DynaHash: Efficient Data Rebalancing in Apache AsterixDB

Chen Luo
University of California, Irvine
cluo8@uci.edu

Michael J. Carey
University of California, Irvine
mjcarey@ics.uci.edu

ABSTRACT

Parallel shared-nothing data management systems have been widely used to exploit a cluster of machines for efficient and scalable data processing. When a cluster needs to be dynamically scaled in or out, data must be efficiently rebalanced. Ideally, data rebalancing should have a low data movement cost, incur a small overhead on data ingestion and query processing, and be performed online without blocking reads or writes. However, existing parallel data management systems often exhibit certain limitations and drawbacks in terms of efficient data rebalancing.

In this paper, we introduce DynaHash, an efficient data rebalancing approach that combines *dyna*mic bucketing with extendible *hash*ing for shared-nothing OLAP-style parallel data management systems. DynaHash dynamically partitions the records into a number of buckets using extendible hashing to achieve good a load balance with small rebalancing costs. We further describe an end-to-end implementation of the proposed approach inside an open-source Big Data Management System (BDMS), Apache AsterixDB. Our implementation exploits the out-of-place update design of LSM-trees to efficiently rebalance data without blocking concurrent reads and writes. Finally, we have conducted performance experiments using the TPC-H benchmark and we present the results here.

PVLDB Reference Format:
Chen Luo and Michael J. Carey. DynaHash: Efficient Data Rebalancing in Apache AsterixDB. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

1 INTRODUCTION

The coming end of Moore's law and the information age have led data management systems to exploit clusters of machines to process large amounts of data growing in an unprecedented speed. As a result, parallel shared-nothing data management systems have become widely used today due to their high scalability. In a parallel shared-nothing data management system, records are partitioned across a cluster of nodes that communicate with each other via an interconnection network [32]. The shared-nothing parallel architecture enables these systems to be horizontally scaled as the number of nodes increases.

Early parallel data management systems [20, 33] generally assumed that the cluster of nodes is relatively static. However, this assumption is no longer true. It is desirable to dynamically adjust the cluster size for a number of reasons. For example, it is economical to dynamically scale the cluster in and out as the workload changes, especially in the era of cloud computing. Moreover, as the stored data accumulates over time, the cluster also needs to be scaled out to better serve the query workloads. In order to scale a cluster in or out, the stored records must be rebalanced so that they can be repartitioned to the new set of nodes. Ideally, rebalance operations should result in a near-perfect load balance, a low overhead for regular database operations, and a small data movement cost. Moreover, rebalancing must be performed online so that reads and writes are not blocked.

In this paper, we focus on data rebalancing for shared-nothing parallel data management systems for analytical (OLAP) workloads. Even though many parallel data management systems today have implemented various data rebalancing functionalities, the existing implementations often exhibit certain limitations or drawbacks for OLAP data management. Data management systems that support OLTP workloads [2, 4, 25, 44, 73, 81] generally perform fine-grained range partitioning to enable efficient data rebalancing. However, this is not suitable for OLAP workloads due to the potential query load imbalance caused by range skews. Existing parallel OLAP systems either rely on shared-data architectures for data rebalancing [26], incur a large data movement cost [49], or block writes during rebalancing [43].

Our Contributions. In this paper, we present DynaHash, an efficient data rebalancing approach for OLAP-style parallel data management system with local secondary indexes. The basic idea of DynaHash is to dynamically partition the records into a set of buckets using extendible hashing [39] and to move buckets for efficient rebalancing. By combining extendible hashing with dynamic bucketing, DynaHash can greatly reduce the data movement cost with a minimal impact on data ingestion and query processing.

As the second contribution, we describe an efficient rebalancing implementation that avoids blocking concurrent reads and writes by exploiting the out-of-place design offered by LSM-trees [68]. The techniques used include bucketed LSM storage, lazy secondary index cleanup, concurrency control for online rebalancing, and an effective approach to fault tolerance and recovery. Even though some similar techniques have been implemented by other systems, our contribution here is to show how to integrate them together to enable an efficient and effective rebalancing implementation.

As the last contribution, we have implemented all of the proposed techniques inside Apache AsterixDB [1]. We have carried out extensive experiments on the TPC-H benchmark [7] to evaluate the effectiveness of the proposed techniques. The experimental results show that the proposed rebalancing approach DynaHash significantly reduces the rebalance cost with a small overhead on query and ingestion performance. It should be noted that even though our approach has been implemented for an LSM-based row store, the design itself can be naturally generalized to column store-based systems since these systems have generally adopted the same out-of-place update design for their data [40, 49, 78].

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

The remainder of this paper is organized as follows. Section 2 discusses background information and related work. Section 3 presents an overview of our proposed rebalancing approach. Section 4 describes how to store buckets efficiently on a single node. Section 5 presents the detailed design and implementation of the rebalance operation. Section 6 experimentally evaluates the proposed techniques. Finally, Section 7 concludes the paper.

2 BACKGROUND

2.1 Data Rebalancing

To exploit the parallelism provided by a cluster of nodes, the records of a dataset must be distributed to each node using a partitioning function. A partitioning function deterministically assigns each record to a node based on its partitioning key. Example partitioning functions include range partitioning and hash partitioning[1]. Range partitioning divides the key space into a set of ranges, each of which is assigned to a node. In contrast, hash partitioning operates on the hashed keys to achieve better a load balance.

When the cluster needs to be scaled in or out, its datasets must be repartitioned through a rebalance process. In general, rebalancing has three important trade-offs, i.e., the load balance, the rebalance cost, and the normal operation overhead. The load balance measures how evenly the data is distributed across different nodes. This directly impacts query performance, as in a shared-nothing system the query time is bottlenecked by the slowest node. The rebalance cost measures how much of the data needs to be accessed and moved during rebalancing. Finally, the normal operation overhead measures the extra overhead for normal read and write operations in order to support the needs of the rebalance operation.

Rebalancing changes the partitioning function. Depending on how the partitioning function changes, existing rebalancing schemes can be classified as either *global* or *local*. A global rebalancing scheme repartitions (nearly) all records of a dataset when the cluster changes. This generally leads to a near-perfect load balance and a small normal operation overhead but a very high rebalance cost. For example, with range partitioning, a global rebalancing scheme can recompute the key range of each node based on the new cluster size and then repartition all records based on the new ranges.

In contrast to global rebalancing, a local rebalancing scheme only changes the partitioning function "locally" so that only a small portion of the records, generally proportional to the affected nodes, are moved. This reduces the rebalancing cost, but generally leads to a worse load balance and a higher normal operation overhead. Commonly used local rebalancing schemes include *static bucketing*, *dynamic bucketing*, and *consistent hashing* [45][2]. In static bucketing, the key space is pre-partitioned to a fixed number of buckets, each of which is assigned to a node through a directory. During rebalancing, only a small number of affected buckets are moved to new nodes, which significantly reduces the rebalance cost. Dynamic bucketing further extends the usability of static bucketing by dynamically splitting or merging buckets as the dataset size grows or shrinks. Finally, consistent hashing eliminates the overhead of the global directory by organizing the (hashed) key space into a ring structure and letting each node serve a key range. When a node is added or removed, its key range is adjusted locally based on its next neighbor node. In general, consistent hashing is more suitable for a (large) peer-to-peer architecture since it does not require a global directory. In contrast, dynamic bucketing works naturally with a more centralized (master-slave) architecture where the bucket assignment information is managed by the master. Moreover, a global directory also provides more flexibility for bucket assignment.

[1]There could be other partitioning functions in practice, such as round-robin partitioning and random partitioning. However, we do not consider them here because those partitioning functions are not deterministic.

[2]The range partitioning counterpart of consistent hashing is rarely used in practice because of the potential for range skews. Thus, that scheme is not considered here.

2.2 Log-Structured Merge Trees

The LSM-tree [68] is a persistent index structure optimized for write-intensive workloads. The LSM-tree adopts an out-of-place update design by always buffering writes into a memory component and appending records to a transaction log for durability. Whenever the memory component is full, writes are flushed to disk to form an immutable disk component. Multiple disk components are periodically merged together to form a larger one, according a pre-defined merge policy.

A query over an LSM-tree has to reconcile the entries with identical keys from multiple components, as entries from newer components override those from older components. A range query searches all components simultaneously using a priority queue to perform reconciliation. A point lookup query simply searches all components from newest to oldest until the first match is found. To speed up point lookups, a common optimization is to build Bloom filters [19] over the sets of keys stored in disk components.

2.3 Apache AsterixDB

Apache AsterixDB [1, 13, 24] is an open-source Big Data Management System (BDMS) that aims to manage massive amounts of semi-structured (e.g., JSON) data efficiently. AsterixDB uses a shared-nothing parallel architecture with local secondary indexes for OLAP-style workloads. An AsterixDB cluster contains a Cluster Controller (CC) that serves as the master and multiple Node Controllers (NCs) that perform data processing tasks. Each NC has multiple partitions to exploit the parallelism of modern hardware. A query in AsterixDB is compiled and optimized by the CC into a Hyracks job [22] that is then executed by the NCs. To support efficient data ingestion, AsterixDB provides data feeds [42], which are long-running jobs that efficiently ingest external data into AsterixDB.

The records of a dataset are hash-partitioned based on their primary keys across multiple NC partitions. Each dataset partition is managed by an LSM-based storage engine [14], including a primary index, a primary key index, and multiple local secondary indexes. The primary index stores records indexed by primary keys, and the primary key index stores primary keys only. The primary key index is built to support COUNT(*) style queries and uniqueness checks efficiently [58] since it is much smaller than the primary index. Secondary indexes use the composition of the secondary key and the primary key as their index keys. AsterixDB supports LSM-based B^+-trees, R-trees, and inverted indexes using a generic LSM-ification framework that can convert an in-place index into an LSM-based index. Each LSM-tree uses a tiering-like merge policy

to merge its disk components. AsterixDB uses a record-level transaction model to ensure that all of the indexes are kept consistent within each partition.

