Demonstration of BitGourmet: Data Analysis via Deterministic Approximation

Saehan Jo
Cornell University
sj683@cornell.edu

Immanuel Trummer
Cornell University
itrummer@cornell.edu

ABSTRACT

We demonstrate BitGourmet, a novel data analysis system that supports deterministic approximate query processing (DAQ). The system executes aggregation queries and produces deterministic bounds that are guaranteed to contain the true value. The system allows users to set a precision constraint on query results. Given a user-defined target precision, we operate on a carefully selected data subset to satisfy the precision constraint. More precisely, we divide each column vertically, bit-by-bit. Our specialized query processing engine evaluates queries on subsets of these bit vectors. This involves a scenario-specific query optimizer which relies on quality and cost models to decide the optimal bit selection and execution plan. In our demonstration, we show that DAQ realizes an interesting trade-off between result quality and execution time, making data analysis more interactive. We also offer manual control over the query plan, i.e., the bit selection and the execution plan, so that users can gain more insights into our system and DAQ in general.

ACM Reference Format:
Saehan Jo and Immanuel Trummer. 2020. Demonstration of BitGourmet: Data Analysis via Deterministic Approximation. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3318464.3384709

1 INTRODUCTION

Approximate Query Processing (AQP) has received an abundant amount of attention due to the growing need to process very large data sets efficiently [1–5, 8–13, 15–17]. The goal of AQP is to produce *approximate results* for aggregation queries and, by doing so, trade off little precision loss for faster response time. However, despite the ongoing research work in AQP, its adoption in industry has been limited [3]. Most prior work uses sampling to produce confidence bounds, containing the true value only with a certain probability. This uncertainty can be unsatisfactory for users in real-world applications. Another problem associated with sampling-based methods is their inherent limitations in supporting aggregation functions that are sensitive to outliers (e.g., maxima and minima).

A recent work [14] proposes Deterministic Approximate Query Processing (DAQ) as an alternative to the probabilistic approach. DAQ produces deterministic bounds, as opposed to confidence bounds, that are guaranteed to contain the exact value. With its stronger guarantees on the error bounds, DAQ can be a more reliable and valid choice in many real-world scenarios. In this presentation, we provide an interactive demonstration of BitGourmet [7], a novel data analysis system that supports deterministic approximation for complex queries. The aforementioned paper focuses on simple queries involving single tables, single aggregates, single predicates, and no grouping. On the contrary, our system supports complex queries, as they appear in standard benchmarks, involving grouping, multiple aggregates, multiple predicates, and multiple tables.

Given an aggregation query and a user-defined error constraint, BitGourmet produces deterministic bounds that meet the target precision (and contain the true value). At the heart of BitGourmet is a specialized query optimizer that decides which subset of data to read and process. Using a cost-based error model, we carefully select data subsets that would produce approximate results satisfying the target precision. However, unlike sampling-based methods that consider row subsets, we divide each column vertically at bit-level granularity. This allows us to access partial information for all rows instead of complete information for a few rows, an essential feature for our system to produce deterministic bounds. The second goal of our query optimizer is to select execution plans on the bit columns that minimize processing cost. BitGourmet uses scenario-specific operators that operate on bit vectors. Also, we support multiple representations of intermediate results (e.g. decompressed and compressed bit

SIGMOD'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3384709

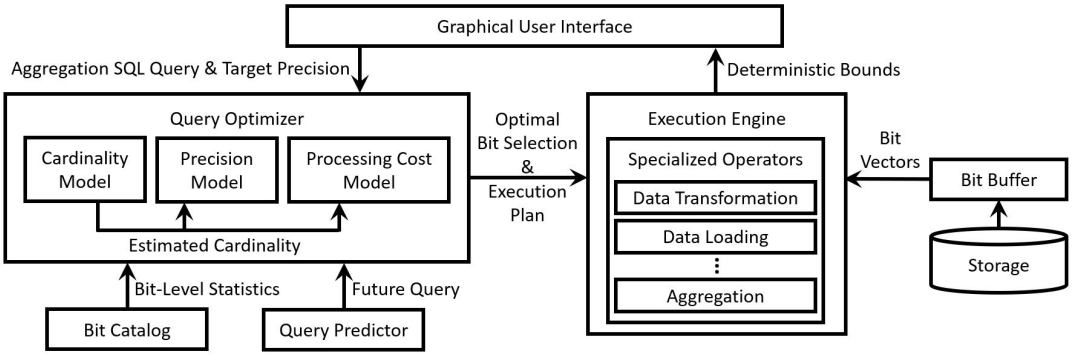

Figure 1: Overview of the BitGourmet system.

