FlashView: An Interactive Visual Explorer for Raw Data

Zhifei Pang, Sai Wu, Gang Chen, Ke Chen, Lidan Shou
College of Computer Science and Technology, Zhejiang University
{zhifeipang, wusai, cg, chenk, should}@zju.edu.cn

ABSTRACT

New data has been generated in an unexpected high speed. To get insight of those data, data analysts will perform a thorough study using state-of-the-art big data analytical tools. Before the analysis starts, a preprocessing is conducted, where data analyst tends to issue a few ad-hoc queries on a new dataset to explore and gain a better understanding. However, it is costly to perform such ad-hoc queries on large scale data using traditional data management systems, e.g., DBMS, because data loading and indexing are very expensive. In this demo, we propose a novel visual data explorer system, FlashView, which omits the loading process by directly querying raw data. FlashView applies approximate query processing technique to achieve real-time query results. It builds both in-memory index and disk index to facilitate the data scanning. It also supports tracking and updating multiple queries concurrently. Note that FlashView is not designed as a replacement of full-fledged DBMS. Instead, it tries to help the analysts quickly understand the characteristics of data, so he/she can selectively load data into the DBMS to do more sophisticated analysis.

1. INTRODUCTION

A big data analyst should have a deep understanding and experience about business intelligence and real-time analytics. Before performing any analysis, he/she will explore and examine a dataset first. Such data exploration process retrieves a few data samples from the dataset and visualizes some common aggregation results to help data analysts understand the data characteristics.

To explore a new dataset in DBMS, we need to load data first and then build a few indexes. However, both data loading and indexing are expensive processes, because data are required to be formatted into predefined structures. Moreover, after the exploration process, data analysts may be only interested in a small subset, on which he/she will do more sophisticated analysis. Hence, loading the full dataset is not necessary and costly. What a data analyst requires is an interactive system that can start the data exploration without start cost.

To address the problem, NoDB system [2][7][3] directly manipulates raw data and completely avoids data loading. Similarly, analytic systems built on top of Hadoop, such as Hive [8], also works on raw data. One drawback of these systems is their suboptimal performance. Without proper index, we must scan the whole dataset to process a query. And since we are handling the raw data (e.g., csv files), we need to parse every tuple on the fly. To improve the performance, NoDB system gradually builds indexes. Unfortunately, such index building process may take quite a long time to complete. HadoopDB [1], on the other hand, relies on the DBMS to index and prune data. This strategy significantly increases data loading cost and is thus not feasible for data exploration process.

So we face the dilemma of whether to use the data loading/indexing technique or suffering the slow query processing performance. However, neither selection is good for data exploration. Moreover, even some data analysts are familiar with the SQL, most of them prefer the visualization results which are more intuitive. Currently, there are a set of interactive visualization tools such as Tableau[1] and QlikView[2]. They provide easy-to-use GUIs for users to explore data. But most of them rely on existing data management systems to do the "real-time" analysis. They neither remove the data loading process, nor support real-time interactive response. For data that cannot be held in memory, users suffer a long waiting time.

Motivated by the requirements of our collaborated data analyst, we design a new data exploration system, FlashView, which has the following three distinguished features.

- FlashView does not need to load data, as it directly manipulates raw data files. The only requirement is to tell FlashView about the metadata of the data. FlashView adopts an adaptive parsing process when user explores the data gradually.

- FlashView provides a real-time response for data aggregation queries by leveraging the approximate query processing techniques [4][9]. Users can issue a series of correlated aggregation queries to reveal insight of the data. FlashView tracks all queries concurrently and updates the results continuously.

[1]https://www.tableau.com
[2]http://www.qlik.com/

Proceedings of the VLDB Endowment, Vol. 10, No. 12

- FlashView guides users to explore data via a simple but intuitive GUI. Data are partitioned and organized as a set of hierarchical subsets. Users can browse aggregation results on all subsets and swiftly create new subsets by further partitioning the data.

Note that as a data exploration tool, FlashView is not designed as a full-fledged data processing systems. Instead, it swiftly shows a rough view of the dataset, so data analyst can decide which part of the data should be loaded into a more complex data processing system (e.g., DBMS) for a complete study. Based on the results from FlashView, data analysts can also identify the interesting queries and get a refined result from other systems.

2. TECHNICAL BACKGROUND

2.1 Approximate Query Processing

In FlashView, data can be queried without any preloading and indexing. To achieve such goal, FlashView adopts the AQP technique. More specifically, we use a similar query processing approach as the online aggregation, which was firstly introduced by Hellerstein et. al. [5]. The intuition is to apply statistical analysis to provide an approximate result which is gradually refined when more random samples are retrieved. The quality of the result is bounded by estimated error bound and confidence.

