Effective Data Distribution and Reallocation Strategies for Fast Query Response in Distributed Query-Intensive Data Environments*

Tengjiao Wang, Bishan Yang, Jun Gao, and Dongqing Yang

Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, China
School of Electronics Engineering and Computer Science, Peking University, Beijing, 100871, China
{tjwang, bishan_yang, gaojun, dqyang}@pku.edu.cn

Abstract. Modern large distributed applications, such as mobile communications and banking services, require fast responses to enormous and frequent query requests. This kind of application usually employs in a distributed query-intensive data environment, where the system response time significantly depends on ways of data distribution. Motivated by the efficiency need, we develop two novel strategies: a static data distribution strategy DDH and a dynamic data reallocation strategy DRC to speed up the query response time through load balancing. DDH uses a hash-based heuristic technique to distribute data off-line according to the query history. DRC can reallocate data dynamically at runtime to adapt the changing query patterns in the system. To validate the performance of these two strategies, experiments are conducted using a simulation environment and real customer data. Experimental results show that they both offer favorable performance with the increasing query load of the system.

Keywords: data distribution, data reallocation, query response.

1 Introduction

In recent years, more and more large distributed applications, such as mobile communications and banking services, require fast responses to enormous and frequent query requests. For example, when a telecom subscriber makes a phone call, a request is sent to a GSM switching system to query his or her location information. At the same time, there may be large amount of query requests waiting for responses when numbers of subscribers are asking for service. Each query must be responded with spectral efficiency, otherwise the phone call connection will be considered failed. This kind of application usually employs in a

* This work is supported by the NSFC Grants 60473051, 60642004, the National '863' High-Tech Program of China under grant No. 2007AA01Z191, 2006AA01Z230, and the Siemens - Peking University Collaborative Research Project.

Y. Zhang et al. (Eds.): APWeb 2008, LNCS 4976, pp. 548–559, 2008.
© Springer-Verlag Berlin Heidelberg 2008

distributed query-intensive data environment, which is different from traditional distributed environment. How to provide fast response time with extremely high query load is an essential issue greatly concerned by the service providers.

Looking into the distributed query-intensive data environment, a query request is basically processed by a system node that contains the queried data. Besides, the query load varies from node to node for the query frequencies are different on the system nodes. As a result, load imbalance will occur when some nodes are facing high access traffic while others are rarely visited or even idle. In this case, the overall performance of the system will slow down due to the imbalanced use of the system resources. Therefore, query load balancing is a key concern for improving system's performance (query response time) in this environment.

For load balancing problem in traditional distributed systems, most previous works concern ways of job allocation to balance the workload. Static load balancing[1,2,3] strategies assign jobs to the system nodes when the jobs are generated. The assignment is computed based on the information about the average behavior of the system. Once the assignment is determined, it will not change throughout the lifetime of the system. Dynamic load balancing[4,5,6,7,8] strategies balance the workload at runtime to adapt the system changes, through transferring jobs or data from an overloaded node to an underloaded node. The transferring decisions can be made based on the state information of the current system, at the expense of communication overhead over the system nodes.

Although the above strategies help balance the workload, they can not be directly used in the distributed query-intensive data environment due to several reasons: (1) The system requires spectral response efficiency for querying the query-intensive data. The response time is highly sensitive to any additional communication and processing cost in the system. (2) Oftentimes, the data size is huge and the data may be distributed among the system nodes without sharing. Therefore the query tasks can not be simply transferred from one node to another. (3) The query load is unknown previously. However, it can be predicated based on the query history in the system, since the query requests have some inherent patterns in practice. If the data is frequently queried by a subscriber in the past, it will possibly be often queried in near future.

