Lightweight Cardinality Estimation in LSM-based Systems

Ildar Absalyamov
University of California, Riverside
iabsa001@cs.ucr.edu

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

Vassilis J. Tsotras
University of California, Riverside
tsotras@cs.ucr.edu

ABSTRACT

Data sources, such as social media, mobile apps and IoT sensors, generate billions of records each day. Keeping up with this influx of data while providing useful analytics to the users is a major challenge for today's data-intensive systems. A popular solution that allows such systems to handle rapidly incoming data is to rely on log-structured merge (LSM) storage models. LSM-based systems provide a tunable trade-off between ingesting vast amounts of data at a high rate and running efficient analytical queries on top of that data. For queries, it is well-known that the query processing performance largely depends on the ability to generate efficient execution plans. Previous research showed that OLAP query workloads rely on having small, yet precise, statistical summaries of the underlying data, which can drive the cost-based query optimization.

In this paper we address the problem of computing data statistics for workloads with rapid data ingestion and propose a lightweight statistics-collection framework that exploits the properties of LSM storage. Our approach is designed to piggyback on the events (flush and merge) of the LSM lifecycle. This allows us to easily create an initial statistics and then keep them in sync with rapidly changing data while minimizing the overhead to the existing system. We have implemented and adapted well-known algorithms to produce various types of statistical synopses, including equi-width histograms, equi-height histograms, and wavelets. We performed an in-depth empirical evaluation that considers both the cardinality estimation accuracy and runtime overheads of collecting and using statistics. The experiments were conducted by prototyping our approach on top of Apache AsterixDB, an open source Big Data management system that has an entirely LSM-based storage backend.

CCS CONCEPTS

• **Information systems** → *DBMS engine architectures*;

KEYWORDS

LSM storage; database statistics

ACM Reference Format:
Ildar Absalyamov, Michael J. Carey, and Vassilis J. Tsotras. 2018. Lightweight Cardinality Estimation in LSM-based Systems. In *Proceedings of 2018 International Conference on Management of Data (SIGMOD'18).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3183713.3183761

SIGMOD'18, June 10–15, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3183761

1 INTRODUCTION

The sheer number of data sources producing data at ever-increasing volumes creates an unprecedented challenge for modern analytical data-processing systems. The problem is exacerbated by the high rate at which producers generate new records. Such systems often seek an engineering trade-off between their abilities to ingest large amounts of rapidly incoming data and running complex analytical queries on top of it. A popular design that has gained popularity in recent years, addresses this challenge by using log-structured merge trees (LSM-trees) [40] as a storage backend. Initially adopted by NoSQL systems [3, 4, 7, 22] LSM-trees later appeared in a new generation of relational database systems [5, 8]. In the LSM model, ingested records are batched in memory into *components*, amortizing the cost of a single insert. Whenever a memory buffer fills up the batch gets persisted (*flushed*) to the disk at once, requiring only one sequential I/O operation. Flushed components are immutable and their number keeps growing until the system triggers a *merge* operation that consolidates multiple components into a single file and *reconciles* deleted and updated entries; this in turn decreases the amount of I/O needed to perform a lookup query.

Reducing the number of I/Os is not the only technique that improves query performance. Numerous research works have shown that in OLAP setups, the quality of execution plans plays a much bigger role in decreasing the total execution time. The ability to pick a plan with a smaller runtime overhead by estimating the intermediate result cardinality and feeding it to a cost model is a discriminative characteristic of a good query optimizer. However, recent research [37] showed that correct cardinality estimation provides substantial benefits in comparison to fine-tuned optimizer cost models. Thus in this paper we concentrate on building statistical data summaries, known as *data synopses*, which are compressed, yet accurate, representations of the underlying data distributions.

Despite the rapid growth in data volumes, the approaches to collect statistical synopses have not significantly changed over the past decades. The common way to obtain data synopses in most commercial DBMSs, as well as research prototypes, is to launch a background job that will rescan all disk-resident datasets and produce appropriate data summaries. However this naïve approach has multiple drawbacks. Firstly, it suffers from the high I/O introduced by repeated data scanning. This overhead is further exacerbated by the data volume in Big Data analytics systems. Moreover, scheduling such heavy-weight bulk operations becomes a problem itself because it could easily detract from the performance of currently executing user queries. This is especially perceptible in the context of multi-tenant elastic cloud deployments where such "noisy-neighbor" might cause significant spikes in query latency.

The problem of prohibitive I/O costs is often solved by sampling, which considers only a portion of records from each disk page and skips some pages altogether. Once obtained, samples could be used as a data summary on their own [29], or serve as inputs to regular

synopsis-building algorithms [24]. Nevertheless, the accuracy of sampling-based methods is bounded by the fact that they do not see all the records and could miss important items that "fly under the radar" for a given query predicate. Sophisticated stratified estimators have been proposed to overcome the problem of biased sampling [25, 33], but they heavily depend on the ability to identify appropriate stratas in the whole dataset, which often relies on knowing the query workload profiles. Moreover in LSM-based systems collecting the sample is further exacerbated by the fact that physical records in the sample might not represent the most recent version of a corresponding logical record. Despite the considerable progress in calculating samples in partitioned distributed systems [18, 28] we are not aware of algorithms which allow unbiased estimates to be obtained in an LSM setting where deleted and inserted records can appear in any component.

Regardless of the use of sampling, data synopses produced by an offline statistics computation can pretty quickly grow out-of-date, especially for continuous ingestion workloads. An ideal solution would require an incremental synopsis maintenance in combination with identifying which records were updated since the last time statistics were collected. Unfortunately, such synopsis maintenance algorithms inevitably introduce errors that over time lead to decreasing accuracy and require periodically recalculating statistics from scratch depending on some heuristic. Designing a robust policy that specifies when such a recomputation should take place is on its own a difficult problem [29].

In this paper, we propose a lightweight approach for collecting statistics in data-intensive systems that does not suffer from the aforementioned problems. Our solution is based on the LSM storage model and avoids doing unnecessary I/O operations by computing synopses on-the-fly. The nature of the LSM component lifecycle implies that at some point in time each record is an input to some LSM-event (flush, merge). Because the data summary generation is bound to these events, our statistics collection algorithms observe *all of the data items*, in contrast to sampling-based methods. Finally, in the LSM storage model new data is periodically persisted by flushing contents of in-memory components to the disk. This allows our synopsis-gathering algorithms to keep statistics up-to-date with dynamically changing datasets. Furthermore, this piggybacking also eliminates the need for a specific mechanism to identify newly updated records.

The proposed statistics collection framework operates on the data storage critical path; hence building the data summary with a low runtime overhead is an *essential* property. This would effectively eliminate synopses-collecting algorithms with high asymptotic complexity (like V-optimal histograms [35]). Instead, in this paper we generate synopses only on *indexed* (primary or secondary) attributes, and hence use the sorted order imposed by the indexes to devise efficient synopsis-gathering algorithms. We leave calculating statistics for unsorted attributes within this framework as future work.

Since the statistics are generated for each LSM component individually, all component synopses must be queried to obtain the overall cardinality estimate. For a large number of components this might incur significant query time overhead. An alternative is to combine individual synopses into a single statistical summary that is kept in addition to the individual synopses. An incoming query will be served by the merged synopsis; the merge synopsis is re-calculated from the individual ones as new components are flushed or existing components are merged. This however requires the synopsis data structure to be inherently *mergeable*. Synopsis mergeability is also critical in shared-nothing setups where datasets are partitioned across distributed nodes in a cluster. Among the implemented statistical synopses, equi-width histograms and wavelets support merging while equi-height histograms do not. We show experimentally how the synopsis mergeability property directly influences the trade-offs in accuracy, query time overhead, and space allocated to the synopses.

We prototyped and evaluated our design on Apache AsterixDB [2], an open source system that uses the LSM-based storage [12]. Currently, AsterixDB relies on a heuristic-based optimizer, so introducing statistics into that system could be the first step towards building a full-fledged cost-based optimizer. Our main contributions can be summarized as:

- We propose a lightweight statistics collection approach that alleviates the high cost of building synopses on disk-resident data by incorporating the statistics accumulation into the common LSM-based database storage layer lifecycle events.
- We implement streaming algorithms for building equi-width and equi-height histograms and introduce a streaming version of the prefix-sum wavelet decomposition algorithm.
- We prototype our solution on top of Apache AsterixDB, a full open source Big Data management system, and carefully assess the overheads introduced by the proposed framework, both while collecting statistics during ingestion and when using them during query optimization.
- Through extensive experimental evaluation, we examine the accuracy of cardinality estimation for different types of synopses, parameters, and workloads.
- We explore how synopsis mergeability influences the trade-offs between accuracy, query time overhead, and space allocated for synopses.

The remainder of the paper is structured as follows: Section 2 discusses related work and emphasizes how our approach is different from earlier research. Section 3 outlines the design of our statistics collection framework. Section 4 presents the experimental evaluation of the proposed methods. Finally, Section 5 provides our conclusions and discusses our plans for future work.

2 RELATED WORK

Determining query cardinality is a classic problem in relational database systems. The seminal work on query optimization [43] describes how statistics can be used by the optimizer to calculate the predicate selectivity, which, in turn, allows an optimizer to infer the total query cardinality and choose an appropriate execution plan. In contrast to the query optimization problem, where statistical synopses are only used as an auxiliary structure, *approximate query processing* (AQP) systems [10, 11, 42] are using data summaries as a primary data source to provide approximate answers to ad-hoc exploratory queries. Cormode et al. [26] thoroughly surveys cardinality estimation methods and their application to the problems of query optimization and approximate query processing.

While calculating cardinality estimates, early systems made a number of assumptions (e.g., data uniformity, attribute independence) that were introduced to simplify the cost models. These strong assumptions, however, often led to approximation errors. Ioannidis et al. [34] showed that even slight estimation errors may lead to severe (several orders of magnitude) performance degradation, thus emphasizing the importance of estimation accuracy.

Poosala et al. [41] studied various types of histograms and provided a taxonomy and evaluation framework for different types of histogram-based synopses. They identified that the V-optimal and MaxDiff histograms provide superior accuracy compared to canonical equi-width or equi-height synopses. However, the increased accuracy comes at a price. The algorithms creating these more specialized histograms either are based on dynamic programming (V-optimal), hence have increased time complexity, or require multiple passes over the sorted data (MaxDiff), which can not be achieved in a streaming environment.

