BitGourmet: Deterministic Approximation via Optimized Bit Selection

Saehan Jo
Cornell University
sj683@cornell.edu

Immanuel Trummer
Cornell University
itrummer@cornell.edu

ABSTRACT

The goal of deterministic approximation is to produce guaranteed bounds for aggregates in SQL queries (i.e., the exact value is guaranteed to be contained within those bounds). We present our ongoing work on BitGourmet, a novel system for deterministic approximation, and first experimental results. BitGourmet reduces processing time, compared to standard processing, by considering only a subset of bit positions in every column. It stores the entire database as a collection of bit vectors. Given user-specified constraints on approximation precision, BitGourmet selects an optimal subset of bit vectors to generate a result of the desired precision with minimal processing overheads. BitGourmet features a specialized processing engine with scenario-specific operators. It uses a multi-objective, cost-based optimizer that employs cardinality, cost, and error models based on bit-level data statistics. Furthermore, it uses a proactive buffer management strategy based on query predictions to fill its buffer with bit vectors that are likely to be relevant for future queries. We provide first experimental results, demonstrating significant speedups over state-of-the-art exact processing engines and increased result precision, compared to sampling-based approximation engines.

1. INTRODUCTION

Approximate query processing (AQP) engines typically use sampling to generate confidence bounds for query aggregates. This is difficult for certain aggregation functions (e.g., the minimum and maximum), and confidence bounds have been shown to be generally difficult to interpret for users [18]. Also, confidence bounds are often calculated based on simplifying assumptions about input data, which do not always hold in practice. Those drawbacks have recently motivated research on deterministic approximate query processing (DAQ) [18]. Here, the goal is to produce bounds for query aggregates that are formally guaranteed to contain exact values.

We present BitGourmet, a novel system for deterministic approximation of SQL queries. Considering row subsets would prevent us from producing guaranteed bounds (since, e.g., for minima or maxima, one single row can change the aggregation result arbitrarily). Instead, we divide the columns of the input database "vertically" by considering bit vectors representing a subset of their bit positions. While this principle has been proposed before for single operators [18], BitGourmet is the first system to realize this principle in a full-blown approximate processing engine that is able to process complex queries under user-defined precision constraints.

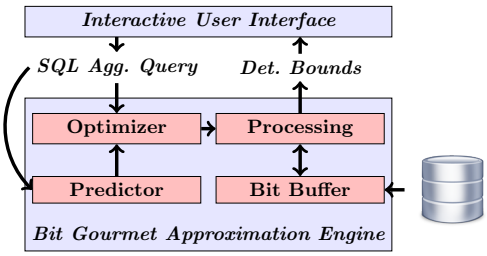

Figure 1: Overview of BitGourmet system.

Figure 1 shows the most important components of BitGourmet and the data flow between them. BitGourmet allows users to specify queries either via a text console or via a graphical user interface. BitGourmet is configured via precision constraints that define the maximum relative distance between upper and lower bounds on aggregate values. Given an incoming query and precision constraints, the BitGourmet optimizer determines an optimal processing plan. Plans specify a subset of the data to process (defined at the granularity of bit positions) as well as the sequence of operations. The cost-based BitGourmet optimizer uses scenario-specific models to predict error, cost, and intermediate result cardinality based on bit-level statistics about the input data. It aims at minimizing estimated processing overheads while meeting precision constraints.

The processing engine supports different representations for bit vectors that are generated during processing and may choose to dynamically convert between different representations (to minimize processing overheads). It features scenario-specific operators that process relations with uncertain rows (i.e., it is unclear, based on the subset of bit positions considered, whether rows satisfy all applicable predicates). To maximize the chances that disk accesses can be avoided, BitGourmet uses a buffer pool of bit vectors that is filled proactively. Based on the history of queries in the current analysis session, BitGourmet tries to predict follow-up queries. Based on those predictions, it generates bit vectors that would be helpful in processing predicted queries

CIDR'20 January 2020, Amsterdam, The Netherlands

and stores them in the buffer pool. The results of processing are deterministic (lower and upper) bounds for query aggregates.

