UNIVERSITY OF CALIFORNIA
RIVERSIDE

Query Processing and Cardinality Estimation in Modern Database Systems

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Ildar Absalyamov

June 2018

Dissertation Committee:

Dr. Vassilis J. Tsotras, Chairperson
Dr. Michael J. Carey
Dr. Walid A. Najjar
Dr. Vagelis Hristidis
Dr. Eamonn Keogh

The Dissertation of Ildar Absalyamov is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

First, I would like to express my deepest gratitude to my academic advisors Professor Vassilis Tsotras and Professor Michael Carey who guided and motivated me through this journey. Professor Tsotras has been a thoughtful mentor, who patiently helped me with every aspect of my Ph.D., supported and inspired me all these years. Professor Carey's immense expertize and critical thinking heavily influenced me and instilled fundamental approach to database research. I am grateful to my collaborators Professor Walid Najjar, Dr. Robert Halstead, Prerna Budhkar and Skyler Windh for all their help with FPGA-related projects. I would also like to thank to Professor Eamonn Keogh, and Professor Vagelis Hristidis for joining my doctoral committee and giving me valuable feedback on this dissertation.

Special thanks goes to my fellow labmates Preston Carman and Steven Jacobs, with whom we drove countless miles between Riverside and Irvine over all these years. I would like to acknowledge all current members and alumnus of UCI AsterixDB team: Vinayak Borkar, Ian Maxon, Till Westmann, Professor Chen Li, Young-Seok Kim, Taewoo Kim, Jianfeng Jia, Pouria Pirzadeh, Abdullah Alamoudi, Murtadha Hubail, Chen Luo, Yingyi Bu, and Xikui Wang. I also thank the rest of UCR Database lab: Moloud Shahbazi, Nhat Le, Shiwen Cheng, Mohiuddin Qader, and Elena Strzheletska. Finally, I am thankful to all my friends who brought joy in my graduate life, my dearest friend Liudmila, Olga K., Olga T., Anton, and many others.

The text of this dissertation, in part or in full, is a reprint of the material as it appears in *Proceedings of 7th Biennial Conference on Innovative Data Systems Research (CIDR)* ("FPGA-based Multithreading for In-Memory Hash Joins", Asilomar, California, January 4-7, 2015), *Proceedings of 12th International Workshop on Data Management on New Hardware (DaMoN)* ("FPGA-Accelerated Group-by Aggregation Using Synchronizing Caches", San Francisco, California, June 27, 2016) and *Proceedings of the 2018 ACM International Conference on Management of Data (SIGMOD)* ("Lightweight Cardinality Estimation in LSM-based Systems", Houston, Texas, June 10-15, 2018). The co-author (Dr. Vassilis Tsotras) listed in that publication directed and super-

vised the research which forms the basis for this dissertation. The co-authors (Robert Halstead, Prerna Budhkar, Skyler Windh and Vasileios Zois) designed, implemented all FPGA algorithms, and performed FPGA-related experiments which appear in Chapter 5.

To my loving parents Khabir and Roza for all their encouragement and support.

Their sacrifices allowed me to come this far.

ABSTRACT OF THE DISSERTATION

Query Processing and Cardinality Estimation in Modern Database Systems

by

Ildar Absalyamov

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, June 2018
Dr. Vassilis J. Tsotras, Chairperson

The past decade has witnessed the proliferation of new ways to ingest, store, index, and query data. This explosion was driven by the needs of the modern applications, including social media, popular web services, and IoT sensors, characterized by high volumes and a rapid rate of incoming data. To cope with such high arrival rates, modern systems rely on the Log-Structured Merge Tree (LSM) storage model that uses sequential I/O instead of in-place updates. In addition to handling incoming data, LSM-based systems should provide useful analytics for their users, which in turn requires accurate statistics.

The first contribution of this thesis is developing a lightweight approach to collect and maintain concise statistical representations of the data in LSM-based systems so that it can be later used to drive cost-based optimizer decisions. In particular, we consider two problems, collecting statistics on indexed columns (which orders the stream of records based on the index key), as well as on non-indexed (unordered) columns. For each case, appropriate statistical summaries are considered so that the overall overhead on the system's critical path remains low, thus not affecting the ingestion process.

Recent hardware trends such as growing main memory capacity, stagnating CPU speeds, and increasing usage of parallel architectures have also influenced the design of data processing systems. These advances in hardware allow analyzing large datasets entirely in memory, which requires

specialized algorithms to process memory-resident data. Apart from the algorithms implemented on traditional CPU architectures, implementations leveraging fine-grained hardware multithreading on FPGAs have been recently proposed. However, up to this moment, there has been a lack of studies directly comparing the performance of these two approaches.

The second part of the thesis is devoted to a comparative study of common analytical operations (joins, aggregations, selections) for in-memory workloads using both multicore CPU and hardware multithreading approaches. We present a thorough experimental evaluation and show that implementations that use hardware multithreading outperform state-of-the-art CPU-based algorithms in terms of raw throughput in many cases while achieving much better memory bandwidth utilization.

Contents

List of Figures

Chapter 1

Introduction & Motivation

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. As a result database systems designed for such workloads must be able to ingest large amounts of rapidly incoming data while still allowing users to run complex analytical queries on top of this data. However, traditional database storage uses in-place updating, which implies random I/Os, and thus cannot cope with the high ingestion rates required in this environment. A popular design that has gained popularity in recent years addresses this issue by using log-structured merge trees (LSM-trees) [88] as a storage backend. The main idea of the LSM-based storage is to substitute slow in-place updates with sequential writes. For example, instead of performing a random I/O to find and delete a record, a deletion is represented as appending a special *anti-matter* (tombstone) record that contains information about the deleted entry. Similarly, a record update is not performed in place; rather, it is implemented as an insertion of the updated version of the record. While this allows modifications to be handled as sequential writes, it complicates query processing.

In the LSM model ingested records are first accumulated into an *in-memory component*, amortizing the cost of a single insert. Whenever the memory buffer fills up, this component gets

persisted (*flushed*) to disk sequentially and becomes an immutable *disk component*. For a continuous ingestion workload, the number of disk components will keep growing, which will eventually affect the query latency. Thus, LSM-based systems also periodically trigger a *merge* operation that consolidates multiple components (based on some merge policy) into a single file. During this merge, the system also *reconciles* deleted and updated entries, eliminating any matching anti-matter records.

Within this environment, we still need to support queries efficiently. Numerous research works have shown that the quality of the execution plan plays a crucial role in decreasing the total execution time for analytical query processing. A query optimizer estimates the cardinality of intermediate results and feeds them to a cost model so as to pick a plan with a smaller running time. Recent research [79] showed that producing an accurate cardinality estimate provides more substantial benefits in comparison to fine-tuning the optimizer's cost model. However, the way to obtain such statistics has largely remained the same since the early days of relational database systems [99] and has not taken into consideration the peculiarities of an LSM-based storage. The first part of this thesis is thus inspired by the following question:

> *How can accurate statistics be collected efficiently in modern database systems that use the LSM storage model?*

Motivated by this question, in Chapter 3 we propose a lightweight statistics-collection framework that exploits the properties of LSM storage. Our approach is designed to piggyback on major events (flush, merge and bulkload) in the LSM lifecycle. This allows us to easily create initial statistics and keep them in sync with rapidly changing data while minimizing the overhead to the existing system. We first concentrate on collecting statistics for indexed attributes, utilizing the sorted order provided by the LSM tree data structure during flush and merge operations. We have implemented and adapted well-known algorithms to produce various types of statistical synopses, including equi-width histograms [76], equi-height histograms [89], and wavelets [83]. 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 [16].

In Chapter 4, we lift the prior restriction of computing synopses for indexed attributes through the use of sketch-based algorithms [44], which sacrifice the estimation accuracy by providing probabilistic guarantees on computed statistics. Moreover, use of the Greenwald-Khanna sketch [67] enables us to maintain statistics on multiple attributes of the incoming data. We measure the overhead of computing such statistics on unordered fields and study the tradeoff between the number of such fields and the accuracy of the computed synopses. We compare the synopsis produced by the approximate sketch-based method to the exact indexed-based algorithm and show that the Greenwald-Khanna sketch provides comparable accuracy for range query cardinality estimates.

Another important factor driving changes in database systems over the last decade is the rapid drop of main memory prices. The amount of memory available on a typical server has increased significantly, thus creating a niche for new systems storing data entirely in memory [52]. In this setting, fetching records from secondary storage is no longer a predominant cost in the query execution pipeline. Instead, main memory databases heavily rely on architectural properties of modern multicore CPUs to process the given query at bare metal speed. However, at the same time memory bandwidth has not been growing fast enough to match the increased memory capacity. This growing gap between the memory bandwidth and the processing capabilities of the CPU is known as the *memory wall*. Multicore architectures have traditionally addressed this problem by introducing large cache hierarchies that rely on data locality (spatial and/or temporal). An alternative approach to mask the memory latency is to use multithreading at the hardware level [55]. Multicore CPUs are leveraging hardware multithreading, but they support relatively small (dozens) numbers of threads, whereas custom architectures (ASICs or FPGAs) have a much smaller thread context and thus can launch thousands of outstanding threads, effectively masking long memory latency.

Given the variety of operators used in relational query execution, it is an important open problem to experimentally compare the best CPU multicore implementations of such operators

with their FPGA multithreading counterparts and identify which approach should be used and when. Such a comparison can be thought of as a first step towards the goal of building a next generation hybrid database system that relies on a combination of CPUs, GPUs, and FPGAs. In Chapter 5, we implement state-of-the-art hash join, hash group-by-aggregation, and selection algorithms, tailored to exploit features of modern CPU processors, such as large caches (data, TLB), multiple cores, and SIMD instructions. We compare these CPU-based methods to their hardware multithreaded implementation prototyped on FPGAs. For all implementations, we study various aspects of performance, including throughput, scalability, memory bandwidth usage, and power efficiency.

Chapter 2

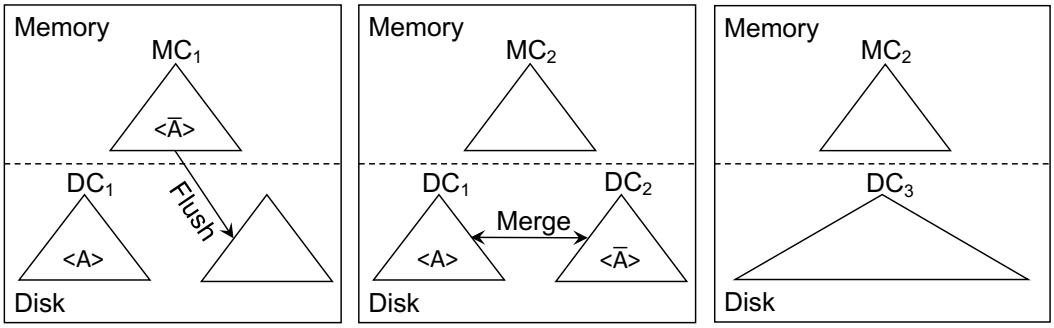
Background

This chapter first gives a short overview of the Log-Structured storage model, which serves as a foundation of our statistics-collection methods described in Chapters 3 and 4. Then we provide a brief explanation of the Haar wavelet decomposition algorithm, used to maintain statistics over indexed attributes (Chapter 3). Finally, we describe the Greenwald-Khanna [67] approximate quantile algorithm, which creates a basis for our approach to collecting statistics on unordered fields in Chapter 4.

2.1 The LSM Storage Model

The traditional way of organizing storage and indexing subsystems in relational databases has involved performing in-place updates of a particular disk-resident data structure (i.e., B-Tree, R-Tree or heap file). Thus, modifications result in performing random writes to the disk. Techniques like pinning disk pages in the buffer cache were introduced to 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 to be able to ingest significant numbers of records/second while providing the ability to run analytical queries on them.



(a) Initial state (b) After flush (c) After merge

Figure 2.1: Typical operations in the 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.

To achieve a high ingestion rate, they perform operations in a log-structured way to avoid the prohibitive cost of random I/O and substitute it with sequential writes.

Instead of periodically performing an I/O operation for each input entry, LSM-based systems operate on batches of records called components. At each point in time, there exists a single in-memory component, which accumulates the records into a current batch. Because it resides in main memory, its contents are inherently mutable, i.e. all modification operations are performed within this component in-place. To allow fast modifications, the in-memory component keeps individual records in an order-preserving data structure (e.g. B-Tree or Skip List). Once its size crosses a certain threshold, the contents of an in-memory component are flushed to disk, creating a new disk component, while the in-memory component's content is reset. To further leverage sequential I/O, contents of disk components are immutable, i.e. changes to records which have 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 in the case of a delete.

Relying on a log-structured model to perform writes, however, increases the complexity of a read operation. In order to lookup a single record, an LSM-based system needs to read the

contents of all disk components, as well as current in-memory components, perform a binary search in each component, and possibly reconcile anti-matter entries along the way. It is obvious that a large number of disk components will quickly deteriorate the read performance, so the database schedules a periodic merge operation that combines several disk components, 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 2.1 illustrates the typical operations of the LSM-based storage model.

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

2.2 Haar Wavelet Decomposition

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

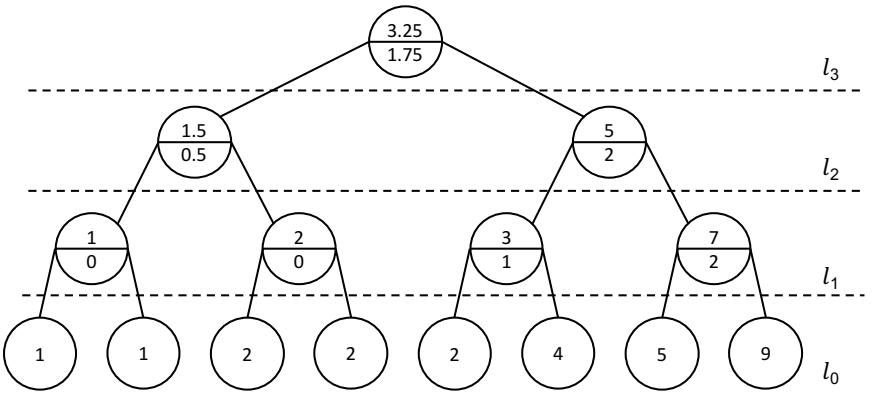


Figure 2.2: Structure of error tree. Nodes on the lowest level l_0 are comprised of the elements of prefix sum vector $\mathcal{F}^+$. Other nodes from the upper levels of the binary tree have the structure $\frac{x}{y}$ where x is an average coefficient and y is a detail coefficient of the wavelet transformation on the appropriate decomposition level.

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.

Consider the problem of estimating the frequency 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. The frequency of each domain key $\mathcal{D}_i$ is defined 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 whose domain $\mathcal{D}$ has $\mathcal{M} = 8$ values and the following frequency vector $\mathcal{F} = [1\ 0\ 1\ 0\ 0\ 2\ 1\ 4]$. 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 [83]. 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 inputs A and B, the average and the detail coefficients are calculated as $\frac{B+A}{2}$ and $\frac{B-A}{2}$ respectively. The recursive nature of Haar decomposition algorithm can be depicted as a binary tree-like data structure called the *error tree* [83] illustrated in Figure 2.2. Each level in that binary tree represents a recursive invocation of the algorithm. For a given example, on the level l_1 averaging produces a 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 on the level l_1, we compute 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 l_i becomes an input vector of the next level l_{i+1}. The final wavelet coefficients consist of the main average and all detail coefficients produced during the decomposition process: [3.25 1.75 0.5 2 0 0 1 2] sorted by their level in the 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 [105].

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"[113].

2.3 Greenwald-Khanna Approximate Quantile Algorithm

The problem of ordered statistics has been attracting attention ever since the seminal work on linear selection [28]. A generalization of the ordered statistics problem for some multiset $\mathcal{V}$ consisting of n elements is a ϕ-quantile that can be defined as an element with rank (i.e. position in a sorted multiset $\mathcal{V}'$) $r = \lfloor \phi n \rfloor$ for some $0 < \phi < 1$. Note that quantiles effectively describe a CDF of a distribution, and a set of ϕ_i-quantiles, where $\phi_i = \frac{i}{k}$ for $i = \overline{1,k}$, represent borders of the equi-height histogram with k buckets. An ϵ-approximate quantile must satisfy the following condition for some $0 < \epsilon < 1$:

$$\lfloor (\phi - \epsilon)n \rfloor \leq r \leq \lfloor (\phi + \epsilon)n \rfloor. \tag{2.1}$$

Greenwald and Khanna [67] proposed the current best-known deterministic comparison-based algorithm, which can compute the ϵ-approximate quantile in $O(log\frac{1}{\epsilon} + log(log(\epsilon n)))$ time per inserted element. The gist of the algorithm is to keep a bounded sample $\mathcal{S}$ of entries sorted by their natural order and use them to answer ϕ-quantile queries. By bounding the sample's size to $O(\frac{1}{\epsilon}log(\epsilon n))$, the algorithm achieves the aforementioned logarithmic update time.

The sample $\mathcal{S}$ consists of triplets $\langle v_i, g_i, \Delta_i \rangle$, such that $v_i \leq v_j$ for $i < j$ where $v_i, v_j \in \mathcal{V}$, while g_i and Δ_i satisfy the following conditions:

$$r_{min}(v_i) = \sum_{j \leq i} g_j \tag{2.2}$$

$$r_{max}(v_i) = \sum_{j \leq i} g_j + \Delta_i \tag{2.3}$$

$$g_i + \Delta_i \leq \lfloor 2\epsilon n \rfloor \tag{2.4}$$

where $r_{min}(v_i)$ and $r_{max}(v_i)$ represent the minimum and maximum possible values for the rank of the element v_i.

By bounding $r(v_i)$ with Equations 2.2 and 2.3 and g_i with Equation 2.4 we obtain an ϵ-approximation of the exact rank $r(v_i)$. The answer for the ϕ-quantile query is an element v_{i-1} that satisfies $\sum_{j \leq i} g_j + \Delta_i > 1 + \lceil \phi n \rceil + \frac{max(g_k + \Delta_k)}{2}$ for the smallest possible i. Also, note that two neighboring elements $\langle v_i, g_i, \Delta_i \rangle$ and $\langle v_{i+1}, g_{i+1}, \Delta_{i+1} \rangle$ from the sample $\mathcal{S}$ can be merged together into $\langle v_{i+1}, g_i + g_{i+1}, \Delta_{i+1} \rangle$ without violating inequalities 2.2 and 2.3 if the following is satisfied (to comply with 2.4):

$$g_i + g_{i+1} + \Delta_{i+1} \leq \lfloor 2\epsilon n \rfloor \tag{2.5}$$

This is the key property we will use to bound the sample's size.

In order to track entries that could be removed the original GK-paper proposes a complex `COMPRESS` procedure. However, Wang et al. [116] presented a modified version (termed GKAdaptive) that simplifies the algorithm while attaining similar performance (and even better in certain cases) when compared to the original. To construct a sketch, the GKAdaptive algorithm keeps sample entries in a sorted list $\mathcal{S}$ and allocates an auxiliary min-heap $\mathcal{M}$ which stores values $g_i + g_{i+1} + \Delta_{i+1}$ for each element in the sample. Upon inserting a single input entry v_i into a sketch, the GKAdaptive executes the following operations:

1. Perform a binary search in $\mathcal{S}$ and locate the successor entry $\langle v_j, g_j, \Delta_j \rangle$, i.e. an entry with minimum j such that $v_i \leq v_j$
2. If $g_j + \Delta_j \leq \lfloor 2\epsilon n \rfloor$ then update the successor entry to $\langle v_j, g_j + 1, \Delta_j \rangle$ (this is equivalent to inserting a new entry and merging it into successor right after)
3. Otherwise, insert the new triple $\langle v_i, 1, g_j + \Delta_j - 1 \rangle$ into $\mathcal{S}$
4. Insert the new value $g_j + \Delta_j + 1$ into $\mathcal{M}$

5. Pick a top element from the min-heap $\mathcal{M}$ and check if it satisfies the inequality 2.5. If yes, merge this element into its successor. If not, none of the entries in the sample can be merged, so the sample size is increased.

Maintaining an additional min-heap $\mathcal{M}$ while performing an insert operation has the same $O(log|\mathcal{M}|) = O(log|\mathcal{S}|)$ complexity as maintaining original sample $\mathcal{S}$.

Chapter 3

Lightweight Statistics for Indexed Attributes

3.1 Introduction

Despite the rapid growth in data volumes, the approaches to collecting statistics in database systems 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" interference might cause significant spikes in query latency.

The problem of prohibitive I/O costs is often mitigated by sampling, which considers only a portion of records from each disk page and skips some pages altogether. Once obtained, samples can be used as a data summary on their own [59], or they can serve as inputs to regular synopsis-building algorithms [37]. 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 [69, 38], but they heavily depend on the ability to identify appropriate strata 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 [57, 31] we are not aware of algorithms which allow unbiased estimates to be obtained in the 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 the 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 [59].

In this chapter, we propose a lightweight approach to 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 [73]). Instead, in this chapter, we generate synopses only on *indexed* (primary or secondary) attributes and hence use the sorted order provided by the indexes to devise efficient synopsis-gathering algorithms. Calculating statistics for unsorted attributes is discussed in Chapter 4.

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 merged 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 [7], an open source system that uses LSM-based storage [16]. Currently, AsterixDB relies on a heuristic-based optimizer, so

introducing statistics into that system could be first step towards building a full-fledged cost-based optimizer. Key contributions of this chapter 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 fully 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 chapter is structured as follows: Section 3.2 discusses related work and emphasizes how our approach is different from earlier research. Section 3.3 outlines the design of our statistics collection framework. Section 3.4 presents the experimental evaluation of the proposed methods. Finally, Section 3.5 provides the conclusions.

3.2 Related Work

Determining query cardinality is a classic problem in relational database systems. The seminal work on query optimization [99] describes how statistics can be used by the optimizer to estimate predicate selectivity, which, in turn, allows an optimizer to estimate 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 [12, 95, 11] are using data summaries as a primary data source to provide *approximate answers* to ad-hoc exploratory queries. Cormode et al. [44] thoroughly survey 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. [72] 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. [93] 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), and 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. [83] 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 [84], computing statistics over data streams [45] and approximate query processing [34].

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 [37]. Gibbons et al.[59] 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 [31] introduced a sampling algorithm that can obtain samples in a partitioned environment, whereas Gemulla et al. [57] 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 that solves the problem of producing unbiased samples in a setting similar to the 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 [25], HAWQ [35] and Hive [110]. 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 set of use cases tend to rely on sampling and choose uniform samples as a statistical data summary [78, 33]. Some warehousing systems provide optimizations on top of their regular statistics-gathering method, such as automatically triggering the recomputation of statistics [8] or skipping recomputation if a non-significant number of records were modified [6]. To the best of our knowledge, there are no systems that use the specific properties of the 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 [10, 32, 104, 103]. 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. [23] 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 [18] 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 more 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 existing related work lies in the fact that we are re-using the already-existing LSM data lifecycle 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 using only linear-time synopsis-collecting algorithms. We implemented histogram-based synopses and wavelets, but we chose 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; we instead keep synopses separate to mitigate the distributed workflow of collecting, storing, and consuming statistics.

3.3 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.

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 [17]. Its LSM-framework is created as a layer around the conventional implementations of various indexes, and it thus provides a unified LSM-ified abstraction for all index types supported by the system (B-Tree, R-Tree, and inverted indexes).

3.3.1 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 first 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 [17] 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 [16]. 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 chapter 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 could be extended to multiple dimensions [117, 114]; we leave computing statistics on composite keys and spatial data for future work.

