Histograms as a Side Effect of Data Movement for Big Data

Zsolt István Louis Woods Gustavo Alonso
Systems Group, Dept. of Computer Science
ETH Zürich, Switzerland
{firstname.lastname}@inf.ethz.ch

ABSTRACT

Histograms are a crucial part of database query planning but their computation is resource-intensive. As a consequence, generating histograms on database tables is typically performed as a batch job, separately from query processing. In this paper, we show how to calculate statistics as a *side effect* of data movement within a DBMS using a hardware accelerator in the data path. This accelerator analyzes tables as they are transmitted from storage to the processing unit, and provides histograms on the data retrieved for queries at virtually no extra performance cost. To evaluate our approach, we implemented this accelerator on an FPGA. This prototype calculates histograms faster and with similar or better accuracy than commercial databases. Moreover, the FPGA can provide various types of histograms such as *Equi-depth*, *Compressed*, or *Max-diff* on the same input data in parallel, without additional overhead.

Categories and Subject Descriptors

H.2 [**Database Management**]: Systems

Keywords

FPGA, Statistics, Histogram, Query Optimization

1. INTRODUCTION

Statistics such as histograms play an important role in modern databases. They have been long used for query optimization, where they influence, *e.g.*, how the data is accessed or what join algorithm is used. Furthermore, now that we have entered the era of big data, histograms and statistics in general are becoming even more important: besides standard query optimization decisions, knowing the data distribution could also influence the choice between different types of CPU cores, the amount of data moved between different computing nodes, etc. One important property of histograms is that they are useful only when accurate enough and up-to-date, but updating them takes processing power away from query processing. As a result, histograms are mostly calculated in maintenance windows, and often with sampling, to reduce their impact on response time [37]. However, the sheer amount of data that needs to be processed means that the traditional techniques for statistical analysis and histogram creation will be increasingly difficult to apply. Take for instance the previously mentioned case of sampling: in order to balance execution time and accuracy, most modern databases do not calculate the histograms on the full data but on a small sample. While this approach provides good results if the sample is representative [5], when the time budget for statistical computation is small, the sampling rate could become so low that reasonable accuracy can not be guaranteed for the resulting histogram.

In this paper, we introduce the idea of *statistics calculated on the data path*. By moving the task of calculating histograms to a specialized hardware device, we can obtain them virtually for 'free' every time we retrieve a table from any form of storage, *i.e.*, without throttling throughput and by only adding negligible latency. Apart from the obvious benefit of off-loading the calculation of histograms to an accelerator, which frees computing cycles on the CPU, there is also a second benefit: If histograms can be refreshed every time a table is scanned, the global freshness of statistics will be higher than that of current systems.

As a proof of concept, we have implemented a statistical accelerator using a field-programmable gate array (FPGA) that, to our knowledge, is the first of its kind. With this prototype we show that not only the histograms are calculated faster than on a CPU but also that we can provide many different types of histograms (Equi-depth, Compressed, and Max-diff) at no additional performance cost. The main benefit of using FPGAs is that due to their inherent parallelism they can stream the data through, minimizing the added latency, and calculate the histogram in parallel at the same time. In future systems, the FPGA could be replaced with an application-specific integrated circuit (ASIC), which has the same advantages but consumes less power, or with a small specialized processor attached directly to storage [10].

Contribution. In this paper, we revisit the state-of-the-art in histograms for databases and propose an accelerator in the data path for building histograms as a side effect to data movement. We implement a prototype of such an accelerator using an FPGA. We show that the FPGA can handle high throughput and very large table sizes – both crucial for processing *big data*. By implementing canonically accepted histogram types we can also guarantee accuracy on the statistics derived by the accelerator.

SIGMOD/PODS'14, June 22 - 27 2014, Snowbird, UT, USA.

http://dx.doi.org/10.1145/2588555.2612174.

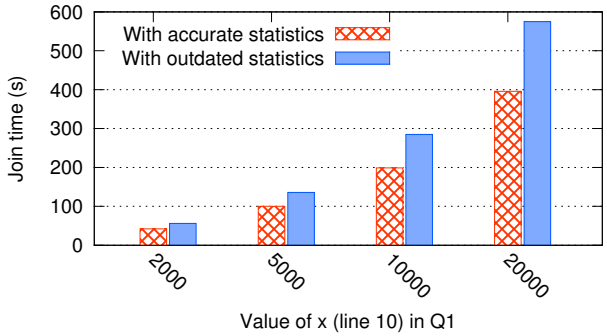

Figure 1: Effect of fresh statistics on query plans

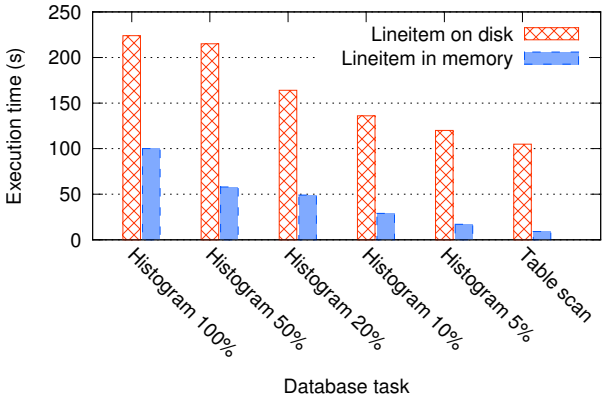

Figure 2: Even with sampling, analysis is more expensive than a full table scan in databases

Outline. The rest of this paper is structured as follows: We motivate our work with an example, and then relate to other work in Section 2. Section 3 discusses the most common histograms used in modern databases. In Section 4, we present the architecture of our system on a high level before covering the implementation details in Section 5. We evaluate our approach in Section 6, discuss ideas for future work in Section 7, and conclude in Section 8.

2. MOTIVATION AND RELATED WORK

In this paper, we propose a hardware accelerator for calculating accurate histograms on tables retrieved in full table scans, at no additional CPU cost. To understand the importance of accurate histograms, consider an example query on two TPC-H [8] tables, *lineitem* and *customer*, that highlights how the query planner decisions are influenced by statistical information in commercial databases. The query calculates a derived final price for items that have some base price, and then counts the number of such entries per client that have a balance larger than the final price:

```
1  with somelines as
2  (
3   select (l_tax*l_extendedprice) as val
4   from lineitem
5   where l_extendedprice=2001 --skewed value
6  )                                              (Q1)
7  select customer.c_custkey, count(*) as cnt
8   from customer, somelines
9   where somelines.val<customer.c_acctbal
10    and customer.c_custkey<x --parameter
11  group by customer.c_custkey;
```

We ran this query on a widely used commercial database, and observed that the number of lines matching the base price changes the optimal query plan. To demonstrate this effect we generated *lineitem* with 60 million rows (scale factor 10), then increased the number of records with price "2001" to 120,000. When we ran Q_1 after updating these lines, without refreshing statistics, the database was operating under the assumption that the value "2001" appears less than fifty times in the *l_extendedprice* column, as the initial statistics on *lineitem* suggested.

Since statistics gathering needs to be explicitly triggered in databases[1], even after running the query multiple times, the query planner was producing the same sub-optimal plan. Figure 1 shows that after updating the statistics on the column, the query planner correctly adjusted the plan, which resulted in better query performance. The main difference between the two query plans is the order in which the tables are joined with Sort Merge Join; and that when running with outdated statistics, the database underestimates the size of one of the join tables by almost four orders of magnitude. As Figure 1 also illustrates, by increasing the number of matching items in the *customer* relation, the effect can be further amplified – this highlights that even for simple queries fresh statistics are very important.