AsterixDB uses a global rebalancing scheme with hash partitioning. Given a cluster with N partitions, AsterixDB assigns each record with key K to the $hash(K)$ mod N partition. When the cluster size changes, the partitioning function is recomputed so that the records of a dataset are redistributed to the new set of nodes. This approach leads to a near-perfect load balance with a minimum normal operation overhead, but the rebalance cost is very high since nearly all records need to be moved during rebalancing. In this work, we explore alternative data rebalancing schemes to make better trade-offs among these three costs.

2.4 Related Work

Rebalancing in Parallel Data Management Systems. Nearly all parallel data management systems today have implemented some form of rebalancing. Here we discuss rebalancing in some representative systems based on the taxonomy of Section 2.1.

For OLTP-style systems, Bigtable [25] and its open-source cousin HBase [2] use dynamic bucketing with a shared-data architecture. Since their underlying distributed storage systems, GFS [41] for Bigtable and HDFS [3] for HBase, already support rebalancing immutable data blocks, Bigtable and HBase only need to manage their in-memory data during rebalancing. Dynamo [31] and its open-source cousin Cassandra [48] are shared-nothing systems that use consistent hashing. Cassandra further introduces the concept of virtual nodes to achieve a better load balance; the basic idea is to let each node use multiple virtual nodes to manage multiple key ranges. Couchbase [21] and Oracle NoSQL Database [5] are shared-nothing systems that use static bucketing with hash partitioning. Both systems set the number of buckets to a relatively high number. Couchbase sets this number to 1024 by default, while Oracle NoSQL Database recommends that each node (in the expected largest cluster) should have 10 to 20 buckets. Minhas et al. [67] applied a similar static bucketing approach to enable efficient scaling for VoltDB [8]. MongoDB [4], TiDB [44], WattDB [73], and CockroachDB [81] each use range-partitioned dynamic bucketing with a very small bucket size, e.g., 64MB. Having a large number of small buckets is suitable for OLTP workloads since each transaction only accesses a small number of (usually one) buckets. However, this may not be suitable for OLAP systems since each query will often access all buckets. Moreover, OLTP systems typically use global secondary indexes due to the high selectivity of OLTP queries.

For OLAP-style systems, Snowflake [26] is based on a shared-data architecture and completely relies on the underlying shared storage system for rebalancing. Vertica [49] is a shared-nothing system that uses global rebalancing with hash partitioning to achieve a better load balance. By performing range partitioning on hashed keys and carefully placing the new nodes into the cluster, Vertica can reduce the rebalance cost by a constant factor [49]. Redshift [43] is shared-nothing and supports both global rebalancing and static bucketing with hash partitioning. However, Redshift does not support concurrent writes during rebalancing. Moreover, it directly uses buckets (called "node slices" in Redshift) as its parallelism unit. This leads to an undesirable side-affect that the node parallelism changes after rebalancing[3]. NashDB [65] adopts an economics framework to automatically distribute data based on user-provided query priorities. However, NashDB targets static read-only workloads and does not consider the rebalancing cost.

Elastic OLTP Databases. Due to the importance and wide adoption of parallel OLTP database systems, a lot of effort has been devoted to making them elastic. Live migration techniques [17, 27, 37, 38, 52, 74] enable OLTP databases to be migrated without blocking ongoing transactions. E-Store [80] uses fine-grained partitioning to elastically scale parallel databases. Morphosys [9], Accordion [76], and Clay [77] perform online database partitioning to reduce the cost of distributed transactions. These research efforts all share some similarity with our work by considering online database rebalancing without blocking concurrent transactions. However, one key difference is that these research efforts mainly focus on ACID transactions, while our work focuses on how to rebalance datasets efficiently in OLAP-style (i.e., query-oriented) systems.

Distributed Access Methods. To efficiently query data stored in a cluster of nodes, a number of distributed access methods have been proposed as well. The basic idea is to distribute an access method efficiently over a cluster of nodes, potentially in a peer-to-peer setting, to support efficient read and write operations. Examples include distributed versions of extendible hashing [36], linear hashing [53, 55], range search trees [47, 54], B^+-trees [10], and R-trees [34, 35]. A key difference between these access methods and our work is that we focus on rebalancing for OLAP systems rather than on a single access method with simple key-value interfaces. Moreover, these access method proposals have rarely been used by today's parallel data management systems due to their increased complexity. Instead, modern systems generally employ a simple partitioning approach that partitions datasets into multiple nodes.

LSM-trees. For data storage in modern systems, a large number of improvements have been proposed to optimize the LSM-tree [68]. These improvements include optimizing write performance [18, 50, 56, 64, 66, 70, 79], supporting auto-tuning of LSM-trees [28–30, 51, 72], optimizing query performance of LSM-trees [12, 58, 63, 69], minimizing write stalls [15, 59, 75], exploiting large memory [16, 23, 57, 60], and extending the applicability of LSM-trees [62, 71]. We refer readers to a recent survey [61] for a more detailed description of these LSM-tree improvements. These LSM-tree improvements have all focused on a single node setting. In contrast, in this work, we focus on their role in a parallel shared-nothing architecture and exploit the LSM-tree's out-of-place design to support efficient data rebalancing with concurrent reads and writes.

3 APPROACH OVERVIEW

As mentioned in Section 2.1, rebalancing involves three important trade-offs, i.e., the load balance, the rebalance cost, and the normal operation overhead. Our goal is to achieve good load balance with a small rebalance cost and a low normal operation overhead. In this section, we provide a high-level overview of DynaHash based on the following design choices.

[3] For example, consider a cluster with 4 nodes. Each node further has 4 node slices to exploit the node parallelism. However, if the cluster is resized to 16 nodes, each node will only have one node slice, which may negatively impact the parallelism of query processing.

Range Partitioning vs. Hash Partitioning. In general, range partitioning is more suitable for OLTP systems since each transaction only accesses a few partitions. It also provides fine-grained partitioning capabilities for the system to eliminate hot ranges. In contrast, most OLAP systems prefer hash partitioning to achieve a better load balance since many queries will access all partitions. Thus, we choose to use hash partitioning here as well.

Global Rebalancing vs. Local Rebalancing. Although global rebalancing schemes achieve a near-perfect load balance, they incur a very large rebalance cost since most records have to be moved during rebalancing. Since our goal is to reduce the rebalance cost, we prefer to use the local rebalancing scheme. Among the three local rebalancing schemes mentioned in Section 2.1, dynamic bucketing dominates static bucketing by elastically adjusting the number of buckets as data accumulates. Dynamic bucketing is also preferable to consistent hashing since most parallel OLAP systems (ours included) adopt a master-slave architecture. Based on these considerations, it is a natural choice to use dynamic bucketing here.

Combining Hash Partitioning with Dynamic Bucketing. The last choice we face is how to combine hash partitioning with dynamic bucketing. One natural design would be to range partition the hashed key space into multiple buckets. Though this solution works, hashing actually provides opportunities for a more efficient design. Since hashed keys are uniformly distributed, one can use an extendible hashing approach [39] to partition the key space into multiple buckets. Figure 1 shows the resulting architecture based on this idea with one Cluster Controller (CC) and two Node Controllers (NCs). Each NC further has two storage partitions. In order to distribute the records of a dataset to these four partitions, the hash key space is divided into multiple buckets. A bucket is defined by taking the d low-order bits of the hash function, where d is the depth of this bucket. When a bucket becomes too large, it is split into two smaller buckets by taking one more hash bit, which thus increments the depth [39]. A rebalance operation can now only move some affected buckets to new partitions, which can greatly reduce the rebalance cost.

As shown in Figure 1, we use a *global directory* stored at the CC to map buckets to partitions. Each directory has a depth D, which is the maximum number of bits used in all buckets. Thus, the size of this directory is always 2^D. Note that in Figure 1 the two hash values 011 and 111 currently correspond to the same bucket 11. To locate where a given key K is stored, one simply needs to look in the global directory using the D low-order bits of K's hash value, where D is the depth of the global directory. During query compilation, each query creates an immutable copy of the global directory that is used throughout query processing. Similarly, a data feed, i.e., a data ingestion job, also employs an immutable copy of the global directory in order to distribute the incoming records of a dataset to the correct NC partitions.

We further use a *local directory* at each partition to keep track of the assigned buckets. To simplify bucket splits, the global directory can be updated lazily before rebalancing is performed. For example, in Figure 1, the bucket 00 has already been split into two buckets 000 and 100 at partition 0, but the global directory has not been updated yet. This does not impact the correctness of the global directory since it can still correctly route all keys to the right partitions.

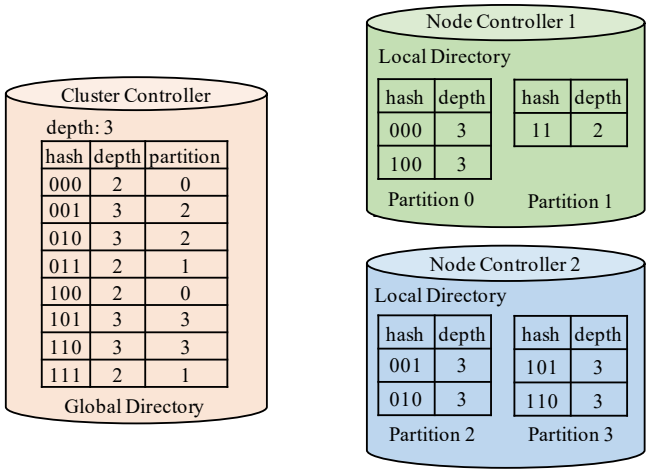


Figure 1: Example Architecture for DynaHash

Even though the basic design in our rebalancing approach is relatively straightforward, two key challenges must be addressed. First, how can we store (i.e., physical organize) multiple buckets within each partition to enable efficient rebalancing with low normal operation overheads for reads and writes? Second, how can we efficiently rebalance buckets while supporting both concurrent reads and writes? In the next two sections, we will detail our solutions to these two challenges.