vectors or the standard row-wise representation) with specialized operator implementations. Our system may choose to transform the intermediate result representation during execution if that is likely to accelerate future operations. Similar to the error model, we rely on a cost model to estimate processing times of different execution plans. Based on both models, the BitGourmet optimizer chooses an optimal processing plan. The optimal plan minimizes processing costs while producing results that meet the user-defined error constraints. By reading and processing less bits, BitGourmet trades little precision loss for significant performance gains as shown in our experiments [7].

Our system comes with a graphical user interface that visualizes the deterministic bounds for end-users. In our demonstration, we also provide additional information regarding the internal decision making process. That is, we show alternative query plans for DAQ (i.e., bit selections and execution plans) and their estimated errors and processing costs predicted by our query optimizer. Furthermore, users will have the freedom to manually choose the bit subset to operate on and see how it affects the precision of generated bounds.

BitGourmet realizes an AQP engine that produces deterministic bounds based on bit subsets and, in exchange, delivers a performance improvement over exact processing. In Section 2, we give an overview of the BitGourmet system. In Section 3, we describe our plan for the demonstration that allows users to explore various aspects of our system.

2 SYSTEM OVERVIEW

Figure 1 shows an overview of our system. We process SQL queries with aggregates. As output, we produce deterministic bounds on each aggregate (i.e., lower and upper bounds that are guaranteed to contain the true value). We trade precision against processing time by reading and processing only carefully selected bits from database columns. We store our database as a set of bit vectors. This allows us to access specific bits of specific columns separately. In the following subsections, we first explain the bit-wise storage scheme and then the execution engine of BitGourmet. Next, we present our cost-based query optimizer. The graphical user interface is described along with our demonstration plan in Section 3.

Table 1: Predicate evaluation with ternary states.

C	Possible Values	C > 5
1?0?	8, 9, 12, 13	✓
0?1?	2, 3, 6, 7	?
1?1?	10, 11, 14, 15	✓
0?0?	0, 1, 4, 5	×

2.1 Data Representation

BitGourmet uses a bit-wise scheme to store data and to represent intermediate results. It allows us to access each bit vector individually and to reason about the effect of a specific bit vector on the query result.

Raw Data Storage. BitGourmet divides and stores raw data vertically at bit-level to provide individual access to a specific bit vector at need. This enables our system to load a subset of bit vectors, even from the same column, to the memory buffer with minimum overhead (compared to row-wise or column-wise storage). As a consequence, we get partial information about the attribute values for each row. For instance, in Table 1, we read the first and the third bit vectors for an integer column C (and do not read the second and the fourth bits). Then, by assuming either zero or one for the unknown bits, we can determine the set of possible tuple values from the incomplete information. The table shows how to evaluate an inequality predicate based on possible values (the example will be explained later in more detail).

Intermediate Result Representation. A particularity of our bit-wise processing scheme is that we assign each

tuple in intermediate results to a ternary state: 1) it is *certain* that the tuple satisfies all applicable predicates, 2) it is *possible* that the tuple satisfies all predicates, and 3) it is *certain* that the tuple *does not* satisfy some predicate. The first and the third cases are the tuple states that we usually see in exact processing engines. For BitGourmet, it we must also consider the case that we do not have enough information to safely include or exclude a tuple. Here, we do not consider null values, which we treat separately. We elaborate more using the example in Table 1. The inequality predicate "C > 5" evaluates to a ternary state vector where each tuple state is determined by the set of possible values.

2.2 Execution Engine

BitGourmet features specialized operators that process bit vectors (and rows if needed) and produce deterministic bounds. Given the optimal bit selection and execution plan, our execution engine loads the corresponding bit vectors from disk, transforms data representations as requested, and aggregates data approximately but in a deterministic manner.

Data Transformation. We provide operators that transforms data from one representation to another. Currently, we support three different representations for intermediate results: compressed or decompressed bit vectors and the standard columnar representation. These operators naturally entail a processing cost, and thus, they are only inserted into an execution plan if the estimated benefit we gain from the remaining operations outweighs their overhead.

