Fine-grained Data Partitioning Framework for Distributed Database Systems

Ning Xu
« Supervised by Bin Cui »
Key Lab of High Confidence Software Technologies (Ministry of Education)
& School of EECS, Peking University
ning.xu@pku.edu.cn

ABSTRACT

With the increasing size of web data and widely adopted parallel cloud computing paradigm, distributed database and other distributed data processing systems, for example Pregel and GraphLab, use data partitioning technology to divide the large data set. By default, these systems use hash partitioning to randomly assign data to partitions, which leads to huge network traffic between partitions.

Fine-grained partitioning can better allocate data and minimize the number of nodes involved within a transaction or job while balancing the workload across data nodes as well. In this paper, we propose a novel prototype system, LuTe , to provide highly efficient fine-grained partitioning scheme for these distributed systems. LuTe maintains a lookup table for each partitioned data set that maps a key to a set of partition ID(s). We use a novel lookup table technology that provides low cost of reading and writing lookup table. LuTe provides transaction support and high concurrency writing with Multi Version Concurrency Control (MVCC) as well.

We implemented a prototype distributed DBMS on Postgresql and used LuTe as a middle-ware to provide fine-grained partitioning support. Extensive experiments conducted on a cluster demonstrate the advantage of the proposed approach. The evaluation results show that in comparison with other state-of-the-art lookup table salutations, our approach can significantly improve throughput by about 20% to 70% on TPC-C benchmark.

1. INTRODUCTION

Large data processing are common in the Internet, particularly for social networks. For example, the graph of web pages is reported to have at least one trillion edges, and the social network graph such as Facebook has one billion users and 140 billion friendship connections [5]. Data partitioning is a key issue to process these "big data" with high efficiency.

These recent emerging parallel data processing systems, for example Pregel, GraphLab, PowerGraph [7, 6, 4] and distributed relation database systems use hashing for data partitioning by default. This simple partitioning strategy which assigns data to partitions at random with balanced data size is fast and easy to maintain. However, it lacks consideration of the relationship between data, and could lead to huge network traffic or distributed transactions for large number of edge cuts between partitions. Taking distributed database system for example, SNS providers, such as Facebook and Twitter, use distributed RDBMS to process OLTP queries on data records with complex relationships. If a query of fetching recent events of a user's friends uses a hash partitioned table, the records are likely to be randomly placed on several partitions, which brings distributed transactions and slows down the query. Fine-grained partitioning is proposed to solve this problem. It allocates data to partitions exploiting internal relationships between data records [3]. For example, we can place the data of a user's friends to the same partition which reduces distributed transactions when fetching recent events mentioned above. A solution of fine-grained partitioning is maintaining a lookup table, which is a mapping from a partitioning key to partitioning ID(s) that stores the data. With the help of lookup table, data can be placed in an arbitrary way [11].

There are many works focus on fine-grained partitioning algorithms, in this paper, we focus on how to provide fine-grained lookup table service efficiently instead of the algorithms. The lookup-table we discussed is distributed stored in each node who needs to access it often. A central lookup-table sever is inefficient in distributed systems since every remote lookup requires another network round trip and the server itself is a bottleneck of the system. For the distributed lookup table, there are mainly two types:

a) Consistent lookup table: The lookup table in every Router is consistent all along. This would provide guarantee that the lookup table entities are always correct. However, when a tuple is moved or inserted, the cost to maintain consistency is prohibitive because distributed transactions would be used to update the other Routers' lookup tables.

b) Inconsistent lookup table: A Router executing a query that changes a certain tuple will not affect the other Routers immediately. Lookup table entities of that tuple on the other Routers stay unchanged and become stale. Thus lookup table update is free of transactions. When stale lookup table entities are used, a broadcast will be issued to correct the lookup tables and to find the desired tuple. In this way, the transactional overhead of updating all the Routers' lookup tables is mitigated. However, accessing a stale lookup table entity generates broadcast for updating lookup table entities.

Copyright is held by the International World Wide Web Conference Committee (IW3C2). IW3C2 reserves the right to provide a hyperlink to the author's site if the Material is used in electronic media.
WWW'14 Companion, April 7–11, 2014, Seoul, Korea.
ACM 978-1-4503-2745-9/14/04.
http://dx.doi.org/10.1145/2567948.2567949.