Note that in the distributed query-intensive data environment, the query load distribution significantly depends on how the data is distributed. Therefore, in this paper, we concern ways of data distribution to improve system's performance through load balancing. More specifically, we make the following contributions:

- We provide a static data distribution strategy DDH[1], that uses a hash-based heuristic technique to distribute data off-line according to the query history. It aims to distribute data reasonably and help balance the query load of the system.
- We design a dynamic data reallocation strategy DRC[2], that can reallocate data dynamically at runtime to adapt the changing query patterns in the

[1] DDH stands for Static Data Distribution using a Hash-based Heuristic Technique.
[2] DRC stands for Dynamic Reallocation with a Central Controller.

system. The query load can be balanced by using a central controller to make reallocation decisions and conduct data migration.
- A simulation environment is designed to test the effectiveness of the two strategies. Experimental results show that they both offer favorable performance improvements with increasing query load in the system.

The remainder of this paper is organized as follows. Section 2 describes the considered distributed query-intensive data environment. In Section 3, the static data distribution strategy DDH is introduced. Section 4 describes the dynamic data reallocation strategy DRC. Section 5 describes the experiment environment and presents the evaluation results of the two strategies. Finally, section 6 concludes the paper.

2 Model Description and Formulation

To better understand our proposed strategies, we consider a distributed query-intensive data environment, which is commonly used in mobile communications[9]. (Fig. 1 shows the system architecture) The considered system can be divided into three parts: the back-end (BE), the front-end (FE) and the application client (AC). The central controller (CC) in the gray rectangle is an additional part which is designed for employing the dynamic data reallocation strategy.

The back-end is comprised of several BE servers. The BE servers hold the permanent data about subscribers, including service profiles and location information. For BE servers in the same area, the stored data is the same (these BE servers are viewed as a whole and one of them is considered as a deputy). For BE servers over different locations, there is no overlap of data, and the data can be migrated among these BE servers. The database on each BE server has homogeneous structure, but the processing power of these BE servers may be different. When receiving a query request, a BE server will execute the query on the local database, and then return the result data.

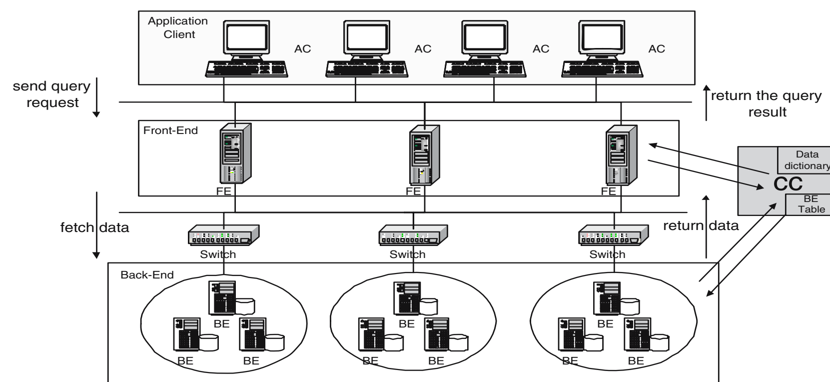

Fig. 1. A distributed query-intensive data environment

The front-end contains several FE servers. Each FE server can contact with all the BE servers in the system. The FE servers store selected administrative information about subscribers in order to service the visiting subscribers. The request submitted by a subscriber will first arrive at a FE server (the FE server is selected according to the subscriber's current location). If the needed data can be found on this FE server, the result can be directly returned to the subscriber, otherwise, the FE server routes the query to the BE server that contains the queried data, and waits for the result data and then return it to the subscriber.

The application client AC is carried by the subscriber. A query request is submitted from AC to one FE server and waits for the response. That is, subscribers only contact with FE servers in the system, and BE servers are transparent to them.

CC is a central controller which we design for employing the dynamic data reallocation strategy. In practice, CC can be implemented by a single server or run as a service in an existing server. More Details about the functionality of CC will be described in Sect. 4.

To better understand the proposed strategies in Sect. 3 and Sect. 4, we first express several definitions.