4 LSM STORAGE FOR BUCKETS

In this section, we discuss how to efficiently store multiple buckets in each partition. For efficient rebalancing, when a bucket needs to be moved out of a partition, it is desirable to only access the records for this bucket. If range partitioning were used and no secondary indexes were built, storing records in their primary key order naturally satisfies this property since records in each bucket would be grouped together. However, with hash partitioning, the primary key order is no longer the same as the bucket order since records are bucketed using hashed keys. Moreover, secondary indexes also complicate this problem because their entries are ordered by secondary keys, not primary keys.

Storage Options. In general, when hash partitioning is used, there are three options to store buckets in each partition:

- Option 1: Store entries in their original key order in one LSM-tree index.
- Option 2: Store entries in their bucket order in one LSM-tree index. Within each bucket, store their entries in their original key order.
- Option 3: Store entries in each bucket in a separate LSM-tree index structure. Within each LSM-tree, store the entries in the original key order.

Let us first consider the trade-offs for the primary index. Option 1 incurs no overhead on reads and writes, but it incurs a large overhead on rebalancing since moving a bucket must scan all entries, including those from other buckets. Options 2 and 3 both reduce the rebalancing overhead since records within each bucket are stored together. Moreover, Option 3 provides more flexibility for splitting

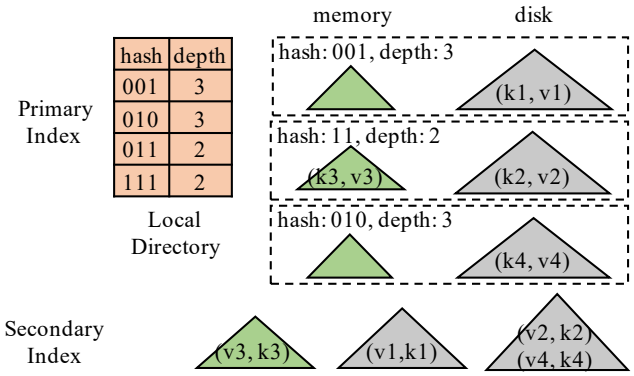


Figure 2: Example Partition with a Bucketed Primary Index and a Secondary Index

buckets and deleting buckets after rebalancing. However, Options 2 and 3 both incur some overhead on short primary index scans since each query must search all buckets. Since short primary key-order scans are not common in OLAP-style systems, we choose to optimize the rebalancing performance by choosing Option 3 for the primary index of a dataset. For secondary indexes, an important difference is that they do not have to be accessed during rebalancing but can be rebuilt on-the-fly at their destination. In order not to incur too much normal runtime overhead on secondary index queries, we choose to use Option 1 for secondary indexes. Figure 2 shows an example of a dataset partition with a primary index and a secondary index. Here we denote each record as a key-value pair, and the secondary index is built on the value field. The primary index uses the bucketed LSM-tree design, which is further described below, to store buckets separately. In contrast, the secondary index uses a traditional LSM-tree design to store all buckets together.

Bucketed LSM-tree Design. Based on these basic design decisions, we introduce a bucketed LSM-tree design for efficiently storing multiple buckets in the primary index. As shown in Figure 2, each bucket can be viewed as a separate LSM-tree with a memory component and multiple disk components. We use reference counting for concurrency handling. That is, whenever a bucket, a memory component, or a disk component is accessed, the reader or writer increments a reference count so that the accessed entity cannot be destroyed until the access completes. All flushes and merges are performed within each bucket. All buckets are coordinated using a local directory, as mentioned in Section 3. Note that in Figure 2, hashes 011 and 111 correspond to the same bucket 11 with depth 2.

Data Ingestion and Query Processing. A bucketed LSM-tree provides the same set of interfaces as a traditional LSM-tree. A write operation, including inserts, deletes, and updates, first checks the local directory using the hash value of the key to locate which bucket the entry belongs to and then adds the entry to that bucket. Similarly, a point lookup query only searches its target bucket, located via the local directory, to get the entry. A primary key range scan query, however, must search all buckets. There are two approaches to process such a range scan query. The first approach is to scan each bucket separately. This will incur no additional overhead compared to the traditional LSM-tree design, but the returned entries

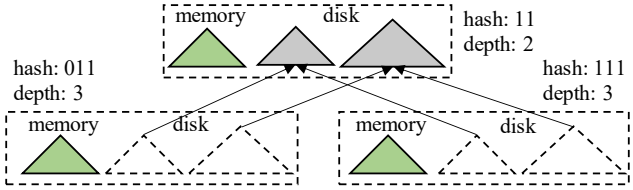


Figure 3: Bucket Split Example

will no longer be sorted on the primary key. The second approach is to use a priority queue to merge-sort the returned entries from all buckets together. This approach provides the same interface as the traditional LSM-tree design by returning sorted results, but it will incur a larger search overhead due to the additional merge-sort step. To decide which approach should be used, we have introduced an optimization rule in AsterixDB as follows. By default, the first approach is used to avoid the merge-sort overhead. However, if the primary key order is required by subsequent query operators, e.g., a user-specified order by clause or a groupby operator on a prefix of the primary key, the second approach will be used to avoid the subsequent sort overhead. Finally, it should be noted that the bucketed LSM-tree design does not change the processing of a secondary index query, which simply searches the secondary index to fetch a list of primary keys and then uses them to fetch records from the (bucketed) primary index.

Efficient Bucket Splits. When a bucket becomes too large, it is split into two smaller buckets by using one more hash bit. A straightforward implementation would be to build two smaller LSM-trees based the original bucket. However, this approach not only causes additional write amplification, but also may need to block reads and writes for a long time. Here we describe a more efficient bucket splitting approach to address these issues.

The pseudocode for splitting a bucket B is depicted in Algorithm 1. The Split function first stops creating new component merges for B and waits for all existing merges to finish. B's memory component is then asynchronously flushed to disk without blocking writes (line 5). After the flush completes, the bucket B is locked to temporarily block new readers and writers so that B can be safely split (lines 6 to 10). Since some writes may have entered the memory component after the last asynchronous flush, B's memory component is now flushed synchronously to persist these writes. It should be noted that AsterixDB uses a no-steal buffer management policy, meaning that a memory component is only flushed after all active writers have completed. Two new buckets B_1 and B_2, whose disk components refer to the disk components of B, are then created. An example is shown in Figure 3. For each disk component of the splitting bucket 11, we create two new *reference disk components* in buckets 011 and 111 respectively. A reference disk component does not store any data; instead, it only points to a real disk component. All queries accessing data through a reference disk component must perform an additional filtering step based on the bucket's hash value to make sure that only the entries belonging to this bucket are accessed. Thus, the actual creation of the new disk components of B_1 and B_2 are effectively postponed until the next round of merges. Finally, a directory metadata file that stores valid buckets is forced to disk, indicating that the split

operation is now complete (line 9), and the old bucket B is destroyed automatically when its reference count becomes 0. Upon recovery, the directory metadata file is used to determine valid buckets. All invalid (partially split) buckets will be cleaned up automatically.

Algorithm 1 Pseudo Code for Bucket Split

```
1:  B ← the bucket to be split
2:  function SPLIT(B)
3:      Pause scheduling merges for B
4:      Wait for B's merges to finish
5:      Asynchronously flush B's memory component
6:      Lock B
7:          Synchronously flush B's memory component
8:          Create two buckets B1 and B2 that refer to B
9:          Force a directory metadata file to disk
10:     Unlock B
11:     Resume scheduling merges for B
```

To simplify the synchronization with the CC, bucket splits are performed at each partition locally without notifying the CC. Instead, the global directory at the CC is only refreshed when a rebalance operation starts, as we will discuss below. This design greatly simplifies the role of the CC since it does not have to know about the existence of bucket splits.

5 EFFICIENT DATA REBALANCING

After considering how to store multiple buckets efficiently, we now discuss how to efficiently rebalance data while supporting concurrent reads and writes. In AsterixDB, data rebalancing is triggered by the user manually after some nodes have been added or before some nodes are removed. In general, a rebalance operation contains three phases, namely initialization, data movement, and finalization. During the initialization phase, all nodes perform some preparation tasks for subsequent data movement. The data movement phase transfers some of the records of a dataset, including concurrent writes, to their new partitions. Finally, during the finalization phase, all nodes unanimously commit or abort the rebalance operation depending on its outcome and some cleanup work is performed as well. It should be noted that a rebalance operation may fail for various reasons. When a rebalance operation fails, the produced intermediate results must be cleaned up correctly. In the remainder of this section, we discuss the three phases in detail as well as how to handle various rebalance failures.

5.1 Initialization Phase

When a rebalance operation starts, the CC first forces a BEGIN log record indicating that a rebalance operation has started. This is required for correctly handling rebalance failures, as we will see in Section 5.4. The CC further decides which buckets should be moved to which partitions by computing a new global directory based on the new set of nodes. In addition, all NCs must also perform some preparation tasks in order to support concurrent updates. The key challenge here is that AsterixDB only supports a very simple record-level transaction model. If full ACID transactions were supported by AsterixDB, then the rebalance operation could be simply implemented using a transaction, which would automatically provide concurrency control for reads and writes. Without ACID transactional support, we must design a customized concurrency control protocol.

