Build your own OctopusDB: Blinktopus Edition

Pavlo Shevchenko, Ali Hashaam, Guzel Mussilova, Ali Memon
Otto-von-Guericke-University, Magdeburg
firstname.lastname@st.ovgu.de

***Abstract*—The support for HTAP workloads is an important challenge in the design of contemporary DBMSs. The space of approaches proposed is quite vast, encompassing systems that leverage shared scans, log-based storage, adaptive layouts, and memory snapshots, among others. In this paper we evaluate a part of this design space. To do this we build upon the ideas of OctopusDB, an HTAP system that exploits logs as primary storage and performs copy-based online layout reorganization as part of query processing. We observed that despite of the design advantages, for OctopusDB like for other HTAP systems it is still a challenging task to guarantee a "freshness" over real-time data for OLAP queries. While the techniques related to AQP can provide interactive response times in answering OLAP queries. Thus, we believe that the exploration of AQP techniques can facilitate in enhancing the real-time properties of HTAP systems. Inspired by these ideas we introduce new "Log-based storage + Different Data Layouts + AQP" system named Blinktopus[1]. We evaluate the performance of OLAP functionality of the OctopusDB architecture, based on our own implementation. We then demonstrate the impact of implemented AQP components on Blinktopus performance.**

Through our implementation we also found that there are design issues that need to be addressed in OctopusDB. In conclusion we outline that the centralized log without compaction or any optimization is one of the current problems in the design of OctopusDB. Also, it is not clear if the concurrency control scheme is efficient or needs improvement.

I. Introduction

Over the last decades we are witnessing that modern enterprises need to pick only specialized DBMSs, (e.g. OLAP, OLTP, streaming systems and etc.) each tailored; to their specific use-case. Consequently, it leads to additional costs in terms of licensing, maintenance, integration and man-hours for DBAs. Although, it is affordable for some companies to adapt these integrated solutions to constantly changing workloads and requirements, it may still be a challenging and non-trivial task to achieve. Thus, in order to cope with these problems an implementation of a new all-purpose system could be a perfect solution.

Nowadays there exist a great variety of systems that claim to solve the aforementioned problems and yet their cost might be quite prohibitive. Some traditional DBMSs (e.g., Microsoft SQL Server, Oracle, ...) have already included the support of both analytical (OLAP, which is characterized by long-running queries over all the values of few columns) and transactional (OLTP, characterized by short-lived transactions that affect multiple attributes of few rows) workloads. Meanwhile, in the last 15 years these systems have been observed to be inefficient for new memory-rich architectures. As a consequence, exploiting the benefits from larger memory, new DBMSs have been proposed which have simpler, more efficient architectures than their traditional disk-based counterparts. Among these recent solutions are the column-stores (e.g., C-Store[13], MonetDB[8], ...) and the row-stores (e.g., Hekaton[7], H-Store[14], MemSQL[15], ...) that are particularly designed for analytical and transactional processing respectively.

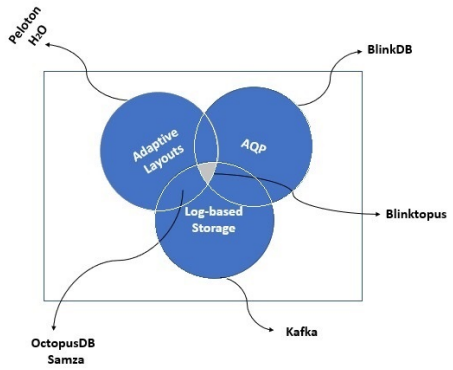

Figure 1. Integration of Contemporary Database Techniques into Blinktopus.

Still, following the *one size does not fit all* observation these systems are mainly specialized either for OLAP or for OLTP workloads[6]. Thus, it has lead the various vendors to try to build the comprehensive solutions, namely Hybrid Transactional/Analytical Processing (HTAP) systems (the term HTAP was defined by Gartner[17]). Some examples including SAP HANA[9] which has engines that are optimized for OLAP workloads, while supporting ACID transactions at the same time. HyPer[10] is another example, which has a hybrid approach that uses memory snapshots based on process forking. Other examples include Peloton[11], OctopusDB[1] and SnappyData[16] also belong to HTAP systems.

One of the solutions that most radically departs from existing architectures was proposed by Jens Dittrich and Alekh Jindal - a new type of database system, named OctopusDB[1]. By dynamically mimicking several types of systems, (OLAP, OLTP, Hybrid of OLAP and OLTP, etc.) OctopusDB is a solution that combines a copy-based mechanism for supporting adaptive layouts and logs as a primary storage. As a result, OctopusDB shows a considerably better performance. Moreover, depending on the use-case it may also emulate data stream

[1]The code and corresponding documentation can be found in the following link: $https://github.com/Urmikl18/Blinking\_Octopus$

management systems by replacing some of the stores with streaming windows.

Another important goal of HTAP systems (aside from maintaining different system components compatible in order to create an illusion of a single system) is to support OLAP queries for analysis over real-time data. The fact that HTAP systems might reduce the need for transforming data from one system to another (via ETL), seems like a good step towards the support an analysis over real-time data. We believe that the exploration of the techniques related to more interactive queries can contribute to the real-time characteristics of HTAP systems. Among the techniques that can handle more interactive queries, Approximate Query Processing (AQP) have recently gained a substantial attention. By processing the compact synopsis of the data rather than the whole dataset, the methods for AQP are often the only viable solution to provide interactive response times when exploring massive datasets and handling high speed data streams[12]. Several successful examples (e.g. BlinkDB, SnappyData on Facebook's Presto) have already proved that there are benefits to be gained by integrating approximation features into existing DBMSs[5].
In this paper we want to evaluate the role that AQP can play as an architectural addition in a HTAP system like Octopus DB, for facilitating real-time queries on the latest data. We believe that when it is possible for a system to retrieve results approximately rather than exactly, AQP could be a reasonably good fit to further improve the performance of query processing (especially OLAP queries). It could also enhance HTAP by answering the OLAP queries over new data that even has not been included yet and when it is guaranteed that a given amount of incoming data will not change the error of an estimation. Combining OctopusDB and AQP techniques could be feasibly a good solution for improving HTAP *freshness* and the performance of our system. In this paper we provide an early evaluation on these aspects.