In the following, we give an overview of the BitGourmet system, show first experimental results, discuss our ongoing research and relations to prior work.

2. SYSTEM OVERVIEW

We give an overview over different aspects of the BitGourmet system in the following.

2.1 User Interfaces

BitGourmet features two different user interfaces. First, BitGourmet allows users to enter SQL queries via a standard console. Second, BitGourmet features a graphical user interface. The graphical interface allows users to formulate grouped aggregation queries (with a single aggregate) with equality and inequality predicates. Users select the aggregation function, the aggregation column, conditioned columns, and columns characterizing groups via a drop down menu. Query results are shown as plots. For each group, the plot shows the upper and lower bound on the aggregate value (derived via deterministic approximation). Beyond the query interfaces, users can configure the behavior of BitGourmet (e.g., the precision constraints used for approximation) via several configuration files.

2.2 Data Representation

BitGourmet stores the database as a collection of bit vectors on disk. Bit vectors either represent the values of specific bit positions in a certain column or the position of specific values within a specific column. Decomposing the database into bit vectors allows BitGourmet to reduce the amount of data processed (by reading only those bit vectors that contribute most towards approximation precision).

BitGourmet stores bit vectors either in raw or in a compressed format (for sparse bit vectors). Furthermore, during processing, BitGourmet may choose to recompose different bit vectors of the same column into a standard representation again (which makes certain operations faster). BitGourmet may generally convert between different data representations during query processing. Its cost-based optimizer dynamically inserts conversion operations into query plans if they are deemed to reduce execution time.

2.3 Processing Engine

BitGourmet's processing engine generally supports select-join-group by-aggregate queries. We assume that the input database has a star schema. Furthermore, joins are restricted to equijoins between columns connected via key-foreign key constraints.

Queries are executed as follows. We treat one dimension table after the other one (following an order decided by the query optimizer, discussed later). For each dimension table, we approximately evaluate all applicable unary predicates and join with the fact table. Finally, we calculate groups for all relevant rows and calculate lower and upper deterministic bounds for the associated aggregates.

Evaluating a predicate approximately on a row, based on a subset of bits of the involved columns (this subset is selected by the optimizer), may either result in a clear decision (i.e., the row is guaranteed to satisfy or violate the predicate) or not (i.e., we cannot decide whether the row satisfies the predicate). Hence, intermediate results are generally represented by a pair of bit vectors in BitGourmet: the first vector represents rows that certainly satisfy all applicable predicates while the second vector captures rows that possibly satisfy all applicable predicates. While treating a dimension table, we maintain such a vector pair for the rows of the dimension table. During the entire query evaluation process, we maintain such a vector pair for rows in the fact table (capturing which rows in the fact table certainly or possible join with qualifying rows in the dimension tables).

2.4 Specialized Operators

BitGourmet uses specialized processing operators. Those operators differ from standard operators in multiple ways. First, they must handle input data columns that are only partially specified (i.e., values for specific bit positions are known while values for other bit positions are unknown). Second, they must handle uncertain input rows (i.e., for some input rows we are unsure whether they would be part of the input in exact processing). Third, their outputs are deterministic bounds instead of exact values (e.g., lower and upper bounds on the result multiplicity of a row for a filter operation or lower and upper bounds on aggregate values for an aggregation operation).

For certain logical operations, BitGourmet features multiple operator implementations for different input data formats (e.g., compressed versus non-compressed bit vector representation). The optimizer selects among different implementations based on a cost model. As an example, we give an informal description of one of BitGourmet's join operator implementations.

The input are bit vectors representing a subset of bit positions for the join columns. As BitGourmet only supports joins along key-foreign key constraints, one of those two columns is the key column (typically a column of a dimension table) and the other one is the foreign key column (typically a column in the fact table). In addition, the input contains a pair of bit vectors specifying which rows in the dimension table (i.e., the key column table) certainly or possibly remain after prior filtering steps. The join algorithm executes the following steps.