To the best of our knowledge, there are few recent works on supporting fine-grained partitioning for distributed database systems or data processing systems. Aubrey et al. [11] proposed a fine-grained partitioning framework for OLTP database systems. The approach is based on inconsistent lookup table which is inefficient when accessing stale lookup table entities. In addition, the framework is tightly coupled with the proposed prototype database systems that cannot be migrated to other systems easily.

In this paper, we propose a novel prototype system LuTe to provide highly efficient fine-grained partitioning for distributed database and other distributed data processing systems. LuTe provides transaction support and high concurrency writing with Multi Version Concurrency Control as well. It uses a novel lookup table technology named semi-consistency lookup table that integrates the advantages of Consistent lookup table and Inconsistent lookup table. Semi-consistency lookup table uses temporary lookup table to maintain the correct result for the stale lookup table entity and updates the stale one when system is idle.

We built a distributed DBMS by extending a popular open source system Postgresql [1] and used LuTe to provide fine-grained partitioning support. We evaluated our approach using TPC-C [2] benchmark on a test bed of 10-node cluster. The evaluation results show that our fine-grained partitioning approach can significantly improve throughput by about 17% in comparison with other state-of-the-art lookup table salutations, and 57% to 70% better throughput than simple hash partitioning.

The remaining of this paper is organized as follows. In Section 2 we give an overview of lookup table and related works. Section 3 discusses the framework of LuTe . In Section 4 we present the experimental study. At last, we conclude this work in Section 5.

2. LOOKUP TABLE AND IMPLEMENTATION

In this section, we introduce the detailed lookup table mechanism and the related works.

To better understand the lookup table with transaction support, in the following of this paper, we discuss the lookup table used in distributed relation DBMS which is more complicated comparing with distributed data processing systems. Thus, we use **tuple** to denote the basic data unit in the system, **Routers** to denote the machines that directly access lookup tables and **Data Nodes** to denote the machines storing the real data. The lookup table maps each tuple to a set of Data Node ID(s) where the tuple is stored. In this paper, we only discuss situations that one tuple is stored on only one Data Node. Note that, in practice, a tuple might be stored on multiple Data Nodes. This can be solved by designating one partition for each value as the primary partition as mentioned in [11].

Basically, there are two implementations of lookup table:

Consistent Lookup Table: Consistent Lookup Table guarantees that all the Routers store the correct Data Node ID for every tuple. When a tuple is updated or a new tuple is inserted, all the Routers update their lookup tables in the same transaction of the update or insertion. In this type of lookup table, every update or insertion operation will introduce a distributed transaction involving all the Routers. It is more suitable for consistent lookup table to execute workload which contains a lot of reading queries and few updating queries.

Inconsistent Lookup Table: Inconsistent Lookup Table stores most of the correct Data Node IDs, and allows missing and incorrect lookup entries. When a query finds a missing lookup table key, a broadcast to all Data Nodes is used to get the correct result. Besides missing entries, there is another case to consider: the table is stale. When the Router receives the query, we cannot determine whether the lookup table entity is up to date or stale. The query may be sent to a partition that used to store the tuple but now deleted or moved. The Router will fall back to broadcast the query, as dealing with the missing entry. This is because when inserting or updating a tuple, only the Router handling the query updates its lookup table entries, thus the other Routers lacking such information would result in missing or stale lookup table entries.

Comparing with consistent lookup table, inconsistent lookup table avoids unnecessary distributed transactions when dealing with update or insert because of the allowance of incorrect lookup table entity. Thus, inconsistent lookup table can handle workloads with frequent updating more efficiently than consistent lookup table. However, when Router is executing queries involving missing or stale lookup table entries, broadcast will occur for the inconsistent state.

[11] proposes a lookup table framework based on Inconsistent Lookup Table with some optimizations. [10, 9, 8] use the similar implementation of [11]. These works use inconsistent lookup table which is inefficient when accessing stale lookup table entities. Besides, these lookup table frameworks are tightly coupling with the proposed prototype systems that cannot be migrated to other systems easily.

