Cost Effective Load-Balancing Approach for Range-Partitioned Main-Memory Resident Data

Djahida Belayadi[1], Khaled-Walid Hidouci[1], Ladjel Bellatreche[2], and Carlos Ordonez[3]

[1] LCSI laboratory, Ecole Nationale Superieure d'Informatique, Algiers, Algeria
(d_belayadi@esi.dz, w_hidouci)@esi.dz
[2] LIAS/ISAE-ENSMA, 86960 Futuroscope, France
bellatreche@ensma.fr
[3] University of Houston, USA
carlos@central.uh.edu

Abstract. Due to the availability of larger main-memory capacities, we are witnessing the presence of parallel memory-based database systems offering increased performance unlike traditional databases. Data partitioning is a precondition for these databases because it significantly improves the performance of certain queries (e.g. range queries). Partitioning creates a problem of *data skew*. As a result, it can contribute to the degradation of query performance. Garcia-Molina's group [1] has proposed an algorithm that offers a low ratio between the maximum and minimum loads of nodes while exploiting information of the global load. This information is stored in a data structure, called *skip graphs*, which requires the exchange of $(log\, p)$ messages between the p nodes. The goal of our work is to reduce the number of these messages. To do this, we propose a vector of approximate partition statistics ($\mathcal{APV}$), where the nodes and clients have an approximate view of the data distribution. The key point of our proposal is the "Approximate Partitioning Vector" ($\mathcal{APV}$), where, both nodes and clients have an approximate information about the load distribution.

Keywords: Data skew, load-balancing, parallel database, range-partitioning.

1 Introduction

Range-partitioning maps tuples to partitions according to a partitioning key. This scheme is the most common type of partitioning and is often used with times (hour/minute). A key requirement in such systems is that the data has to be uniformly partitioned on all nodes. This requirement is challenging enforcing when the input data is skewed. *Data skew* is a well-known concern in range-partitioning where only few partitions (nodes) may be more loaded than others. In that case, data migration approaches are an appealing solution. The out-of-range data has to be moved from the over-loaded area to under-loaded one in order to satisfy the storage balance requirement. Data movement must

be accompanied by a change in the partition statistics (partition boundaries, neighbors loads, and the position of the most and least loaded node, etc.). All nodes/clients of the system must be aware of these changes in order to decide whether to call the balancing algorithm and address the right node. One of the dominant measures that we want to optimize in such a situation is communication cost. The focus is on the solutions that reduce the cost of maintaining data distribution information.

Ganesan et al.[1] proposed an on-line load-balancing algorithm on a linearly ordered nodes. Their algorithm called AdjustLoad guarantees a low imbalance ratio between the maximum and the minimum load among nodes. Although, the AdjustLoad algorithm is easy to state, each balancing operation may require global load information, that may be expensive in term of operations costs. Their algorithm uses a data structure called skip graph [2] to maintain load information and ensure efficient range queries. Each invoked balancing operation may require global information with a cost of $O(log\, n)$ messages. Moreover, a change of partition boundaries between neighbors in load-balancing will necessitate a change in the two skip graphs used.

In this paper, we improve the Ganesan et al. work [1] by reducing the cost of maintaining partition statistics. We propose an on-line balancing technique of range-partitioned data with approximate information. Our algorithm uses the same primitive operations, NbrAdjust and Reorder as in Ganesan et al. The primitive NbrAdjust transfers the surplus of data from the current node to one of its neighbors. The primitive Reorder changes the nodes order to achieve the storage balancing requirements. As a result, the partition boundaries change as well as the data size of each partition. However, our algorithm is not based on skip graphs to maintain partition statistics. The key point of our contribution is the Approximate Partitioning Vector ($\mathcal{APV}$), where, both nodes and clients have approximate information on data distribution statistics. Each entry $APV[i]$ in this vector, is an estimate of partition boundaries and data size related to node N_i. After a balancing operation, the participating nodes may change their own boundaries. These nodes do not need to inform the other ones by these changes. The clients use their $\mathcal{APV}$ to direct the queries. As a result, clients may address the wrong node when their $\mathcal{APV}$ are outdated. Nevertheless, whenever an interaction happens between two peers (node or client), they exchange their $\mathcal{APV}$ in order to correct each other. Our solution outperforms the state-of-the-art methods in terms of communication cost. There is no additional cost for maintaining load statistic as in Ganesan et al.

