AQP++: Connecting Approximate Query Processing With Aggregate Precomputation for Interactive Analytics

Jinglin Peng[◇] Dongxiang Zhang[◇†] Jiannan Wang[◇] Jian Pei[¶◇]
Simon Fraser University[◇] National University of Defense Technology[†] JD.com[¶]
{jinglin_peng, jnwang, jpei}@sfu.ca zhangdongxiang12@nudt.edu.cn

ABSTRACT

Interactive analytics requires database systems to be able to answer aggregation queries within interactive response times. As the amount of data is continuously growing at an unprecedented rate, this is becoming increasingly challenging. In the past, the database community has proposed two *separate* ideas, sampling-based approximate query processing (AQP) and aggregate precomputation (AggPre) such as data cubes, to address this challenge. In this paper, we argue for the need to connect these two separate ideas for interactive analytics. We propose AQP++, a novel framework to enable the connection. The framework can leverage both a sample as well as a precomputed aggregate to answer user queries. We discuss the advantages of having such a unified framework and identify new challenges to fulfill this vision. We conduct an in-depth study of these challenges for range queries and explore both optimal and heuristic solutions to address them. Our experiments using two public benchmarks and one real-world dataset show that AQP++ achieves a more flexible and better trade-off among preprocessing cost, query response time, and answer quality than AQP or AggPre.

1 INTRODUCTION

Data analytics is essential to enable data-driven decision making. Batch analytics is often run offline and may take several hours or even days to generate results. In comparison, interactive analytics aims to answer queries within interactive response times (e.g., less than 1s depending on a user's tolerance for waiting time), allowing human analysts to rapidly iterate between hypotheses and evidence. In this paper, we focus on interactive analytics. Specifically, we aim to enable database systems to answer aggregation queries on large-scale datasets interactively.

Due to the high demand for human-in-the-loop data analytics, interactive analytics was recently attracted a lot of attentions [54]. One way to achieve interactive analytics is to build a fast query engine using modern database techniques [7, 43, 48, 67]. As the amount of data continues to grow at an unprecedented pace, this is becoming increasingly challenging. Complementary to building a fast query engine, another way is to avoid scanning all of the data associated with a query [9]. In the past, two *separate* ideas have been proposed by database community to achieve this goal.

One is *sampling-based approximate query processing* (AQP) [4, 6, 15, 56]), which creates a random sample of data and uses the sample to estimate query results. The other is *aggregate precomputation* (AggPre) such as data cubes [30, 32, 34, 53], which precomputes the answers to some aggregation queries and then uses the precomputed aggregates to speed up query performance. In comparison, although both approaches can answer queries fast, they achieve the goal via different trade-offs. Essentially, AQP varies sample size to balance the trade-off between answer quality and query response time; AggPre varies the size of precomputed aggregates to balance the trade-off between *preprocessing cost* (i.e., the time and space used to calculate and store aggregates) and query response time.

AQP has been extensively studied in the past [3, 4, 13, 15, 27, 29, 33, 37, 56, 57] and there has been a resurgence of interest in recent years [5, 6, 17, 19, 24, 42, 44, 45, 45, 52, 55, 59, 69, 71, 72]. However, to the best of our knowledge, none of existing AQP work has sought for a *general* framework to connect AQP with AggPre (see Section 2 for a detailed comparison).

There are (at least) two strong motivations for building such a connection. Firstly, it is often the case that a data warehouse has already precomputed the answers to a large number of aggregation queries (e.g., data cubes). Suppose one wants to apply AQP to the data warehouse in order to reduce the query response time. Without building the connection with AggPre, AQP will miss the opportunity to leverage the (free) precomputed aggregates to improve its query performance. Secondly, as mentioned above, AQP and AggPre implement interactive analytic through different trade-offs (i.e., answer quality vs. response time, preprocessing cost vs. response time). Connecting AQP with AggPre will allow a more flexible and better trade-off between preprocessing cost, query response time, and answer quality, and thus is more likely to satisfy user needs.

To this end, we propose AQP++[1], a general framework to connect AQP with AggPre for interactive analytics. The key idea of AQP++ is that, unlike AQP that uses a sample to directly estimate a query result, it uses a sample to estimate the difference of the query result from a precomputed aggregate. Consider a simple example. Suppose we want to estimate the answer to the following query q.

$q(\mathcal{D})$: `SELECT SUM(`A`) FROM` $\mathcal{D}$ `WHERE` $1 < C < 10$.

Suppose the answer to the following aggregate query, pre, has been precomputed.

$pre(\mathcal{D})$: `SELECT SUM(A) FROM` $\mathcal{D}$ `WHERE` $2 < C < 10$.

To estimate $q(\mathcal{D})$, AQP++ first uses a sample to estimate the difference of $q(\mathcal{D})$ from $pre(\mathcal{D})$, i.e.,

$q(\mathcal{D})$-$pre(\mathcal{D})$: `SELECT SUM(A) FROM` $\mathcal{D}$ `WHERE` $1 < C \leq 2$.

Then, it adds the estimated difference to the known $pre(\mathcal{D})$ to obtain the estimation of $q(\mathcal{D})$.

[1] AQP++ is open-sourced at https://github.com/sfu-db/aqppp

Acknowledgements. We thank Liwen Sun and Bolin Ding for their comments on the paper. This work was supported in part by the NSERC Discovery Grant program, the Canada Research Chair program, the NSERC Strategic Grant program, the NSERC CRD Grant program, the NSERC Engage Grant program, and an SFU Presidents Research Start-up Grant. All opinions, findings, conclusions and recommendations in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

SIGMOD'18, June 10–15, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3183747

We find that AQP++ framework is very general because (1) it works for a wide range of aggregate functions such as SUM, COUNT, AVG, and VAR; and (2) it can be easily extended to support many optimization techniques (that were originally proposed for AQP) such as stratified sampling [3, 6, 8, 13–15, 27, 66] and outlier indexing [13]. Furthermore, for any aggregation query, if AQP can estimate its result unbiasedly, AQP++ can return an unbiased estimate as well. That is, in expectation the estimated value is equal to the true value.

To quantify the uncertainty of an estimated answer, we investigate how AQP++ can compute a confidence interval, and demonstrate analytically and empirically when AQP++ can return a tighter confidence interval (i.e., a more accurate answer) than AQP. Specifically, we treat the estimated answers to a user query and a precomputed query as two random variables. We find that the more correlated the two random variables are, the more accurate answer AQP++ can return. In an extreme case, if a user query and a precomputed query are identical, which has a correlation coefficient of 1, AQP++ will return the exact answer.