In this paper, we develop a novel prototype system, LuTe , to provide fine-grained partitioning for distributed systems with a novel **semi-consistency lookup table**. Semi-consistency lookup table integrates the advantages of Consistent lookup table and Inconsistent lookup table. It uses the temporal Router to maintain the correct result for the stale lookup table entities and update the stale ones when system is idle. LuTe can be used as an independent middleware which can be migrated to other systems with transaction support and high concurrency writing with MVCC model.

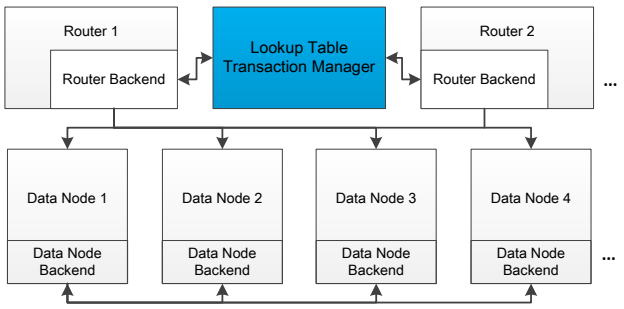


Figure 1: Architecture of LuTe on a Distributed DBMS

3. LUTE FRAMEWORK

In this section, we describe the overview of our system design and present the semi-consistent lookup table.

3.1 Overall

Figure 1 shows the architecture of LuTe implemented on a distributed DBMS.

LuTe components: LuTe is composed of three major components: LTTM (Lookup Table Transaction Manager), RB (Router Backend) and DNB (Data Node Backend) as shown in Figure 1. LTTM is a key component of LuTe to provide consistent transaction management and MVCC control. Router Backend is the interface for distributed systems to access the lookup service and each RB stores a copy of the whole lookup table entities. We place RB on the same node of the distributed system Router for local accessing. Data Node Backend is placed on Data Node which stores real data and executes sub query generated from Router. DNB can temporarily store lookup entity as a lookup proxy for RB which will be discussed below.

Distributed DBMS components: The Router is the interface for user to execute query on the DBMS. When a query comes, the Router parses the query to get the tuples used in this query. Then it checks the tuples from Local RB and rewrites the query to Data Node(s) involved in the query. After that, Data Node(s) will execute the query received from Router and return the result to Router for merging.

3.2 Semi-consistent Lookup Table

For consistent lookup table, lookup table in every Router is consistent all along, providing a strong guarantee that the location pointer is always correct. However, distributed transactions will occur when updating the other Routers' lookup tables. For inconsistent lookup table, the update is free of transactions. A Router executing a query that moves a certain tuple will not affect the other Routers immediately. Lookup tables on the other Routers stay unchanged until a tuple cannot be found using the local lookup table. Thus, when a query uses the missing or stale lookup table entities, a broadcast will be issued to correct the entities and to find the desired tuple.

To integrate the advantages of the above two approaches and conquer the weakness, we propose a novel approach named semi-consistent lookup table. The motivation is that in most cases, the insertion or movement of a tuple to certain Data Node does not require to be done immediately. The movement or insertion to the target Data Node can be delayed until system is not busy which we called **delayed update**. Thus when inserting new tuple, we can place the tuple on the Data Node by hash result without updating other Routers immediately until a proper time to tell Routers modify their lookup table entities. For example, when the client application issues a request to insert a $Tuple_t$ to $DataNode_N$, our approach operates in two steps. First, our approach simply stores $Tuple_t$ on the $DataNode_M$ which is computed by the hash function Hash($Tuple_t$)=M. Then, move $Tuple_t$ from $DataNode_M$ to $DataNode_N$ until the system and both $datenode_N$ and $DataNode_M$ are idle. During the moving necessary updates on the route's lookup tables is performed as well. In our experiment, we use the running transaction count as the threshold to determine whether the system is busy. When $Tuple_t$ is accessed between the time stamps of insertion to $DataNode_M$ and movement to $DataNode_N$, DNB will act as a temporary lookup table that takes charge of $Tuple_t$. In this way, all Routers could always find $Tuple_t$, using the hash function before the second step, or using lookup tables after the second step and access $Tuple_t$ with updated lookup table after second step. The insert operation is shown in Figure 2.