Computing the Global Directory. Recall from Section 4 that buckets splits are performed at each node locally without notifying the CC. In order to compute the new global directory, the CC contacts all NCs to get their latest local directories. Moreover, bucket splits for this dataset at each NC are disabled until the rebalance completes. Since buckets may have different sizes, it is straightforward to show that an optimal algorithm that maximizes the load balance is NP-hard by considering the partition problem[4].

Algorithm 2 Pseudo Code for Computing New Global Directory

```
1:  function BALANCE
2:      for each unassigned bucket B do
3:          Assign B to the least loaded partition
4:      while true do
5:          P_max ← the most loaded partition
6:          B ← the smallest bucket in P_max
7:          P_min ← the least loaded partition
8:          if abs((|P_max|−|B|)−(|P_min|+|B|)) < |P_max|−|P_min|
    then
9:              Assign B to P_min
10:         else
11:             break
```

To compute the new global directory efficiently, we use a greedy algorithm as shown in Algorithm 2. To describe this algorithm, we first introduce some useful concepts. Given a directory with depth D and a bucket B with depth d, we define the *normalized size* of the bucket B (denoted as $|B|$) as 2^{D-d}. Given a partition P or a node N, we denote $|P|$ or $|N|$ as the sum of the normalized size of P's buckets and N's buckets, respectively. Given two partitions, P_1 on node N_1 and P_2 on node N_2, P_1 is said to be more loaded than if $|P_1|$ is larger than $|P_2|$, or $|N_1|$ is larger than $|N_2|$ if $|P_1|$ equals $|P_2|$. The BALANCE function first assigns the unassigned buckets (buckets being displaced due to node removals) to the least loaded partitions (lines 2-4). After all such buckets are assigned, the algorithm balances the bucket assignment using a series of iterations (lines 4-11). In each iteration, it tries to assign the smallest bucket B from the most loaded partition P_{max} to the least loaded partition P_{min} (lines 8-11). If this assignment reduces the difference between the normalized sizes of P_{max} and P_{min}, the assignment is then performed; otherwise, the algorithm terminates. It is possible to incorporate other heuristics to define the load order among partitions. For example, one could further consider the total storage size of a partition, including all datasets. We leave the exploration of this direction as future work.

Preparing for Concurrent Writes. During the rebalance operation, which may take a relatively long time to finish, some records may be updated by concurrent writers. For each bucket that needs to be moved, these concurrent writes must still be applied to its old

[4]The goal of the partition problem is to partition a multiset S of positive integers into two subsets S_1 and S_2 such that the difference between the sum of elements in S_1 and the sum of elements in S_2 is minimized [6].

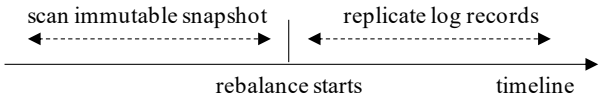


Figure 4: Concurrency Control for Writes

partition since the rebalance operation may fail. Moreover, the concurrent writes must also be applied to the new partition to ensure that there are no lost writes if the rebalance operation succeeds.

To ensure correctness with concurrent writes, we use a concurrency control protocol that splits all the writes of a bucket based on the rebalance start time, as depicted in Figure 4. For all writes that happened before the rebalance operation starts, an immutable snapshot is created so that it can be safely scanned. For all writes that happen after the rebalance operation starts, their log records are replicated to the new partition so that the new partition will not miss any writes. It should be noted that AsterixDB only supports a very simple record-level transaction model without supporting snapshot scans. To implement the required snapshot scan, we exploit the immutability of LSM disk components. Specifically, the memory component of the moving bucket is flushed synchronously during the initialization phase. Thus, the flush time is treated as the rebalance start time, and the resulting disk components become the immutable copy of all writes that happened before the rebalance operation starts. To reduce the blocking of concurrent writes due to the synchronous flush, the two-flush approach described in Algorithm 1 (lines 5-7) can be used. Specifically, one can first flush the memory component asynchronously and then use a synchronous flush to persist the leftover writes. In this case, the rebalance start time becomes the time of the second (synchronous) flush.

5.2 Data Movement Phase

After the initialization phase, the rebalance operation starts to move the affected buckets to their new partitions. This involves adding scanned records and replicated log records to both the primary index and secondary indexes at their destination partitions. Moreover, queries must be handled properly so that they are not affected by the rebalance operation.

Data Movement. By comparing the current global directory and the new global directory, it is straightforward to determine the new partition of each affected bucket. For ease of discussion, here we first describe how to move one bucket B from its old partition P_{old} to its new partition P_{new}, which is then extended to moving multiple buckets together.

Figure 5 shows the basic data movement process for a single bucket with a primary index and one secondary index. At the old partition, the primary index disk components of this bucket are scanned and the log records of any concurrent writes will be replicated. The scanned log records are then used to load disk components at the new partition, and the data represented by the replicated log records are inserted into the memory components. In order to simplify concurrency control and recovery, the moved records are always stored separately from local user writes at the new partition. For a primary index that uses the bucketed LSM-tree design, the received records are simply stored in a new bucket. For

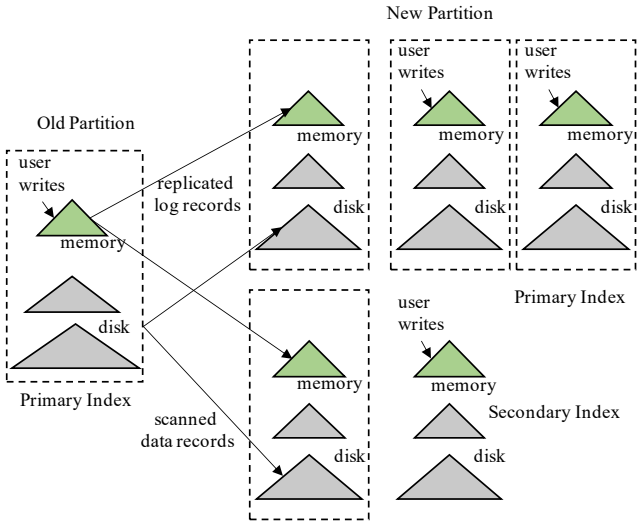


Figure 5: Data Movement Process (One Bucket): *Scanned data records are loaded into disk components and replicated log records are inserted into memory components. These rebalance writes are stored separately from user writes.*

a secondary index that stores all buckets together, the received records are stored into a new list of components that are kept invisible to queries. This design greatly simplifies concurrency control and recovery. These new components that store moved records will be made invisible to queries until the rebalance completes. Moreover, in case the rebalance operation fails, these new components can then be simply deleted to cleanup the intermediate results. Finally, to ensure correctness, the scanned data records must be treated as being strictly older than the replicated log records. This is achieved by placing the loaded disk component after the disk components storing replicated log records.

It is straightforward to extend the basic data movement process to move multiple buckets at the same time. One can simply scan multiple buckets at the same time and repartition them using the new global directory so that the scanned records can be sent to their new partitions. As an optimization, when adding multiple buckets to a secondary index partition, the records can be added to a single list of components instead of creating one list per bucket. This will help to reduce the number of disk components present after the rebalance operation completes.

Handling Concurrent Queries. As mentioned before, the data movement process shown in Figure 5 greatly simplifies the required concurrency control for queries. Since the moved records are stored separately from user writes, the partially loaded buckets are invisible to queries until rebalancing completes. If a query starts before a rebalance operation completes, the query accesses all buckets using the old global directory. Otherwise, the query uses the new global directory, updated by the rebalance operation, to access all buckets. Moreover, since accessed buckets and LSM components are reference counted, they can be accessed safely by the query even if a rebalance operation completes in the middle of the query.

5.3 Finalization Phase

After all data records of moving buckets have been transferred to their new partitions, the system is ready to commit or abort the rebalance operation depending on its outcome. It should be noted that there could still be active log replication activities, due to concurrent writes, at this stage. Thus, to ensure that all nodes always reach a unanimous decision, we use a two-phase commit protocol with a prepare phase and a commit phase.

Prepare Phase. After all data records have been moved to their new partitions, the CC initializes the prepare phase, which will block incoming queries and writes on the rebalancing dataset. The CC further waits for all NCs to complete their log replication and to flush the memory components that store rebalancing writes to disk. If all NCs succeed in doing so, i.e., they all vote yes, the CC enters the *commit* phase as discussed below. Otherwise, the rebalance operation must be aborted and the rebalancing dataset will be left unchanged. It should be noted that all incoming reads and writes will be blocked during the finalization phase. However, this blocking is expected to be very short since the CC only waits for existing writers to complete and the number of log records pending for replication are bounded[5].

Commit Phase. Once the CC enters the commit phase, it forces a COMMIT log record to disk indicating that the rebalance operation is committed. The CC then updates the global directory of the rebalancing dataset and notifies all NCs to install their received buckets and cleanup the moved buckets. To install a received bucket at a partition, one simply needs to add (i.e., register) the loaded disk components to the component lists of the primary index and secondary indexes. To cleanup a moved bucket from the primary index of a partition, the bucket can be simply removed from the bucketed LSM-tree's local directory so that it cannot be accessed by all new queries. It should be noted that because of reference counting, the actual components of this bucket will not be deleted until the last reader exits. To cleanup a secondary index, we use a lazy delete approach that adds the hash value and the depth of this bucket to the metadata of each LSM component. A query then performs an additional validation check to ignore all invalid entries that belong to this moved bucket. Thus, the cleanup of secondary index components is effectively postponed to the next round of merges. All these operations, e.g., adding and removing buckets, are made persistent by forcing metadata files to disk. After all NCs have completed these tasks, the CC can resume query processing and data ingestion on the rebalancing dataset. Finally, the CC produces a DONE log record to indicate that no additional work is needed for this rebalance operation.

