input blocks, the job scheduler computes the hash keys of the input blocks. With the hash keys of input data blocks, the job scheduler identifies which worker servers own the hash keys and it creates map tasks in the worker servers accordingly. Based on the number of cores of each worker server, the job scheduler determines the number of map and reduce slots that can run simultaneously. By default, the number of map and reduce slots is set to the number of cores available in the system, but it can be customized by system configuration.

If enough slots are available in worker servers, Eclipse job scheduler runs multiple jobs concurrently as in Hadoop fair scheduler. However, if there are not enough slots available for submitted multiple jobs, Eclipse job scheduler currently uses the first-come, first-served scheduling policy. In order to further improve the system throughput, we plan to implement multiple query optimization scheduling policies that rearrange the execution ordering of jobs to maximize the data reuse ratio. It should be noted that the prior works regarding the multiple query optimization are complementary to our work in such a sense.

In a distributed environment, job scheduler should evenly spread workloads across multiple servers. However, planning load-balanced schedules while maintaining high cache hit ratio is a hard problem because cached data objects are frequently replaced and it is not practical for a job scheduler to keep track of a large number of cached data objects in distributed caches. Therefore, Eclipse resorts to a heuristic algorithm to achieve high cache hit ratio and good load balance.

Eclipse job scheduler collects the statistics about data access patterns of submitted tasks and periodically monitors current load of each server through heartbeat messages. The job scheduler makes scheduling decisions based on current load of the worker server, resource requirements of applications, application types, and most importantly hash key distribution of input data blocks.

```
input: Tasks task
1  hkey = getHashKey(task.data) ;
2  for $server \leftarrow 0$ to $NumServers$ do
3      if $hkey < server.hashKeyUpperBound$ then
4          allocateTaskSlot(task, server) ;
5          break ;
6      end
7  end
8  /* update data access patterns */
9  newHistogram = updateHistogram(newHistogram,
   hkey) ;
10 if $histogram.size == UPDATE\_FREQ$ then
11     for $b \leftarrow 0$ to $NumOfBins$ do
12         histogram[b] = histogram[b]*(1-$\alpha$) +
           $\alpha$*newHistogram[b] ;
13     end
14     pdf = constructPDF(histogram) ;
15     partitioned[] = partitionPDF(pdf,NumServers) ;
16     for $server \leftarrow 0$ to $NumServers$ do
17         server.hashKeyUpperBound =
           partitioned[server] ;
18     end
19     initializeHistogram(newHistogram) ;
20 end
```

Algorithm 4. Eclipse Job Scheduling & Load Balancing Mechanism: The job scheduler dynamically partitions the hash key space so that each hash key range has equal probability distribution value to achieve better load balancing.

To achieve good load balancing while preserving data locality, Eclipse job scheduler estimates the data access distribution using Kernel Density Estimation (KDE) method as shown in Algorithm 4. With accurately estimated data access distribution, the task scheduler can assign even number of tasks to the servers and increase the probability of cached data reuse by assigning similar tasks to the same server. Kernel Density Estimation (KDE) estimates the probability density function of a random variable based on a finite set of samples and we use a box kernel density estimator as the probability density function.

Using the hash keys of input data blocks accessed within a time window, the job scheduler constructs a histogram of the recently accessed hash keys in order to predict the trend of data access patterns. The job scheduler partitions hash key space into a large number of fine-grained histogram bins, and it increases the counter of multiple adjacent k bins for each input data block access by 1/k, where k is a bandwidth parameter. As k becomes larger, the curve of probability distribution function (PDF) becomes smoother.

The data access patterns can be considered as time series data, thus we smooth out short-term fluctuations of the recent data access patterns and attenuate historic data access patterns by convoluting the recent data access patterns and old data access patterns with classic moving average equation. That is, the job scheduler constructs a hash key access PDF for a predefined number of recent tasks, attenuates historic PDF by multiplying a weight factor, and computes the moving average (line 12 in Algorithm 4).

Since each server keeps only certain number of recently cached data using LRU policy, ideally the moving average should reflect the average hash key of currently cached data. Thus, we want to choose the decay factor alpha such that the weight sum of the replaced cached data is below a small threshold value.

With the moving-averaged data access histogram (PDF), Eclipse job scheduler partitions the hash key space of distributed semantic cache layer as shown in Figure 14, so that each hash key range has equal probability distribution value.

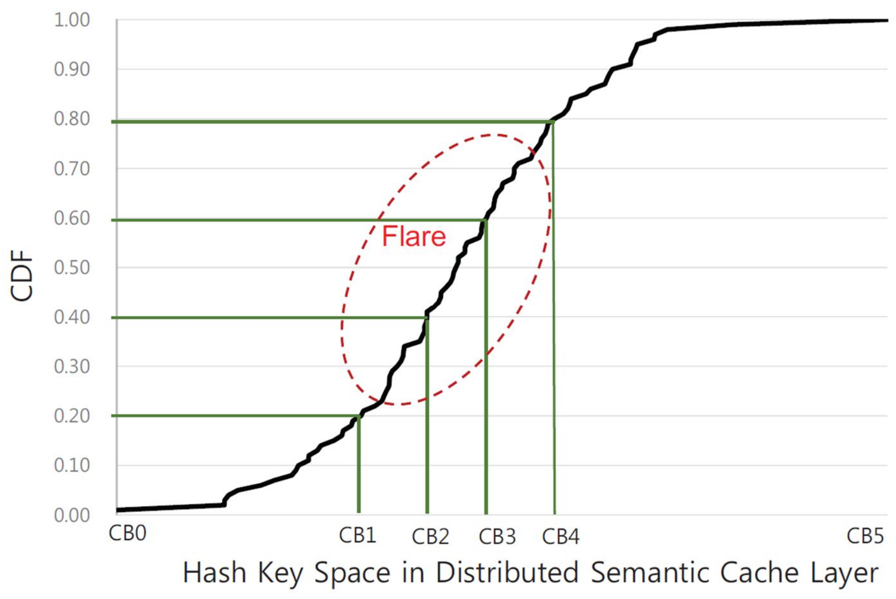


Figure 14. Eclipse job scheduler adjusts hash key ranges of distributed semantic caches and make scheduling decisions based on the ranges.

7.4. Flare: Misaligned Hash Key Ranges

Skew (or flare) is a well-known problem in MapReduce framework that can be caused by an uneven distribution of input data. Several prior works investigated to mitigate the flare problem by implementing speculative scheduling policies, or by repartitioning unprocessed input data and utilizing idle servers in the cluster.

Eclipse addresses the skew problem by dynamically partitioning the hash key space as described earlier, and resolves the uneven input data distribution proactively. If certain tasks take much longer than others even when the input data distribution is well balanced because of some other factors, e.g., some input data processing is more complicated than others, the hash key space ranges should be dynamically adjusted in the middle of job execution and the unprocessed input data should be re-allocated to other idle servers, but we haven't finished implementing this advanced feature that dynamically balances the workload on the fly. In later section, we will show our proactive scheduling based on dynamic hash key ranges improves data processing throughput, but there is still a chance to improve Eclipse regarding dynamic workload reallocation on the fly.

Figure 13 shows an example of task scheduling in Eclipse for a cluster with three servers. Suppose the range of one dimensional hash key space is between 0 and 100, which meets at 0 in a circular way. Initially the job scheduler evenly divides the hash key space ([0, 33), [33, 66), and [66, 100)). The equal-sized partitioning can achieve good load balancing if scheduled map tasks access uniformly distributed input data blocks. However, if more tasks need to access input data blocks in server 2, the Eclipse job scheduler reduces the hash key range of server 2 to achieve better load balance.

One drawback of the dynamic adjustment of hash key ranges is that it results in misaligned hash key space ranges of semantic cache layer with the hash key space ranges of Chord DHT file systems. As shown in Figure 13(b), the server 2's hash key range of Chord DHT file system is [33, 66) while its hash key range of semantic cache is [40, 66). Although the number of tasks running in each server is well balanced, some tasks in server 1 need to access remote data stored in server 2. For example, if another map task that needs to read 37 is scheduled, the job scheduler will assign it to server 1. Unfortunately the dCache in server 1 does not have the input data block 37, thus it sends a read request to its local DHT file manager, which fetches the input data block 37 from server 2. Once the input data block 37 is fetched from server 2, the input data block 37 is cached in server 1's dCache.

If hash key ranges of distributed semantic caches change over time, some hash keys of cached data objects will not be covered by the hash key range of the semantic cache, and they will not be used by subsequent jobs since the job scheduler does not keep track misplacement of individual cached data in hash key space. An easy solution to the misplaced cached data problem is to migrate the cached data to a neighbor worker server in the ring in either pull mode or push mode. For the input data cache miss, Eclipse pulls the input data block from the server that has the input file block in Chord DHT file system. For intermediate results, Eclipse provides an option to check neighbor worker servers and pull the cached results.

The performance effect of the in-memory caching in HDFS is not very different from deploying HDFS on ramdisks, but the distributed semantic caching in Eclipse is more intelligent in a sense that it enables to cache globally popular input data blocks anywhere in the cluster servers in a seamless fashion. As an extreme case, if all the submitted map tasks need input data blocks stored in a specific server's disks, the input data blocks will be distributed and replicated throughout the distributed semantic cache layer. Thus load balancing in terms of the number of running tasks across servers is still guaranteed and subsequent tasks will take the advantage of cache hits.

Although the number of tasks is well balanced, still the specific file server that contains popular input data blocks may suffer from high I/O traffic. However, once the popular input data blocks are replicated in distributed caches, subsequent tasks will read the input data blocks from local caches and I/O traffic will be evenly distributed in the long run.

7.5. Reducer: Pro-active Shuffling

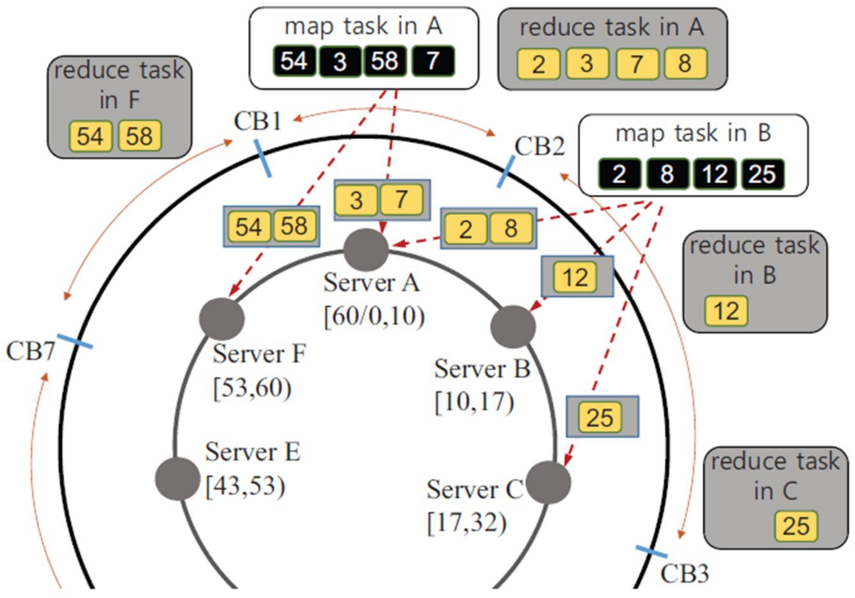


Figure 15. Sorting example in Eclipse: Map tasks read input data from dCache or DFS and writes its intermediate outputs to buffer cache. When the buffer cache fills up, it spills the outputs to DFS (shown as a dotted line). Reduce tasks read the intermediate results from iCache or DFS. If the hash key boundaries of distributed semantic cache layer and Chord DHT file systems are aligned, most disk I/Os occur in local host.

When all the map tasks are completed, the job scheduler starts a set of reduce tasks. By default, each reduce task in Eclipse is responsible for intermediate results locally stored in Chord DHT file system. I.e., if hash keys of intermediate results are within the hash key range of server s's Chord DHT file system, the server s creates a reduce task and reads the intermediate results from its own local DHT file manager. For example, if two map tasks generate intermediate results and distribute them across server A, B, C and F as shown in Figure 15, four reduce tasks will collect the intermediate results and generate final outputs. Eclipse allows users to control the number of reduce tasks. If a user wants to run a smaller number of reduce tasks than the number of servers that have the intermediate results in its local disks, some reduce tasks need to read the intermediate results from remote server's DHT file manager.

Similar to map task scheduling, the reduce tasks are scheduled according to the hash keys of intermediate results and the static hash key range of Chord DHT file system. Since the hash key range of Chord DHT file system is static unlike the dynamic hash key range of distributed semantic caches, reduce tasks may suffer from load imbalance problem. However as we will show in later section, load imbalance problem is more likely to occur in map tasks rather than reduce tasks since map tasks can be accelerated by reusing cached results.

However in Hadoop, the intermediate results are stored in local disks of a server where map tasks run. When map tasks complete, the shuffle phase in Hadoop sort the intermediate results generated by multiple mappers, split them, and send them to reducers. Hence, shuffling phase in Hadoop starts only after all the map tasks are completed. The shuffle phase is known to be very expensive and it significantly hurts the performance, and several prior works have been proposed to decouple the shuffling phase from reduce tasks.

In order to avoid the expensive shuffling phase in Eclipse, we let each mapper pipelines the intermediate results to distributed file system while they are being generated. Based on the hash keys of intermediate results, each map task stores the intermediate results in 64 MBytes memory buffer for each hash key range. When this buffer fills up or when the map task is completed, the buffered results are transmitted to DHT file manager so that they can be stored in Chord DHT file system.