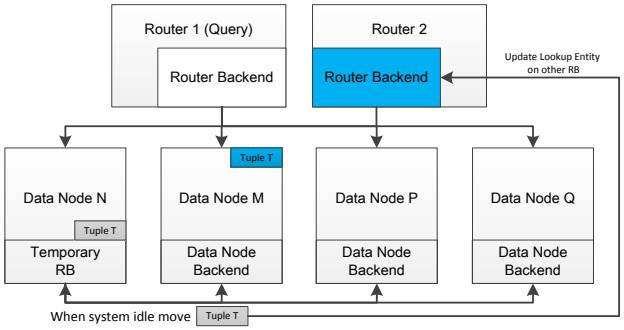


Figure 2: Tuple Insert on Semi-consistent Lookup Table

The same method is applicable to situations of tuple update as shown in Figure 3. Suppose that a $Tuple_t$ is stored at $DataNode_P$ which is recorded in Routers' lookup tables. When $Tuple_t$ is moved to $DataNode_Q$, we use the original $DataNode_P$ as a temporal Router, which records the correct Data Node holding $Tuple_t$. Therefore, $DataNode_P$ could act as a proxy and direct requests for $Tuple_t$ from the other Routers to the right node $DataNode_Q$, even if they have not updated their lookup tables. When the system is idle, $DataNode_P$ issues a broadcast and tells the other Routers to update their lookup tables. In this way, all the Routers contain the correct lookup table entities for all the existing tuples without distributed transaction when insert or move tuple. The lookup table entities on other RB will be updated when the system is idle.

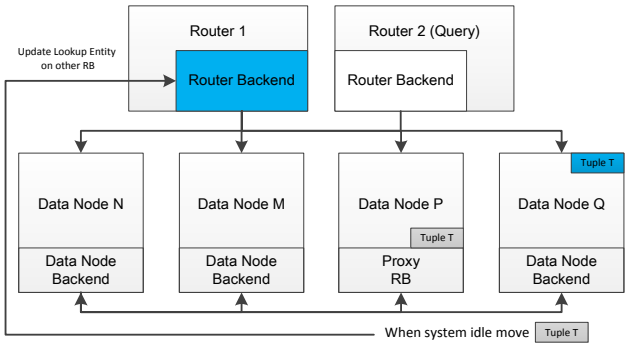


Figure 3: Tuple Update on Semi-consistent Lookup Table

3.3 Transaction and MVCC

LTTM is a key component for LuTe to provide consistent transaction management and MVCC control. When a Router updates lookup table entities, the update is executed in a transaction with a given unique transaction ID from LTTM. The transaction ID is in ascending order to distinguish the execution order. Besides, every lookup table entity has a transaction ID pair min_t and max_t that determine the visibility for certain transaction. The min_t is set when the lookup table entity is inserted into LuTe , and max_t is set when it is deleted. Thus, lookup table entities is visible for the transaction with transaction ID in range (min_t, max_t). In fact, to provide high concurrency writing,

update or delete only creates a new lookup entity and does not really remove the old one. To remove the lookup entity that will be not visible to any future transactions, LuTe conducts an auto-removing processing on each RB and DNB at a configurable frequency.

3.4 Storage

Lookup table needs to be frequently accessed in distributed system processing. To avoid performance penalty, it is stored in main memory on each Router. Lookup table maps a tuple to a list of partitions where the tuple is stored. We can use an N-bit number to present the partitions that stores the key where N is the number of partitions. In our implementation, we further use bloom filter to reduce the memory cost, which provides a more compact representation though it has the disadvantage of producing some false positives. A false positive means that a query may access some partitions which do not contain the targeting tuples. These false positives may decrease performance, but will not affect the correctness. Besides, we can control the false positive rate low enough with high compact rate.

4. EVALUATION

In this section, we present the performance of our lookup table approach. We firstly introduce our experimental setup in subsection 4.1. Then we provide partitioning scheme of different partitioning approaches in subsection 4.2. At last, we show the simulation query and system throughput in subsection 4.3 and subsection 4.4.