In the case of this particular example, the query planner error can be easily avoided by updating the histogram on the *l_extendedprice* column. In commercial databases however, even with sampling, statistics collection is an expensive process. Figure 2 illustrates how long it takes for the database to update the histogram with different levels of sampling on one column of the TPC-H *lineitem* table (scale factor 10) residing on disk and in memory. Additionally, we included in the figure the time it takes to perform a very simple query with a full table scan on the same data. This is to show how much more work the database performs to build a histogram even from only a 5% sample compared to answering a query with a full table scan. While not discussed here, in Section 6.2 we show that sampling may also negatively affect query planning. This is why providing histograms built from the complete data, delivered with no performance penalty as a by-product of full table scans, is a valuable addition to modern database systems.

Related Work. In this paper we propose a statistics accelerator that *passively* operates on the data that is moved to the processor for query processing. A similar idea, though using conventional CPUs, was first proposed by Zhu et al [37]. The key idea is what Zhu et al. call the *piggyback method*, that collects statistics by piggybacking additional data retrievals during the processing of a user query in order to update existing statistics. While this method improves the *freshness* of histograms, the CPU still has to process the data and derive the statistics. The authors admit that as a consequence their method may slow down query processing in favor of more up-to-date statistics.

Using an accelerator to compute statistics was previously suggested by Heimel et al. [13]. In this work, a GPU was

[1]For instance, in Oracle this can be done with the following command: DBMS_STATS.GATHER_TABLE_STATS. The columns to analyze, the number of buckets in the histogram, and the sampling rate can be specified as parameters.

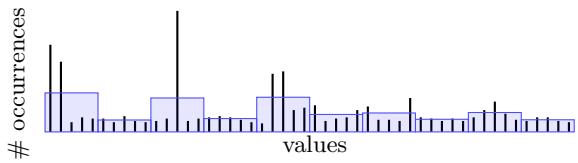

Figure 3: Equi-width

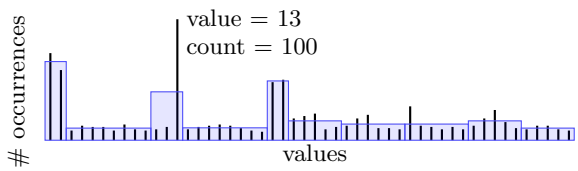

Figure 4: Equi-depth

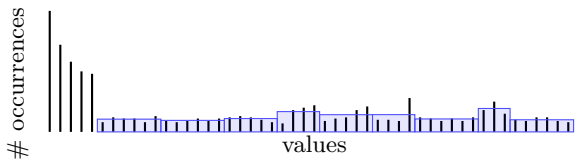

Figure 5: Compressed

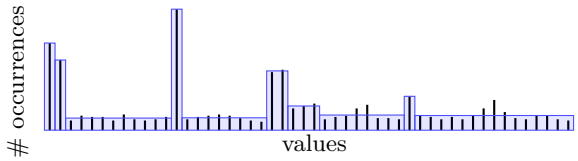

Figure 6: Max-diff

used to provide the query optimizer with statistics such as range selectivity estimations. However, the authors observe that copying whole tables to the GPU quickly becomes a bottleneck, and therefore they rely on sampling for larger tables. This means that even though the parallel resources on the GPU provide the estimates much faster than the CPU, the solution still suffers from all the drawbacks of sampling.

Utilizing specialized hardware in the data path to compute statistics as a side effect does not suffer from these shortcomings, and the existence of such accelerators is reasonable since many systems are already using similar accelerators today to improve query performance. For instance, IBM/Netezza's data warehouse appliance [15] uses FPGAs close to storage to which it pushes simple SQL operators. The work in [4] explores FPGAs as accelerators for join operations in columnar main-memory databases; in Cipherbase [2] they are used for encrypted SQL query processing, and in [9, 35] for partial SQL query execution. Furthermore, smart SSDs [10] and flash stores [18] have been proposed, which exploit the processing elements inside storage devices to off-load different database operators.

In the bigger picture, specialized hardware, and FPGAs in particular, have been successfully used in many areas of data processing, *e.g.*, for pattern matching and XML filtering [21, 34, 32], network intrusion detection [36], traffic control information processing [33], algorithmic trading [29], streaming query execution [24, 30, 23]. Furthermore, FPGAs have been recently used even for accelerating main-memory distributed hash tables [1, 6, 17].

3. HISTOGRAMS IN DATABASES

In this section, we revisit the most common types of histograms used in databases today [16]. It is important to understand the high-level differences between these histograms before we discuss the hardware implementation of each of them (Section 5). To illustrate these differences, we consider an arbitrary data distribution, which we use for Figures 3 through 6. These figures have on the x-axis the different values and on the y-axis their count as bars. The histogram buckets are blue rectangles. The area of every rectangle corresponds to the total count of a bucket. Within a bucket, uniform distribution is assumed, *i.e.*, the height of the rectangle corresponds to the estimated count of each value within the respective bucket.

Equi-width Histogram. The simplest histogram is the so-called *equi-width* histogram, displayed in Figure 3. It divides the value range into n buckets and then counts the number of occurrences in each bucket. In Figure 3, the value range is divided into $n = 10$ buckets. Each bucket counts the number of occurrences of five consecutive values in this range. This type of histogram is easy to construct in linear time using *bucket sort* [7] (sometimes referred to as *bin sort*). However, as can be seen in Figure 3, this histogram does not represent *skewed* data very well, which is why equi-width histograms are seldom used in databases.

Equi-depth Histogram. A more common histogram used in databases is the *equi-depth* histogram, which represents skewed data more accurately. PostgreSQL, Oracle and DB2 all use some version of this histogram. In an equi-depth histogram, all buckets have an equal or close to equal count but the value range that a bucket spans is no longer fixed. A common way to generate an equi-depth histogram is to first sort the data. The bucket boundaries are then given by the values found at every $\#occurrences/n$ position in the sorted data. Figure 4 displays an equi-depth histogram constructed from the same data used to generate the equi-width histogram in Figure 3.

Compressed Histogram. While the equi-depth histogram handles skewed data better than the equi-width histogram, heavy hitters are still a problem. For instance, in Figure 4, a single value (the annotated bar) appears more often than any other item, but its count is not high enough to occupy a full bucket – when estimating the values belonging to this bucket some of them will be overestimated, and the peak will be underestimated. The so-called *Compressed* histogram mitigates this effect by counting the most frequent values separately and then generating an equi-depth histogram on the remaining values. Figure 5 shows a Compressed histogram, where the five most frequent values are counted separately.

V-Optimal and Max-diff Histogram. Poosala et al. [27] proved that from all possible histograms, the so-called *v-optimal* histogram provides the best approximation on the real data distribution. In this type of histogram the bucket boundaries are chosen such that the variance within every bucket is minimized. However, computing a v-optimal histogram is prohibitively expensive [27], which is why these histograms are not used in practice. A heuristic that is easier to compute and approximates a v-optimal histogram is the *Max-diff* histogram, which is implemented in Microsoft's

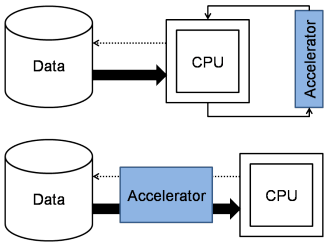

Figure 7: Explicit (top) vs. implicit (bottom) accelerator

SQL Server [20]. In a Max-diff histogram the bucket boundaries are positioned where the difference in occurrence count are maximal between adjacent pairs of values. This is a much easier optimization problem than minimizing the variance within buckets, as in the v-optimal histogram. In Figure 6, a Max-diff histogram is depicted, representing the same data as the other histograms in this section.