VIII. Evaluations

8.1. DEMB Scheduling Policy

8.1.1. Experimental Setup

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

8.1.2. Experimental Results

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

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

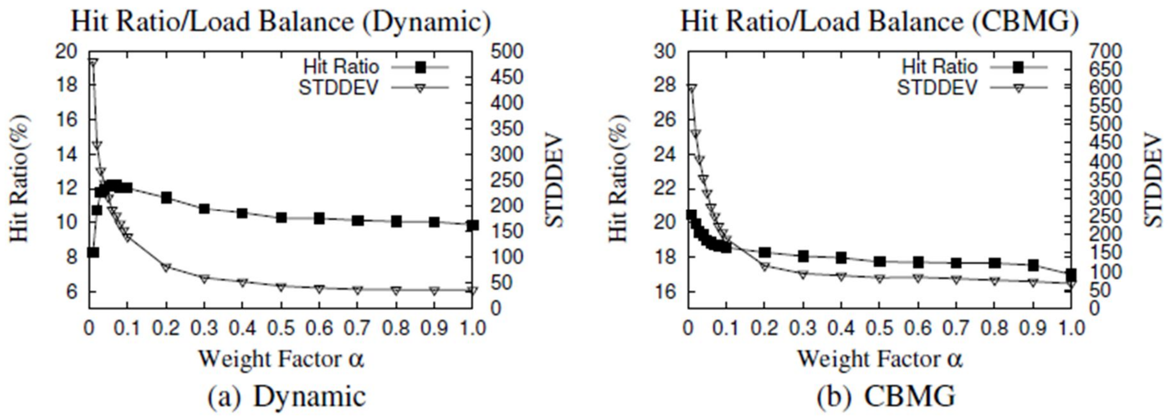


Figure 16. Cache Hit Rate and Load Balancing with Varying Weight Factor

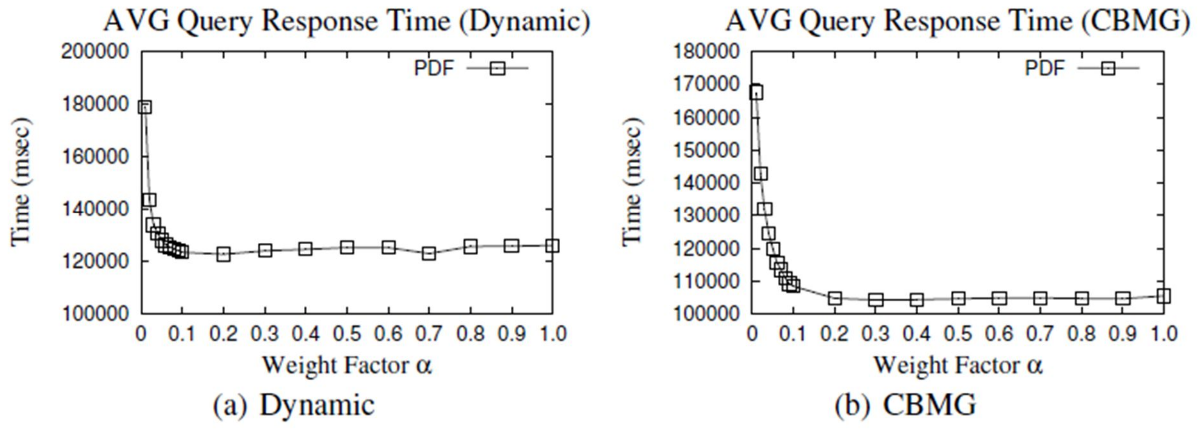


Figure 17. Query Response Time with Varying Weight Factor

For the experiments shown in Figure 16 and 17, we employed 50 servers, and fixed the window size to a small number (200), i.e. only 4 recent queries are used to calculate the boundaries of each back-end server. Since the window size is small, we updated the boundaries of each server whenever 10 new queries arrive. These numbers are chosen to minimize the effects of the weight factor alpha on query response time.

With smaller alpha, the boundaries move more slowly. If alpha is 0, the boundaries will not move at all. For both the dynamic query distribution shown in Figure 16(a) and the CBMG distribution shown in Figure 16(b), load balancing (STDDEV) becomes worse as we decrease alpha because the boundaries of the back-end application servers fail to adjust to the changes in the query distribution.

However the cache hit rate slightly increases from 16.9 % to 20.4 % as we decrease alpha for the CBMG query distribution because the CBMG query pattern is rather stationary and a small alpha value, close to 0, makes the boundaries fixed, which increases the probability of reusing cached objects. However for the dynamic query distribution stationary boundaries decrease the probability of cache reuse, hence the cache hit rate decreases from 12.2 % to 8.1 % as we decrease alpha. Figure 17 shows the average query response time determined by the cache hit rate and load balancing. As we decrease alpha, the query response time increases exponentially since it hurts overall system load balance greatly although it improves the cache hit rate slightly.

Both leveraging cached results and achieving good load balance are equally important in maximizing overall system throughput. However we note that a very small improvement in cache hit rate might be more effective in improving average query response time than a large standard deviation improvement in certain cases. In Figure 16(a), the load balancing standard deviation is approximately 3 times higher when the weight factor is 0.1 than it is 0.3. But the average query response times shown in Figure 17(a) are similar because the cache hit rate is slightly higher when the weight factor is 0.1.

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

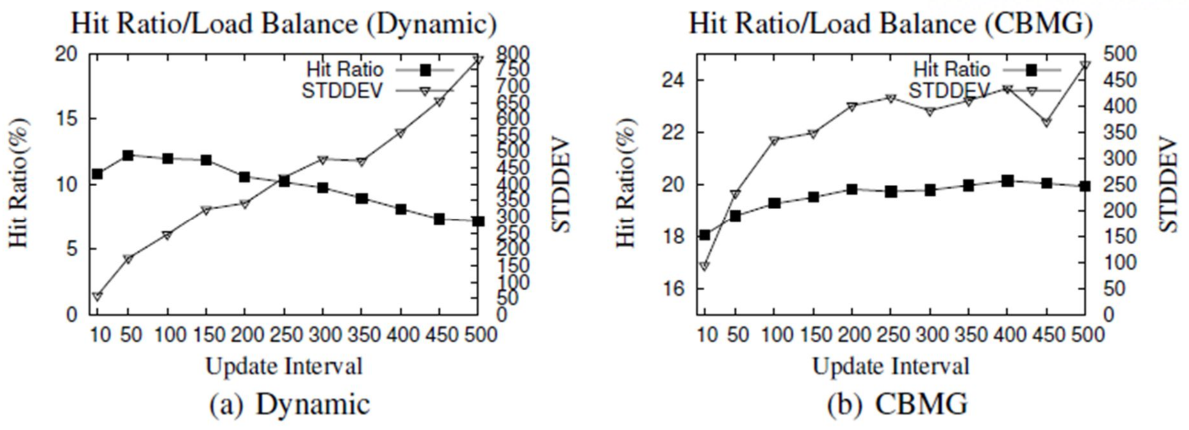


Figure 18. Cache Hit Rate and Load Balancing with Varying Update Interval

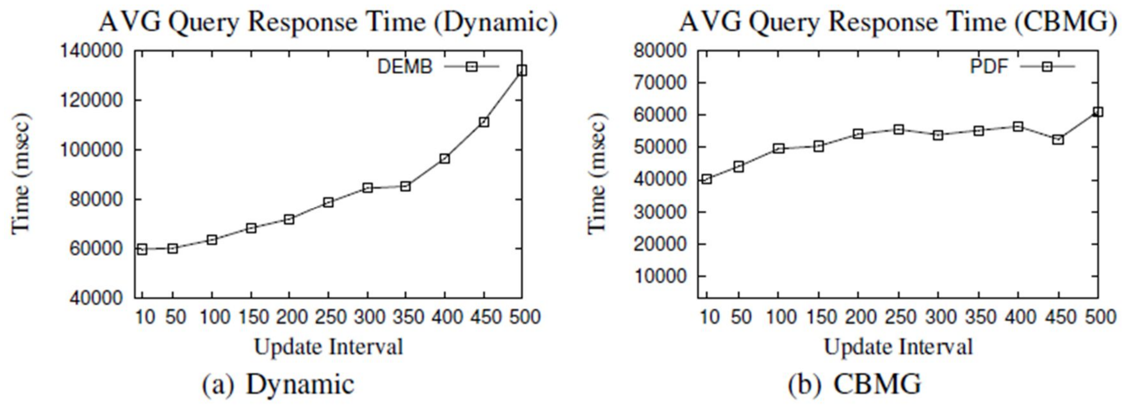


Figure 19. Query Response Time with Varying Update Interval

In the experiments shown in Figure 18 and 19, we measured the performance of the DEMB scheduling policy varying the update interval. In this set of experiments, the alpha and WS were fixed to 0.3 and 200, respectively. As we decrease the update interval, the boundaries are updated more frequently and they reflect the recent query distribution well. As a result, the DEMB scheduling policy takes good advantage of clustering and load balancing for the dynamic query distribution, as shown in Figure 18(a). As the update interval increases, the boundaries move more slowly and DEMB suffers from poor load balancing and cache misses.

With the stationary CBMG queries shown in Figure 18(b), the cache hit rate does not seem to be affected by the update interval, but the standard deviation increases slightly, although not as much as for the dynamic query distribution. These results indicate that we should update the boundaries frequently as long as that does not cause significant overhead in the scheduler.

Window Size: The front-end scheduler needs to store the recent queries in a queue so that they can be used to construct query distribution and determine the boundaries of back-end servers. With a larger window size (more queries), the front-end scheduler can estimate query distribution more accurately. Also, a larger WS allows a query to stay in the queue longer, i.e. the same queries will be used more often to construct query histograms and boundaries. On the other hand, a small WS makes the front-end scheduler uses a smaller number of recent queries to determine the boundaries, and the query distribution estimate is more likely to have large errors. In the experiments shown in Figures 20 and 21, we measured performance with various window sizes. The weight factor alpha and the update interval UI were both fixed to 1, in order to analyze the effects of only WS.

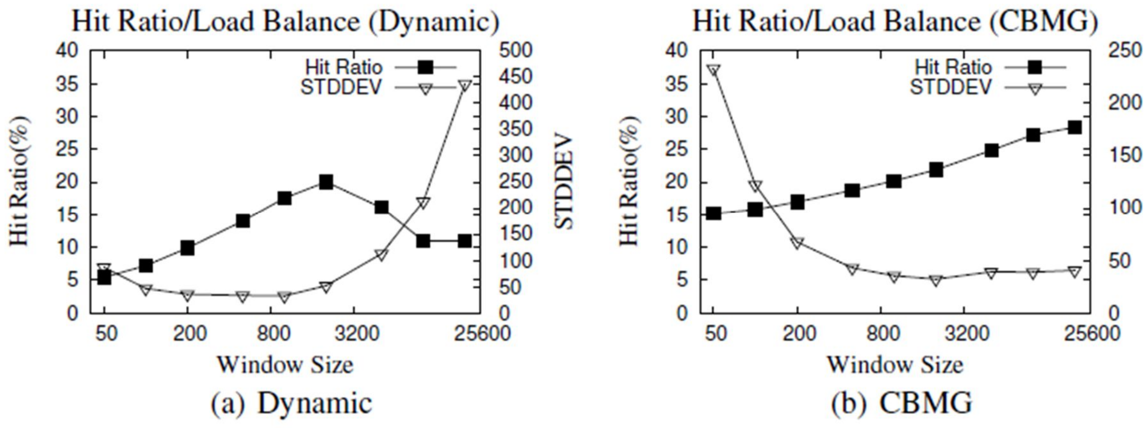


Figure 20. Cache Hit Rate and Load Balancing with Varying Window Size

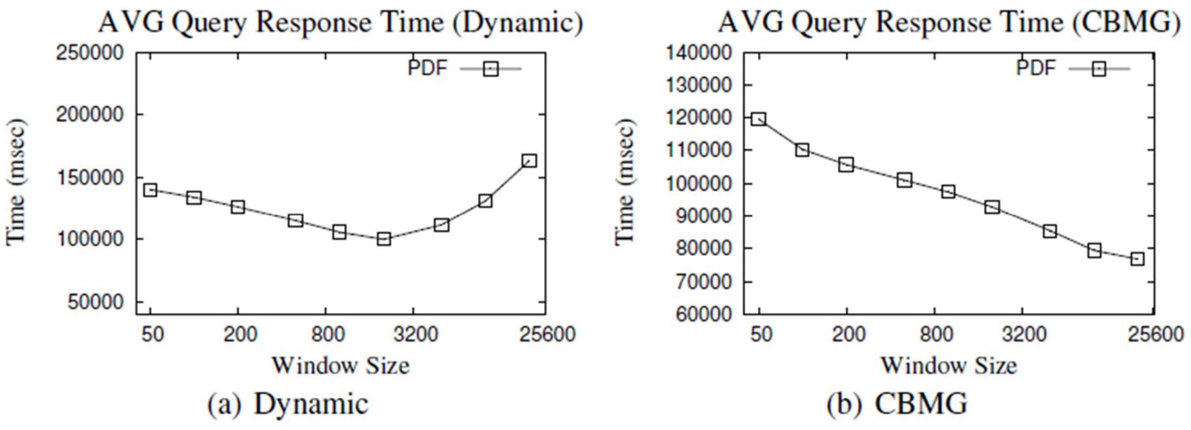


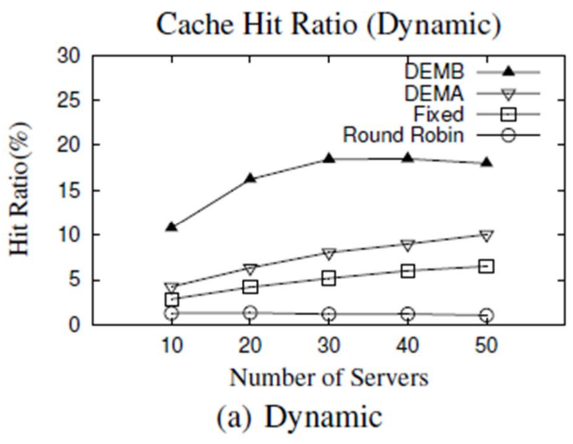
Figure 21. Query Response Time with Varying Window Size

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

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

8.1.3. Comparative Study

In order to show that the DEMB scheduling policy performs well compared to other scheduling policies, we compared it with three other scheduling policies - round-robin, Fixed, and DEMA. For this set of experiments, we employed 50 back-end application servers and a single front-end server. In order to measure how the other scheduling policies behave under different conditions, we used both the dynamic and CBMG query distributions. The parameters for the DEMB scheduling were set to good values from the previous experiments - the weight factor alpha is 0.2, the update interval is 500, and the window size is 1000. These numbers are not the best parameter values we obtained from the previous experiments, but we will show that the DEMB scheduling policy shows good performance even without the optimal parameter values. Figures 22, 23 and 24 show the cache hit rate, load balance, and query response time for the different scheduling policies.