3.3.2 Streaming Synopsis-building Algorithms

In the context of this chapter 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 (buckets/coefficients) 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 the 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 on 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 the 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 that compute histogram-based synopses, producing a wavelet requires performing a wavelet decomposition on the input data as it has been previously discussed in Section 2.2. 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 [113]). 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. [63]. The algorithm uses a priority queue to store the B most significant coefficients and an auxiliary array of $logN$ *straddling* coefficients that are used to track the current root-to-leaf path in the error tree. However, this algorithm is restricted to work only on "raw"

Algorithm 1 Streaming wavelet decomposition algorithm

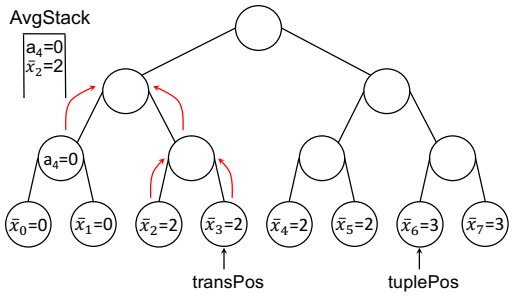
```
1:  procedure waveletTransform(stream)
2:      prefix ← 0, tuplePos ← 0, avgStack ← ∅, priorityQueue ← ∅
3:      for all tuple in stream do
4:          transformTuple(tuple.pos,prefix,tuple.value)
5:          prefix ← prefix + tuple.value
6:      if lastTuple.pos ≠ domainEnd then transformTuple(domainEnd,prefix,0)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

data or requires an additional pass to precompute a prefix sum. Note that in the 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 prefix sum, which is proportional to the domain length.

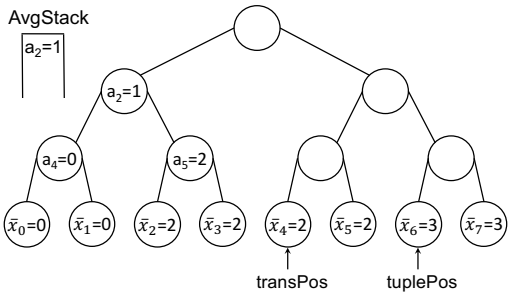
Algorithm 1 builds on the approach by Gilbert et al. [63] and is a streaming version of the discrete Haar wavelet decomposition algorithm that encodes a "dense" prefix sum signal. For simplicity, the figure omits the coefficient normalization, but we perform all appropriate transformations in our implementation. 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 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 6). 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 allows us to efficiently answer range-sum queries.

The gist of the streaming transform algorithm lies in its TRANSFORMTUPLE procedure (line 11) that performs an individual step of the transform. The procedure first calls the function GETTRANSFROMPOS 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 12).

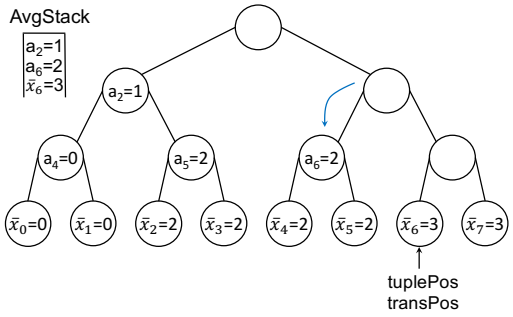
Due to the sparsity of the incoming signal, there can be a "gap" between the current transform position *transPos* and the position of the processed tuple *tuplePos*. Figure 3.1 shows an example of processing the gap between tuples $x_2 = 2$ and $x_6 = 1$ from the input sequence $X = [0\ 0\ 2\ 0\ 0\ 0\ 1\ 0]$. 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



(a) State after adding $\overline{x}_3 = 2$.



(b) State after cascade averaging



(c) State after adding $\overline{x}_6 = 3$

Figure 3.1: Example of the Algorithm 1 executing on the input $X = [0\ 0\ 2\ 0\ 0\ 0\ 1\ 0]$. The 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 onto the stack; (b) shows the result of averaging, triggered by pushing coefficient onto the stack; (c) depicts the step after covering interval $[\overline{x}_4; \overline{x}_5]$ with coefficient $a_6 = 2$ and adding final tuple $\overline{x}_6 = 3$.

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*, which 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 3.1c, where a newly added sibling coefficient is converted to $a_6 = 2$ (blue arrow). On each iteration of the

loop (line 26) the sibling coefficient's support interval is decreased by 2 while its value is multiplied by 2. After that, the calculated *dyadicCoeff* is pushed onto the *avgStack* and the position *transPos* is "advanced" according to the new coefficient on the top of the stack (line 28).

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*, and 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 onto the stack. Figures 3.1a and 3.1b depict exactly this situation, where pushing tuple $x_3 = 2$ onto the stack leads to a series of averaging operations (showed as red arrows) and puts the final average $a_2 = 1$ on the top of the *avgStack*. The AVERAGE procedure calculates and 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 onto the top of the *avgStack* (line 14 and Figure 3.1c). Because this new coefficient represents a new item on the bottommost level of the error tree it has $level = 0$ and value *prefix*+*tupleVal*.

3.3.3 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 [59, 84], 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 better 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 give estimates $E_{\mathcal{S}}$ and $E_{\overline{\mathcal{S}}}$ respectively. The total cardinality is then reported as $E_{\mathcal{S}} - E_{\overline{\mathcal{S}}}$.

3.3.4 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 [48], where a master node coordinates job scheduling and orchestrates 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; the 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 3.3.5 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.

3.3.5 Synopsis Mergeability

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/bulkloaded 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 need to 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 3.3.2 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 [72], we choose to keep all statistics, even mergeable ones, as separate entries in the catalog.

Maintaining synopses separately is a valid approach to manage statistics on the master node when we want to obtain an aggregate cardinality estimate. However, when computing local statistics during LSM-merges, we choose to create new synopses from the 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 the 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 master node 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.

3.3.6 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 the *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

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
```

which case it is subtracted. 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 synopsis on the master node. Queries being optimized can then obtain it from the 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 performing a full wavelet decomposition in reverse order, but instead it is computed by a single root-to-leaf path traversal in the error tree corresponding to this wavelet.

3.4 Experimental Evaluation

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.

3.4.1 Experimental Setup

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.

3.4.1.1 Datasets

To evaluate various data distributions we adopted the experimental framework proposed by Poosala et al. [93]. 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 a 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 similarity of these results, we present data only for the experiments involving 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* [68] that allowed us to create a continuous channel through which records are inserted into 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 a 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 [19]. 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.

3.4.1.2 Query Workload

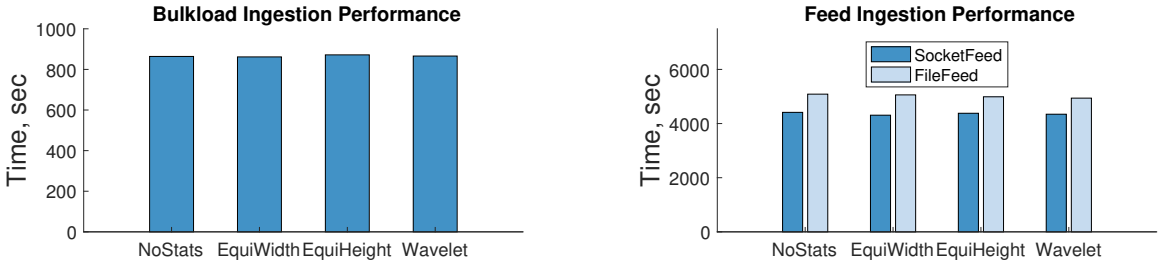
To evaluate the accuracy of the proposed framework we experimented with several types of range queries:

- 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: $\sum_{i=1}^{1000} \frac{e_i^{abs}}{1000}$.

3.4.2 Overhead Evaluation

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.



(a) Bulkload (b) Feed

Figure 3.2: 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).

In AsterixDB data can be persisted in two different ways: by bulkloading a dataset upfront or by performing DML insert/update/delete statements. Figure 3.2a presents the bulkloading execution times for the case when equi-height, equi-width 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 3.2b 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 that are 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 3.2, 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 (not shown). This allows us to affirm that the proposed statistics-collection framework indeed does not interfere with the normal LSM-based storage workflow in AsterixDB.

3.4.3 Accuracy Evaluation

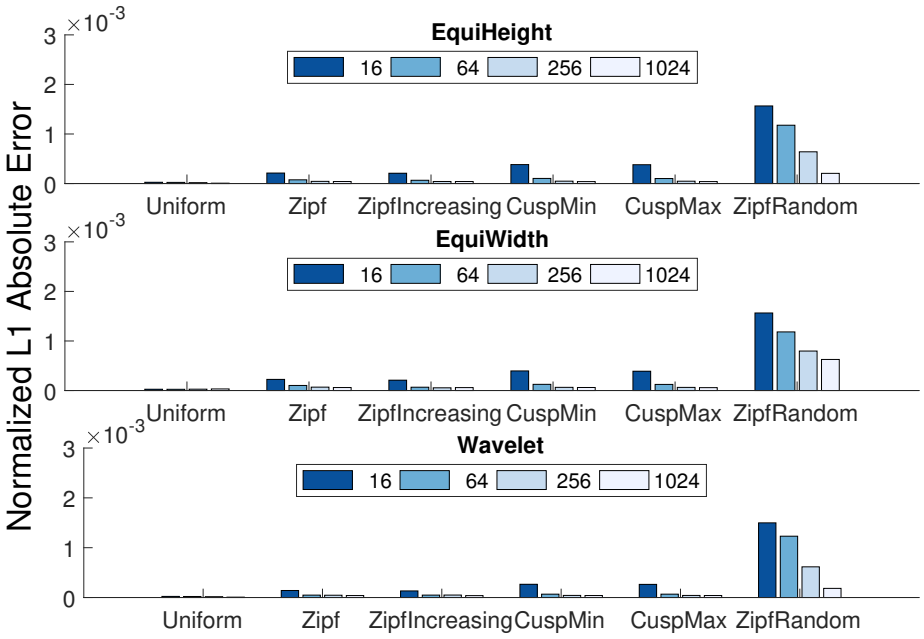
Due to the large size of the parameter space, we began our evaluation by fixing some of the variables for the experimental procedure.

3.4.3.1 Varying the Synopsis Size

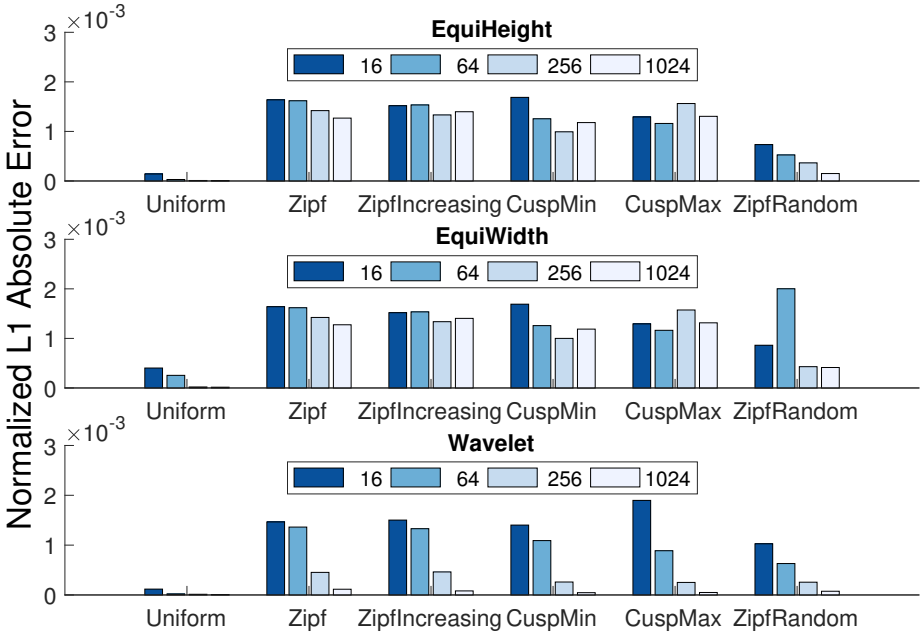
Figure 3.3 depicts the accuracy results while we increase the size of the synopsis for a FixedLength query workload with range length of 128. In the case of the histogram synopses, we vary the number of histogram buckets, whereas for wavelets the number of wavelet coefficients is increased. Note that 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 3.3a 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 3.3b. 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.

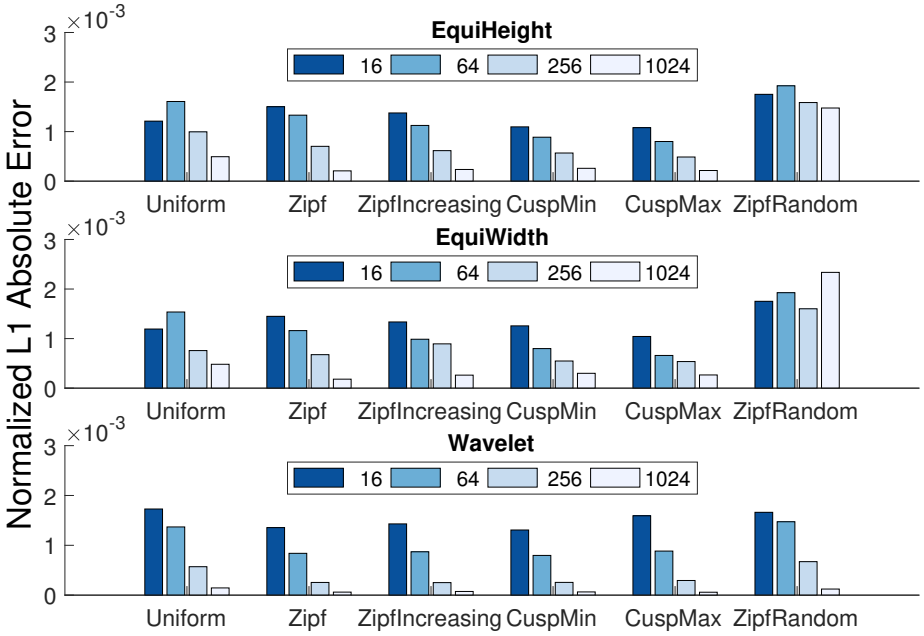
In contrast, in all the other cases in Figures 3.3b and 3.3c, as well as the ZipfRandom spread distribution in Figure 3.3a, the random permutation of spreads creates distributions with much more complex CDFs and makes estimation more complicated. Generally, for these we see a common trend that the cardinality estimation error is negatively correlated with the synopsis size.



(a) Dataset with Uniform frequencies



(b) Dataset with Zipf frequencies



(c) Dataset with ZipfRandom frequencies

Figure 3.3: 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.

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 [83].

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

3.4.3.2 Varying the Query Type

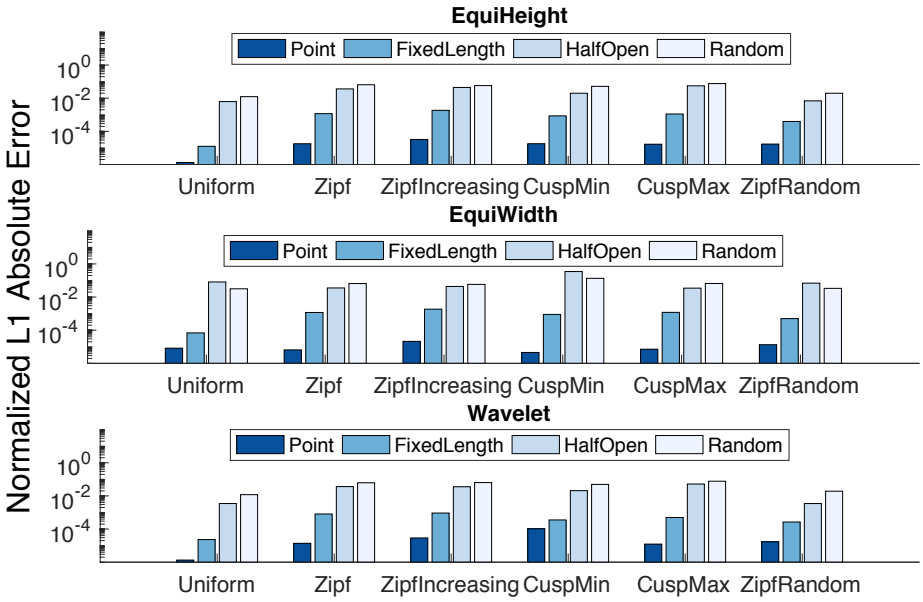


Figure 3.4: Estimation accuracy results for 4 different types of queries and a dataset with Zipf frequencies.

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 3.4 shows the accuracy for all query types mentioned in Section 3.4.1.2. 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 errors 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 the L1 absolute error metric emphasizes that difference.

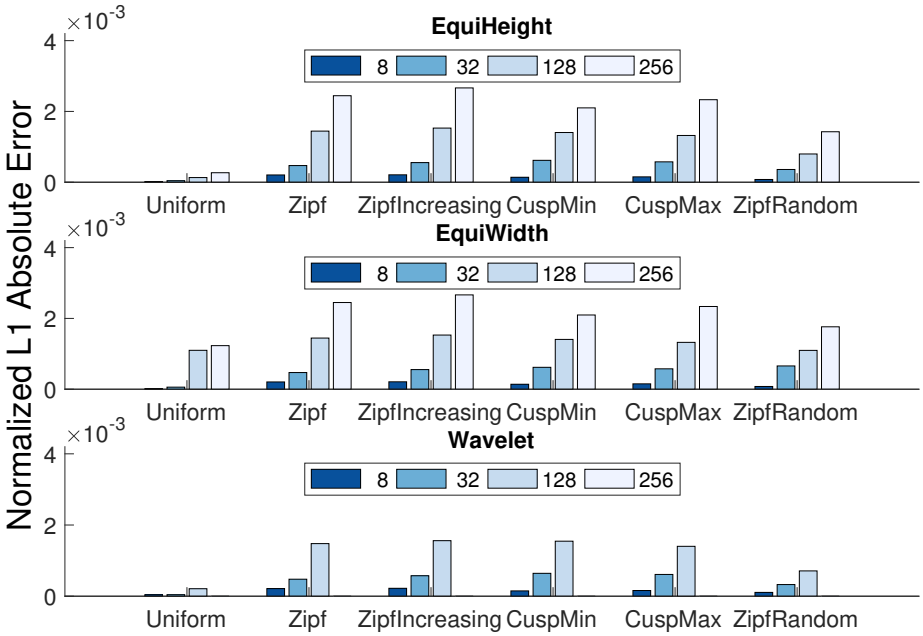


Figure 3.5: Estimation accuracy results for FixedLength queries and a dataset with Zipf frequencies for varying query sizes.

Figure 3.5 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.

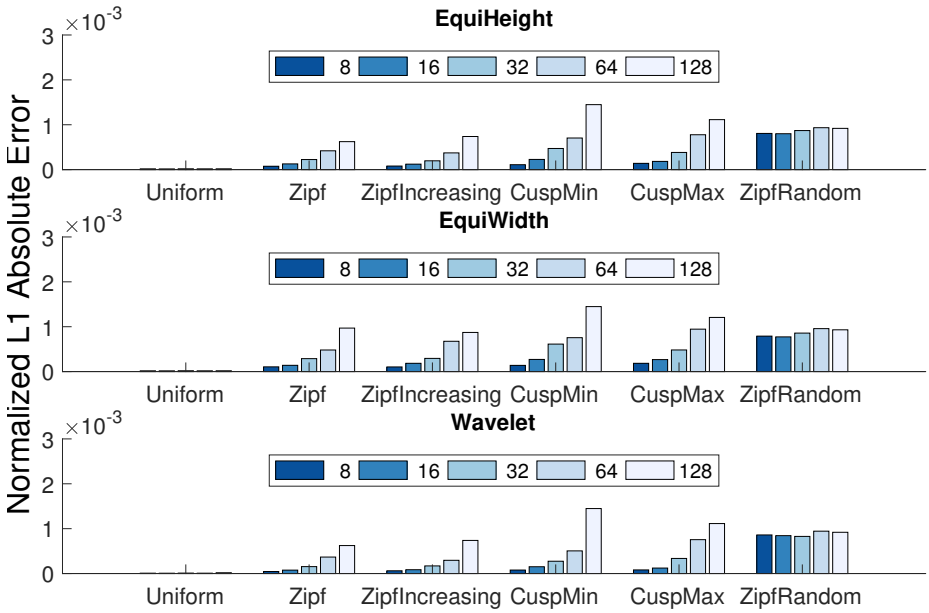
3.4.3.3 Varying the Number of LSM Components

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.

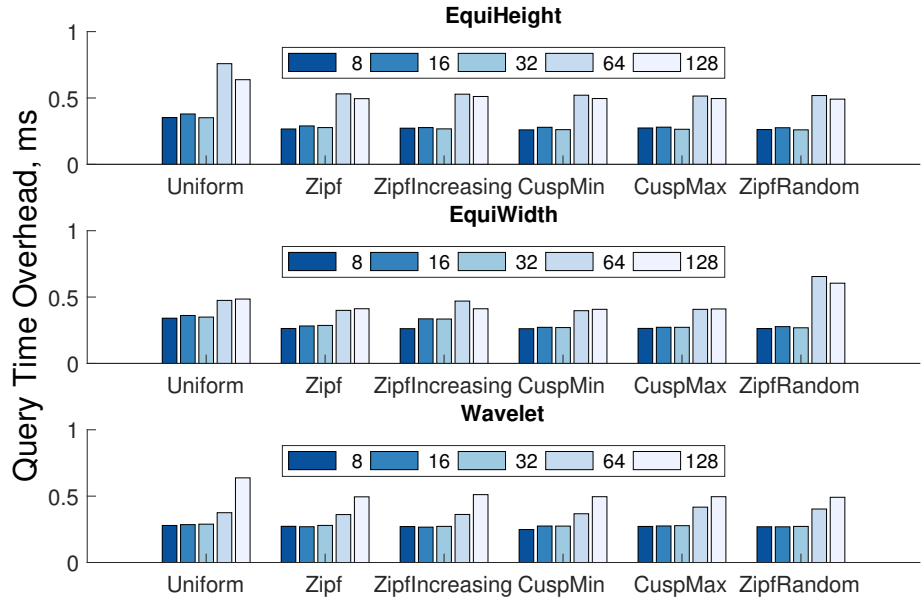
Figure 3.6a depicts the accuracy measurements, while Figure 3.6b considers the overhead during the querying stage of the experiment for various synopses types, spread distributions, and component numbers. As we can see, increasing the total number of components does slightly increase the estimation error, 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.

3.4.3.4 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



(a) Accuracy for varying number of LSM components



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

Figure 3.6: Experiments for 3 types of synopses, while varying the total number of 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.

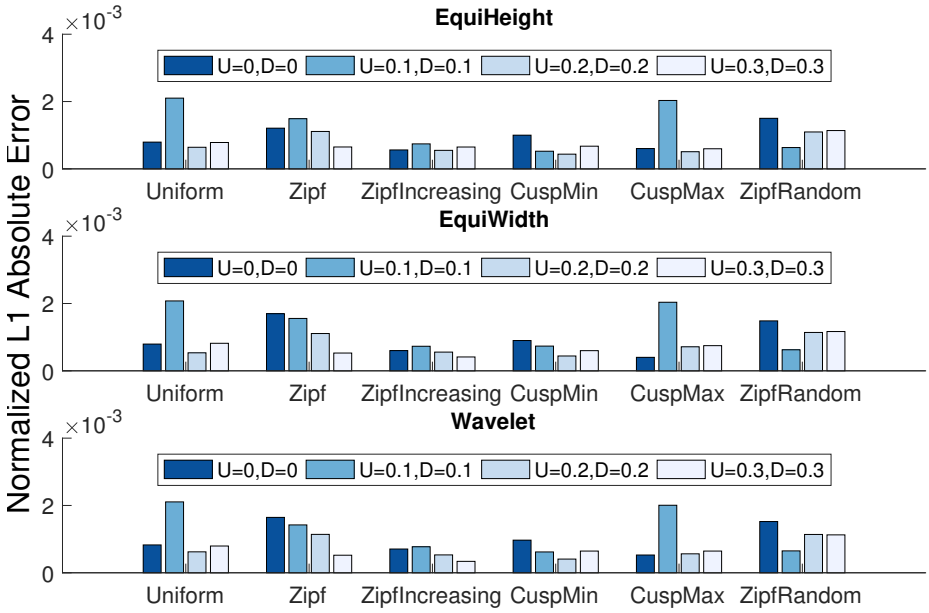


Figure 3.7: Estimation accuracy results for the workload with varying ratio of updates and inserts. Results obtained for various types of synopses on a dataset with ZipfRandom frequency distribution. The ratio of updates (U) and deletes (D) is scaled from 0 to 0.3.