Automated Statistics. Since histogram calculation and statistic gathering is a costly operation, with many parameters, most modern database engines offer automated statistic gathering on the tables. Oracle [26], IBM DB2 [14] and Microsoft SQL Server [19] all will decide based on the table contents and workloads which tables need statistics, which columns need more detailed statistics such as histograms, and when to update the statistics for different tables. While these methods are effective, they operate under a very strict time budget, meaning that statistics and histograms cannot be refreshed as often as they should be. This is why using accelerators is highly beneficial: the automated tools could operate at higher efficiency, and refresh statistics more frequently than before.

4. SYSTEM OVERVIEW

An accelerator can be integrated into a system either as an explicit accelerator located on the side of the host system, or as an implicit accelerator, one that is on the data path (Figure 7). The fundamental difference between the two is that while the former needs to be accessed explicitly, and data needs to be copied to it, the latter is active every time data is streaming through it. In our case this means that histograms can be calculated *every time* data is retrieved for processing, as opposed to *on demand*, when the host decides to send data to the accelerator. Copying data from the CPU to the accelerator might not even be feasible without disrupting query processing if the tables are very large. By contrast, the only important requirement of an accelerator on the data path is that it must not introduce significant latency into the data stream. In this paper, we show that it is possible to offer sophisticated histograms and still guarantee only a negligible delay in the data stream.

FPGAs. We chose to implement our prototype system using an FPGA because FPGAs are well suited for dataflow type applications [22, 36], and can handle stream processing very well [29, 30, 23]. This makes them an ideal candidate for statistical accelerators, because we can dedicate hardware circuits to forward the incoming data, and use other parallel parts of the chip for calculating histograms. This way, the only cost of calculating histograms on the FPGA from the point of view of the host is the time it takes to stream the original data through the FPGA.

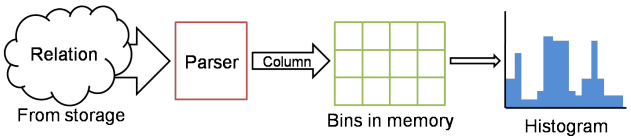

Figure 8: The conceptual steps performed in the accelerator

Figure 8 depicts the main steps that need to be carried out to produce histograms, and Figure 9 shows the implementation on the FPGA, consisting of the corresponding three modules: Parser, Binner and Histogram creation. In the following we will refer to the collection of these three as the *statistical circuit*. The role of the *Parser* is to parse the input data (database pages), and to keep only the column that will be processed by the rest of the circuit. The choice of the column is communicated by the host, with a metadata packet piggybacked on the original command to the data storage. To this end, the logic to filter the column of interest is a simple counting state machine. The second stage in the global pipeline is the *Binner* module, which calculates the in-memory sorted representation of the data on the streaming input. Finally, the *Histogram* module reads this representation from memory and uses it to build various histograms – Equi-depth, Max-diff and Compressed. These two modules are decoupled in their operation, since they only interact through regions in memory. This means that while for some data the histogram is calculated in the Histogram module, another input table can be already processed and binned at a different region in memory.

Histograms in linear time. Even though FPGAs excel at pipeline processing, the available on-chip memory (to keep state) is very limited, and the off-chip high-capacity memory is much slower to access. As a result, we were looking for ways to minimize the state required for performing the steps depicted in Figure 8. Parsing can be done in constant space and time with a simple finite-state machine (FSM). More challenging is the binning step, in which the data is radix sorted and aggregated in memory. For this we use an algorithm that runs in linear time, requires constant state and the amount of memory depends not on the size of the dataset but its cardinality. Inspired by the *bin sort* algorithm [7], the FPGA will associate a memory range with the range of values in the column. Assuming that the data in the column is of integer or fixed-point type, its corresponding count aggregate in memory can be updated directly on every appearance. Once the whole dataset has streamed by, the memory will hold a sorted view of the data. As later explained, the histogram creation step can also be carried out in linear time on the binned data. Since the cardinality is always smaller or equal than the input data set, the whole circuit will run in linear time, and require linear memory to store the sorted view.

Histograms as a side effect. As already mentioned, an accelerator in the data path is most useful if it never slows down the regular flow of data to the host. To fulfill this goal, our design, shown in Figure 9, has a dedicated cut-through path for the data going from storage to host. The actual statistical calculation is performed on a copy of the input, obtained by introducing a *Splitter* in the data path that duplicates the data flow. In terms of throughput, the FPGA can easily implement complex logic at 10 Gbps or even 100 Gbps throughput [17, 11, 3], so a cut-through path

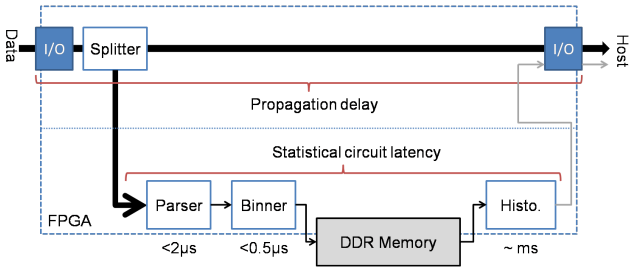

Figure 9: The FPGA implements cut-through logic for the data stream, and performs statistical operations on the side

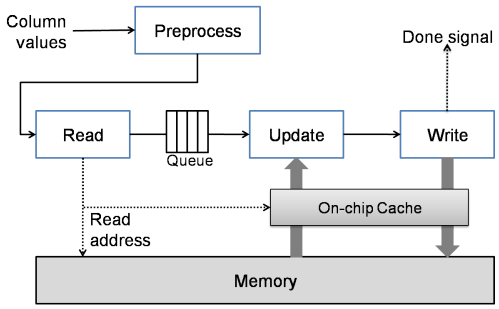

Figure 10: Binner: Module that processes the input stream into bins in memory

should never be the bottleneck. As for latency, the delay introduced by the FPGA on the data is the combination of the Splitter and the I/O logic of the FPGA. The latency of the Splitter is in the order of nanoseconds, as it only needs to replicate the data, and the I/O logic's latency usually is in the order of microseconds, depending almost exclusively on the transmission medium and protocol. When compared to the retrieval time of large database tables, these latencies become negligible.

The latency of the core statistics circuit, which operates asynchronously and in parallel to the cut-through logic, can be broken down to 1) parsing, 2) binning and 3) histogram building. The time it takes for the Parser to extract the relevant information from the input rows depends on the data source type, but based on the current state of the art for FPGAs and with a conservative estimate, it can be performed with an FSM below 2μs for all data source types. The Binner module's latency is dominated by the memory access latency, which in our development platform was measured to be on average around 0.4μs (60 cycles at 150 MHz). The latency added by the Histogram module is the only latency in the millisecond range, and is not constant but grows with the number of bins residing in memory. A detailed discussion on the latency of different operations in this module is presented in Section 6.3.

Memory acts as a decoupling element between the Binner and the Histogram module, as they interact in a producer-consumer-like manner. As a result of this separation, only the throughput of the Binner module needs to be large enough to handle all input data without dropping rows. As we discuss in Section 6, in our prototype the bottleneck is the memory access rate, but even so we can handle sufficiently high input rates.

Based on the above analysis it is safe to say that the FPGA indeed is just a "bump in the wire", and that it will not degrade the read/write performance of the system. Since the histogram calculation happens in parallel to data transmission, and does not interfere with it.

5. IMPLEMENTATION

In this section, we focus on the implementation of the two most important modules of the statistical circuit: the Binner and the Histogram module.

5.1 Binner Module

Figure 10 shows the high-level view of the Binner module, composed of a preprocessor and a pipeline that interacts with off-chip memory. It is configured and fed with data by the Parser module, and its control output is connected to the Histogram module.