Definition 1. *Data unit T is defined as the smallest unit with a unique ID in the system. A data unit contains a record with a primary key in the relational database, or an entry with a DN in the LDAP database.*

Definition 2. *The query load associated with data unit T in time interval t is the total frequency of read execution on T during t, denote as $L(T)$.*

Definition 3. *Given the entire data-set $D = T_1, T_2, \ldots, T_n$, where $T_i (1 \leq i \leq n)$ is a data unit. A data group G is made up of k data units, $(1 \leq k \leq n)$. Then there are $\lceil n/k \rceil$ data groups in the system.*

Definition 4. *Given a data group $G = T_1, T_2, \ldots, T_k$, the query load associated with G is the sum of query loads associated with the data units contained in G, so $L(G) = \sum_{i=1}^{n} L(T_i)$.*

The basic idea of defining data groups is to make the management of the massive information more flexible. The control granularity of the system can be determined by the data group size. To better perform data distribution, we introduce a data structure called *data dictionary*, which stores the information of all data groups in the system including *data group ID, data group location and read load.* Each FE server keeps a data dictionary and CC also has one. For the dictionary in a FE server, the *read load* of a data group records the query frequency on the data group detected by the FE server (we only consider read load here since in this environment read is the main operation). For the dictionary in CC, the *read load* of a data group is the sum of the read load values of the data group recorded in all FE servers. *BE table* is another structure stored in CC which records the information of each BE server, including *BE ID, the process power(PC) and the query load(Load).* The *BE Table* gives an overview of the global query load information which can help make better decisions to reallocate data.

3 DDH-A Static Data Distribution Strategy

DDH is designed to make reasonable data distribution off-line to help obtain query load balancing in the system. As the query behavior of subscribers often has some inherent pattern, the data frequently queried by a subscriber is very likely to be queried again. Therefore we believe that it is important to take the query load history into account when concerning a reasonable data distribution.

With the query history information, DDH distributes data using a hash-based heuristic technique. In the distribution, the entire data set is presents as numbers of data groups. The main idea is to distribute these data groups randomly to buckets through hashing. Each bucket contains similar query load instead of similar data size (The query load is measured by the query frequency on the data). Then through assigning buckets averagely to the BE servers, each server can be expected to have near-equal query load.

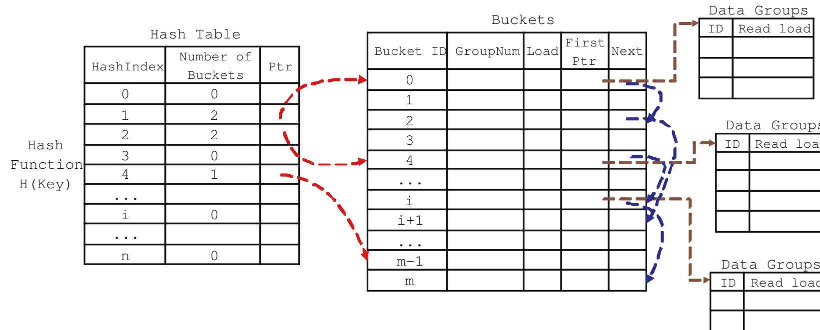

Fig. 2. Hashing process

Since the data size is usually very large in practice, a secondary-storage hash table is used. A hash function maps the search keys (the unique key feature of the data group) into range 0 to $n-1$, where n is the length of the array which consists of pointers to the header of bucket lists (n is determined by the total data size, the data group size and the maximum capacity of a bucket). Data groups that are hashed to the same address will be distributed to the same bucket. The capacity of a bucket has a upper bound, and we denote the max number of data groups that can be held in a single bucket as GN_{max}. The mean load of all the data groups in the system is DL_{avg}. A bucket overflows when the total load of its containing data groups exceeds a threshold φ, which is calculated as $GN_{max} * DL_{avg}$. If a bucket overflows, a chain of overflow buckets can be added to the bucket. A bucket table is used to organize all the buckets by storing their links. The hashing process is given in Fig. 2. As the hash table and the bucket table records the pointers instead of specific data information, they can be expected to be maintained in memory. When the hashing process is finished, all the buckets are assigned to the BE servers averagely. In this way, each BE server contains data groups with approximately the same load. The algorithm is given in Algorithm 1.