To examine whether AQP++ can achieve a better trade-off among preprocessing cost, response time, and answer quality compared to AQP or AggPre alternatives, we conduct an in-depth study of AQP++ for *range queries*. A range query applies an aggregation operation over all records that are selected by a conjunction of range conditions. We choose this form of queries because (1) it is one important class of queries for analytical workloads, and (1) it can not only be well supported by AQP but also has been extensively studied in the AggPre literature [12, 21, 28, 34, 47]. The study requires solving two challenging problems.

- *Aggregate Identification.* Given a user query, there might be a large number of precomputed aggregate values available. How should we quickly identify the best one for the query?
- *Aggregate Precomputation.* Precomputing all possible aggregate values is prohibitively expensive. Given a space budget, how should we decide which aggregate values to be precomputed?

We use a simple example to illustrate the challenges of the two problems as well as our contributions made to address them. Suppose a user wants to interactively issue a query in following form:

SELECT SUM(A) FROM $\mathcal{D}$ WHERE $x \leq C \leq y$

where $x, y = 1, 2, \cdots, 100$. An efficient AggPre approach is to precompute a *prefix cube* [34]. In this example, the (1-dimensional) prefix cube consists of 100 cells:

SELECT SUM(A) FROM $\mathcal{D}$ WHERE $C \leq t$

where $t = 1, 2, \cdots, 100$. Once the cube is created, it is easy to see that any user query can be answered by accessing at most 2 cells in the cube. Since there are $\frac{101 \cdot 100}{2} = 5050$ different user queries, the precomputed cube (with only 100 cells) can be thought of as containing 5050 aggregate values.

Let us first consider the aggregate-identification problem. Suppose that, due to the space constraint, only 10 cells (e.g., $t = 10, 20, \cdots, 100$) in the cube can be stored. In this situation, the cube contains $\frac{11 \cdot 10}{2} = 55$ aggregate values. Given a query, e.g.,

SELECT SUM(A) FROM $\mathcal{D}$ WHERE $15 \leq C \leq 41$,

AQP++ needs to decide which one of the following 55 precomputed values should be used to estimate the answer to the above query.

SELECT SUM(A) FROM $\mathcal{D}$ WHERE $x \leq C \leq y$

where $x, y = 10, 20, \cdots, 100$.

Since the query's answer quality highly depends on the identified value, the decision has to be made carefully. But, we cannot afford trying out all aggregate values as it will increase query processing time significantly. To this end, we propose an efficient aggregate-identification approach. The key idea is to quickly identify a small number of aggregate values whose corresponding queries look similar to the user query and then examine which one is the best only among them. We prove the optimality of this approach under certain assumptions and show that it achieves good performance empirically in the general setting.

Next, let us turn to the aggregate-precomputation problem. Suppose the space budget is only available for creating a prefix cube with 10 cells. Our goal is to decide which 10 out of 100 cells should be selected to precompute. Note that there are $\binom{100}{10}$ different ways to select the 10 cells, so a brute-force search will not work. We formally define the problem in the paper, and find that an *equal-partition scheme* (i.e., $t = 10, 20, \cdots, 100$) is not always optimal. To solve the problem, we find that two factors, attribute correlation and data distribution, may affect the cell-selection decision. We first prove that if attributes A and C are independent, and C has no duplicate values, the equal-partition scheme is optimal. The theoretical result guides us to develop a hill-climbing based heuristic approach that can adjust the partition scheme *adaptively* based on the actual attribute correlation and data distribution.

Please note that the above simple example only demonstrates the challenges for one-dimensional range queries (i.e., with a single range condition). There are many other challenges involved when dealing with multidimensional cases. In the paper, we discuss these challenges in detail and propose effective approaches to them.

In summary, our paper makes the following contributions:

- We argue that the two separate ideas for interactive analytics, AQP and AggPre, should be connected together, and propose AQP++, the first general framework to enable the connection.
- We conduct an in-depth study of AQP++ for range queries, and formalize two challenging problems in the study: aggregate identification and aggregation precomputation.
- We develop an efficient aggregate-identification approach and show effectiveness of the approach analytically and empirically.
- We identify two important factors that affect the solution to the aggregation-precomputation problem. We prove that the equal-partition scheme is only optimal under certain assumptions and propose an effective hill-climbing approach for general situations.
- We evaluate AQP++ using a commercial OLAP system on three datasets. Experimental results show that AQP++ can achieve up to 10x more accurate answers than AQP for a relatively small preprocessing cost.

The remainder of this paper is organized as follows. We review related work in Section 2. In Section 3, we formally defines the aggregate identification and aggregate precomputation problems. To address these problems, we first present the AQP++ framework in Section 4, and then propose effective approaches for them in Section 5 and Section 6, respectively. We describe the results of our experimental studies in Section 7 and present conclusions and future work in Section 8.

2 RELATED WORK

Approximate Query Processing. Sampling-based AQP has been extensively studied in the last several decades [17, 23, 51]. A lot

of techniques have been proposed to optimize AQP's performance. Most of them are focused on generating better *stratified samples* [3, 6, 8, 13–15, 27, 66]. There has also been some work that tries to augment samples with auxiliary indices [13, 24, 50]. Unlike them, AQP++ is a framework that can connect *any* existing AQP engine with AggPre. Thus, all these techniques can be easily extended to the AQP++ framework (see Section 4.2 for a detailed discussion).

AQP++ is similar in spirit to some recent work about AQP systems [26, 59], which observe that previous answers can be beneficial to estimating the answers to future queries. AQP++ is fundamentally different from these work in two aspects: (1) AQP++ utilizes precomputed exact answers over full data rather than approximate answers over samples to improve future queries; (2) AQP++ uses a sample to estimate the difference between the answers to a previous query and a future query, rather than build a model to predict unobserved data points.

Our work is also related to *Approximate Pre-Aggregation* [35, 38], which combines samples with a small set of statistics of the data to improve answer quality. However, they assume that the set of statistics are available in the system without considering how to precompute a BP-Cube as well as how to use it for result estimation.

Online aggregation [33, 42, 45, 58, 62, 63, 70, 71] is another popular application scenario of AQP systems, which progressively improves answer quality by dynamically increasing sample size. In this paper, we consider the scenario where samples are created before queries, but it would be an interesting future direction to investigate the use of AQP++ in an online-sampling setting.