5.1.1 *Preprocessor*

Before the bins can be updated in memory, the input values need to be translated into actual memory addresses. The transformation from value space (contents of the column) to address space (the corresponding memory location) is the simplest for integer-type columns: for these it is enough to subtract the minimum value in the column from all values, and use the difference as an address. It is also possible to divide the values by some constant to assign multiple values to the same bin. This is useful when the granularity of interest is higher than that of the underlying data type. As an example, a column could store date values that change with the granularity of days, but the representation could be a second-based timestamp in the database. Additionally, databases often store data using their proprietary formats. For instance Oracle stores date objects not as a single epoch number, but unpacked, encoding the year, month, day, etc., explicitly [25]. The preprocessor can be used to convert a handful of predefined unpacked types to integers. This way, even if the internal representation is not a single integer, it is possible to calculate histograms on the column.

5.1.2 *Pipeline*

The Binner performs a bin-sort [7] operation, where values of the column are mapped to bins, and the counter in the bin is incremented on every appearance of the value. In software, this logic would be implemented as a tight loop:

```
for all input items {
  index = preprocess(input)
  array[index]++
}
```

The above representation however hides some of the complexity of interfacing with memory. In order to increment a bin count multiple operations have to be performed on the memory, which means that every iteration of the loop will take time in terms of memory latency. To hide this latency, we can break the loop up into a pipeline. We annotated each operation with the name of the pipeline stage that performs it below:

```
pipeline input items {
  index = preprocess(input)    [PREPROCESS]
  count = array[index]         [READ]
  newcount = count + 1         [UPDATE]
  array[index] = newcount      [WRITE]
}
```

These stages can be performed in parallel for different items, which means that we are not bound anymore by the memory

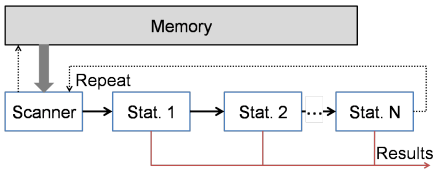

Figure 11: Histogram module: built from a series of statistic blocks which process the bins in a pipeline

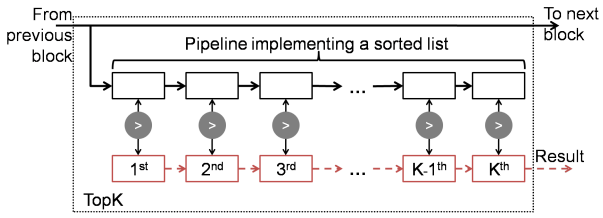

Figure 12: The TopK block implements a sorted list that stores the largest frequencies seen so far

access latency, but simply by the number of access operations the memory can handle per second, *i.e.* the pipeline can take full advantage of the memory.

The implementation of each pipeline stage follows intuitively from the pseudocode description. The *read* stage takes the addresses created by the preprocessor, and converts them into read commands to the memory. The physical read address in the memory may differ from the logical index of items because memory lines pack multiple bins (in our implementation eight). Once a read command has been issued, the logical address is put into the FIFO queue between the read and update stages. It is necessary to buffer the logical address so that the read stage can issue multiple commands to the memory while the *update* stage is waiting for the read data. When the memory provides the requested data, the *update* stage will pop an item from the queue, and increment the bin corresponding to the logical address and pass the updated memory line and the address to the next stage. The *write* stage will write back the memory line, and increment an internal counter that keeps track of the total number of items processed, which will be used in the histogram building phase. When the last input item reaches the write stage, the Binner module will send the total count to the Histogram module, signaling that it finished.

5.1.3 On-chip Caching

One issue with the pipeline presented above is that when two identical values follow each other, so called "read after write" conflicts can occur. Due to the high latency of the off-chip memory, it can happen that the second value will cause a read from memory before the incremented bin from the previous value could be stored back to memory. To overcome this issue, an often utilized technique for FPGAs is to simply stall the pipeline until it is certain that the first write was finished [12]. In our case this behavior is not acceptable, because it would mean that the processing speed of the circuit is dependent on the contents of a column. We want to guarantee same performance for the Binner module, regardless of the amount of skew in the data distribution.

To ensure that the circuit can always process items at the maximum speed allowed by the memory, we dedicate a small amount of on-chip memory to be used as a cache (1KB), which in essence forwards the values of recently accessed bins between the pipeline stages. The cache is a write-through cache whose size is adjusted to be able to store the maximum number of items that can arrive in the fixed timeframe defined by the memory access latency. The memory lines are stored on a small on-chip block RAM (BRAM), and this BRAM is indexed with the help of a lookup table that stores the memory addresses belonging to the items currently in the pipeline. Whenever a new item enters the pipeline, its data is either served from memory, or if fresher data is available, it is retrieved directly from the cache. The architecture of this cache is highly scalable, and it can be easily ported to other memory types with higher or lower latency.

5.2 Histogram Creation and Statistics

The histogram building step begins once all the values in a column are binned in memory. The histogram building is carried out by the Histogram module, which is a pipeline internally. As shown in Figure 11, the first block in the pipeline is the *Scanner* block that scans the area of memory that contains the bins associated with the processed column. The start address and the amount of memory to read is provided by the Binner module after it completed processing the last input item.

As soon as the Scanner block starts receiving data from memory, it passes on the bin counts to a daisy-chain of statistical blocks. These blocks implement different histogram types or calculate different statistical properties on the input data. What they have in common is that they always relay the input unchanged to the next block. As we will show later, there are some blocks that need multiple scans of the bins – these blocks can tell the scanner through a feedback channel (*repeat*) to read and stream the bins again if necessary. Apart from the input and output ports used for streaming the bin data, these blocks also have a "result output", a port on which they communicate the histograms to the host.

For the Histogram module we have implemented four different types of statistics algorithms (and as a consequence, there can be up to four different types of statistical blocks in the chain). These are: a list of the most frequent items (TopK), as well as histograms of type equi-depth, Max-diff and Compressed. In the following we will explain first how the TopK block and the equi-depth histogram work, and then show how, by simple modifications to these two, we can build blocks that calculate the other two types of histograms.

5.2.1 Basic Blocks

Some operations on the sorted bin data can be performed in linear time, with only one scan of the data. We have implemented two such operations: maintaining the list of K most frequent items (TopK) and an equi-depth histogram.

Equi-depth Histograms. The block that calculates the equi-depth histogram is initialized with the number of histogram buckets to create, and the total number of items as counted by the Binner module. The number of buckets to create is a parameter of this block that is stored in a small on-chip memory. As a consequence, it is possible to change this value when a read request is sent. This gives the host flexibility of choosing the granularity of the histogram.

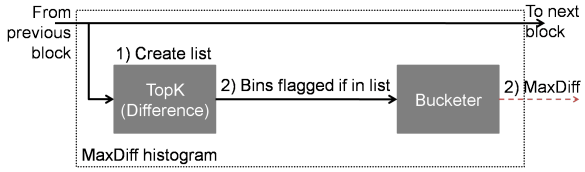

Figure 13: The Max-diff histogram is calculated in two scans, using modified TopK and equi-depth blocks

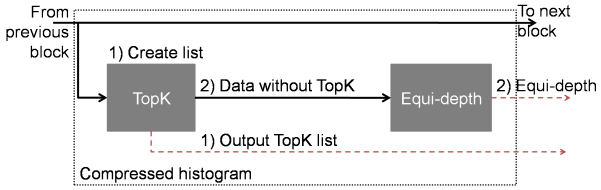

Figure 14: The Compressed histogram is calculated in two scans, using a modified TopK and an equi-depth block

In an equi-depth histogram the sum of frequencies in each bucket should be equal to the total sum of frequencies divided by the number of buckets. Therefore, as a first step, this block will divide the total count of items (provided by the Binner module) by the number of buckets this block is configured to produce, and store the result in an internal memory. When bins are streamed through this block, both a running sum and the count of bins are maintained. When the sum is larger or equal than the limit, a new bucket is created. While the buckets might be larger than the limit, they will always contain all appearances of the same item in one bucket. This behavior is identical to Oracle's *hybrid equi-depth* histograms. The final output of this block consists of the aggregate sum in the bucket and the number of bins in it.