4.1 Experiment Setup

To evaluate LuTe , we built a distributed DBMS by extending a popular open source system Postgresql [1] and implemented LuTe together with hash(H), consistent lookup table(C) and inconsistent lookup table(IC) on this prototype system. We conducted our experiment in a cluster of 10 commodity machines with AMD Opteron 4180 2.6Ghz CPU, 48GB memory and a 50GB disk. Due to the space limit, here we mainly present the result with two Routers and eight Data Nodes. The experimental results on the other data sets show the similar performance improvement. We used TPC-C benchmark for our experimental evaluation which contains 9 tables and 5 types of queries which contain about 48% writing (update and insert) and 47% reading. In the experiment, graph of the tuples was horizontally partitioned into each Data Node according to hash or fine-gained partitioning mentioned in [3]. In addition, to evaluate the performance with different types of reading/writing ratio, we used a synthetic dataset containing 5 million tuples and 0.35 million transactions with different reading/writing ratio. The detailed configuration of five workload refers to table 1.

4.2 Partitioning Scheme

In our experiments, we compared the performance of hash partitioning and fine-grained partitioning that was supported by three kinds of lookup table. Hash partitioning: We used tuple-level hash partitioning which places the tuple according to its primary key using a hash function. When the query involved the attribute which is not primary key, broadcast will occur for processing the query. Fine-gained partitioning: we used tuple-level fine-gained partitioning algorithm mentioned in [3] to partition the dataset. The partitioning was based on a tuple-level graph where each edge represents a transaction and nodes spanned by the edge represent tuples in the same transaction from workload log. Due to the space limit, the fine-gained partitioning algorithm will not be further discussed.

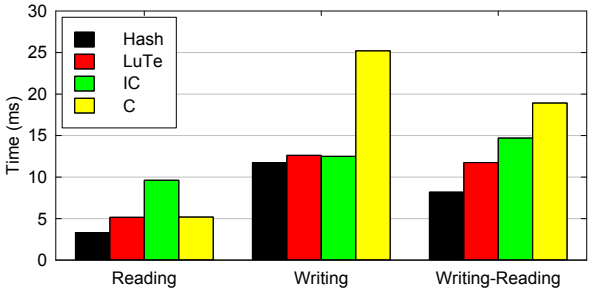


Figure 4: Simulation Query on Different Lookup Tables

4.3 Simulation Query

We first measure the lookup table efficiency when dealing with simulation query which is one tuple insert (writing) or one tuple select (reading) on cluster with 2 Routers and 2 Data Nodes. The experiment uses a table which contains only one attribute with no tuple at the beginning of the experiment. Firstly, we execute 3,000 transactions that insert a tuple with random value into the table. Then we randomly read one of these 3,000 tuples. At last, we mix read and write transactions (each 50%). We compare the three types of lookup table together with hash on these three types of workloads. Figure 4 shows the comparison of average processing time. As we can see, for writing transaction, consistent lookup table shows poor performance since it must update lookup table entity on the other Routers for insert operation with inefficient two phase commit. The result also indicates that for reading intensive transaction, inconsistent lookup table needs more time to process because the Routers may not have lookup table entity for some reading tuples and have to broadcast the query to all the Data Nodes. On the contrary, LuTe integrates the advantages and conquers the weakness, it takes almost the same processing time comparing with hash.

Table 1: Workload Statistics

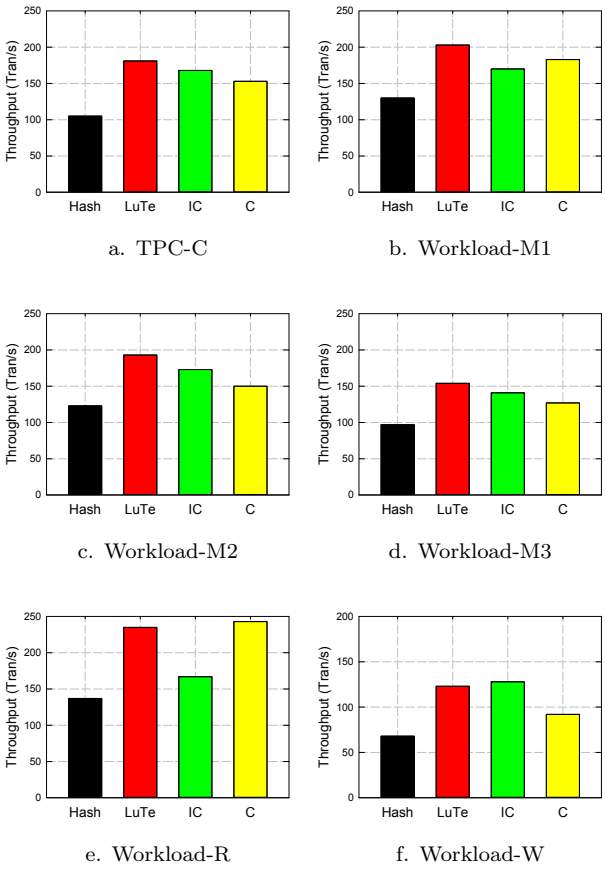
Workload	Read Ratio	Write Ratio
Ratio-M1	75%	25%
Ratio-M2	50%	50%
Ratio-M3	25%	75%
Ratio-R	100%	0%
Ratio-W	0%	100%