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 3.7 illustrates the accuracy measurements for the dataset with ZipfRandom frequencies as 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

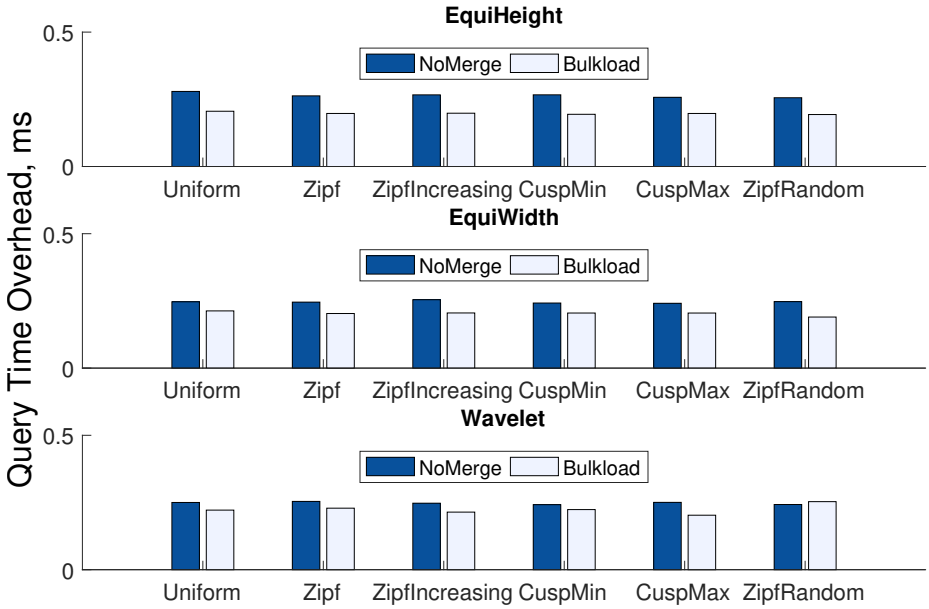


Figure 3.8: Query time overhead results for the case when a dataset is bulkloaded, creating a single LSM component, vs. a workload with the NoMerge policy, creating a maximum number of components.

algorithms) of dealing with changeable workloads while increasing the synopsis storage cost by only a constant factor (of 2).

3.4.3.5 Synopsis Mergeability

Figure 3.8 further studies how the number of LSM components influences the query optimization 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 the 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 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.

3.4.4 WordCup Dataset Experiments

Figure 3.9 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 equi-width 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, equi-height 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.

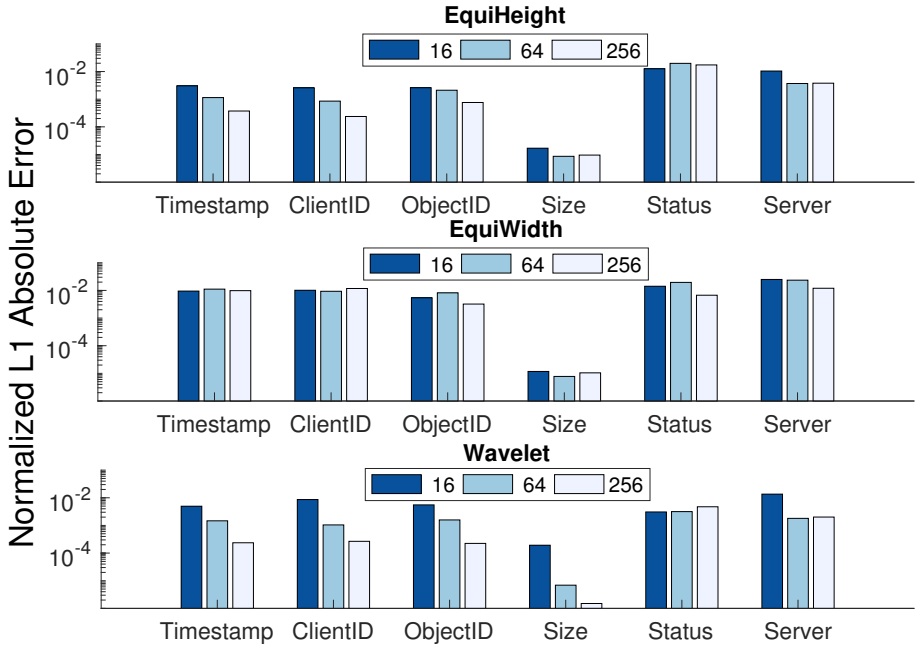


Figure 3.9: Estimation accuracy results, for all types of synopses, for 6 fields from the WorldCup dataset. The sizes of the synopses are increased from 16 to 256 elements.

3.5 Conclusions

In this chapter, we have proposed a novel lightweight approach to collecting statistics that exploits the properties of the LSM-based storage to obtain statistical data summaries of the underlying data. Our solution is integrated into 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 statistical 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.

Chapter 4

Sketchy Statistics: Lightweight Cardinality Estimation for Unordered Attributes

4.1 Introduction

Despite its novelty, the statistics-collection approach presented in Chapter 3 poses a major restriction by only allowing statistics to be collected on indexed fields. Leveraging the sorted order of the values of a particular field imposed by an index allowed us to use synopsis algorithms with linear complexity. However, maintaining an index creates an additional overhead both in terms of additional storage and increased latency of inserting new entries. Spending $O(nlog(n))$ time on sorting the values of a particular field just to get statistics on that field will cause a significant overhead in a Big Data context. In order to find a tradeoff between the statistics-related latency and the external overhead in the LSM storage layer, we need an algorithm with a sub-logarithmic (per input entry) runtime complexity. Given these requirements, we propose to extend our framework

and use sketches [44] as one of the synopsis-computing methods. A common alternative to gathering statistical information on a set of unordered items is to perform sampling, sort the sampled data, and then apply traditional synopsis-gathering algorithms that require sorted input. By setting the maximum size of the sample we can easily bound the time spent on computing unordered statistics. Sampling achieves the most benefit when it can decrease the number of I/O operations; however in a log-structured storage model where all data is streamed through regardless its advantages are diminished. On the contrary, sketch algorithms were specifically designed for the use case when the whole input is observed, but only the necessary information is kept, which allows the sketch to describe the distribution with the desired accuracy.

Unlike traditional methods of obtaining statistical data such as histograms, samples, and wavelets, sketch algorithms are a relatively recent development among these methods, primarily developed for the early data streaming systems [5]. However similar problems appear in the context of Big Data processing. As a result, many researchers and database practitioners have been applying different sketching methods in various settings outside of traditional streaming systems [42]. The streaming setting assumed that large amounts of data are passing through the system in real time (e.g. stock price data) and that it requires immediate analysis for fast decision making. In addition, data streaming systems were usually deployed on equipment with low processing and storage capabilities (e.g. edge networking devices). These rigid constraints required specialized algorithms that could guarantee sub-linear (in terms of the length of the input) processing time per input element and provide upper bounds on the amount of memory or storage used during computation. Moreover, records that appear in the events streams usually do not have an inherent order and support cash register, i.e. consist only of insertions, or turnstile semantics, i.e. contain both 'insert' and 'delete' events. Using algorithms that are designed either for cash register or for turnstile environment allows us to capture the statistical distribution of the values of unordered fields. Finally, sketching methods were designed with approximation in mind and thus provide a tunable parameter for accuracy guarantees. Increasing this approximation accuracy also yields a higher runtime complexity,

which allows us to control the desired tradeoff between statistics-collecting overhead and the normal LSM-lifecycle latency.

There are multiple sketch algorithms that capture various aspects of the dataset and answer different queries, e.g. provide the number of distinct values [56], answer item membership queries [27] or capture the frequency distribution of the dataset [64, 67, 36, 47, 46]. Since we are interested in statistics for the purpose of range query selectivity estimation, in this chapter we are concentrating on *frequency-based* sketching techniques. Among those sketches, we further identify two types of algorithms: *deterministic* and *randomized*. The former provides an approximate answer within a specified error bound ϵ, while the latter relaxes this condition by bounding the accuracy only with a given probability δ. Since it is more desirable to have better guarantees, we choose the deterministic algorithm proposed by Greenwald and Khanna [67] to obtain statistics (quantiles) on unordered fields. This algorithm has been rigorously tested in many research efforts and has been shown to have the best performance among deterministic algorithms [116]. Many alternative methods [64, 47, 101] rely on dividing the data domain $\mathcal{D}$ into non-overlapping chunks (usually dyadic intervals), which imposes a restriction that $|\mathcal{D}|$ should be a fixed size value which is a power of 2. In contrast, the Greenwald-Khanna sketch is a comparison-based algorithm, i.e., it relies solely on comparisons between input items in order to construct the resulting quantiles. This enables us to obtain statistics for domains with a total order of entries but with unbounded size, e.g. strings. Finally, quantiles produced by this sketch algorithm could be seamlessly integrated into our statistics framework, because a set of quantiles essentially represents an equi-height histogram of the data distribution.

The following summarizes the key contributions of this chapter:

- We implement the Greenwald-Khanna sketch [67] and integrate it into our statistics-collection framework to compute statistics on a used-specified number of unordered fields in the primary LSM component.

- We experimentally study the effect of computing multiple synopses as well as computing statistics with varied accuracy and establish a tradeoff between these parameters and the runtime overhead of computing those statistics.

- We compare the synopses obtained by the approximate sketch-based approach to the exact, indexed-based algorithms (studied in Chapter 3) and show that the Greenwald-Khanna sketch provides comparable accuracy for range query cardinality estimates.

The rest of the chapter is organized as follows: we review related work on sketches in Section 4.2, while Section 4.3 presents implementation details and describes how a quantile sketch (described earlier in Section 2.3) was integrated into our LSM-based statistics collection framework. We provide an empirical evaluation in Section 4.4, while Section 4.5 summarizes our findings.

4.2 Related Work

A seminal piece of work on estimating statistics via sketching by Alon et al. [15] introduced a method to approximate the F_2 measure (i.e. the sum of squares) of the frequency vector assuming a stream of records with 'cash register' (i.e. unordered streams without deletes) semantics. It was later shown that sketching the F_2 measure can be directly applied to estimate arbitrary inner-products, including relation join and self-join sizes [14], as well as a number of other problems. The original paper describes a version of the sketch algorithm that relies on averaging observations to reduce the variance, which leads to suboptimal bounds. However, this algorithm has been further improved by using hashing, namely the "Fast-AMS" [43] and "Fast-Count" [109].

Subsequent works, such as the Count [36] and Count-Min [47] sketches, have extended the AMS sketch to estimate the frequency of point queries. Both use the same sketch data structure but diverge in the way that they compute the estimate and provide different space/accuracy bounds. In addition to point queries, the Count-Min sketch provides a way to estimate range queries and inner-products (i.e. join/self-join cardinalities).

Sketching techniques have also been used as an auxiliary method to build other summary types and answer range and point queries. Gilbert et al. have demonstrated how atomic AMS sketches could be used to reconstruct near-optimal histogram [62], as well as wavelet synopses [63, 60] (called GKMS) for the 'turnstile' data model - a generalization of 'cash register' stream that allows deletions. These methods use heavy partitioning of the data domain into a series of disjoint dyadic intervals in order to reduce the problem of range query estimation to inner-product computation. However, the GKMS solution for approximate wavelet decomposition suffers from significant query complexity. As a result, Cormode et al. [45] proposed a novel Group-Count sketch which performs sketching entirely in the wavelet domain. Unlike GKMS, the Group-Count sketch has a tunable tradeoff between fast per-item update time and query time sub-linear in the size of the sketch.

Range query estimation is closely related to the problem of determining dataset quantiles. Gilbert et al. [64, 61] showed that quantiles could be computed using sketching techniques. By splitting the data domain into non-overlapping dyadic intervals the Random Subset Sum algorithm presented in [64] reduced the problem of identifying quantiles in the 'turnstile' model to that of calculating range sums.

All sketching techniques described so far rely heavily on randomization and are guaranteed to succeed with a certain (high) probability $1 - \delta$. There is also a class of deterministic algorithms which do not relax the approximation guarantees. Manku et al. [81] proposed a framework for deterministic quantile computation. A randomized version of this algorithm was introduced in [82] which does not need a priory knowledge of the stream length and improves space bounds. Finally, Greenwald and Khanna proposed a robust deterministic approximate quantile algorithm in [67]. The aforementioned quantile algorithms can work in domains of unbounded size, which sets them apart from methods that divide a fixed size domain into dyadic intervals. This is because they are comparison-based, i.e. they rely only on keeping a sorted sample of records from some totally ordered domain, which makes them widely applicable for non-numeric data types, e.g. variable length strings.

Sketching methods differ based on both their characteristics (randomized or deterministic, comparison-based or fixed universe, supporting inner-product or point/range queries, etc) and their performance (different bounds for accuracy, update time and space). Moreover, the provided theoretical bounds are sometimes overly pessimistic while, in practice an algorithm might provide performance significantly better than was predicted. This has incentivized empirical studies comparing various sketch algorithms against each other in a common setup. Rusu and Dobra [97] perform a comparison of the AMS [15, 14], Fast-AMS [43], Count-Min [47] and Fast-Count [109] sketches by measuring the accuracy of join and self-join size estimation on datasets with varying skews. It was determined that the Fast-AMS and Count-Min sketches perform significantly better on skewed data distributions and that the Fast-AMS sketch in all of the cases is close to the theoretical error irrespective of the data distribution. A recent study by Luo et al. [116] compared GK [67], MRL [82], Q-Digest [101], Count-Min [47], Random Subset Sum [64], a dyadic version of Count-Sketch [36], and a proposed new variation of the MRL algorithm called Random. The study identified Random and dyadic Count-Sketch to be the best among 'cash register' (i.e. unordered streams without deletions) and 'turnstile' (i.e. unordered streams with deletions) algorithms respectively. Moreover, the GK-sketch was still found to be the best among comparison-based approaches.

Despite the considerable research on new sketch methods, few systems have adopted these algorithms in practice. It was reported in [106] that Oracle uses the Random [116] and Count-Min [47] sketches to accelerate the computation of approximate aggregates. In addition they use AMS sketches for query optimization [112], specifically to perform sampling for join queries with dynamic predicates.

4.3 Implementation Details

This section describes modifications to our framework, presented in Section 3.3.1, in order to integrate the GK-sketch as one of the synopsis algorithms. In particular we implement the

GKAdaptive version of the algorithm (described in Section 2.3) since it is simpler and has been shown [116] to have the same performance characteristics as GK.

4.3.1 Multiple Statistics

Since we do not require the input stream to be indexed on a particular key, we can capture multiple fields' statistics during the same flush/merge/bulkload LSM operation. Algorithm 3 shows how we integrate GKAdaptive algorithm to into the LSM operation pipeline. All extracted quantiles create a set of right borders of an equi-height histogram that can be later used to get range query cardinality estimates as described in Section 3.3.6.

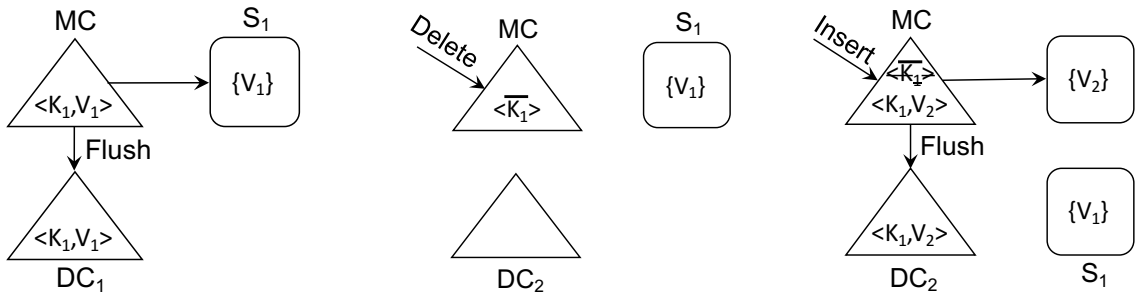
Algorithm 3 Algorithm for computing sketch-based statistics during the LSM flush/merge/bulkload for a set of fields

```
1:  function SKETCHSTATISTICS(stream,statsFields)
2:      histograms ← ∅, gkSketches ← ∅
3:      for all element in stream do
4:          for all field in statsFields do
5:              gkSketches[field].INSERT(element)
6:      for all field in statsFields do
7:          φ ← 1
8:          for φ ≤ k do
9:              buckets[φ] ← gkSketches[field].EXTRACT_QUANTILE(φ/k)
10:             φ++
11:         histograms[field] ← buckets
12:     return histograms
```

4.3.2 Handling Anti-matter Records

In order to handle LSM anti-matter records in non-indexed fields, we apply an approach similar to the one described in Section 3.3.3. As with index-based statistics, we fork the synopsis computation into two streams: one stream produces a regular synopsis, while another creates an anti-matter "twin" that describes the statistical distribution of anti-matter records in a given component.

However, in AsterixDB the anti-matter record format in the case when the attribute is indexed is different from the case when the attribute has no secondary indexes. Figure 4.1 shows



(a) State after insertion of the record $R = \langle K_1, V_1 \rangle$. Flush creates a new disk component DC_1 and synopsis S_1

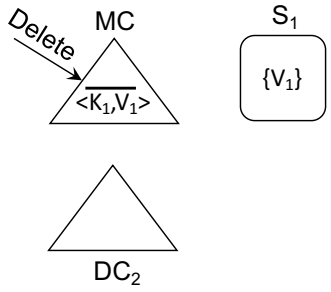
(b) State after deletion of the record R. An anti-matter record $\langle K_1 \rangle$ is inserted into an empty memory component MC

(c) State after insertion of the new record $R' = \langle K_1, V_2 \rangle$. The previous anti-matter record is replaced. Flush creates a new disk component DC_2 and synopsis S_2

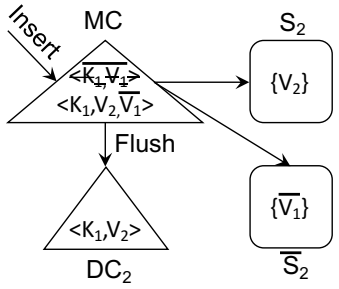
Figure 4.1: Example of a wrong cardinality estimate after a series of LSM operations.

an example when this could lead to an incorrect cardinality estimate. Consider a single two-field record $R = \langle K_1, V_1 \rangle$, which is identified by the key K_1 and has value V_1 on the field for which we would like to calculate statistics. Figure 4.1a shows the state after the insert and flush operations, which results in the creation of synopsis S_1 with contents $\{V_1\}$. Later on, record R is deleted, which generates an anti-matter entry that contains *only the key* of the deleted record $\overline{R} = \overline{\langle K_1 \rangle}$, depicted in Figure 4.1b. We further insert a new record $R' = \langle K_1, V_2 \rangle$ with the same key K_1. Since it is identified by the same key it will replace the existing anti-matter entry in place. Finally, Figure 4.1c shows the state after another flush that created a new synopsis, now containing the new value $\{V_2\}$. However, the overall statistics $S_1 \cup S_2 = \{V_1, V_2\}$ are now overestimating the total number of entries. In AsterixDB this can occur because in order to delete an original record $R = \langle K_1, V_1 \rangle$ the system does not generate an anti-matter entry containing values of all non-indexed attributes; only the key fields are required to be a part of the anti-matter entry.

To avoid this situation, we propose augmenting the in-memory representation of the records in order to capture the correct statistics. Figure 4.2 shows the stages (b) and (c) from the previous example. Each time a record is deleted (or replaced by some other record with the same key) we can append the previous values of all fields for which we keep statistics. Figure 4.2a shows



(a) State after deletion of the record R. Value V_1 of the field of a deleted record is a part of the anti-matter record $\langle K_1, V_1 \rangle$ along with the key.



(b) State after insertion of a new record $R' = \langle K_1, V_2 \rangle$. The previous anti-matter record is replaced, but the value of the field V_1 of a previously deleted record is carried over to the new record. Flush now creates a pair of new synopses S_2 and $\overline{S_2}$.

Figure 4.2: Proposed solution for incorrect cardinality estimation.

this as the augmented version of the record $\overline{R} = \overline{\langle K_1, V_1 \rangle}$. Whenever a new entry replaces an augmented anti-matter record $\overline{R}$ (or a regular augmented entry) we carry over the old values of the statistics fields. Figure 4.2b depicts this with new entry $\langle K_1, V_2, \overline{V_1} \rangle$. After the second flush, two new synopses will be generated so that the overall cardinality now reflects the correct combined estimate $S_1 \cup S_2 \cup \overline{S_2} = \{V_1\}$. Note that this record augmentation happens *only* within the context of the in-memory component; the records are converted back to their normal representation during the flush. This allows up to maintain backwards-compatibility with the storage format in AsterixDB.

4.4 Experimental Evaluation

All experiments presented in this section use the same experimental setup as in Section 3.4.1. The only difference is a that no indexes are required. Instead, we introduce a special hint that is provided by the user at the time of dataset creation. This hint contains a concatenated list of unindexed field names for which the user would like to keep statistics.

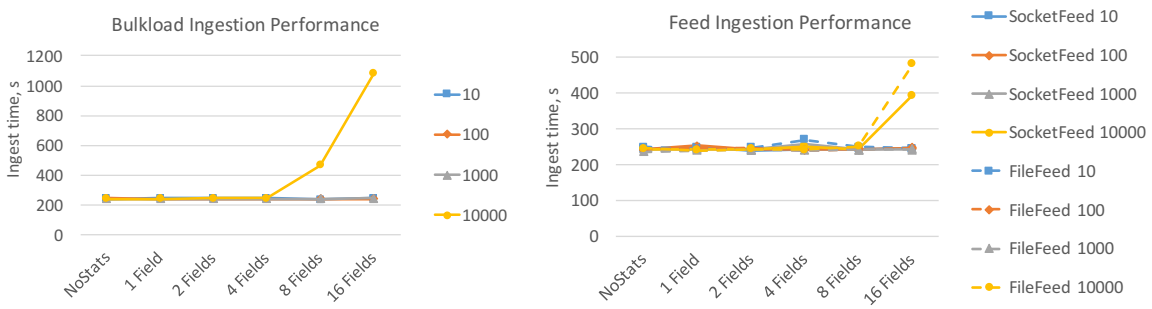
4.4.1 Overhead

We start by studying the effects of unordered statistics collection on the overall ingestion performance. According to the description of the GK-sketch (Section 2.3), the runtime complexity of the algorithm is:

$$O(n(log\frac{1}{\epsilon} + log(log(\epsilon n)))) \tag{4.1}$$

where n is the number of input entries and ϵ is the desired sketch accuracy. Since the number of entries in our experiments is fixed we mainly vary the accuracy of the sketch and the number of the fields for which we keep statistics (i.e. number of sketches). According to the definition of an element's rank $r_{min} = 0$ and the ϵ-accurate ϕ-quantile inequality 2.1 we get that $\phi_{min} = \epsilon$. Thus, in the following we treat ϕ as equal to the sketch accuracy ϵ, i.e. a 0.01-quantile sketch (which yields 100 buckets) is obtained by setting $\epsilon = 0.01$. In the following we use the terms sketch accuracy and number of extracted buckets/quantiles interchangeably.