1. We determine bit positions in the join columns for which we have the associated bit vectors in the key and in the foreign key column. We set all other bit positions to zero in the foreign key and the key column and call the result the "partial foreign key" and "partial key" in the following.

2. We iterate over all rows in the dimension table. We associate each partial key with a status value. Possible status values are "All Rows In", "Some Rows In", and "No Rows In", indicating whether rows associated with the partial keys satisfy the predicate.

3. We iterate over the rows in the fact table. For each row, we compare the partial foreign key against the key status values in the previously created table. The fact table row is certainly in the join output if the associated partial key has status "All Rows In", is possibly in the join result if the associated status is "Some Rows In", and is certainly not in the join result if the associated status is "No Rows In".

2.5 Cost-Based Optimizer

BitGourmet features a cost-based query optimizer. The task of the optimizer is to select which bit vectors to read from the database, to determine the order of processing steps, and to select the physical operator implementations. Furthermore, the optimizer dynamically inserts data conversion operators (e.g., conversion from raw to compressed bit vectors) into the query plan if it makes processing steps cheaper.

Optimization considers two metrics: the estimated approximation error of the result (i.e., how far lower and upper deterministic bounds are apart) and estimated processing time. Approximation error only depends on which set of bits is used for processing. The more bits are used, the higher precision can be (assuming that the *optimal* bits are selected). Processing time is influenced by the set of selected bit vectors as well as by other planning choices. The BitGourmet optimizer uses models for predicting approximation error and processing time for specific plan choices. Those models are discussed in the next subsection. The goal of optimization is to minimize processing time under constraints on the maximal approximation error.

To make optimization fast, we use several heuristics to restrict the search space. First, we use heuristics for the order in which we consider different bit vectors from the same column (i.e., instead of considering all subsets of bit positions with a given cardinality for a column, we only consider one subset per cardinality by selecting higher-priority bit positions first). Bit vector priorities are based on the role of the column in the query: for aggregation columns, for instance, we consider bits in decreasing order of weight. For columns used in equality predicates, we consider bit vectors in decreasing order of entropy (i.e., starting with bit positions where fixing a value reduces column entropy more) instead. Second, we only consider one-way data transformations (i.e., we do not consider transforming intermediate results back and forth between two data representations). Within that search space, we use a relatively simple enumeration strategy that is fast enough for typical query sizes. We are currently extending the optimizer to use a dynamic programming-based algorithm instead.

2.6 Cost and Error Models

The BitGourmet query optimizer uses a model for predicting approximation error and a model for predicting processing costs. The approximation error model only depends on the subset of bit positions selected from the database but not on the query plan. The processing cost model depends however on both, the selected bits as well as on the processing plan (e.g., order of operations).

Cost and error model are both based on a scenario-specific cardinality model. The cardinality model estimates the sizes of intermediate results. More precisely, it estimates for each intermediate result the number of rows possibly and certainly satisfying all applicable predicates (whereas standard cardinality models only estimate one single value). To estimate cardinality, we make several simplifying assumptions (e.g., uncorrelated query predicates and uniform data distribution) that are not uncommon in the area of query optimization.

Cardinality and error estimates are based on statistics about the input database. Typically, data statistics cover the value distribution in specific columns. Such statistics are not sufficiently fine-grained for our scenario as our goal is to select specific bit positions of specific input data columns. Hence, we maintain data statistics that describe specific bit positions of specific input columns. Here, we maintain simple statistics such as the ratio of 1s or the compressed size but also the entropy (i.e., how much fixing a value for this bit position decreases column value entropy).

2.7 Proactive Buffer Management

BitGourmet stores bit vectors in memory in the buffer pool. The buffer pool contains data loaded from hard disk as well as intermediate result vectors, generated during data processing. It allows to reuse both types of buffer content across consecutive queries.