4.4 System Throughput

In order to evaluate the throughput of our approach, we partition the TPC-C tables by fine-grained partitioning algorithm [3] to 8 partitions and evaluate LuTe 's throughput comparing other two lookup table implementations and hash partitioning scheme. To ascertain the suitability of different types of workloads, we use workloads Workload-M1, Workload-M2, Workload-M3, Workload-W, Workload-

R and original TPC-C workload which contain different proportions of reading and writing ratio. The Workload-W workload executes inserting only transaction while Workload-R workload is reading only. Proportions of other workloads are shown in table 1. All the experiments use two Routers each of which takes charge half of the queries.

As shown in Figure 5, there is a substantial reduction of throughout for hash partitioning compared with fine-grained partitioning since fine-grained partitioning approaches reduce the distributed transactions with better tuple placement. LuTe beats inconsistent and consistent lookup table almost on every workload with about 17% better throughput which is attributed to the combination of the advantage of the two implementations. Besides, it scores a 57% to 70% better throughput than simple hash partitioning.



a. TPC-C b. Workload-M1

c. Workload-M2 d. Workload-M3

e. Workload-R f. Workload-W

Figure 5: Throughput on Different Workloads

The above evaluation results confirm that LuTe can provide highly efficient and transaction supported fine-grained partitioning support for distributed systems and improve the throughput of these systems.

5. CONCLUSIONS

In this paper, we systematically investigated lookup table implementation in current distributed systems. We proposed a novel semi-lookup table technology and designed a fine-grained partitioning framework LuTe base on it. Prototype and experimental results confirmed the improvements of our new lookup table approaches.

There are several promising directions for our future work. First, a further theoretical analysis of partitioning algorithm on different workloads is valuable. Second, experiments on distributed data processing system with LuTe for partitioning are interesting as well. At last, how to use LuTe for distributed index support is another open problem.

6. ACKNOWLEDGMENTS

This research was supported by the National Natural Science foundation of China under Grant No. 61272155.

7. REFERENCES

[1] Postgresql. http://www.postgresql.org/.

[2] Tpc-c benchmark. http://www.tpc.org/tpcc/.

[3] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. Schism: a workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment*, 3(1-2):48–57, 2010.

[4] H. Gu D. Bickson J. Gonzalez, Y. Low and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.

[5] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *Proc. of WWW*, pages 591–600.

[6] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. volume 5, pages 716–727. VLDB Endowment, 2012.

[7] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proc. of SIGMOD*, pages 135–146, 2010.

[8] Josep M Pujol, Vijay Erramilli, Georgos Siganos, Xiaoyuan Yang, Nikos Laoutaris, Parminder Chhabra, and Pablo Rodriguez. The little engine (s) that could: scaling online social networks. In *ACM SIGCOMM Computer Communication Review*, volume 40, pages 375–386. ACM, 2010.

[9] Abdul Quamar, K Ashwin Kumar, and Amol Deshpande. Sword: scalable workload-aware data placement for transactional workloads. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 430–441. ACM, 2013.

[10] Zechao Shang and Jeffrey Xu Yu. Catch the wind: Graph workload balancing on cloud. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 553–564, 2013.

[11] Aubrey L Tatarowicz, Carlo Curino, Evan PC Jones, and Sam Madden. Lookup tables: Fine-grained partitioning for distributed databases. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 102–113. IEEE, 2012.