Matias et al. [38] proposed the first work that used wavelet-based synopsis for query cardinality estimation. Their approach relied on performing a wavelet decomposition over the input dataset and choosing the most significant coefficients to form a wavelet synopsis. Wavelet-based methods demonstrated substantial accuracy improvements while having other significant advantages over histograms like alleviating the curse of dimensionality and allowing for synopsis mergeability. Wavelets have been also successfully applied in dynamic synopsis maintenance [39], computing statistics over data streams [27] and approximate query processing [21].

Arguably the most popular approach to AQP is to sample the input data. Sampling is very robust and applies to a wide variety of queries. An approximate answer is obtained by applying a specifically designed estimator that evaluates the query over a sample and "scales up" the result in an unbiased manner to return a final answer. The simplest way of producing a uniform sample is a sequential scan, which has prohibitively large I/O. Alternatively one can pick only a subset of disk pages and then perform page-level sampling within those. This design needs careful tuning between keeping a sample uniform and the amount of random I/O required to produce it [24]. Gibbons et al.[29] proposed a way to maintain a sample for a dynamically evolving (in time) dataset. However, this mechanism requires allocating additional memory for a backing sample, which builds up memory pressure on local nodes that must simultaneously perform memory-intensive processing. Brown and Haas [18] introduced a sampling algorithm that can obtain samples in a partitioned environment, whereas Gemulla et al. [28] generalized it to process streams with insertions and deletions. However both of these approaches consider the case when partitions contain records with non-overlapping keys, which does not hold for systems based on the LSM storage abstraction. To the best of our knowledge there is no algorithm which solves the problem of producing unbiased samples in a setting similar to LSM storage.

Traditionally, databases rely on DBAs to manually launch a special *RUN ANALYZE* job that collects statistical data summaries. This approach remains still popular in various Big Data analytics systems, including Impala [17], HAWQ [23] and Hive [47]. Since these systems are targeting OLAP workloads, which tend to read all or a large part of the data, their statistics collection is based on sequential scanning and producing histogram-based synopses. On the other hand, systems that focus on a broader sets of use cases tend to rely on sampling and choose uniform samples as a statistical data summary [20, 36]. Some warehousing systems provide optimizations on top of their regular statistics-gathering method, such as automatically triggering the recomputation of statistics [6] or skipping recomputation if a non-significant number of records were modified [1]. To the best of our knowledge, there are no systems which are using the specific properties of LSM storage to do any kind of statistics computation.

An alternative, self-tuning workload-based approach that does not involve I/O operations to create a statistical summary has been followed by [9, 19, 44, 45]. These methods are based on analyzing the result cardinality of a given query workload and building histograms that rely solely on that feedback information. Histograms are consecutively refined as more queries are issued against a particular range of the dataset. While this approach introduces a low-overhead way of computing histograms, it heavily depends on the properties of the query workload and makes strong uniformity assumptions about "unexplored" ranges of the dataset.

The idea of using indexes for creating statistical synopses was first introduced by Barbara et al. [16] and is based on the observation that the upper levels of balanced indexes like B-Trees produce a bucketization of the value domain, thus essentially creating a hierarchical equi-height histogram. However, it was noted [14] that adopting an index for selectivity estimation would require storing additional entries in the index nodes. From a software engineering perspective, decoupling the synopsis data structure from the information stored in index pages allows statistics to be easily serialized and transported to a place where they can be consumed, which in a shared-nothing cluster-based environment is often a remote machine.

To summarize, our distinction from the related work lies in the fact that we are re-using the already-existing LSM dataflow to collect statistical summaries instead of building a separate I/O-intensive statistics collection pipeline. This allows us to cut back on additional I/O operations, yet without relying on a query workload as is typical in self-tuning approaches. However, this design restricts us to use only linear-time synopsis-collecting algorithms. We implemented histogram-based synopses and wavelets, but choose not to use sampling-based summaries because of the high memory costs associated with maintaining samples. Although our approach is based on using indexes, we are not altering their data structures, but are instead keeping synopses separate to mitigate the distributed workflow of collecting, storing, and consuming statistics.

3 AN LSM-BASED STATISTICS COLLECTION FRAMEWORK

We proceed with the design of the statistics collection framework. The main idea behind this design is to compute statistical synopses on-the-fly, piggybacking on the events of the LSM-framework. In our framework, statistics are always in sync with the underlying data because their computation is an ongoing part of the storage lifecycle. While doing this we ignore the statistics on the in-memory component because its size is relatively small with respect to an

overall persisted dataset. Eliminating the need for statistics recomputation also lifts the burden of determining statistics staleness and creates an "always on" user experience.

An overview of the LSM storage model appears in Appendix A. In our prototype implementation we adopted Apache AsterixDB since it uses the LSM model for storing both its main records as well as secondary indexes [13]. Its LSM-framework is created as a layer around the conventional implementations of various indexes, and thus provides a unified LSM-ified abstraction for all index types supported by the system (B-Tree, R-Tree and inverted indexes).

3.1 Local statistics collection

In order to achieve on-the-fly statistics-gathering, our approach should incur low overhead during data ingestion. The time complexity of synopsis-building algorithms is often dominated by sorting the records on attributes for which the statistics are to be computed. Given a tight latency budget, we restrict ourselves to only building synopses on primary keys (PK) or on secondary keys (SK) of index components. Disk operations in the LSM-framework can be generalized by a single *bulkload()* routine [13] that receives a stream of records $\mathcal{R} = r_1, ..., r_n$ ordered by $\langle PK \rangle$ in case of primary index components, or pairs $\langle SK, PK \rangle$ for secondary index components. We define our synopsis-computing algorithm as a function $\mathcal{S}(\mathcal{R})$ that computes a statistical summary of the aforementioned stream of records. While algorithms that compute comparison-based synopses (i.e. histograms) can operate on any totally ordered record stream, approaches based on hierarchical transformations (i.e. wavelets) require stream entries to be drawn from a fixed-size universe whose size is a power of 2. To provide a common ground for various synopsis-building algorithms we define the function $\mathcal{S}$ only over arguments of fixed-length integer numeric types (int8, int16, int32 and int64) supported by the AsterixDB data model [12]. Note that any value from a fixed-length domain could be padded with 0's to the length of the nearest power of 2, while variable-length types, e.g. strings, can leverage dictionary-encoding to reduce them to the former problem.

To validate our framework, in this paper we concentrate on one-dimensional synopses, thus calculating statistics only on B-Tree indexes with non-composite keys. However, all of the synopses-constructing algorithms that we implement potentially could be extended to multiple dimensions [49, 50]; we leave computing statistics on composite keys and spatial data as future work.

3.2 Streaming synopsis-building algorithms

In the context of this paper we describe implementations of the following synopses:

- Equi-width histograms
- Equi-height (aka equi-depth) histograms
- Wavelets

The construction algorithms each produce a synopsis with a predefined number of elements (bucket/coefficient budget) that is specified in the system's configuration file. An individual synopsis element is a single bucket, defined by its right border and the number of records that fell into that bucket, for histograms; it is a normalized wavelet decomposition coefficient, defined by the index in the error tree and its value, for wavelet-based synopsis. In both cases a synopsis element occupies the same amount of space, so we can directly and fairly compare the storage cost for different synopsis types.

The algorithm for creating an equi-width histogram is straightforward: first we calculate the histogram invariant — bucket width, depending of the total bucket budget and domain size of the indexed field. After that buckets can be populated left-to-right as the records are received from the sorted input stream. Building an equi-height histogram is done in a similar manner, but with the exception that it is parameterized with the total number of records in the input stream to calculate its invariant — bucket height. In case of the LSM-flush operations, the total number can be easily obtained by keeping a counter for the records in the flushing component; for LSM-merge it is composed of the number of records in the merged components, while a bulkload receives this information from a sort operator at the bottom of the execution plan.

Unlike the trivial algorithms which compute histogram-based synopses, producing a wavelet requires to perform a wavelet decomposition on the input data. This algorithm is well-known – and is primarily used in signal and image processing. The interested reader can find a detailed example of the wavelet decomposition process in Appendix B. The classical version of this algorithm requires allocating large arrays containing partial results of the decomposition process. This overhead is often neglected when wavelets are used for image or signal processing, as the maximum resolution of the images or signals is on the order of thousands. In contrast, when used for tuple frequency estimation for a large domain, e.g., 64-bit wide integers, this approach will quickly run into space problems. On top of that, in cardinality estimation the input frequency signal is often sparse, so the allocated arrays will largely consist of zeros; this results in wasted CPU-cycles during the decomposition process. Both of these issues presented an opportunity to optimize the computation of wavelet decomposition in our setting.

To avoid sparsity in the incoming data, instead of using "raw" tuple frequencies we compute on-the-fly prefix sum of the input signal (i.e. convert it to a one-dimensional datacube [48]). Our preliminary experiments showed that using a "dense" prefix sum as an input for the wavelet decomposition significantly improves the accuracy of range-sum queries. A streaming version of the discrete Haar wavelet decomposition algorithm that avoids excessive memory allocation was first proposed by Gilbert et al. [30]. The algorithm uses a priority queue to store the B most significant coefficients and an axillary array of *logN straddling* coefficients, which are used to track the current root-to-leaf path in the error tree. However this algorithm is restricted to work only on a "raw" data or requires an additional pass to precompute a prefix sum. Note that in a latter case a significant overhead will be caused not only by scanning the input twice, but by running the wavelet decomposition for each entry in the datacube, which is proportional to the domain length.