Our contributions are:

- We review the ideas of OctopusDB and AQP along with the concepts of AQP main data synopses (Section 2).
- We propose a novel concept of a system called Blinktopus and explain why exactly some types of AQP data synopses were chosen for our system (Section 3).
- We provide an experimental evaluation on the benefits of Blinktopus's functionality based on the results of OLAP queries performed on the Storage Views (SVs) with the different physical layouts, namely LogSV, ColSV and RowSV. In the second experiment we compare the above-mentioned results with the results obtained from AQP storage views (Section 4).
- We discuss related work (Section 5) and future directions for our research, including necessary improvements to the Blinktopus design (Section 6).

II. Fundamentals

Firstly, we discuss the core idea of OctopusDB, its motivation and architecture (Subsection A). Afterwards we explain the main concept of AQP and elaborate AQP on the main data synopses such as samples, histograms, sketches, wavelets and etc (Subsection B).

A. OctopusDB

OctopusDB is one of the representatives of HTAP or *one-size-fits-all* architecture systems. It builds upon 2 ideas:
- Logs as a primary structure (which is similar to Samza[18], and other streaming systems).
- A storage engine with a programmable interface that allows users to specify how data or external architectural components will look like (e.g., RodentStore[23]).
All data in OctopusDB is stored in the central log named *primary log*. The data is being collected into the log through the creation of logical log-entries of the insert/update operations, each record is identified by internally unique log sequence number *lsn*. OctopusDB exploits the write-ahead loging (WAL) protocol and stores the primary log on durable storage (HDD or SSD). Depending on the workload, a so-called Storage View (SV) can represent the entire central log or some of its parts in different physical layouts (e.g., row-store, column-store, PAX, index etc.). This feature also makes OctopusDB an HTAP system which supports adaptive layouts. For instance, for OLTP queries OctopusDB can create Row SV, Col SV - for OLAP. Moreover, based on given queries, it can optionally decide to create other materialized view such as Index SV or even to mimic streaming systems. At the same time, primary log is another type of SV for OctopusDB.

A key component of Octopus DB called *Holistic Storage View Optimizer* solves a non-trivial optimization task of *storage view selection* (i.e., to determine automatically which type of SV is proper to be created for an observed workload). It maintains all SVs including primary log as well as it is responsible for creation and maintenance of secondary SVs. By means of scanning the indices and whole tables, later depending on cost model the optimizer resolves either to create new SV or to maintain an existing one or even to scan the log when none of SVs can answer the query. Furthermore, by applying a transformation cost model it might transform different SVs one into another. Additionally, it can remove from system all SVs. Thus, based on the workload the holistic SV optimizer operates a *storage view lattice* (i.e., the dependency graph between SVs), within the Storage View Store of the OctopusDB. This lattice organizes views that depend on others, for example one ColumnView could answer a range query over values A-J, while two dependent views in the lattice could answer ranges A-D and E-J.

In terms of consistency guarantees, OctopusDB fully supports ACID. The authors explain that *Consistency* can be ensured by validating the set of integrity constraints at commit time and *Durability* holds because OctopusDB keeps WAL. The *Isolation* algorithm of OctopusDB is represented via

optimistic concurrency control. In fact, a lock-free, append-only, log-based form of Multiversion Concurency Control (MCC). Whose basic concept is to store all committed or uncommitted changes in the primary log and to write only committed data in secondary SVs. Committed data is available for *read* by uncommitted transactions from log or any secondary SV. Moreover, the latter modifications are possible by adding records to the log but these data will not be further propagated to the secondary SVs until committed or other system configuration that allows to tune freshness. To that end, *Atomicity* is guaranteed by storing in SVs only committed transactions for their later considerations by other transactions.

However, it is not clear if the concurrency control scheme of OctopusDB is efficient. Although, it keeps with MCC, it does not have concepts for garbage collection (removing uncommitted versions) or for pointers between versions. These aspects might significantly slow-down the processing. Furthermore, the log structure might not be as cache-efficient as time-travel or delta tables, where versions can be stored together for high-performance access.

OctopusDB can be recovered by easily copying the primary log from durable storage to main memory. Meanwhile all SVs that were in the OctopusDB before the system crash will be re-created.

B. Approximate Query Processing

It is already evident that nowadays an enormous amount of data is being generated, processed and stored. Even the fastest systems can get stuck for hours in answering simple queries and this response time is less than satisfactory for users and applications. At the same time, in many cases (e.g., a/b testing, exploratory analytics, big data visualization and etc) providing the exact answers to a user query is not vitally important as long as estimations can result in the right decisions. AQP methods can achieve interactive response times (e.g., sub-second latencies) over a massive amount of data by processing only small fraction of the relevant records.

AQP systems differ according to their methodology and design options. AQP operates the datasets through its predefined types of summaries (e.g., *samples, histograms, wavelets, sketches* and so on) that capture main features of initial data while using less space. Such aspects as accuracy, range of applicability, space and time efficiency, error bounds on approximate answers strongly depend upon the chosen types of data synopses and their parameters. In addition, other issues in terms of data summaries (i.e., offline or online data summarization, various storage and maintenance strategies and etc) also need to be taken care of.