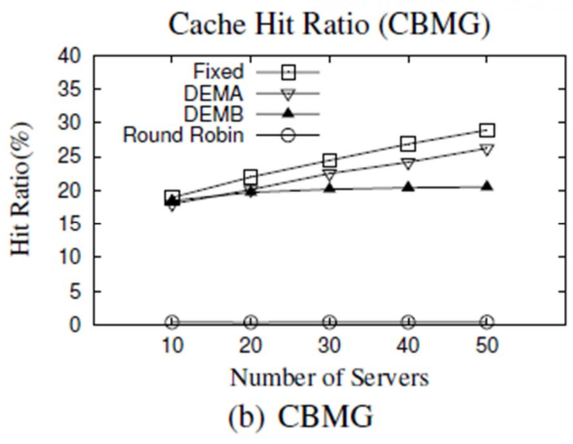
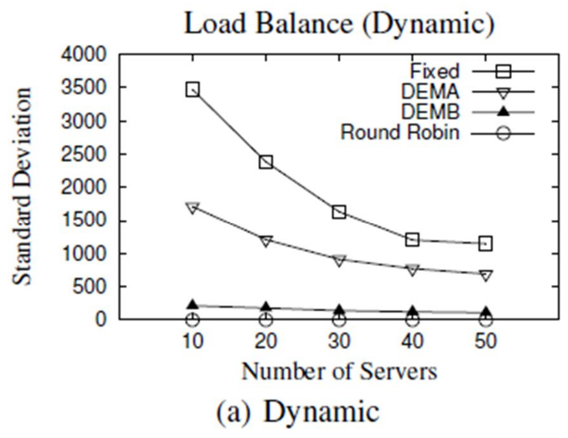


Figure 22. Cache Hit Rate



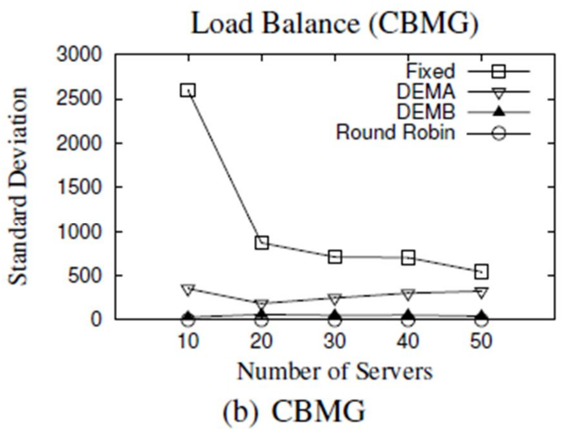
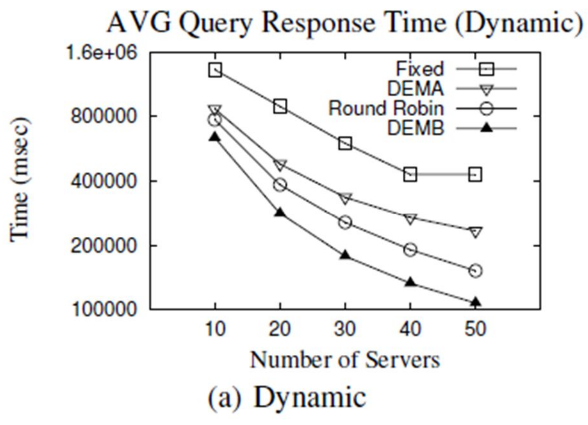


Figure 23. Load Balance



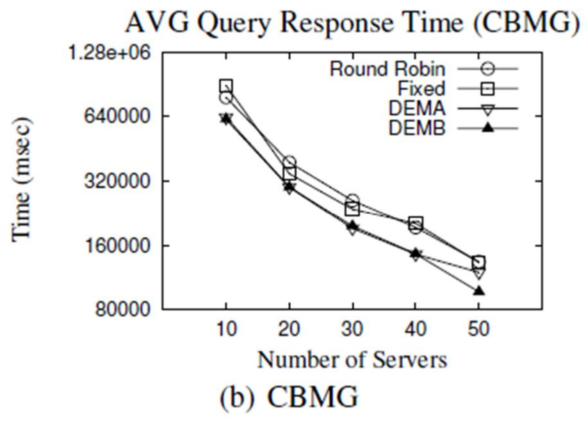


Figure 24. Query Response Time

The Fixed scheduling policy partitions the problem space into several sub-spaces using the query probability distribution of the initial N queries, similar to the DEMB scheduling policy(where N is the number of servers), and each server processes subsequent queries that lie in its sub-space. But the sub-spaces are not adjusted as queries are processed, unlike DEMB. When the query distribution is stable, Fixed scheduling has a higher cache hit rate than round-robin since it takes advantage of spatial locality in the queries while round-robin does not. Since the Fixed scheduling policy does not change the boundaries of sub-spaces once they are initialized, it obtains a higher cache hit rate than the DEMA or DEMB scheduling policies for certain experimental parameter settings, as shown in Figure 22(b). For the CBMG query distribution, there are 200 fixed hot spots and the servers that own those hot spots and have large cache spaces obtain very high cache hit rates. However, when the query distribution changes dynamically the cache hit rate for the Fixed scheduling policy drops significantly, and is much lower than that of the DEMA and DEMB scheduling policy.

Although the Fixed scheduling policy outperforms DEMA and DEMB as measured by cache hit rate when the query distribution is stable, the Fixed policy suffers from serious load imbalance. Some servers process many fewer queries than others, if their sub-spaces do not contain hot spots. Therefore the standard deviation of the Fixed scheduling is the worst of all the scheduling policies, as shown in Figure 23, and the load imbalance problem becomes more especially bad for small numbers of servers. Due to its poor load balancing, the average query response time for the Fixed scheduling is higher than for the other scheduling policies, as shown in Figure 24. Note that the query response time is shown on a log scale.

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

However, for the CBMG query distribution, the quick response to the changed query distribution seems to lower the cache hit rate of DEMB somewhat. As we mentioned earlier, the DEMB scheduling policy is designed to overcome the drawback of the DEMA scheduling policy, which is seen most when the query distribution changes dynamically. In CBMG distribution the query distribution pattern is stable, and both the DEMA and the DEMB scheduling policies perform equally well. In Figure 22(b), the cache hit rate for DEMB is lower than for the DEMA scheduling policy because the boundaries determined by DEMB are likely to move rapidly, since some spatial locality for the queries is lost.   When the boundaries are adjusted rapidly based on short term changes from a small number of recent queries, that may improve load balance but it will decrease the cache hit rate when the long term query distribution is stable. Due to the DEMB scheduling policy's fast response time, its load balancing performance is similar to that of the round-robin scheduling policy. The DEMA scheduling policy also balances server load reasonably well for the CBMG query distribution, but for the dynamic query distribution DEMA suffers from load imbalance due to its slow adjustment of EMA points.

Figure 24 shows that the average query response time improves as we add more servers to the system. The DEMB scheduling policy outperforms all the other scheduling policies for the dynamic query distribution. For the CBMG query distribution, the DEMB query response time is the lowest in most cases. For 50 servers, DEMB shows the best performance although its cache hit rate is not the highest. This result shows that load balancing plays an important role in large scale systems. However load balancing itself is not the only factor in overall system performance because round-robin does not show good performance.

8.1.4. Automated Parameter (WS) Adjustment

In the DEMB scheduling policy, the three parameters (weight factor, update interval, and window size) determine how fast the boundaries of each server adjust to the new query distribution. As seen in Figure 19, the update interval (UI) should be set close to 1 to be more responsive, i.e. the boundaries of servers must be updated for every incoming query. The weight factor (alpha) also determines how much weight is given to recent queries so that the boundaries adjust to the new distribution. However, when the window size (WS) is large enough, the current query distribution already captures recent changes in query distribution and the weight factor (alpha) does not affect query response time significantly. If the window size is not much greater than the number of backend servers (within a factor of 2 or 3), the query response time is not as resilient to changes in the weight factor when alpha is larger than 0.1, as shown in Figure 17. This is because the large window size (WS) and the small update interval (UI) makes a single query counted for multiple (WS) times (such as in a sliding window) to determine the boundaries for each server. Hence, the most effective performance parameter that system administrators can tune for the DEMB scheduling policy is the window size (WS).

As shown in Figure 21, a larger window size yields lower query response times when the query distribution is stable. But if the query distribution changes rapidly, the window size must be chosen carefully to reduce the query response time. In order to automate selecting a good window size based on changes in the query distribution, we make the scheduler compare the current query distribution histogram with a moving average of the past query distribution histogram using Kullback-Leibler divergence, which is a measure of the difference between two probability distributions. If the two distributions differ significantly the scheduler decreases the window size so that updates to boundaries happen more quickly. When the two distributions are similar, the scheduler gradually makes the window size bigger (but no bigger than a given maximum size), which makes the boundaries more stable over time to achieve a higher cache hit rate. In experiments not shown due to page limitation, we have observed that this automated approach improves query response time significantly.

8.2. EM-KDE Scheduling Policy

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.

The performance improvement from the proposed scheduling policies is 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 msec.

To measure the performance of the scheduling policies, we use Dynamic dataset and CBMG dataset which is used to evaluate the performance of DEMB. Same as above, 40,000 2-dimensional queries are synthesized with various distribution models.

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.

8.2.1. Weight Factor

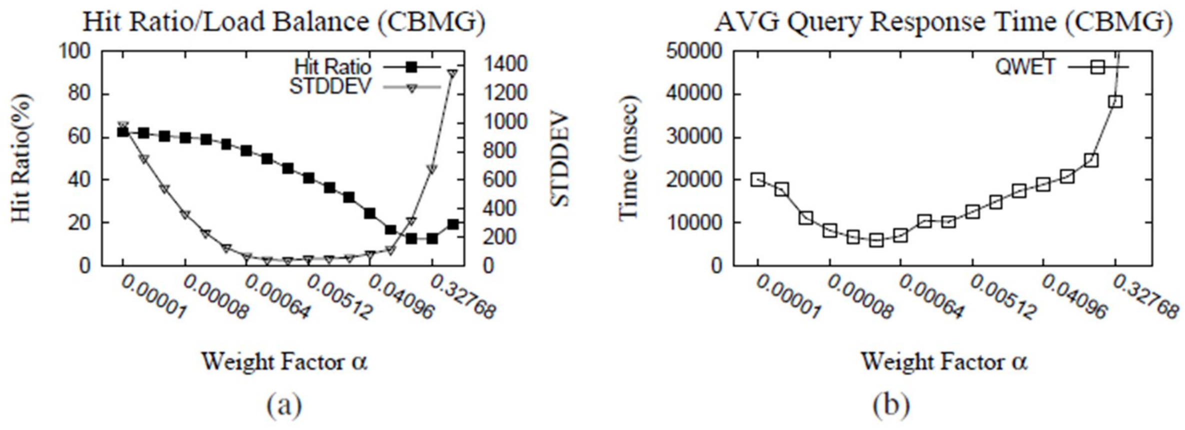


Figure 25. Performance of EM-KDE with Varying Weight Factor (Realistic Workload)

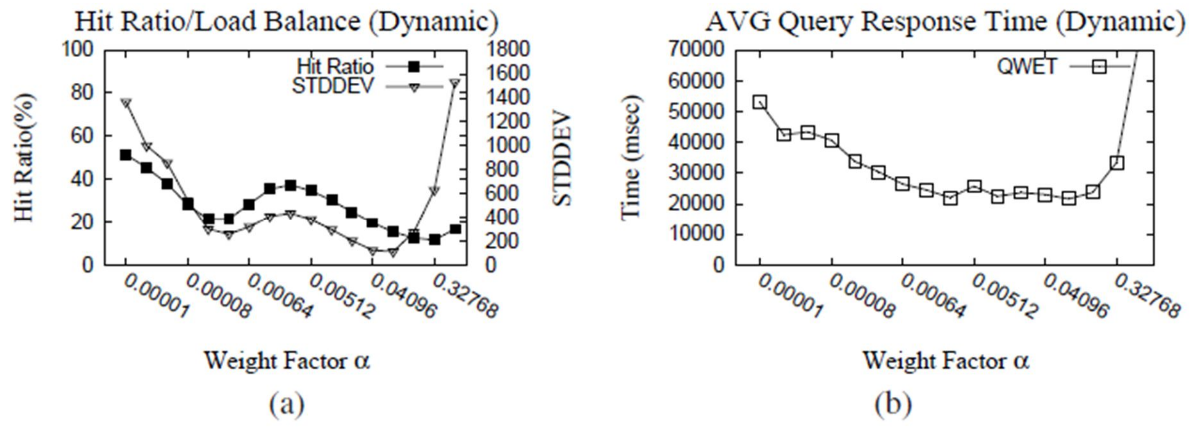


Figure 26. Performance of EM-KDE with Varying Weight Factor (Dynamic Workload)

In the first set of experiments shown in Figure 25 and 26, 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 alpha from 0.00001 to 0.32768. The weight factor alpha 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 alpha, the boundaries will move faster since it gives more weight to recent queries. With smaller alpha, the boundaries move more slowly. If alpha is 0, the boundaries will not move at all. The optimal alpha 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 (STDDEV) becomes worse as alpha 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 alpha is higher than 0.04.

For the realistic workload (CBMG) shown in Figure 25, the cache hit ratio increases as alpha decreases. This is because the CBMG query pattern is rather stationary and a small alpha value, close to 0, makes the boundaries fixed, which increases the probability of reusing cached objects. However, when the alpha is too small, the boundaries move slowly and they hurt the load balancing. On the contrary, when the alpha becomes large, the load balancing also suffers from frequent boundary fluctuation. The Figure 25(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 26, the results are similar except that the cache hit ratio and load balancing are good in two different alpha values. This happens because we synthetically merged four different query workloads that have different query arrival patterns as in the evaluation of DEMB scheduling policy. The best alpha values for the query workloads are all different and the cache hit ratio and the standard deviation graphs are convoluted. In Figure 26(b), the range of alpha values for good query response time is wider than the realistic workload, which means the dynamic query workload is less sensitive to alpha than stationary query workloads. Also, note that with the dynamic query workload the query arrival pattern suddenly changes at certain intervals, hence the higher alpha 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.