In fact, AQP++ was initially inspired by SampleClean [44, 69]. SampleClean enables fast and accurate query processing on dirty data. Its basic idea is to clean a sample of dirty data and then use the cleaned sample to correct dirty query result. Specifically, SampleClean is given one query, and the goal is to estimate the difference of its results computed on two datasets (a dirty dataset and a cleaned dataset). In contrast, AQP++ is given two queries (a user query and a precomputed query), and the goal is to estimate the difference of their results computed on one dataset.

In addition to sampling-based AQP, a number of non-sampling based techniques are proposed recently [11, 61]. They aim to support more complex queries as well as provide deterministic guarantees by using indices rather than samples. However, for the query class we support, they are not as effective as sampling-based AQP.

Aggregate Precomputation. Aggregate precomputation is another extensively studied topic in the database community for improving analytical query performance [30, 34, 49, 53, 64]. Data cubes, which store data aggregated across all possible combinations of dimensions, are widely used in data warehouses [16, 53]. Since it is often very expensive to store a complete data cube [65], there is some work on partial data cube precomputation [25, 25, 32]. There is also some work that tries to apply sampling or approximation techniques to data cubes [10, 36, 40, 41, 46, 68]. For example, Li et al. [46] studied how to compute the confidence intervals for a cube constructed from a sample. Vitter and Wang [68] proposed an I/O efficient technique to build an approximate data cube based on wavelets. Compared with AQP, these techniques still suffer from (1) much higher preprocessing cost or/and (2) not good at answering ad-hoc queries. Thus, having a unified framework like AQP++ is more desirable.

Appendix D discusses how AQP relates to materialized views.

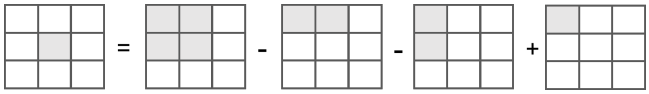


Figure 1: A geometric illustration of the 2-D case.

3 PROBLEM FORMALIZATION

This section formally defines our problems. For ease of presentation, we assume that our queries do not have a group-by clause. The extension to group-by queries will be discussed in Appendix C.

Definition 1 (Query Template). *A query template, denoted by* $Q : [f(A), C_1, C_2, \cdots C_d]$*, represents a collection of queries of the following form*[2]*:*

```
SELECT f(A) FROM table
WHERE x_1 ≤ C_1 ≤ y_1 and ··· and x_d ≤ C_d ≤ y_d
```

where f*,* A*, and* $C_i (i \in [1, d])$ *are called aggregation function, aggregation attribute, and condition attributes, respectively, and* x_i, y_i *are in the data domain of* C_i *for each* $i \in [1, d]$*.*

For example, if a user wants to explore the relationship between product sales and customer ages, she can specify a query template like $[\mathsf{SUM}(sale), age]$.

For ease of presentation, we assume that $f = \mathsf{SUM}$ in later text and discuss the extension to other aggregation functions in Appendix C. Moreover, we assume that the data domain of each C_i is $\mathrm{dom}(C_i) = \{1, 2, \cdots, |\mathrm{dom}(C_i)|\}$[3], and abbreviate a range query as $\mathsf{SUM}(x_1 : y_1, x_2 : y_2, \cdots, x_d : y_d)$.

In AggPre literature [12, 21, 28, 34, 47], people often precompute a *prefix cube* (P-Cube) (or its variations) to answer range queries.

Definition 2 (Prefix Cube). *Given a query template* $Q : [\mathsf{SUM}(A), C_1, C_2, \cdots C_d]$*, the prefix cube consists of the answers to all the queries of the following form:*

$$\mathsf{SUM}(1 : y_1, 1 : y_2, \cdots, 1 : y_d),$$

where $y_i \in dom(C_i)$ *and* $i \in [1, d]$*.*

Each answer is called a *cell* in the cube. A nice property about P-Cube is that for any range query in Q, its answer can be computed from no more than 2^d cells. For example, consider a 1D range query: $\mathsf{SUM}(3 : 5)$. The answer for this query can be obtained from 2 cells, i.e., $\mathsf{SUM}(1 : 5) - \mathsf{SUM}(1 : 2)$. For a 2D range query, the answer can be obtained from at most 4 cells. There is a geometric illustration in Figure 1.

However, it is often expensive to precompute the entire P-Cube (with $\prod_{i=1}^{d} |\mathrm{dom}(C_i)|$ cells). Thus, we precompute a *blocked prefix cube* (BP-Cube) [34] consisting of a small portion of the cells.

Definition 3 (Blocked Prefix Cube). *Given a query template* Q*, let* $dom(C_i)_{small}$ *denote a subset of* $dom(C_i)$ *for each* $i \in [1, d]$*. The blocked prefix cube consists of the answers to all the queries of the following form:*

$$\mathsf{SUM}(1 : y_1, 1 : y_2, \cdots, 1 : y_d),$$

where $y_i \in dom(C_i)_{small}$ *and* $i \in [1, d]$*.*

BP-Cube reduces the number of cells to be precomputed from $\prod_{i=1}^{d} |\mathrm{dom}(C_i)|$ to $\prod_{i=1}^{d} |\mathrm{dom}(C_i)_{small}|$. For example, consider a 2D query template $Q : [\mathrm{SUM}(A), C_1, C_2]$, where $\mathrm{dom}(C_1) =$

[2]Note that this form of queries subsumes those with other forms of range conditions, e.g., "$C = x$", "$C \geq x$", "$x \leq C < y$". This paper focuses on single-table queries, but it is straightforward to extend AQP++ to handle foreign key joins using a similar idea from [6]. We defer other complex join queries to future work.

[3]If C_i does not have a natural ordering (e.g., country), we use an alphabetical ordering.

$\{1, 2, \cdots, 15\}$ and $\text{dom}(C_2) = \{1, 2, \cdots, 8\}$. The full P-Cube contains $15 * 8 = 120$ cells, i.e., $\text{SUM}(1 : y_1, 1 : y_2)$ for all $y_1 \in [1, 15]$ and $y_2 \in [1, 8]$. Suppose $\text{dom}(C_1)_{small} = \{5, 10, 15\}$ and $\text{dom}(C_2)_{small} = \{4, 8\}$. Then the BP-Cube only contains $3 * 2 = 6$ cells, i.e., $\text{SUM}(1 : y_1, 1 : y_2)$ for all $y_1 \in \{5, 10, 15\}$ and $y_2 \in \{4, 8\}$.