TopK. One way of finding the bins with the highest count is to sort them by their count. To carry out this operation in software most likely an algorithm such as quick-sort would be chosen, but on the FPGA there is not enough on-chip memory to act as intermediary storage for all bins while being sorted (accessing the off-chip DRAM for sorting purposes is not an option due to the very high access latency). Our solution is to use a sorting method inspired by insertion sort (like in [31]) that can be implemented using a pipeline structure. The role of the TopK block is to provide only the K highest ranking elements, so the list used for the insertion sort can be bounded at K. The pipeline, shown in Figure 12, consists of items entering on the left side and propagating through all elements of the list. If an item reaches the position where it should be in the sorted list, and this location is still empty, it is stored there. On the other hand, if the location is already occupied by a smaller item, the two will be swapped, with the smaller item traveling further in the pipeline. This means that instead of shifting the whole list right for insertions (as one would do in software), the swapping is done in the pipeline. This design decision is important because it allows for a continuous flow of items without the need to stall the input while the list is being shifted. A consequence of the pipelined design is that the number of logical elements needed for implementing this block are a linear function of the length of the list. In our experiments, we synthesized the circuit with a list size of 64, but larger sizes are also possible. They are not necessarily needed, however, because the role of this block is not to sort the entire data, but to find the set of K highest-ranking values.

5.2.2 Composite Blocks

Based on the previous two blocks we can build more elaborate histograms. These composite blocks need two scans of the binned data in memory. On the first scan they preprocess the data using a modified TopK block. The histogram is created on the second scan, using a modified equi-depth block.

Max-diff. In order to calculate the Max-diff histogram, first the K largest differences between bins need to be determined. For this, we can use a modified TopK block which instead of looking at the size of the bins looks at the difference between two consecutive bins. This is achieved by putting a subtract logic at the entry to said block that will replace the count of a bin with the difference between consecutive bins. When the first scan has finished, the list of top K differences is kept in the TopK circuitry. As Figure 13 shows, on the second scan bins are compared to the elements of this list, and flagged when found (that is, the bins that created the K biggest differences are flagged). A modified equi-depth block will use the flag as a bucket-creation condition instead of comparing the current count to some threshold value. The output of this block has the same format as the original equi-depth histogram block's output.

Compressed Histograms. The Compressed histogram is even easier to obtain from the two building blocks we have (Figure 14). On the first scan the top K frequent items are determined and kept in the list of the modified TopK block. They are output at the end of the scan, just as the regular top frequency list would be but their values are not removed from the registers. On the second scan, the modified TopK block filters out all bins that appear in the frequent list (flagging them as invalid data), and forwards the rest of the data to an instance of the equi-depth block that then constructs an equi-depth histogram of the less frequent values.

6. EVALUATION

Our evaluation setup is a Maxeler workstation[2], which has a quad-core Intel I7 2600S processor and 32 GB of main memory, running CentOS 6. The FPGA is a *Xilinx Virtex-6 SXT475* chip on a PCIe card (8 lanes, Gen1). As shown in Figure 15, the FPGA has its own dedicated 24 GBs of DDR3 memory, accessible directly from the chip. Our custom circuit was clocked at 150 MHz for all experiments. While the platform at our disposal did not allow for using the FPGA directly as an "implicit accelerator", as we show in this section the performance of the statistical circuit can be accu-

[2] http://www.maxeler.com/products/desktop/

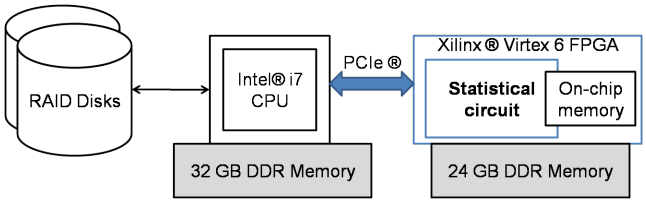

Figure 15: The Maxeler Box we used for our experimental evaluation has a Xilinx FPGA attached via PCIe

	Binner performance (values/second)	Equiv. rate for 1 column table	Equiv. rate for *lineitem*
Cache never hit (Worst)	20Million/s	80MB/s	2.9GB/s
Cache always hit (Best)	50Million/s	200MB/s	7.4GB/s
Pipeline (Ideal)	75Million/s	300MB/s	11.1GB/s

Table 1: Measured and ideal performance of the Binner module

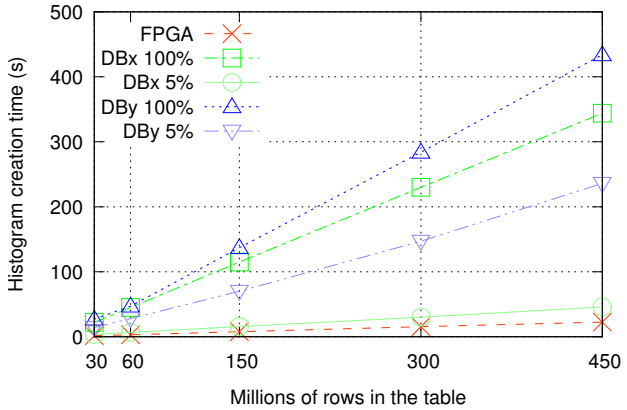

Figure 16: Effect of sampling on database histogram calculation

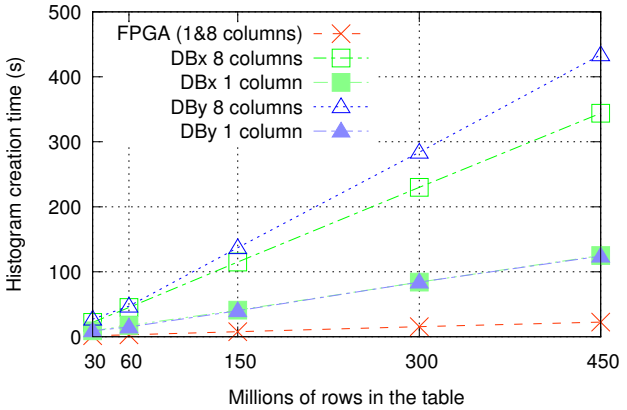

Figure 17: Reducing the number of columns in the table makes analysis faster

rately evaluated nonetheless. We compare the performance of the FPGA with two common commercial databases, referred to as *DBx* and *DBy* in the rest of this evaluation section.

6.1 Memory Access Rate

As explained in Section 5.1.2, the performance of the Binner module is determined by the off-chip memory: The maximum rate at which the Binner can handle input items is bound by the number of operations the memory controller can perform per second. We measured this number on our development system, and observed that it is relatively low because in the Maxeler workstation the memory attached to the FPGA is optimized for high bandwidth scans, not random access on small data chunks. Even so, as Table 1 summarizes, in the worst case, the memory can sustain 20 million updates to the bins per second, which translates to processing a one column table at 80 MB/s[3]. When processing a column of a table that has wider rows, such as *lineitem* from TPC-H , the rate that can be sustained for the table as a whole is significantly higher.

In case the data is heavily skewed, and therefore similar items often follow each other in bursts, it is possible to perform a higher number of updates per second. This is because in our implementation we do not issue read commands for items that are already in the cache. Consequently, the average number of memory accesses per bin update decreases, which in turn increases throughput. Additionally, when accessing rows in a less random manner, the memory also exhibits a higher access speed. Since the performance increase depends on the data distribution, we decided to conduct all our experiments with data that does not take advantage of the cache. In addition to the two measured values in Table 1, we also show the maximum throughput the statistical pipeline can achieve with sufficiently fast memory.