1) ***Samples****:* Perhaps the most researched and extensively implemented type of data summaries. Their prevalence was induced by many reasons. As one the oldest concepts in statistics, there exists a great variety of schemes to extract and maintain samples of data varying in precision and accuracy, such as *Bernoulli sampling, stratified sampling, random sampling with and without replacement* and others.

By using the same schema as the original relation, most queries can be performed over a sample (i.e. small "representative" subset of data) with slight or no alterations to existing systems, so they can answer the widest range of queries. By avoiding *AVI(Attribute Value Independence)* assumption, most sampling algorithms might support high quality selectivity estimation[19]. In spite that an immense diversity of queries can be evaluated using sampling methods, performing MIN, MAX, top-K and COUNT DISTINCT queries on a sample is quite impractical. Furthermore, most samples are not suitable for handling outliers in skewed distributions.

Moreover, sampling-based approximations do not suffer from "curse of dimensionality", i.e., their accuracy does not deteriorate with the increasing number of data dimensions.

Finally, adding the estimation from several samples can incrementally enhance an imprecise estimate of a query result in interactive exploration of large dataset in "Online aggregation" algorithms. Notwithstanding, when the data is constantly updated in the massive data streams and the majority of future queries are not known in advance, it is crucial to make sure of samples being kept optimal and up to date.

2) ***Wavelets****:* Another means of data synopses utilized by AQP systems over large datasets. The core idea of wavelet-based approximations is to transform the input relation in order to acquire a compact data summary that will consist of a small set of wavelet weights or coefficients. By capturing significant features of the massive dataset wavelets can ensure substantial data compression. While an appropriately-defined AQP algebra, which manages the domain of wavelet coefficients, assures answering range and more general SPJ (select, project, join) queries.

As an example of the wavelet synopsis *Haar Wavelet Transform(HWT)* is probably conceptually the easiest and therefore the most widely implemented wavelet transformation. Based on recursive pairwise averaging and differencing the resulting wavelets are straightforwardly computed and have been observed to show practically acceptable performance for numerous applications (e.g., image editing, querying to OLAP and streaming-data).

Wavelets, like histograms have the same "curse of dimensionality" limitation. Though this issue can be tackled by efficiently built wavelet decomposition, most implementations revolve around the one-dimensional cases, which can answer only a limited spectrum of queries. Moreover, the error guarantees provided by wavelets seem not to be always meaningful in the context of AQP[22].

3) ***Histograms****:* Another type of data synopses for summarizing the frequency distribution of an attribute or sets of attributes. Basically, a histogram groups input data values into subsets (i.e., "buckets", "bins") for each bucket it computes a small summary statistics in order to use it further for approximate reconstruction of the data in the bucket. In designing histograms the following aspects are to be carefully

considered.

- *Bucketing scheme*. Data items must be distributed within the buckets in order to better represent its structure. It can be decided depending on some local criteria (e.g., value frequencies or similarities etc.) or some global "optimality" criteria of the histogram according to a workload or a query class.

- *Intra-bucket approximation scheme*. In essence, there have been used two methods (e.g., *continuous value assumption* approach and approach of *uniform spread assumption*[24]) for intra-bucket approximation scheme of one-dimensional histogram. Both of them assume that values are uniformly distributed within the bucket range, with the first neglecting the number of the values and the second storing that number in the bucket. For the multi-dimensional histograms these approaches are to be extended. The choice of approximation scheme can also depend on the class of queries that histograms answer.

- *Statistics stored per bucket* can contain the number of items in each bucket along with the subset boundaries. The method chosen for the data approximation usually defines what information will be stored.

- *Bucket width*. The representation of real data via histograms may involve a loss of information, especially depending on the wrongly defined bucket width. Sturges's rule[30] is one of the commonly used rules. Moreover, other rules have been suggested that try to improve its performance without a normality assumption such as Doane's formula[29] and the Rice rule[28].

- *Class of queries answered*. Even though histograms can answer a wide spectrum of queries (e.g., queries related to "selectivity estimation", range-count queries etc.), class of queries answered still needs to be carefully considered while building a histogram.

- *Efficiency*. The cost of utilizing the histogram to provide query approximations, together with the time and space requirements can define the histogram from its efficiency perspective.

- *Accuracy* in answering queries to a given size constraint is one of the most significant aspects when constructing an "optimal" histogram.

- *Error estimates*. Many histogram construction approaches claim to provide some "average" or "maximum" error estimates over a set of queries, which however might not hold for a specific query, or even class of queries. As long as histograms answer queries approximately, it is a question of demand to determine exact error boundaries possible for an each query issued. At the same time, mostly when these error estimates are provided, there is no other details on the errors. Hence, some works propose to store the maximum difference between the actual and the approximate frequency of a value within the bucket and by exploiting that to provide upper bounds on the error estimates produced by the histogram for range and other queries.

- *Incremental maintenance*. This aspect is particularly significant when the values in the dataset are constantly and rapidly updated.

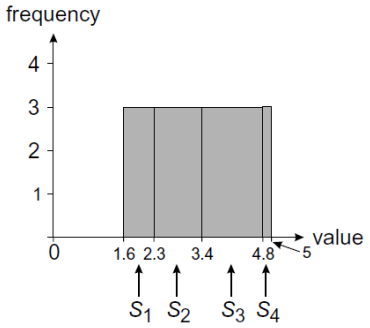

Figure 2. Equi-depth histogram on continuous data.[2]

As a result of an active research in the area of histograms over the last decades, a great diversity of histograms schemes have been proposed. As long as they also suffer from the "curse of dimensionality" limitation, therefore one-dimensional histogram is more preferred type.