Aggregate Identification. We now start defining the aggregate-identification problem. Let P denote a BP-Cube, i.e.,

$$P = \Big\{\text{SUM}(1 : y_1, 1 : y_2, \cdots, 1 : y_d) \mid \text{for all } y_i \in \text{dom}(C_i)_{small} \text{ and } i \in [1, d]\Big\}. \quad (1)$$

Note that although only P is precomputed, due to the properties of the BP-Cube, we can assume that the answers to all the queries in P^+ are available.

$$P^+ = P \cup \{\phi\} \cup \Big\{\text{SUM}(x_1 + 1 : y_1,\ x_2 + 1 : y_2,\ \cdots,\ x_d + 1 : y_d) \mid \text{for all } x_i, y_i \in \text{dom}(C_i)_{small} \text{ and } i \in [1, d]\Big\}. \quad (2)$$

For example, suppose $\text{dom}(C_1)_{small} = \{4, 6\}$. Then, we have $P = \{\text{SUM}(1 : 4), \text{SUM}(1 : 6)\}$, and $P^+ = \{\text{SUM}(1 : 4), \text{SUM}(1 : 6), \text{SUM}(5 : 6), \phi\}$, where $\text{SUM}(5 : 6)$ can be obtained from $\text{SUM}(1 : 6) - \text{SUM}(1 : 4)$ and ϕ is an empty query with an always-false condition.

Given a user query q and a sample S, for each $pre \in P^+$, AQP++ can return an approximate answer along with a confidence interval (see Section 4.2 for more detail about this). For example, suppose $q : \text{SUM}(1 : 4)$. Imagine AQP++ leverages the precomputed query $pre : \text{SUM}(2 : 4)$ and returns 1000 ± 5 (confidence level: 95%). This result indicates that the true answer of q is in the range of $[995, 1005]$ with 95% probability. The larger the width of the confidence interval, the less accurate the approximate answer. For this reason, we define the *query error* of q w.r.t. pre, denoted by $error(q, pre)$, as half the width of the confidence interval. For the above example, the confidence interval has the width of 10, thus $error(q, pre) = 5$.

Once a BP-Cube P is precomputed, since any value in P^+ can be leveraged to answer a user query, we aim to select the best one with the minimum query error. We denote the minimum query error w.r.t. P by $error(q, P) = \min_{pre \in P^+} error(q, pre)$. Problem 1 formally defines the problem.

Problem 1 (Aggregate Identification). *Given a user query q, a sample S, and a BP-Cube P, the goal of aggregate identification is to identify the best value in P^+ such that the query error is minimized:*

$$\underset{pre \in P^+}{\text{argmin}}\ error(q, pre)$$

Consider a 1D example. Given a user query $q : \text{SUM}(2 : 5)$ and a BP-Cube $P = \{\text{SUM}(1 : 4), \text{SUM}(1 : 6)\}$, there are four precomputed values in $P^+ = \{\text{SUM}(1 : 4), \text{SUM}(1 : 6), \text{SUM}(5 : 6), \phi\}$. The aggregate-identification problem aims to identify the best value among the four values and use it to estimate the answer to q.

Aggregate Precomputation. Next, we define the aggregate-precomputation problem. Given a space budget, we want to find the best BP-Cube that satisfies the space budget. Let $|P|$ denote the number of cells in a BP-Cube P and k denote a threshold bounding $|P|$. Given a threshold k, there are a lot of different ways to construct a BP-Cube such that $|P| \le k$. For example, suppose $k = 6$ and $d = 2$. The shapes of BP-Cubes can be 1×6, 2×3, 3×2 or 6×1. For each possible shape, e.g., 2×3, a BP-Cube can be constructed by choosing any 2 values from $\text{dom}(C_1)$ and any 3 values from $\text{dom}(C_2)$.

To decide which one is the best, we define *query-template error* to quantify the benefit of each BP-Cube and return the one with the minimum query-template error. Given a query template Q and a BP-Cube P, since a user might issue any query in Q, we define the *query-template error* w.r.t P as $error(Q, P) = \max_{q \in Q} error(q, P)$, which is the maximum query error over all possible user queries. We choose this error metric because it is more beneficial to reduce the errors of highly inaccurate queries rather than the ones who have already gotten very accurate results. Certainly, there are many other ways to define a query-template error. In the experiments, we show that our aggregate-precomputation approach, which is designed to optimize the maximum error, can also significantly reduce other types of errors, such as average error and median error. Problem 2 defines the aggregate-precomputation problem.

Problem 2 (Aggregate Precomputation). *Given a query template Q, a sample S, and a threshold k, the goal of aggregate precomputation is to determine the best BP-Cube P such that $|P| \le k$ and the query-template error is minimized:*

$$\underset{P}{\text{argmin}}\ error(Q, P) \text{ s.t. } |P| \le k$$

Consider a 1D example. Given a threshold k = 2 and a query template $Q : [\text{SUM}(A), C_1]$, where $\text{dom}(C_1) = \{1, 2, \cdots, 5\}$, the aggregate-precomputation problem aims to determine the best BP-Cube $P = \{\text{SUM}(1 : t_1), \text{SUM}(1 : t_2)\}$ with the two values t_1 and t_2 chosen from $\text{dom}(C_1)$ such that $error(Q, P)$ is minimized. For multiple-dimensional cases, the problem becomes even more challenging because we also need to determine the shape of the best BP-Cube, e.g., 2×3 or 3×2.

4 FROM AQP TO AQP++

In this section, we first provide some background knowledge about AQP, and then present the AQP++ framework.

4.1 Sampling-based AQP

AQP's mathematical foundations are built on sampling and estimation theories [23].

Result Estimation. Given a large relational table $\mathcal{D}$ and a random sample $\mathcal{S}$ of the table, for certain aggregation queries q, their answers can be estimated based on the sample, i.e.,

$$q(\mathcal{D}) \approx \hat{q}(\mathcal{S}), \quad (3)$$

or simply $q \approx \hat{q}$ if the context is clear. The supported aggregation queries are of this form:

```
SELECT f(A) FROM D WHERE Condition.
```

Please note that `Condition` can be ***any*** function that takes a record as input and returns True or False (e.g., `age > 30 and country = "USA"`, `tweet like "%sigmod%"`).

Note that f cannot be an arbitrary aggregation function (e.g., min, max). Typical aggregation functions include AVG, SUM, COUNT, and VAR. A recent paper shows that some kinds of User Defined Functions (UDFs) can be supported as well [5].