[3]This number is derived from the fact that the memory controller can handle 40 million read or write accesses per second in the worst case, and each bin update requires a read and a write operation. Assuming the input to be 32 bit numbers the 1 column throughput is 4B*20M/s=80MB/s

6.2 Histograms Building Speed

The goal of the following experiments is to show that our prototype accelerator can provide the same histograms as DBx and DBy at a fraction of the cost. The type of the histogram calculated on all three systems was equi-depth, left at the default bucket size for the databases, and set to 256 in the case of the FPGA. The tables we used for the experiments were cached in memory, and are derived from the *lineitem* table of the TPC-H benchmark. For the first measurements we created an eight column version of *lineitem* using the first eight numeric columns of the original table – this was achieved by truncating the output of the data generator. We used this setup to increase the number of rows that fit in memory.

Baseline. As a first experiment we measure a baseline on how much time it takes for the two commercial databases to calculate histograms on the eight column table created with TPC-H scale factors 5, 10, 25, 50 and 75. On the CPU we report the times to calculate the histogram on *L_quantity* as the execution time of the stored procedures which perform the statistical analysis; and for the FPGA we defined as runtime the time it takes from sending the first byte of the data through PCIe, until all result bytes from the FPGA were received. There is no delay inside the FPGA between receiving the first byte of the data and the start of the binning pipeline. Also, the FPGA is always processing the full input data, sampling is only done in case of the DBMSs.

Figure 16 shows that the FPGA performs much better than either of the databases, even when they use sampling. An interesting artifact of the graph is that the runtime of DBy does not decrease proportionally with the decrease in sampling rate – this means that for very large tables DBy

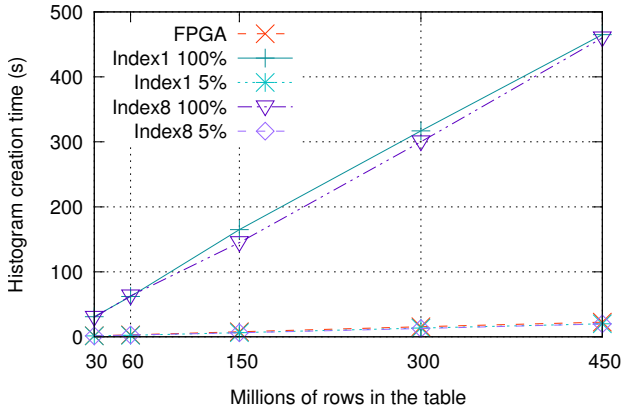

Figure 18: Calculating histograms on indexed tables in DBx

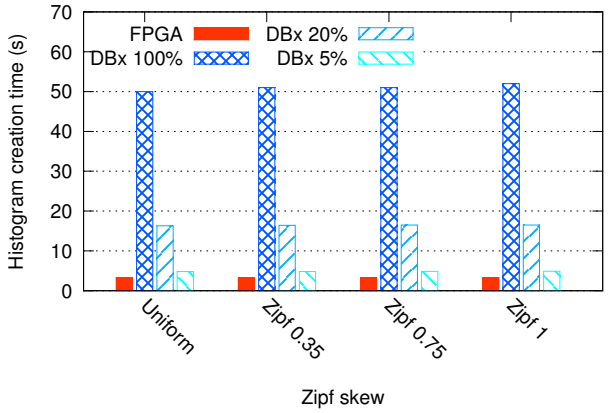

Figure 20: Effect of skew in columns on analysis time

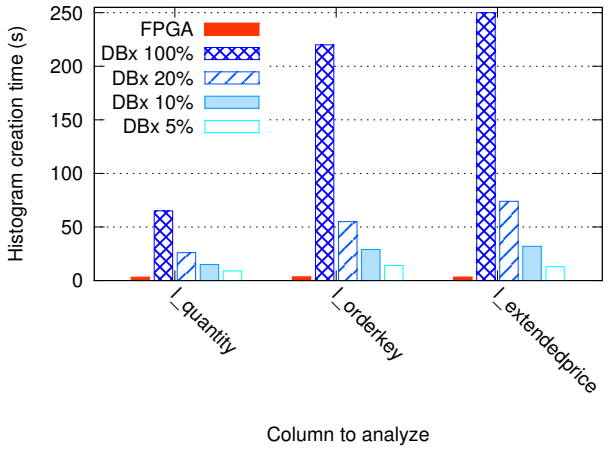

Figure 19: Effect of cardinality on histogram creation

might not be able to provide any sophisticated statistics if the time limit for acquiring them is too small.

As a follow up experiment, we reduced the *lineitem* table to a one column version, and repeated the previous measurements without sampling on this table. By having only one column, the runtimes for the databases should reflect the true CPU time needed to analyze a single column without other overheads. Figure 17 shows that even in this best-case scenario for DBx and DBy performing analysis without sampling will still take almost an order of magnitude longer than on the FPGA.

Indexed column. The third set of experiments that we show targets only DBx because it has the functionality of calculating histograms on indexes, whereas DBy does not. We created an index both on the 8 column *lineitem* (Index8) and the 1 column variant (Index1). Figure 18 shows how the performance of DBx improves when dealing with indexed columns – independent of the width of the rows the index was created on. This is most likely due to the fact that the index is a sorted representation of the underlying data, and hides the width of the original rows. Actually, with 5% sampling DBx is so fast that it catches up with the FPGA. The FPGA, however, is doing full table scans in this time. It is also important to note that 1) the creation and maintenance of the index introduces high costs, which are not represented at all in this graph, and 2) the histograms created are not equivalent because the FPGA version is built on the complete view of the data and the other only on a small sample of it.

Skew and Cardinality. As further exploration of histogram calculation costs, we compared the times for different columns of the *lineitem* table (scale factor 10), having different cardinalities and data types. Figure 19 shows that low-cardinality columns (like *l_quantity* with less than 100 different values) are cheaper to analyze than very high cardinality columns (such as *l_extendedprice* and *l_orderkey*). These results suggest that dealing with fixed-point arithmetic instead of integers introduces further overheads in DBx. The FPGA on the other hand is not noticeably influenced by the cardinality of the input dataset. A detailed discussion on the exact cost for the histogram building phase in the FPGA is presented in Section 6.3, but for tables with millions of rows or more, these costs are negligible.

Figure 20 shows that, as opposed to cardinality, skew has little effect on runtimes. The table we built for this experiment is populated with synthetic data (8 columns), with a cardinality of 2048 and Zipf distribution.

Automatic choice of sampling rate. While it is true that sampling can yield very accurate approximations of the actual data distributions [5], it can also happen that if the data is under-sampled important features are lost. What is even worse is that randomly, some features do not always show up in the histogram, which in turn leads to oscillations in the query plan. To illustrate how even very small errors in the histogram can lead to inadequate query plans, we revisited the query Q_1 from Section 2 using PostgreSQL. We chose PostgreSQL for this experiment because it is open-source and by modifying the analyzer's source code it is possible to force results into the histogram data structure.