AsterixDB, as well as most other LSM-based systems, can be modeled for performance analysis purposes as a process with 2 active threads: the ingestion thread and the IO-thread. During insertion, the former is responsible for reading a record from the input, parsing it, inserting the new entry into the in-memory LSM component, and checking if a flush should be triggered in the event that the size of the in-memory component has crossed some threshold. Once triggered, the flush request is asynchronously received by the IO-thread which writes the contents of a component onto the disk, while the ingestion thread starts populating a new in-memory component (such technique is commonly called 'double buffering'). In addition to flushing records to disk, the IO-thread is also responsible for collecting statistics during an LSM-event and building a sketch. The latency of the IO-thread is increased by either increasing sketch accuracy or collecting statistics on multiple fields concurrently.



(a) Bulkload (b) Feed

Figure 4.3: Wall clock time for ingesting 1M records (a) using a bulkload operation, or (b) through socket/file feed while varying the number of extracted quantiles from 10 to 10000 and the number of statistics fields from 0 to 16.

During the normal ingestion process the disk latency imposed by the IO-thread is expected to be low compared to ingestion thread overhead; however computing statistics during LSM-events increases the work performed by the IO-thread. If we keep gradually increasing the I/O latency at some *crossover* point these latencies would be equal and if we will keep increasing statistics-imposed latency the ingestion process will not be able to makes a progress and will start blocking.

In the first set of experiments we try to identify where the latencies of IO-tread and ingestion tread cross over. We measure the wall clock time for the simple workload, which ingests 1M records on a single-node, single-partition AsterixDB installation. Figure 4.3 shows the result of this experiment for three different types of ingestion workloads (Bulkload, SocketFeed, and FileFeed) while we varied the number of fields for which statistics are calculated and the number of extracted quantiles. Figure 4.3b shows that there is no significant difference between FileFeed and SocketFeed, which assures us that the experiment emphasizes the overhead in the storage layer of the system rather than the latency associated with reading the result from the input (a network socket or a file). Unless otherwise noted, we will be using a socket-based feed for all further experiments.

Figure 4.3a above shows that the wall clock time for bulkloading data increases significantly when we consider more than 4 fields and extract more than 1000 quantiles, as compared to the similar setup in the feed-based ingestion experiments depicted on Figure 4.3b. This is explained by the fact

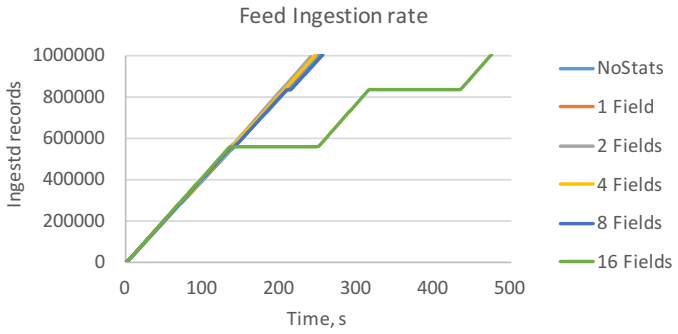


Figure 4.4: Ingestion rate of the socket-based feed workload for a sketch with 10000 quantiles while varying number of statistics fields.

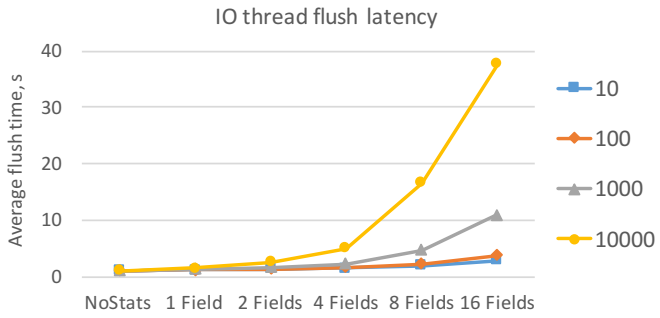


Figure 4.5: Average time spent during the LSM-flush in IO thread while varying the number of extracted quantiles from 10 to 10000 and the number of statistics fields from 0 to 16.

that feed ingestion here uses a NoMerge [17] LSM merge policy that, as the name implies, never triggers the LSM merge operation. Thus the sketches are calculated only during component flushes, so the number of records (and therefore n in equation 4.1) is small. In contrast, during a bulkload the system populates one big LSM component with $n = 1M$ entries, so the runtime overhead of calculating statistics cannot be masked by other storage-related latencies. Furthermore, even for feed-based ingestion workloads, we can see the overhead of computing statistics increases when we are extracting 10000 quantiles.

In order to study the cause of the increased ingestion wall clock time we perform additional experiments. Figure 4.4 measures the number of ingested records over time for a sketch with 10000

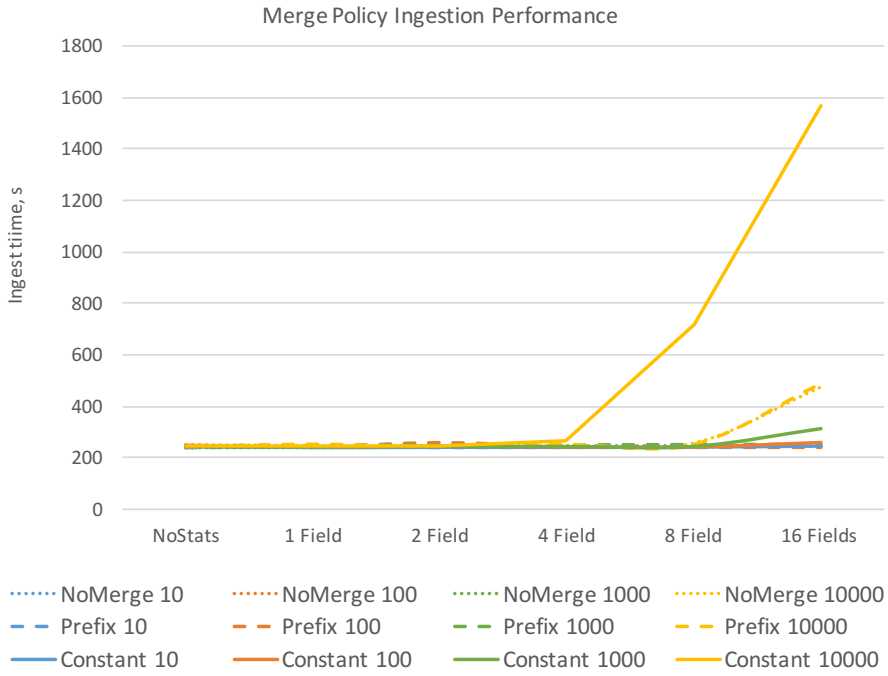


Figure 4.6: Wall clock time of ingesting 1M records NoMerge,Prefix and Constant LSM merge policies, while varying the number of extracted quantiles from 10 to 10000 and the number of statistics fields from 0 to 16.

quantiles and a varying number of statistics fields. The graph shows that for a small number of statistics fields (0-8) the number of records grows linearly with time. However for 16 statistics fields the ingestion speed plateaus, which shows that the ingestion was being blocked by statistics-related overhead. Additionally, Figure 4.5 plots the average time for a flush operation for the varying numbers of quantiles and statistics fields. The graph shows that the overhead of computing LSM-based statistics (latency of the IO-thread) grows as we increase the number of sketch quantiles, or the number of statistics fields. However, Figure 4.3b did not show the ingestion process being blocked until the number of concurrently collected sketches equals to 16. This supports our analytical model and shows that the crossover point is not reached until we collect statistics on 16 fields here, despite the increasing latency of the IO-thread.

All of the graphs presented earlier in this section have used the NoMerge policy This policy has virtually zero write amplification, but it instead results in poor read latency because for each query all components must be accessed. As we have seen in Figure 4.3, the statistics-related overhead grows with the number of records in the flushed or merged components. During merges our sketch-based statistics are simply recomputed from the scratch, so a more practical merge policy, one that does perform periodic merges, will likely shift the crossover point. On the other side of the spectrum we have the Constant merge policy, which keeps the number of disk components k at each given moment constant. The extreme case when $k = 1$ will result in maximum write amplification because flushing a new component will always cause a merge, and overall the total time spent during ingestion is $O(n^2)$ in terms of number of ingested records. More practical polices would use higher k or take into account other factors, like maximum size of the component in the default AsterixDB policy Prefix [17], to determine if a component needs to be flushed. Figure 4.6 presents measurements of the feed-based ingestion performance for 3 different LSM merge policies. As was expected, the Constant policy starts to significantly degrade the system's performance for 10000 quantiles, even for the modest number of statistics fields.At the same time, the performance of the Prefix policy is almost the same as for NoMerge, and it starts influencing the ingestion only for 10000 quantiles and 16 statistics fields. This experiment shows that for a reasonable number of fields (up to 16) and a practical LSM merge policy, computing a GK-sketch with 1000 quantiles (or alternatively accuracy $\epsilon = 0.0001$) does not have a significant effect of the ingestion performance in AsterixDB.

4.4.2 Accuracy

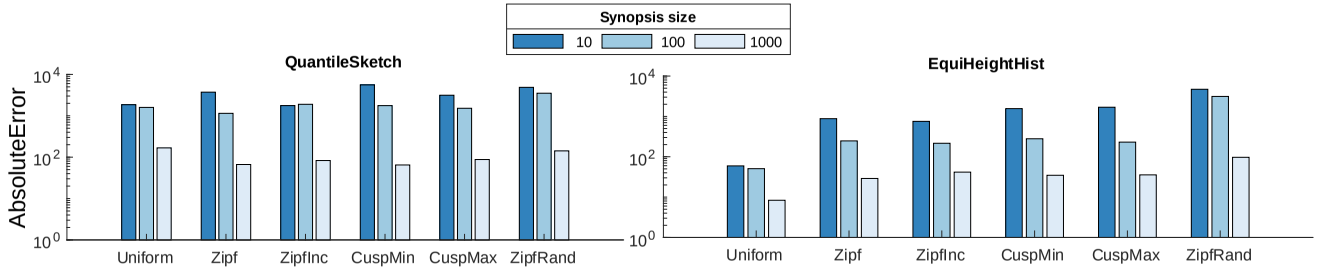
We now proceed to our experiments that measured the GK-sketch accuracy of range query cardinality estimation. For our evaluation we use the same synthetic datasets, described in Section 3.4.1.1: 3 different distributions for item's frequency (Uniform, Zipf, ZipfRandom) and 6 types of value distribution (Uniform, Zipf, ZipfDecreasing, ZipfRandom CuspMin, CuspMax). To measure

accuracy, we have executed 100 different range queries with a fixed length $l = 64$ and report the average error using a particular metric.

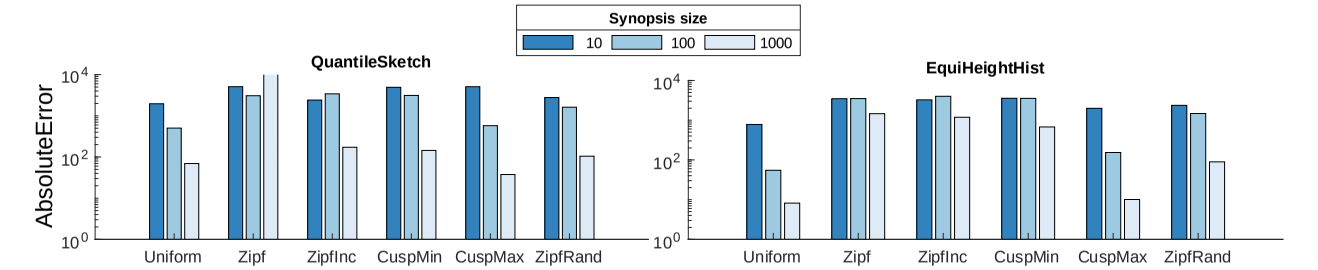
In order to evaluate our approximate sketch-based approach for unordered attributes, we will measure its estimation accuracy against the index-based method from Chapter 3, which produced an equi-height histogram. Because the quantile sketch produces the same type of statistical synopsis (equi-height histogram), this experiment emphasizes only the quality of the produced synopsis (since we factor out the accuracy associated with the type of synopsis used). In both cases, we vary the number of buckets (quantiles) that we allocate for the histogram. During ingestion, we use the AsterixDB [16] Prefix merge policy for non-indexed statistics and it's Correlated-Prefix policy for index-based synopses to ensure that the same number of components/synopses is generated in both cases.

Figure 4.7 shows the average absolute error measurements for all aforementioned datasets. As expected, by increasing the number of buckets allocated to the histogram, the error decreases for both synopses. Moreover, apart from a few notable exceptions (uniformly distributed values with Zipf and ZipRandom frequencies), the error associated with both types of statistics is comparable.

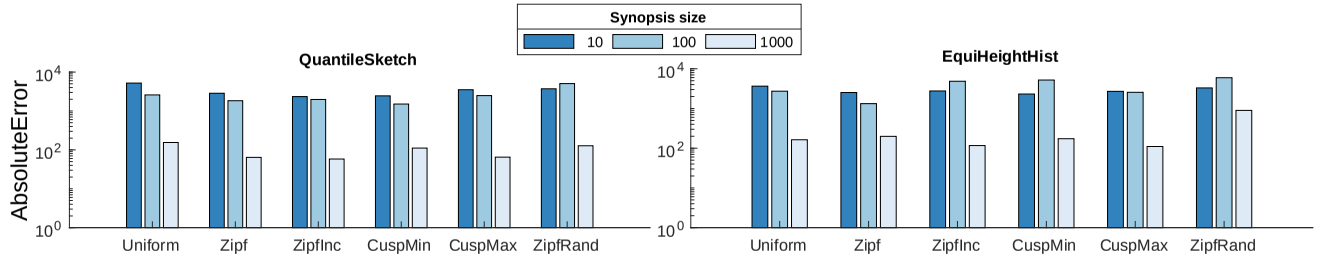
While the *absolute error* might be a good metric in various cases, it does not correctly describe accuracy in situations when both the cardinality and the estimate are small in absolute terms. On the other hand, the *simple relative error* tends to treat overestimates and underestimates unequally. The *q-error*[87] is an alternative error metric, calculated as $max(\frac{C}{\widehat{C}}, \frac{\widehat{C}}{C})$, which intuitively shows the factor by which the cardinality C is different from the estimate $\widehat{C}$ while treating overestimation and underestimation symmetrically. In addition, the q-error provides an upper bound on the quality of the produced plan, which makes it especially relevant for evaluating query optimizers. Figure 4.8 shows the same accuracy measurements using an average q-error metric. We observe that the sketch error is comparable for a small number of buckets. For 1000 buckets and non-uniform frequencies, however, its accuracy is even lower than that of the histogram. Moreover, we observe



(a) Uniform frequencies



(b) Zipf frequencies



(c) ZipfRandom frequencies

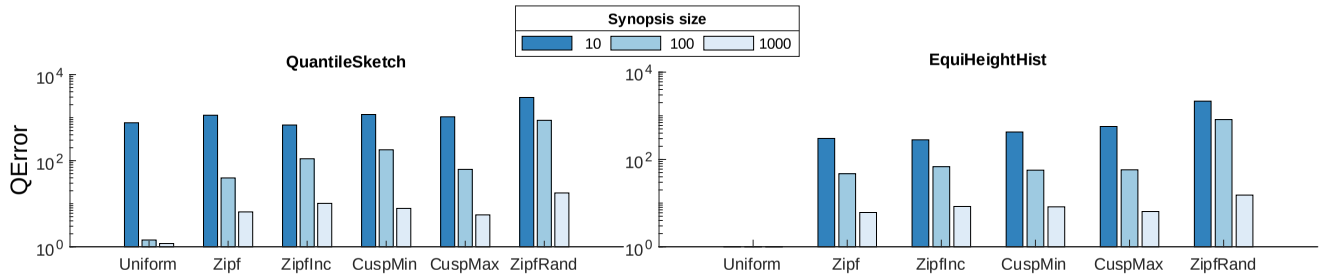
Figure 4.7: Average absolute error of measuring estimation accuracy for Greenwald-Khanna sketch and equi-height histogram while the number of buckets/quantiles is increased from 10 to 1000.

that for Uniform value sets, histograms provide very accurate estimates regardless of the number of buckets, whereas a sketch with less than 1000 quantiles has a significantly higher error for Zipf frequencies.

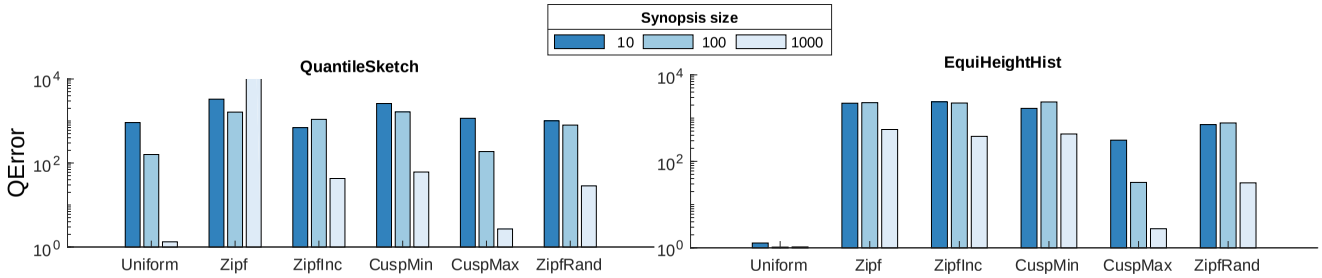
Overall we observe that allocating 1000 quantiles for sketching provides comparable or in some cases even better accuracy. Since the experiments in the previous Section identified that a sketch with 1000 quantiles does not cause significant overhead during ingestion, this a good candidate for the default number of extracted quantiles.

4.5 Conclusions

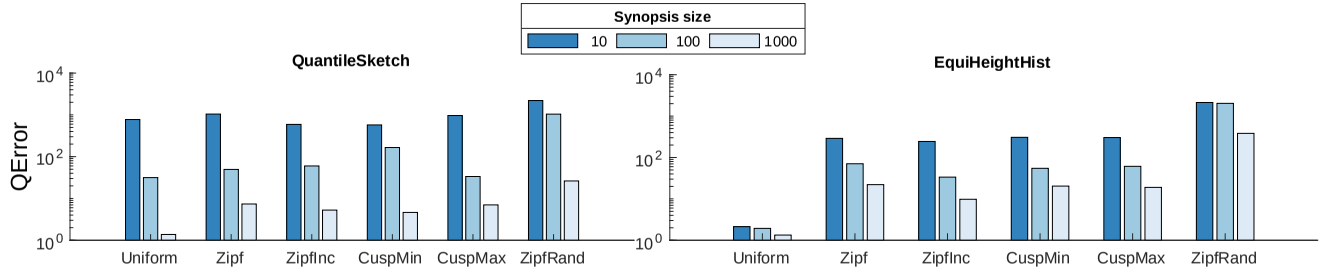
We have extended the earlier proposed framework for LSM-based statistics-collection and enabled the gathering of statistics on unordered fields. We did so by incorporating the Greenwald-Khanna sketch as another local synopsis-gathering algorithm. We have thoroughly studied the effects of non-ordered statistics and identified the tradeoffs between the number of fields for which we collect statistics, the number of quantiles allocated to the synopsis (i.e. accuracy), and the overhead of ingestion. We have also demonstrated that the approximate sketch-based statistics method is comparable to the exact index-based algorithm in terms of cardinality estimation accuracy. Our experiments have demonstrated that using a sketch with 1000 quantiles does not have a significant impact on ingestion time, while it is still able to provide accuracy comparable to the index-based algorithm.



(a) Uniform frequencies



(b) Zipf frequencies



(c) ZipfRandom frequencies

Figure 4.8: Average q-error of measuring estimation accuracy for Greenwald-Khanna sketch and equi-height histogram while the number of buckets/quantiles is increased from 10 to 1000.

Chapter 5

A Comparative Study of In-Memory Analytical Operations

5.1 Introduction

Business decisions are nowadays driven by advanced database analytics and companies thrive on how quickly and how well they can analyze the available data. Recent hardware trends, including the proliferation of parallel architectures and the rapidly decreasing cost of RAM, have created a niche for in-memory analytics solutions. As a result, many companies, as well as academic institutions developed databases that operate on the data stored entirely in main memory (Oracle TimesTen [77], SAP HANA [53], MS SQL Hekaton [49], IBM BLU [24], MemSQL [100], HyPeR[74], Peloton[4]). These systems rely on efficient query processing algorithms optimized for in-memory workloads which allows them to significantly boost the overall performance.

While memory capacity continues to increase, the past decade has seen a stagnation of processor clock speeds caused by the end of the Dennard scaling. This leaves parallelism as the only option to allow fast processing for the growing amounts of memory-resident data. The computer

architecture community considered two approaches to leverage this parallelism, namely (i) off-the-shelf multi-core architectures, including CPUs and GPUs, [54] or (ii) customizable architectures such as CPUs with FPGAs [1, 94, 2, 86, 108, 98]. While multi-cores typically have much higher clock speeds, specialized hardware has both the advantages of customization (the hardware design is optimized for a specific application) and parallelism.

The major issue limiting the performance of in-memory algorithms is the growing gap between the memory bandwidth and the speed of the processing unit (the so-called *memory wall*), which is even more important for multi-cores given their higher clock speeds. The multi-core approach addressed this problem by introducing large *cache hierarchies*, relying on the data locality (spatial and/or temporal) to mitigate memory latency. This solution does not come for free: cache hierarchies can take up to 80% of the chip area thus are becoming the main factor limiting the number of cores that can be accommodated on a single chip, as well as the primary contributor to energy consumption through leakage current. GPUs offer a different solution leveraging on massive SIMD parallelism and high bandwidth specialized memory (i.e. GDDR). Similarly, modern CPUs introduced SIMD support to enable data level parallelism. However these architectural solutions still inherently rely on data locality.

At the same time, many data-intensive computational problems are moving away from data locality towards irregular memory access behavior. Such irregularity can be of two types: (1) *Dataflow irregularity* is caused by the indirection in the data access, leading to cache (data or TLB) misses. For example, popular hash-based database analytics algorithms (joins, group-by aggregations) exhibit a very poor degree of spatial and/or temporal localities and do not benefit from large cache hierarchies [39]. (2) *Control flow irregularity* is caused by the dynamic control flow and leads to branch misprediction, which contributes a large fraction of stall time on CPUs [96, 111] or thread divergence on GPUs [102]. Examples in this category include transaction processing and evaluation of multi-predicate selection conditions.

Rather than relying on a cache hierarchy, *hardware multithreading* aims to completely mask high memory latency. This model relies on spawning a large number of treads that are not backed by the physical core. Only one thread is running at each moment in time and the execution is relinquished as soon as the thread performs a long-latency operation. The executing thread is then suspended until the long-latency operation completes and eventually returns to a ready state again. This approach has been used in CPUs (Hyper-Threading, UltraSparc [54]), however, it could support only a relatively small number of threads because the CPU has to provision a full hardware context for each ready/waiting thread, thereby limiting the amount of parallelism. In a custom architecture (e.g., based on reconfigurable FPGA fabric or application-specific hardware) where the datapath is designed for a small number of predefined operations, the required context for each thread is much smaller than in a general-purpose CPU and hence more threads can be supported. In this hardware multithreaded model, the parallelism is limited only by the number of active threads (ready, executing or waiting).

Recently there has been a surge in research papers proposing new CPU algorithms for basic in-memory database operations [75, 26, 22, 21, 13, 40, 41, 119, 96, 30, 92]. However comparing the performance of these CPU algorithms to custom architectures, such as hardware multithreading, is still an open problem. The key contribution of the following chapter is performing a comparative study of the state-of-the-art software algorithms that implement common database analytical operations for in-memory workloads. We provide an empirical evaluation of various performance characteristics (throughput, scalability, power efficiency) of these algorithms and compare them with the customized reconfigurable FPGA implementation that leverages multithreaded design.