Algorithm 1. Static Data Distribution DDH

1: Create hash table HT
2: **for all** data group G **do**
3: Get the last bucket LB in the entry of $H(G.ID)$ /* H is the hash function */
4: **if** $LB.Load \geq \varphi$ **then**
5: Create a new bucket NB and $LB.NextBucket \leftarrow NB$
6: Append G to NB
7: $NB.Load \leftarrow G.ReadLoad$
8: $NB.GroupNum \leftarrow 1$
9: **else**
10: Append G to LB
11: $LB.Load \leftarrow LB.Load + G.ReadLoad$
12: $LB.GroupNum \leftarrow LB.GroupNum + 1$
13: **for all** nonempty bucket B_j in HT **do**
14: distribute the data groups in B_j to $BE[j\%n]$ /* n is the number of BE servers in the system */

In sum, DDH provides a static strategy to distribute data off-line to help balance the query load of the system. The data distribution is a hash-based heuristic process based on the information of the query load history. Furthermore, the distribution process is very efficient due to the effectiveness of hash, and the location index of the data groups can be built up simultaneously during the distribution process. Distributing data reasonably with DDH can greatly help obtain query load balancing in the system, and thus the query response time can be expected to be sped up.

4 DRC-A Dynamic Data Reallocation Strategy

Since the query behavior of subscribers is not invariable in practice, one can not expect the query load to be balanced throughout the lifetime of the system based on the initial data distribution. Therefore, we propose DRC, a dynamic data reallocation strategy, that can reallocate data dynamically at runtime to adapt the changing query patterns. The query load can be balanced by using a central controller to make reallocation decisions and conduct data migration. The reallocation decisions are made based on the global query load information. Similar to the dynamic load balancing strategy in [4], DRC is comprised of three parts: information rule, location rule and migration rule. Together with the migration rule, a data migration mechanism for ensuring data consistency is proposed.

Information rule. The information rule describes the collection and storing methods of the query load information of the runtime system. To avoid making decisions favoring only some system nodes, it needs to collect the global

information of the system. Denote the read load associated with data group G_i on FE_j as $L_{FE}(i,j)$. Assume there are k FE servers in the system called $FE_1, FE_2, \ldots, FE_k$. The total load of data group G_i is the sum of read loads recorded in all the FE servers, expressed as $L(i) = \sum_{j=1}^{k} L_{FE}(i,j)$. Since the data groups have no overlap on BE servers, we can calculate the query load on BE_i as $L_{BE}(i) = (\sum_{j=1}^{m} L(j))/C(i)$, where m is the number of data groups stored on BE_i, and $C(i)$ is the query processing power of BE_i.

To avoid large expense of communication, CC initiatively notifies all the FE servers to collect the load information of each data group. Then the query load information of all the BE servers can be calculated. In order to avoid traffic, the collection can be done periodically or at a time when the system is relatively idle. When obtaining the query load of a BE server, CC stores it in the corresponding entry in the *BE Table*.

Location rule. The location rule aims to choose a sender node and a receiver node for data migration. The basic idea is to define a maximum deviation of the average query load to help determine whether the node is overloaded or under-loaded. We define $AvgL$ as the mean load of all BE servers in the system, $AvgL = (\sum_{i=1}^{n} L_{BE}(i))/n$, where n is the number of BE servers in the system. It can help to estimate whether the system is busy or idle currently. With deviation θ (can be set based on experience), $AvgL - \theta$ is defined as the lower bound and $AvgL + \theta$ the upper bound of query load for each BE server. For a BE server, if the load on it exceeds $AvgL + \theta$, then it will be chosen as a sender candidate (sender condition); if the load is under $AvgL - \theta$, then it will be chosen as a receiver candidate (receiver condition). Since $AvgL$ is the mean load of all BE servers, if there is a BE server satisfying the sender condition, there must be another BE server satisfying the receiver condition.

Migration rule. The migration rule determines when or whether or not to migrate data. First, the algorithm chooses the node with the max load among all the sender candidates as a sender node. Then it scans all the data groups on the sender node, in decreased order of their load values, to find whether there is one that do not make the load on the sender node decrease below $AvgL - \theta$ and the load on the receiver node increase upon $AvgL + \theta$ after data migration. The receiver node is the node with the minimum load among all the receiver candidates. Once the data group is found, the migration decision can be made. This procedure will be executed iteratively until there are no sender candidates in the system. The algorithm is given as algorithm 2. The data migration mechanism is described in the *Migration Mechanism* part.