Confidence Interval. Along with an estimated result, AQP often returns a confidence interval to quantify the estimation uncertainty. The confidence interval is an interval estimate of the true value. For example, a 95% confidence interval $\hat{q} \pm \epsilon$ means that the true value is within this range with 95% probability. In AQP, there are two kinds of approaches computing a confidence interval. The first one is an analytical approach, aiming to derive a closed-form confidence interval often based on the Central Limit Theorem. The limitation of this approach is the lack of generality. Each query has its own form of the confidence interval. Example 1 shows how to compute the closed-form confidence interval for a SUM query.

Example 1. *Suppose we want to use AQP to estimate the answer along with the confidence interval for the following query:*

$$q(\mathcal{D}) = \texttt{SELECT SUM}(A) \texttt{ FROM } \mathcal{D} \texttt{ WHERE } C \geq 0$$

To facilitate the calculation, we first rewrite it as follows:

$$q(\mathcal{D}) = \texttt{SELECT SUM}(A \cdot \text{cond}(C \geq 0)) \texttt{ FROM } \mathcal{D}$$

where $\text{cond}(C \geq 0)$ *returns 1 if* $C \geq 0$ *holds; 0, otherwise.*

For ease of presentation, we denote $A \cdot cond(C \geq 0)$ *by A', the table size* $|\mathcal{D}|$ *by N, and the sample size* $|\mathcal{S}|$ *by n. Then, we can obtain the estimated answer to q:*

$$\hat{q}(\mathcal{S}) = \texttt{SELECT } N \cdot \frac{\texttt{SUM}(A')}{n} \texttt{ FROM } \mathcal{S}$$

Based on the Central Limit Theorem, we can derive that the closed-form confidence interval of the query is $\hat{q} \pm \epsilon$*, where*

$$\epsilon = \texttt{SELECT } \lambda \cdot N\sqrt{\frac{\texttt{VAR}(A')}{n}} \texttt{ FROM } S$$

Here λ *is a parameter determined by a confidence level. For example,* $\lambda = 1.96$ *for a 95% confidence interval and* $\lambda = 2.576$ *for a 99% confidence interval.*

For more complex queries, it may be very hard to get a closed-form confidence interval, thus AQP often computes an empirical confidence interval using bootstrap in this situation. The approach generates a set of resamples, $\mathcal{S}_1, \mathcal{S}_2, \cdots, \mathcal{S}_m$, of the original sample $\mathcal{S}$, and estimates the query answer using each resample. The obtained answers, $\hat{q}(\mathcal{S}_1), \hat{q}(\mathcal{S}_2), \cdots, \hat{q}(\mathcal{S}_m)$, form an estimate of the distribution of $q(\mathcal{D})$, from which we can compute a confidence interval. This is a more general approach, but a naive implementation will suffer from high computational cost because it needs to generate m resamples and then run queries on each of the resamples. Some sophisticated approaches were proposed to overcome the limitation [60, 72].

4.2 AQP++ Framework

The AQP++ framework is tailored to connect AQP with AggPre for interactive analytics. In this section, we first answer two fundamental questions about AQP++: (1) *How does AQP++ estimate a query result?* (2) *How does AQP++ compute a confidence interval?*

4.2.1 Result Estimation. Unlike AQP that uses a sample to *directly* estimate the answer to a user query (see Equation 3), AQP++ seeks to use a sample to estimate the difference of the query result from a precomputed aggregate value. Let q denote a *user query* and pre denote a *precomputed aggregate query*.

q: SELECT $f(A)$ FROM $\mathcal{D}$ WHERE Condition_1
pre: SELECT $f(A)$ FROM $\mathcal{D}$ WHERE Condition_2

AQP++ estimates the difference of their query results as follows:

$$q(\mathcal{D}) - pre(\mathcal{D}) \approx \hat{q}(\mathcal{S}) - \hat{pre}(\mathcal{S})$$

Since $pre(\mathcal{D})$ has been precomputed (i.e., a constant), we have

$$q(\mathcal{D}) \approx pre(\mathcal{D}) + \left(\hat{q}(\mathcal{S}) - \hat{pre}(\mathcal{S})\right) \tag{4}$$

To compute Equation 4, AQP++ first employs AQP (Equation 3) to estimate the user query's answer $\hat{q}(\mathcal{S})$, and then estimate the precomputed query's answer $\hat{pre}(\mathcal{S})$. Finally, it plugs the two values into Equation 4 to obtain the final estimated answer.

Example 2. *This example illustrates how to use AQP++ to estimate the answer to the same query as Example 1:*

$$q(\mathcal{D}) = \texttt{SELECT SUM}(A) \texttt{ FROM } \mathcal{D} \texttt{ WHERE } C \geq 0$$

Suppose the following aggregate query has been precomputed:

$$pre(\mathcal{D}) = \texttt{SELECT SUM}(A) \texttt{ FROM } \mathcal{D} \texttt{ WHERE } C > 0$$

AQP++ first uses AQP (Equation 3) to estimate the answers to q and pre, respectively.

$$\hat{q}(\mathcal{S}) = \texttt{SELECT } N \cdot \frac{\texttt{SUM}(A \cdot \text{cond}(C \geq 0))}{n} \texttt{ FROM } \mathcal{S}$$
$$\hat{pre}(\mathcal{S}) = \texttt{SELECT } N \cdot \frac{\texttt{SUM}(A \cdot \text{cond}(C > 0))}{n} \texttt{ FROM } \mathcal{S}$$

Then, it plugs $pre(\mathcal{D})$*,* $\hat{q}(\mathcal{S})$*, and* $\hat{pre}(\mathcal{S})$ *into Equation 4 to get the estimated answer to q.*

Unification. AQP++ connects AQP with AggPre using Equation 4. Interestingly, the equation shows that AQP and AggPre are only the special cases of AQP++ when no aggregate is precomputed and all aggregates are precomputed, respectively.

(1) *AQP++ subsumes AQP.* Define ϕ as an *empty query*, whose condition is always false.

$$\phi = \texttt{SELECT SUM}(A) \texttt{ FROM } \mathcal{D} \texttt{ WHERE False}^4.$$

Suppose no aggregate is precomputed, i.e., $pre = \phi$. In this case, we have $pre(\mathcal{D}) = \hat{pre}(\mathcal{S}) = 0$. Thus, Equation 4 is reduced to $q(D) \approx \hat{q}(S)$, which returns the same result as AQP.