Deterministic approximation is most helpful if it allows answering queries from main memory for which exact processing requires disk access (since the entire data set does not fit into main memory). To increase the chances of answering queries from memory, BitGourmet exploits pauses between consecutive queries in an analysis session to proactively fill empty buffer pool slots. For that, it uses a simple query prediction model, assuming that the next query will refer to similar columns as the previous one. Currently based on a few simple rules, it identifies bit vectors that have the potential to reduce processing overheads significantly for the next query. For instance, assuming that the next query uses specific grouping and aggregation columns, BitGourmet would proactively intersect the highest value bits of the aggregation column with bit vectors indicating positions of specific values in the grouping column (and store the intersection result in memory). Hence, if the next query uses those columns as predicted, BitGourmet can avoid accessing aggregation and grouping columns separately and instead access only the intersection result in memory.

3. FIRST EXPERIMENTAL RESULTS

The purpose of our experimental evaluation is fourfold. First, we examine specific components of our systems, e.g. the cost and quality model, to validate their effectiveness (Section 3.1). Second, in an end-to-end evaluation with an exact query processing system, MonetDB, we show that BitGourmet realizes attractive tradeoffs between processing time and quality (Section 3.2). Third, we compare BitGourmet against a sampling-based AQP system, BlinkDB (Section 3.3). For the above experiments, We use the Star Schema Benchmark [13], a simplified version of the TPC-H benchmark. Last, we show how BitGourmet performs on a real-world, exploratory data analysis workload (Section 3.4).

3.1 Model Validation

Setting. We use the Star Schema Benchmark (SSB) with scaling factor 100. We run SSB queries on a denormalized version of the database by joining all dimension tables with the fact table according to primary key-foreign key relationships. All experiments are executed on a laptop with 16GB main memory and a 2.5GHz Intel i5-7200U CPU. We use a Toshiba HDTB310AK3AA disk with up to 5GB/s transfer rate. We use a relative approximation error below 10% as our target precision. Given a lower bound l and a upper bound u, the relative error is calculated as $(u-l)/(u+l)$.

Quality Model. We evaluate our quality model and compare estimated against actual error. In Figure 2, we compare actual errors of the plans with lowest and second

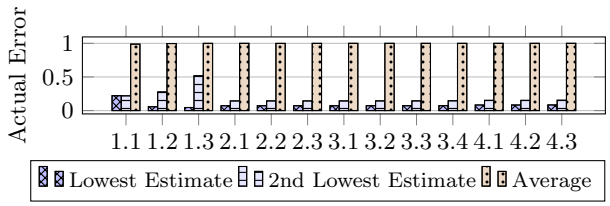

Figure 2: Quality model on all SSB queries: Actual errors of bit selections with lowest error estimates.

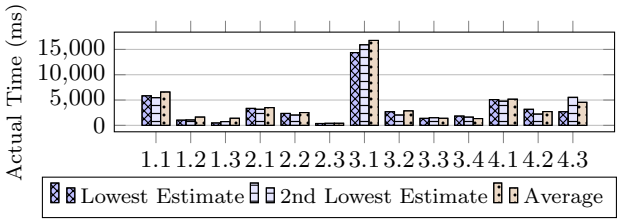

Figure 3: Time cost model on all SSB queries: Actual processing times of execution plans with lowest cost estimates.

lowest error estimates against the average error (averaging over all possible bit selections). Figure 2 shows that our quality model effectively identifies bit selections with small real errors. Note that the error for average plans is typically close to one (the theoretical maximum). This demonstrates the need for optimized bit selections.

Cost Model. We evaluate precision of our time cost model by comparing estimated and actual processing time of different processing strategies, given a fixed (and optimal) bit selection. Figure 3 compares execution time of best and second best plan, according to our model, against the average execution cost (averaging over all possible plans). The plan ranked first, according to our model, outperforms the average in 11 out of 13 cases. We conclude that our execution cost model is helpful in reducing execution costs.

3.2 Comparison against MonetDB

Setting. We use all SSB queries to measure cold start (i.e., systems have to fetch data from hard disk) execution times. Other settings are identical to Section 3.1.