Algorithm 1 builds on the approach by Gilbert et al. [30] and presents a streaming version of the discrete Haar wavelet decomposition algorithm that encodes a "dense" prefix sum signal. For simplicity it omits the coefficient normalization, but we perform all appropriate transformations in our implementation. The algorithm keeps a bounded priority queue, but replaces a fixed-size array of straddling coefficients with a stack to store the average coefficients on different levels. The main loop of the algorithm repeatedly calls

Algorithm 1 Streaming wavelet decomposition algorithm

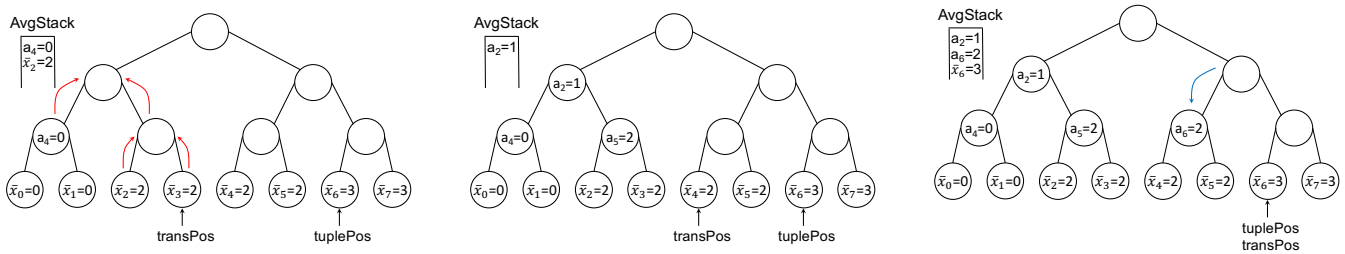
```
1:  procedure waveletTransform(stream)
2:      prefix ← 0, tuplePos ← 0
3:      avgStack ← emptyStack(), priorityQueue ← emptyQueue()
4:      for all tuple in stream do
5:          transformTuple(tuple.pos,prefix,tuple.value)
6:          prefix ← prefix + tuple.value
7:      if lastTuple.pos ≠ domainEnd then
8:          transformTuple(domainEnd,prefix,0)
9:      priorityQueue.add(avgStack.pop())
10:     waveletSynopsis ← priorityQueue.items
11:     return createBinaryPreOrder(waveletSynopsis)
12:
13: procedure transformTuple(tuplePos,prefix,tupleVal)
14:     transPos ← getTransformPos(avgStack.peek())
15:     calcDyadicIntervals(tuplePos,transPos,prefix)
16:     pushToStack(newCoeff(tuplePos,0,prefix + tupleVal))
17:
18: function getTransformPos(coeff)
19:     if coeff.level < 0 then
20:         return domainStart
21:     else if coeff.level == 0 then
22:         return coeff.key + 1
23:     else
24:         return ((coeff.key + 1) << (coeff.level))−
                    1 << (maxLevel − 1)
25:
26: procedure calcDyadicIntervals(tuplePos, transPos, prefix)
27:     while tuplePos != transPos do
28:         topCoeff ← avgStack.peek()
29:         dyadicCoeff ← newCoeff(topCoeff.key + 1,topCoeff.level,prefix)
30:         while dyadicCoeff.covers(tuplePos) do
31:             dyadicCoeff ← newCoeff(topCoeff.key * 2 + 1,dyadicCoeff.level - 1,prefix)
            pushToStack(dyadicCoeff)
32:         transPos ← getTransformPos(dyadicCoeff)
33:
34: procedure pushToStack(newCoeff)
35:     while !avgStack.isEmpty && avgStack.peek().level == newCoeff.level do
36:         topCoeff ← avgStack.pop()
37:         newCoeff ← average(newCoeff,topCoeff)
38:     avgStack.push(newCoeff)
39:
40: function average(coeff1,coeff2)
41:     avgCoeff.key ← coeff1.key >> 1
42:     avgCoeff.level ← coeff1.level + 1
43:     avgCoeff.value ← (coeff1.value + coeff2.value)/2
44:     detailCoeff ← avgCoeff
45:     detailCoeff.value ← (coeff1.value - coeff2.value)/2
46:     priorityQueue.add(detailCoeff)
47:     return avgCoeff
```

the transformTuple procedure for each tuple from the incoming stream while simultaneously calculating a prefix sum of tuple values. After the whole stream is consumed this procedure is called once more to compute the total average, unless the last processed tuple's position is the end of the value domain (line 8). Because the main average is also a valid wavelet coefficient, it is added to the priority queue along with all detail coefficients. Finally, we create a wavelet synopsis based on all coefficients left in the priority queue, and reorder them using a binary tree *pre-order* that would allow to efficiently answer range-sum queries.

The gist of the streaming transform algorithm lies in its transformTuple procedure (line 13) that performs an individual step of the transform. The procedure first calls the function getTransformPos to determine the point in the domain where the wavelet transform has currently stopped. This position is determined by examining the top coefficient on the stack and saved into *transPos* (line 14).

Due to the sparsity of the incoming signal there can be a "gap" between the current transform position *transPos* and the position of the processed tuple *tuplePos*. Figure 1 shows an example of processing the gap between tuples $x_2 = 2$ and $x_6 = 1$ from the input sequence $X = [0\ 0\ 2\ 0\ 0\ 0\ 1\ 0]$. Because we are performing a transform over the *prefix sum* of the signal (i.e. $\overline{X} = [0\ 0\ 2\ 2\ 2\ 2\ 3\ 3]$), this gap must be "filled" with appropriate wavelet coefficients so that the sum of the subtree values under these coefficients adds up to a current prefix (i.e. $\overline{x}_2 = 2$). In the wavelet transform each coefficient represents a *dyadic interval* (i.e., interval $[k * 2^{(\log \mathcal{D})-l}; (k+1) * 2^{(\log \mathcal{D})-l} - 1]$ for $k = 0, ..., 2^l - 1$) on some resolution level $l = 0, ..., \log \mathcal{D}$ in the value domain, where $\mathcal{D}$ is the length of the domain. The process of filling the gap is analogous to representing it as a series of non-overlapping dyadic intervals where the beginning of one interval is the end of the previous. The procedure calcDyadicIntervals computes this set of intervals in a greedy approach: it starts with *dyadicCoeff* that is a sibling (i.e., has the same level, but the coefficient's key is incremented by 1) of the current top of the *avgStack*. We repeatedly try a smaller *dyadicCoeff* until its support interval stops covering the current tuple's position *tuplePos*. This process is pictured in Figure 1c where a newly added sibling coefficient is converted to $a_6 = 2$ (blue arrow). On each iteration of the loop (line 30) the sibling coefficient's support interval in decreased by 2 while its value is multiplied by 2. After that, the calculated *dyadicCoeff* is pushed to the *avgStack* and the position *transPos* is "advanced" according to the new coefficient on the top of the stack (line 32).

Note that the *avgStack* has an important property: its coefficients appear in strictly descending order of their levels because they represent current averages on each level of the transform. Pushing a new coefficient might violate this invariant. When this happens the algorithm pops the current top of the stack *topCoeff*, calculates the average between *topCoeff* and a new coefficient *newCoeff*. The process is repeated until the stack constraint is no longer violated, possibly triggering a "domino" effect. In the end the averaged coefficient is pushed on to the stack. Figures 1a and 1b depict exactly this situation when pushing tuple $x_3 = 2$ to the stack leads to a series of averaging (showed as red arrows) and puts the final average $a_2 = 1$ on the top of the *avgStack*. The average procedure calculates and



(a) State after adding $\overline{x}_3 = 2$. **(b) State after cascade averaging** **(c) State after adding $\overline{x}_6 = 3$**

Figure 1: Example of the Algorithm 1 executing on the input $X = [0\ 0\ 2\ 0\ 0\ 0\ 1\ 0]$. Figure shows intermediate steps in filling the gap between tuples $x_2 = 2$ and $x_6 = 1$ (i.e. adding tuples $\overline{x}_3 = \overline{x}_4 = \overline{x}_5 = 2$ and $\overline{x}_6 = 3$ to the prefix sum transform). (a) illustrates the state after entry $\overline{x}_3 = 2$ is pushed to the stack; (b) shows the result of averaging, triggered by pushing coefficient on to the stack; (c) depicts the step after covering interval $[\overline{x}_4; \overline{x}_5]$ with coefficient $a_6 = 2$ and adding final tuple $\overline{x}_6 = 3$.

returns an average between two input coefficients while saving a detail coefficient in a priority queue. The input coefficients should always have the same *level* and the calculated average coefficient's level is the children's level + 1.

After the gap is filled with dyadic intervals and *transPos* is equal to *tuplePos*, a new coefficient is pushed on the top of the *avgStack* (line 16 and Figure 1c). Because this new coefficient represents a new item on the bottommost level of the error tree it has $level = 0$ and value *prefix+tupleVal*.

3.3 Incorporating anti-matter into statistics

A distinctive characteristic of the LSM-based storage is that newer components can potentially have anti-matter records that offset records in earlier immutable disk components. While there is work on dynamically maintaining histogram and wavelet synopses [29, 39], we chose to address this issue in a synopsis-agnostic way by keeping a separate and explicit "anti"-synopsis. This data summary contains statistics on all anti-matter records that were encountered in the input stream. Moreover, this generic approach allows us to easily handle the case when a distribution of anti-matter records is significantly different from the distribution of regular entries.

When computing the total estimate E we issue a query to both the regular synopsis $\mathcal{S}$ and its "anti"-twin $\overline{\mathcal{S}}$, which gives estimates $E_{\mathcal{S}}$ and $E_{\overline{\mathcal{S}}}$ respectively. The total cardinality is then reported as $E_{\mathcal{S}} - E_{\overline{\mathcal{S}}}$.

3.4 Collecting statistics in a distributed cluster

Zooming out from the local statistics computed at each node, we now consider the process of obtaining statistics in a distributed cluster. AsterixDB uses the popular shared-nothing design, where a single master node coordinates job scheduling and the execution of queries on a set of slave nodes. Each LSM-framework event creates a local synopsis which is sent over the network to the master node; synopsis is persisted in the *system catalog*, so that it can be used during query optimization. In order to prevent rapid catalog growth, statistics from different nodes could be merged together; however as Section 3.5 discusses, not every synopsis type can be combined and, moreover, there is an inherent trade-off between the space occupied by a synopsis and its accuracy.

3.5 Synopsis mergeablity

The proposed statistics-collection framework benefits greatly from piggybacking on LSM lifecycle events, but this also creates an additional challenge when it comes to extracting estimates from statistical synopses. In this design, each individual synopsis captures the statistics only for the records in a particular flushed/merged/bulk-loaded component. So, after some ingestion, the algorithm ends up creating multiple synopses, each summarizing only a part of the overall data distribution. Moreover, this partialness is exacerbated by the fact that statistics-gathering is executed in a distributed system where each node computes statistics only for the subset of data stored on a particular machine. Thus, estimates from various synopses should be combined together to get the overall result. Alternatively, querying each synopsis separately will create an overhead during query optimization which could take a significant portion of the total runtime for short queries.