(2) *AQP++ subsumes AggPre.* Suppose all aggregates are precomputed, i.e., $pre = q$. In this case, we have $\hat{pre}(\mathcal{S}) = \hat{q}(\mathcal{S})$. Thus, Equation 4 is reduced to $q(\mathcal{D}) \approx pre(\mathcal{D})$, which returns the same result as AggPre.

Generality. The AQP++ framework is very general. Firstly, it works for any aggregate function that AQP can support such as SUM, COUNT, AVG, and VAR, and some UDFs (see Lemma 1). This can be easily proved based on Equation 4 because $q(\mathcal{D})$ only depends on $pre(\mathcal{D})$, $\hat{q}(\mathcal{S})$, and $\hat{pre}(\mathcal{S})$, where $pre(\mathcal{D})$ has been precomputed, and $\hat{q}(\mathcal{S})$ and $\hat{pre}(\mathcal{S})$ can be obtained using AQP (since AQP can support their aggregation function).

Lemma 1. *For any aggregation function f, if AQP can answer the queries of the form:* `SELECT f(A) FROM D WHERE Condition`*, AQP++ can answer the queries as well.*

Proof. The proofs to the lemmas and theorems of this paper can be found in Appendix A. □

Furthermore, for any aggregate function, if AQP has an unbiased estimator, AQP++'s estimator is also unbiased.

Lemma 2. *For any aggregation function f, if AQP can unbiasedly estimate the queries of the form:* `SELECT f(A) FROM D WHERE Condition`*, the answers that AQP++ returns are also unbiased.*

Lastly, AQP++ can be easily extended to support many existing optimization techniques that were proposed for AQP, such as stratified sampling [3, 6, 8, 13–15, 27, 66] and auxiliary indices [13, 24]. These optimization techniques aim to improve the estimation quality of Equation 3. Since Equation 4 is obtained by treating Equation 3 as a black box, the optimization techniques work for AQP++ as well. For example, consider stratified sampling optimization. Unlike uniform sampling, where each row has the same probability to be sampled, stratified sampling divides data into a set of subgroups and tends to apply a higher sampling rate to smaller subgroups. This was shown to be an effective sampling approach to deal with skewed group-size distributions. Let S_{op} be a stratified sample of data. Then, we have an optimized AQP++ framework:

$$q(\mathcal{D}) \approx pre(\mathcal{D}) + \left(\hat{q}(\mathcal{S}_{op}) - \hat{pre}(\mathcal{S}_{op})\right), \tag{5}$$

where $\hat{q}(\mathcal{S}_{op})$ and $\hat{pre}(\mathcal{S}_{op})$ apply AQP to the stratified sample to estimate the answers to q and pre, respectively.

[4] Note that AQP++ also works when q and pre have different aggregation functions.

4.2.2 Confidence Interval. Like AQP, AQP++ also has two kinds of approaches to compute a confidence interval for an estimated answer. The analytical one needs to manually compute a closed-form confidence interval for each type of query. Example 3 illustrates how to compute a confidence interval for a SUM query.

Example 3. *Continuing Example 2, suppose we want to compute a confidence interval for* $pre(\mathcal{D}) + \left(\hat{q}(\mathcal{S}) - \hat{pre}(\mathcal{S})\right)$. *Since* $pre(\mathcal{D})$ *is a constant, we only need to compute the confidence interval for* $\hat{q}(\mathcal{S}) - \hat{pre}(\mathcal{S})$, *i.e.,*

$$\texttt{SELECT } N \cdot \frac{\texttt{SUM}(A \cdot \texttt{cond}(C=0))}{n} \texttt{ FROM } S.$$

Following the idea of Example 1, we obtain the confidence interval $\hat{q} \pm \epsilon$, *where*

$$\epsilon \texttt{ = SELECT } \lambda \cdot N\sqrt{\frac{\texttt{VAR}(\texttt{A} \cdot \texttt{cond}(C=0))}{n}} \texttt{ FROM } S.$$

For more complex queries, when it is hard to derive closed-form confidence intervals for them, AQP++ can use the bootstrap to compute their empirical confidence intervals. It first generates a set of resamples, $\mathcal{S}_1, \mathcal{S}_2, \cdots, \mathcal{S}_m$. Then, for each resample $\mathcal{S}_i$ ($i \in [1, m]$), it computes $pre(\mathcal{D}) + \left(\hat{q}(\mathcal{S}_i) - \hat{pre}(\mathcal{S}_i)\right)$. Please note that this step is different from AQP (which computes $\hat{q}(\mathcal{S}_i)$ instead). The computed results form an estimate of the distribution of $q(\mathcal{D})$, from which we derive a confidence interval.

Back of the envelope analysis. We investigate why AQP++ may produce more accurate answers (i.e., tighter confidence intervals) compared to AQP. One may be tempted to think that this is impossible because AQP++ has to estimate *two* random variables and then get their difference, which should have introduced more error than AQP who only needs to estimate *one*. However, this analysis ignores the correlation between the two random variables. For AQP++, the variance of its estimator (involving two random variables) is:

$$\text{Var}\left(\hat{q} - \hat{pre}\right) = \text{Var}(\hat{q}) + \text{Var}(\hat{pre}) - 2\text{Cov}(\hat{q}, \hat{pre})$$

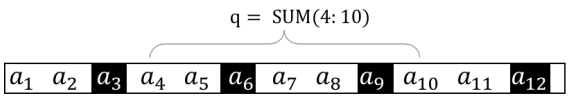
For AQP, the variance of its estimator is $\text{Var}(\hat{q})$. By comparing them, we can see that if $\text{Var}(\hat{pre}) < 2\text{Cov}(\hat{q}, \hat{pre})$, AQP++ can have a smaller variance than AQP. That is, AQP++ tends to return a more accurate result in this situation. To further elaborate on the situation, we know that $\text{Cov}(\hat{q}, \hat{pre})$ depends on the degree of the correlation between a user query result and a precomputed aggregate. If they are highly correlated, $\text{Cov}(\hat{q}, \hat{pre})$ becomes very large, thus AQP++ is more likely to return a more accurate answer. We also validate this interesting phenomenon in the experiments.

5 AGGREGATE IDENTIFICATION

With the new AQP++ framework, we now study the two problems defined in Section 3. This section studies the aggregate-identification problem. We first consider a simplified setting of the problem and present an optimal solution for it. We then extend the solution to the general setting.