Results. We compare our system against MonetDB with respect to processing time. Our system shows significant speedups while providing deterministic bounds within a user-defined target precision. In Figure 4, BitGourmet is up to 13.2× faster than MonetDB with an average speed up of 6.5× for all SSB queries. Figure 5 compares the number of bits required for exact and approximate processing. Only a small number of bit vectors are needed to produce reasonable bounds. BitGourmet reads significantly fewer bits from hard disk, compared to MonetDB, and therefore achieves significant speed ups (see Figure 6 for correlations between amount of data read and speedups). For BitGourmet, we use the number of bits read for a query as a measure for associated data size. For MonetDB, we compute the sum of storage byte sizes of columns that appear in a query.

3.3 Comparison against Sampling-based AQP

Setting. For BlinkDB, we follow instructions provided in the BlinkDB documentation 'Running BlinkDB Locally' and use 99% confidence error bounds (i.e., its default configuration). We use a denormalized version of the SSB database

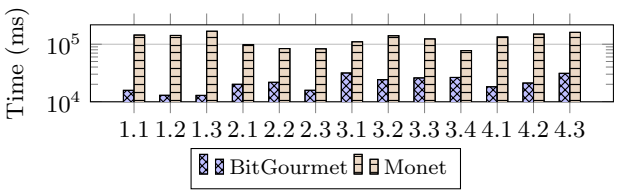

Figure 4: Execution time on all SSB queries.

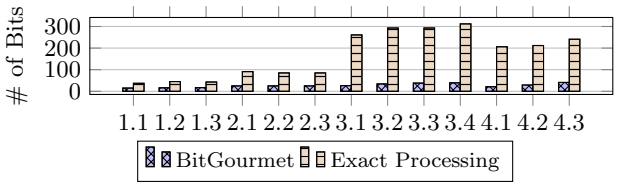

Figure 5: Number of bits used for deterministic approximation and exact processing.

with scaling factor 10. If a query result has multiple groups, we present the average of errors. Other settings are identical to Section 3.1.

Results. Figure 7 compares BitGourmet against BlinkDB. We compare in terms of the relative error as well as the amount of data read (percentage of rows or percentage of bits). For BlinkDB, we consider different sampling rates, ranging from 1 to 40%. BitGourmet realizes Pareto-optimal tradeoffs between error and amount of processed data for most queries (nine out of 13). Drilling down to results for single queries reveals the relative strengths and weaknesses of the two systems. Sampling is prone to errors for small data subsets. Hence, BitGourmet produces better results, compared to BlinkDB, for queries with highly selective predicates or small groups (e.g., this applies to queries 3.2, 3.3, 3.4, and 4.3). In exchange, BlinkDB performs very well for queries with less selective predicates or without grouping (e.g., this applies to queries 1.1, 1.2, 3.1, and 4.1).

Approximate processing via sampling generates confidence bounds that may not contain the actual value. Figure 8 reports the number of groups where the actual value falls outside of the confidence bounds generated by BlinkDB (for different sampling rates). The figure omits queries for which the confidence bounds contain the actual value for each group. BitGourmet, on the other side, generates bounds that are guaranteed to contain the accurate aggregate value. Furthermore, for grouped aggregation queries, a result produced via sampling may miss groups that do not appear in the sample. For a sampling rate of 1%, for instance, BlinkDB misses 132, 5, 3, and 743 groups for queries 3.2, 3.3, 3.4, and 4.3, respectively.

3.4 Real-world Data Analysis

Setting. We conducted a user study to collect workloads that are representative of exploratory data analysis. Partic-

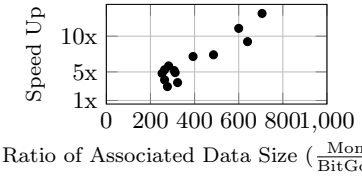

Figure 6: Data size associated with a query versus speed up against MonetDB (one dot per SSB query).