The remainder of this chapter is structured as follows. First, we start by reviewing the related work. Next, we describe the common principles and designs of the FPGA-based multithreaded implementation. In the subsequent sections, we provide details of the in-memory CPU algorithms for (i) hash joins, (ii) hash group-by aggregations and (iii) selections respectively and compare their performance with the FPGA-based hardware approaches. Finally, we provide the conclusions.

5.2 Related Work

Many recent works consider the in-memory implementation of joins (hash or sort-merge). Manegold et al. [80] presented the first work that emphasized the importance of TLB misses in partitioned hash joins and proposed a radix clustering algorithm to keep the partitions cache resident. Later Blanas et al.[26] studied the performance of hash joins by comparing simple hardware-oblivious algorithms and *hardware-conscious* approaches (since the radix clustering algorithm is tightly tailored to the underlying hardware architecture). Results showed that the simple implementations surpass approaches based on radix clustering. However recently, Balkesen et al. [21] applied a number of optimizations and found that hardware-conscious solutions in most cases are prevalent over hardware-oblivious.

The implementation of sort-merge joins on modern CPUs was studied by Kim et al. [75]. This paper explored the use of SIMD operations for sort-merge joins and hypothesized that its performance will surpass the hash join performance, given wider SIMD registers. Subsequent work [13] implemented a NUMA-aware sort-merge algorithm that scaled almost linearly with the number of computing cores. This algorithm did not use any SIMD parallelism, but it was reported to be already faster than its hash join counterparts. Recently, Balkesen et al. [22] reconsidered the issue and found that hash joins still have an edge over sort-merge implementations even with the latest advance in the width of SIMD registers and NUMA-aware algorithms.

Hardware-oblivious implementations of the group-by aggregation were explored by Cieslewicz et al. [40], who showed that performance largely depends on input characteristics (key cardinality). Follow up work [41] explored the partitioning step of hash aggregation and concluded that the thread coordination is a key component influencing the performance of this step. In addition, Ye et al. [119] proposed hybrid algorithms and showed that they outperform pure hardware-conscious and -oblivious implementations. Finally, Wang et al. [115] described a NUMA-aware hardware-conscious

in-memory hash aggregation algorithm, which avoids cache coherency misses and minimizes locking costs.

In addition to complex analytics operations the problem of efficient evaluation of selection operation also have brought attention of multiple researchers. A seminal work by Ross [96], described tradeoffs of implementing conjunctive selection conditions for row-oriented data on CPUs. It showed that in due to dynamic control flow simple short circuiting-based implementation of conjunctive predicate might not always yield the best performance due to branch misprediction penalties. Polychroniou et al. [91] explored the same problem in a column-oriented storage model and showed that SIMD algorithms significantly outperform earlier scalar implementations. Finally [30] performed a through experimentation with various variants of selection operator implementations, demonstrated importance of number of predicates in addition to individual predicate selectivity. Similarly, Sitaridi et al. [102] evaluated performance of GPU selection kernel and demonstrated that it also suffered from a dynamic control flow due to branch diversion of threads in GPU thread block.

5.3 Introduction to Hardware Multithreaded Design

In this section we describe the overall design of the hardware multithreaded architecture implemented on FPGAs.[1]. We are assuming that the input relation fits in main memory but is too large to fit locally on the FPGA's memory. To fully utilize the memory bandwidth available to the FPGA we employ a hardware multithreaded model, which allows the FPGA to process ready jobs while idle jobs wait on (long) memory accesses. Figure 5.1 depicts a main elements of multithreaded FPGA engine. At each given moment in time a single thread is processed in the *FPGA datapath*. In addition FPGA maintains two queues of *ready threads* and *waiting threads* that can be accessed in a single clock cycle. Whenever a thread issues a memory request the FPGA saves the thread's state into *waiting threads* queue and picks up the next ready job. Once a memory request is fulfilled the thread state is updated, and queued back into the *ready threads* FIFO. If the FPGA can

[1] All FPGA design and implementations were performed by Robert Halstead, Prerna Budhkar and Skyler Windh

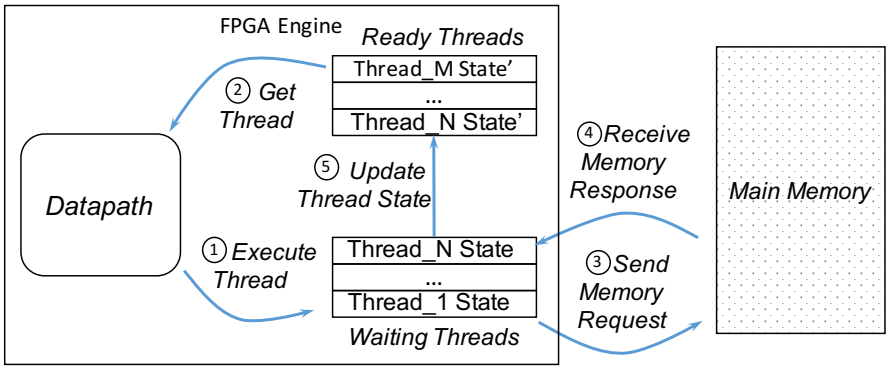


Figure 5.1: Multithreaded model implemented on FPGAs.

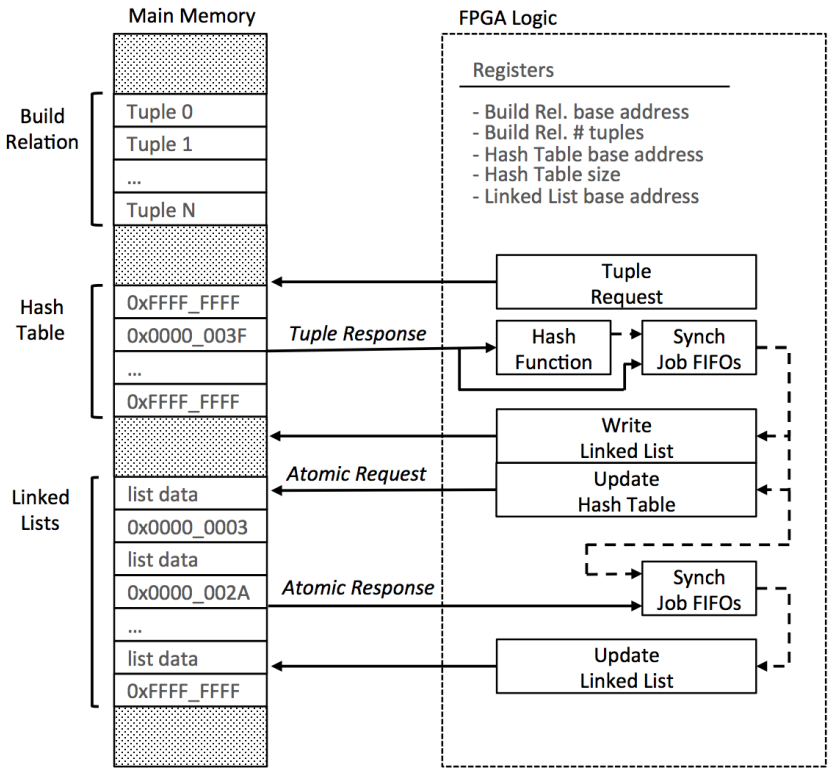


Figure 5.2: The FPGA datapath for building the hash table.

maintain more thread states than the memory latency then full latency masking is achieved, thus the bandwidth is fully utilized.

The key component of the multithreaded design is application-specific datapath, which includes a series of operations executed in a pipelined fashion. Figure 5.2 shows the state diagram

of a thread that builds a hash table, which is a building block of hash join and hash group-by aggregation algorithms. The hash table data structure uses a traditional bucket chaining design with an array of buckets pointing to a number of singly-linked lists. In the FPGA logic, local registers are programmed at runtime and hold pointers to the relation, hash table, and linked lists. They also hold information about the number of tuples, tuple sizes, and the join key position in the tuple. Lastly, the registers hold the hash table size, which is used to mask off results from the hash function.

Initially, the *Tuple Request* component will stream each relation's tuple from memory, request its join key, assign a separate FPGA thread (job) and start its pipelined execution. The design assumes the join key size is between 1 and 8 bytes, and it is set at runtime with a register. The actual tuple can be of arbitrary size. Requests are continually issued until all tuples have been processed, or the memory architecture stalls. When a thread issues a request the tuple's pointer is added to the thread state, and the thread goes idle.

As join key requests are completed, the thread is activated, and the key along with its hash value is stored in the thread's state. The *Write Linked List* component writes the key and tuple pointer to a new node into the appropriate bucket linked list. The *Update Hash Table* component issues an atomic request to read and update the hash table. The old bucket head pointer is read while the new node pointer replaces it. An atomic request is needed here because a single engine can have hundreds of threads in flight, and issuing separate reads and writes would create race conditions. While the atomic request is issued the new node pointer is added to the thread's state.

As the atomic requests are fulfilled the thread is again activated, and the *Update Linked List* component updates the bucket chain pointer. If no previous nodes hashed to that location then the atomic request will return the empty bucket value, which is used to signify the end of a list chain. Otherwise, the old head pointer is used to extend the list.

All experiments presented in this chapter are performed on a specialized Convey HC-2ex heterogeneous machine that offers a shared global memory space between the CPU and FPGA

regions. Each processing unit (CPU or FPGA) can access data from both memory regions, but remote data accesses go across PCIe and have significantly longer latency. We chose the Convey HC-2ex as our target platform because it allows direct comparison between hardware and software implementations on the same memory architecture. In addition, it has 4 FPGAs with direct high bandwidth (19.2 GB/s) access to 64 GB of 1600 MHz DDR3 memory. The software region of Convey HC-2ex has 2 Intel Xeon E5-2643 processors running at 3.3 GHz with a 10 MB L3 cache which are connected to local 128 GB of 1600 MHz DDR3 memory. Each CPU processor has a peak memory bandwidth of 51.2 GB/s. To perform a fair comparison, we run our experiments on 2 FPGAs to match memory bandwidth as close as possible (38.4 GB/s for the FPGA vs 51.2 GB/s for the CPU) or normalize obtained throughput to the unit of bandwidth.

5.4 The Join Operator

In the following section, we describe software implementations of the in-memory relational join operator and compare their performance with the FPGA-based multithreading approach [2].

5.4.1 Software Implementations

As the state-of-the-art multi-core hash join approach we use the implementation from [21]. It includes 2 types of hash join algorithms:

- Hardware-oblivious non-partitioned join
- Hardware-conscious partitioning-based algorithm.

Both implementations perform the traditional hash join with build and probe phases, however the main difference lies in the way they are utilizing multi-core CPU architecture.

The non-partitioned approach builds a single hash table which is shared among all threads, therefore requiring explicit synchronization. In addition, this algorithm does not make any hardware-

[2] All FPGA performance measurements were done by Robert Halstead

specific choices relying only on hyper-threading to mask cache miss and thread synchronization latencies, similarly to the FPGA multithreading approach.

On the other hand, the partitioning-based algorithm introduces a preliminary step to divide the input table and avoid contention among executing threads during the build phase. Later during the join operation, each thread will process a single partition without explicit synchronization. The *Radix clustering* [29] algorithm, which is a backbone of the partitioning stage needs to be parameterized with the number of TLB entries and cache sizes, making the approach hardware-conscious. In our experiments, we use a two-pass clustering and produce 2^{14} partitions, which yields the best cache residency for our CPU.

5.4.2 Dataset Description

Our experimental evaluation uses 4 datasets. Within each dataset, we have a collection of build and probe relations ranging in size from 2^{20} to 2^{30} elements. Each dataset uses the same 8-byte wide tuple format, commonly used for performance evaluation of in-memory query engines [21, 29, 26]. The first 4 bytes hold the join key, while the rest is reserved for the tuple's payload. Since we are only interested in finding matches (rather than joining large tuples), our payload is a random 4-byte value. However, it could just as easily be a pointer to an actual arbitrarily long record, identified by the join key.

The first dataset, termed *Unique*, uses incrementally increasing keys which are randomly shuffled. It represents the case when the build relation has no duplicates, thus keys in the hash table are uniformly distributed with exactly one key per bucket (assuming simple modulo hashing). The next dataset (*Random*) uses random data drawn uniformly from a 32-bit integer range. Keys are duplicated in less than 5% of the cases for all build relations having less then 2^{28} tuples. The largest relations have no more than 20% duplicates. For this dataset, bucket lists average 1.6 nodes when the hash table size matches the relation size, and 1.3 nodes when the hash table size is double the relation size. The longest node chains have about 10 elements regardless of the hash table size. To

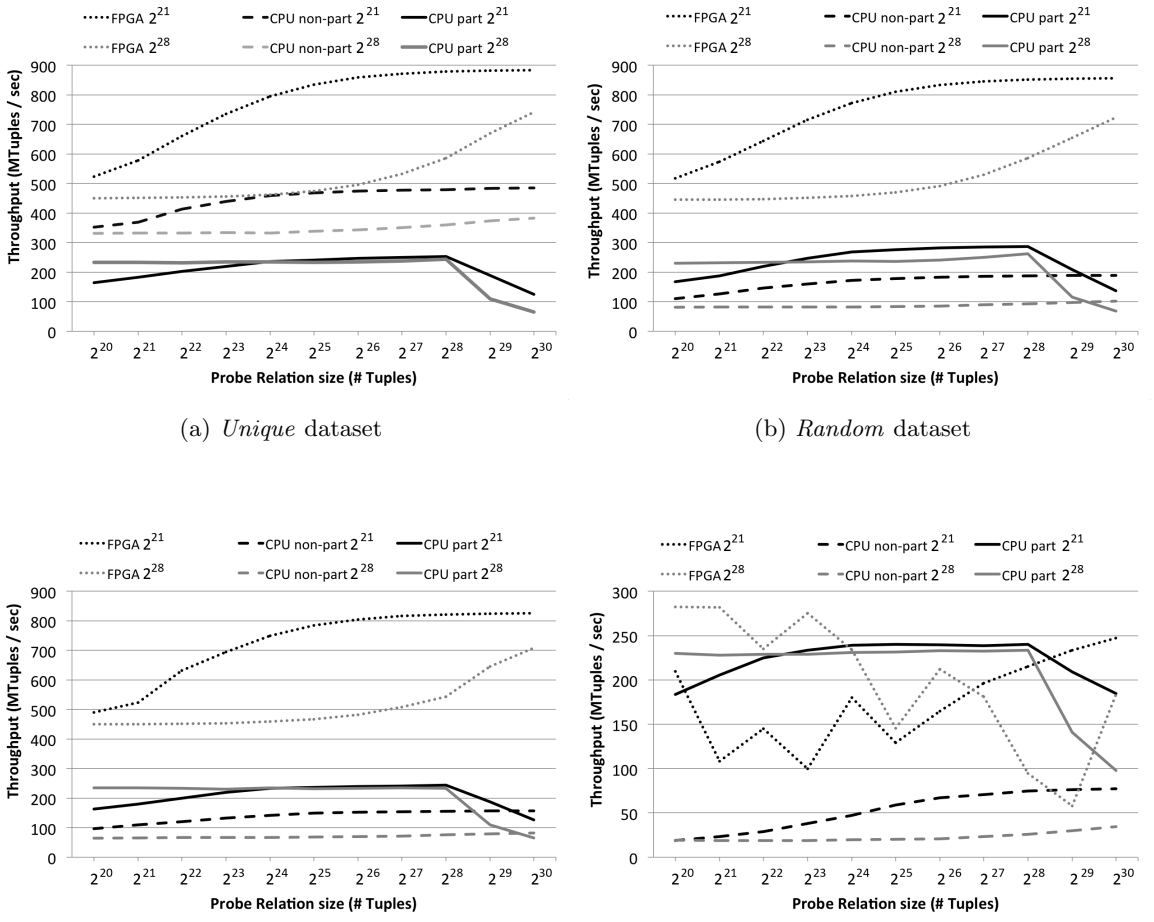
explore the performance on non-uniform input, the keys in the final two datasets are drawn from a Zipf distribution with coefficients 0.5 and 1.0 (*Zipf_0.5* and *Zipf_1.0* respectively); these datasets are generated using the algorithms described by Gray et al. [66]. In *Zipf_0.5* 44% of the keys are duplicated in the build relation. The bucket list chains have on average 1.8 keys regardless of the hash table size, while the largest chains can contain thousands of keys. In *Zipf_1.0* the build relations have between 78% and 85% of duplicates. Their bucket list chains have on average from 4.8 to 6.7 keys. The longest chains range from about 70 thousand keys in the relation with 2^{20} tuples to about 50 million in the 2^{30} relation.

5.4.3 Experimental Evaluation

5.4.3.1 Throughput

We report the multi-core results for both partition-based and non-partitioned algorithms. Results are obtained with a single Intel Xeon E5-2643 CPU, running on full load with 8 hardware threads. However because of the memory-bounded nature of hash join we use two FPGAs to offset the CPUs bandwidth advantage: a single CPU has 51.2 GB/s of memory bandwidth while two FPGAs have 38.4 GB/s (even with this bandwidth adjustment, the CPU still has almost a 30% advantage). By matching the bandwidth be can get a more accurate comparison between the approaches. Obviously, given of the parallel nature of hash join, the CPU and FPGA performance could easily be improved by adding more hardware resources.

Figure 5.3 shows the join throughput for two build relations, with 2^{21} and 2^{28} tuples respectively, while increasing the probe relation size from 2^{20} to 2^{30} for all datasets mentioned in Section 5.4.2. The FPGA performance shows two plateaus for the *Unique*, *Random* and *Zipf_0.5* data distributions on Figures 5.3a, 5.3b, and 5.3c. The FPGA sustains throughput of 850 MTuples/s when the probe phase dominates the computation (that is when the size of the probe relation is much larger than the size of the build relation) and it is close to the peak theoretical throughput of 1200



(a) *Unique* dataset

(b) *Random* dataset

(c) *Zipf_0.5* dataset

(d) *Zipf_1.0* dataset

Figure 5.3: Dataset throughput as the build relation size is increased.

MTuples/s which can be achieved with 8 engines on 2 FPGAs. When the build phase dominates the computation, atomic operations restrict FPGA throughput to about 450 Mtuples/s (in the FPGA 2^{28} plot, the throughput stays almost constant until the probe relation becomes comparable in size to the build relation). Clearly, in real-world applications, the smaller relation should be used as the build relation. In the worst case, we can expect FPGA throughput to be 600 MTuples/s when both relations are of the same size. For the extremely skewed dataset, *Zipf_1.0*, (shown in Figure 5.3d) the FPGA throughput decreases significantly and varies widely depending on the specific data. This happens because extremely long bucket chains create a lot of stalling during the probe phase that greatly affects throughput.

The CPU results are consistent with numbers reported in [21]. The partitioned algorithm peak performance is around 250 MTuple/s across all datasets, regardless of whether the computation is dominated by the build or the probe phase. It is also not affected by the data skew. For the non-partitioned algorithm, the throughput depends on the relative sizes of the relations, since like in the FPGA case, the throughput of the build phase is lower than the probe phase. The non-partitioned algorithm behaves always worse than the FPGA approach. Interestingly, for the *Unique* dataset, the non-partitioned version has better throughput than the partitioned one, because the bucket chain lengths are exactly one. As the average bucket chain length increases (moving from the *Unique* to the *Random* to the skewed datasets) the throughput of non-partitioned approach decreases. For the extremely skewed *Zipf_1.0* dataset, it falls approximately to 50 MTuples/s.

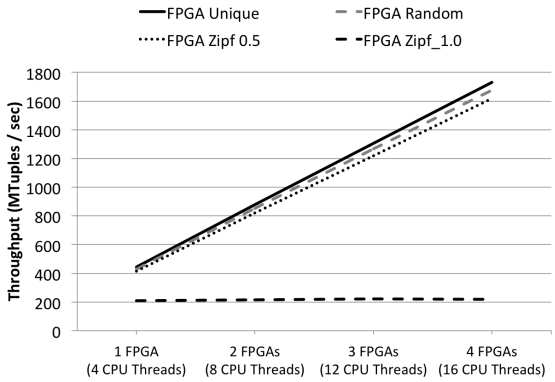
Averaging the data points within all datasets yields the following results: the FPGA shows a 2x speedup over the best CPU results (non-partitioned) on *Unique* data, and a 3.4x speedup over the best CPU results (partitioned) on *Random* and *Zipf_0.5* data. The FPGA shows a 1.2x slowdown compared to the best CPU results (partitioned) on *Zipf_1.0* data.

5.4.3.2 Scalability

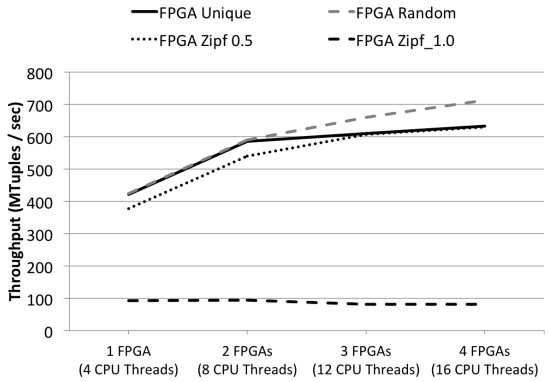
To examine scalability, in the next experiments we attempt to match the bandwidth between software and hardware as closely as possible: every 4 CPU threads are compared to one FPGA (note that this still provides a slight advantage to the CPU in terms of memory bandwidth). We examine two cases, when the probe relation is much larger than the build one, and when they are of equal size.

Figures 5.4a,5.4c and 5.4e show the results when the probe phase dominates the computation. The FPGA scales linearly on datasets *Unique*, *Random* and *Zipf_0.5* (Figure 5.4a).