This paper is organized as follows: in Section 2, we present the load-balancing approach of Ganesan et al. We describe the system model in Section 3. In Section 4, we present our approach. We experimentally evaluate it in Section 5. Finally, we discuss the related work in Section 6.

2 Load-Balancing Solution of Ganesan et al. [1]

Ganesan et al.'s work is believed to be representative of the prior art. It suggested an inspiring algorithm for reducing data skew. It guarantees a small imbalance ratio σ between the largest load and the smallest one. This ratio is always bounded by a small constant which is 4.24. The algorithm uses two operations: 1) NbrAdjut, where the node N_i transfers its surplus of data to one of its neighbors (N_{i+1} or N_{i-1}, 2) Reorder: the least loaded node (N_r) among all the nodes transfers its entire content to one of its neighbors and change its logical position to share data with the node performing the load-balancing algorithm.

In both operations, the over-loaded node requires non-local information (neighbors loads, the position of the most and least loaded node). A given node attempts to shed its load whenever its load increases by a factor δ. For some constant c, Ganesan et al. define a sequence of thresholds $T_i = c\delta^i$, for all $i \geq 1$. The node N_i attempts to trigger the AdjustLoad procedure whenever its load $L(N_i)$ is greater than its threshold T_i.

Ganesan et al. use two skip graphs. Skip graphs are circular linked lists [3], in which every node has $log(n)$ pointers. Routing between two nodes needs $O(logn)$ messages. The first skip graph is used to get neighbors loads (one message) and to route range queries to the appropriate node. The second skip graph is used to get the positions of the most and least loaded node in the system ($O(logn)$ messages for locality plus costs of updating the two skip graphs). The main disadvantage of this work is the costly need to maintain the load-balancing information.

3 System Architecture

In this section, we define a simple abstraction of a parallel database and make some assumptions:

- $N = \{N_1, N_2, \ldots, N_p\}$, a set of p nodes connected by a fast local area network as in a Shared-Nothing architecture. We consider a relation (or a data set) divided into p range-partitions on the basis of a key attribute, with boundaries $R_0 \leq R_1 \leq \ldots \leq R_p$. The node N_i manages a range $[R_{i-1}, R_i($. We consider that the nodes are ordered by their ranges, this ordering defines left and right relationships between them. The need to preserve the order between the nodes requires a communication only between the neighboring nodes. Note that the data is In-memory resident and it is organized row wise. Each node N_j has its own Approximate partitioning vector APV_{n_j}. An entry $APV_{n_j}[i]$ in this vector (for i different from j), is an estimate of partition boundaries and data size related the node N_i. The entry $APV_{n_j}[j]$ contains exact information about partition boundaries and data size of the node N_j (each node has its exact local information).
- $C = \{C_1, C_2, \ldots, C_m\}$, a set of m clients performing insert, delete or range queries. Point queries can be considered as special case of range queries, where upper and lower bounds are equal. The clients may join or leave the

system at any time. Each client C_j has its own Approximate partitioning vector APV_{c_j}. An entry $APV_{c_j}[i]$ in this vector, is an estimate of partition boundaries and data size related to the node N_i. Clients use their Approximate partitioning vector APV_c to find the right nodes for insert, delete or search operations.

- We assume that each node N_i sets a local load threshold. When the load goes outside this limit, the node performs a load-balancing operation with its neighbors. This operation updates the partition boundaries of the participating nodes. Our load-balancing algorithm is invoked on a node at which the insert or delete is occurring or a node receiving data from its neighbors. At this stage, we assume that no central site is used to direct queries. We also ignore concurrency control issues and consider only the serial schedule of inserts and deletes, interleaved with the executions of the load-balancing algorithm.

4 Our Approach

The main feature of our approach is the $\mathcal{APV}$ concept, where each node or client has an approximate knowledge about the partition statistics. Based on this knowledge, the node performs a load-balancing whenever its load passes a local threshold. It invokes the NbrAdjust procedure that transfers the out-of-range data and its vector towards its neighbors if it is possible, otherwise, it performs the Reorder procedure. The neighbor, after having received the data, performs the load-balancing algorithm and eventually updates its vector. The process is repeated at each node receiving the data until the whole system is balanced.