Given these restrictions, it seems more desirable to combine separate statistics into a single synopsis and use it later for cardinality estimation. However, not all types of synopses presented in Section 3.2 can be easily merged. For example, equi-height histograms cannot be combined due to their varying bucket borders. At the same time, wavelets allow merging, but lose some accuracy along the way due to the thresholding process. Finally, equi-width histograms can naturally be combined.

Since statistics are saved in the system catalog, the amount of space occupied by the metadata can become another factor that we should consider while building the statistics collecting framework. Because of the approximate nature of creating synopses, there is an inherent data loss associated with this process. If we consider two synopses $\mathcal{A}$ and $\mathcal{B}$, in the general case an estimate $E_{\mathcal{A}} + E_{\mathcal{B}}$ calculated from treating these synopses separately has a greater accuracy than an estimate $E_{\mathcal{A}\oplus\mathcal{B}}$ obtained from a combined synopsis (where $\oplus$ designates a synopsis merge operation). Thus, there is a natural trade-off between the total space allocated for statistics on a particular dataset and the estimation accuracy. Since we are primarily focused on using statistics for query optimization, where a slight mis-estimation could lead to significant errors [34], we choose to keep all statistics, even mergeable ones, as separate entries in the catalog.

Algorithm 2 Algorithm for computing total cardinality estimate of a range query for a particular attribute

```
1:  function rangeQueryEstimate(queryAttribute,range)
2:      total_estimate ← 0
3:      if mergeable then
4:          merged_synopsis ← retrieveFromCache()
5:          merged_anti_synopsis ← retrieveFromCache()
6:          if isStale(merged_synopsis) && isStale(merged_anti_synopsis) then
7:              merged_synopsis ← NULL
8:              merged_anti_synopsis ← NULL
9:          else
10:             return getEstimate(merged_synopsis,range) - getEstimate(merged_antimatter_synopsis,range)
11:     for all synopsis in getSynopses(queryAttribute) do
12:         estimate ← getEstimate(synopsis,range)
13:         if synopsis is anti-matter then
14:             estimate ← estimate ∗ (−1)
15:         total_estimate ← total_estimate + estimate
16:         if mergeable then
17:             if synopsis is anti-matter then
18:                 Merge(merged_anti_synopsis, synopsis)
19:             else
20:                 Merge(merged_synopsis, synopsis)
21:     if mergeable then
22:         Cache(merged_synopsis)
23:         Cache(merged_anti_synopsis)
24:     return total_estimate
```

Maintaining synopses separately is a valid approach to manage statistics on the Cluster Controller when we want to obtain an aggregate cardinality estimate. However, when computing local statistics during an LSM-merges we choose to create new synopses from scratch directly on the newly merged component, discarding earlier statistics altogether. This alleviates the propagation of estimation errors during a long chain of merge operations, where a multiplier effect could be triggered. In addition, this decision does not change any of our streaming algorithms, as the input stream created by a merge cursor provides a unified sorted record stream abstraction over the individual record streams of merged components. Lastly, this enables a universal method of creating statistics during an LSM-merge, given that not all synopsis types are inherently mergeable (e.g., equi-height histograms).

To amortize the cost of computing estimates during query optimization, we periodically merge appropriate synopses (i.e., wavelets and equi-width histograms) and cache the produced synopsis on the Cluster Controller side where the query rewriter can access them. Similar to the case when merging components, we recompute a whole combined synopsis whenever a new piece of statistics is received from a storage node rather than maintaining it incrementally, and we invalidate the previous merged version at that time.

3.6 Estimating query cardinality

Once the synopses are computed, transferred over the network, and persisted in the catalog, they are available to drive query optimizer decisions. Since our statistics-collection algorithm relies on having a B-Tree index on a particular field f, we focus here on estimating the cardinality of range queries which could potentially use this index, i.e. queries Q like

```
SELECT * FROM T
WHERE T.f >= x AND T.f <= y
```

Cardinality information and computed estimates could be used in the following scenarios during the query optimization process:

(1) Skipping low selectivity index probes
(2) Deciding whether to use an indexed nested-loop join

Algorithm 2 describes how we compute the total cardinality estimate for a given range query. Procedure getSynopses retrieves all synopses for a particular query attribute from a system catalog (line 11). The main loop of the algorithm uses these synopses to compute *total_estimate*, by combining estimates of each individual component. An estimate produced by a component's synopsis is simply added to the total estimate, unless it comes from an anti-matter synopsis, in which case it is substracted. While calculating the total estimate the algorithm also computes a merged synopsis for equi-width histograms and wavelets (line 16). In the end, procedure Cache saves a merged version of the synopses on the Cluster Controller. Queries can then obtain it from cache using procedure retrieveFromCache (line 4) and can thus skip fetching statistics from the catalog. Procedure isStale compares timestamps of retrieved synopses (both regular and anti-matter) and invalidates them if they are stale; otherwise it obtains the estimate directly from the merged synopsis (line 10).

For histogram-based synopses, the *getEstimate*() trivially returns the sum of all buckets that are located between borders $[x; y]$ of the range. For partially overlapped buckets we use a *continuous-value assumption* which expects that the values within a bucket have a uniform distribution. For a wavelet-based synopsis *getEstimate*() obtains a cardinality estimate for query Q by reconstructing the wavelet's value at two border points: $E_Q = W_y - W_x$. Due to construction the wavelet signal at a given point p stores a *prefix sum* of the records' frequencies, rather than their raw frequencies: $W_p = \sum_{i=0}^{p} f(i)$, where $f(i)$ is a raw tuple frequency. Reconstructing value W_p does not require to perform a full wavelet decomposition in reverse order, but instead is computed by a single root-to-leaf path traversal in the error tree corresponding to this wavelet.

4 EXPERIMENTAL EVALUATION

4.1 Experimental setup

In the following section we experimentally evaluate the implementation of our statistics framework from the perspectives of (i) the overhead caused by the statistics collection algorithms, and (ii) the accuracy of the produced statistics. We ran all experiments using a modified version of AsterixDB v0.9.1 on a small cluster with 4+1 nodes (slaves+master), connected by a Gigabit Ethernet network, each running CentOS Linux. Each machine is equipped with a dual-core AMD Opteron CPU, 8 GB of main memory, and two 1 TB

drives. All NCs have two data partitions to leverage I/O parallelism, thus creating a cluster with 8 partitions.

The experimental pipeline consists of several disjoint stages:

- Preparatory data definition language (DDL) statements for creating types, datasets and indexes.
- Data ingestion, during which data is loaded into the system and statistics are collected as a by-product of the LSM-based loading process.
- Querying the loaded data and measuring the accuracy of the resulting cardinality estimates.

4.1.1 Datasets. To evaluate various data distributions we adopted the experimental framework proposed by Poosala et al. [41]. This framework describes a synthetic data distribution used for query cardinality evaluation in terms of two independent parameters:

- Frequency set: a set of numbers, where each defines the *number* of records having a particular value of the secondary key.
- Value set: a set of numbers, where each defines the *position* of secondary keys in the key domain (e.g., 32-bit wide integers). The domain distance between two neighboring values is called the value set's **spread**.

In our experimental evaluation we considered several synthetic spread distributions, which in turn define value sets for our datasets:

- Uniform: All spreads have the same length, calculated from the total domain length and the number of generated values.
- Zipf: Spreads have skewed lengths drawn from a Zipfian distribution with skew coefficient $\alpha = 1$ and ordered in a decreasing manner.
- ZipfIncreasing: Same as above, but ordered from shortest to largest spread.
- ZipfRandom: Same as above, but randomly ordered.
- CuspMin: Two-sided skewed distribution where the first half of the values follow a Zipf distribution and the second half follow a Zipf Increasing distribution.
- CuspMax: Same as above, but the first half obeys a Zipf Increasing distribution, while the second half follows a regular Zipf distribution.

The values for the frequency set were obtained similarly from the Uniform, Zipf and ZipfRandom distributions.

Note that since the contents of the frequency and value sets are independent, one could consider all possible correlations between them, e.g., positive (the first record in the value set corresponds to the first entry in the frequency set), negative, and random. However, given that some of the value set distributions are inherently symmetric (Uniform, CuspMax, CuspMin) and some are mirror image of each other (Zipf and ZipfIncreasing) we did not find a significant difference between positive and negative correlations. On the other hand, random correlation (irrespectively from value set distribution) produces results similar to positively correlated ZipfRandom. Given the space limitations and similarity of these results we present data only for the positive correlation between the frequency and value sets.

To evaluate the performance of data ingestion, we used a built-in feature of AsterixDB called *data feeds* [32] that allowed us to create a continuous channel though which records are inserted into

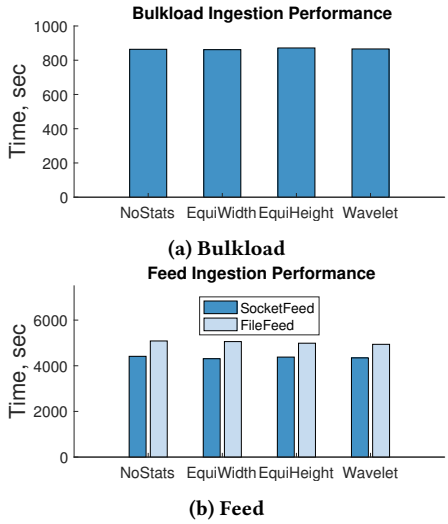


Figure 2: Total execution time of ingesting 50M records (a) using a bulkload operation, which produces a single component, and (b) through a continuous data feed channel, which creates multiple LSM components. Both experiments are performed for 3 types of synopses (equi-width histograms, equi-height histograms and wavelets) and for a baseline case when the statistics collection is turned off (NoStats).

the system. In the synthetic experiments we emulated a Twitter Firehose-like external data source to ingest generated records resembling real Tweets. We utilized two types of data feed sources: a push-based feed that uses a TCP socket and a file feed with a pull-based model that reads records from local files. The size of each generated record was around 1 KB, while each of the generated datasets contained 50 million records. In addition to the regular tweet fields (such as username, message, location, etc) each record was augmented with a special integer field with value that was drawn randomly from the synthetic distributions described above. To enable statistics gathering for this field, we have defined a secondary B-Tree index on it.

Finally, in addition to synthetically generated data we experimented with a real-life dataset consisting of web server log entries collected during World Cup 1998 [15]. The dataset contains 1.35 billion preprocessed 20-byte records, each containing four 32-bit integer fields and four 8-bit byte fields. After excluding fields where almost all the values are duplicates (i.e. fields *method* and *type*) we created a secondary index for each of the remaining fields.