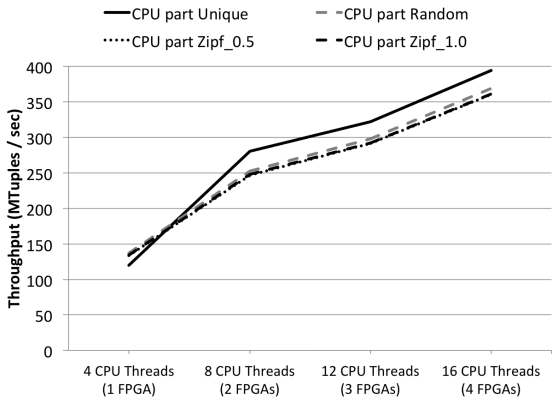
However, for the *Zipf_1.0* dataset, the performance does not scale because of the extreme skew. Each probe job searches through an average of 4.8 to 6.7 nodes in the linked list. Therefore most jobs are recycled through the datapath multiple times. Having too many jobs being recycled



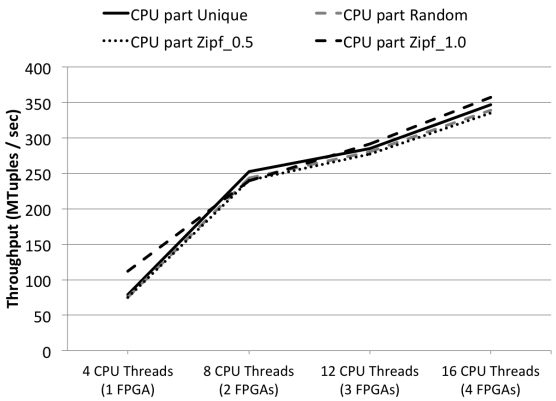
(a) FPGA scalability when build relation has 2^{21}, probe relation has 2^{28} tuples



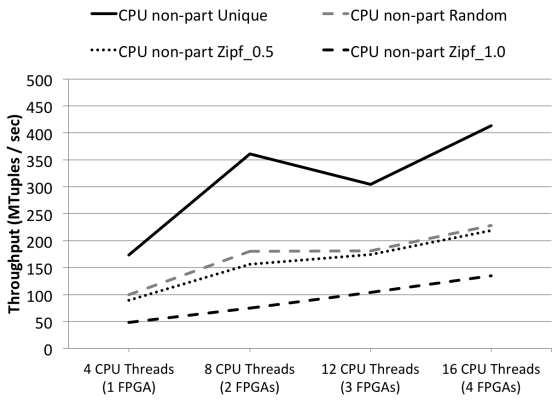
(b) FPGA throughput scalability when build and probe Relations both have 2^{28} tuples



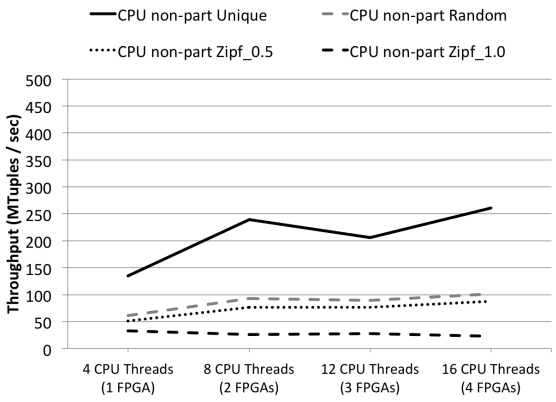
(c) Partitioned CPU throughput scalability when build Relation has 2^{21}, probe relation has 2^{28} tuples



(d) Partitioned CPU throughput scalability when build and probe relations both have 2^{28} tuples

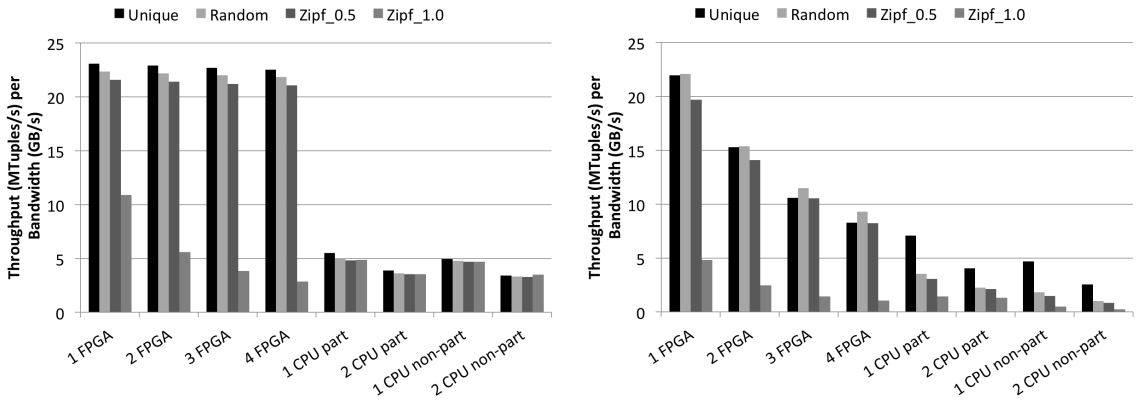


(e) Non-partitioned CPU throughput scalability when build relation has 2^{21}, probe relation has 2^{28} tuples



(f) Non-partitioned CPU throughput scalability when build and probe relations both have 2^{28} tuples

Figure 5.4: FPGA (a-b), Partitioned CPU (c-d) and non-partitioned CPU (e-f) throughput comparison as the bandwidth and number of threads are increased.



(a) Build Relation has 2^{21}, Probe has 2^{28} tuples (b) Build and Probe Relations both have 2^{28} tuples

Figure 5.5: Throughput efficiency.

limits the new jobs entering the datapath causing back pressure and stalling. The partitioned CPU algorithm scales as the number of threads increases but at a lower rate than the FPGA approach (depicted in Figure 5.4c). The non-partitioned software algorithm shows a drop in performance while moving from 8 to 12 threads because of the NUMA latency emerging while moving from 1 to 2 CPUs (Figure 5.4e).

The FPGA scales at a lower rate when the build and probe relations are of the same size (Figure 5.4b), since the throughput of the build phase remains constant while the probe phase scales. The slope of the scale graph is almost comparable to the CPU implementations (shown on Figures 5.4d and 5.4f) Again the extreme skew case does not scale for the FPGA.

5.4.3.3 Throughput Efficiency

To get a direct comparison of throughput we normalize it to the available bandwidth. As it was mentioned earlier each FPGA has 19.2 GBs of bandwidth, while each CPU has 51.2 GBs. The normalized results are shown in Figure 5.5.

When the probe relation dominates the computation (Figure 5.5a) the FPGA shows speedup between 3.2x and 6x on the *Unique* dataset. It shows a speedup between 4.4x and 10x on the *Random* and *Zipf_0.5* datasets. Finally, it shows speedup between 0.6x and 8.3x on the *Zipf_1.0* dataset.

When neither relation dominates the computation (Figure 5.5b) the FPGA shows speedup between 1.7x and 8.6x on the *Unique* dataset. It shows a speedup between 1.7x and 23.1x on the *Random* and *Zipf_0.5* datasets. Finally, it shows speedup between 0.2x and 21.6x on the *Zipf_1.0* dataset.

5.5 The Group-by Aggregation

Despite the similarities with the hash join covered in Section 5.4, the two operators are using the hash table in a very different manner: the hash join has a clear delineation between the build phase, when the hash table is modified, and the probe phase during which the table is only read. In the group-by aggregation the read- and write-requests are instead mixed in a single phase. Moreover during the build phase of the hash join, a key will always create a new node in the appropriate bucket list. For the aggregation, a key is first searched within the appropriate bucket list and then it either updates an entry value (if this key has been found) or inserts a new entry into bucket list. All these dissimilarities become especially important in the multithreaded environment, when explicit synchronization is needed to guarantee correctness, leading to different optimization strategies for the two hash-based operators. For FPGAs we consider two different multithreaded designs: regular and multiplexed [3]. Multiplexing efficiently reuses parts of the FPGA datapath, increases the number of engines we can put onto a single FPGA and leverages inter-engine parallelism to increase memory utilization.

5.5.1 Software Implementations

We implemented the following state-of-the-art multithreaded software aggregation algorithms: (i) Independent Tables[40], (ii) Shared Table [40], (iii) Hybrid Aggregation [40], (iv) Partition with Local Aggregation Table [119] and (v) Partition & Aggregate [119]. Here, (i) and (ii) are

[3] All FPGA performance measurements were done by Robert Halstead and Prerna Budhkar

considered as non-partitioned implementations, (v) is a partitioning-based algorithm while (iii) and (iv) are hybrid approaches.

- **Independent Tables** is the approach most similar to the hardware implementation. The tuples are evenly split among separate software threads (without partitioning), and each thread aggregates the result into its own hash table. Once the aggregation is complete all tables are merged together, which requires write synchronization.

- **Shared Table (with locking or atomic synchronization)** splits the tuples evenly between threads, but all threads aggregate their results into a single hash table, hence no extra merge step is required. The algorithm could use different synchronization primitives: either mutexes or hardware atomic instructions. Preliminary experiments showed that atomic primitives are significantly better on low key cardinalities and do not have any difference from mutexes on medium and large cardinalities, so we choose atomics as a default synchronization primitive in all further experiments.

- **Hybrid Aggregation** is a combination of two previous approaches. This algorithm allocates a small hash table for each thread. The size of the table is calculated based on the processor's L2 size to avoid cache misses. If the local table has enough space for a new value, or the value already exists in the table, that tuple is locally aggregated. Once the local table is filled and the next tuple requires a new slot, the oldest entry in the cached table will be spilled into the larger shared table residing in the main memory, thus maintaining only "hot" data in the L2 cache. Once aggregation is complete all small cached tables are merged into the large shared table. Merge step is synchronized as in **Independent Tables** case.

- **Partition & Aggregate** (also known as count-then-move [41]) uses individual tables per thread, but before aggregation is performed the tuples are partitioned, in contrast to all aforementioned approaches. Separate partitioning step makes sure that all threads will work on non-overlapping values, hence aggregation could be done without any synchronization and

the final tables are simply concatenated, rather than merged. As with the partitioned join implementations, the *radix clustering* [29] algorithm is a backbone of this preliminary step.

- **PLAT (Partitioning with Local Aggregation Table)** is a combination of two previous techniques. The algorithm takes advantage of the fact that we are performing an additional data scan while doing a preprocessing step. While partitioning tuples into groups with mutually exclusive keys, each thread tries to aggregate values into its own small L2-resident table, as in *Hybrid Aggregation* approach. Values that do not fit into the small table are partitioned using *radix clustering* algorithm. Once preprocessing is done standard lock-free aggregation is applied. In the end, all tables that were produced during aggregation are concatenated together, while local aggregation tables are synchronously merged in.

5.5.2 Dataset Description

We use five datasets with various s key distributions, namely: Uniform, Heavy Hitter, Moving Cluster [40], Self Similar and Zipf_0.5.

- In the **Uniform** dataset, all key values are picked from *uint64* key range with uniform probability. After that generated key/value pairs are randomly shuffled.
- A half of the tuples in the **Heavy Hitter** dataset share the same a key value. The remaining key values are picked uniformly and evenly distributed throughout the entire relation.
- In the **Moving Cluster** dataset, tuples are grouped into clusters depending on their key values. Lower key values are more likely to appear at the beginning of the relation, whereas tuples with higher key values tend to appear at the end of the relation.
- **Self Similar** uses Pareto rule to model key distribution in a dataset: a single key value is shared by 20% of the tuples. Of the remaining 80% of tuples, 20% of those share another key value. This process is repeated recursively to generate the relation. Tuples are randomly shuffled. The generation algorithm is described by Gray et al. [66].

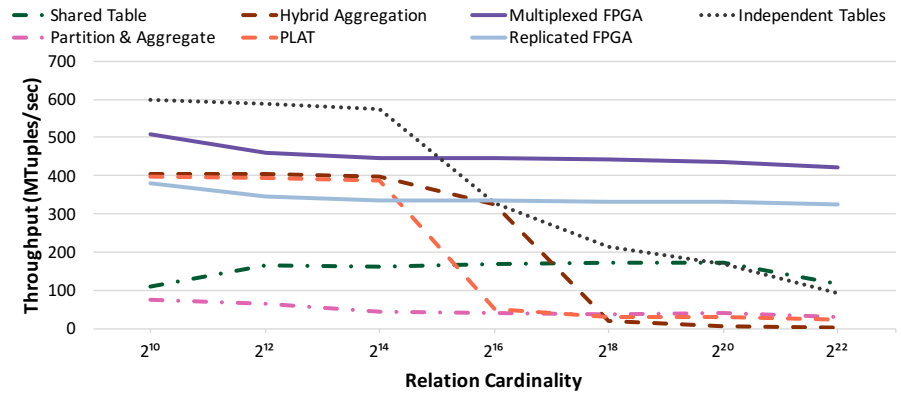
- In the **Zipf** dataset key values follow the Zipf distribution with a skew coefficient of 0.5. The generation algorithm appears in aforementioned work[66].

Each dataset consists of several benchmarks with cardinalities ranging from 2^{10} to 2^{22} unique keys. The relation size in all of the experiments was 256 million tuples (in line with previous research [119]). Each dataset used the same 8-byte wide tuple format as in join experiments mention earlier in Section 5.4. The first 4 bytes of the tuple hold the unique primary key, while the rest is reserved for the grouping key. Since we are only interested in counting records with the same grouping keys, our tuples do not store any other information. However, none of the design choices prevent the use of "wide" tuples (i.e. containing fields other than primary and grouping keys). This could be easily supported by adding a key extraction component into the FPGA design. Moreover experimenting with such "skinny" tuple format yields the best performance for software implementations, since it minimizes the cache capacity misses, which would decrease caching effectiveness otherwise.

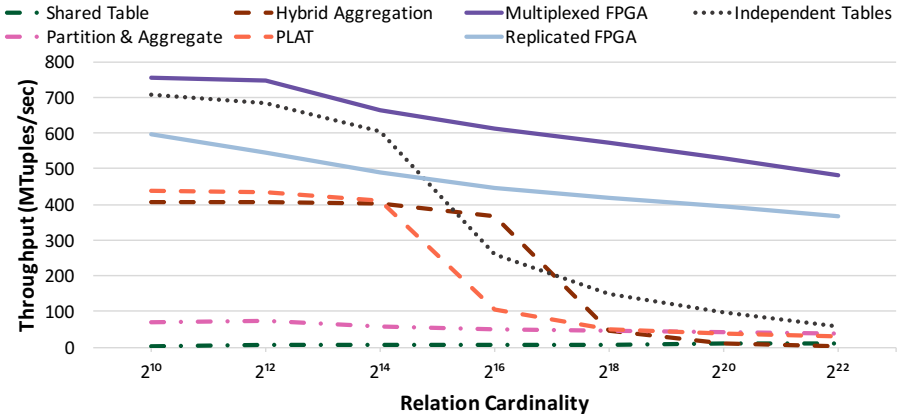
5.5.3 Experimental Evaluation

5.5.3.1 Throughput

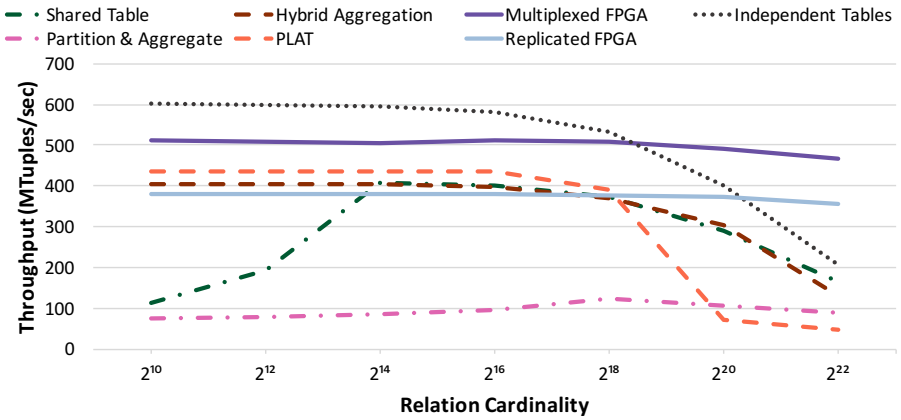
Figure 5.6 displays the throughput of the group-by aggregation as the key cardinality is increased, obtained for various datasets. Throughput was measured across two FPGA engine designs (regular and multiplexed), and five software (two non-partitioned, two hybrid and one partitioned) implementations. Throughput for skewed Heavy Hitter dataset Figure 5.6d resembles the results for Self Similar dataset Figure 5.6b, while the throughput for moderately skewed data Zipf_0.5 5.6e is similar to the results obtained for Uniform dataset Figure 5.6a. Software implementations demonstrate the best performance on Moving cluster dataset Figure 5.6c due to the property of the data distribution: similar grouping keys appear in the input stream clustered together, increasing CPU-cache hit rates.



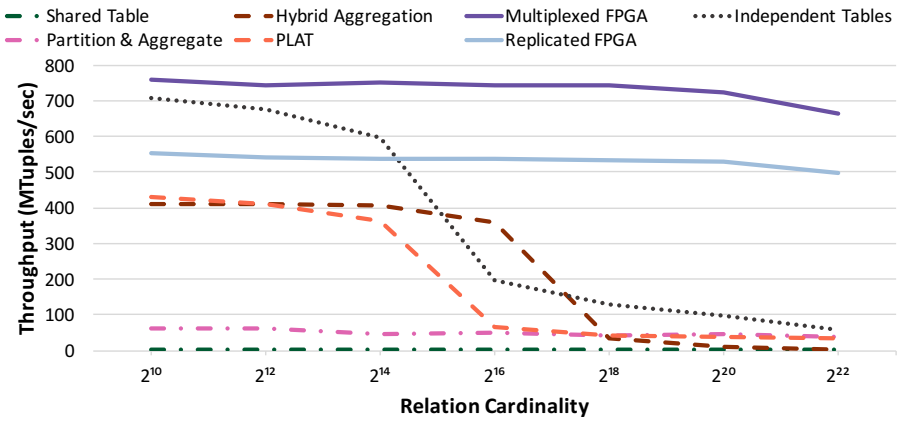
(a) Uniform



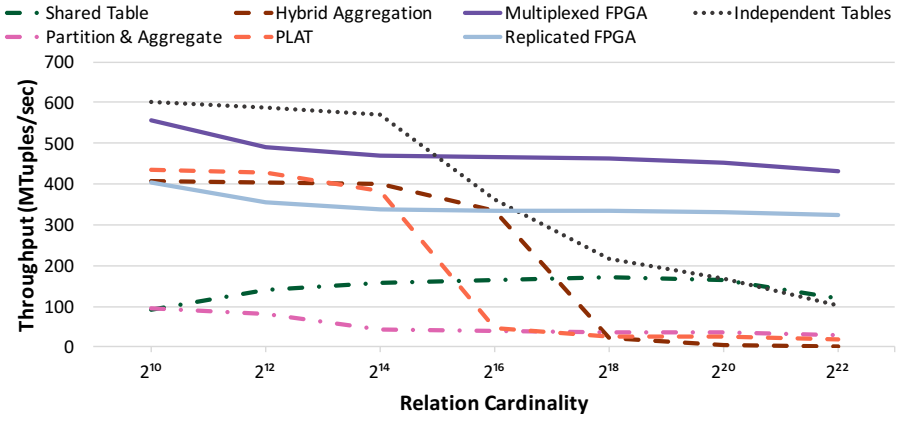
(b) Self Similar



(c) Moving Cluster



(d) Heavy Hitter



(e) Zipf 0.5

Figure 5.6: Aggregation throughput of hardware and software approaches for datasets with 256M tuples.

Despite all the differences in data distribution CPU aggregation performance mainly depends on the dataset's key cardinality. While the number of unique keys is low, hash tables can fit into the CPU cache entirely. However, as the cardinality increases, cache misses start to hamper the throughput due to high latency memory round-trips. Software performance severely deteriorates at cardinalities higher than 2^{18} on all datasets for all algorithms. Another trend, which appears in all experiments, is that the Independent Tables approach yields the best result across all software algorithms. Nevertheless, that algorithm exhibits poor scalability, since the amount of memory needed for aggregation processing grows linearly with the number of parallel threads and the key cardinality. As the number of parallel threads increases, the amount of available memory could quickly become a bottleneck. We could also see that hybrid algorithms (PLAT and Hybrid Aggregation) outperform traditional partitioned (Partition & Aggregate) and non-partitioned (Shared Table) approaches by amortizing the cache miss cost and sustain a throughput around 400 MTuples/sec. This trend continues for cardinalities up to 2^{16}, which marks the end of L3-cache residency. After that point, the performance advantage of hybrid algorithms vanishes and drops below 100 MTuples/sec.

The FPGA performance also drops as the key cardinality increases, however this effect is much less profound. Unlike the software throughput, this result is explained by the overhead, introduced by the post-processing merge step. However the overall performance is still up to 10x higher than the software throughput. The results also clearly show the benefits of the multiplexed engine design. Typically the throughput of the multiplexed FPGA engine is up to 30% more than the initial design. It should be also noted that the FPGA throughput does not deteriorate on heavily skewed data (*Self Similar*), as it was the case with the hash join [70].

5.5.3.2 Memory Bandwidth Usage

It should be noted that the performance benefits of the FPGA-based approaches does not come not architecture-specific features, but from multithreading, which allows to utilize the available memory much better than any of the software implementations. Figure 5.7 depicts the ratio of

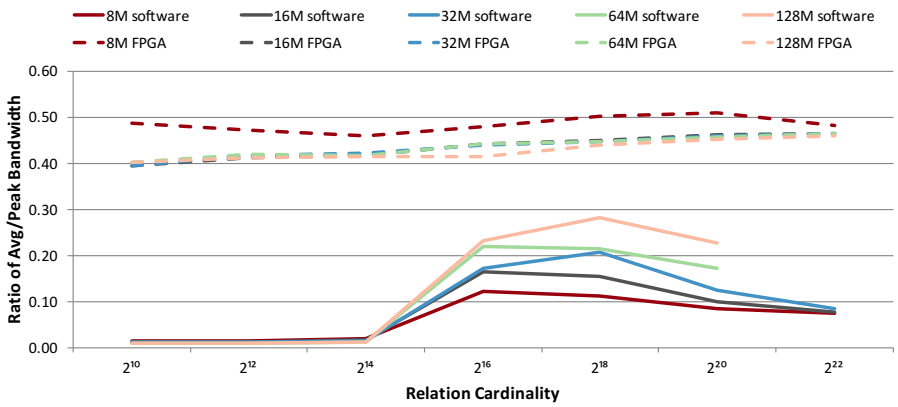


Figure 5.7: Ratio of average effective memory bandwidth to peak theoretical bandwidth achieved by the Independent Tables software algorithm and the Multiplexed FPGA design for varying dataset sizes and key cardinalities.

effective average memory bandwidth to peak theoretical memory bandwidth for the best software (Independent Tables) and FPGA (multiplexed) implementations while varying dataset sizes and key cardinalities. Hardware multithreading approach allows the FPGA implementation to keep the ratio almost constant, irrespectively of dataset size or key cardinality. On the contrary, the ratio for the software approach varies greatly. The effective memory bandwidth of the CPU implementation tends to grow as the size of the relation increases (from 8M to 128M), whereas the FPGA-based approach is less susceptible to data size variations. For low cardinality the aggregated relation and hash table are cached and there are almost no memory accesses, hence the ratio approaches 0. The software ratio peaks at around 0.3 for cardinality 2^{18}, but drops significantly for higher key cardinalities. For very large cardinalities the FPGA implementation ratio is almost 5 times higher.

5.6 The Selection Operator

We proceed with algorithms implementing the selection operator on CPU as well as GPU architectures and present an experimental study of its performance compared to an FPGA multi-threaded implementation [4].

[4] All FPGA performance measurements were done by Prerna Budhkar, while the GPU performance evaluation was done by Vasileios Zois

Here we assume that selection operates directly on the input array of records (i.e. is not pipelined to an input of an earlier operator) and materializes qualified tuples into a new output array. This is a common scenario for query plans where the selection is pushed all the way down to the data scan operator. For each tuple t_i, we will be evaluating k predicates. There are 3 commonly used selection evaluation algorithms: (i) branching scan, (ii) bitwise-AND and (iii) no-branch [96].

- **Branching scan** is the most straightforward way of implementing selection. Listing 5.1 shows this method for a conjunctive query with '<' comparison.

```
for (i=0; i < number_of_tuples; i++)
  if((t_i[0] < v_1) && (t_i[1] < v_2) && ... (t_i[k] < v_k))
  { out[j++] = i; }
```

Listing 5.1: Algorithm - Branching Scan

The evaluation continues until one of the attributes is assessed to be *false* or all the attributes of a query have been examined. This technique is often called a 'short-circuit evaluation' because the computation of further predicates can be skipped when the first predicate is already evaluated to false. The logical-AND operator (&&) is typically compiled into k conditional branch instructions. Assuming that the predicates have increasing selectivity, this method is optimal in terms of processing cycles. However, it was shown that on CPUs it leads to heavy branch mispredictions, causing considerable performance penalties [96, 30].

- **Bitwise-AND**, presented in Listing 5.2, is an alternative implementation that uses bitwise-AND operation (&) instead of logical-AND (&&). This approach reduces k conditionals to a single branch.

```
for (i=0; i < number_of_tuples; i++)
  if(t_i[0] < v_1 & t_i[1] < v_2 & ... t_i[k] < v_k )
  { out[j++] = i; }
```

Listing 5.2: Algorithm - Predicate Scan with Bitwise-AND (&)

Here all predicates for a given tuple, t_i, are evaluated and then, depending upon the result of the evaluation, a branch is executed. This method reduces branch misprediction penalties at the cost of higher computational work.

- Finally, a **no-branch** implementation is shown in Listing 5.3. This approach completely eliminates penalties caused by branch mispredictions by increasing computation cost even further.

```
for (i=0; i < number_of_tuples; i++)
  out[j] = i;
  j += (t_i[0] < v_1 & t_i[1] < v_2 & ... t_i[k] < v_k )
```

Listing 5.3: Algorithm - No-Branch

The techniques presented in Listings 5.2 and 5.3 either reduce or completely eliminate the branches. Yet, both approaches process all predicates and miss the opportunity to skip irrelevant evaluation as in the *branching scan*.