Based on the two-phase commit protocol, the final outcome of the rebalance operation is determined by whether the COMMIT log record has been forced to disk successfully by the CC. In other words, the rebalance operation is committed if the COMMIT log record has been successfully forced to disk. Otherwise, the CC simply aborts the rebalance operation and leaves the original dataset as is. We will further discuss how to handle various rebalance failures below.

[5]In AsterixDB, each sender node uses multiple log buffers to store the log records to be replicated. Whenever a log buffer is full, it is replicated to the destination node synchronously. Thus, the total number of pending log records is bounded by the total log buffer size, which is usually a few MBs.

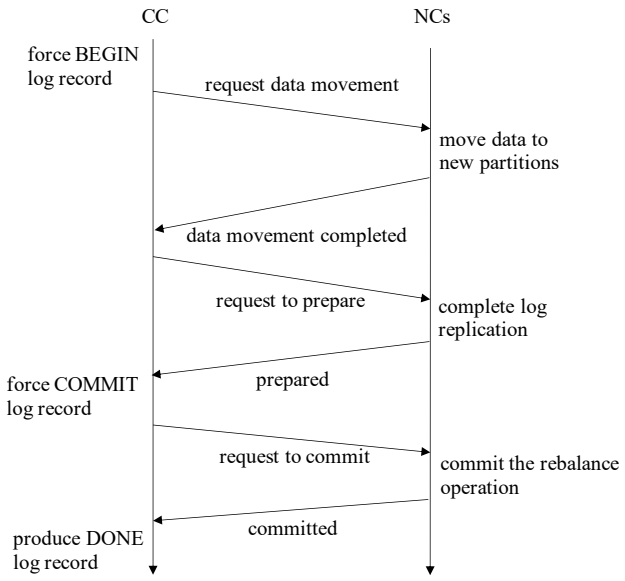


Figure 6: Rebalance Operation Timeline

5.4 Handling Rebalance Failures

During the rebalance operation, some node(s) may potentially fail at any time. Since node failures are expected to be rare, we simply abort the rebalance operation if some node(s) fails before the rebalance operation commits. However, the intermediate results produced by the rebalance operation must be cleaned up carefully to ensure that the dataset remains in a consistent state. Here we assume that the failed node(s) eventually recovers, i.e., no permanent node failures. We plan to extend DynaHash to incorporate replication [11] to handle permanent node failures as future work.

Before discussing how to handle various rebalance failures, we first summarize the basic timeline of a rebalance operation, which is shown in Figure 6. The CC first forces a BEGIN log record indicating that the rebalance operation has started. It then requests all NCs to move their affected buckets to new partitions. After all data movement is done, the CC enters the prepare phase by waiting for all NCs to complete log replication. After all NCs have successfully prepared, the CC enters the commit phase by forcing a COMMIT log record and notifying all NCs to commit this rebalance operation. Finally, after all NCs have committed, the CC produces a DONE log record indicating that the rebalance operation can be safely forgotten. Based on this timeline, we present a case analysis to discuss how to handle various possible rebalance failures.

Case 1: NC fails before voting "prepared". In this case, the CC simply aborts the rebalance operation and asks all NCs (including the failed NC after its recovery) to cleanup their received buckets. Recall from Figure 5 that the received records are always added to a separate list of components. Thus, to cleanup the received buckets, a partition can simply delete those lists of components for both the primary index and secondary indexes of the dataset. It should be noted that cleaning up a received bucket is idempotent since cleaning up a non-existent bucket can be simply treated as a no-op. It is thus safe to cleanup a received bucket from a partition

multiple times. After all NCs complete the cleanup task, the CC writes a DONE log record so that this rebalance operation can be safely forgotten.

Case 2: NC fails after voting "prepared". After the failed NC recovers, it contacts the CC to report its presence. The NC will further receive instructions about how to handle the pending rebalance operation. If the rebalance operation is aborted, the NC simply cleans up the intermediate results as in Case 1. Otherwise, the NC performs the commit tasks as in Case 4.

Case 3: CC fails before forcing the COMMIT log record. After the CC recovers and sees the BEGIN log record for the rebalance operation, it aborts the rebalance operation as in Case 1.

Case 4: NC fails before responding "committed". The rebalance operation is committed but the CC does not know whether the NC has committed the rebalance operation or not. When the NC recovers, the CC requests this NC to commit the rebalance operation by adding the received buckets and cleaning up the moved buckets. Similar to case 3, both adding the received buckets and cleaning up the moved buckets are idempotent operations, which means it is safe to apply these operations multiple times.

Case 5: CC fails after forcing the COMMIT log record but not the DONE log record. In this case, the rebalance operation is effectively committed but it is possible that some NCs have not completed the commit tasks yet. Thus, after the CC recovers, it notifies all NCs to add received buckets and cleanup moved buckets as in Case 4. Finally, the CC writes a DONE log record as well.

Case 6: CC fails after the DONE log record is persisted. No additional task needs to be performed in this case since the DONE log record indicates that this rebalance operation has completed.

The two-phase commit protocol used in our rebalance operation has some subtle differences from the traditional two-phase commit protocol used in distributed transactions. For example, the CC forces a BEGIN log record when a rebalance operation starts and NCs always contact the CC during recovery. This is because in AsterixDB the rebalance operation is implemented as a metadata transaction, and only the CC can produce metadata log records. In contrast, in traditional distributed transactions each participant can produce log records. Because of this difference, without forcing the BEGIN log record, the CC may not know the existence of a rebalance operation if the entire cluster shuts down before the rebalance operation completes. Similarly, the NC must always contact the CC upon recovery since the NC cannot certainly know the status of ongoing rebalance operations. (Contacting the CC upon recovery does not involve additional overhead since the NC must register itself with the CC for cluster management anyway.)

6 EXPERIMENTAL EVALUATION

In this section, we experimentally evaluate the proposed rebalancing techniques in the context of Apache AsterixDB [1]. Throughout the evaluation, we mainly focus on the three aspects of different rebalancing approaches, namely their rebalancing performance, ingestion performance, and query performance. In the remainder of this section we describe the general experimental setup followed by the detailed evaluation results.

6.1 Experimental Setup

Hardware. All experiments were performed on a cluster of nodes with a single CC and multiple NCs on AWS. The number of NCs ranged from 2 to 16. The CC ran on a m5.xlarge node with 4 vCPUs, 16GB of memory, and a 500GB elastic block store (EBS). Each NC ran on an i3.xlarge node with 4 vCPUs, 30.5GB of memory, a 950GB SSD, and a 500GB EBS. We configured 4 partitions on each NC to exploit the parallelism provided by multiple cores. The native SSD was used for LSM storage and the EBS was used for transactional logging. Each NC used a thread pool with 4 threads to execute LSM flush and merge operations. Each LSM-tree used a size-tiered merge policy with a size ratio of 1.2 throughout the experiments, which is similar to the one used in other systems. This policy merges a sequence of components when the total size of the younger components is 1.2 times larger than that of the oldest component in the sequence. We allocated 26GB of memory for the AsterixDB instance. The buffer cache size was set at 8GB and the memory component budget was set at 2GB. Each memory-intensive query operator [46], such as sort, hash join, and hash group by, received a 128MB memory budget. Both the disk page size and memory page size were set at 16KB.

Workload. To understand the performance impact of different rebalancing approaches on OLAP-style workloads, we used the TPC-H [7] benchmark in our evaluation. We built two secondary indexes, on LineItem and Orders, to enable index-only plans for certain queries. The LineItem index contains l_shipdate, l_partkey, l_suppkey, l_extendedprice, l_discount, and l_quantity. The Orders index contains o_orderdate, o_custkey, o_shippriority, and o_orderpriority. The scale factor of the TPC-H benchmark was set to 100 times the number of NCs so that the total amount of data scales linearly as the cluster size increases. Thus, each NC stored 100GB of TPC-H raw data. The primary index was compressed for better storage efficiency. The total storage size at each NC, including compressed primary indexes and uncompressed secondary indexes, was about 130GB.

Evaluated Rebalancing Approaches. We evaluated three rebalancing approaches. The first approach that we evaluated is AsterixDB's original global rebalancing approach with hash partitioning (called "hashing") as the baseline. This approach simply creates a new dataset that is hash partitioned based on the new (target) set of nodes during rebalancing. Although hashing achieves a near perfect load balance, it incurs a very high rebalance cost and nearly doubles the dataset's disk usage during rebalancing. Second, we evaluated DynaHash, where the maximum bucket size was set at 10GB. After loading the TPC-H data, each partition always had 4 buckets. Finally, we evaluated a static bucketing approach, called StaticHash, that always splits a dataset into 256 buckets. Here 256 was determined by considering the largest cluster size in our evaluation, i.e., 16, so that each partition can have 4 buckets. The actual number of buckets per partition ranged from 32 to 4 as the number of nodes varied from 2 to 16. Thus, evaluating StaticHash also shows the performance impact of the number of buckets per partition.