4.1 Node/Client Approximate Partitioning Vector

Consider that the partition statistics of a node N_j are encoded in a vector $APV_{n_j}[1, p]$. Each element $APV_{n_j}[i]$ of this vector is a triplet, it stores an estimate of the node N_i information. Node information is mainly the upper boundary ($APV_{n_j}[i].Upper\_Bound$), the local data size ($APV_{n_j}[i].Load$) and the last updating time ($APV_{n_j}[i].Last\_Update$). The last field ($APV_{n_j}[i].Last\_Update$) is used to indicate the time when the entry $APV_{n_j}[i]$ was last updated. Node vector is updated in case of insert or delete queries, range queries and data migration from the current node to one of its neighbors or vice versa.

The data stored in the nodes is manipulated through the insert, delete and range queries sent by the clients. Consider that the partition statistics of a client C_j are encoded in a vector $APV_{c_j}[1, m]$, where each $APV_{c_j}[i]$ stores the information about the node N_i. Node information is essentially its upper boundary ($APV_{c_j}[i].Upper\_Bound$), data size ($APV_{c_j}[i].Load$) and the last updating time ($APV_{c_j}[i].Last\_update$). Inserting, deleting or searching for a tuple with a given key k are performed as follows:

- The client sends the request to N_i so that: $APV_{c_j}[i-1].Upper\_Bound \leq k \leq APVc_j[i].Upper\_Bound$. The local APV_{c_j} vector is also included in the same message.
- A node N_i checks whether the included key k fits its range, if so, it executes the specified request (insert, delete or just point-search), updates eventually its partition statistics and sends a positive acknowledge consisting of its APV_{n_i} to the client.
- If k is outside the node's range, a vector adjustment message (VAM) including a negative acknowledge and the current APV_{n_i} is sent to the client. Another solution may be proposed where the node redirects the request to the apropriate node, this solution may reduce the communication cost but the client vector will be little updated.
- If a client receives a positive acknowledgment from the node, it just updates its vector if it was outdated. Else, if it receives an adjustment message, it updates its vector and repeats the operation until receiving a positive acknowledgment.

4.2 Data Load-Balancing Algorithm

Our load-balancing algorithm uses the two universal load-balancing primitives, Reorder and NbrAdjust as in Ganesan et al.'s work. However, we use the $\mathcal{APV}$ concept to maintain the global load information instead of using the skip graphs. Our algorithm, that we call DataLoadBalancing, is presented bellow (Algorithm 1). A node N_i executes the load-balancing algorithm whenever its load increases beyond a threshold T_i. The algorithm uses its partitioning vector to check if data can be shared with the lightly loaded neighbors. If the load of one of its neighbors is less than half of N_i's load, then N_i performs the primitive NbrAdjust to average out the load with it. Else, N_i attempts to perform Reorder with the least loaded node in the system. Note that one can avoid the additional launch of the load-balancing algorithm by raising the threshold or more precisely by increasing the factor δ.

5 Experimental Evaluation

In this Section, we present results from our simulation of our approach on a network of 8 nodes and 2 clients. Processing node software and client software were executed on machines with Intel(R) Core(TM) i7-5500U CPU@2.40GHz and 8GiB of RAM. Both nodes and clients were connected through a Gigabit Ethernet network. Algorithms are implemented in C language using the Message Passing Library (Open MPI). In the experiments, we present the algorithm for the insert-only case. This case is simpler to analyze and provides general ideas on how to deal with the general case. It is also of practical interest because in many applications, as in a file sharing, deletions rarely occur. In order to evaluate the new approach in heavy skewed environment, the system is studied under a simulation model that we call $HOTSPOT$. All insert operations are directed to a