Migration Mechanism. To ensure data consistency, a data migration mechanism is designed. Fig. 3 shows the mechanism. At the beginning of the migration,

Algorithm 2. Dynamic Data Reallocation DRC

/* T_{BE} is the BE table on CC, D is the dictionary stored on CC*/
1: **while** the set of sender candidates is not empty **do**
2: Select sender node BE_s with the max load from the sender candidates
3: **for all** data group G_i on BE_s (in decreased order on load) **do**
4: Select receiver node BE_r with the minimum load from the receiver candidates
5: **if** $T_{BE}[BE_r].Load + D[G_i].load/T_{BE}.[BE_r].PC \geq AvgL + \theta$ and $T_{BE}[BE_s].Load - D[G_i].load/T_{BE}[BE_s].PC \leq AvgL - \theta$ **then**
6: Migrate data G_i from BE_s to BE_r
7: Update the $Load$ value for BE_r and BE_s in T_{BE}
8: Update location information for G_i in data dictionaries on CC and all FE servers
9: **if** $T_{BE}[BE_s].Load \leq AvgL + \theta$ **then**
10: break

CC sends a command to the sender node BE_s for migrating data G to the receiver node BE_r (step 1). Then BE_s migrates the data G to BE_r (step 2). If BE_r successfully receives and stores G, it sends a success signal to CC (step 3). CC changes the location record of G from BE_s to BE_r in its own data dictionary (step 4), and then broadcasts messages to all FE servers to ask for location update in their data dictionaries (step 5). Each FE server performs location update (step 6), and then sends a success signal to CC (step 7). After receiving the success signals from all FE servers, which means the location information has been updated in all data dictionaries in the system, CC sends a command to BE_s for deleting G (step 8). Then BE_s performs the deletion (step 9). Only after these 9 steps are successfully done can the migration be considered accomplished. At each step, logs are created to record the state information of the current system. If a failure occurs at any step, the system can be recovered to its original state. For example, if the data dictionary on a FE server is unsuccessfully modified, then all the dictionaries on FE servers and CC will be restored to their original state and the migrated data will be deleted from the receiver node BE_r. During the migration process, CC plays an important part in controlling the message signals to and from the other system nodes and conducting the actions of these nodes.

The above strategy overlooks the distance between a FE server and a BE server in the system. A problem will arise when some FE servers are set at a remote site from BE servers, since the transform time between them should not be neglected. We define $D(FE_i, BE_j)$ to be the distance from FE_i to BE_j. Suppose the distance information can be obtained before taking the reallocation strategy. To tackle the distance problem, the reallocation algorithm can be modified by the following heuristic technique. For each sender candidate BE_s, the load values of its data groups are calculated as $\sum_{FE_j \in SFE} L_{FE}(i,j)\}$, where $L_{FE}(i,j)$ is the query load associated with data group G_i which is detected by

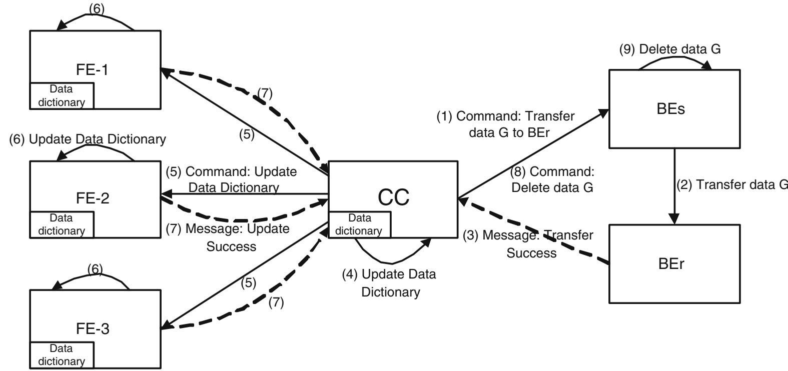

Fig. 3. Data Migration Process

FE_j, m is the number of data groups storing on BE_s, and SFE is a set storing k FE servers which are the nearest to BE_s.