4.1.2 Query workload. To evaluate the accuracy of the proposed framework we experimented with several types of range queries:

- Fixed length: These are range queries with a predefined distance between the starting and ending points. The starting point position is drawn randomly from the value domain.
- Half open: Range queries where one of the borders of the range, e.g., the starting point (ending point respectively), is drawn randomly, while the other is the maximum (minimum respectively) point in the domain.

- Random: Range queries where both the starting point and the ending point are drawn randomly from the domain.
- Point: Degenerate range queries where the starting and ending points are the same randomly drawn domain point.

For the accuracy experiments we executed 1000 queries of a particular type, recorded their true cardinality C, and computed the statistical estimate $\widehat{C}$. For each query we calculated the *absolute error* and normalized it by dividing it by the total number of records in the dataset $N = 50M$: $e^{abs} = \frac{|C-\widehat{C}|}{N}$. To compute the final accuracy across all queries we used the L1 (average) metric: $\sum_{i=1}^{1000} \frac{e_i^{abs}}{1000}$.

4.2 Overhead Evaluation Experiments

To determine the overhead introduced by collecting and storing the statistics, we have measured the execution time of the ingestion stage of the experimental pipeline for all three statistical synopses types and, as a control case, for a configuration where no statistics are captured. Each measurement was repeated 3 times and the average value is reported.

In AsterixDB data can be persisted in two different ways: by bulkloading a dataset upfront or by performing DML insert/update/delete statements. Figure 2a presents the bulkloading execution times for the case when EquiHeight, EquiWidth histograms or Wavelets synopses are produced as the bulkloading is performed, and for the baseline case when no synopses are calculated. During bulkload the dataset is populated in a bottom-up fashion, producing a single large LSM component, so it does not utilize all the possible events of the LSM lifecycle. To isolate the effect of statistics-gathering in this experiment we were using pre-sorted datasets, so the bulkloading process included only partitioning and building an upper level of a B-Tree index (thus excluding expensive external sorting which is not involved in computing statistics). Bulkloading was done in a partitioned parallel manner on all 4 cluster nodes to minimize the load time.

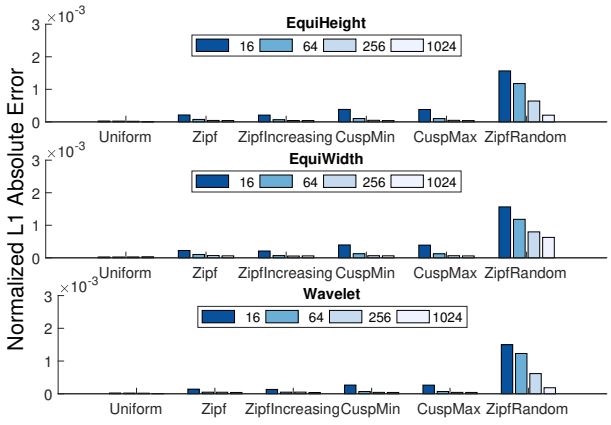
Figure 2b shows the case when a data feed is used to populate the dataset. A feed populates the dataset's storage structure incrementally, in a top-down manner, thus triggering the full spectrum of LSM lifecycle events. We experimented with 2 different types of feeds available in AsterixDB: a socket-based feed, where the records were received via a network socket from an external source, and a file-based feed, where the source of the records were local files.

For both graphs in Figure 2, the values for different synopsis types vary slightly due to measurement error; however there is no significant overhead introduced by any of the statistics-gathering algorithms, as compared to the baseline case when synopses are not produced. The same results were observed with the real-world WorldCup dataset. These allow us to affirm that the proposed statistics-collection framework indeed does not interfere with the normal LSM-based storage workflow in AsterixDB.

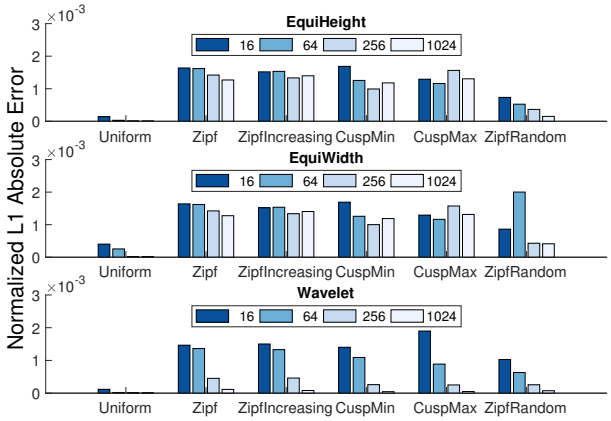
4.3 Accuracy Evaluation Experiments

Due to the large size of the parameter space, we began our evaluation by fixing some of the variables for the experimental procedure.

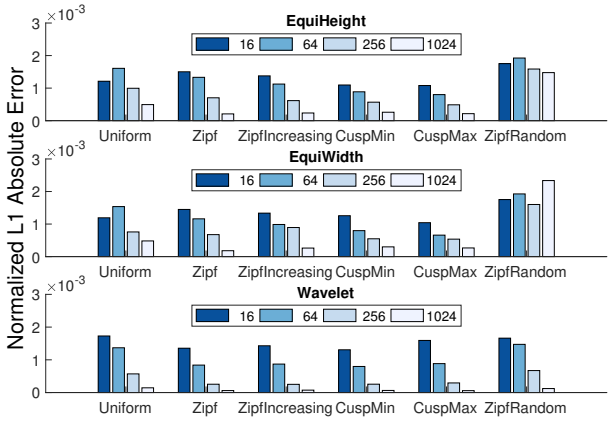
4.3.1 Varying the synopsis size. Figure 3 depicts the accuracy results while we increase the size of the synopsis for a FixedLength



(a) Dataset with Uniform frequencies



(b) Dataset with Zipf frequencies



(c) Dataset with ZipfRandom frequencies

Figure 3: Estimation accuracy results, while varying the size of the synopsis for datasets with (a) Uniform, (b) Zipf and (c) ZipfRandom frequency distributions. Submitted queries have a fixed range length of 128. The sizes of synopses are increased from 16 to 1024 elements.

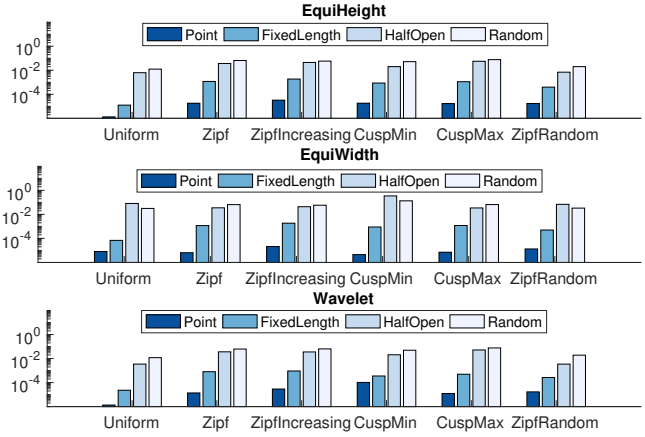


Figure 4: Estimation accuracy results for 4 different types of queries and a dataset with Zipf frequencies.

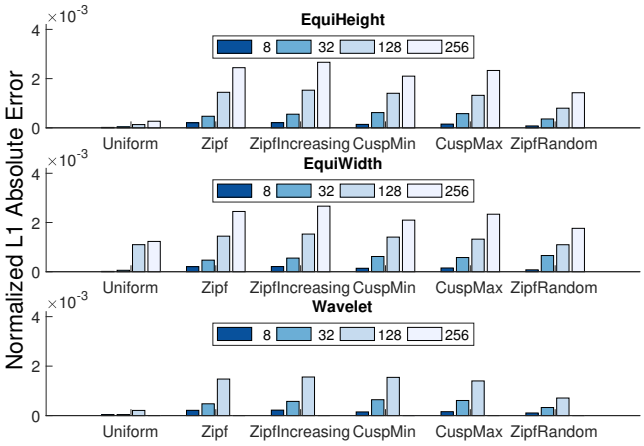


Figure 5: Estimation accuracy results for FixedLength queries and a dataset with Zipf frequencies for varying query sizes.

query workload with range length of 128. In the case of the histogram synopses, we vary the number of histogram buckets, whereas for wavelets the number of wavelet coefficients is increased. Note that the amount of storage allocated to synopses is the same in both cases, because the space taken by one histogram bucket is the same as the space allocated for a single wavelet coefficient. In this experiment we have increased the size of synopses from 16 to 1024 elements (buckets or wavelet coefficients).

Figure 3a shows that estimation errors for datasets with Uniform frequencies are close to 0 in all of the cases except for the ZipfRandom spread distribution. The same result can be seen for Uniform spreads in Figure 3b. In all of these cases, the data distribution creates a smooth CDF that can be easily estimated even with a small number of synopsis elements.

In contrast, in all the other cases in Figures 3b and 3c, as well as the ZipfRandom spread distribution in Figure 3a, the random permutation of spreads creates distributions with much more complex CDFs and makes estimation more complicated. Generally, for these we see a common trend that the cardinality estimation error is negatively correlated with the synopsis size. However, there are few exceptions for the histogram-based synopses, namely the Zipf, ZipfIncreasing, CuspMin and CuspMax datasets, where increasing the synopsis size does not significantly improve their accuracy. The poor histogram accuracy on these datasets can be explained by the fact that the dataset is skewed: in the Zipf distribution, some of the frequencies are so large that they exceed the height of the equi-height's histogram bucket. In contrast, wavelets demonstrate the expected behavior, supporting previous research findings that wavelets on average provide better accuracy [38].

Among these results, the synopsis with 256 elements provides excellent accuracy, so we will fix this parameter throughout the rest of the evaluation section.

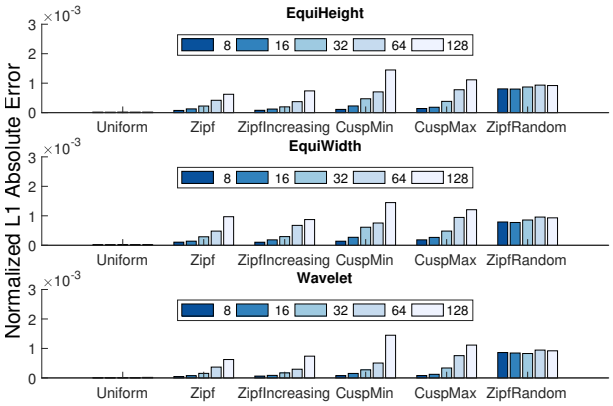
4.3.2 Varying the query type. We proceed with exploring how the properties of the query workload influence estimation accuracy. Again, we present the datasets with Zipfian frequencies here, but the data drawn from other distributions behaved similarly.