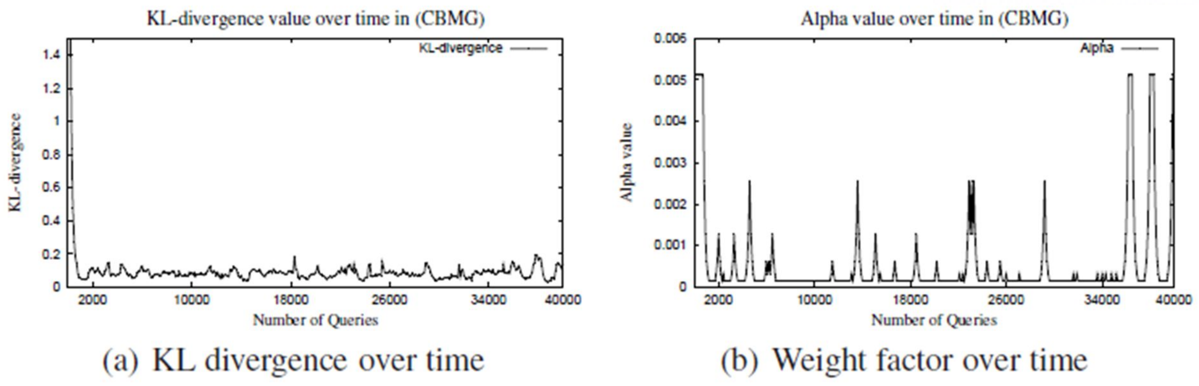


Figure 27. Weight factor automation (Realistic Workload)

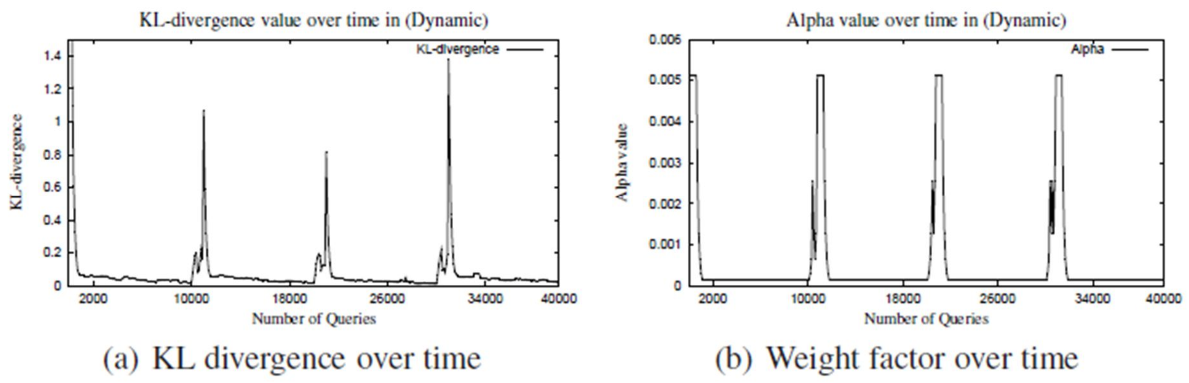


Figure 28. Weight factor automation (Dynamic Workload)

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

Figure 27 and 28 show how the Kullback-Leibler divergence and the selected weight factor value change for the two query workloads. In the synthetically generate 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 26, and 684 msec for the realistic query workload, which is also lower than any static weight factor value shown in Figure 25(b).

8.2.2. Number of Histogram Bins

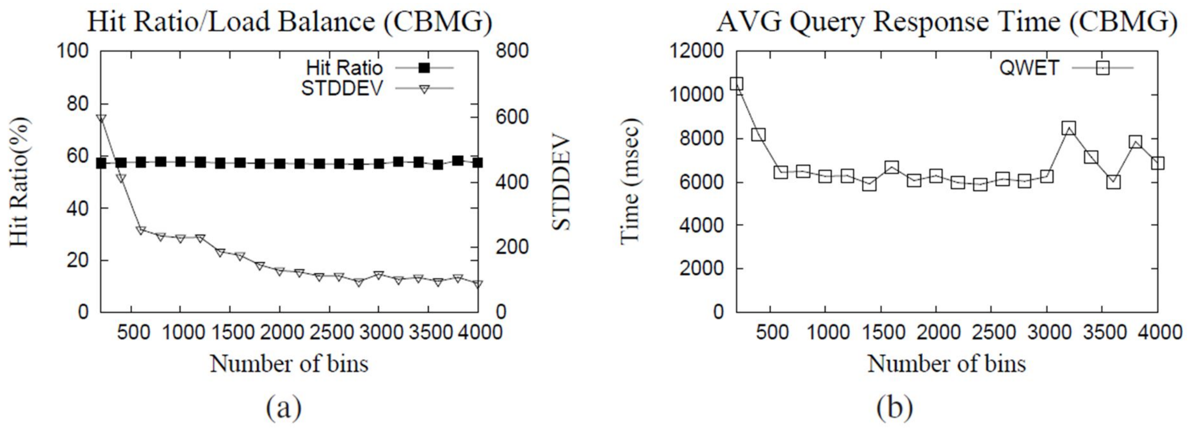


Figure 29. Realistic Workload: Number of Bins

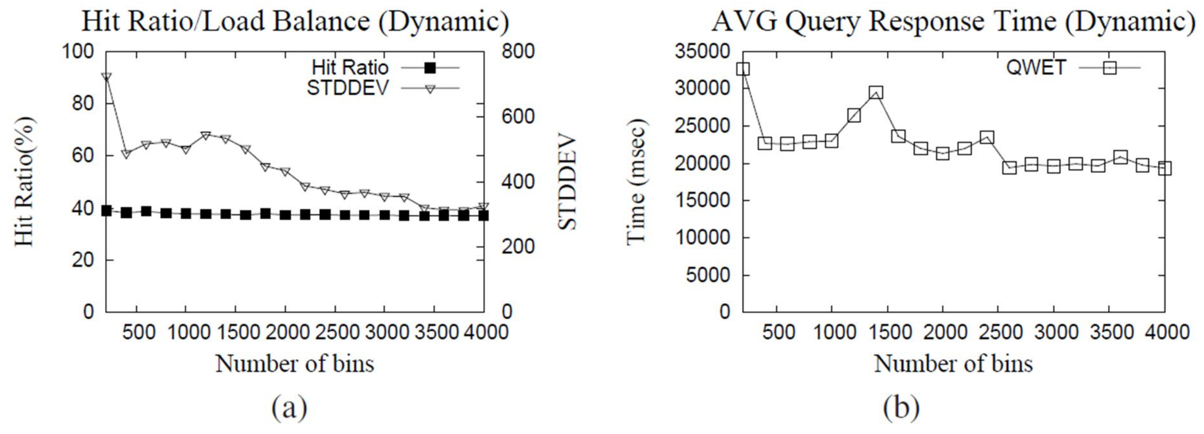


Figure 30. Dynamic Workload: Number of Bins

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 29 and 30 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 29, 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 30, 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.

8.2.3. Scheduling Throughput with a Batch of Queries

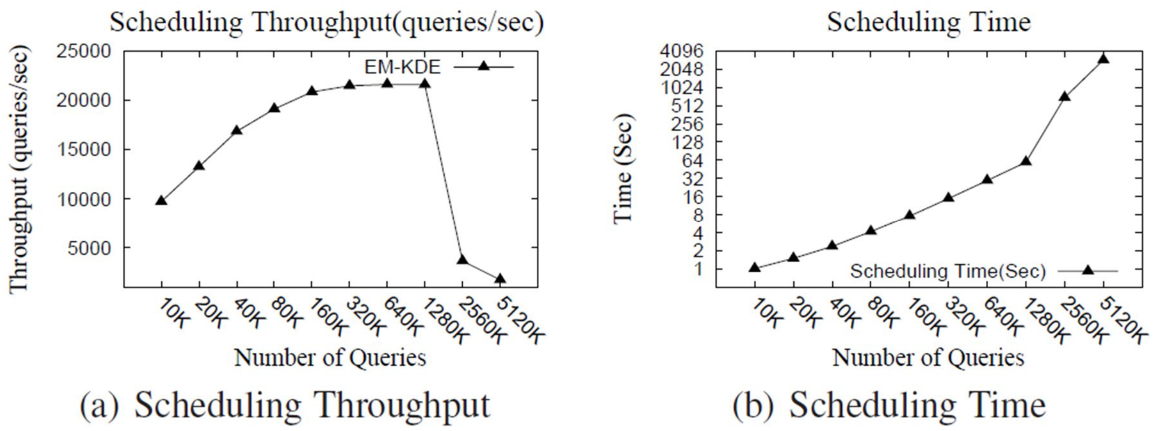


Figure 31. Scheduling Throughput of a Single Front-end Scheduler

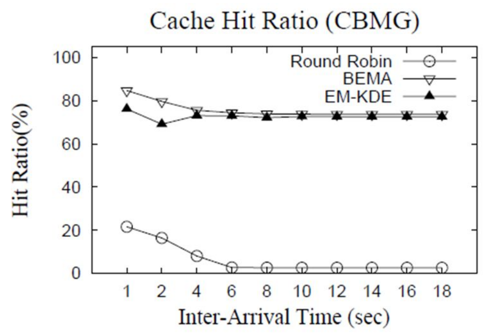
In order to evaluate the scheduling throughput of our presented distributed query processing framework, we deployed 312 back-end application servers with a single front-end scheduler in our testbed. In order to measure the scheduling throughput of EM-KDE scheduler, we submit a batch of very lightweight 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 million. 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 31(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 million. However when more than 2 million 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 31(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 million of queries are submitted, thrashing occurs and the job execution time sharply increases.

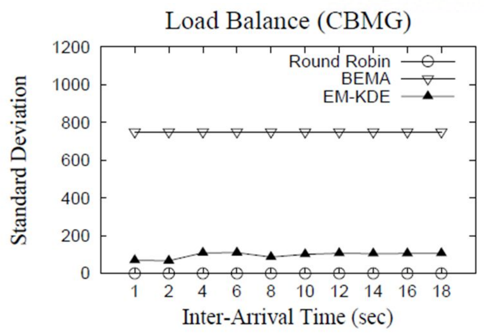
We believe it is not very common to process more than 20,000 queries in a second in large scale scientific data-intensive applications. However, if a more scalable scheduler is needed for a certain type of applications, we may employ multiple front-end schedulers to distribute user queries. With multiple 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.

8.2.4. Inter-Arrival Time

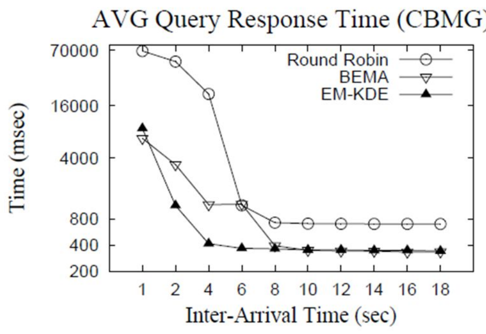
To evaluate the performance of the EM-KDE scheduling policy, we compare the cache-hit ratio and load balance of EM-KDE method with round-robin and BEMA scheduling policies for both realistic and dynamic workloads.



(a) Hit Ratio

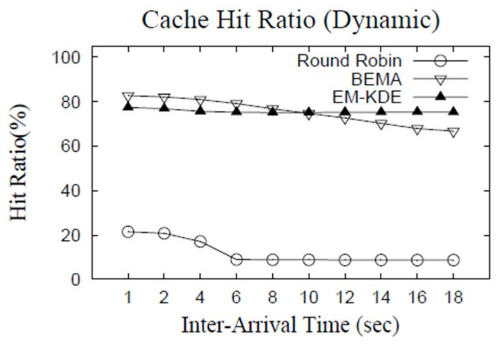


(b) Load Balance

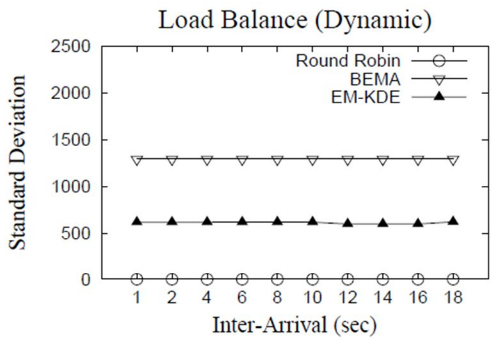


(c) Query Execution Time

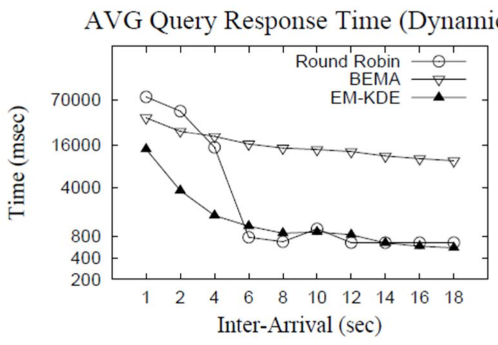
Figure 32. Performance Comparison with Varying Inter-arrival Time (Realistic Workload)



(a) Hit Ratio



(b) Load Balance



(c) Query Execution Time

Figure 33. Performance Comparison with Varying Inter-arrival Time (Dynamic Workload)

Figure 32 and 33 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 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 33, 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.

8.2.5. Scalability

In the experiments shown in Figure 34, 35 and 36, 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 alpha value of 0.03 which shows its best performance while the EM-KDE scheduling automatically adjusts its weight factor alpha for realistic and dynamic workload.

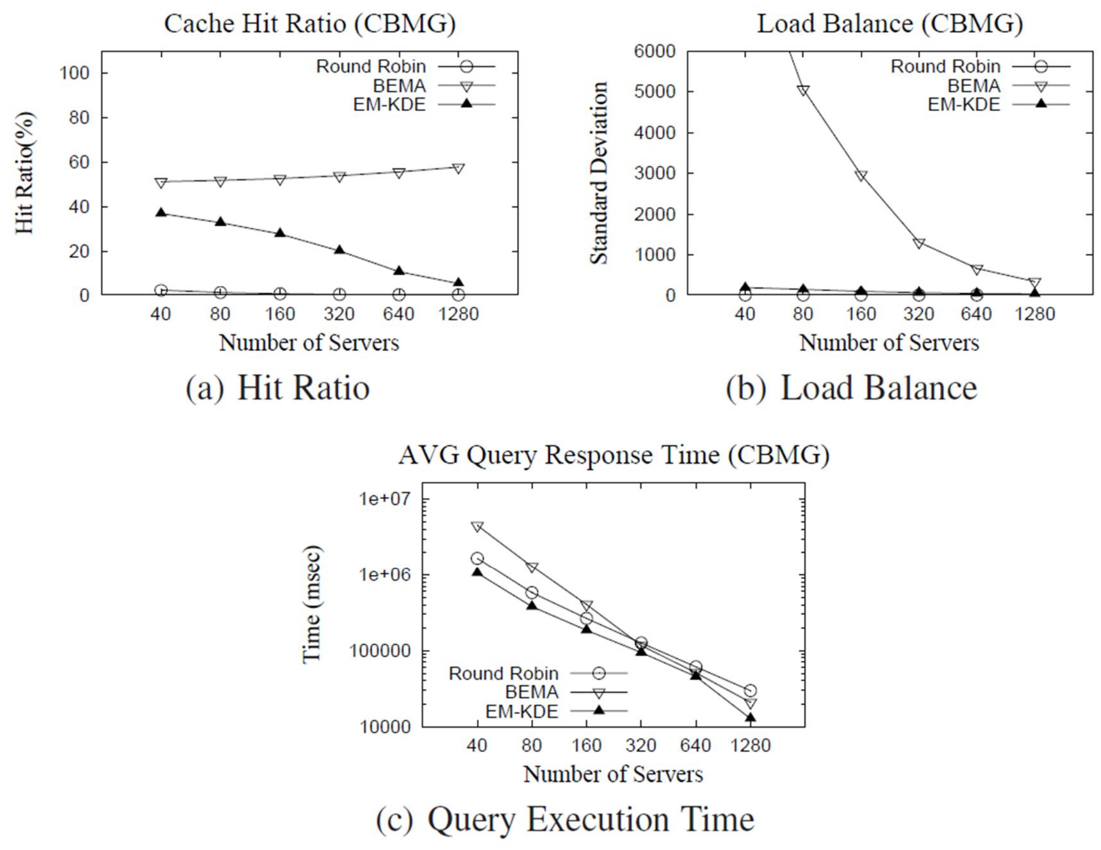


Figure 34. Performance Comparison with a Large Number of Servers (Realistic Workload)

For the experiments shown in Figure 34, 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.