5.1 Optimal Solution

We assume that (1) Q is one-dimensional (i.e., [SUM(A), C]); (2) A and C are independent. For ease of presentation, we denote a relational table by $\mathcal{D} = [a_1, a_2, \cdots, a_N]$, which is a list of attribute values of A ordered by C. Since A and C are independent, the list can be thought of as being randomly shuffled. Each user query is denoted by $\text{SUM}(x : y) = \sum_{x \le i \le y} a_i$. We denote a BP-Cube by $P = \{\text{SUM}(1 : t) = \sum_{1 \le i \le t} a_i \text{ for all } t \in \text{dom}(C)_{small}\}$, and we call each $t \in \text{dom}(C)_{small}$ *a partition point.*



P = {SUM(1: 3), SUM(1: 6), SUM(1: 9), SUM(1: 12)}

P^+ = {SUM(1: 3), SUM(1: 6), SUM(1: 9), SUM(1: 12), SUM(4: 6), SUM(4: 9), SUM(4: 12), SUM(7: 9), SUM(7: 12), SUM(10: 12), ∅}

P^- = {SUM(4: 9), SUM(4: 12), SUM(7: 9), SUM(7: 12), ∅}

Figure 2: An illustration of P, P^+, and P^- for the 1D case.

Given a user query q, to identify the best aggregate value for q, the brute-force approach needs to compute the query error w.r.t *every* $pre \in P^+$ and return the one with the minimum error. Since $|P^+|$ can be very large, this approach is prohibitively expensive. The key observation of our approach is that a user query can benefit more from a correlated aggregate query than a uncorrelated one. For example, consider a user query SUM(4 : 10). Suppose both SUM(4 : 9) and SUM(1 : 3) have been precomputed. Without the need to compute the actual query error w.r.t them, we can immediately deduce that SUM(4 : 9) is preferable because it is highly correlated to the user query while SUM(1 : 3) tells us nothing about the user query.

Based on this observation, given a user query $\text{SUM}(x : y)$, we can prove that only five aggregate values in P^+ need to be considered. We call them *candidate aggregate values*, denoted by

$$P^- = \Big\{\text{SUM}(l_x + 1 : l_y), \text{SUM}(l_x + 1 : h_y), \text{SUM}(h_x + 1 : l_y), \text{SUM}(h_x + 1 : h_y)\Big\} \cup \Big\{\phi\Big\}, \quad (6)$$

where l_x (h_x) is the first partition point that is lower (higher) than x; l_y (h_y) is the first partition point that is lower (higher) than y. Intuitively, x falls between partition points l_x and h_x, and y falls between partition points l_y and h_y. Based on the four partition points, we find the four most correlated aggregate queries to the user query. We also add ϕ into P^- to handle some special cases, e.g., x and y fall between the same two partition points (i.e., $l_x = l_y$). Lemma 3 proves the correctness of this solution.

Lemma 3. *Given* $\mathcal{D} = [a_1, \cdots, a_N]$, *a query template* Q, *and a BP-Cube* P, *we have:*

$$\min_{pre \in P^+} error(q, pre) = \min_{pre \in P^-} error(q, pre).$$

For example, consider the BP-Cube P and the available aggregate values P^+ in Figure 2. Given a user query $q = \text{SUM}(4 : 10)$, we want to identify the best value from P^+ for the query. Based on Lemma 3, we only need to consider five values. To get the five values, we can see that $x = 4$ falls between the precomputed points of 3 and 6, thus $l_x = 3$ and $h_x = 6$; $y = 10$ falls between the precomputed points of 9 and 12, thus $l_x = 9$ and $h_x = 12$. Based on Equation 6, we obtain $P^- = \{\text{SUM}(4 : 9), \text{SUM}(4 : 12), \text{SUM}(7 : 9), \text{SUM}(7 : 12), \phi\}$.

5.2 Aggregate-Identification Approach

Lemma 3 significantly reduces the number of aggregate values that need to be considered for answering a user query. While we can only prove its correctness under certain assumptions, it inspires us to develop an efficient heuristic approach for the general setting.

The key idea of our approach is to quickly identify a small number of candidate aggregate values from P^+ and then examine which one is the best among them. Similarly, we denote the candidate

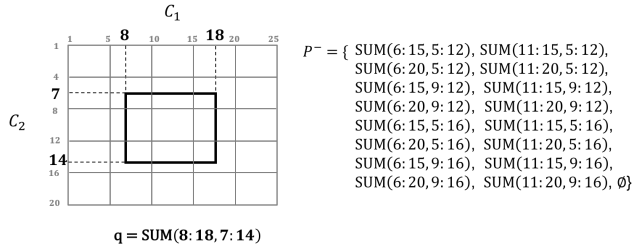


Figure 3: An illustration of P^- for the 2D case.

aggregate values by P^-. Equation 6 shows the computation of P^- for the 1D case. We now extend it to the d-dimensional case. Given a user query $q = \text{SUM}(x_1 : y_1, x_2 : y_2, \cdots, x_d : y_d)$, suppose that for each $i \in [1, d]$, x_i falls between l_{x_i} and h_{x_i}, and y_i falls between l_{y_i} and h_{y_i}. We define P^- w.r.t q as

$$P^- = \Big\{\text{SUM}(u_1 + 1 : v_1,\ u_2 + 1 : v_2,\ \cdots,\ u_d + 1 : v_d) \mid u_i \in \{l_{x_i}, h_{x_i}\}, v_i \in \{l_{y_i}, h_{y_i}\} \text{ for each } i \in [1, d]\Big\} \cup \Big\{\phi\Big\} \quad (7)$$

For example, consider a 2D BP-Cube in Figure 3. The first dimension has the partition points of $\text{dom}(C_1)_{small} = \{1, 5, 10, 15, 20, 25\}$; the second dimension has the partition points of $\text{dom}(C_2)_{small} = \{1, 4, 8, 12, 16, 20\}$. Given a user query $q = \text{SUM}(8 : 18, 7 : 14)$, for the first dimension C_1, we can see that $x_1 = 8$ falls between 5 and 10, thus $l_{x_1} = 5$ and $h_{x_1} = 10$; $y_1 = 18$ falls between 15 and 20, thus $l_{y_1} = 15$ and $h_{y_1} = 20$. Similarly, for the second dimension, we have $l_{x_2} = 4$ and $h_{x_2} = 8$ for $x_2 = 7$; we have $l_{y_2} = 12$ and $h_{y_2} = 16$ for $y_2 = 14$. Based on Equation 7, we obtain P^- consisting of the 17 aggregate values shown on the right side of the figure.