In our case, analysts want to take a quick explore on a huge raw data and expect to find some interesting facts which they can dig further. During this process, analysts may issues multiple queries in a short time and want to get the results instantly, even with compromised accuracy. Therefore, the design philosophy of FlashView is completely consitent with the online aggregation.

In FlashView, the accuracy is depicted by an error bound ε and confidence p, which will be optimized gradually by our AQP model through fast random sampling from raw data. In an extreme case, when all raw data have been scanned, FlashView will return a precise query result, but this will be very inefficient and is not what FlashView is designed for.

To obtain a precise estimation, we choose Hoeffding Inequalition over the Central Limit Theory as our background statistical model.

Let variable T represents a table which contains m tuples, denoted as t_i, so we have $t_1, t_2, ..., t_m \in T$. Suppose we have a query as follows.

SELECT AVG(expression(t_i)) FROM T

The *expression* denotes arithmetic operations on t_i, and let $v(i)$ be its result. Let S denote the samples randomly chosen from T. If $|S| = n$ and $|T| = m$, we can get $P(s_i \in S) = \frac{1}{m}, 1 \leq i \leq n, \forall s_i \in T$. So we can get an approximate aggregation results for the given samples.

$$\bar{Y} = (\frac{1}{n}\sum_{i=1}^{n} v(S_i))$$

Also, the accurate results, denoted as A, can be estimated as

$$A = (\frac{1}{m}\sum_{i=1}^{m} v(i))$$

Assume a and b are the lower and upper bounds for $v(i)$ $(a \leq v(i) \leq b)$.

Generally, the bounds can be roughly estimated atomically according current samples. According to Hoeffding Inequalition, we can get

$$P\{|\bar{Y} - A| < \varepsilon\} > 1 - 2e^{\frac{-2n\varepsilon^2}{(b-a)^2}}$$

We use p to represent the confidence of estimation, namely $P\{|\bar{Y} - A| < \varepsilon\} = p$. We get ε from equation 1.

$$\varepsilon = (b-a)(\frac{1}{2n}\ln(\frac{2}{1-p}))^{\frac{1}{2}} \quad (1)$$

Now, we can guarantee the probability of the accurate results falling in the interval $[\bar{Y} - \varepsilon, \bar{Y} + \varepsilon]$ is equal or larger than p. Then, ε and p will keep updating in turn once more samples are obtained.

Compared to the conventional AQP system, FlashView applies two optimization approaches. First, one major problem of the approximate query processing is the lack of enough random samples for small groups (due to *group by* operation or selection predicate). In FlashView, we build both in-memory and disk index for processed data to speed up sampling process for a specific group. Second, data exploration is a continuous process where a series correlated queries are issued. Therefore, FlashView keeps track of multiple queries and shares query processing process amongst them. We introduce the details in the next section.

2.2 Interactive Visualization

BI tools, such as Tableau and QlikView, provide interactive visualization tools for users to gain a better understanding of their data. Most existing visualization systems adopt the typical load-and-query solution. For example, Tableau can be connected to a DBMS, and data should be first imported into the DBMS before processing any query. Alternatively, Tableau can just to load all data into its own in-memory processing engine to execute queries. Both strategies suffers the long waiting time of data loading. Disk-based processing is very slow for big data even with parallelism, whereas memory-based processing is limited by the memory size.

On the contrary, FlashView has an user friendly graphic interface as Tableau and QlikView. It is build on top of its AQP engine and provides real-time response. Users can query raw data file and view dynamic visualization of results by interacting with FlashView GUI. They can also use the GUI to partition or filter the data and issue new exploration requests which are transformed into a set of queries for the AQP engine. We allow users to add a fixed number of queries to his/her watch list. The status, charts and parameters related to queries in watch list are updated continuously. Users can navigate and switch between these query visualization results. Finally, after a such exploration, FlashView provides tools to import a portion of data (normally related to a query in watch list) into other processing systems, e.g., DBMS and Hadoop, to do a more sophisticated analysis.