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.

Throughput 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 34(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.

For realistic workloads, as shown in Figure 35, 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.

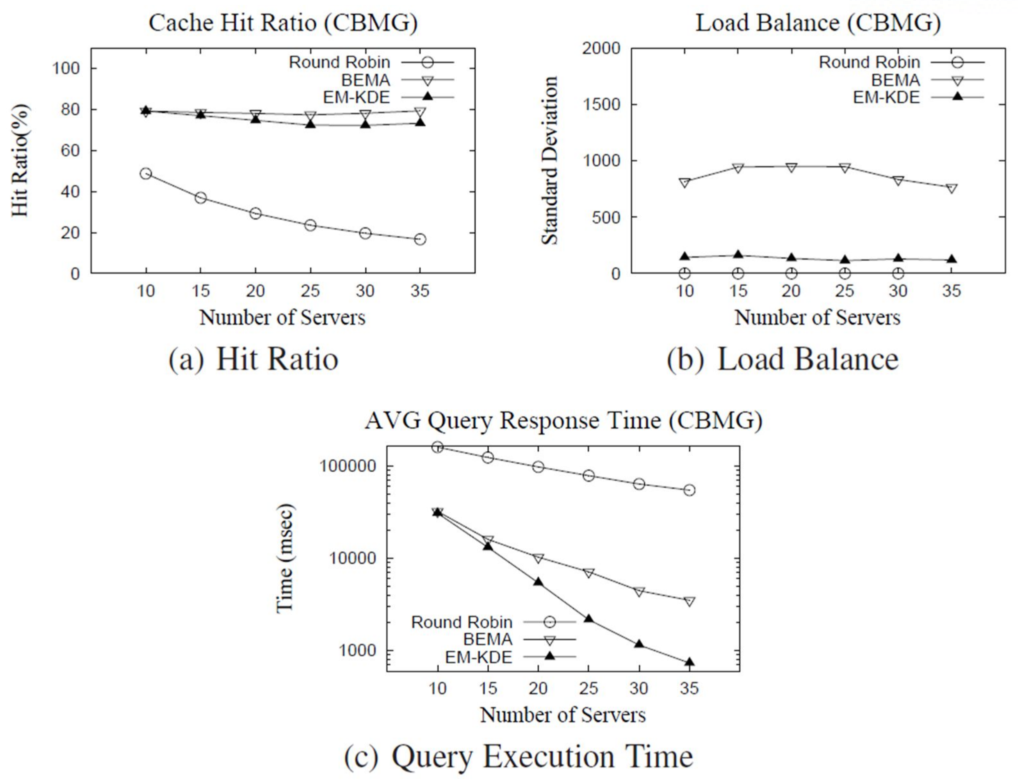


Figure 35. Performance Comparison with Varying Number of Servers (Realistic Workload)

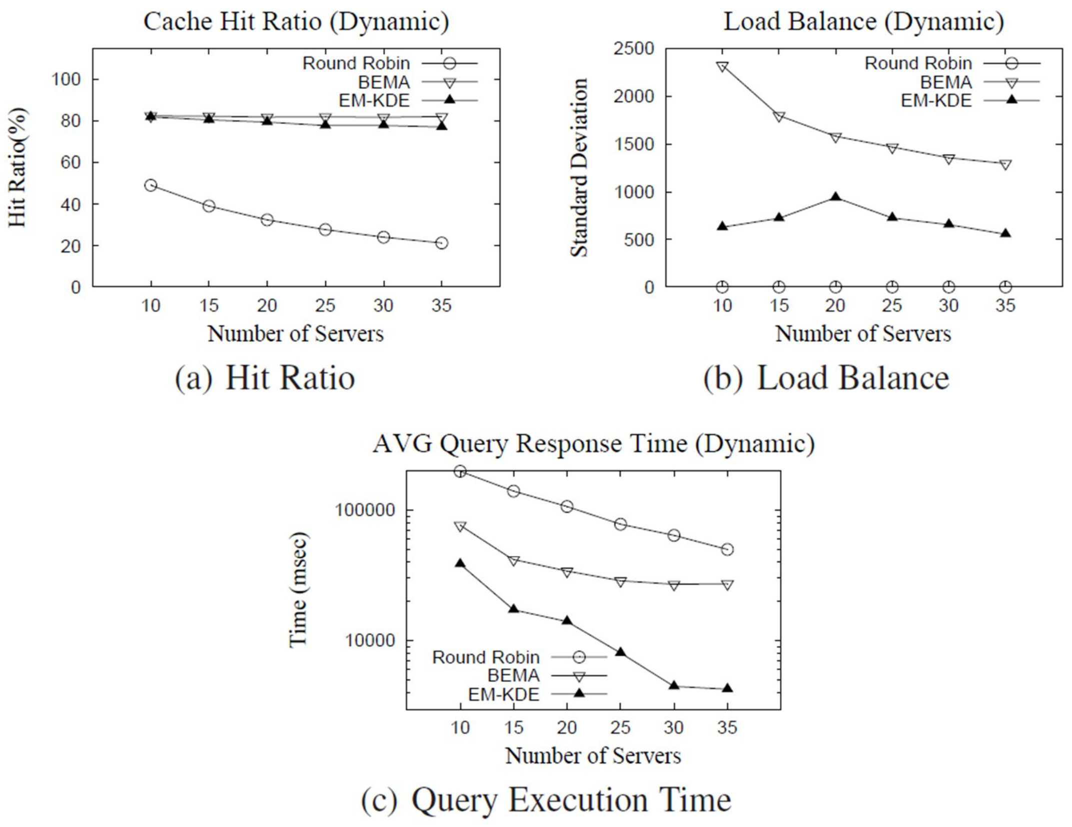


Figure 36. Performance Comparison with Varying Number of Servers (Dynamic Workload)

Figure 36 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.

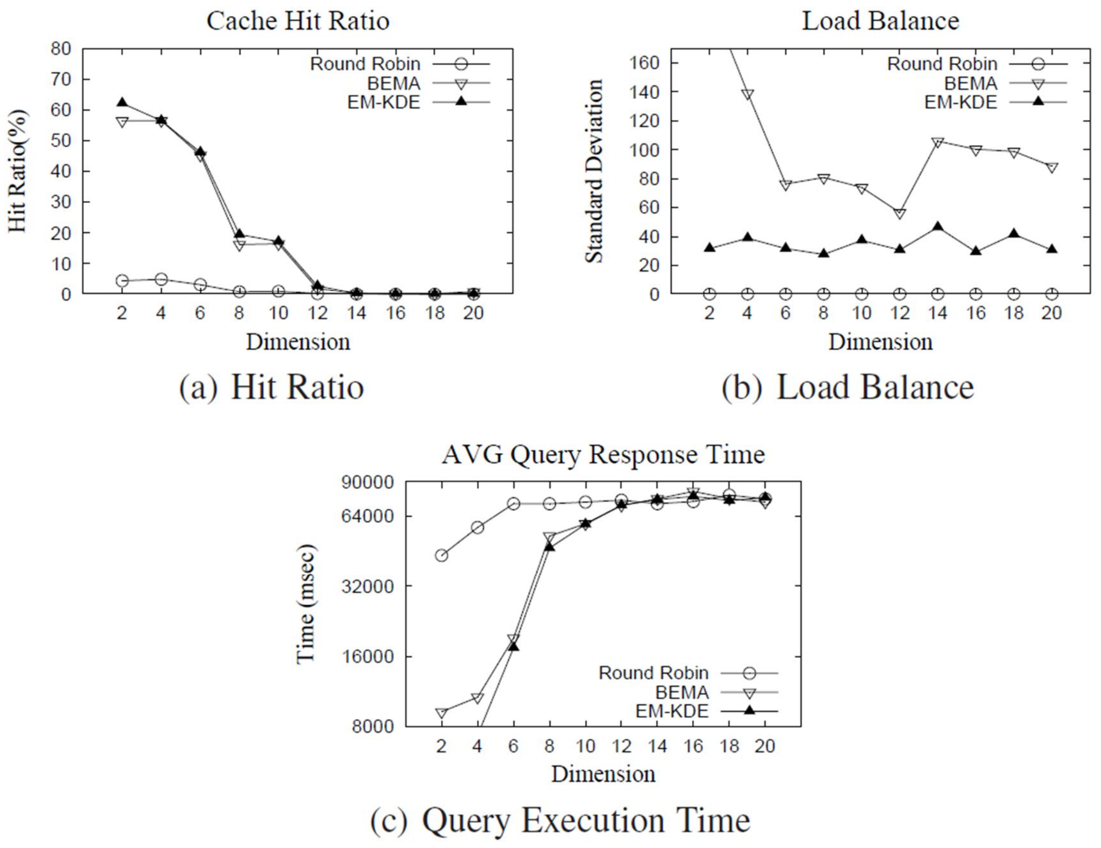


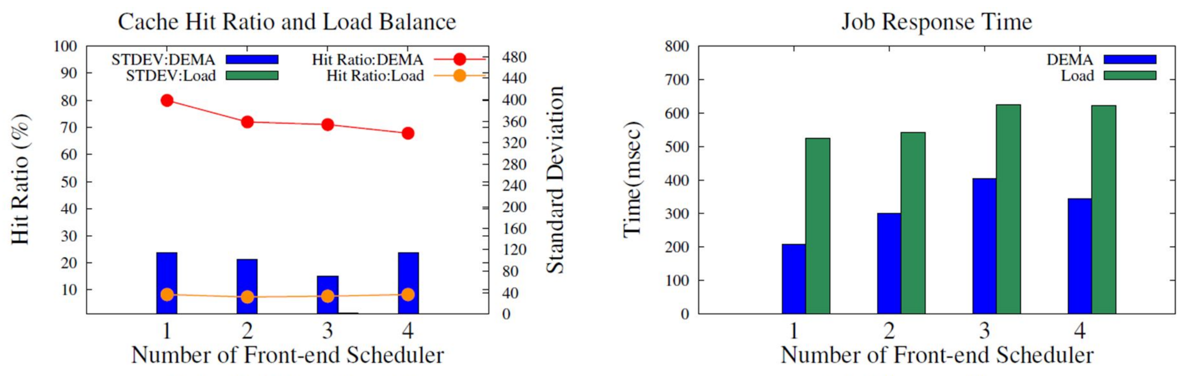
Figure 37. Performance Comparison with Varying Number of Dimensions (Uniform Workload)

In the experiments shown in Figure 37, 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.

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.

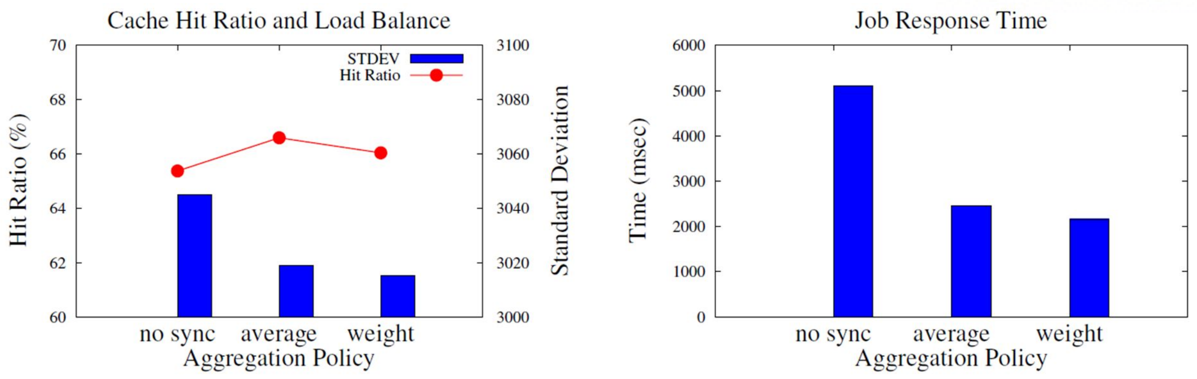
8.3 Multiple Front-end Schedulers Synchronization

To evaluate the synchronization policies, 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) Job Response Time

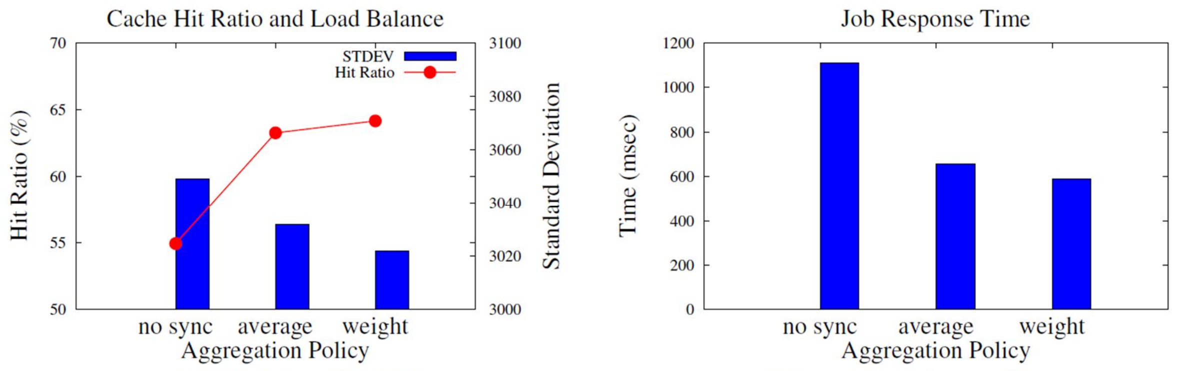
Figure 38. Performance Comparison Varying Number of Schedulers



(a) Cache Hit Ratio and Load Balance (b) Average Query Response Time

Figure 39. The Performance Comparison of Aggregation Policy(5 msec of inter-arrival time)

For the experiments shown in Figure 38, 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 scheduler. 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.



(a) Cache Hit Ratio and Load Balance (b) Average Query Response Time

Figure 40. The Performance Comparison of Aggregation Policy(10 msec of inter-arrival time)

Figure 39 and 40 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 the previous section. 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 39, 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 40, 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 highest 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 39 and 40, the averaging method and weighted averaging method perform significantly better than nosync scheduling.