We made a few changes to the query[4] and introduced small spikes at random in the data distribution of the *lineitem* table (scale factor 1): for a handful of prices we increased their occurrences to 2000 each, which is a very small skew given that there are six million rows in the table. PostgreSQL's automatic sampling detects these spikes only with roughly 50% probability each, and this leads to an oscilla-

[4]We introduced an ORDER BY clause in the first part and are joining on equality; This is needed so that PostgreSQL will consider not only Nested Loops Join in the optimizer

Block	Resource Usage	Resource Scaling	Result Latency	Result Size	Scans	Max. Freq.
TopK	2.5% (T=64)	$\mathcal{O}(T)$	2Δ+2T	T * 8bytes	1	170MHz
Equi-depth	<1%	$\mathcal{O}(1)$	2Δ/B	B * 8bytes	1	240MHz
Max-diff	<3% (B=64)	$\mathcal{O}(B)$	(2Δ+2B) + 2Δ/B	B * 8bytes	2	170MHz
Compressed	<3% (T=64)	$\mathcal{O}(T)$	(2Δ+2T) + 2Δ/B	(T+B) * 8bytes	2	170MHz

Table 2: Summary of properties and resource consumption on a Virtex6 FPGA of the four statistical blocks

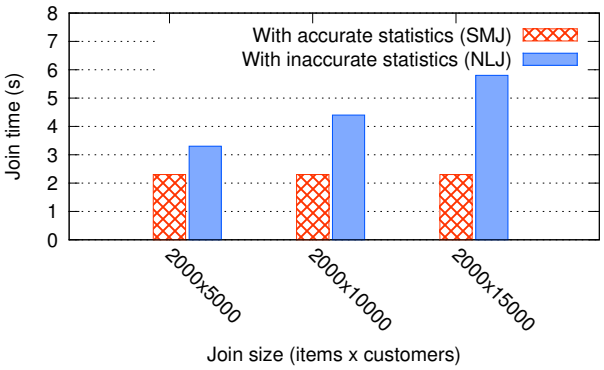

Figure 21: In PostgreSQL wrongly chosen query plans can lead to significant performance differences

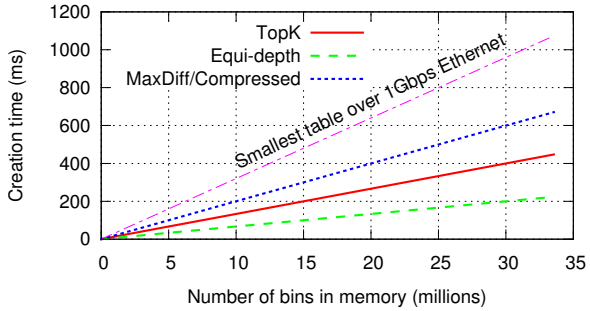

Figure 22: The time to process the binned representation grows linearly with the bucket count for every block

tion between a Nested Loops (NLJ) and a Sort Merge Join (SMJ) based query plan every time the table is analyzed. As Figure 21 shows, the runtimes are very different for the same data depending on the accuracy of the histograms. Furthermore, as the number of records participating in the join is increased, the more significant the difference becomes. When forced to use accurate histograms, PostgreSQL will correctly use Sort Merge Join.

6.3 Statistics Blocks

In addition to evaluating the accelerator as a whole, we present the different properties of the four statistical blocks (TopK, Equi-depth, Max-diff and Compressed) that were implemented for this paper. The most important input parameters to the Histogram module are:

- T : Number of top frequencies (exact counts) to maintain, and number of frequencies to remove in Compressed histograms
- B : Number of buckets to create for the equi-depth, Max-diff and the Compressed histogram
- Δ : The number of bins in memory that have to be read out for histogram creation

The most important properties of the four blocks are:

- Resource consumption : This property shows how much real estate each block is occupying. We express this as a percentage of the total resources on our FPGA.
- Resource scaling : This property shows how the block scales in terms of logic resource usage with increased bucket/frequency counts. This is an important metric because it shows how large each block can be made on a given FPGA.
- Result latency : This is the amount of time it takes for the block to produce the first result byte from the moment the first bin is retrieved from memory. This is expressed in number of clock cycles, which at 150 MHz last for 6.6 ns.
- Result size : The size of data in bytes that is output as a result of the statistical algorithm. Since each bucket is output as a pair of 32-bit integers, each bucket needs 8 bytes.
- Scans : How many times the block needs to scan the bins in order to finish its respective algorithm.
- Maximum frequency : The maximum frequency at which the given block can be clocked on the chip. This number is important, because in an FPGA design the final clock rate has to be chosen to be lower or equal than the minimum of all blocks.

Table 2 summarizes these properties for the four blocks we have implemented. Apart from the Equi-depth block all the others need logic resources relative to the number of buckets they produce. Consequently, these should always be scaled to a number of buckets that ensures good quality histograms but still fits on the FPGA. In terms of result latency, again the Equi-depth block will return the first bucket as soon as the threshold has been reached, but the other blocks take longer to create their results. The TopK block needs to read all bins before the top-list can be finalized. Similarly, both the Max-diff and Compressed histograms will start outputting buckets only once the data has been read in its entirety. With respect to pass-through latency, each block adds a constant number of cycles, which are necessary for copying the data into the internal logic. In our implementation this latency is 2 cycles per block. This means that with all four blocks on the chip, the last one will receive the contents of the first bin only 40ns (6*6.66ns) later than the first one.

To quantify the cost of creating histograms from the binned representation in our implementation we measured the time it takes for each block from the first retrieved bin, to the last result bucket output. As can be seen in Figure 22, this time

increases linearly for each type of block, and the Max-diff and Compressed take as long as the TopK and equi-depth combined. This is not surprising, because these are a combination of the latter two. The reason for the TopK block to take longer than the equi-depth is that depending on the contents of the top-list, it can take two cycles to process a single input item, while the equi-depth always takes one cycle per input. The times in the graph are not additive, that is, if more blocks are chained together, their completion times will still be as shown in the graph. As a point of reference, Figure 22 also includes the minimum time it would take to stream a 1 column table over 1 Gbps Ethernet. Since we are targeting big data use-cases, it is very likely that the total number of rows in a table is orders of magnitudes higher than the cardinality of the column – making the cost of these histograms negligible.

Histogram variety. While we showed in the previous subsections that the FPGA can calculate histograms faster than software databases, we did not compare their feature set yet. In the following, we compare the types of statistics that four popular databases use with what the FPGA can offer. The Oracle database engine creates either equi-depth histograms (end-balanced or simple) or TopK representation on the data [26]. IBM DB2 and PostgreSQL gather both simple equi-depth histograms and TopK statistic [14, 28] on the data. Microsoft SQL Server calculates only Max-diff histograms [20]. The FPGA can provide TopK and equi-depth together with no additional cost. This means that as long as the FPGA processes at least as much of the data as the databases it will always provide the same, or more accurate, histograms as the databases. However, on top of this standard set of statistics, for no added cost, the FPGA can also create Max-diff and Compressed histograms.

7. FUTURE WORK

A next step for our prototype would be to scale it up to be able to handle single columns arriving at 10 Gbps line rate. Since in our architecture the memory is the immediate bottleneck, the first step would be to move the prototype to an FPGA board with faster memory than our current development board. Then the Parser and Binner modules would become the next bottleneck. However, these modules could easily be replicated to provide the aggregated throughput of 10 Gbps (Figure 23) – in fact, achieving higher data rates by replication is a common practice in FPGA development. The input items copied from the pass-through line can be distributed in a round-robin fashion to the different copies of the Binner module, which will calculate the partial counts in their memories.

The histogram module would not need to be modified even if the circuit is targeting higher input rates because it is completely decoupled from the input stream through the memory. If there are multiple memory chips used in the system, the partial counts can be aggregated in constant time before they are fed into the statistical blocks.

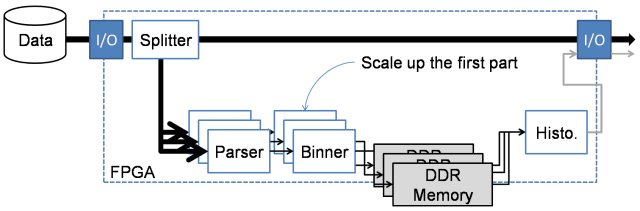

Figure 23: Higher data rates can be achieved by scaling up on future platforms

8. CONCLUSION

In the era of big data, data processing happens in increasingly distributed and heterogeneous environments. As a result, the choice of the best processing platform for a particular data processing task is crucial. As the state of the art in database design shows, statistics are successfully used to derive the right execution plans, but unfortunately obtaining these statistics is becoming increasingly difficult. In this paper, we propose an approach to change the traditional way of calculating histograms in batches, to an always-on statistical accelerator. This accelerator, which sits in the data path, acts as a bump in the wire and creates histograms for the data that it relays.