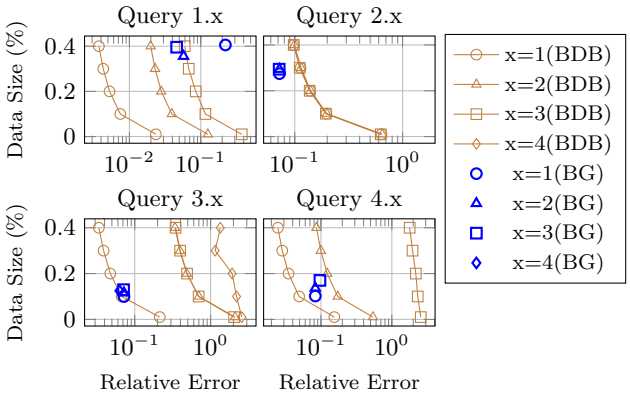

Figure 7: Amount of data read versus relative error of BitGourmet (BG) and BlinkDB (BDB) on all SSB queries.

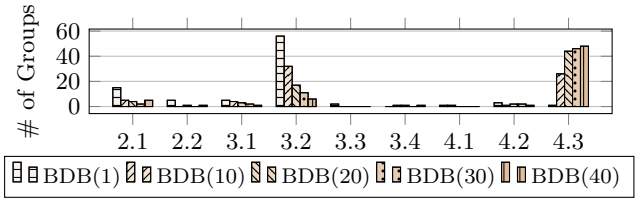

Figure 8: Numbers of out-of-bounds for BlinkDB with different sampling rates (%).

ipants were asked to find interesting trends from a 21.9 GB data set describing all United States births from 1969 to 2008[1], using the BitGourmet GUI. We had five participants and each participant was given 20 minutes to analyze the data. We collected the queries issued by the participants along with the timestamps at which each query was issued. In summary, we gathered a total of five workloads consisting of analytical queries and their timestamps.

The motivation for collecting a real interactive analysis workload was in particular to evaluate our buffer management strategies in a realistic setting (which is not possible with SSB). The associated timestamps are required to evaluate proactive buffer management strategies (since their effectiveness may depend on the available time for proactive pre-processing between consecutive queries).

Results. We evaluate BitGourmet, with and without proactive buffer management, as well as MonetDB on all five workloads. Figure 9 reports the results. BitGourmet with proactive buffer management is up to 16.2% faster than BitGourmet without proactive buffer management with an average speed up of 7.1%. Compared to MonetDB, BitGourmet is in average 28.5× faster for all workloads.

4. ONGOING WORK

In the following, we discuss our ongoing work and future work plans with regards to BitGourmet and deterministic approximation in general.

4.1 Online Aggregation with DAQ

Online aggregation [8] continuously updates an aggregation result as more data is being processed. This approach frees users from having to specify a desired approximation precision a-priori (which can be difficult before seeing initial

[1]https://www.kaggle.com/bigquery/samples

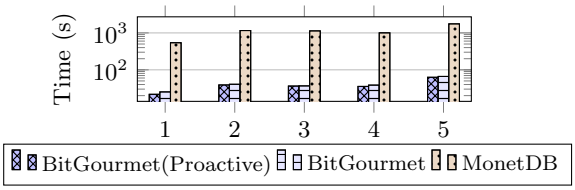

Figure 9: Overall execution time on each workload.

results). Prior work [8] has shown that nontrivial extensions in query optimization and execution are required for the database systems to support online aggregation. The same is true in our scenario. Our goal is to extend the graphical user interface of BitGourmet for deterministic online approximation. During each refinement step, we want to avoid redundant work and reuse intermediate results generated in previous refinement steps as much as possible. Hence, an optimal query plan should aim at minimizing processing cost not only of the current but also of future refinement steps. This motivates a holistic planning approach that optimizes sequences of refinement steps.