3. OVERVIEW OF FLASHVIEW

3.1 Architecture of FlashView

In this section, we present the architecture and design of FlashView. The front-end and back-end of FlashView are

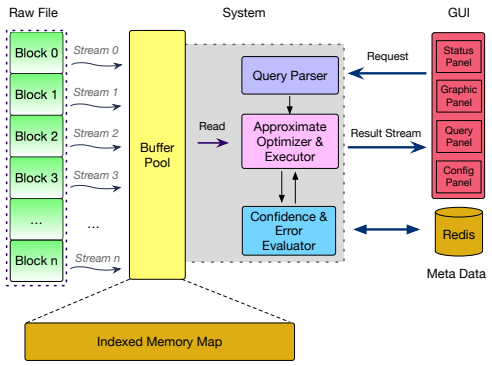

Figure 1: FlashView Architecture

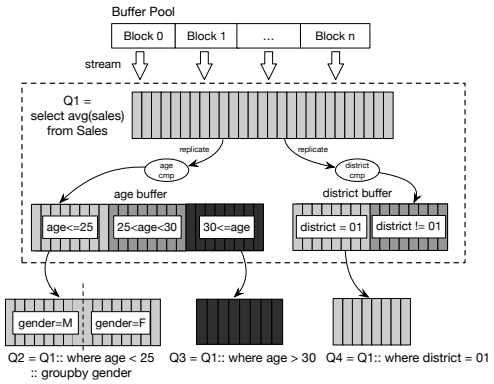

Figure 2: Data Flow of SQL Tree

implemented in JavaScript and Java/Scala respectively. Redis[3] is used to maintain metadata of FlashView. FlashView starts multiple threads to read samples from some random blocks. Samples are maintained in a memory buffer where a scrambling process is applied to permute sample sequence. If memory is full, we apply memory-map approach to let the operating system decide which data should be shuffled to disk. As shown in Figure 1, FlashView consists of four main modules - 1) visualization, 2) query engine, 3) memory data management and 4) metadata storage.

Before data exploration starts, visualization module asks a user to select his/her data file and configure necessary schema properties. Then, the user can start the interactive visual data exploration. Visualization module formalizes interactions between GUI and users via a XML-based language FvQL, similar as the one used in Tableau (VizQL).

The back-end AQP engine accepts a FvQL query and parses it as a data exploration query q. A typical data exploration query consists of a filter set S_F, an arithmetic operators set S_O and a group set S_G. Note that one query can ask multiple aggregation results (e.g., both average value and variance). The AQP engine first checks whether q is derived from a previous query. If so, q will join the specific data flow to share samples. Otherwise, a new data flow is started to retrieve random samples. AQP updates all queries in watch list when a batch of samples are obtained. Users can terminate a query by removing it from the list or set an upper bound for the error rate and confidence.

[3]https://redis.io/

In a data exploration process, new queries are normally derived from previous queries. For instance, an analyst observes a low average income for those born in a specific area. He/she may further issue a query via our GUI to check the average income of those people grouped by ages in that area. For the new query, fewer samples are available due to more selection filters. To address the problem, we maintain processed samples in memory and build a hierarchical index for them. In particular, queries are structured as trees based on their derivation relationship. If samples cannot be maintained in memory, we apply the memory-map technique to flush them back into disk adaptively. For a new query, we first retrieve correlated samples from the index which can immediately generate a coarse result for the query. Then, it is inserted into the watch list for periodical update.

Finally, metadata of all explored data are maintained in Redis which will be used by our AQP engine and index module.

3.2 Indexing and Query Processing

During data exploration process, users can manipulate data via FlashView's GUI. The interactions are translated into data cube operations such as drilling down and rolling up. Each operation is considered as a new query derived from the old one. To avoid a cold start, we maintain used samples in memory buffer and process the new query using those samples. We also build an index search tree as queries are being processed. The idea is borrowed from database cracking techniques [6].

Due to space limitation, we use the example in Figure 2 to illustrate our idea. Suppose there are four running queries in the system on a sales dataset, Q1, Q2, Q3 and Q4. The root node represents Q1, which is the first query when the exploration starts. Q2, Q3 and Q4, as children of Q1, are the drilling down queries derived from Q1. We use symbol :: to denote the filter or operator conjunction.

Q1 reads samples from the buffer pool and maintains an index for samples that it has consumed. After checking the results of Q1, the analyst is interested in people under 25 years old and issues query Q2. Q1 scans samples via its index and creates an index node *age buffer* which is cracked by the indicator $age = 25$. The indicator divides the random buffer into two parts, one for data which has $age \leq 25$, the other for $age > 25$. Furthermore, we create an index node for Q2 by assembling tuples satisfying $age < 25$. The index node is organized into two groups for *male* and *female*.

Like Q2, Q3 has *where* $age > 30$ which makes Q1 has to crack the *age buffer* into three segments. Then Q3 just fetches samples like Q2.

Q4 is a special case, because there is no existing buffer with *district* indicator. So Q1 will create a *district buffer* to process queries with *district* filter. Both *age buffer* and *district buffer* can be considered as a replica of Q1's index. But they reorganize their buffer to create new indexes for consecutive queries.