5.6.1 Software Implementations

An efficient implementation of algorithms 5.1-5.3 must be designed for a specific data layout to effectively leverage the extensive cache hierarchy. Many efficient algorithms have been proposed for both row [96] and column [91, 121, 51] storage layouts. Moreover, the columnar format allows effective compression and increases the throughput by more than an order of magnitude over traditional CPU row-store database systems [51]. However, all these approaches require an overhead of converting data in a row-major format to a columnar layout either in the background or keeping two separate representations of the same record.

In addition to thread-parallelism, modern architectures support data-level parallelism and provide new opportunities for *vectorized* implementations. All modern CPUs are equipped with SIMD registers that can be used to accelerate many database operations [120, 118], including selection. All three techniques described in Section 5.6 can be vectorized. However, the algorithm in Listing 5.1 requires additional bookkeeping to track which row should be dropped from future evaluation. This can be achieved by using non-contiguous loads/store (scatter/gather) operations[92] available only in the latest processors supporting AVX2 vectorization which make the design less portable. However, *branching scan* still can be implemented using older SIMD intrinsics but it will

be sensitive to small predicate selectivity. The other two techniques are independent of predicate selectivities. They do not take advantage of early termination and hence do more predicate evaluations per query than the first algorithm.

For the CPU we implemented a *scalar* as well as a *SIMD vectorized* version of the no-branch algorithm (Listing 5.3). (We note that for the scalar implementation we first performed a preliminary experiment also considering the algorithms in Listings 5.1 and 5.2, but determined that no-branch outperforms them, which is in line with earlier findings [96].) In both the scalar and vectorized implementations, we leverage multithreading by partitioning the input relation across different threads, pinned to physical CPU cores to avoid cache trashing. For each approach, we use the tuple storage format that allows it to extract the greatest benefits, namely row-major format for scalar implementation and columnar data layout for the vectorized version. Predicate constants are statically compiled, predicate loops are manually unrolled to avoid additional cache misses and allow additional compiler optimizations. The SIMD implementation does not use the latest AVX2 primitives (scatter/gather) to allow experimentation on older CPUs but leverages the permutation table technique to materialize the result recordIDs [92].

Recently, GPUs have also been considered a viable alternative to modern CPUs for accelerating common relational operators [71], including selections [65]. GPUs implement *no-branch* algorithm because conditional execution (Listings 5.1 and 5.2) results in thread divergence and reduces the overall system performance. Selection is executed in two steps: (1) evaluating the predicate conditions and (2) gathering the indices of the qualifying rows [71]. During processing, all threads within a warp execute in lock-step irrespective of the predicate selectivity. However, being oblivious to predicate selectivity can lead to wasted memory bandwidth since it does not provide any opportunities to stop fetching new predicates based on the last evaluation (Listing 5.1). For low selectivity, GPUs may overcome this limitation by relying on indexes [71], thus reducing the overall tuple evaluation latency at the expense of processing throughput. Overall, maintaining an

index can be costly when frequent updates are expected thus in this work, we focus on throughput optimization thus avoiding the use of indexes.

The GPU implementation relies on the findings of [102] and [71]. We assign each GPU thread to a distinct tuple for processing and evaluate the conditions of a query using a for-loop. Each evaluation is aggregated to a local register using bitwise-AND operands [102]. The final evaluation results are written back to global memory into a flag vector. In order to identify the qualifying tuple-ids, we utilize a stream compaction kernel available from NVIDIA's CUB [85] library. This kernel applies the given selection criterion, as indicated by the flag vector, to construct the corresponding output sequence from the selected items in the tuple-id input sequence. Gathering the qualifying tuples using the aforementioned method was also proposed in [71].

5.6.2 Datasets and Queries

In order to study in detail how the number of predicates and the total selectivity affects runtime on different platforms we used synthetically generated data. Each tuple consists of 8 fixed size 64-bit columns. We have considered both row-major and column-major storage formats. In the former case, the tuple's data is aligned contiguously occupying exactly one cache line. In the latter scenario, values of a particular column for all tuples are stored adjacent to each other. Values of different columns are drawn from a uniform distribution and are not correlated between individual tuples. This allows us to easily calculate the total query selectivity from the probability of each predicate.

In our experiments query selectivity was varied from 0% to 100% (0%: no row qualifies, 100% all rows qualify) with 10% increments, while the number of predicates in queries was independently changed from 1 to 8. We also tried different dataset sizes (8M, 16M, 32M, 128M) and found that throughput is independent of the dataset size. For brevity, we show only experiments on a 128M dataset.

To the best of our knowledge, there is no separate benchmark concentrating on evaluating the selection operator. Standard analytical database benchmarks (like TPC-H) evaluate complex SQL queries, consisting of multiple operators such as projections, joins, aggregations, etc. Among the TPC-H queries, only Q_6 involves single table selections; Section 5.6.3.3 presents our experiments for this query using data created by the TPC-H benchmark.

5.6.3 Experimental Evaluation

5.6.3.1 Runtime

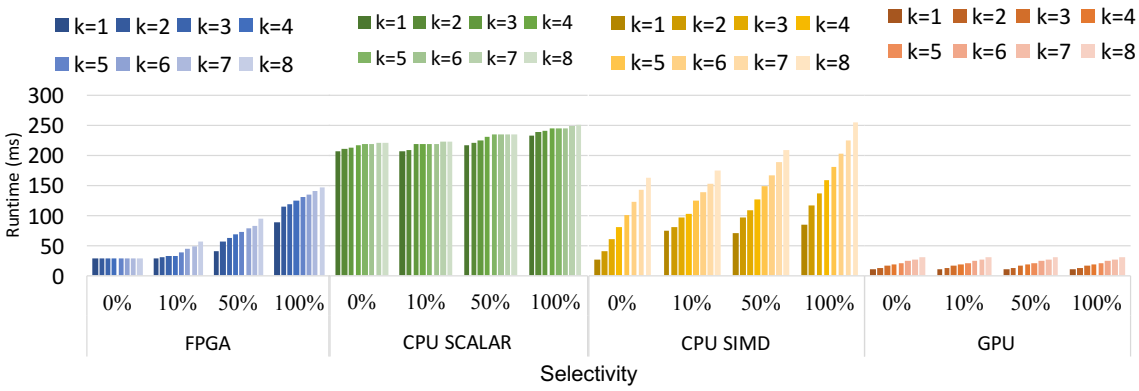


Figure 5.8: Query evaluation runtime measured on FPGA, CPU, and GPU with varying selectivity and number of predicates.

Figure 5.8 compares the absolute query runtime on the CPU, GPU, and FPGA implementations. The GPU delivers the best raw performance across different predicate values and selectivities, followed by the FPGA-based multithreaded approach, which performs better on low selectivity values and smaller values of predicates. The CPU SIMD implementation comes next, and finally the CPU Scalar implementation achieves the highest runtime among all other architectures.

Adhering to the branching-scan characteristics, the performance of the FPGA implementation is sensitive to predicate probability. The predicate probability of a conjunctive query increases with the selectivity (S) as well as with the number of predicates (k). As a consequence, the query evaluation can terminate early for the lower value of selectivities but builds up for higher values.

Additionally, for high values of S, the work required to write tuple into the output increases too, resulting in a quick drop in throughput.

Furthermore, in our experiments we define S using p and k assuming that all predicates have the same probability. On the other hand, real-world queries might have a different combination of predicate probabilities for the same total selectivity. If a query optimizer performs a good job of arranging predicates in the order of their likelihood of being false (true) for conjunctive (disjunctive) queries, the FPGA-based approach will work independently of the number of predicates in a query and is only limited by the number of memory accesses required for query evaluation. This can be seen in Figure 5.8, where the FPGA runtime does not change with the number of predicates for 0% selectivity.

Unlike the FPGA implementation which is based on early termination, all other platforms implement a variant of the *No-Branch* algorithm. The CPU Scalar graphs clearly show that its execution time is independent both from the number of predicates and from the query selectivity. This independence is an expected behavior because in a row-major storage format selection is a memory-bounded computation. Each access to a particular tuple will bring from memory the values for *all* its columns, whether they will be evaluated later or not.

On the other hand, the runtime of the CPU SIMD implementation grows linearly as we increase the number of predicates in a query from 1 to 8. Again, this behavior is explained by the fact that in a columnar storage format we are fetching only the values that will be later used for evaluating the predicate. For the queries with 8 predicates, the runtimes of Scalar and SIMD converge because they perform the same amount of memory accesses. However, we can also see another trend for the vectorized implementation: its runtime grows as we move from 0% selectivity to 100%. This is explained by the increasing amount of qualifying recordIDs which are written to the output buffer. The SIMD implementation is susceptible to growing number of output tuples because it requires additional permutations in order to retrieve IDs of the rows that were qualified from a SIMD lane.

We should also note that for the CPU implementations, the runtime is a function of the main memory bandwidth utilization, not the penalty of fetching data into CPU cache. In both experiments, the data access patterns (contiguous load for Scalar or load with constant strides for SIMD) are easily recognized by the CPU prefetcher, which was verified by preliminary experiments where we had disabled the prefetcher.

Similarly, the GPU implementation evaluates all predicates despite their different selectivities, resulting in more evaluation work for the respective query. This translates to a higher number of memory fetches that quickly dominate the total execution time, as their cost is several magnitudes higher than that of evaluating the predicate conditions. Therefore, an increase in the number of predicates corresponds to increasing runtime as indicated by our experimental results.

5.6.3.2 Throughput Efficiency

To better capture the memory-bounded nature of the selection and provide a direct comparison between widely different architectures we normalize the FPGA, CPU, and GPU throughput to the memory bandwidth available on each architecture. As discussed earlier in Section 5.3 the Convey HC-2ex has 4 FPGAs with the cumulative bandwidth of 76.8 GB/s, the CPU system has a memory bandwidth of 51.2 GB/s, and the GPU system has a memory bandwidth of 480 GB/s. The normalized results are shown in Figure 5.9.

The CPU SIMD implementation is remarkably efficient for $k = 1$ and $S = 0\%$. Since only *one* predicate is evaluated, the columnar data layout makes the cache access extremely effective in this case. Moreover, with 0% selectivity, there is no result materialization overhead. However, the CPU SIMD throughput drops quickly as S and, especially, k are increased.

In contrast, since the hardware multithreading-based design is only susceptible to the predicate probability, it takes better advantage of early termination on lower selectivity values and processes more tuples/sec per bandwidth. It can be seen from Figure 5.9 that for $k > 1$ and $S = 0\%$, FPGA is 1.2x - 5x more bandwidth efficient. However, as selectivity increases, we start seeing the

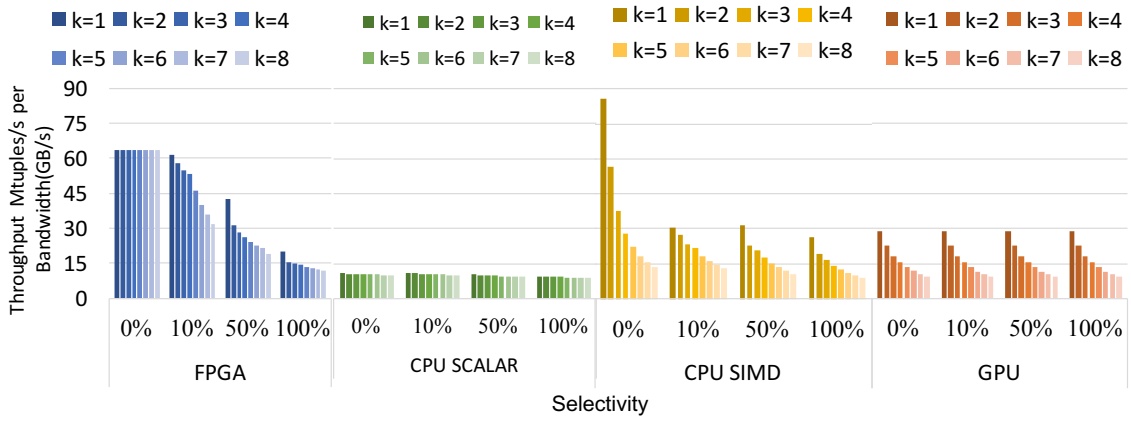


Figure 5.9: Throughput achieved by FPGA, CPU and GPU implementation normalized to their respective bandwidth. Note that the legend description is same as that of Figure 5.8.

effect of the writeback pressure. For instance, for 50% selectivity, an effective speedup of only 1.2x - 1.7x is achieved over the CPU SIMD implementation. Overall the FPGA design remains 1.6x - 4.7x more efficient in comparison to the CPU Scalar implementation within the wide range of selectivities ($0\% \leq S \leq 50\%$).

A similar trend is also observed while comparing to GPU. For lower selectivity values, say, $S = 0\%$ the FPGA-based approach is 1.8x - 5x more efficient than GPU. However, as we increase the selectivity, the GPU throughput remains unaffected while the FPGA sees a performance drop to 1.5x - 1.8x for $S = 50\%$. Finally, for $S = 100\%$, the hardware multithreaded throughput is similar to the GPU, CPU SIMD and has a 2x edge over the CPU Scalar. At this selectivity, the probability of each predicate is also 100%. For a conjunctive query, this leads to more time spent on predicate evaluations. Additionally, with the maximum value of selectivity, writeback work is at its peak. Therefore, at this stage, the FPGA design sees diminishing returns from the advantage of early termination as writeback becomes the bottleneck.

5.6.3.3 TPC-H Query Evaluation

To evaluate the performance of our implementations on a standard workload we considered the well-known TPC-H benchmark [9]. We have profiled all 22 TPC-H queries to understand the

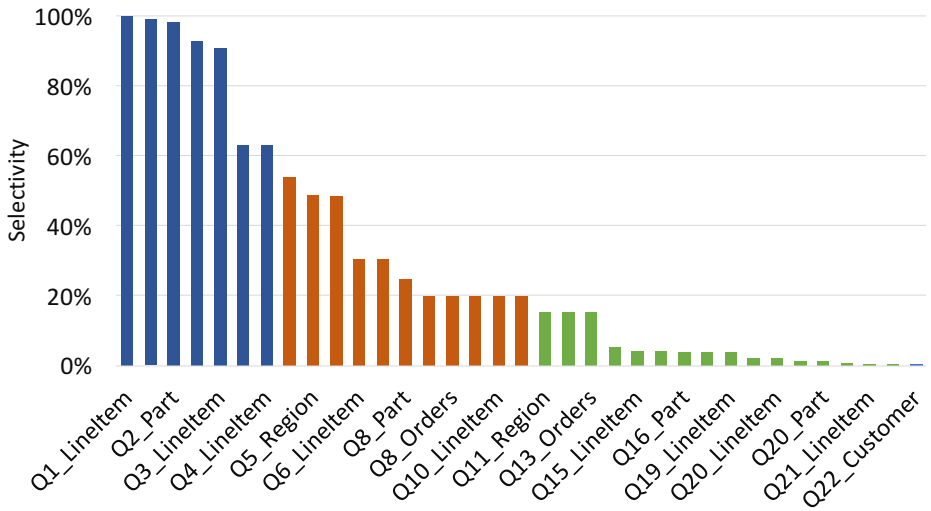
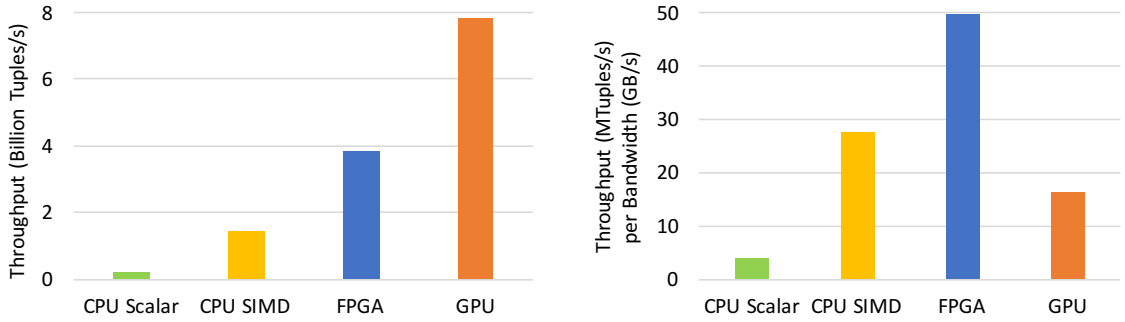


Figure 5.10: Selectivity of TPC-H queries. Each color marks the range of selectivity: (blue) above 60%, (orange) 50%-20%, (green) below 10%.

```
SELECT count(*)
FROM lineitem
WHERE l_shipdate >= date '1995-01-01'
  and l_shipdate < date '1996-01-01'
  and l_discount between 0.04 and 0.06
  and l_quantity < 24;
```

Figure 5.11: TPC-H Query6.

various characteristics (selectivity, number of predicates, predicate types) of the selection operator in this benchmark. Most queries in the TPC-H workload involve complex joins and group-by aggregations, so not all predicates in the WHERE clause might be used as filtering conditions in selection operators. Instead, we have considered optimized plans where selections are pushed down and executed before the joins and right after table scan operators. Figure 5.10 presents the selectivity(%) of different TPC-H queries. In this experiment, the average number of predicates in the selection was 2, with an exception of queries Q_6 and Q_{19} which have 5 and 8/12 (Part/Lineitem tables) predicates respectively. All of the selections in the benchmark were conjunctions, again excluding Q_{19} which has a mix of conjunctions and disjunctions.



(a) Raw performance comparison of CPU, FPGA, and GPU implementations

(b) Bandwidth normalized performance comparison of CPU, FPGA, and GPU implementation

Figure 5.12: TPC-H query Q6 performance evaluation.

In order to capture the real effect of the FPGA-based design in the selection operation, we would like to isolate the effect of all relational operators in the query. This makes Q_6, shown in Figure 5.11, an ideal candidate for our evaluation. We run the query Q_6 on the TPC-H Lineitem table with the scale factor of 10. The measured selectivity of this query is 1.91%. Figure 5.12a presents the raw performance of query Q_6 executed by the various architectures. We observe the same throughput trend as discussed in Section 5.6.3.2. Due to the high memory bandwidth GPU achieves the highest raw performance followed by the hardware multithreaded design, CPU SIMD, and CPU Scalar implementation. However, when we compare the throughput efficiency in Figure 5.12b, the FPGA implementation is 13x, 3x and 1.8x more bandwidth efficient than the CPU Scalar, GPU, and CPU SIMD, respectively.

	CPU Scalar	CPU SIMD	GPU	FPGA
Memory Fetches(%)	100	18.75	18.75	11.4
Evaluations (per Row)	3	3	3	1.83
Effective Bandwidth speedup	0.47	3.44	2.01	6.13
Peak Bandwidth Utilization (%)	47.6	61.2	36.6	70.3

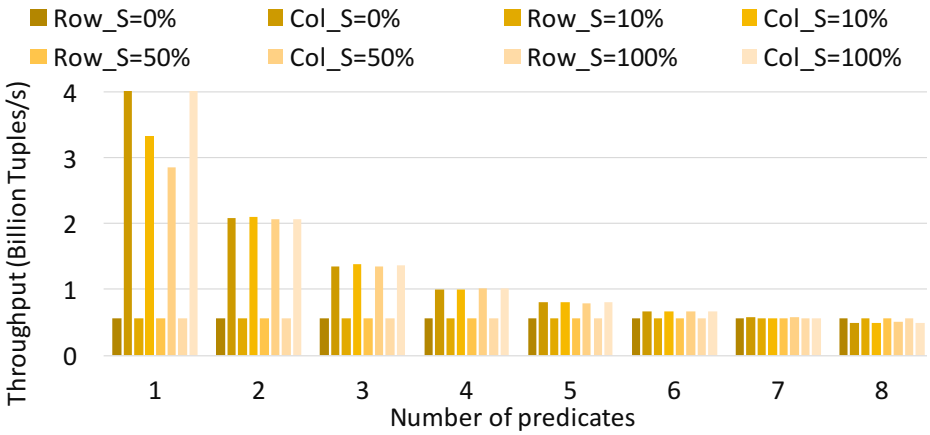
Table 5.1: Performance Evaluation of TPC-H query Q6 on CPU, GPU and FPGA implementations.

Furthermore, to confirm the advantage of the FPGA-based design, we also measured the total number of predicate evaluations and memory fetches. The CPU and GPU implementations access the common columns ($l_{shipdate}$, $l_{discount}$) only once. However, FPGA treats them as two independent attributes and fetches the same column again only if required for further evaluation. The CPU Scalar implementation accesses all 16 columns per row of the table, therefore it is considered to have a 100% memory access. The CPU SIMD and the GPU implementations need only 3 out of 16 columns to evaluate the query, contributing to 18.75% of memory accesses. We used counters to keep track of the number of memory fetches and the number of evaluated predicates for the FPGA implementation. Memory fetches with FPGA amount to 11.4% of total memory accesses. As suggested in [107], we compute the effective bandwidth speed-up by taking the ratio of the total size of the Lineitem relation processed per unit time and the peak bandwidth. Finally, we also report the actual peak bandwidth utilization in Table 5.1.

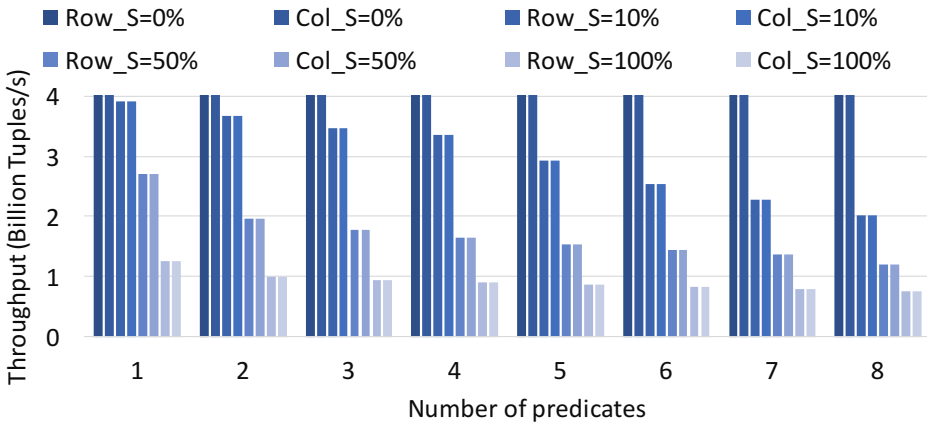
5.6.3.4 Data Layout Independence

We proceed with an experiment that tests the behavior of each approach for the row and column storage formats. The FPGA design, utilizing the Convey HC-2ex memory subsystem, achieves performance that does not rely on any form of data alignment in memory. Its performance only depends on accessing individual columns of a tuple for query evaluation.

On the other hand, both CPUs and GPUs are optimized for cache line accesses. As a result, we expect their performance to depend heavily on the different storage formats (row and column major). This is well known for GPUs as they depend on grouping the execution of threads into warps. This grouping is not only relevant to computation, but also to global memory accesses, making column access more advantageous. Related literature has unanimously promoted the use of the column-major [20, 58, 50, 102] data format for GPUs given its superior performance against the row data format. Hence we do not consider GPUs in the next experiment.



(a) CPU throughput measured for row vs column data layouts

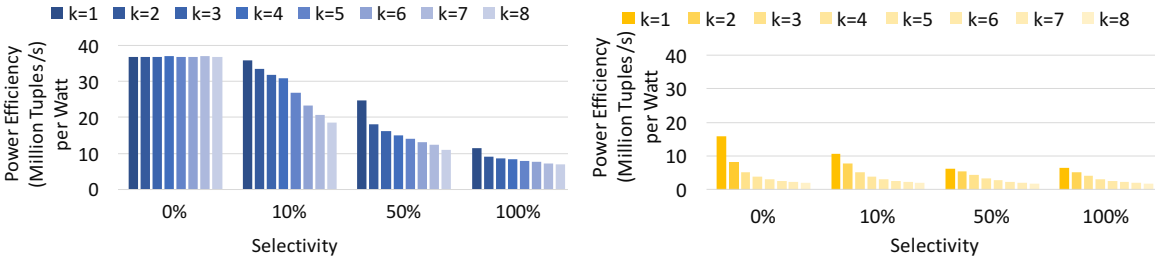


(b) FPGA throughput for the row vs column layouts

Figure 5.13: Performance comparison of the CPU and the FPGA implementations with row-major and columnar data layouts.