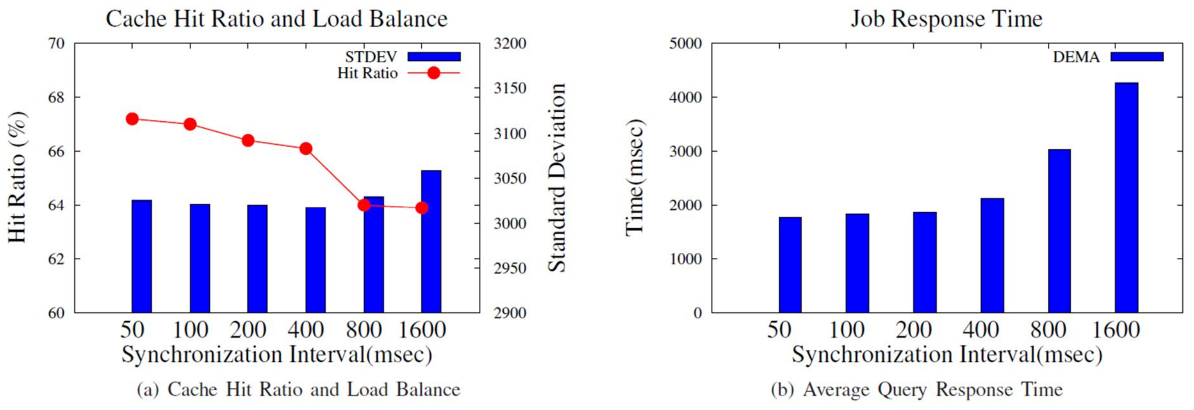


Figure 41. The Performance Effect of Synchronization Interval (5 msec of inter-arrival time)

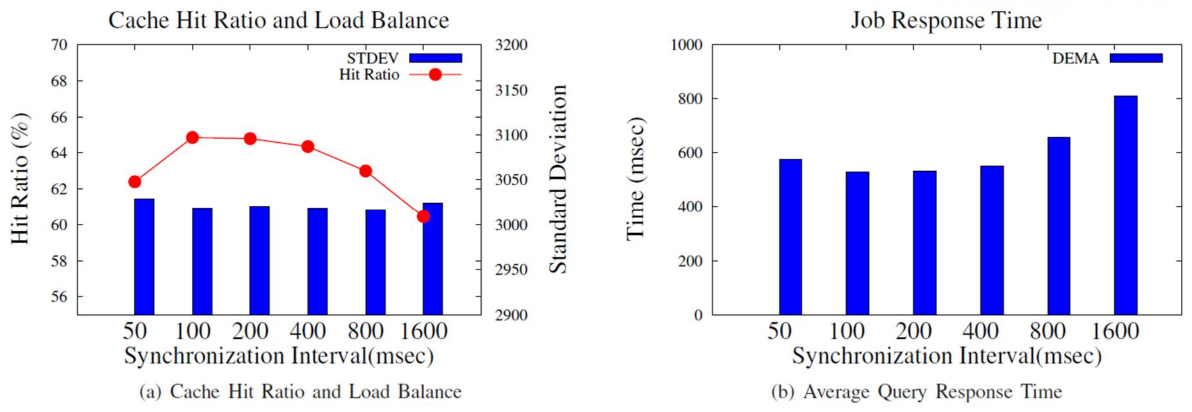


Figure 42. The Performance Effect of Synchronization Interval (10 msec of inter-arrival time)

For the experiments shown in Figure 41 and 42 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 decisions with more accurate prediction of cached data objects.

Figure 41 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.

Figure 42 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 Figure 41, 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.

8.4. Data Migration policies

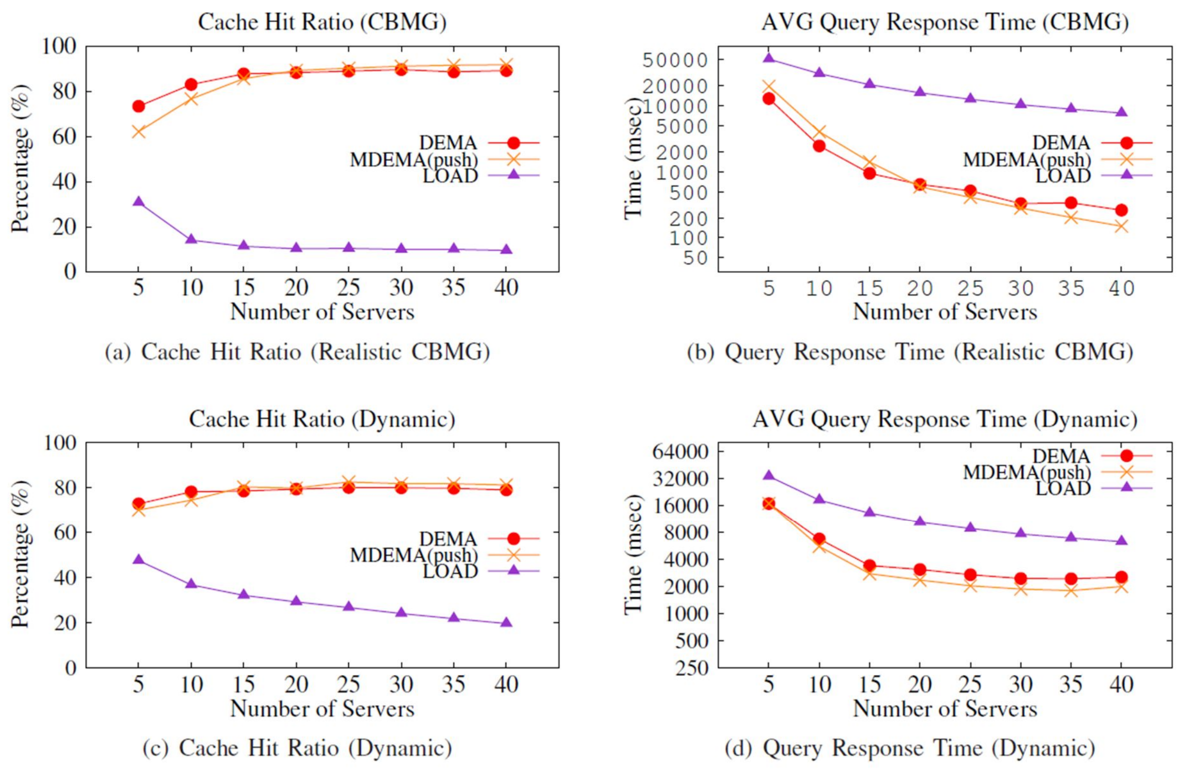


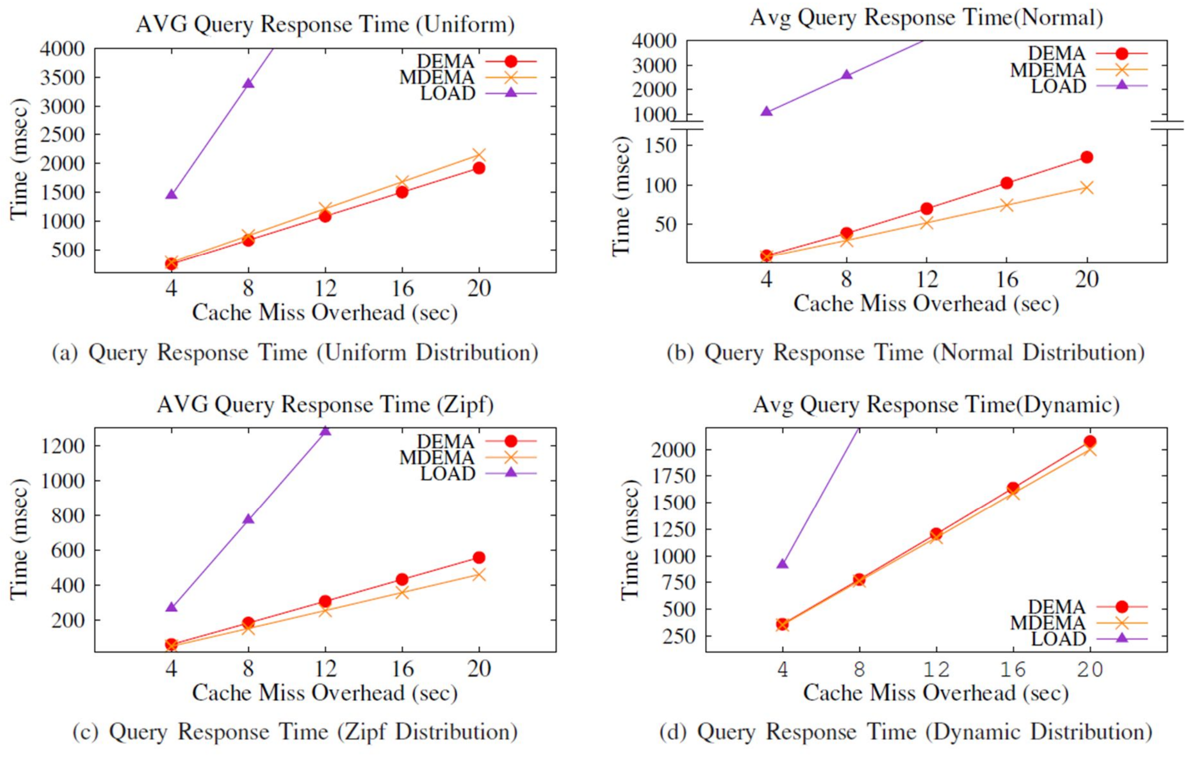
Figure 43. Performance Comparison Varying Number of Servers

8.4.1. Scalability

Figure 43 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 in this section, we fixed the EMA smoothing factor alpha to 0.03. The smoothing factor determines how much weight is given to each past query result. When alpha 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 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 server, DEMA with data migration policy outperforms DEMA scheduling policy.



(a) Query Response Time (Uniform Distribution)
(b) Query Response Time (Normal Distribution)
(c) Query Response Time (Zipf Distribution)
(d) Query Response Time (Dynamic Distribution)

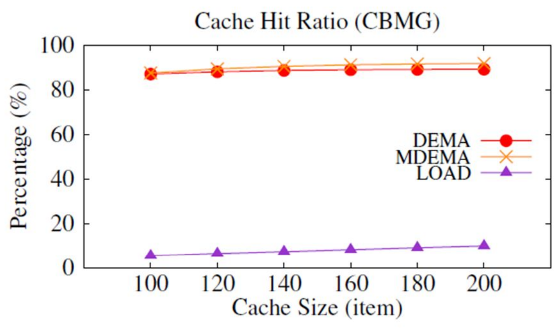
Figure 44. The Effect of Cache Miss Overhead

8.4.2. Cache Miss Overhead

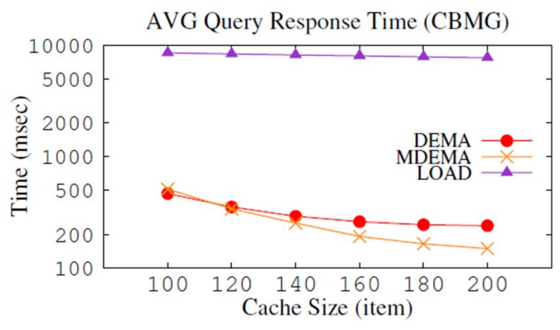
Figure 44 depicts the average query response time for uniform, normal, Zipf's and dynamic input data distributions with varying the cache miss penalty. If a query needs a larger raw data object and more computations, cache miss causes a higher performance penalty. For the experiments, we fixed the cache size so that it can hold at most 200 query results, and employed 40 back-end application servers. In the Figure 44, 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 ~ 83 % to 55 ~ 92 %, thus it exhibits up to 40 % lower average query response time. Load-based scheduling policy exhibits significantly higher average query response time because of low cache hit ratio. Hence note that y-axis in Figure 44(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 44 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 the 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.

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 msec. In the graphs shown in Figure 45, Load-based scheduling policy gains the benefit of increased cache size. Note that load-based scheduling policy didn't get the benefit of the 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.

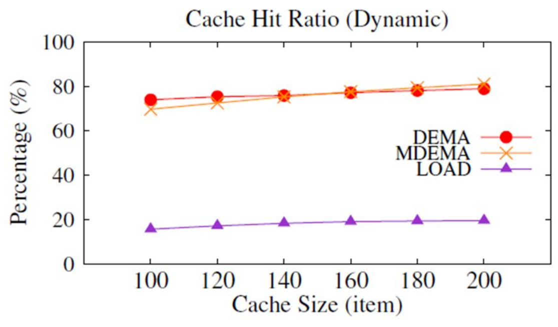
Unlike load-based scheduling policy, DEMA scheduling policy consistently shows very high cache hit ratio (> 80 %) even when cache size is small. This is because DEMA scheduling policy considers all cached objects in distributed caching system. Thus even if a single server's cache size is small, the total 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.



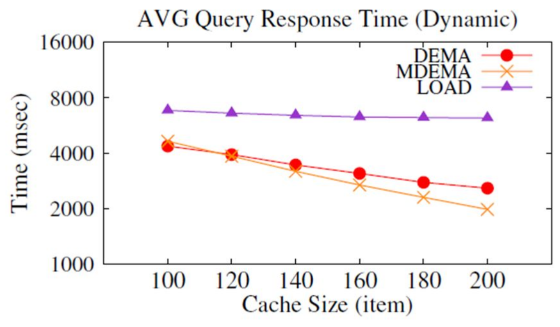
(a) Cache Hit Ratio (Realistic CBMG)



(b) Query Response Time (Realistic CBMG)



(c) Cache Hit Ratio (Dynamic)



(d) Query Response Time (Dynamic)

Figure 45. Effect of Cache Size with Realistic CBMG Workload and Dynamic Workload

It should be 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 45, 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.

8.5. Eclipse

In this section, we evaluate and analyze the performance of Eclipse with against the performance of Hadoop 2.5 and Spark 1.0.0. Spark allows user programs to load input data into main memory, but we compare the performance of Eclipse against disk-based Spark and Hadoop for fair comparisons. Chord DHT file system in Eclipse can be also improved to store input data blocks in main memory.