Note that, all the segments in parent nodes can be combined to deal with more complicated queries. When a query contains both *where* $age < 25$ and *where* $district = 1$, Q1 will combine the two qualified segments from *age buffer* and *district buffer* together and data in both segments will be taken as possible valid samples for the new query. Since that sub-queries sustain samples from parent query node, we can achieve one reading for multiple updating which im-

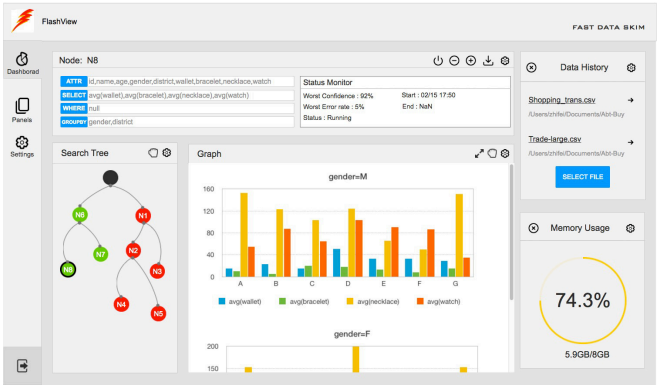

Figure 3: GUI of FlashView

proves the sampling performance greatly. However, samples from parent nodes may not be sufficient, because available samples decrease greatly as increase of search tree height. Therefore, during the runtime, new samples retrieved from raw data will flow from Q1 to Q2, Q3 and Q4 based on the search tree.

For rolling up query requests, instead of adding a child to the current node, a new node will be built and inserted into between the current node and its former parent. The rest operations are just like drill down process mentioned above.

If a query is suspended by the user and there is no child query, we discard its index. Otherwise, we keep maintaining its index until all its child queries have been completed.

4. DEMONSTRATION

We will demonstrate FlashView with two real datasets. The participants attending FlashView demonstration can start a data exploration through the web interface in Figure 3. We show three main modules:

Load: By clicking the select file button, the user specifies the file path. Schema configuration should be provided if the file is first seen. Data type will be automatically inferred before the processing starts.

Query/Navigate: The black node represents the root node. The green nodes and red ones stand for running queries and suspended queries respectively. Currently selected node is emphasized with a black circle, and its details are shown in the node panel. With the node panel, the user can start, terminate and dump the query by clicking different buttons. Users can navigate current running queries through the search tree. Visualization diagrams will be changed accordingly. Users can also create new queries by extending any existing running queries.

Visualization: With the graph panel, the user can see the dynamic results from the current running query. Graph panel also supports zoom operations to show diagrams in full screen mode. Currently, FlashView just supports 2-D graph plotting. When there are multiple group-by, the combinations will shown as tiles denoting results calculated in each group.

5. CONCLUSION

In this demo, we present FlashView, a data exploration system which supports on-the-fly data exploration on large raw data. FlashView avoids the long data loading process and supports real-time response for analytic queries by adopting the approximate query processing technique. It provides an intuitive GUI for users to understand the characteristics of the data.

6. ACKNOWLEDGE

This research is supported by National Natural Science Foundation of China (Grant No. 61661146001) and National Basic Research Program of China (No.2015CB352402).

7. REFERENCES

[1] A. Abouzied, K. Bajda-Pawlikowski, J. Huang, D. J. Abadi, and A. Silberschatz. Hadoopdb in action: building real world applications. In *SIGMOD*, pages 1111–1114, 2010.

[2] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. Nodb in action: Adaptive query processing on raw data. *PVLDB*, 5(12):1942–1945, 2012.

[3] I. Alagiannis, R. Borovica-Gajic, M. Branco, S. Idreos, and A. Ailamaki. Nodb: efficient query execution on raw data files. *Commun. ACM*, 58(12):112–121, 2015.

[4] P. J. Haas and J. M. Hellerstein. Online query processing. In *SIGMOD*, page 623, 2001.

[5] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *SIGMOD*, pages 171–182, 1997.

[6] S. Idreos, M. L. Kersten, and S. Manegold. Database cracking. In *CIDR*, pages 68–78, 2007.

[7] M. Karpathiotakis, M. Branco, I. Alagiannis, and A. Ailamaki. Adaptive query processing on RAW data. *PVLDB*, 7(12):1119–1130, 2014.

[8] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy. Hive - a petabyte scale data warehouse using hadoop. In *ICDE*, pages 996–1005, 2010.

[9] S. Wu, B. C. Ooi, and K. Tan. Continuous sampling for online aggregation over multiple queries. In *SIGMOD*, pages 651–662, 2010.