The simplicity in implementation and interpretation contributed to the popularity of *equi-width* and *equi-depth* histograms. Equi-width histograms do not overlap among the ranges of attribute values within the bucket and independent of their value frequencies have the same range size (or the number) of values in every bucket[21]. Since they store a greater amount of information than trivial ones, therefore often demonstrate better estimations. The reverse of equi-width histograms - equi-depth (or *equi-height*) class of histograms has the same the sum of the frequencies of the attribute values with respect to each bucket, independent of the range size (i.e., the number) of these values. According to the studies of Piatetsky-Shapiro and Connell who also gave this class of histograms its name, equi-depth histograms have lower worst-case and average error values for a diverse set of selection queries than equi-width histograms. The example of equi-depth histogram on continuous data can be seen from Figure 2 [12]. Beyond the aforementioned types, others have been introduced such as *serial*[21], *end biased* and *high biased*, *maxdiff* and other generalizations.

The aspect also worth taking into consideration is a construction cost. With regard to it, histograms can be divided into two groups: *static* and *dynamic* or *adaptive* histograms[20]. Static histograms are commonly used in DBMSs. After they are built, even when the data is being changed, they remain unaltered. Hence, with the time a static histogram drifts away from its "up-to-date" state and its estimations may deteriorate with progressively larger errors. When this occurs, these histograms are recalculated and the old histograms are replaced with the "fresher" versions. Despite of the fact that there have been proposed some efficient calculation algorithms for static histograms, adaptive histograms are the only viable option in a data stream environment. The most known exam-

[2]Cormode, Graham, Minos Garofalakis, Peter J. Haas, and Chris Jermaine. "Synopses for massive data: Samples, histograms, wavelets, sketches." Foundations and Trends in Databases 4, no. 13 (2012): 1-294.

ples of dynamic histograms are *equi-depth and compressed* histograms[25], *v-optimal* and *spline-based* histograms.

Histograms efficiently answer range-sum queries and their variations. Furthermore, they can be utilized to approximate more general classes of queries (e.g., aggregations over joins), as well as real-valued data and set-valued queries.

As it is already known histograms are one of the most implemented data summaries. Since they can be incorporated into existing database system without obtaining an additional storage overhead, they are mainly exploited by query optimizers of almost all DBMSs to produce cost and selectivity estimations of various query plans. However, wrong estimations in query optimizers continue to be an important problem in database systems.

4) ***Sketches****:* They are especially germane for streaming data, where the data in a large scale is being processed and the sketch synopsis must be updated promptly and effectively without exploiting huge compute resources. Sketching algorithms use hash functions with defined mathematical properties that provide their guarantees, but in practice these hash functions are rather simple and rapid, with reference implementations that are broadly available.

Sketch summaries can be designed so that each new update will be independent of the latest state of synopsis. For instance, linear sketches consider a numerical dataset as a matrix or vector, and multiply the data by another fixed matrix. These types of sketches are easily parallelizable. Moreover, for the some queries (e.g., count distinct, most frequent items, joins, graph analysis and matrix computations), sketches may be the only viable solution.

Frequency based sketches summarize a dataset frequency distribution and provide accurate estimates of individual frequencies. This results in algorithms for calculating "heavy hitters" - items that account for a large portion of the frequency mass - and quantiles like median and its generalizations. These sketch summaries can similarly be utilized to approximate range queries, the sizes of (equi)joins between relations and self-join sizes. Moreover, such sketches may be exploited to extract histogram and wavelet representations of streaming data and as elements of more complex mining operations.

As was mentioned above sketch-based summaries are especially suitable for "distinct-value" queries (i.e., to count the number of distinct values in the dataset). Once built, these sketches can estimate the cardinality of a given attribute or combination of different attributes. The basic algorithm of sketch-based summaries is illustrated in Figure 3. Furthermore, they can calculate the cardinality of various operations performed on them (e.g., selections based on arbitrary predicates along with the set operations such as union and difference). One of the most well-known examples of using sketches for DISTINCT COUNT queries is a near-optimal probabilistic algorithm of *HyperLogLog*.

HyperLogLog methodology has been successfully implemented by Yahoo and improved the performance of its internal platform from days to hours and in some instances, even to

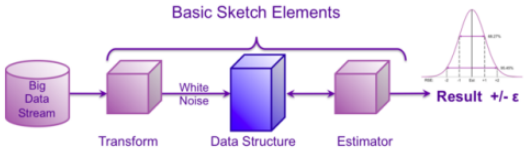

Figure 3. Basic scheme for using sketches.[3]

minutes[26]. It exploits an auxiliary memory of m units and by passing over the multiset once, estimates its cardinality with the standard error of nearly $1.04/\sqrt{m}$. For instance, while utilizing only 1.5 kilobytes of a memory, this algorithm is capable of estimating cardinalities beyond 10^9 with a relative accuracy of 2%[27].

Although, the sketching techniques can answer a considerable number of query types. However, they are not observed to scale well to more complex queries that imply sub-queries. Thus, the main drawback of sketches, especially in comparison with all-purpose sampling approaches, that they tend to answer a single type of query.

Finally, the target workload defines which data summary to implement in the system. Most AQP systems that tailored for accelerating a diverse predefined set of queries, exploit sketches, wavelets and histograms, while sample-based approximations are usually used by all-purpose AQP systems. As a rule of thumb, AQP systems that combine several data synopses seem to demonstrate considerably efficient performance[5].

III. Blinktopus

In this section we introduce a novel concept of our system called Blinktopus and give an explanation for the chosen types AQP data synopses implemented in Blinktopus.