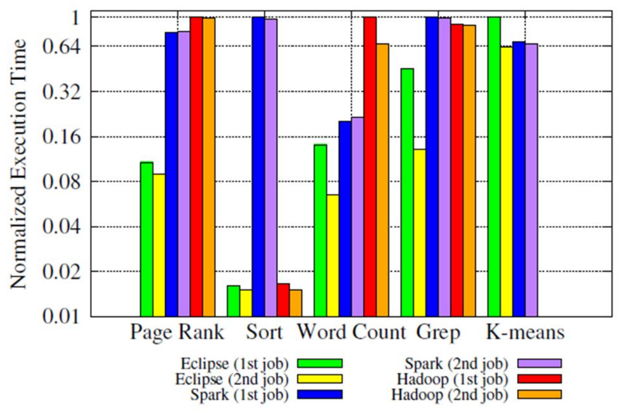
It is reported that the median job input sizes for the majority of data analytics jobs in Microsoft and Yahoo are under 14 GBytes, thus we used maximum 50 GBytes of datasets for each benchmark application. We run the experiments on two clusters of different sizes. One testbed is the same 40-node cluster which is also used in previous evaluations. Each node has dual Intel Xeon Quad-core E5506 processors and 8 GBytes memory, and all nodes are connected via 1 gigabit Ethernet. For this commodity testbed, we set both the number of map task slots and the number of reduce task slots to 8 since each node has 8 vCPUs with hiperthreading enabled. Another scale-up testbed is a dedicated 9-node Dell R720 CentOS 6.5 Linux cluster. In this scale-up testbed, each node has 128 GBytes memory and dual Intel Xeon Octa-core E5-2640v2 processors that disabled hyperthreading, and all nodes are connected via Infiniband QDR interconnects. For this scale-up testbed, we set both the number of map task slots and the number of reduce task slots to 16.

8.5.1. Job Execution Time

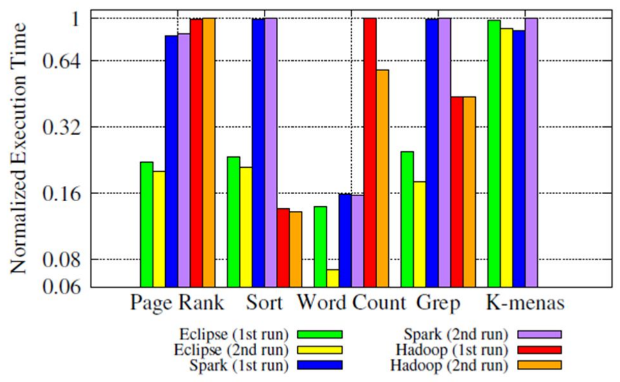
In the first set of experiments shown in Figure 46, we compare the normalized job execution times using several representative MapReduce applications and 20 GBytes input datasets. For each application, we submitted a batch of two jobs that share 50 % of input data blocks without warming up the distributed semantic caches and in-memory caches in HDFS.

For page rank application, we fixed the number of iterations to 1 in our experiments. In the commodity testbed, Spark runs 20 % faster than Hadoop while Eclipse is 9.17 ~ 11.1 times faster than Hadoop. In our scale-up testbed, Spark is only 15 % faster than Hadoop, but Eclipse performs 4.47 ~ 4.95 times faster than Hadoop. Since the 2nd job reuses the cached input data in Eclipse, it runs about 10 % faster than the first job. However, the 2nd jobs in Hadoop and Spark do not run any faster than the 1st jobs.

Interestingly Spark runs about 5 ~ 10 times slower than Hadoop for sort application. Note that Spark is designed for iterative MapReduce jobs, but the MapReduce jobs shown in Figure 46 except K-means are not iterative. This result is consistent with the previously known results; if a job does not iterate Spark shows worse performance than Hadoop because of the overhead in using Scala. Eclipse also does not show better performance than Hadoop mainly because the size of intermediate results of sort is not smaller than the input data size. Since the hash keys of intermediate results are likely to span over the entire hash key space, a large amount of intermediate results are distributed across all the servers and the benefits of pro-active shuffling are offset by the overhead. Note that the 2nd sort job does not run faster than the 1st job for all three frameworks because sort is a type of application that is hard to reuse intermediate results. The input data blocks in dCache can be reused, but the computation time dominates the I/O time.



(a) Commodity Testbed (1+38 nodes/gigabit ethernet)



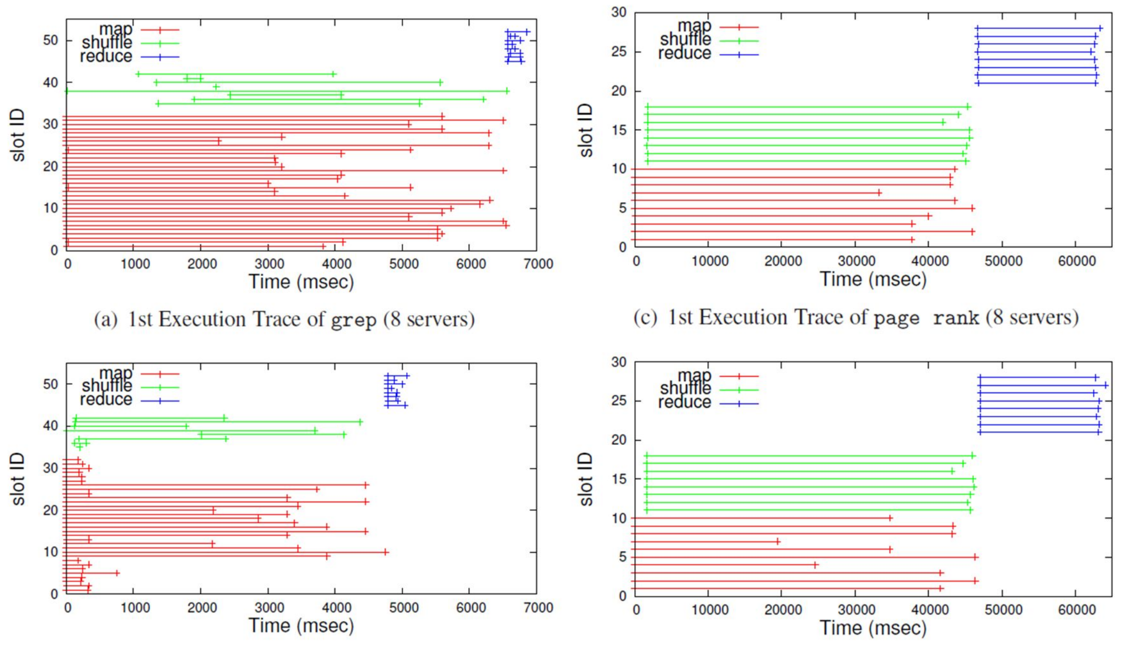
(b) Scale-up Testbed (1+8 nodes/Infiniband interconnects)

Figure 46. Normalized job execution time

WordCount is a rather I/O-bound application that can easily reuse intermediate results of previous jobs. In the commodity testbed, Eclipse runs WordCount application 7 ~ 15 times faster than Hadoop while Spark is 5 times faster than Hadoop. The 1st job in Eclipse is faster than Hadoop and Spark mainly because of the pro-active shuffling and load balancing. The 2nd job in Eclipse runs 2.2 times faster than the 1st job because the 2nd job reuses the cached intermediate results. Hadoop also runs the 2nd job about 50 % faster than the 1st job, which can be explained by the in-memory caching in HDFS. It should be noted that the in-memory caching in HDFS helps reducing the 2nd job's execution time for other applications as well. But if applications are computation intensive the input data caching in HDFS does not save significant portion of total execution time.

Grep is another simple application whose performance is mainly determined by two performance factors in map phase, i.e., load balance and input data locality. Again Eclipse shows superior performance to Hadoop and Spark. For the 1st job, Spark shows about the same performance with Hadoop, but Eclipse shows 2.12 times faster performance than Hadoop. For the 2nd job, Eclipse runs 7.7 times faster than Hadoop because of the cached intermediate results.

For K-means application, both the 1st and the 2nd jobs iterate 10 times using 20 GBytes. Since Hadoop takes too long, we couldn't quantify the performance of Hadoop, but it run at least 5 times slower than Spark. Eclipse and Spark exhibit similar performance for K-means application.



(a) 1st Execution Trace of grep (8 servers)

(c) 1st Execution Trace of page rank (8 servers)

(b) 2nd Execution Trace of grep (8 servers)

(d) 2nd Execution Trace of page rank (8 servers)

Figure 47. Execution trace of map/shuffle/reduce tasks with Eclipse

In the experiments shown in Figure 47, we capture the task traces of two consecutive Eclipse jobs using PageRank and Grep applications. Since shuffling phase in Eclipse starts as soon as some amount of intermediate results become available, the time spent on shuffling phase is masked by the execution time of map tasks. The concurrent execution of map phase and shuffle phase is not likely to have resource conflicts because shuffling phase mostly uses network resources while map tasks are CPU and disk I/O bound.

Figure 47(a) and 47(b) show half of the 2nd job's map tasks complete earlier than the 1st job's map tasks because they reuse cached intermediate results. Shuffling also completes instantly if it finds the same valid intermediate results are available in Chord DHT file system.

In Figure 47(c) and 47(d), the 2nd PageRank job also takes the advantages of cached intermediate results, but it does not result in reducing the job execution time because the total job execution time is often determined by some tasks that take longer than others. However Eclipse saves significant amount of computation for some task slots, which allows other tasks in the queue to be scheduled on the slot. As we will show, job processing throughput of Eclipse is superior to the other two frameworks.

From the trace analysis, we find there is still a room for improving the job execution time by rebalancing the workloads of map tasks that do not find reusable cached data. As a future enhancement, we plan to implement the dynamic task reallocation that migrates unprocessed input data from the straggler to other early terminating map tasks.

8.5.2. Job Processing Throughput

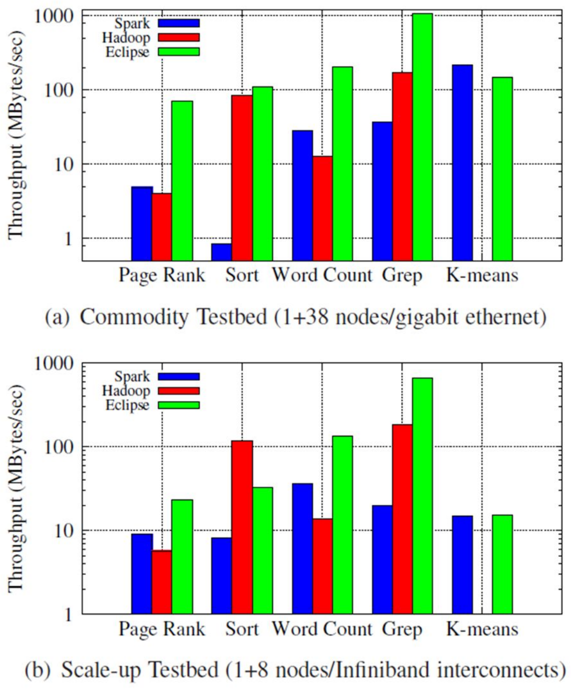


Figure 48. Throughput

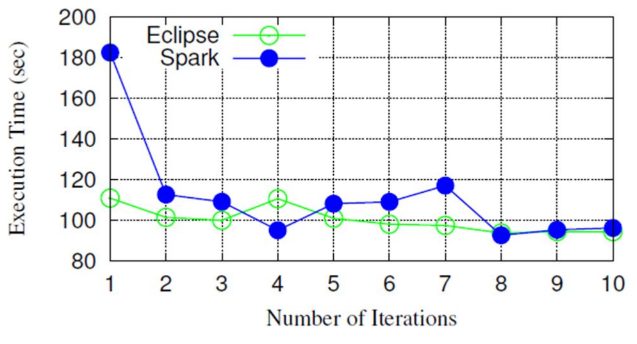
Figure 48 shows the data processing throughput of the three frameworks. For the experiments, we submitted 10 jobs at the same time that access 20 GBytes of data in total. Each job shares 50 % of its input data blocks with other jobs and read the other 50 % of input data blocks from the distributed file system. We use the default fair scheduling in Hadoop, and Spark uses the delay scheduling that delays a job for a certain amount of time if data locality is not satisfied. Eclipse uses the locality-aware job scheduling policy we described in previous section, which makes the best efforts to utilize all available map/reduce slots in the cluster.

For most application types in both commodity testbed and scale-up testbed, Eclipse shows much higher data processing throughput than the other two. Eclipse processes 70 MBytes of data per second in PageRank application while Hadoop and Spark processed only 4 ~ 4.8 MBytes per second in the commodity testbed. In the scale-up testbed, Eclipse processes 23 MBytes per second, which shows the number of cores is more important than network speed or memory size for PageRank application. Sort is the only application that Eclipse does not show the ehighest throughput, i.e., Hadoop shows 3.6x higher throughput than Eclipse in the scale-up testbed.

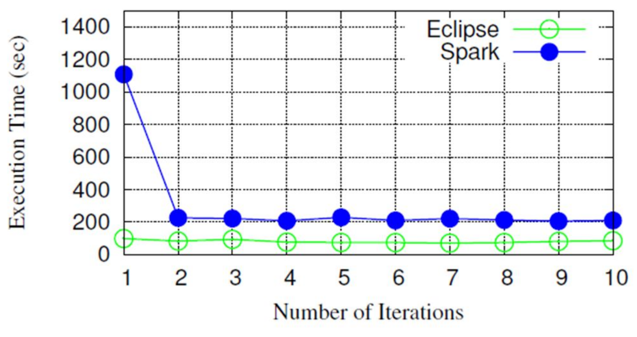
The highest data processing throughput is observed in Grep application, which requires minimal computation. Eclipse shows about 6.2 times higher throughput (1059 MBytes/sec) than Hadoop (170 MBytes/sec), and 4.9 times higher throughput than Spark (216 MBytes/sec) in commodity testbed. In the scale-up testbed the throughput of Eclipse (669MBytes/sec) is 35x higher than Spark (35 MBytes/sec). While investigating the data processing throughput of the three frameworks, we find that the delay scheduling in Spark is not very effective for a batch of jobs that read small (< 10 GBytes) input datasets. Spark tries to balance the job scheduling fairness and data locality as in Eclipse, but the delay scheduling badly affects system throughput. Even when there are empty slots, Spark makes subsequent jobs wait and the jobs are serialized in contrast to Eclipse which tries to utilize all the available slots in the cluster with subsequent jobs.

8.5.3. Iterative Applications

In the experiments shown in Figure 49 we compare the performance of Eclipse against Spark for an iterative application - K-means and PageRank using 15 GBytes datasets on the scale-up testbed. We do not show the performance of Hadoop since Hadoop is an order of magnitude slower than Eclipse and Spark especially for iterative applications. It is reported that disk-based Spark runs 10 times faster than Hadoop for iterative applications. They also report the first iteration of K-means and PageRank takes about 20 times slower than subsequent iterations. Our experiments also confirm this behavior of Spark. For the first iteration, Spark reads input dataset from HDFS, which is one performance bottleneck. The first iteration in Spark constructs RDDs that can be used by subsequent iterations, which also makes the first iteration slower than later iterations.