4.2 Optimal Physical Design for DAQ

In our current implementation, we vertically divide a column as is and store these bit vectors on disk. Even though this representation works reasonably well, there are several directions in which we can improve the physical mapping from a database to bit vectors. One approach for better performance is to order data so that processing only a subset of tuples (and a subset of bit vectors) gives us reasonable bounds. One intuitive example is when we calculate a lower bound of a summation over a column. If the values in the column are sorted in a decreasing order, the lower bound will narrow down faster as we process the first few tuples compared to a random order. Another approach is to encode values in a column to a fixed-length bit representation which produces narrower deterministic bounds. The approximation error is determined by the amount of remaining uncertainty after reading a subset of bit vectors. Thus, the question is to find an encoding that maximizes the information gain from a bit subset (usually depends on value distributions).

4.3 Smarter Proactive Buffer Management

Currently, BitGourmet uses a relatively simple model for predicting future queries. Based on those predictions, empty slots in the buffer pool are proactively filled with bit vectors that are likely to be useful. Our experimental results show that this simple model is already beneficial and improves performance. For the future, we are considering two extensions of the proactive buffer management approach. First, we are working on extending our current, rule-based prediction model towards a more precise probabilistic model. For instance, we plan to take into account not only occurrence frequencies of single columns in past queries but also correlations between the occurrence of different columns. Second, we plan to exploit query predictions more systematically. Currently, we apply a set of relatively simple rules to determine useful bit vectors based on query predictions. In the future, we plan to use our cost-based optimizer to predict execution cost for expected queries, based on different buffer contents. Filling the buffer with an optimal combination of bit vectors for the expected query workload can then be formalized as a global optimization problem.

5. RELATED WORK

Potti and Patel [18] introduce approximate processing via deterministic approximation. They also reduce the number of processed bits to obtain speedups. In that, their work is similar to BitGourmet. However, the approaches proposed by Potti and Patel are only applicable to simple queries on a single table, referring to a single aggregate or predicate. BitGourmet supports a much broader range of queries, featuring for instance joins and supporting multiple predicates, aggregates, and grouping. It is a full-blown processing engine that handles issues such as dynamic data conversions, automated bit selections, and query optimization. Note that challenges, such as the selection of optimal subsets of bits for processing, do not arise unless complex queries are considered. To support optimization, we introduce cardinality, cost, and result quality models that are specific to the domain of deterministic approximation.

Beyond the area of deterministic approximation, our use of bit vectors to represent intermediate results (as a subset of rows satisfying predicates) bears some resemblance to prior work in the area of bit-sliced indexing for efficient processing of OLAP-style workloads [16]. However, our goal is different, as we exploit bit vectors for fast approximate processing while the prior work is focused on exact processing.

Beyond the initial work by Potti and Patel, there has been follow-up work on deterministic approximation which is however specific to time series [4, 3]. BitGourmet is focused on relational data instead. Our work also relates to prior work on generating partial results in the face of incomplete data [11, 20, 22]. Here, the motivation is however not in increasing processing efficiency but rather to account for missing values.

BitGourmet is generally situated in the area of approximate processing for which a large body of work is already available [15, 8, 7, 2, 6, 1, 9, 10, 12, 14, 19, 23, 17, 21]. A complete survey of this area is beyond the scope of this publication while we refer to prior work [5]. Overall, prior work on approximate processing has focused on sampling which does not result in deterministic bounds. BitGourmet differs from most prior work as it produces bounds that are guaranteed to contain accurate values.

6. CONCLUSION

We give an overview of our work towards a novel system for deterministic approximation, BitGourmet, which processes carefully selected subsets of bits to obtain near-optimal tradeoffs between processing time and result precision. BitGourmet features a specialized optimizer, based on custom cardinality, cost, and error models, as well as specialized processing operators. We presented experimental results for an early version of BitGourmet. Those results demonstrate that deterministic approximation can result in significant speedups compared to exact processing. Also, our results indicate that deterministic approximation can lead to better result quality than sampling.

7. REFERENCES

[1] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. The Aqua Approximate Query Answering System. In *SIGMOD*, pages 574–576, 1999.