A. Architecture

The core idea of Blinktopus derives from two cornerstones:

- Blinktopus is an adaptive system that according to user command mimics different types of systems (e.g., OLAP, OLTP) and as an output provides an SV which answers user queries. It is based on OctopusDB and aims to evaluate its potential for OLAP queries.
- Since AQP techniques guarantee better interactive response times to a user query (especially OLAP queries) over the "fresher" data, Blinktopus uses AQP techniques to further enhance its performance.

In the Figure 4 we present the architecture of Blinktopus. Similarly to Octopus, Blinktopus also stores all data in a *primary storage* or *primary log*. All data from the log or its some parts is represented via materialized SV namely *LogSV*, which is as another type of SV can be used for answering the queries. The component of *LogManager* defines the structure of LogSV and operates it within a *Primary Log Store*.

[3]https://yahooeng.tumblr.com/post/135390948446/data-sketches

In general, besides LogSV, Blinktopus can manipulate the dataset via other secondary types of storage views: column SV(*ColSV*), row SV(*RowSV*) and AQP SV(*AQPSV*). Although, the latter is rather different from the first two, Blinktopus still considers it as another storage view over the data. This simplifies the implementation and allows further views to be added. Along with the data, relevant to the query, such views contain other information like the number of entries or the range of a data.

Once created, the views can be accessed via *SVManager*. This component is responsible for storing the views in the *Storage View Store* and it also allows a user to get one or all of them and/or to remove them. In spite of the fact that the above-mentioned views can serve as the data storages, Blinktopus uses them only to provide the results to the user query.

Finally, the central component that connects all the components of Blinktopus together is *QueryProcessor*. It analyzes the query issued by a user and redirects it to the component which is responsible for it.

Since Blinktopus answers only range queries and histograms are especially suitable for this class of queries and also due to their simplicity in implementation and interpretation, together with acceptable error estimates we have chosen equi-depth type of histograms for AQP module of Blinktopus.

In order to calculate the number of buckets k we used the simplified and enhanced version of Sturge's Rule - *Rice Rule*.

$$k = \left\lceil 2n^{1/3} \right\rceil \tag{1}$$

where n - the number of elements in the dataset.

We decided that histograms in Blinktopus will answer COUNT queries. Therefore, for the aspect of "statistics stored per bucket" we store the number of elements with its lower boundaries per bin, along with the highest boundary for the rightmost bucket.

Additionally, to answer DISTINCT COUNT queries we included in our AQP module sketch-based data summaries based on *HyperLogLog* algorithm. We estimate the cardinality of a chosen attribute using *DataSketches* library by Yahoo. Finally, along with the cardinality we return to user upper and lower boundaries of the estimated error.

For both of the data synopses we calculate an absolute error, so the end user will not be overwhelmed with numbers that could seem difficult to interpret.

The front end part of Blinktopus is a user-friendly interface that facilitates user to act as *Holistic SV Optimizer* in OctopusDB, i.e., to decide which of RowSV or ColSV to create/delete, or to which SV (e.g., RowSV, ColSV, LogSV or AQPSV) to address his query.

Likewise, via this interface user is provided with opportunities to list all SVs created earlier in tabular or JSON format and delete all existing SVs. The front end and back end parts of Blinktopus communicate via JSON files. After receiving and interpreting JSON files from the back end, front end provides user the results in the tabular format. Additionally, along with the query results, the back end visualizes the pre-calculated query statistics in the form of bar charts (i.e., query runtimes and a comparison of the results performed over the different SVs).

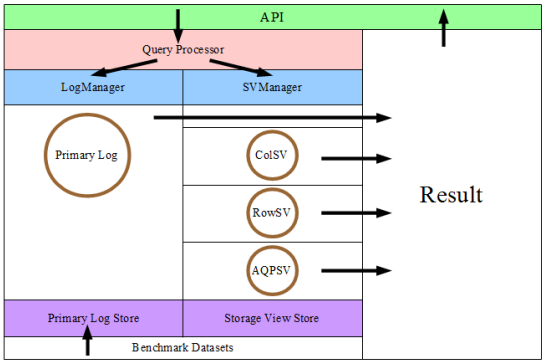

Figure 4. Blinktopus Architecture.

B. Workflow

In this subsection we describe the workflow of user query processing in Blinktopus.

- After query was issued, QueryProcessor interprets it. The query contains the table, column and a range of interest, as well as the SV type that user wants to use for the given query. In contrast to OctopusDB, where the Holistic SV Optimizer decided which SV to provide/create, in Blinktopus, this decision is made by user himself. However, a similar component that automatically will decide upon what storage view on a given query to provide can be easily plugged into the Blinktopus system.
- After the query had been analyzed, QueryProcessor redirects it whether to the SVManager, if the specified view is either Column, Row or AQP module, or to the LogManager.
- If user had chosen to look up over the log, then the whole primary storage will be scanned and the relevant tuples along with the information about the query (e.g. the table, the attribute, the range as well as query processing time) will be retrieved.
- If any other type of secondary SV was chosen, depending on the user's choice, the SVManager will either invoke the creation of the specified view or retrieve the required data from already existing SV.
- If user had chosen to query over the AQPSV, the function *queryHistograms* sums up the number of elements for every bin that lie exactly within a given range. For the histograms buckets whose ranges were not covered completely within the bins, in order to provide an approximate result we use *continuous value assumption*. This is a design decision that returns approximations for COUNT queries which might not be integer numbers. Nevertheless, with millions of records in the datasets, rounding the results by few decimals will also provide acceptable accuracy.