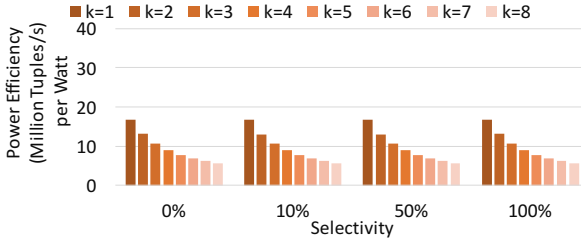
Figure 5.13a compares the performance achieved by the CPU implementations, namely, Scalar on row and SIMD on column store for varying selectivities and number of predicates. The columnar data layout leads to efficient cache access when only a few predicates are required for evaluation. This directly translates into high throughput which is 8x over the performance achieved by a row-store data layout. However, as the number of predicates increases, the amount of data accessed by both the row-store and the column store implementations converge and so does their performance. Figure 5.13b shows the FPGA performance on row and column store data layouts. For a given number of predicates and selectivity, the number of memory accesses does not depend upon the type of data layout and therefore the FPGA performance remains unaffected.

5.6.3.5 Power Efficiency



(a) Power efficiency of FPGA with varying S and k

(b) Power efficiency of CPU with varying S and k



(c) Power Efficiency of GPU with varying S and k

Figure 5.14: Comparison of Power Efficiency on FPGA, CPU and GPU systems.

To further justify the FPGA-based hardware multithreaded design, Figure 5.14 presents the power efficiency results measured in MTuples/s per Watt. We compared the power efficiency of CPU SIMD, GPU, and FPGA. The measured on-chip power consumption on each FPGA of the HC-2ex machine is 21 Watt and the total power supply is 25 W, that gives us the total FPGA power consumptions as 109W (25 + 21*4). On CPU, we use the manufacturer TDP (thermal design power) rating of 130 W per CPU as prescribed in [90]. On GPU, the average power consumption was measured to be 155W for the design with a power supply of 600W [3]. We compare the power efficiency of 4 FPGAs, 2 CPUs and 3584 cores on GPU in Figure 5.14. The power efficiency is coherent with our throughput evaluation. Since the FPGA throughput drops with the selectivity and number of predicates, it directly affects the power efficiency too. It is 2x - 19x better than CPU-SIMD on 0% selectivity. However, for $S = 100\%$ the power efficiency drops to 1.8x-4x. Similarly, in comparison to GPU we get 2x - 6.6x power savings but for $S = 100\%$, the efficiency is on par with GPU.

We use the same power statistics to evaluate the power efficiency of different architectures on query Q_6 of TPC-H benchmark. Overall, the FPGA design is 3.4x and 6.5x more power efficient than GPU and CPU-SIMD implementations on this query.

5.7 Conclusions

In this chapter, we provided an empirical study of algorithms optimized for in-memory computation of hash join, group-by aggregation, and selection. We have performed an extensive experimental evaluation of the performance of these operators and compared them against the hardware multithreading implementations prototyped of FPGAs. We showed that in many cases performance achieved by the FPGA-based implementations is higher than the throughput of the algorithms implemented in software. FPGA hardware multithreading was able to achieve higher throughput by utilizing available memory bandwidth in a more efficient way. In light of the end of Denard scaling era, we envision that increasing bandwidth utilization will become the only form of scaling in-memory workloads performance.

Chapter 6

Conclusions and Future Work

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 fast analytics to the users is a major challenge for today's data-intensive systems. In this dissertation, we have tackled several problems related to query processing in modern database systems.

In Chapter 3 we have proposed a novel statistics-collection framework which uses properties of the LSM-based storage to gather statistics. This approach piggybacks on inherent events of the LSM framework and allows a database system to collect statistics in a lightweight fashion without introducing additional runtime overhead. Within this framework, we have implemented 3 popular types of statistical synopses and presented algorithms to efficiently compute these synopses using a sorted stream abstraction provided by the index field (primary or secondary) of the LSM component. We have carried out an extensive experimental evaluation that verified that our statistics-collection approach does not slow down the ingestion rate of the database system. We have also presented experiments showing that calculated statistics provide good cardinality estimates for various values of input parameters such as input data distributions, query workloads, merge policy types and amount of space allocated to synopses.

In Chapter 4 we have extended this LSM framework to support statistics gathered on unordered attributes. Instead of relying on a sorted order provided by the indexed key we have chosen sketch algorithms (specifically, the Greenwald-Khanna approximate quantile sketch) to capture the statistical distribution of the unordered stream of values of a particular field. We have performed careful experiments to identify the limits of applicability of our sketch-based method and have found that it does not hamper the system's ingestion performance across a wide variety of settings. We have also shown that the cardinality estimation accuracy, provided by approximate statistics leveraging sketch algorithms, matched the accuracy of the exact indexed-based methods. Finally, we identified the default value of sketch accuracy that provides a practical tradeoff between ingestion rate and cardinality estimation error in AsterixDB.

As future work we propose to extend this 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 research direction is 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 allowing to correct estimation of cardinalities for correlated data. Finally, our sketch-based approach to gathering statistics on unordered fields could be further extended by implementing additional sketch algorithms [47, 45] to create wavelet-based synopses on unordered attributes and estimate cardinalities of joins.

In the second part of this thesis, we have considered the problem of analytical query processing for in-memory workloads. In Chapter 5 we have presented an in-depth empirical evaluation of typical database algorithms optimized for modern multicore CPU architectures. We have considered key relational operators, namely joins, aggregations and selections, and have implemented specialized versions of these algorithms which (i) are optimized for execution in the multicore environment, (ii) alleviate the number of cache misses, (iii) reduce control flow branch mispredictions and (iv)

apply vectorized execution techniques. Our experimental evaluation compared CPU implementations against the hardware multithreading algorithms implemented on FPGAs and demonstrated that FPGA-based approaches outperform software implementations in a large number of cases by utilizing memory bandwidth in a more efficient manner. Given the recent hardware threads, like the end of Denard scaling and memory wall problem, we predict that maximizing memory bandwidth utilization would be one of the key problems for future in-memory database systems.

In our experimental evaluation, we have studied the performance of each relational operator individually. However, in a real system the operators are executed in a pipelined manner feeding the output of one operator to the input of another. As a future work, we propose building a unified FPGA-based query execution engine that will be able to process complex analytical queries holistically by leveraging the hardware multithreading approach and comparing its performance against the software implementations. Finally, in addition to end-to-end query execution, it would be interesting to study how the FPGA-based hardware multithreading approach might be used in hybrid platforms, such as CPUs co-located with FPGAs.

Bibliography

[1] http://www.teradata.com/.

[2] http://www.ibm.com/software/data/netezza/.

[3] Nvidia. https://www.nvidia.com/en-us/geforce/products/10series/titan-x-pascal/.

[4] Peloton. http://pelotondb.io/.

[5] Data Streams: Algorithms and Applications. *Foundations and Trends® in Theoretical Computer Science*, 1(2):117–236, 2005.

[6] Amazon Redshift. Database Developer Guide. Analyzing Tables, July 2017.

[7] Apache AsterixDB, July 2017.

[8] Database Statistics in Greenplum Database, July 2017.

[9] (TPC). TPC-H Benchmark. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.17.3.pdf.

[10] Ashraf Aboulnaga and Surajit Chaudhuri. Self-tuning histograms: Building histograms without looking at data. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, pages 181–192, New York, NY, USA, 1999. ACM.

[11] Swarup Acharya, Phillip B. Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. The aqua approximate query answering system. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, pages 574–576, New York, NY, USA, 1999. ACM.

[12] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. 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*, page 29. ACM Press, 4 2013.

[13] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. Massively Parallel Sort-merge Joins in Main Memory Multi-core Database Systems. *Proceedings of the VLDB Endowment*, 5(10):1064–1075, 2012.

[14] Noga Alon, Phillip B. Gibbons, Yossi Matias, and Mario Szegedy. Tracking join and self-join sizes in limited storage. In *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems - PODS '99*, pages 10–20, New York, New York, USA, 1999. ACM Press.

[15] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing - STOC '96*, pages 20–29. ACM Press, 7 1996.

[16] 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. Asterixdb: A scalable, open source bdms. *PVLDB*, 7(14):1905–1916, October 2014.

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

[18] Paul M. Aoki. Algorithms for index-assisted selectivity estimation. In *Proceedings of the 15th International Conference on Data Engineering, Sydney, Austrialia, March 23-26, 1999*, page 258, 1999.

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

[20] Peter Bakkum and Srimat Chakradhar. Efficient data management for gpu databases. *High Performance Computing on Graphics Processing Units*, 2012.

[21] C. Balkesen, J. Teubner, G. Alonso, and M.T. Özsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *Proceedings of the 2013 IEEE International Conference on Data Engineering*, pages 362–373.

[22] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. *Proceedings of the VLDB Endowment*, 7(1):85–96, 2013.

[23] 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. The New Jersey Data Reduction Report. *IEEE Computer Society Technical Committee on Data Engineering*, 20:3–45, 1997.

[24] Ronald Barber, Guy Lohman, Vijayshankar Raman, Richard Sidle, Sam Lightstone, and Berni Schiefer. In-memory BLU acceleration in IBM's DB2 and dashDB: Optimized for modern workloads and hardware architectures. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 1246–1252. IEEE, 2015.

[25] 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. Impala: A modern, open-source sql engine for hadoop. In *Proceedings of the 7th Biennial Conference on Innovative Data Systems Research*, 2015.

[26] Spyros Blanas, Yinan Li, and Jignesh M. Patel. Design and Evaluation of Main Memory Hash Join Algorithms for Multi-core CPUs. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, pages 37–48, 2011.

[27] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[28] Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, and Robert Endre Tarjan. Time bounds for selection. *J. Comput. Syst. Sci.*, 7(4):448–461, 1973.

[29] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In *Proceedings of the 25th International Conference on Very Large Data Bases*, pages 54–65, 1999.

[30] David Broneske, Andreas Meister, and Gunter Saake. Hardware-sensitive scan operator variants for compiled selection pipelines. In *BTW*, pages 403–412, 2017.

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

[32] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. Stholes: A multidimensional workload-aware histogram. *ACM SIGMOD Record*, 30(2):211–222, 6 2001.

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

[34] Kaushik Chakrabarti, Minos Garofalakis, Rajeev Rastogi, and Kyuseok Shim. Approximate query processing using wavelets. *The VLDB Journal*, 10(2-3):199–223, September 2001.

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

[36] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *International Colloquium on Automata, Languages, and Programming*, pages 693–703. Springer, 2002.

[37] Surajit Chaudhuri. 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*, pages 34–43. ACM Press, 5 1998.

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

[39] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. Improving hash join performance through prefetching. *ACM Transactions on Database Systems*, 32(3):17–es, 8 2007.

[40] John Cieslewicz and Kenneth A. Ross. Adaptive Aggregation on Chip Multiprocessors. VLDB'07, pages 339–350, 2007.

[41] John Cieslewicz and Kenneth A. Ross. Data Partitioning on Chip Multiprocessors. DaMoN'08, pages 25–34, 2008.

[42] Graham Cormode. What is Data Sketching, and Why Should I Care? *Communications of the ACM (CACM)*, 60(9):48–55, 2017.

[43] Graham Cormode and Minos Garofalakis. Sketching streams through the net: Distributed approximate query tracking. *VLDB '05 Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, pages 13–24, 2005.

[44] Graham Cormode, Minos Garofalakis, and Peter J. Haas. Synopses for massive data. Now Publishers Inc., 1 2012.

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

[46] Graham Cormode and Marios Hadjieleftheriou. Finding frequent items in data streams. *Proceedings of the VLDB Endowment*, 1(2):1530–1541, 2008.

[47] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[48] David J. DeWitt and Jim Gray. Parallel database systems: The future of high performance database systems. *Commun. ACM*, 35:85–98, 1992.

[49] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1243–1254. ACM, 2013.

[50] Gregory Frederick Diamos, Haicheng Wu, Ashwin Lele, and Jin Wang. Efficient relational algebra algorithms and data structures for GPU. Technical report, Georgia Institute of Technology, 2012.

[51] Amr El-Helw, Kenneth A. Ross, Bishwaranjan Bhattacharjee, Christian A. Lang, and George A. Mihaila. Column-oriented query processing for row stores. In *Proceedings of the ACM 14th International Workshop on Data Warehousing and OLAP*, DOLAP '11, pages 67–74, New York, NY, USA, 2011. ACM.

[52] Franz Faerber, Alfons Kemper, Per-Åke Larson, Justin Levandoski, Thomas Neumann, Andrew Pavlo, et al. Main memory database systems. *Foundations and Trends® in Databases*, 8(1-2):1–130, 2017.

[53] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. The SAP HANA Database – An Architecture Overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[54] John Feehrer, Sumti Jairath, Paul Loewenstein, Ram Sivaramakrishnan, David Smentek, Sebastian Turullols, and Ali Vahidsafa. The Oracle Sparc T5 16-Core Processor Scales to Eight Sockets. *IEEE Micro*, 33(2):48–57, March 2013.

[55] Edward B Fernandez, Walid A Najjar, Stefano Lonardi, and Jason Villarreal. Multithreaded fpga acceleration of dna sequence mapping. In *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on*, pages 1–6. IEEE, 2012.

[56] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *AofA: Analysis of Algorithms*, pages 137–156. Discrete Mathematics and Theoretical Computer Science, 2007.

[57] Rainer Gemulla, Wolfgang Lehner, and Peter J. Haas. Maintaining bounded-size sample synopses of evolving datasets. *The VLDB Journal*, 17(2):173–201, March 2008.

[58] Pedram Ghodsnia et al. An in-GPU-memory column-oriented database for processing analytical workloads. In *The VLDB PhD Workshop. VLDB Endowment*, volume 1, 2012.

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

[60] A.C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M.J. Strauss. One-pass wavelet decompositions of data streams. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):541–554, 5 2003.

[61] A.C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M.J. Strauss. Domain-driven data synopses for dynamic quantiles. *IEEE Transactions on Knowledge and Data Engineering*, 17(7):927–938, 7 2005.

[62] Anna C. Gilbert, Sudipto Guha, Piotr Indyk, Yannis Kotidis, S. Muthukrishnan, and Martin J. Strauss. Fast, small-space algorithms for approximate histogram maintenance. In *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing*, STOC '02, pages 389–398, New York, NY, USA, 2002. ACM.

[63] Anna C Gilbert, Yannis Kotidis, S Muthukrishnan, and Martin Strauss. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases*, pages 79–88, 2001.

[64] Anna C. Gilbert, Yannis Kotidis, S. Muthukrishnan, and Martin J. Strauss. How to summarize the universe: dynamic maintenance of quantiles. pages 454–465, 8 2002.

[65] Naga K Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin, and Dinesh Manocha. Fast computation of database operations using graphics processors. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 215–226. ACM, 2004.

[66] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter Weinberger. Quickly Generating Billion-record Synthetic Databases. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 243–252, 1994.

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

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

[69] Peter J. Haas and Arun N. Swami. Sequential sampling procedures for query size estimation. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, SIGMOD '92, pages 341–350, New York, NY, USA, 1992. ACM.

[70] Robert J. Halstead, Ildar Absalyamov, Walid A. Najjar, and Vassilis J. Tsotras. FPGA-based Multithreading for In-Memory Hash Joins. CIDR'15, 2015.

[71] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K Govindaraju, Qiong Luo, and Pedro V Sander. Relational query coprocessing on graphics processors. *ACM Transactions on Database Systems (TODS)*, 34(4):21, 2009.

[72] Yannis E. Ioannidis and Stavros Christodoulakis. 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.*, pages 268–277, 1991.

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

[74] Alfons Kemper and Thomas Neumann. HyPer: A hybrid OLTP & OLAP main memory database system based on virtual memory snapshots. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 195–206. IEEE, 2011.

[75] Changkyu Kim, Tim Kaldewey, Victor W. Lee, Eric Sedlar, Anthony D. Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-core CPUs. *Proceedings of the VLDB Endowment*, 2(2):1378–1389, August 2009.

[76] Robert Philip Kooi. *The optimization of queries in relational databases*. PhD thesis, 1980.

[77] Tirthankar Lahiri, Marie-Anne Neimat, and Steve Folkman. Oracle timesten: An in-memory database for enterprise applications. *IEEE Data Eng. Bull.*, 36(2):6–13, 2013.

[78] Per-Ake Larson, Wolfgang Lehner, Jingren Zhou, and Peter Zabback. Cardinality estimation using sample views with quality assurance. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 175–186, New York, NY, USA, 2007. ACM.

[79] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *PVLDB*, 9:204–215, 2015.

[80] Stefan Manegold, Peter Boncz, and Martin Kersten. Optimizing Main-Memory Join on Modern Hardware. *IEEE Transactions on Knowledge and Data Engineering*, 14(4):709–730, July 2002.

[81] Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce G. Lindsay. Approximate medians and other quantiles in one pass and with limited memory. *ACM SIGMOD Record*, 27(2):426–435, 6 1998.

[82] Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce G Lindsay. Random sampling techniques for space efficient online computation of order statistics of large datasets. In *ACM SIGMOD Record*, volume 28, pages 251–262. ACM, 1999.

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

[84] Yossi Matias, Jeffrey Scott Vitter, and Min Wang. Dynamic maintenance of wavelet-based histograms. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pages 101–110, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[85] Duane Merrill and NVIDIA-Labs. CUDA UnBound (CUB) Library.

[86] Abhishek Mitra, Marcos R. Vieira, Petko Bakalov, Vassilis J. Tsotras, and Walid A. Najjar. Boosting XML filtering through a scalable FPGA-based architecture. CIDR'09, 2009.

[87] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. Preventing bad plans by bounding the impact of cardinality estimation errors. *Proceedings of the VLDB Endowment*, 2(1):982–993, 2009.

[88] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, June 1996.

[89] Gregory Piatetsky-Shapiro and Charles Connell. Accurate estimation of the number of tuples satisfying a condition. *ACM Sigmod Record*, 14(2):256–276, 1984.

[90] Meikel Poess and Raghunath Othayoth Nambiar. Energy cost, the key challenge of today's data centers: A power consumption analysis of tpc-c results. *Proc. VLDB Endow.*, 1(2):1229–1240, August 2008.

[91] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. Rethinking SIMD Vectorization for In-Memory Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data - SIGMOD '15*, pages 1493–1508, New York, New York, USA, 2015. ACM Press.

[92] Orestis Polychroniou and Kenneth A. Ross. Vectorized Bloom filters for advanced SIMD processors. In *Proceedings of the Tenth International Workshop on Data Management on New Hardware - DaMoN '14*, pages 1–6, New York, New York, USA, 2014. ACM Press.

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

[94] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. ISCA'14, June 2014.

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

[96] Kenneth A. Ross. Conjunctive selection conditions in main memory. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, pages 109–120, New York, NY, USA, 2002. ACM.

[97] Florin Rusu and Alin Dobra. Statistical analysis of sketch estimators. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 187–198. ACM, 2007.

[98] M. Sadoghi, R. Javed, N. Tarafdar, H. Singh, R. Palaniappan, and H.-A Jacobsen. Multi-query Stream Processing on FPGAs. In *Proceedings of the 2012 IEEE International Conference on Data Engineering*, pages 1229–1232, April 2012.

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

[100] Nikita Shamgunov. The MemSQL in-memory database system. In *IMDM@ VLDB*, 2014.

[101] Nisheeth Shrivastava, Chiranjeeb Buragohain, Divyakant Agrawal, and Subhash Suri. Medians and beyond: new aggregation techniques for sensor networks. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 239–249. ACM, 2004.

[102] Evangelia A. Sitaridi and Kenneth A. Ross. Optimizing select conditions on GPUs. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware - DaMoN '13*, page 1, New York, New York, USA, 2013. ACM Press.

[103] U. Srivastava, P. J. Haas, V. Markl, M. Kutsch, and T. M. Tran. Isomer: Consistent histogram construction using query feedback. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE '06, pages 39–, Washington, DC, USA, 2006. IEEE Computer Society.

[104] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. Leo - db2's learning optimizer. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 19–28, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

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

[106] Hong Su, Mohamed Zait, Vladimir Barrière, Joseph Torres, and Andre Menck. Approximate Aggregates in Oracle 12C. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management - CIKM '16*, pages 1603–1612, New York, New York, USA, 2016. ACM Press.

[107] Bharat Sukhwani, Hong Min, Mathew Thoennes, Parijat Dube, Balakrishna Iyer, Bernard Brezzo, Donna Dillenberger, and Sameh Asaad. Database Analytics Acceleration Using FPGAs. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, pages 411–420, 2012.

[108] Jens Teubner and Rene Mueller. How Soccer Players Would Do Stream Joins. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, pages 625–636, 2011.

[109] Mikkel Thorup and Yin Zhang. Tabulation based 4-universal hashing with applications to second moment estimation. In *SODA*, volume 4, pages 615–624, 2004.

[110] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: A warehousing solution over a map-reduce framework. *PVLDB*, 2(2):1626–1629, August 2009.

[111] Pınar Tözün, Brian Gold, and Anastasia Ailamaki. OLTP in wonderland: where do cache misses come from in major OLTP components? In *Proceedings of the Ninth International Workshop on Data Management on New Hardware*, page 8. ACM, 2013.

[112] David Vengerov, Andre Cavalheiro Menck, Mohamed Zait, and Sunil P. Chakkappen. Join size estimation subject to filter conditions. *Proceedings of the VLDB Endowment*, 8(12):1530–1541, aug 2015.

[113] Jeffrey Scott Vitter, Min Wang, and Bala Iyer. Data cube approximation and histograms via wavelets. In *Proceedings of the seventh international conference on Information and knowledge management - CIKM '98*, pages 96–104, New York, New York, USA, nov 1998. ACM Press.

[114] Hai Wang and Kenneth C Sevcik. 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*, pages 328–342. IBM Press, 2003.

[115] Li Wang, Minqi Zhou, Zhenjie Zhang, Ming-Chien Shan, and Aoying Zhou. NUMA-Aware Scalable and Efficient In-Memory Aggregation on Large Domains. 27(4):1071–1084, April 2015.

[116] Lu Wang, Ge Luo, Ke Yi, and Graham Cormode. Quantiles over data streams: An experimental study. In *Proceedings of the 2013 international conference on Management of data - SIGMOD '13*, page 737. ACM Press, 6 2013.

[117] Min Wang, Jeffrey Scott Vitter, Lipyeow Lim, and Sriram Padmanabhan. Wavelet-based cost estimation for spatial queries. In *Advances in Spatial and Temporal Databases*, pages 175–193, 2001.

[118] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. Simd-scan: Ultra fast in-memory table scan using on-chip vector processing units. *Proc. VLDB Endow.*, 2(1):385–394, August 2009.

[119] Yang Ye, Kenneth A. Ross, and Norases Vesdapunt. Scalable aggregation on multicore processors. DaMoN'11, pages 1–9, 2011.

[120] Jingren Zhou and Kenneth A. Ross. Implementing database operations using SIMD instructions. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 145–156, New York, NY, USA, 2002. ACM.

[121] Marcin Zukowski, Niels Nes, and Peter Boncz. DSM vs. NSM: CPU performance tradeoffs in block-oriented query processing. In *Proceedings of the 4th International Workshop on Data Management on New Hardware*, DaMoN '08, pages 47–54, New York, NY, USA, 2008. ACM.