Figure 4 shows the accuracy for all query types mentioned in Section 4.1.2. We can see that, for all distributions, Point queries produce smaller errors than FixedLength queries, which in turn are smaller than HalfOpen and Random queries. Note that the error scale on this graph is logarithmic to emphasize that larger queries introduce noticeably larger than those elsewhere in our results. This can be explained by the fact that the number of tuples that fall into a wider range now represent a larger fraction of the total dataset and L1 absolute error metric emphasizes that difference.

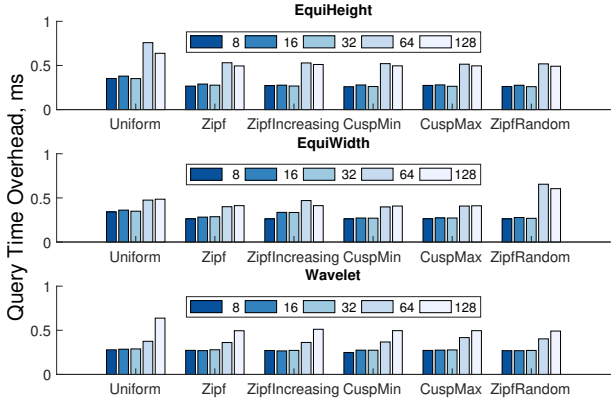
Figure 5 presents similar measurements, but specifically for fixed-length queries with the length parameter varied from 8 to 256. We can see that the trend is preserved in this experiment, as the error keeps growing as the query range is increasing. Because fixed-length queries allow us to increase the number of returned tuples in a controllable manner, we will use them with the range size of 128 as the query type of choice for the following experiments.

4.3.3 Varying the number of LSM components. To study how the number of individual synopses affects the overall estimation accuracy, we now control the number of LSM components produced during the data insertion stage by utilizing the Constant LSM merge policy that is available in AsterixDB. As its name implies, this merge policy allows one to have only a predefined number of LSM disk components per partition across a cluster. We also alter the individual synopsis size here, with respect to the increasing number of produced components, so that the total space allocated for statistics remains the same.

Figure 6a depicts the accuracy measurements, while Figure 6b considers the overhead during the querying stage of the experiment for various synopses types, spread distributions and component numbers. As we can see, increasing the total number of components does slightly increase the estimation error, as each component's synopsis contains fewer elements, inevitably deteriorating the estimation performance. At the same time, the overhead during query optimization increases, but not significantly.



(a) Accuracy for varying number of LSM components



(b) Query time overhead for varying number of LSM components

Figure 6: Experiments for 3 types of synopses, while varying the total number of produced LSM components for a dataset with Uniform frequency distribution. The number of components is increased from 8 to 128. (a) depicts the normalized L1 absolute error, whereas (b) shows the query optimization overhead of obtaining statistics during the same experiment.

4.3.4 Workload with varying percentage of anti-matter. In all previous experiments, we considered cases where the input data workload was insert-only. In the next set of graphs we study how the estimation accuracy changes if we add updates and deletes into the ingested mix to trigger anti-matter records. For this purpose, we used a special kind of AsterixDB data feed, called a *changeable feed*, which allowed us to mark incoming data records so that they will perform a regular insert, update an already inserted record, or delete an existing record. To make sure that the updates and deletes do actually generate anti-matter, as opposed to their just being silently deleted (from the perspective of statistics) within in-memory components, we broke the ingestion process up into stages. As each stage is processed, we forced a flush operation, which puts all previously ingested records on disk. After that, all updates and deletes that reference records in the previously processed stages will generate anti-matter records. Note that because AsterixDB enforces update and delete constraints (i.e., it does not allow updating or deleting a record unless it already exists), the maximum ratio of each operation type in the insert/update/delete mix cannot exceed 0.33 (assuming that each record is updated only once).

Figure 7 illustrates the accuracy measurements for the dataset with ZipfRandom frequencies, while the ratio of the updates and deletes in the data workload is gradually increased from 0 to 0.3. We observe that increasing the fraction of anti-matter records does not degrade the accuracy of cardinality estimation for all types of synopses. This demonstrates that our approach for dealing with anti-matter records, which is based on persisting and querying them separately, performs well. Note that the approach also provides a synopsis-agnostic way (in contrast to specialized maintenance algorithms) of dealing with changeable workloads while increasing the synopsis storage cost by only a constant factor (of 2).

4.3.5 Synopsis mergeability. Figure 8 further studies how the number of LSM components influences the query optimization

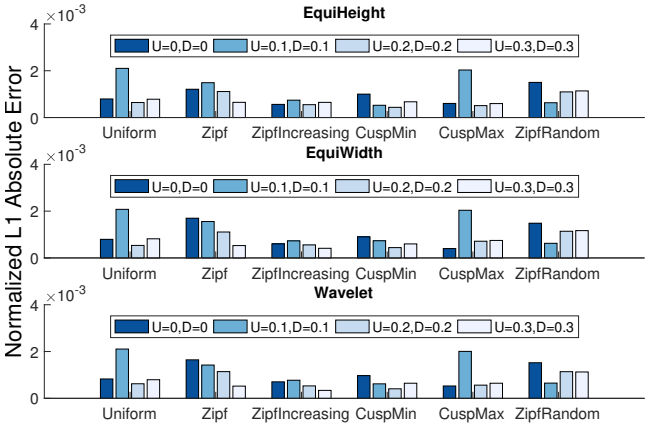


Figure 7: Estimation accuracy results for the workload with varying ratio of updates and inserts. Results obtained for various types of synopses on a dataset with ZipfRandom frequency distribution. The ratio of updates (U) and deletes (D) is scaled from 0 to 0.3.

time overhead of cardinality estimation. In this experiment we performed ingestion in two different ways: using a bulkload, which is guaranteed to create a single LSM component, and using feed-based ingestion with a NoMerge policy that leads to a maximum possible number of components.

The figure shows the results for a dataset with Zipf frequencies, but the results for other distributions are similar. It can be seen that query time overhead for the NoMerge policy is consistently higher than the same results for the bulkload-based workload. However, we note that this time difference is negligible between all types of

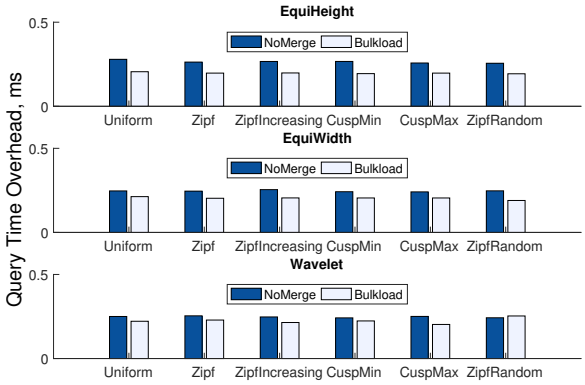


Figure 8: Query time overhead results for the the case when a dataset is bulkloaded, creating a single LSM component, vs. a workload with the NoMerge policy, creating a maximum number of components.

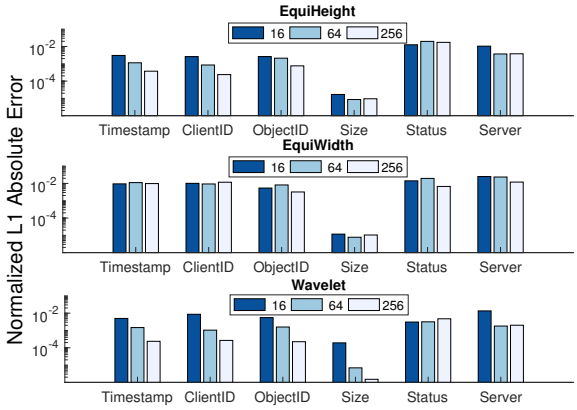


Figure 9: Estimation accuracy results, for all types of synopses for 6 fields from the WorldCup dataset. The sizes of the synopses are increased from 16 to 256 elements.

data synopses. These results highlight the fact that the mergeability of a particular synopsis type has a more profound effect on the total space allocated to the synopses rather than on the query time.

4.4 WordCup Dataset Experiments

Figure 9 depicts the results of the accuracy experiment for the WordCup dataset. In this experiment we have used feed-based ingestion using the Constant LSM merge policy with the default number of components (5). We have used range queries for each of 6 examined fields in the dataset. The length of the range for each query was equal to 1% of the range for a particular field (i.e. the difference between the maximum and minimum values of that field). Unlike the earlier experiments where the values were spread throughout the whole field domain, in real-world data, values are typically placed away from the domain extremes. This property explains why the accuracy of the EquiWidth histograms does not improve as we allocate more buckets. In fact, for fields *Timestamp*,*ClientID* and *ObjectID* all values fell into a single bucket. In contrast, EquiHeight histograms and Wavelets were able to dynamically adjust to the distribution of values in a particular field. Moreover, wavelets tend to be 5-10 times more accurate.

Field *Size* presents an interesting example of highly skewed data with a long distribution tail. We can observe that wavelets represent such distributions significantly better given enough coefficients. Finally, fields *Status* and *Server* represent categorical data. The distribution of the values in these fields has a lot of "spikes" separated by values with zero cardinality. Because all synopses estimate the values relying on proximity-based similarity, this leads to vast over/under-estimation errors.

In summary, the real world data experiments show that in certain cases the Wavelets and EquiHeight histograms are more robust, being less susceptible to changes in the input data.

5 CONCLUSIONS AND FUTURE WORK

In this paper we have proposed a novel lightweight approach to collecting statistics that exploits the properties of LSM-based storage to obtain statistical data summaries of the underlying data. Our solution is integrated into the common operations of the LSM framework and thus allows us to natively and inexpensively keep statistics in sync with rapidly changing data. We have implemented 3 different types of synopses (equi-width histograms, equi-height histograms, and wavelets) and developed efficient approaches to compute them during LSM lifecycle events. We have shown experimentally that computing these data synopses introduces a negligible runtime overhead, both during the ingestion, when data summaries are created, and during query optimization, when cardinality estimates are obtained. We have also performed an extensive experimental evaluation of the synopses' cardinality estimation errors using various data distributions, query workloads, and merge policy parameters. Our experiments have shown that our design provides good accuracy in a broad number of cases.