(a) k-means



(b) page rank

Figure 49. Execution trace with varying the number of iteration (1+8 nodes/Infiniband)

K-means implementation in Eclipse benefits from reusing input data blocks in dCache, but the intermediate results generated from map tasks are hard to reuse for the next iteration unless the k sample points do not change. Taking advantage of input data reuse, the execution time of each iteration with Eclipse slightly decreases. i.e., when the number of iteration is 1 Eclipse takes 110 seconds, and when the number of iteration is 10 it takes 94 seconds, and when the number of iteration is 10 it takes 94 seconds. The execution time of Spark K-means also decreases with iterations, but the initial iteration overhead makes its first iteration much slower than Eclipse (182 sec vs 110 sec). In subsequent iterations, Spark performs similar to Eclipse. The performance gap between the 1st and later iterations in Spark is likely due to overhead of the first iteration - reading input data from HDFS and caching them as RDDs. K-means implementation in Spark does not exploit incremental computation but it stores nothing but input data as RDDs. Since HDFS also caches input data blocks, input data RDDs created by Spark K-means seem redundant and unnecessary.

As shown in Figure 49(b) Eclipse outperforms Spark using iterative PageRank application. Again, Spark runs 11 times slower than Eclipse for the first iteration. The subsequent iterations in Spark exhibits about 5.5 times faster performance than the first iteration, but still it is about 3 times slower than Eclipse. The intermediate results of PageRank map tasks in iCache are also hard to reuse for the subsequent iterations, thus subsequent iterations in Eclipse do not perform faster than the initial iteration.

In Eclipse, the final results of reducers are stored in Chord DHT file system and the next iteration should read it from remote disks. We believe we can further improve the performance of Eclipse by providing an option to store the final results of reducers in iCache of map tasks instead of Chord DHT file system. With this enhancement, the performance gap between Eclipse and Spark is likely to become wider.

8.5.4. Input Data Size

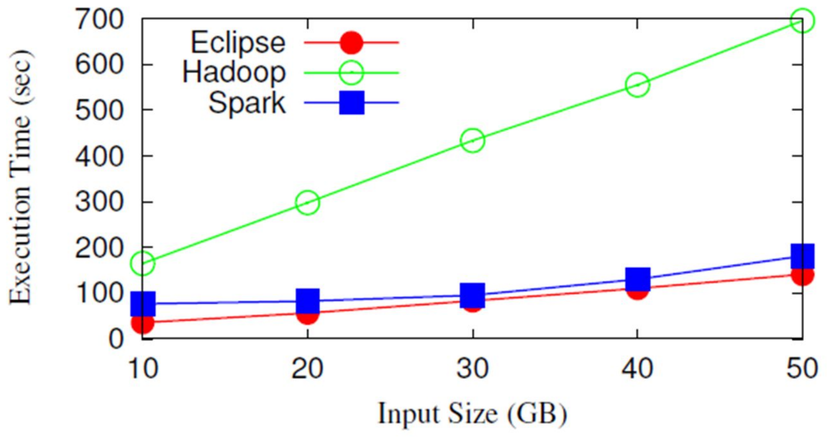


Figure 50. Job execution time with varying the input data size (1+8 nodes/Infiniband)

Figure 50 shows the job execution time of WordCount application on the scale-up testbed with varying the size of input datasets. In these experiments, we submitted a single job for each input data size, hence the cache effect of Eclipse is not reflected in the results. As the input size increases, the performance gap between Eclipse and Hadoop slightly increases. When the input size is 10 GBytes, Eclipse is 4.6 times faster than Hadoop (35.8 seconds vs 164.9 seconds), and when the input size is 50 GBytes, Eclipse is 4.92 times faster than Hadoop (141.24 seconds vs 695.241 seconds). Compared to Spark, Eclipse consistently outperforms Spark with increasing the input data size.

8.5.5. Input Data Similarity and Data Reuse Ratio

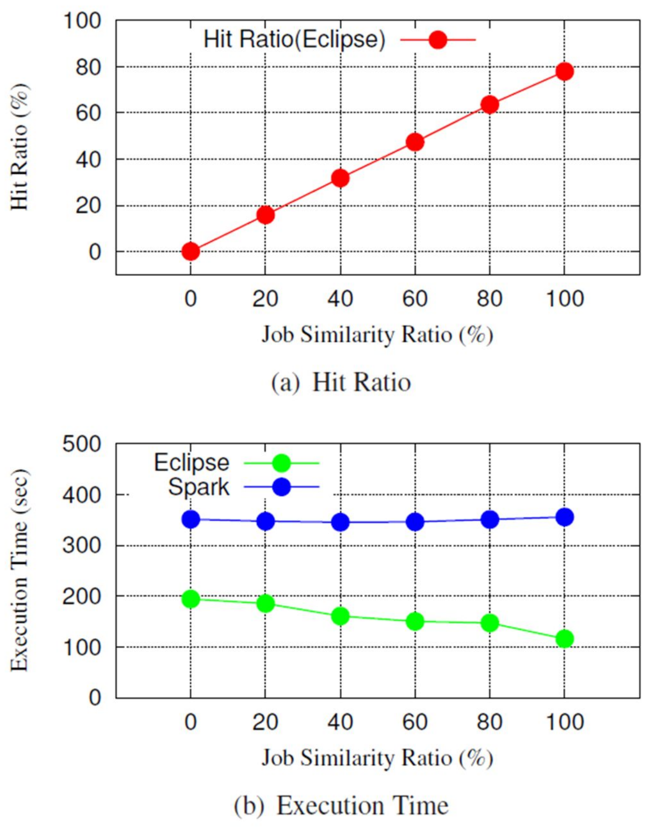


Figure 51. The effect of the job similarity on execution time (1+8 nodes/Infiniband)

For the experiments shown in Figure 51, we generated a set of WordCount jobs with varying the similarity of input datasets. When the job similarity is 0 %, we submit five jobs that do not share any input data blocks. When the job similarity is 100 %, we submit identical five jobs. Each job accesses 10 GBytes of input data out of total 50 GBytes datasets.

Figure 51(a) shows the cache hit ratio increases linearly as the job similarity increases. For WordCount application, most hits occur in iCache. If both input data and intermediate results are available in dCache and iCache, Eclipse uses the intermediate results from iCache and ignore the input data in dCache. Hence, the hit ratio of iCache is often much higher than dCache if applications can reuse intermediate results.

As cache hit ratio increases, the job execution time decreases in Eclipse. When the job similarity is 100 %, job execution time decreases by 50 %. As the size of iCache is fixed to 1 GBytes, cache hit ratio is no higher than 80 % even when the job similarity is 100 %. Contrary to Eclipse, Spark does not show different performance even when all five jobs are identical.

8.5.6. Data Reduction Ratio

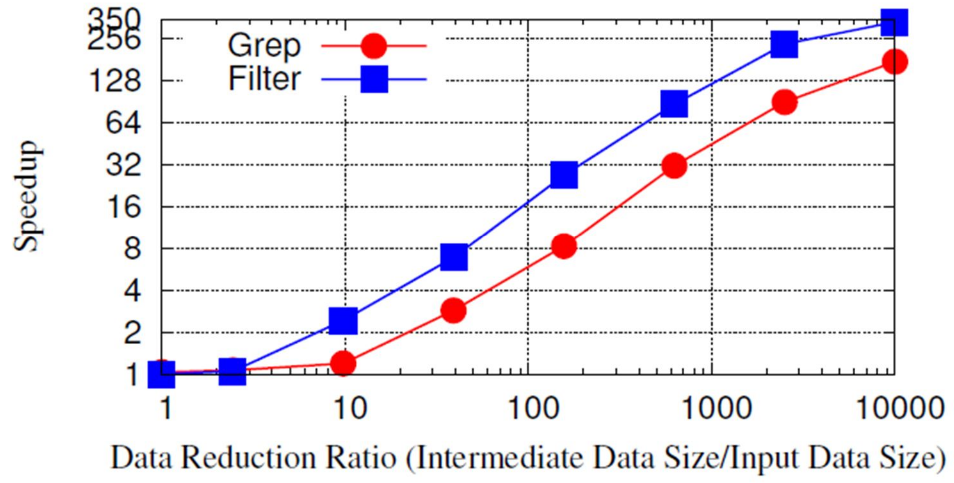


Figure 52. Speedup with varying data reduction ratio (1+38 nodes/gigabit ethernet)

For the experiments shown in Figure 52, we measure the performance effect of data reduction ratio (size of intermediate results/size of input data). When the data reduction ratio is 1, all input data are passed to reducers and when the data reduction ratio is 10,000, only one out of 10,000 input data are passed to reducers. We submit Grep and Filter job twice with the same input dataset in order to warm up the cache, and we measure the speed-up of the second job against the first job. When the data reduction ratio is small (< 10), distributed semantic caches are not large enough to hold all the input data. Thus its speed-up is not very high (< 3). But as the data reduction ratio increases, we observe up to 350x speed-up for Grep and Filter applications.

IX. CONCLUSION

In this work, we propose scheduling of the distributed query processing framework to obtain both high cache hit ratio and balanced load. Since the distribution of query arrival pattern can change over time, it is hard to maintain the load balance and cache hit ratio at the same time with a static scheduling algorithm. Therefore we need a scheduling algorithm which dynamically captures the current temporal trend of query arrival and adapts to the trend.

For this purpose, we propose DEMB scheduling algorithm as improvement of our previous work, DEMA scheduling algorithm. Although DEMA is also good at maintaining high cache hit ratio and load balance, it cannot adapt to a sudden change of query arrival pattern as it moves only one EMA point per one query arrival. To overcome the limitation of the DEMA scheduling, DEMB uses a linked list to represent the probability density function of future query arrival pattern, and it divide the one-dimensional problem space into equally probable sub-spaces for each back-end server. By this mechanism, Entire sub-ranges dedicated to each back-end server change altogether at every update interval, which overcomes the limitation of the DEMA scheduling policy.

The BEMA scheduling algorithm, another next version of DEMA scheduling algorithm also has similar weakness as DEMA scheduling policy. Although the weakness of the BEMA scheduling policy can be resolved by DEMB scheduling policy, DEMB scheduling policy also have a drawback that it cannot have a big window size to maintain query histogram because of a considerable overhead from having a big window. The EM-KDE resolves this issue by using fixed number of bin to maintain the query histogram instead of using a linked list, and outperforms other scheduling policies as well.

If the system is large, we may have servers distributed physically far from each other. In that case, we may need multiple front-end schedulers to service clients from various sites with low latency. As scheduling independently may break the data locality, and sending query information to all other front-end servers will incur significant overhead, we need a compromise of both approaches. For that purpose, EMA point synchronization is proposed. Among three synchronization policies, Weighted Averaging method performs the best. To further increase the cache hit ratio, we also implemented two data migration policies, pull migration mode and push migration mode. Each migration modes have pros and cons, and their performance difference is shown in evaluations section.

Finally, the Eclipse is implemented to utilize the EM-KDE algorithm and other techniques introduced in this work. The Eclipse uses DHT routing table for the file system, and uses EM-KDE scheduling to maintain the cache memory sub-region for each slave node. The pro-active shuffling is another important feature which significantly reduces the processing time. And as shown in the evaluations, the Eclipse outperforms the Spark and Hadoop in most of the evaluation cases.

References

1. Aron, M., Sanders, D., Druschel, P., Zwaenepoel, W.: Scalable content-aware request distribution in cluster-basednetwork servers. In: Proceedings of Usenix Annual Technical Conference (200)

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

3. Kullback, S., Leibler, R.A.: On information and sufficiency. Annals of Mathematical Statistics 22(1), 79-86 (1951)

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

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

6. Andrade H., Kurc T., Sussman A., and Saltz J. Multiple query optimization for data analysis application clusters of SMPs. In Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid). IEEE Computer Society Press, May 2002.

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

8. Isard M., Prabhakaran V., Currey J., Wieder U., Talwar K., and Goldberg A. Quincy: Fair scheduling for distributed computing clusters. In ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP), 2009.

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

10. Nam B. S., Hwang D. Y., Kim J. W., and Shin M. H. High-throughput query scheduling with spatial clustering based on distributed exponential moving average. Special issue on data intensive eScience, Distributed and Parallel Databases, 30(5-6): 401-414, 2012.

11. Zaharia M., Borthakr D., Sarma J. S., Elmeleegy K., Shenker S., and Stoica I. 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.

12. Zaharia M., Konwinski A., Joseph A. D., Katz R., and Stoica I. Improving MapReduce performance in heterogeneous environments. In Proceedings of the 8th USENIX conference on Operating Systems Design and Implementation (OSDI), 2008.

13. Apache Hadoop. Apache Hadoop. http://hadoop.apache.org/.

14. Y. Guo, J. Rao, and X. Zhou. iShuffle: Improving Hadoop performance with shuffle-on-write. In 10th USENIX International Conference on Autonomic Computing (ICAC), pages 107-117, 2013.

15. Y. Kwon, M. Balaziniska, B. Howe, and J. Rolia. SkewTune: mitigating skew in mapreduce applicatitons. In Proceedings of 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD), pages 25-36, 2012.

16. T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, and I. Raicu1. Zht: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table. In 27th IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2013.

17. T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. Sharing across multiple MapReduce jobs. ACM Transactions on Database Systems, 39(2): 12: 1-12:46, 2014.

18. L. Popa, M. Budiu, Y. Yu, and M. Isard. Dryadinc: Reusing work in large-scale computations. In Proceddings of the 2009 USENIX Conference on Hot Topics in Cloud Computing (HotCloud), 2009.

19. W. Pugh and T. Teitelbaum. Incremental computation via function caching. In the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 1989.

20. S. R. Ramakrishnan, G. Swart, and A. Urmanov. Balancing reducer skew in MapReduce workloads using progressive sampling. In Proceedings of the Third ACM Symposium on Cloud Computing (SOCC), 2012.

21. S. Sakr, A. Liu, and A. G. Fayoumi. The family of MapReduce and large-scale data processing systems. ACM Computing Surveys, 46(1):11:1-11:44, 2013.

22. A. Shinnar, D. Cunningham, B. Herta, and V. Saraswat. M3R: Increased performance for in-memory Hadoop jobs. Proceedings of the VLDB Endowment (PVLDB), 5(12): 1736-1747, 2012.

23. D. Tiwari and Y. Solihin. MapReuse: Reusing computation in an in-memory MapReduce system. In 28th IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2014.

24. J. Urbani, A. Margara, C. Jacobs, S. Voulgaris, and H. Bal. AJIRA: a lightweight distributed middleware for MapReduce and stream processing. In 34th International Conference on Distributed Computing Systems (ICDCS), 2014.

25. V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O’Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache Hadoop YARN: yet another resource negotiator. In Proceedings of the Third ACM Symposium on Cloud Computing (SOCC), 2013.

26. Y. Wang, X. Que, W. Yu, D. Goldenberg, and D. Sehgal. Hadoop acceleration through network levitated merge. In Proceedings of the ACM/IEEE SC2011 Conference, 2011.