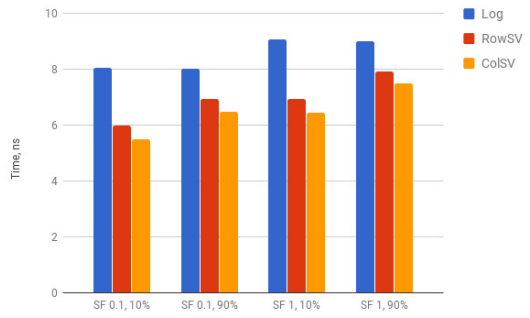

Figure 5. Experiment 1: Average response time for a range query on the Orders table. Predicate selectivity=0.1, 0.9.

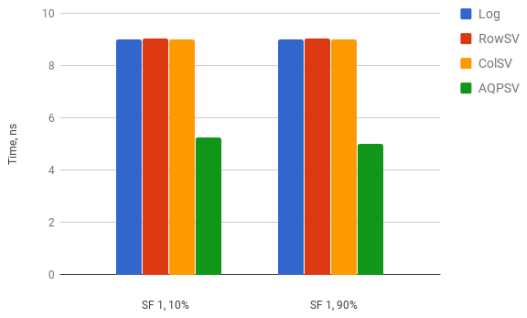

Figure 6. Experiment 2: Average response time for a count-range query on the Orders table. Comparison with an equi-depth histogram. Predicate selectivity=0.1, 0.9.

- In case user wants to know the number of distinct values over the whole data. These sketch-based approximated results will be returned using DataSketches library together with upper and lower boundaries of the estimated error.
- As it is not desirable for SVManager in order to retrieve the relevant tuples to scan the data, therefore SVs themselves are responsible for the result further retrieval and representation. Now when the view is created for the first time, the primary storage will be scanned and the relevant tuples will be formatted to fit to the desired view. Otherwise, an SV itself will scan the data it already stores and return to a user the only existing subset of the data.

IV. Experimental Part

In this section we provide the results and compare a query performance depending on various parameters (e.g. table size, query selectivity, query types) in the physical layouts presented in Blinktopus (LogSV, RowSV, ColSV and AQPSV). The Section is structured as follows: firstly, we discuss the benchmark datasets that were used and queries that were chosen for the experiment. Then, in respective subsections, we analyze the results obtained from three experiments and provide an outlook on the performance of Blinktopus system.

A. Setup

The experiments were computed on a commodity multi-core machine running CentOS Linux 7.1.1503 and Java SDK 8u131-b11-linux-x64, with 2 Intel(r) Xeon (TM) E5-2630 v3s CPU @ 3.2 GHz processors (8 cores each) and 1024 GiB of Memory.

In Blinktopus we exploit TPC-H datasets, namely *Orders* and *Lineitems* relations, but for our evaluation we decided to run queries only over *Totalprice* attribute from *Orders*.

For all experiments the numbers we report are an average runtime of each query for 100 runs. The ranges we have especially chosen for the queries cover 10% and 90% of Orders. Lastly, for better representation and interpretation the results were normalized to $\log_{10}$.

B. Experiment 1

The goal of the first experiment was to compare the performance of a range query over three storage views: Log, Row, and Column. For this purpose, we ran 100 range queries over two benchmark datasets with approximately 770k and 7,7M entries, and analyzed how the size of a dataset influences the performance of restricted storage views like Row and Column in comparison to Log. By restricted views, we mean that RowSV and ColSV contained only the tuples relevant to the query. Furthermore, for each dataset we ran queries with low and high selectivity to see how the amount of data, stored in a single view influences the query response time.

In Figure 5, we show the results of this experiment. As expected, Log store has the largest response time as it has to scan the whole data in order to retrieve relevant tuples. Row view performs better than Log, while Column storage view is slightly better than Row SV. Furthermore, we see that the restricted storage views do not become more beneficial with the increasing amounts of data, as the response time of all views is growing linearly with the increasing size of datasets. However, interesting results can be obtained while comparing the performance of Log, Row, and Column, depending on predicate selectivity. With the decreasing predicate selectivity (i.e., from 0.1 to 0.9), the speedup of RowSV and ColSV in comparison to Log is also decreasing: from 115 to 12 for RowSV, and 360 to 35 for ColSV. When the views contain all the data from given relation we expect the speedup to be between 1 and 12.

C. Experiment 2

In the second experiment, we analyze the performance of an AQP component in the Blinktopus system. For this purpose, we ran range queries with low and high selectivities over a larger dataset with approximately 7,7M records. We compared the performance of an equi-depth histogram to the one of Log, Row, and Column views. In this experiment, a histogram was created offline prior to query processing, and Row and Column views contain the data from the whole Orders table. Furthermore, we analyzed how the decrease in selectivity influences the performance of histograms.

We show the results of the second experiment in Figure 6. As expected, the histograms show a tremendous improvement in response time in contrast to other views, with the speedup

ranging from 5500 up to 10000 and while having the estimation error of 1.7% and 0.1% for the queries covering 10% and 90% of the data, respectively. Although we do not take in the consideration the time it takes to create a histogram, we expect the results to be similar when adding this number (i.e., no improvement over a log scan). As the histogram is created once over the whole data and can be used to answer all queries over the data, the time to create a histogram can be distributed over all the queries and for a large enough number of iterations, this number will not make a significant difference.

As mentioned before, we tried to find a connection between the number of bins that include the queried range and the speedup. The number of such bins can be estimated by the predicate selectivity. If a query covers 10% of the data, then the histogram uses approximately 10% of its bins to answer the query. So, with a growing number of buckets used by a histogram, we expect the response time to grow. However, we do not witness such behavior in our experiment. This can be explained by the fact that highly optimized histogram compacts the data well and the number of bins used to respond to a query does not make a difference.