[2] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In *EuroSys*, pages 29–42, 2013.

[3] E. Boursier, J. J. Brito, C. Lin, and Y. Papakonstantinou. Plato: Approximate Analytics over Compressed Time Series with Tight Deterministic Error Guarantees. *CoRR*, abs/1808.04876, 2018.

[4] J. J. Brito, K. Demirkaya, E. Boursier, Y. Katsis, C. Lin, and Y. Papakonstantinou. Efficient Approximate Query Answering over Sensor Data with Deterministic Error Guarantees. *CoRR*, abs/1707.01414, 2017.

[5] S. Chaudhuri, B. Ding, and S. Kandula. Approximate Query Processing: No Silver Bullet. In *SIGMOD*, pages 511–519, 2017.

[6] A. Cuzzocrea. Providing Probabilistically-bounded Approximate Answers to Non-holistic Aggregate Range Queries in OLAP. In *DOLAP*, pages 97–106, 2005.

[7] A. Dobra, C. Jermaine, F. Rusu, and F. Xu. Turbo-Charging Estimate Convergence in DBO. *PVLDB*, 2(1):419–430, 2009.

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

[9] S. Joshi and C. Jermaine. Materialized Sample Views for Database Approximation. *TKDE*, 20(3):337–351, 2008.

[10] S. Kandula, A. Shanbhag, A. Vitorovic, M. Olma, R. Grandl, S. Chaudhuri, and B. Ding. Quickr: Lazily Approximating Complex AdHoc Queries in BigData Clusters. In *SIGMOD*, pages 631–646, 2016.

[11] W. Lang, R. V. Nehme, E. Robinson, and J. F. Naughton. Partial results in database systems. In *SIGMOD*, pages 1275–1286, 2014.

[12] F. Li, B. Wu, K. Yi, and Z. Zhao. Wander Join: Online Aggregation via Random Walks. In *SIGMOD*, pages 615–629, 2016.

[13] P. O. Neil, B. O. Neil, and X. Chen. Star Schema Benchmark. 2009.

[14] S. Nirkhiwale, A. Dobra, and C. M. Jermaine. A Sampling Algebra for Aggregate Estimation. *PVLDB*, 6(14):1798–1809, 2013.

[15] F. Olken and D. Rotem. Random sampling from databases: a survey. *Statistics and Computing*, 5(1):25–42, 1995.

[16] P. E. O'Neil and D. Quass. Improved Query Performance with Variant Indexes. In *SIGMOD*, pages 38–49, 1997.

[17] J. Peng, D. Zhang, J. Wang, and J. Pei. AQP++: Connecting Approximate Query Processing With Aggregate Precomputation for Interactive Analytics. In *SIGMOD*, pages 1477–1492, 2018.

[18] N. Potti and J. M. Patel. DAQ: A New Paradigm for Approximate Query Processing. *PVLDB*, 8(9):898–909, 2015.

[19] C. Qin and F. Rusu. PF-OLA: A High-performance Framework for Parallel Online Aggregation. *Distrib. Parallel Databases*, 32(3):337–375, 2014.

[20] S. Razniewski, F. Korn, W. Nutt, and D. Srivastava. Identifying the Extent of Completeness of Query Answers over Partially Complete Databases. In *SIGMOD*, pages 561–576, 2015.

[21] A. Rudra, R. P. Gopalan, and N. Achuthan. An Efficient Sampling Scheme for Approximate Processing of Decision Support Queries. In *ICEIS*, pages 16–26, 2012.

[22] B. Sundarmurthy, P. Koutris, W. Lang, J. F. Naughton, and V. Tannen. m-tables: Representing Missing Data. In *ICDT*, pages 21:1–21:20, 2017.

[23] B. M. Yongjoo Park, Ahmad Shahab Tajik, Michael Cafarella. Database Learning: Toward a Database that Becomes Smarter Every Time. In *SIGMOD*, pages 587–602, 2017.