As future work we plan to extend the proposed statistics-collection approach to other value domains (i.e., not just integer numerics) as well as to multidimensional index types (e.g., B-Trees with composite keys and R-Trees). Another potential direction is to relax the condition of relying on a sorted order already provided by a primary or secondary index in order to compute statistics on arbitrary record attributes. Methods based on sketches [31] seem to be a promising data summary variant for this scenario. Finally, we would like to explore sampling-based statistics-collection methods and assess their accuracy and runtime overhead in comparison to precomputed synopses. Sampling tuples from base relations provides significant advantages over field-level statistics because it does not rely on calculating synopses only for preselected attributes, thus allows to correctly estimate cardinality for correlated data.

ACKNOWLEDGMENTS

This research was partially supported by National Science Foundation grants CNS-1305253, CNS-1305430, III-1447826 and III-1447720.

REFERENCES

[1] 2017. Amazon Redshift. Database Developer Guide. Analyzing Tables. (July 2017). Retrieved July 13, 2017 from http://docs.aws.amazon.com/redshift/latest/dg/t_Analyzing_tables.html

[2] 2017. Apache AsterixDB. (July 2017). Retrieved July 13, 2017 from http://asterixdb.apache.org

[3] 2017. Apache Cassandra. (July 2017). Retrieved July 13, 2017 from http://cassandra.apache.org

[4] 2017. Apache HBase. (July 2017). Retrieved July 13, 2017 from http://hbase.apache.org

[5] 2017. CockroachDB. (July 2017). Retrieved July 13, 2017 from http://github.com/cockroachdb/cockroach

[6] 2017. Database Statistics in Greenplum Database. (July 2017). Retrieved July 13, 2017 from http://gpdb.docs.pivotal.io/4360/admin_guide/intro/about_statistics.html

[7] 2017. LeveDB. (July 2017). Retrieved July 13, 2017 from http://leveldb.org

[8] 2017. MyRocks. (July 2017). Retrieved July 13, 2017 from http://myrocks.io

[9] Ashraf Aboulnaga and Surajit Chaudhuri. 1999. Self-tuning Histograms: Building Histograms Without Looking at Data. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data (SIGMOD '99)*. ACM, New York, NY, USA, 181–192.

[10] Swarup Acharya, Phillip B. Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. 1999. The Aqua Approximate Query Answering System. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data (SIGMOD '99)*. ACM, New York, NY, USA, 574–576.

[11] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. 2013. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *Proceedings of the 8th ACM European Conference on Computer Systems - EuroSys '13*. ACM Press, 29.

[12] 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. *PVLDB* 7, 14 (Oct. 2014), 1905–1916.

[13] Sattam Alsubaiee, Alexander Behm, Vinayak Borkar, Zachary Heilbron, Young-Seok Kim, Michael J. Carey, Markus Dreseler, and Chen Li. 2014. Storage Management in AsterixDB. *PVLDB* 7, 10 (June 2014), 841–852.

[14] Paul M. Aoki. 1999. Algorithms for Index-Assisted Selectivity Estimation. In *Proceedings of the 15th International Conference on Data Engineering, Sydney, Austrialia, March 23-26, 1999*. 258.

[15] Martin Arlitt and Tai Jin. 2000. A workload characterization study of the 1998 world cup web site. *IEEE network* 14, 3 (2000), 30–37.

[16] Daniel Barbará, William DuMouchel, Christos Faloutsos, Peter J. Haas, Joseph M. Hellerstein, Yannis Ioannidis, H. V. Jagadish, Theodore Johnson, Raymond Ng, Viswanath Poosala, Kenneth A. Ross, and Sevcik Kenneth C. 1997. The New Jersey Data Reduction Report. *IEEE Computer Society Technical Committee on Data Engineering* 20 (1997), 3–45.

[17] MKABV Bittorf, Taras Bobrovytsky, Casey Ching Alan Choi Justin Erickson, Martin Grund Daniel Hecht, Matthew Jacobs Ishaan Joshi Lenni Kuff, Dileep Kumar Alex Leblang, Nong Li Ippokratis Pandis Henry Robinson, David Rorke Silvius Rus, John Russell Dimitris Tsirogiannis Skye Wanderman, and Milne Michael Yoder. 2015. Impala: A modern, open-source SQL engine for Hadoop. In *Proceedings of the 7th Biennial Conference on Innovative Data Systems Research*.

[18] Paul G Brown and Peter J Haas. 2006. Techniques for warehousing of sample data. In *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*. IEEE, 6–6.

[19] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. 2001. STHoles: A Multidimensional Workload-Aware Histogram. *ACM SIGMOD Record* 30, 2 (6 2001), 211–222.

[20] Sunil Chakkappen, Thierry Cruanes, Benoit Dageville, Linan Jiang, Uri Shaft, Hong Su, and Mohamed Zait. 2008. Efficient and scalable statistics gathering for large databases in Oracle 11g. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, 1053–1064.

[21] Kaushik Chakrabarti, Minos Garofalakis, Rajeev Rastogi, and Kyuseok Shim. 2001. Approximate Query Processing Using Wavelets. *The VLDB Journal* 10, 2-3 (Sept. 2001), 199–223.

[22] 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 Transactions on Computer Systems (TOCS)* 26, 2 (2008), 4.

[23] Lei Chang, Zhanwei Wang, Tao Ma, Lirong Jian, Lili Ma, Alon Goldshuv, Luke Lonergan, Jeffrey Cohen, Caleb Welton, Gavin Sherry, et al. 2014. HAWQ: a massively parallel processing SQL engine in hadoop. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 1223–1234.

[24] Surajit Chaudhuri. 1998. An overview of query optimization in relational systems. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems - PODS '98*. ACM Press, 34–43.

[25] Surajit Chaudhuri, Gautam Das, and Vivek Narasayya. 2007. Optimized stratified sampling for approximate query processing. *ACM Transactions on Database Systems (TODS)* 32, 2 (2007), 9.

[26] Graham Cormode, Minos Garofalakis, and Peter J. Haas. 2012. Synopses for Massive Data. Now Publishers Inc.

[27] Graham Cormode, Minos Garofalakis, and Dimitris Sacharidis. 2006. Fast approximate wavelet tracking on streams. In *Advances in Database Technology - EDBT 2006*, Vol. 3896 LNCS. 4–22.

[28] Rainer Gemulla, Wolfgang Lehner, and Peter J. Haas. 2008. Maintaining Bounded-size Sample Synopses of Evolving Datasets. *The VLDB Journal* 17, 2 (March 2008), 173–201.

[29] Phillip B. Gibbons, Yossi Matias, and Viswanath Poosala. 2002. Fast incremental maintenance of approximate histograms. *ACM Transactions on Database Systems* 27, 3 (9 2002), 261–298.

[30] Anna C Gilbert, Yannis Kotidis, S Muthukrishnan, and Martin Strauss. 2001. Surfing Wavelets on Streams: One-Pass Summaries for Approximate Aggregate Queries. *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases* (2001), 79–88.

[31] Michael Greenwald and Sanjeev Khanna. 2001. Space-efficient online computation of quantile summaries. *ACM SIGMOD Record* 30, 2 (6 2001), 58–66.

[32] Raman Grover and Michael J. Carey. 2015. Data Ingestion in AsterixDB. In *Proceedings of the 18th International Conference on Extending Database Technology, EDBT 2015, Brussels, Belgium, March 23-27, 2015*. 605–616.

[33] Peter J. Haas and Arun N. Swami. 1992. Sequential Sampling Procedures for Query Size Estimation. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data (SIGMOD '92)*. ACM, New York, NY, USA, 341–350.

[34] Yannis E. Ioannidis and Stavros Christodoulakis. 1991. On the Propagation of Errors in the Size of Join Results. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data, Denver, Colorado, May 29-31, 1991*. 268–277.

[35] Yannis E Ioannidis and Viswanath Poosala. 1995. Balancing histogram optimality and practicality for query result size estimation. In *ACM SIGMOD Record*, Vol. 24. ACM, 233–244.

[36] Per-Ake Larson, Wolfgang Lehner, Jingren Zhou, and Peter Zabback. 2007. Cardinality Estimation Using Sample Views with Quality Assurance. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD '07)*. ACM, New York, NY, USA, 175–186.

[37] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *PVLDB* 9 (2015), 204–215.

[38] Yossi Matias, Jeffrey Scott Vitter, and Min Wang. 1998. Wavelet-based histograms for selectivity estimation. *ACM SIGMOD Record* 27, 2 (6 1998), 448–459.

[39] Yossi Matias, Jeffrey Scott Vitter, and Min Wang. 2000. Dynamic Maintenance of Wavelet-Based Histograms. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB '00)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 101–110.

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

[41] Viswanath Poosala, Peter J. Haas, Yannis E. Ioannidis, and Eugene J. Shekita. 1996. Improved histograms for selectivity estimation of range predicates. *ACM SIGMOD Record* 25, 2 (6 1996), 294–305.

[42] Jags Ramnarayan, Barzan Mozafari, Sumedh Wale, Sudhir Menon, Neeraj Kumar, Hemant Bhanawat, Soubhik Chakraborty, Yogesh Mahajan, Rishitesh Mishra, and Kishor Bachhav. 2016. Snappydata: A hybrid transactional analytical store built on spark. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2153–2156.

[43] P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. 1979. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*. ACM, 23–34.

[44] U. Srivastava, P. J. Haas, V. Markl, M. Kutsch, and T. M. Tran. 2006. ISOMER: Consistent Histogram Construction Using Query Feedback. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE '06)*. IEEE Computer Society, Washington, DC, USA, 39–.

[45] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. 2001. LEO - DB2's LEarning Optimizer. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 19–28.

[46] E.J. Stollnitz, a.D. DeRose, and D.H. Salesin. 1995. Wavelets for computer graphics: a primer.1. *IEEE Computer Graphics and Applications* 15, September (1995).

[47] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive: A Warehousing Solution over a Map-reduce Framework. *PVLDB* 2, 2 (Aug. 2009), 1626–1629.