In sum, DRC is developed to reallocate data dynamically at runtime for query load balancing in the changing system. The strategy is implemented by a central controller CC, which makes reallocation decisions based on the global query load information and controls data migration for protecting the system from migration failures. To avoid incurring large communication overhead, CC sends a collection request to the systems nodes at a suitable time rather than receives notifications from the system nodes all the time. Through balancing the query load at runtime, the query response time can be expected to be fast with the changing query load in the system.

5 Experiments

For the purpose of this study, we prepared real customer data from a mobile communication company. The subscriber database is stored in the LDAP database (LDAP database is commonly used in the query-intensive environment). It contains 200,000 subscribers with size 350MB, covering attributes like the basic DN, the current location and IMSI (International Mobile Subscriber Identification Number) of each subscriber. The simulation system model consists of a collection of six computers connected by a communication network with 100.0 Mbps bandwidth. The central controller CC ran on a AMD Athlon 64 3200+ processor with 1G RAM. One FE server and two BE servers used AMD Athlon 64 3200+ processor with 1G RAM, and one FE server and one BE server used 4 AMD Opteron 2.4G processors with 32G RAM.

In the experiments, a query was created by generating an id number, which was used as the key for searching the subscribers' location information in the database. Queries were generated 6000 times per second and randomly sent to one of the FE servers. Then the FE server routed the query to a corresponding BE server where the query can be processed.

First, we evaluated the effectiveness of the static data distribution strategy DDH. For comparison purpose, an algorithm NoLB was used. It distributes data based on the commonly used horizontal fragmentation method and does not consider any query load information. As for DDH, a hash function is chosen to take u modulo p, where p is a prime number that is approximately equal to n. The data group size is set to be 10 records. The system load level is measured by the ratio of the total query arrival rate in the system to the total processing power of all BE servers.

Fig. 4 compares the average response time of DDH to NoLB. The result shows that the average response time of DDH is much more faster than that of NoLB with the increasing system load level. This is because the data is distributed more fairly by DDH at the initial stage of the system, and the query load on the system nodes is more balanced. The query response time can be reduced due to the efficient use of the processing resources in the system. The improvement is more significantly when the system load is high.

To measure the fairness of data distribution, a fairness index I proposed in [10] is used. $I = \frac{[\sum_{i=1}^{n} F_i]^2}{n \sum_{i=1}^{n} F_i^2}$, where F_i is the expected response time of node i. If the expected response time is the same for all nodes, then $I = 1$, which means the system achieves fairness distribution. Otherwise, if the expected response time differs significantly from one node to another, then I decreases, which means the system suffers load imbalance to some extent. Fig. 5 shows that DDH decreases more slowly on fairness index with the increasing system load.

In the next, we evaluated the effectiveness of the dynamic data reallocation strategy DRC. In the experiment, CC collected query load information from all FE servers at a time when the system is idle (we control the time by detecting the performance of the runtime system). The deviation of the average query load θ is set at 10% of the average load. The data group size is the same as that used in the first experiment. The performance of NoLB, DRC and DRC* is compared. DRC* is the improved version of DRC that takes account of the distance factor. A FE server was set out of the inner-net to increase the communication distance. Form the result in Fig. 6, DRC and DRC* both show better performance than NoLB on the average response time. Because dynamically reallocating data can contribute to achieve load balancing and adapt the changing query patterns. When a system node is detected as overloaded, the queried data can be migrated from it to a relatively idle node. Therefore the overall performance of the system can be improved at runtime. As expected, DRC* spends less response time than DRC, since it makes better reallocation decisions by taking account of the distance factor.