D. Experiment 3

The goal of the last experiment was to analyze the performance of the second element of AQP component - the sketches. In order to do this, we calculate the cardinality of a Totalprice attribute from Orders relation, using Log, Row, and Column storage views for the exact results and HyperLogLog sketch to estimate the result. Similar to the previous experiment, Row and Column views contained all data from Orders table, and the sketch was pre-calculated. In this experiment we test how the building of a hash table can be compared to the query with a pre-calculated sketch.

As it can be seen from Figure 7, the sketch provides the speedup of about 1M in comparison to Log, 300k - to Row, and 140k - to Column, while having the estimation error of 1%. Although we do not take the sketch building time into consideration, these values represent accurate behavior of sketches in database applications. An offline-created sketch is used to answer the queries approximately, which means that the response time is technically equal to a time it takes a system to return a pre-calculated estimated value. As the sketch can be quickly updated when new data is inserted, this time has no big influence on the query response time, which again equals the time it takes to return an estimated value.

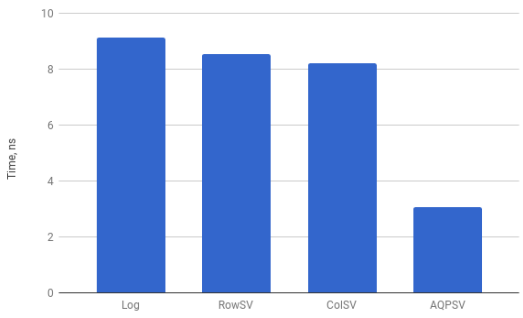

Figure 7. Experiment 3: Average response time for a count distinct query on the Orders table. Comparison with a HLL sketch.

V. Related Work

In this section we briefly discuss several prior works and describe similarities and dissimilarities that they share with Blinktopus.

Logs as a primary structure. How the data is stored defines which data access techniques to implement in order to obtain the best performance possible for a certain database system. Most of modern DBMSs prefer to use a single data storage layout. Although, relying on the initial goal of the system such single-layout systems may acquire an exceptional efficiency, none of these choices may be a universally perfect solution. As a result, the most architecturally different systems have implemented new types of data storage layouts like hybrids PAX, or more basic ones, like logs.

Log is most probably one of the simplest data structures for DBMS. Furthermore, by supporting an appending write and sequential read, logs may not even require additional access methods. Therefore, these systems eventually may show a better performance for write operations. Apache Samza[31], OctopusDB[1] are examples of the systems that utilize logs as their primary storage. In contrast to OctopusDB and Blinktopus, Samza demonstrates a better scalability by replicating these logs on multiple nodes. Moreover, Samza also supports a so-called log compaction mode, in which older messages are garbage-collected[31].

Data representation. Now after query issued, reading from a log is another challenging task because of the log full scan. Thus, representing the data in the different physical layout which will answer different queries more efficiently, can be quite a viable solution.

Here Samza again exploits much researched idea of materialized views by consuming the events in a log and building the cached views[18]. These materialized views resemble somehow SVs in Blinktopus. However, differently from Blinktopus's types of SVs, they have index-based structures such as a full-text search index, a graph index, a key-value store, etc.

Rodent Store is another example of the system that shares the same idea of representing data in the various physical layouts. Rodent Store claims itself as a declarative and adaptive system that provides DBAs a high-level interface to specify the data physical representation by means of storage algebra[23]. In particular, adiministrators declare how to group and order a logical schema into sets of columns, rows and/or arrays. In case of Blinktopus, user is also behind the decisions in terms of data physical representation. In addition, aside from relatively "usual" row- and column-based layouts, user can accelerate retrieving the queries results through AQPSVs.

Adaptive indexing is another approach where, instead of layouts, data can be reorganized, according to queries. One method for adaptive indexing, database cracking shares with

OctopusDB the use of copies for data reorganization and the approach of reorganizing data at query-time[33]. Finally, in a general way, we observe that the closely related work regarding data representation is also related to traditional research in managing materialized views[34].

HTAP with AQP support. Snappy Data has integrated OLTP, OLAP, processing of streaming data along with AQP techniques in a single solution. For AQP part Snappy Data successfully implemented sketches, uniform and stratified samples, namely count-min sketch(CMS) for "heavy hitters" (i.e., top-K queries) and stratified samples for queries with selective WHERE conditions on stratified columns and etc[32]. Currently they extended the AQP functionality by allowing user to specify the number of column sets to approximate the results. Moreover, by means of high-level accuracy contract(HAC) approach user is allowed to define as a single percentage an acceptable for him/her a level of accuracy. Both of these concepts seem though to be a possible directions for future work for Blinktopus.

VI. Conclusions and future work

In this paper we combined the core concepts of OctopusDB and state-of-the-art AQP techniques and introduced a new system coined Blinktopus. Firstly, we compared the performance of several queries over "traditional" storage views supported by the system (i.e, LogSV, RowSV and ColSV). Additionally, in order to achieve interactive response times in answering COUNT and DISTINCT COUNT queries we included the support of AQP techniques in Blinktopus. Thus, by implementing several types of AQP data synopses in Blinktopus, namely histograms and sketches based on HyperLogLog algorithm we proved that OLAP queries can benefit from AQP techniques. Through our implementation we also discovered the following issues in the design of OctopusDB. Using the centralized log as a primary data storage structure without any optimizations (e.g., log replication, garbage collection, etc.) might be a quite prohibitive. Particularly, we also doubt the efficiency of the concurrency control scheme in OctopusDB. Even though it keeps with MCC, it still lacks such concepts as pointers between versions or aforementioned garbage collection. We believe that these ideas can also be possible direction for future work on Blinktopus. Besides, we believe that in the future Blinktopus needs to support transaction model. Moreover, we propose that future work should evaluate the memory footprint of histograms and sketches in our implementation, to extend the understanding of how they compare to basic SVs. Further extensions to our prototype include the SV-lattice, and the evaluation of the potentials from adding AQP over the basic views. Finally, an implementation of more general sample-based data synopses might also facilitate to extend the spectrum of query classes supported by the system.