We have implemented the proposed idea on an FPGA to show the feasibility of moving the statistical logic into small dedicated circuits. We show both a speedup for the histogram calculation as well as the potential increase in accuracy when compared to two commercial databases. We also discuss the current platform limitations and propose ways of removing them.

Acknowledgements

This work is funded in part by grants from Xilinx, as part of the Enterprise Computing Center (www.ecc.ethz.ch), and Microsoft Research, as part of the Joint Research Center MSR-ETHZ-EPFL. The FPGA equipment used in the paper was acquired under the Maxeler University Program.

9. REFERENCES

[1] H. Angepat, D. Chiou, et al. An FPGA-based in-line accelerator for Memcached. *IEEE Computer Architecture Letters*, 99, 2013.

[2] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. Orthogonal security with Cipherbase. *Proc. of the 6th CIDR, Asilomar, CA*, 2013.

[3] M. Bando, N. S. Artan, and H. J. Chao. Flashlook: 100Gbps hash-tuned route lookup architecture. In *High Performance Switching and Routing, 2009. HPSR 2009. International Conference on*, pages 1–8. IEEE, 2009.

[4] J. Casper and K. Olukotun. Hardware acceleration of database operations. In *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, pages 151–160. ACM, 2014.

[5] S. Chaudhuri, R. Motwani, and V. Narasayya. Random sampling for histogram construction: How much is enough? In *ACM SIGMOD Record*, volume 27, pages 436–447. ACM, 1998.

[6] Convey. Memcached with hybrid-core computing white paper. `http://www.conveycomputer.com/files/6113/7998/5068/CONV-13-047_MCD_whitepaper.pdf`, 2013.

[7] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[8] T. P. P. Council. TPC-H benchmark specification, 2008. http://www.tcp.org/hspec.html.

[9] C. Dennl, D. Ziener, and J. Teich. Acceleration of SQL restrictions and aggregations through FPGA-based dynamic partial reconfiguration. In *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, pages 25–28. IEEE, 2013.

[10] J. Do, Y.-S. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt. Query processing on Smart SSDs: Opportunities and challenges. In *Proc. of the 2013 ACM SIGMOD Conference on Management of Data*, pages 1221–1230, New York, NY, USA, 2013.

[11] J. J. Garnica, S. Lopez-Buedo, V. Lopez, J. Aracil, and J. M. G. Hidalgo. A FPGA-based scalable architecture for URL legal filtering in 100GbE networks. In *Reconfigurable Computing and FPGAs (ReConFig), 2012 International Conference on*, pages 1–6. IEEE, 2012.

[12] R. J. Halstead, B. Sukhwani, H. Min, M. Thoennes, P. Dube, S. Asaad, and B. Iyer. Accelerating join operation for relational databases with FPGAs. In *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, pages 17–20. IEEE, 2013.

[13] M. Heimel and V. Markl. A first step towards GPU-assisted query optimization. In *The Third International Workshop on Accelerating Data Management Systems using Modern Processor and Storage Architectures, Istanbul, Turkey*, 2012.

[14] IBM. DB2 Version 10.1 for Linux, UNIX, and Windows. http://pic.dhe.ibm.com/infocenter/db2luw/v10r1/index.jsp.

[15] IBM/Netezza. The Netezza data appliance architecture: A platform for high performance data warehousing and analytics, 2011. http://www.redbooks.ibm.com/abstracts/redp4725.html.

[16] Y. Ioannidis. The history of histograms (abridged). In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 19–30. VLDB Endowment, 2003.

[17] Z. Istvan, G. Alonso, M. Blott, and K. Vissers. A flexible hash table design for 10Gbps key-value stores on FPGAs. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pages 1–8. IEEE, 2013.

[18] S.-W. Jun, M. Liu, K. E. Fleming, et al. Scalable multi-access flash store for big data analytics. In *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, pages 55–64. ACM, 2014.

[19] Microsoft. SQL Server Autostat Functionality. http://support.microsoft.com/kb/195565.

[20] Microsoft. SQL Server Documentation: Statistics. http://technet.microsoft.com/en-us/library/ms174384.aspx.

[21] R. Moussalli et al. Accelerating XML query matching through custom stack generation on FPGAs. In *HiPEAC'10*, Pisa, Italy, Jan. 2010.

[22] R. Mueller, J. Teubner, and G. Alonso. Data processing on fpgas. *Proceedings of the VLDB Endowment*, 2(1):910–921, 2009.

[23] R. Mueller, J. Teubner, and G. Alonso. Streams on wires: a query compiler for fpgas. *Proceedings of the VLDB Endowment*, 2(1):229–240, 2009.

[24] M. Najafi, M. Sadoghi, and H.-A. Jacobsen. Flexible query processor on FPGAs. *Proceedings of the VLDB Endowment*, 6(12):1310–1313, 2013.

[25] Oracle. Call Interface Programmer's Guide. http://docs.oracle.com/cd/B28359_01/appdev.111/b28395/oci03typ.htm#g467721.

[26] Oracle. White Paper on Understanding Optimizer Statistics, 2012. http://www.oracle.com/technetwork/database/focus-areas/bi-datawarehousing/twp-optimizer-stats-concepts-110711-1354477.pdf.

[27] V. Poosala, P. J. Haas, Y. E. Ioannidis, and E. J. Shekita. Improved histograms for selectivity estimation of range predicates. *ACM SIGMOD Record*, 25(2):294–305, 1996.

[28] PostgreSQL. 9.3 On-line Documentation: Statistics. http://www.postgresql.org/docs/9.3/static/view-pg-stats.html.

[29] M. Sadoghi et al. Efficient event processing through reconfigurable hardware for algorithmic trading. *VLDB'10*, Sept. 2010.

[30] M. Sadoghi, R. Javed, N. Tarafdar, H. Singh, R. Palaniappan, and H.-A. Jacobsen. Multi-query stream processing on fpgas. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 1229–1232. IEEE, 2012.

[31] J. Teubner, R. Mueller, and G. Alonso. FPGA acceleration for the frequent item problem. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 669–680. IEEE, 2010.

[32] J. Teubner, L. Woods, and C. Nie. Skeleton automata for fpgas: reconfiguring without reconstructing. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 229–240. ACM, 2012.

[33] P. Vaidya et al. Symbiote: A reconfigurable logic assisted data stream management system (rladsms). In *SIGMOD'10*, Indianapolis, IN, USA, June 2010.

[34] L. Woods, J. Teubner, and G. Alonso. Complex event detection at wire speed with FPGAs. *Proceedings of the VLDB Endowment*, 3(1-2):660–669, 2010.

[35] L. Woods, J. Teubner, and G. Alonso. Less watts, more performance: an intelligent storage engine for data appliances. In *Proceedings of the 2013 international conference on Management of data*, pages 1073–1076. ACM, 2013.

[36] Y.-H. E. Yang, W. Jiang, and V. K. Prasanna. Compact architecture for high-throughput regular expression matching on FPGA. In *ANCS'08*, San Jose, CA, USA, Nov. 2008.

[37] Q. Zhu, B. Dunkel, N. Soparkar, S. Chen, B. Schiefer, and T. Lai. A piggyback method to collect statistics for query optimization in database management systems. In *Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research*, page 25. IBM Press, 1998.