6.2 Ingestion Performance

We first evaluated the ingestion performance of the different rebalancing approaches. We used a TPC-H client that ran on a separate

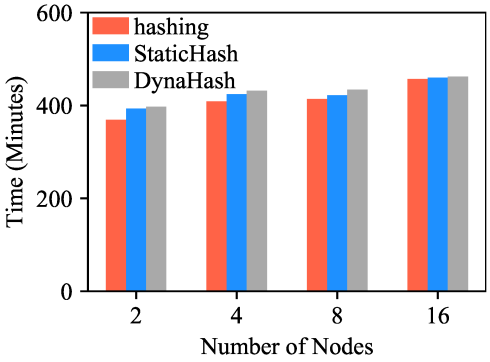
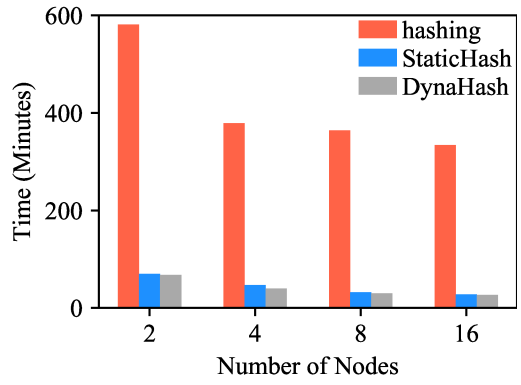
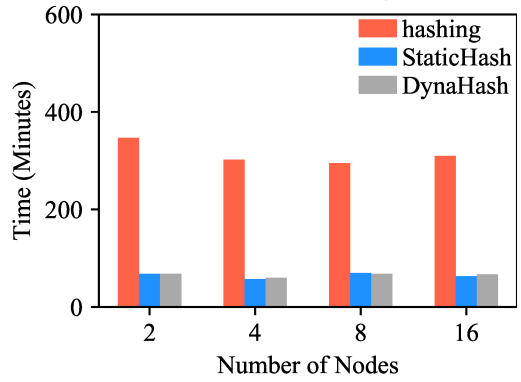


Figure 7: Ingestion Time



(a) Rebalance Time for Removing Nodes



(b) Rebalance Time for Adding Nodes

Figure 8: Rebalance Time

node to ingest all TPC-H data into the AsterixDB cluster. The number of nodes of the AsterixDB cluster varied from 2 to 16.

The resulting ingestion time for each rebalancing approach under different cluster sizes is shown in Figure 7. In general, DynaHash incurs very a small overhead compared to the AsterixDB's original hashing approach. Moreover, by comparing DynaHash and StaticHash, we see that the number of buckets per partition also has just a small impact on the ingestion performance. When the cluster size increases, the ingestion time of all rebalancing approaches slightly increases because of the write stall problem of LSM-trees:

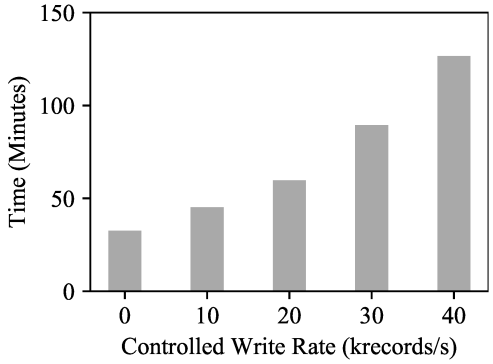


Figure 9: Rebalance Time with Concurrent Data Ingestion

In general, we found that data ingestion is relatively CPU-heavy in AsterixDB due to record parsing. When a node has active merges, its ingestion rate will slow down due to the CPU contention caused by merges. This in turn will slow down the entire cluster because the overall performance of a shared-nothing system is bottlenecked by the slowest node, even though other nodes may not have ongoing merges. Thus, when the number of nodes increases, the write stall problem becomes more obvious, which increases the overall ingestion time.

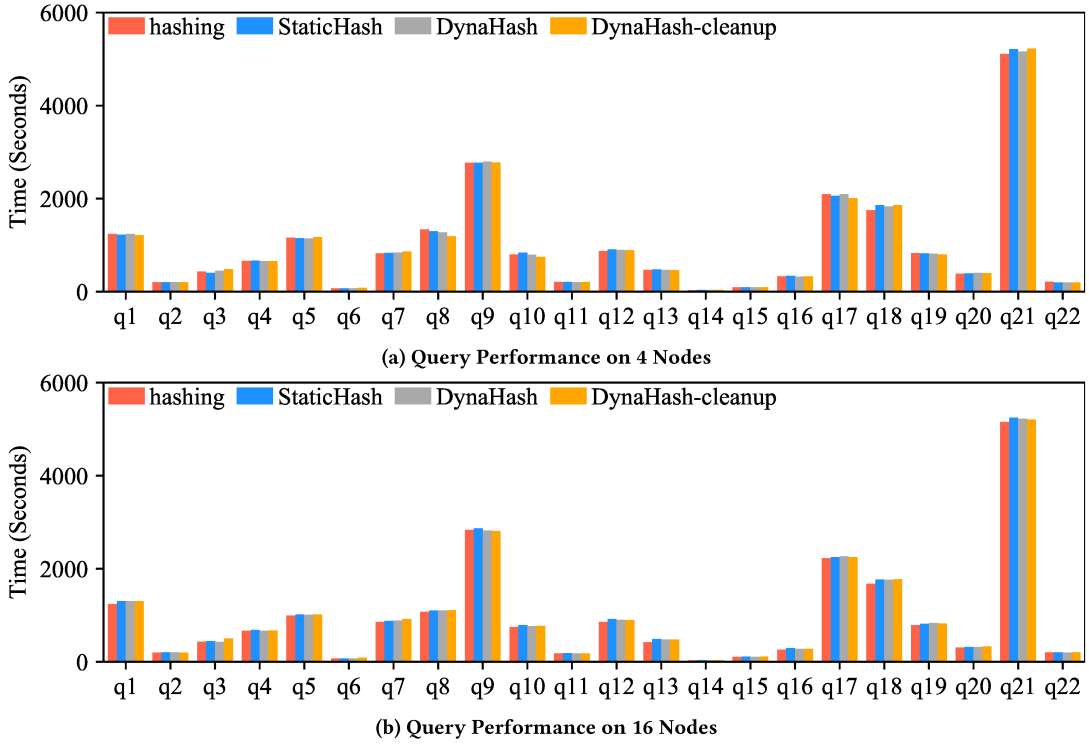
6.3 Rebalancing Performance

Next we evaluated the rebalancing performance of the alternative rebalancing approaches, both for adding nodes and for removing nodes. We further evaluated the impact of concurrent writes on the rebalance performance.

Basic Rebalancing Performance. To under the basic rebalancing performance of the different rebalancing approaches, we conducted the following experiments. We first loaded the TPC-H datasets into an AsterixDB cluster with N nodes (N ranged from 2 to 16). We then rebalanced all datasets to N-1 nodes to measure the time to remove one node. Finally, we rebalanced all datasets back to N nodes to measure the time to add one node.

The rebalance times for removing and adding nodes are shown in Figure 8. In general, both StaticHash and DynaHash substantially reduce the rebalancing time compared with hashing for both removing and adding nodes. Moreover, both of the bucketing approaches also have similar rebalance times, which shows that the number of buckets per partition has a small impact on the rebalancing performance. Interestingly, we see that hashing has different performance trends compared with StaticHash and DynaHash. Hashing has better rebalancing performance for adding than for removing nodes since the rebalancing work is distributed across N nodes. When a node is removed, however, the rebalancing work is only distributed over N-1 nodes. In contrast, for StaticHash and DynaHash, removing a node is more efficient than adding one since the rebalancing work for node removal is distributed across the remaining N-1 nodes. However, when a node is added, the new node becomes the bottleneck because it receives data from all N-1 existing nodes.

Impact of Concurrent Writes. We further evaluated the impact of concurrent writes on the rebalancing performance of DynaHash. In this experiment, we rebalanced the datasets from 4 nodes



(a) Query Performance on 4 Nodes

(b) Query Performance on 16 Nodes

Figure 10: Query Performance on Original Cluster

to 3 nodes and inserted new records into the LineItem dataset while rebalancing was active. The resulting rebalancing time under different write rates is shown in Figure 9. As one can see, the rebalance operation takes longer to finish when the write rate becomes larger. This is expected because these concurrent writes compete for CPU and I/O resources with the rebalancing operation. Thus, it is desirable to schedule rebalance operations during off-peak hours to minimize contention with user workloads. Moreover, as the result shows, even under high write rates, the rebalance operation can still be completed in a reasonable amount of time.

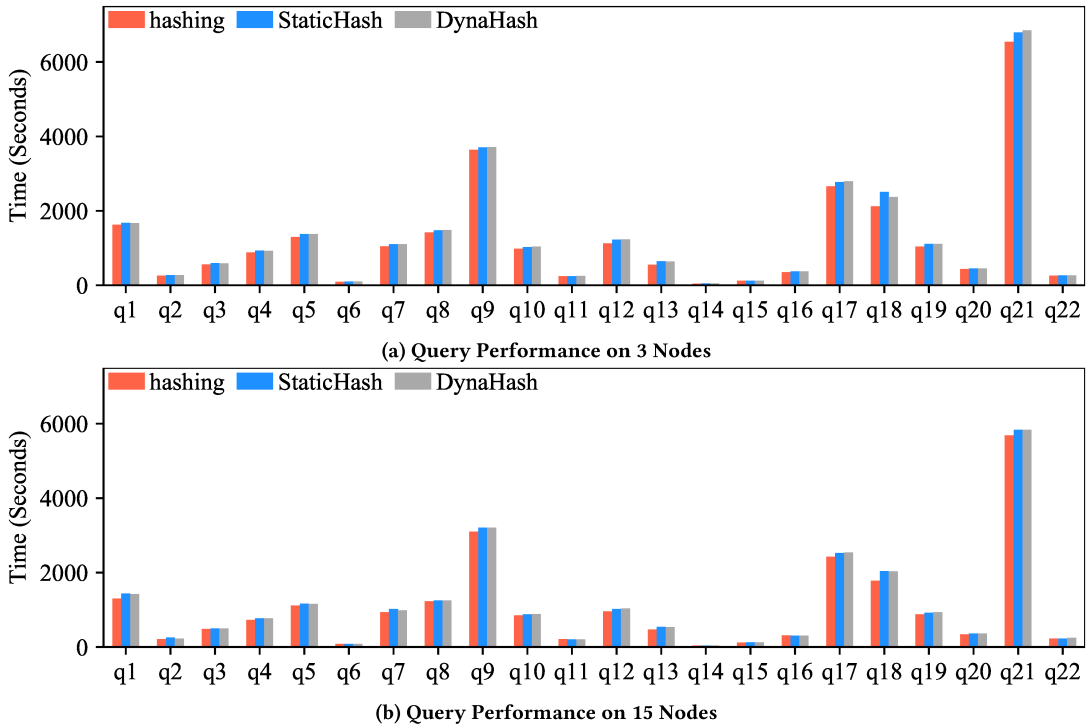
6.4 Query Performance

Last but not least, we evaluated the query performance of the different rebalancing approaches, focusing on the following three questions: First, what is the overhead of the proposed bucketed LSM-tree structure on query performance? Second, what is the impact on query performance of the load balance of the various rebalancing approaches? Finally, what is the overhead due to lazy secondary index cleanup on query performance?