Data Loading. BitGourmet can load raw data from disk at bit granularity. We first check whether the required bit vector resides in the memory buffer. If not, we load the bit vector from disk and evict other bit vectors as needed. An advantage of storing bit subsets in memory (instead of storing the entire column data) is that we can partition the available memory space over more columns, keeping less (and the most essential) data for each column. This contributes to a higher cache hit ratio across queries, compared to exact processing, especially when the hot set for DAQ fits into main memory.

Aggregation. BitGourmet's aggregation operators have multiple implementations, tailored to different input representations. Bit-wise aggregation directly works with bit vectors. For instance, consider the sum aggregation. We first count the number of ones in each bit vector and multiply each count with the corresponding weight (e.g. for integer columns, 2^i where i is the bit position of a bit vector). Row-wise aggregations are similar in spirit to the standard implementation in exact processing engines. To produce lower and upper bounds, we consider the range of all possible values in the aggregation column for each tuple.

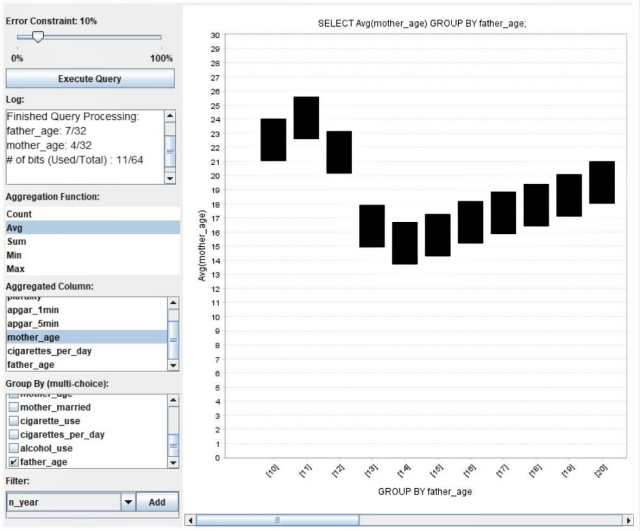

Figure 2: Screenshot from the BitGourmet interface.

2.3 Query Optimizer

BitGourmet's query optimizer chooses the optimal bit selection and execution plan for a given aggregation query and a user-defined target precision. It relies on precision and processing cost models to estimate the quality of a query plan (i.e., its expected precision and execution cost). The estimates computed by these two models are internally based on cardinality estimates provided by a cardinality model. To enable this kind of scenario-specific optimization for bit selections, we utilize bit-level statistics as well as the standard data catalog. A query predictor supports the optimizer in adjusting the execution plan (including plans for proactive buffer management), whenever possible, to maximize the expected overall performance for current and future queries.

Error Model. For a given bit selection, the error model estimates the expected result quality for the query. Its core functionality is to reason about how the uncertainty associated with the tuple values affects the lower and upper bounds (or their relative distance) in a principled way. For each aggregation function supported by BitGourmet (i.e., sum, average, count, minimum, and maximum), we derive a closed-form formula that computes an estimated error for any bit selection based on bit-level statistics.

Processing Cost Model. The processing cost depends on both, the bit selection and the execution plan. We model the processing cost of a BitGourmet operator as sum over the atomic costs of bit-level operations. The costs of atomic operations on bit vectors (e.g., *AND*, *OR*, *NOT*, *GetBitAt*, and *NrOnes*) are calculated based on the type of operation, the input representation (e.g., decompressed versus compressed), and the sparsity of bit vectors.

Query Predictor. The query predictor relies on a log of past queries to anticipate upcoming queries. Based on these predictions, the query optimizer might prefer an execution plan that (while taking slightly longer than the locally optimal plan for the current query) generates bit vectors that can be reused when processing likely future queries.

3 DEMONSTRATION

We demonstrate the BitGourmet system and show how it facilitates data analysis on real-world data sets. As a baseline, we provide the option to change the underlying database to an exact processing system, MonetDB [6]. Also, users can manually select which bit selection to use and examine its effect on result quality and processing efficiency.

3.1 Data Set

We use two real-world data sets during our demonstration. One data set is about the number of crimes in London per geographic region and per crime type[1]. The other data set describes all childbirths in the United States between years 1969 and 2008[2]. It provides information about mothers, fathers, and babies, for instance, their age, race, residence state, smoking habit, drinking habit, and other health-related statistics. An experiment demonstrates that we get an average speed up of 28.5× for analytical workloads on this data set, compared to an exact processing system [7] (we refer to our previous paper for the experimental setup).