Algorithm 1: DataLoadBalancing (N_i, APV_{n_i})

```
1  Let N_j be the lightly loaded neighbor of N_i (N_{i+1} or N_{i-1});
2  if (APV_{n_i}[i].Load/2 ≥ APV_{n_i}[j].Load) then
3      //The nod will call the NbrAdjust procedure;
4      Send the (APV_{n_i}[i].Load − APV_{n_i}[j].Load)/2 tuples to N_j;
5      Update N_i's vector
       (APV_{n_i}[i].Load, APV_{n_i}[j].Load, APV_{n_i}[i].Upper_Bound);
6      //Re-call the load-balancing procedure again in N_i and N_j;
7      DataLoadBalancing (N_i, APV_{n_i}[i]);
8      DataLoadBalancing (N_j, APV_{n_i}[j]);
9  else
10     //The nod will call the Reorder procedure;
11     Find N_r so that: ∀k ∈ [1,p], APV_{n_i}[k].Load ≥ APV_{n_i}[r].Load;
12     if (APV_{n_i}[i].Load/4 ≥ APV_{n_i}[r].Load) then
13         //N_r is going to change its location to be a N_i's neighbor ;
14         Let N_j be the lightly loaded node between N_{r+1} and N_{r-1}, the two
           neighbors of N_r;
15         Send APV_{n_i}[r].Load tuples to N_j;
16         N_j changes its position to be N_i's neighbor;
17         Send APV_{n_i}[i].Load/2 to N_r;
18         Update N_i's vector (APV_{n_i}[i].Load, APV_{n_i}[r].Load, N_i, N_j, and N_r
           upper bounds);
19         DataLoadBalancing (N_i, APV_{n_i});
20         Rename nodes appropriately after the Reorder;
21     else
22         The system is balanced;
23     end
24 end
```

single hot node. We use a sequence of $5*10^4$ frequent insert operations. The basic performance factors of our load-balancing mechanism are the imbalance ratio, the number of client addressing errors, data movement cost and the number of invocation of DataLoadBalancing algorithm.

5.1 Imbalance Ratio

We measure the imbalance ratio as the ratio between the largest and smallest load after each insert operation. As the system's thresholds are an infinite, increasing geometric sequence, as in Ganesan et al.'s work, we measure the imbalance ratio with three values of the factor $\delta, (\delta = \phi, \delta = 2, \delta = 4)$. ϕ is the golden ratio, $\phi = (\sqrt{5}+1)/2 = 1.62$. Figure 1 shows the imbalance ratios (Y-axis) against the number of insert operations (X-axis). When $\delta = 1.62$, the curve shows that the imbalance ratio is always bounded by a constant 6 and it converges to 1.8 after $2 * 10^4$ operations unlike the imbalance ratio of Ganesan et al., that is bounded by 4.24 and converges towards 3.3. The spikes in the curve mean that an invocation of DataLoadBalancing algorithm has been lunched.

However, the results of the 3rd experiment $\delta = 4$ are different from the previous ones, the ratio values are larger and converge towards 5.

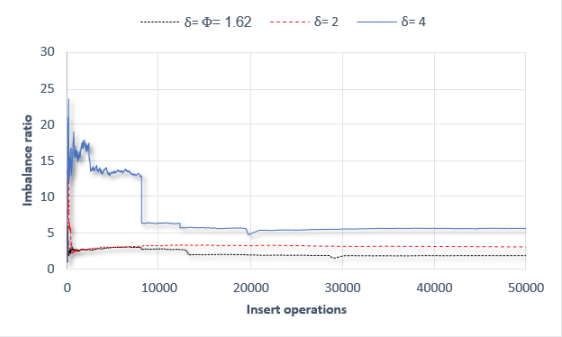

Fig. 1. The imbalance ratio when $\delta = \phi = 1.62$, $\delta = 2$, and $\delta = 4$.

Table 1. Comparison between the AdjustLoad algorithm of Ganesan et al., and our load DataLoadBalancing algorithm.

Procedure	Largest load (Tuples)	Mean load (Tuples)	Imbalance ratio	Query performance
AdjustLoad	1885	781	2.41	21%
DataLoadBalancing	1492	781	1.91	0
AdjustLoad	2102	1048	2.02	27%
DataLoadBalancing	1568	1048	1.49	0

Simulations were run comparing performance of our load-balancing algorithm and the AdjustLoad procedure of Ganesan et al.[1]. The data from Table 1 represents the comparative load-balancing results of the two procedures. The comparison parameter is the imbalance ratio which is measured here as the ratio between the largest load and the system average load.