We also compare the off-line probability of NoLB, DRC and DRC* versus the system load level. In mobile communication systems, the response time for a submitted query request must be within 0.4ms, otherwise, the system will consider the connection as off-line. In practice, subscribers require the off-line probability to be under 0.5%. Fig. 7 shows that DRC and DRC* both have lower

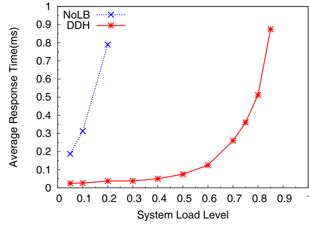

Fig. 4. Average response time(1)

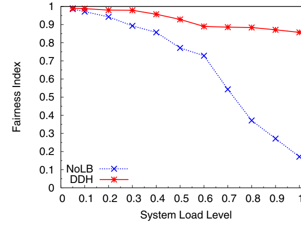

Fig. 5. Fairness Index

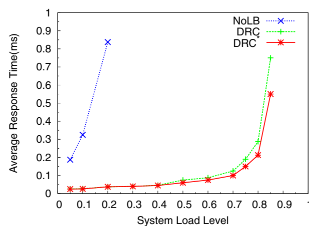

Fig. 6. Average response time(2)

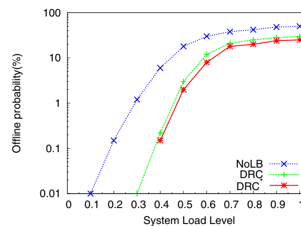

Fig. 7. Off-line probability

off-line probability than NoLB. This is because the query response time is sped up by using DRC and DRC*, and the off-line probability is significantly reduced.

6 Conclusion

To address the response efficiency challenge in the distributed query-intensive data environment, we have presented two effective strategies to speed up system's query response time through load balancing. One is DDH, a static data distribution strategy, that uses a hash-based heuristic technique to distribute data off-line according to the query history. The query load of the system can be more balanced due to DDH's reasonable data distribution. The other is DRC, a dynamic data reallocation strategy, that can reallocate data dynamically at runtime for query load balancing in the changing system. A central controller CC is used to make reallocation decisions based on the global load information for better decision quality. Much communication overhead can be avoided by using CC to collect query load information at a suitable time. Moreover, CC can control the data migration process and ensure the data consistency in the process. A simulation environment is designed to test the effectiveness of these two strategies, experimental results show that they both offer favorable performance with the increasing query load of the system.

Acknowledgments. We gratefully thank Ling Wu for her previous work on this paper.

References

1. Grosu, D., Chronopoulos, A.T., Leung, M.-Y.: Load Balancing in Distributed Systems: An Approach Using Cooperative Games. In: Parallel and Distributed Processing Symposium, pp. 52–61 (2002)
2. Li, J., Kameda, H.: Load balancing problems for multiclass jobs in distributed/parallel computer systems. IEEE Transactions on Computers 47(3), 322–332 (1998)
3. Kim, C., Kameda, H.: Optimal static load balancing of multi-class jobs in a distributed computer system. In: 10th International Conference on Distributed Computing Systems, pp. 562–569 (1990)
4. Lin, H.-C., Raghavendra, C.S.: A Dynamic Load-Balancing Policy with a Central Job Dispatcher (LBC). IEEE Transactions on Software Engineering 18(2), 148–158 (1992)
5. Sundaram, V., Wood, T., Shenoy, P.: Efficient Data Migration in Self-managing Storage Systems. In: IEEE International Conference on Autonomic Computing, pp. 297–300 (2006)
6. Zhang, Y., Kameda, H., Hung, S.-L.: Comparison of dynamic and static load-balancing strategies in heterogeneous distributed systems. Computers and Digital Techniques, IEE Proceedings 144(2), 100–106 (1997)
7. Qin, X., Jiang, H., Zhu, Y., Swanson, D.R.: A dynamic load balancing scheme for I/O-intensive applications in distributed systems. In: International Conference on Parallel Processing Workshops, pp. 79–86 (2003)
8. Kuo, C.-F., Yang, T.-W., Kuo, T.-W.: Dynamic Load Balancing for Multiple Processors. In: 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, pp. 395–401 (2006)
9. Feldmann, M., Rissen, J.P.: GSM Network Systems and Overall System Integration. Electrical Communication, 2nd Quarter (1993)
10. Jain, R.: The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling. Wiley-Interscience, New York (1991)