3.2 Graphical User Interface

Figure 2 presents a screenshot of the BitGourmet interface. The plot on the right shows the deterministic bounds for the aggregation query "SELECT AVG(MOTHER_AGE) GROUP BY FATHER_AGE" with error constraint as 10%. Users can choose the error constraint from 0% to 100% (a lower value leads to a higher result quality) using a slider in the top-left corner. On the left, users can specify the aggregation query they want to execute by selecting the aggregation function, aggregated column, group by columns, and predicates. When users press the "Execute Query" button, the system processes the query and presents the approximate result. On the right side of the figure, each aggregate is shown by a bar which indicates the lower and upper bounds. Note that these bounds are deterministic, and thus, the true value is always within the two bounds. The log displayed on the left provides additional information regarding the current query execution.

3.3 System Performance Analysis

We provide users with access to the internals of BitGourmet. The query optimization process is exposed by showing users possible bit selections and their estimated errors. Similarly, our system also displays different execution plans and their estimated processing costs. For interactive demonstration, users will be able to choose for themselves how many bit vectors to read per column and from which bit positions. With this control, users can explore how the bit selection affects both, the result quality and the execution time.

[1] https://www.kaggle.com/LondonDataStore/london-crime
[2] https://www.kaggle.com/bigquery/samples

REFERENCES

[1] Swarup Acharya, Phillip B. Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. 1999. The Aqua Approximate Query Answering System. In *SIGMOD*. 574–576.
[2] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. 2013. BlinkDB: queries with bounded errors and bounded response times on very large data. In *EuroSys*. 29–42.
[3] Surajit Chaudhuri, Bolin Ding, and Srikanth Kandula. 2017. Approximate Query Processing: No Silver Bullet. In *SIGMOD*. 511–519.
[4] Alfredo Cuzzocrea. 2005. Providing Probabilistically-bounded Approximate Answers to Non-holistic Aggregate Range Queries in OLAP. In *DOLAP*. 97–106.
[5] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. 1997. Online Aggregation. In *SIGMOD*. 171–182.
[6] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. 2012. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Eng. Bull.* 35, 1 (2012), 40–45.
[7] Saehan Jo and Immanuel Trummer. 2020. BitGourmet: Deterministic Approximation via Optimized Bit Selection. In *CIDR*.
[8] Shantanu Joshi and Christopher Jermaine. 2008. Materialized Sample Views for Database Approximation. *TKDE* 20, 3 (2008), 337–351.
[9] Srikanth Kandula, Anil Shanbhag, Aleksandar Vitorovic, Matthaios Olma, Robert Grandl, Surajit Chaudhuri, and Bolin Ding. 2016. Quickr: Lazily Approximating Complex AdHoc Queries in BigData Clusters. In *SIGMOD*. 631–646.
[10] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. 2016. Wander Join: Online Aggregation via Random Walks. In *SIGMOD*. 615–629.
[11] Supriya Nirkhiwale, Alin Dobra, and Christopher M. Jermaine. 2013. A Sampling Algebra for Aggregate Estimation. *PVLDB* 6, 14 (2013), 1798–1809.
[12] Frank Olken and Doron Rotem. 1995. Random sampling from databases: a survey. *Statistics and Computing* 5, 1 (1995), 25–42.
[13] Jinglin Peng, Dongxiang Zhang, Jiannan Wang, and Jian Pei. 2018. AQP++: Connecting Approximate Query Processing With Aggregate Precomputation for Interactive Analytics. In *SIGMOD*. 1477–1492.
[14] Navneet Potti and Jignesh M. Patel. 2015. DAQ: A New Paradigm for Approximate Query Processing. *PVLDB* 8, 9 (2015), 898–909.
[15] Chengjie Qin and Florin Rusu. 2014. PF-OLA: A High-performance Framework for Parallel Online Aggregation. *Distrib. Parallel Databases* 32, 3 (2014), 337–375.
[16] Amit Rudra, Raj P. Gopalan, and Narasimaha Achuthan. 2012. An Efficient Sampling Scheme for Approximate Processing of Decision Support Queries. In *ICEIS*. 16–26.
[17] Barzan Mozafari Yongjoo Park, Ahmad Shahab Tajik, Michael Cafarella. 2017. Database Learning: Toward a Database that Becomes Smarter Every Time. In *SIGMOD*. 587–602.