When the performance of the system is measured by query response time, it is proportional to the largest node load. The worst-case relative performance of the DataLoadBalancing algorithm versus the AdjustLoad procedure is the ratio of the highest load-balance ratios obtained for the two algorithms, or 1.0-(1.91/2.41)=0.21. One can expect to reduce query response time up to 28% as compared to a system using the AdjustLoad.

5.2 Client Adjustment Messages

After a balancing operation, two nodes at least change their partition boundaries due to the data migration, which leads to changing the partitioning vectors of

these two nodes. The client with an outdated vector can address a wrong node that has changed its boundaries. In our set of experiments, we were interested in determining how efficiently a client obtains a true view about the nodes. We measure the number of times the client sends a query to a wrong node and hence makes an addressing error and receives a vector adjustment message (VAM). The results showed in figure 2 present a rapid increase of addressing errors number in the growing phase (from 1 to 1000 insert operations). This comes back to the fact that there is a frequent invocation of the balancing algorithm, and therefore a frequent change of the partition boundaries.

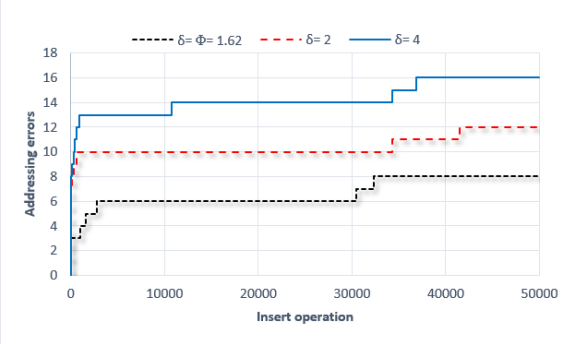

Fig. 2. The number of addressing errors ($\delta = \phi = 1.62$, $\delta = 2$, and $\delta = 4$)

5.3 Performance Analysis

For balancing loads among nodes, we are also concerned with the minimization of the movement cost as much as possible. After measuring the imbalance ratio and the client vector adjustment, we next measure the data movement cost. 3(a) plots the cumulative number of tuples migrated by our algorithm (Y-axis) against the number of insert operations (X-axis) during a run with $\delta = 1.62$, $\delta = 2$. We observe that in the growing phase, the number of migrated tuples for different δ values is rising, this is because keeping the system tightly balanced causes a larger number of re-balancing operations. Figure 3(b) plots the data movement cost when $\delta = 4$.

Figure 4 illustrates the number of invocations of our load-balancing algorithm (Y-axis) against the number of insert operations (X-axis) during a run. The observation we make is that the number of invocations of the algorithm increases for the three values of δ during the growing phase. The number of algorithm invocations presented was measured when $\delta = 1.62$, $\delta = 2$ and $\delta = 4$. We observe that the number of invocations is relatively small when $\delta = 1.62$. This comes back to the fact that we are supporting some imbalance situations.

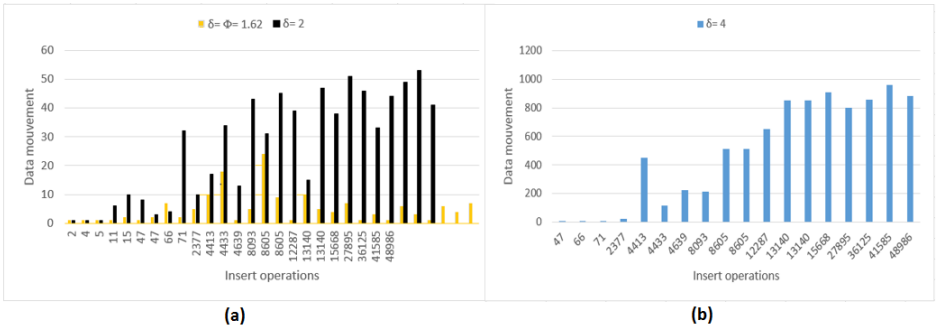

Fig. 3. (a): Data movement cost when $\delta = \phi = 1.62$, $\delta = 2$. (b): Data movement cost when $\delta = 4$