To answer these questions, we designed a series of experiments as follows. First, we evaluated the query performance on a cluster with 4 or 16 nodes without rebalancing, which helps to answer the first question. We then rebalanced the datasets to 3 or 15 nodes so that we can evaluate the load balance impact of the rebalancing approaches, which answers the second question. Finally for DynaHash, we rebalanced the datasets back to 4 or 16 nodes so that we can evaluate the performance impact of lazy secondary index cleanup (denoted as “DynaHash-cleanup").

The resulting query times on the original cluster size (4 or 16 nodes) are shown in Figure 10. Note that on 16 nodes, StaticHash and DynaHash are expected to have similar behavior because they have the same number of buckets per partition. In general, both StaticHash and DynaHash are seen to add a negligible overhead on most TPC-H queries when compared with hashing. This shows that bucketed LSM-trees have a negligible overhead for OLAP-style queries. Moreover, all bucketing approaches achieve very good scale-up because the query times remain nearly constant when both the number of nodes and the dataset size increase. Here one minor exception is q18, where StaticHash and DynaHash incur a small overhead compared with hashing. The reason is that q18 performs a groupby on the prefix of LineItem’s primary keys, which requires the scanned records to be ordered on the primary keys. In this case, the bucketed LSM-tree incurs some additional overhead because it has to merge-sort more disk components. Moreover, StaticHash also incurs a larger overhead on q18 under 4 nodes (Figure 10a) because it has 16 buckets per partition. Finally, we see that lazy secondary index cleanup (DynaHash-cleanup) also has a negligible overhead on TPC-H queries, which shows the effectiveness of this technique. This is because with lazy secondary index cleanup, queries only need to access some obsolete secondary index entries, and the added processing time is very small compared to the overall query time.

The resulting query times on the resized cluster (3 or 15 nodes) are shown in Figure 11. Since the number of buckets cannot be divided by the number of partitions, both StaticHash and DynaHash result in some load imbalance where some partitions may have one more bucket than others. Despite this load imbalance,



(a) Query Performance on 3 Nodes

(b) Query Performance on 15 Nodes

Figure 11: Query Performance on Resized Cluster

both StaticHash and DynaHash only incur a very small overhead on most TPC-H queries. This is because the load imbalance only impacts the data scan time, while most TPC-H queries are relatively computation heavy. However, for scan-heavy queries, such as q17, q18, and q21, the overhead caused by a load imbalance becomes more noticeable. For example, q17 and q18 each perform a full scan over the LineItem dataset to perform groupby and aggregation, and q21 further scans the LineItem dataset multiple times. Moreover, as shown in Figure 11a, using more buckets per partition as in the StaticHash approach slightly reduces the overhead of the load imbalance for some queries, such as q21, but doing so incurs some additional overhead for queries that require a sorted order coming from primary index scans, such as q18.

6.5 Summary of Experimental Results

In general, our experimental results are consistent with the discussion in Section 2.1. Global rebalancing with hash partitioning achieves the best ingestion and query performance, but results in a very large rebalance cost. In contrast, DynaHash significantly reduces the rebalance time with only a small overhead on the ingestion and query performance. The proposed bucketed LSM-tree structure only incurs a small overhead for queries that require the scanned records to be ordered by primary keys. Moreover, the load imbalance caused by DynaHash mainly impacts the dataset scan performance. Thus, overall DynaHash incurs a negligible overhead on computation-intensive queries with a small overhead on scan-heavy queries. By comparing StaticHash and DynaHash, it can be seen that having more buckets per partition achieves a better load balance, but it also leads to a larger overhead for queries that require the scanned records to be ordered on primary keys. Moreover, DynaHash has better usability since the resulting number buckets per partition is dynamically adjusted as the cluster and dataset size scales, resulting in more stable performance.

7 CONCLUSION

In this paper, we have described the design and implementation of DynaHash, an efficient data rebalancing approach that combines dynamic bucketing with extensible hashing in Apache AsterixDB. We first introduced a bucketed LSM-tree design for efficiently storing multiple buckets. We further described an efficient rebalancing implementation that exploits the LSM-tree's out-of-place update design to support concurrent reads and writes. An experimental evaluation using the TPC-H benchmark has shown that the proposed techniques significantly reduce the rebalance cost with negligible overheads for data ingestion and query processing. In the future, we plan to extend DynaHash to incorporate replication to provide better availability and fault tolerance.

ACKNOWLEDGMENTS

This work has been supported by NSF awards CNS-1305430, IIS-1447720, IIS-1838248, and CNS-1925610 along with industrial support from Amazon, Google, and Microsoft and support from the Donald Bren Foundation (via a Bren Chair).

REFERENCES

[1] 2020. AsterixDB. https://asterixdb.apache.org/.
[2] 2020. HBase. https://hbase.apache.org/.

[3] 2020. HDFS Architecture. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html.
[4] 2020. MongoDB. https://www.mongodb.com/.
[5] 2020. Oracle NoSQL Database Cloud Service. https://www.oracle.com/database/nosql-cloud.html.
[6] 2020. Partition problem. https://en.wikipedia.org/wiki/Partition_problem.
[7] 2020. TPC-H. http://www.tpc.org/tpch/.
[8] 2020. VoltDB. https://www.voltdb.com/.
[9] Michael Abebe, Brad Glasbergen, and Khuzaima Daudjee. 2020. MorphoSys: Automatic Physical Design Metamorphosis for Distributed Database Systems. *Proc. VLDB Endow.* 13, 13 (2020), 3573–3587.
[10] Marcos K. Aguilera, Wojciech Golab, and Mehul A. Shah. 2008. A Practical Scalable Distributed B-Tree. *Proc. VLDB Endow.* 1, 1 (Aug. 2008), 598–609.
[11] Murtadha Makki Al Hubail. 2016. *Data Replication and Fault Tolerance in AsterixDB.* Master's thesis. UC Irvine.
[12] Wail Y. Alkowaileet, Sattam Alsubaiee, and Michael J. Carey. 2020. An LSM-Based Tuple Compaction Framework for Apache AsterixDB. *Proc. VLDB Endow.* 13, 9 (2020), 1388–1400.
[13] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak Borkar, Yingyi Bu, Michael Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, Eugenia Gabrielova, Raman Grover, Zachary Heilbron, Young-Seok Kim, Chen Li, Guangqiang Li, Ji Mahn Ok, Nicola Onose, Pouria Pirzadeh, Vassilis Tsotras, Rares Vernica, Jian Wen, and Till Westmann. 2014. AsterixDB: A Scalable, Open Source BDMS. *Proc. VLDB Endow.* 7, 14 (2014), 1905–1916.
[14] Sattam Alsubaiee, Alexander Behm, Vinayak Borkar, Zachary Heilbron, Young-Seok Kim, Michael J. Carey, Markus Dreseler, and Chen Li. 2014. Storage Management in AsterixDB. *Proc. VLDB Endow.* 7, 10 (2014), 841–852.
[15] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. 2019. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 753–766.
[16] Oana Balmau, Rachid Guerraoui, Vasileios Trigonakis, and Igor Zablotchi. 2017. FloDB: Unlocking Memory in Persistent Key-Value Stores. In *Proceedings of the Twelfth European Conference on Computer Systems*. 80–94.
[17] Sean Barker, Yun Chi, Hyun Jin Moon, Hakan Hacigümüş, and Prashant Shenoy. 2012. "Cut Me Some Slack": Latency-Aware Live Migration for Databases. In *Proceedings of the 15th International Conference on Extending Database Technology (EDBT '12)*. 432–443.
[18] Laurent Bindschaedler, Ashvin Goel, and Willy Zwaenepoel. 2020. Hailstorm: Disaggregated Compute and Storage for Distributed LSM-Based Databases. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 301–316.
[19] Burton H. Bloom. 1970. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (1970), 422–426.
[20] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. 1990. Prototyping Bubba, a highly parallel database system. *IEEE Transactions on Knowledge and Data Engineering* 2, 1 (1990), 4–24.
[21] Dipti Borkar, Ravi Mayuram, Gerald Sangudi, and Michael Carey. 2016. Have Your Data and Query It Too: From Key-Value Caching to Big Data Management. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. 239–251.
[22] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica. 2011. Hyracks: A flexible and extensible foundation for data-intensive computing. In *2011 IEEE 27th International Conference on Data Engineering*. 1151–1162.
[23] Edward Bortnikov, Anastasia Braginsky, Eshcar Hillel, Idit Keidar, and Gali Sheffi. 2018. Accordion: Better Memory Organization for LSM Key-Value Stores. *Proc. VLDB Endow.* 11, 12 (2018), 1863–1875.
[24] M. J. Carey. 2019. AsterixDB Mid-Flight: A Case Study in Building Systems in Academia. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 1–12.
[25] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.* 26, 2, Article 4 (2008), 26 pages.
[26] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. 215–226.
[27] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. 2011. Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud Using Live Data Migration. *Proc. VLDB Endow.* 4, 8 (May 2011), 494–505.
[28] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 79–94.
[29] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. In *Proceedings of the 2018 International Conference on Management of Data*. 505–520.
[30] Niv Dayan and Stratos Idreos. 2019. The Log-Structured Merge-Bush & the Wacky Continuum. In *Proceedings of the 2019 International Conference on Management of Data*. 449–466.
[31] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*. 205–220.
[32] David DeWitt and Jim Gray. 1992. Parallel Database Systems: The Future of High Performance Database Systems. *Commun. ACM* 35, 6 (1992), 85–98.
[33] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. . Hsiao, and R. Rasmussen. 1990. The Gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering* 2, 1 (1990), 44–62.
[34] C. du Mouza, W. Litwin, and P. Rigaux. 2007. SD-Rtree: A Scalable Distributed Rtree. In *2007 IEEE 23rd International Conference on Data Engineering*. 296–305.
[35] Cédric du Mouza, Witold Litwin, and Philippe Rigaux. 2009. Large-scale indexing of spatial data in distributed repositories: the SD-Rtree. *The VLDB Journal* 18, 4 (2009), 933–958.
[36] Carla Schlatter Ellis. 1983. Extendible Hashing for Concurrent Operations and Distributed Data. In *Proceedings of the 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*. 106–116.
[37] Aaron J. Elmore, Vaibhav Arora, Rebecca Taft, Andrew Pavlo, Divyakant Agrawal, and Amr El Abbadi. 2015. Squall: Fine-Grained Live Reconfiguration for Partitioned Main Memory Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 299–313.
[38] Aaron J. Elmore, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2011. Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*. 301–312.
[39] Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger, and H. Raymond Strong. 1979. Extendible Hashing—a Fast Access Method for Dynamic Files. *ACM Trans. Database Syst.* 4, 3 (Sept. 1979), 315–344.
[40] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. 2012. SAP HANA Database: Data Management for Modern Business Applications. *SIGMOD Rec.* 40, 4 (2012), 45–51.
[41] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google File System. *SIGOPS Oper. Syst. Rev.* 37, 5 (2003), 29–43.
[42] R. Grover and M. Carey. 2015. Data Ingestion in AsterixDB. In *EDBT*. 605–616.
[43] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. 2015. Amazon Redshift and the Case for Simpler Data Warehouses. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. 1917–1923.
[44] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. 2020. TiDB: A Raft-Based HTAP Database. *Proc. VLDB Endow.* 13, 12 (2020), 3072–3084.
[45] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing (STOC '97)*. 654–663.
[46] Taewoo Kim, Alexander Behm, Michael Blow, Vinayak Borkar, Yingyi Bu, Michael J. Carey, Murtadha Hubail, Shiva Jahangiri, Jianfeng Jia, Chen Li, Chen Luo, Ian Maxon, and Pouria Pirzadeh. 2020. Robust and efficient memory management in Apache AsterixDB. *Software: Practice and Experience* 50, 7 (2020), 1114–1151.
[47] Brigitte Kröll and Peter Widmayer. 1994. Distributing a Search Tree among a Growing Number of Processors. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*. 265–276.
[48] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.* 44, 2 (April 2010), 35–40.
[49] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. 2012. The Vertica Analytic Database: C-Store 7 Years Later. *Proc. VLDB Endow.* 5, 12 (2012), 1790–1801.
[50] Yongkun Li, Helen H. W. Chan, Patrick P. C. Lee, and Yinlong Xu. 2019. Enabling Efficient Updates in KV Storage via Hashing: Design and Performance Evaluation. *ACM Trans. Storage* 15, 3 (2019).
[51] Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. 2016. Towards Accurate and Fast Evaluation of Multi-Stage Log-structured Designs. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. 149–166.
[52] Yu-Shan Lin, Shao-Kan Pi, Meng-Kai Liao, Ching Tsai, Aaron Elmore, and Shan-Hung Wu. 2019. MgCrab: Transaction Crabbing for Live Migration in Deterministic Database Systems. *Proc. VLDB Endow.* 12, 5 (2019), 597–610.