Acknowledgement

We would like to thank our supervisor Gabriel Campero Durand for his help, and Alekh Jindal, Jens Dittrich and Barzan Mozafari for their awesome ideas that really inspired this project.

References

[1] Dittrich, Jens, and Alekh Jindal. "Towards a One Size Fits All Database Architecture." CIDR. 2011.
[2] Jindal, Alekh. "OctopusDB: flexible and scalable storage management for arbitrary database engines." (2012).
[3] Jindal, Alekh. "The mimicking octopus: Towards a one-size-fits-all database architecture." VLDB PhD Workshop. 2010.
[4] Mozafari, Barzan. "Approximate query engines: Commercial challenges and research opportunities." SIGMOD, 2017.
[5] Mozafari, Barzan, and Ning Niu. "A Handbook for Building an Approximate Query Engine." IEEE Data Eng. Bull. 38, no. 3 (2015): 3-29.
[6] M. Stonebraker and U. Cetintemel. "One Size Fits All": An Idea Whose Time Has Come and Gone. In ICDE, pages 211, 2005.
[7] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL Servers memory-optimized OLTP engine. In SIGMOD, pages 12431254, 2013.
[8] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In CIDR, 2005.
[9] F. Frber, N. May, W. Lehner, P. Groe, I. Mller, H. Rauhe, and J. Dees. The SAP HANA Database An Architecture Overview. IEEE DEBull 35(1):2833, 2012.
[10] A. Kemper and T. Neumann. HyPer A Hybrid OLTP'&'OLAP Main Memory Database System Based on Virtual Memory Snapshots. In ICDE, pages 195206, 2011.
[11] A. Pavlo, J. Arulraj, L. Ma, P. Menon, T. C. Mowry, M. Perron, A. Tomasic, D. V. Aken, Z. Wang, and T. Zhang. Self-Driving Database Management Systems. In CIDR, 2017.
[12] Cormode, Graham, Minos Garofalakis, Peter J. Haas, and Chris Jermaine. "Synopses for massive data: Samples, histograms, wavelets, sketches." Foundations and Trends in Databases 4, no. 13 (2012): 1-294.
[13] Mike Stonebraker, Daniel Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran and Stan Zdonik. "C-Store: A Column-oriented DBMS." VLDB, pages 553-564, 2005.
[14] H-Store. http://hstore.cs.brown.edu/
[15] MemSQL. http://www.memsql.com/
[16] SnappyData. https://www.snappydata.io/
[17] https://www.gartner.com/doc/3179439/predicts–inmemory-computingenabled-hybrid
[18] https://www.confluent.io/blog/turning-the-database-inside-out-with-apache-samza/
[19] B. Babcock and S. Chaudhuri. Towards a robust query optimizer: A principled and practical approach. In SIGMOD, 2005.
[20] Y. E. Ioannidis, "The history of histograms (abridged)," in Proceedings of the International Conference on Very Large Data Bases, pp. 1930, 2003.
[21] Y. E. Ioannidis, V. Poosala, "Histogram-Based Solutions to Diverse Database Estimation Problems." IEEE Data Eng. Bull. 18 (1995) 10-18.
[22] S. Guha and B. Harb. Wavelet synopsis for data streams: minimizing non-euclidean error. In KDD, 2005.
[23] Philippe Cudr-Mauroux, Eugene Wu, and Samuel Madden. "The Case for RodentStore: An Adaptive, Declarative Storage System." In CIDR. www.cidrdb.org, 2009.
[24] Poosala V., Ioannidis Y., Haas P., Shekita E.: Improved Histograms for Selectivity Estimation of Range Predicates. SIGMOD Conf. (1996) 294-305
[25] Gibbons P., Matias Y., Poosala V.: Fast Incremental Maintenance of Approximate Histograms. VLDB Conf. (1997) 466-475
[26] https://yahooeng.tumblr.com/post/135390948446/data-sketches
[27] P. Flajolet, Èric Fusy, O. Gandouet, and et al., Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm," in IN AOFA '07: PROCEEDINGS OF THE 2007 INTERNATIONAL CONFERENCE ON ANALYSIS OF ALGORITHMS, 2007.
[28] Lane, D.M.: Online statistics education: an interactive multimedia course of study (2015). http://onlinestatbook.com/2/graphing distributions/histograms.html. Accessed 03 Dec 2015
[29] Doane, D.P.: Aesthetic frequency classifications. Am. Stat. 30(4), 181183 (1976)
[30] Hyndman, R.J.: The problem with sturges rule for constructing histograms. Monash University (1995)

[31] Kleppmann M., Kreps J.: Kafka, Samza and the Unix philosophy of distributed data. IEEE Data Engineering Bulletin 38(4):414, December 2015.
[32] Ramnarayan, Jags, Barzan Mozafari, Sumedh Wale, Sudhir Menon, Neeraj Kumar, Hemant Bhanawat, Soubhik Chakraborty, Yogesh Mahajan, Rishitesh Mishra and Kishor Bachhav. "SnappyData: Streaming, Transactions, and Interactive Analytics in a Unified Engine." (2016).
[33] S. Idreos, M. L. Kersten, and S. Manegold, "Database Cracking," in Proceedings of the 3rd International Conference on Innovative Data Systems Research (CIDR), Asilomar, California, 2007, pp. 68-78.
[34] Ashish Gupta and Iderpal Singh Mumick (Eds.). 1999. Materialized Views: Techniques, Implementations, and Applications. MIT Press, Cambridge, MA, USA.