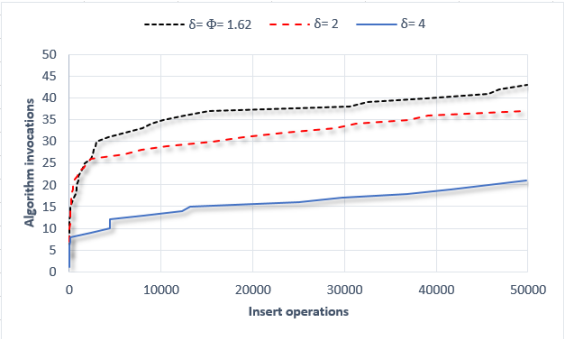

Fig. 4. Number of DataLoadBalancing invocations when $\delta = 1.62$, $\delta = 2$ and $\delta = 4$.

6 Related Work

In this section, we describe the current researches that relate to ours achieved load-balancing method while supporting range queries.

Peer-To-Peer network: in P2P networks, a number of recent load-balancing approaches have been proposed [4–8]. Structured P2P networks are an efficient tool for storage and location of data since there is no central server which could become a bottleneck. Many researches have been proposed on search methods in Structured P2P networks.

A concurrent work to Ganesan et al.'s work is presented by Karger and Ruhl [9]. The load-balancing algorithm is randomized, it offers a high bound on the imbalance ratio (more than 128). In [6], the team Jakarin et al. has also improved the work of Ganesan et al. However, Jakarin et al. use the skip graphs just like Ganesan et al.'s work. The main drawback is the costly maintenance of these data structures.

Parallel/Distributed databases: a load-balance operation in a parallel database is performed as a transaction. Work in [10] propose a multi-reorder

operation that finds a sequence of multiple adjacent nodes that have a small average load of any such sequence. This technique uses partition statistics which include an estimate of the number of tuples stored on each node for every relation in the database. Based on this information the system skew is calculated. The problem that could be noted is the cost of maintaining partition statistics.

7 Conclusion

The present paper relates to load-balancing in parallel database systems. We proposed an effective on-line data load-balancing algorithm that deals with the problem of skewed data. Our experimental results that we set in our laboratory show that our approach does not need extra cost of maintaining partition statistics as opposed to the cost of efficient solutions from the state-of-art. Our procedure needs a very low overhead (or almost no cost) to locate the data even in the presence of high degree of skew. Although our proposal was presented in the context of balancing storage load, it can be generalized to balance execution load, all that is required is an ability to partition load evenly across two machines.

References

1. Ganesan, P., Bawa, M., Garcia-Molina, H.: Online balancing of range-partitioned data with applications to peer-to-peer systems. In: (e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004. (2004) 444–455
2. Aspnes, J., Shah, G.: Skip graphs. In: Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 12-14, 2003, Baltimore, Maryland, USA. (2003) 384–393
3. Pugh, W.: Skip lists: A probabilistic alternative to balanced trees. Commun. ACM (1990)
4. Felber, P., Kropf, P., Schiller, E., Serbu, S.: Survey on load balancing in peer-to-peer distributed hash tables. IEEE Communications Surveys and Tutorials (2014)
5. Mirrezaei, S.I., Shahparian, J.: Data load balancing in heterogeneous dynamic networks. CoRR (2016)
6. Chawachat, J., Fakcharoenphol, J.: A simpler load-balancing algorithm for range-partitioned data in peer-to-peer systems. Networks (2015)
7. Antoine, M., Pellegrino, L., Huet, F., Baude, F.: A generic API for load balancing in structured P2P systems. In: 26th IEEE International Symposium on Computer Architecture and High Performance Computing Workshop, SBAC-PAD Workshop 2014, Paris, France, October 22-24, 2014
8. Takeda, A., Oide, T., Takahashi, A., Suganuma, T.: Efficient dynamic load balancing for structured P2P network. In: 18th International Conference on Network-Based Information Systems, NBis 2015, Taipei, Taiwan, September 2-4, 2015
9. Karger, D.R., Ruhl, M.: Simple efficient load-balancing algorithms for peer-to-peer systems. Theory Comput. Syst. (2006)
10. Rishel, W.S., Rishel, R.B., Taylor, D.A.: Load balancing in parallel database systems using multi-reordering (September 30 2014) US Patent 8,849,749.