[48] Jeffrey Scott Vitter, Min Wang, and Bala Iyer. 1998. Data cube approximation and histograms via wavelets. In *Proceedings of the seventh international conference on Information and knowledge management - CIKM '98*. ACM Press, New York, New

York, USA, 96–104.
[49] Hai Wang and Kenneth C Sevcik. 2003. A multi-dimensional histogram for selectivity estimation and fast approximate query answering. In *Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 328–342.
[50] Min Wang, Jeffrey Scott Vitter, Lipyeow Lim, and Sriram Padmanabhan. 2001. Wavelet-Based Cost Estimation for Spatial Queries. In *Advances in Spatial and Temporal Databases*. 175–193.

A BACKGROUND ON LSM STORAGE MODEL

The traditional way of organizing storage and indexing subsystems in relational databases has implied performing in-place mutations of a particular disk-resident data structure (i.e., B-Tree, R-Tree or heap file). Thus, any modification required performing random writes to the disk. Techniques like pinning disk pages in the buffer cache were introduced to slightly alleviate the problem, but performing structural updates in tree-like index structures still imposed significant per-update write overhead.

Modern data-intensive workloads require data management systems that are able to ingest significant numbers of records/second while providing the ability to run analytical queries on them. To achieve high ingestion rate, new storage models were adopted that rely on performing modifications in a log-structured way i.e., by breaking up a continuous stream of updates/inserts/deletes into groups based on their arrival time. Individual records within a group are stored in a order-preserving tree data structure to allow efficient lookup. These groups are periodically merged in order not to deteriorate read performance.

The LSM-framework operates on batches of records called *components*. At each point in time there exists a single mutable component which resides in main memory (called the *in-memory component*). All modification operations are performed within this component, *in-place*. Once its size crosses a certain threshold, the contents of an in-memory component are *flushed* to disk, creating an immutable *disk component*, while the in-memory component's content is reset. Since disk components are immutable, changes to records which already made their way to disk should only happen within the in-memory component, creating a *new version* of the record in the case of an update, or a special *anti-matter* record (an entry that cancels the record in an earlier flushed component) in the case of a delete. In a workload with continuous ingestion, the number of disk components keeps increasing. An indefinite growth would create multiple component files that should be probed during a read operation, thus hampering the performance of any selective lookup query. Instead, a periodic *merge operation* is scheduled to combine several disk components, possibly *reconciling* the anti-matter with regular records, creating a single merged component. The frequency of merges and the number of components deemed to be combined is determined by the *merge policy*. Figure 10 illustrates the typical operations of the LSM-based storage model.

The LSM-framework is usually implemented on top of some order-preserving index data structure, thus both events of the LSM lifecycle operate on records sorted by a particular key (primary or secondary). In the case of the LSM-flush the sorted order is directly imposed by the index structure of the in-memory component, while for the LSM-merge the order is obtained by merging pre-sorted input components. This allows to define both LSM-events as an index *bulkloading* operation, i.e., creating a new index from the pre-sorted data, and consider bulkload as another event in the LSM-fied index lifecycle.

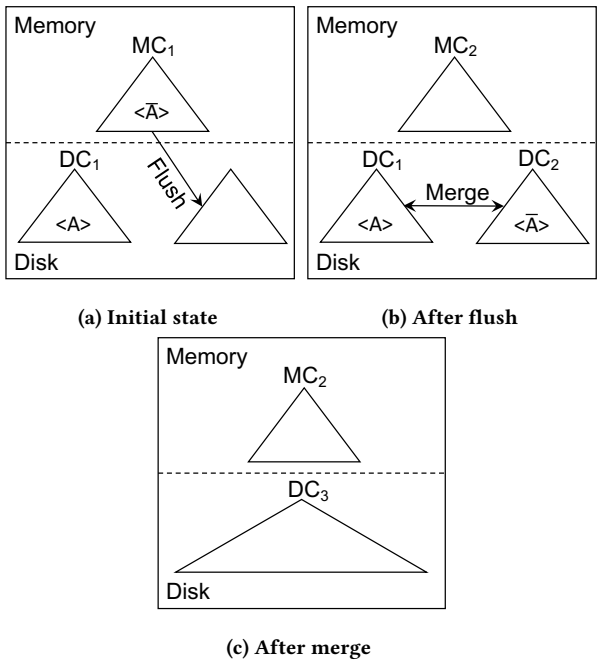


Figure 10: Typical operations in LSM-based storage model. (a) State prior to flush in which there already exists a disk component DC_1 with some record $\langle A \rangle$ and an in-memory component MC_1 with the result of deleting this record — an anti-matter record $\langle \bar{A} \rangle$. (b) LSM components after a flush operation has persisted the in-memory component and created a new disk component DC_2. (c) Result of a merge operation that combined the contents of components DC_1 and DC_2. Resulting component DC_3 does not contain any $\langle A \rangle$ records because they were reconciled during the merge.

B BACKGROUND ON WAVELETS

Wavelets are a mathematical tool for multi-resolution analysis based on hierarchical function decomposition. They are heavily used for compression in image processing, signal analysis, and other domains. The process of converting an original function signal into the wavelet domain, called *wavelet decomposition*, is a transformation which "breaks up" the original data into a coarse-grained base function and a number of *detail coefficients* that add more fine-level details to the high-level representation. Wavelet decomposition provides a function-agnostic method for the space-efficient representation of an underlying signal.

As a wavelet basis we have chosen the Haar basis because it provides a simple and efficient decomposition algorithm. Moreover, Haar wavelets provide a natural way of compressing the initial signal (a process called *thresholding*). Finally, because Haar decomposition is a linear transformation, it provides an easy way of combining different synopses once they are converted into the wavelet domain by summing up appropriate wavelet coefficients.

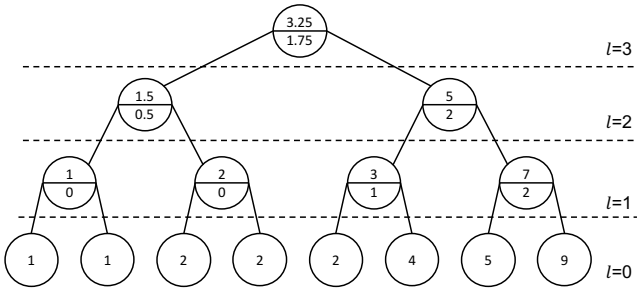


Figure 11: Structure of error tree. Nodes on the lowest level ($l = 3$) are comprised of the elements of prefix sum vector $\mathcal{F}^+$. Other nodes from the upper levels of the binary tree have the structure $\frac{x}{y}$ where x is an average coefficient, y is a detail coefficient of the wavelet transformation.

We formulate our purpose as estimating the frequency function of the records in some dataset $\mathcal{R} = (r_1, ..., r_n)$. Suppose that each record's key is defined on some bounded domain $\mathcal{D} = \overline{1, ..., \mathcal{M}}$, where $\mathcal{M}$ is some power of 2. Let's define a frequency of each domain key i as the number of records with an ID $\mathcal{D}_i$:

$$f_i = |r_j|, \forall j \in \{1, ..., n\}, \exists i \in \{1, ..., \mathcal{M}\}, where\ r_j.ID = \mathcal{D}_i.$$

All of these frequencies define a frequency vector $\mathcal{F} = f_1, ..., f_{\mathcal{M}}$. After that, we apply the Haar decomposition algorithm to convert the frequency vector into the wavelet domain. Once the decomposition is computed, a cardinality estimate for a particular key range can be obtained by issuing a range-sum query over the wavelet.

To illustrate the Discrete Haar wavelet decomposition, consider a simple example. Suppose we have a small dataset with domain $\mathcal{D}$ with $\mathcal{M} = 8$ and frequency vector for this dataset looks like $\mathcal{F} = [1\ 0\ 1\ 0\ 0\ 2\ 1\ 4]$, meaning that there are, for example, $f_1 = 1$ records where the record ID is equal to equal to $\mathcal{D}_1$. As a first preprocessing step we generate a prefix sum of the frequency vector $\mathcal{F}^+ = [1\ 1\ 2\ 2\ 2\ 4\ 5\ 9]$. This step is an optimization to convert the original frequency function into a dense signal which is known to be approximated more accurately by wavelets [38]. The Haar decomposition algorithm involves a recursive process of pairwise averaging and calculating the average differences of the input vector items, which produces *average coefficients* and *detail coefficients* respectively. Given two input wavelets coefficients A and B an average and a detail coefficients are calculated as $\frac{B+A}{2}$ and $\frac{B-A}{2}$ respectively. For the given example, averaging produces a lower-resolution $level_1$ vector $[\frac{1+1}{2}\ \frac{2+2}{2}\ \frac{4+2}{2}\ \frac{9+5}{2}] = [1\ 2\ 3\ 7]$. To recover the information that was lost during averaging, we compute $level_1$ a detail coefficients vector $[\frac{1-1}{2}\ \frac{2-2}{2}\ \frac{4-2}{2}\ \frac{9-5}{2}] = [0\ 0\ 1\ 2]$. This process is repeated recursively for all levels $1, ..., \log_{\mathcal{M}} = 3$ such that the average obtained on the $level_i$ becomes an input vector of the $level_{i+1}$. The result is a binary tree-like data structure called the *error tree* [38] illustrated in Figure 11. The final wavelet coefficients consist of the main average and all detail coefficients produced during the decomposition process: [3.25 1.75 0.5 1 0 0 1 2] sorted by their level in decomposition.

Note that the decomposition process is lossless, i.e., the number of values in the original signal is the same as the number of final coefficients. To subsequently compress the wavelet, we first normalize the original signal by dividing the coefficients by $\sqrt{2^{\log M - l}}$, where l is the coefficient's level. Thus, coefficients on lower resolution levels are considered more important than the similar coefficients on higher levels. Given a predefined budget K, we pick the top-K coefficients with the greatest normalized absolute values and create a *wavelet synopsis*, which has a provably optimal error under the L2-metric [46].

To reconstruct the original data back from the wavelet domain, we reverse the decomposition process and consider all non-significant coefficients to be 0. The intuition behind this approach to compression is that many of the detail coefficients by themselves are equal or close to 0, so removing them does not introduce significant changes to the reconstructed signal.

The wavelet decomposition algorithm extends naturally to the multi-dimensional case. Moreover, unlike histogram-based methods, it does not suffer from the "curse of dimensionality"[48].