[53] Witold Litwin, Marie-Anne Neimat, and Donovan A. Schneider. 1993. LH: Linear Hashing for Distributed Files. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*. 327–336.

[54] Witold Litwin, Marie-Anne Neimat, and Donovan A. Schneider. 1994. RP*: A Family of Order Preserving Scalable Distributed Data Structures. In *Proceedings of the 20th International Conference on Very Large Data Bases*. 342–353.

[55] Witold Litwin, Marie-Anna Neimat, and Donovan A. Schneider. 1996. LH*—a Scalable, Distributed Data Structure. *ACM Trans. Database Syst.* 21, 4 (1996), 480–525.

[56] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. WiscKey: Separating Keys from Values in SSD-Conscious Storage. *ACM Trans. Storage* 13, 1 (2017).

[57] Chen Luo. 2020. Breaking Down Memory Walls in LSM-Based Storage Systems. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2817–2819.

[58] Chen Luo and Michael J. Carey. 2019. Efficient Data Ingestion and Query Processing for LSM-Based Storage Systems. *Proc. VLDB Endow.* 12, 5 (2019), 531–543.

[59] Chen Luo and Michael J. Carey. 2019. On Performance Stability in LSM-Based Storage Systems. *Proc. VLDB Endow.* 13, 4 (2019), 449–462.

[60] Chen Luo and Michael J. Carey. 2020. Breaking Down Memory Walls: Adaptive Memory Management in LSM-based Storage Systems. *Proc. VLDB Endow.* 14, 3 (2020), 241–254.

[61] Chen Luo and Michael J Carey. 2020. LSM-based storage techniques: a survey. *The VLDB Journal* 29, 1 (2020), 393–418.

[62] Chen Luo, Pinar Tözün, Yuanyuan Tian, Ronald Barber, Vijayshankar Raman, and Richard Sidle. 2019. Umzi: Unified Multi-Zone Indexing for Large-Scale HTAP. In *22nd International Conference on Extending Database Technology*. 1–12.

[63] Siqiang Luo, Subarna Chatterjee, Rafael Ketsetsidis, Niv Dayan, Wilson Qin, and Stratos Idreos. 2020. Rosetta: A Robust Space-Time Optimized Range Filter for Key-Value Stores. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2071–2086.

[64] Q. Mao, S. Jacobs, W. Amjad, V. Hristidis, V. J. Tsotras, and N. E. Young. 2019. Experimental Evaluation of Bounded-Depth LSM Merge Policies. In *2019 IEEE International Conference on Big Data (Big Data)*. 523–532.

[65] Ryan Marcus, Olga Papaemmanouil, Sofiya Semenova, and Solomon Garber. 2018. NashDB: An End-to-End Economic Method for Elastic Database Fragmentation, Replication, and Provisioning. In *Proceedings of the 2018 International Conference on Management of Data*. 1253–1267.

[66] Fei Mei, Qiang Cao, Hong Jiang, and Jingjun Li. 2018. SifrDB: A Unified Solution for Write-Optimized Key-Value Stores in Large Datacenter. In *Proceedings of the ACM Symposium on Cloud Computing*. 477–489.

[67] U. F. Minhas, R. Liu, A. Aboulnaga, K. Salem, J. Ng, and S. Robertson. 2012. Elastic Scale-Out for Partition-Based Database Systems. In *2012 IEEE 28th International Conference on Data Engineering Workshops*. 281–288.

[68] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Inf.* 33, 4 (1996), 351–385.

[69] Mohiuddin Abdul Qader, Shiwen Cheng, and Vagelis Hristidis. 2018. A Comparative Study of Secondary Indexing Techniques in LSM-Based NoSQL Databases. In *Proceedings of the 2018 International Conference on Management of Data*. 551–566.

[70] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building Key-Value Stores Using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 497–514.

[71] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. 2017. SlimDB: A Space-Efficient Key-Value Storage Engine for Semi-Sorted Data. *Proc. VLDB Endow.* 10, 13 (2017), 2037–2048.

[72] Subhadeep Sarkar, Tarikul Islam Papon, Dimitris Staratzis, and Manos Athanassoulis. 2020. Lethe: A Tunable Delete-Aware LSM Engine. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 893–908.

[73] D. Schall and T. Härder. 2015. Dynamic physiological partitioning on a shared-nothing database cluster. In *2015 IEEE 31st International Conference on Data Engineering*. 1095–1106.

[74] Oliver Schiller, Nazario Cipriani, and Bernhard Mitschang. 2013. ProRea: Live Database Migration for Multi-Tenant RDBMS with Snapshot Isolation. In *Proceedings of the 16th International Conference on Extending Database Technology*. 53–64.

[75] Russell Sears and Raghu Ramakrishnan. 2012. BLSM: A General Purpose Log Structured Merge Tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 217–228.

[76] Marco Serafini, Essam Mansour, Ashraf Aboulnaga, Kenneth Salem, Taha Rafiq, and Umar Farooq Minhas. 2014. Accordion: Elastic Scalability for Database Systems Supporting Distributed Transactions. *Proc. VLDB Endow.* 7, 12 (Aug. 2014), 1035–1046.

[77] Marco Serafini, Rebecca Taft, Aaron J. Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. 2016. Clay: Fine-Grained Adaptive Partitioning for General Database Schemas. *Proc. VLDB Endow.* 10, 4 (Nov. 2016), 445–456.

[78] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. 2005. C-Store: A Column-Oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases*. 553–564.

[79] X. Sun, J. Yu, Z. Zhou, and C. J. Xue. 2020. FPGA-based Compaction Engine for Accelerating LSM-tree Key-Value Stores. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 1261–1272.

[80] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J. Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. 2014. E-Store: Fine-Grained Elastic Partitioning for Distributed Transaction Processing Systems. *Proc. VLDB Endow.* 8, 3 (2014), 245–256.

[81] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. 2020. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1493–1509.