The size of P^- is independent of the BP-Cube size. For each dimension, there are four possible cases, i.e., $\{l_{x_i}, h_{x_i}\} \times \{l_{y_i}, h_{y_i}\}$. Thus, we have $|P^-| = 4^d + 1$. For example, $|P^-| = 4^1 + 1 = 5$ for the 1D case and $|P^-| = 4^2 + 1 = 17$ for the 2D case. Furthermore, we can quickly identify P^-. Let $k_i = |\text{dom}(C_i)_{small}|$. Every dimension only needs $O(\log k_i)$ time to search for the partition points, thus all dimensions need $O(\sum_i \log(k_i)) = O(\log k)$ time. To generate $4^d + 1$ queries based on the partition points, we need $O(4^d)$ time. Thus, the total time complexity is $O(\log k + 4^d)$.

Once P^- is obtained, we then need to decide which one in P^- should be finally identified. We adopt a subsampling based approach. The idea is to create a subsample of $\mathcal{S}$, and use the subsample to estimate the query error w.r.t. each aggregate value in P^-. Specifically, given a user query, for each precomputed aggregate value $pre \in P^-$, we compute the confidence interval of the user query w.r.t. pre and select the one with the smallest confidence interval. The subsampling rate is a parameter that can balance the trade-off between the effectiveness and efficiency of aggregate identification. In the experiments, we set it to less than $\frac{1}{4^d}$ to ensure that the overhead added is smaller than the actual query processing time.

6 AGGREGATE PRECOMPUTATION

We now study the aggregate-precomputation problem. There are two major challenges. One is how to determine the BP-Cube's shape. That is, given a threshold k, we need to assign a number k_i to each dimension such that $\prod_{i=1}^{d} k_i \le k$. The total number of possible assignments can be quite large. The other challenge is how to decide which k_i points should be chosen from $\text{dom}(C_i)$ for each $i \in [1, d]$. Again, there are a large number of different choices, $\binom{|\text{dom}(C_i)|}{k_i}$, and a brute-force approach does not work.

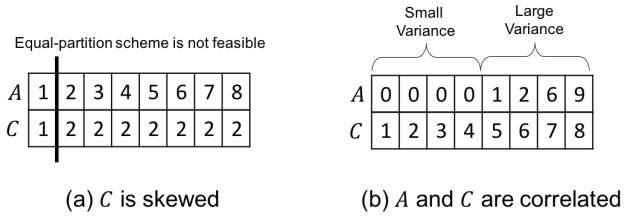


Figure 4: (a) The equal-partition scheme is not feasible; (b) The equal-partition scheme is not optimal.

In this section, we first explore the 1-dimensional case and then extend to multidimensional cases.

6.1 One-Dimensional Query Template

For the one-dimensional case, since the BP-Cube's shape has only one possibility, the only challenge left is how to choose the best k points, $1 \le t_1 < t_2 < \cdots < t_k = |\text{dom}(C)|$, from $\text{dom}(C)$ such that the query-template error is minimized[5]. The most natural idea, called *equal-partition scheme*, is to partition $\text{dom}(C)$ into k equal parts. But, this idea ignores two important factors.

- *Data Distribution.* If C does not follow a uniform distribution (i.e., some values appear more frequently than others), the equal-partition scheme is often not feasible. This is, we cannot use range queries with conditions over C to partition A equally. For example, consider the relational table (with attributes A and C) in Figure 4(a). The only way to partition the data is shown in the figure, which is not an equal-partition scheme.
- *Attribute Correlation.* If A and C are correlated, when sorting A by C, this process is not equivalent to a random shuffle of A. Thus, the variances of different parts of A may differ a lot. For example, consider the relational table in Figure 4(b). Suppose when $1 \le C \le 4$, A is always equal to 0; when $5 \le C \le 8$, A follows a normal distribution with a large variance. Since a larger variance leads to a higher query error, it might be better to choose more points from the second half of A rather than adopt an equal-partition scheme.

In the following, we first make some assumptions about data distribution and attribute correlation, and prove that the equal-partition scheme is optimal under these assumptions. Then, we relax the assumptions and propose an adaptive approach for the general setting.

6.1.1 Optimal Partition Scheme. We assume that (1) C has no duplicate values; (2) A and C are independent. The first (resp. second) assumption removes the impact of the data distribution of C (resp. the correlation between attributes A and C) on the optimal partition scheme. Similar to Section 5.1, we denote a relational table by $\mathcal{D} = \{a_1, a_2, \cdots, a_N\}$, which is the list of attribute values of A ordered by C. Assumption 1 suggests that any sub-list of $\mathcal{D}$ can be precomputed (without being constrained by the skewed distribution of C); Assumption 2 means that $\mathcal{D}$ can be thought of as being randomly shuffled.

We can prove that the equal-partition scheme is optimal under Assumptions 1 and 2. The corresponding BP-Cube is denoted by[6]

$$P_{eq} = \{\text{SUM}(1 : \frac{i}{k}N) \mid i = 1, 2, \cdots k\}.$$

[5]We compute the sum of all the values in each aggregation attribute because these sum values are independent of condition attributes and can be reused across query templates. Therefore, we assume $t_k = |\text{dom}(C)|$

[6]Here, we assume $N\%k = 0$. Otherwise, we will choose $\lceil \frac{i}{k}N \rceil$ for $i \in [1, N\%k]$, and $\lfloor \frac{i}{k}N \rfloor$ for $i \in (N\%k, k]$. The proof can be extended to this case.

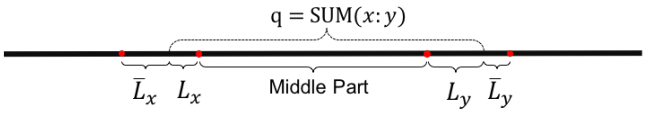


Figure 5: An illustration for notations of $L_x, \bar{L}_x, L_y$, and $\bar{L}_y$ (q is a user query and red dots represent partition points).

The proof's basic idea is that in Lemma 4, we compute the query-template error, $error(Q, P_{eq})$, w.r.t. the equal-partition scheme; in Lemma 5, we prove that for any other partition scheme, the resulting query-template error cannot be smaller than $error(Q, P_{eq})$. Hence, P_{eq} is optimal since it has the minimum query-template error.

Lemma 4. *Given $\mathcal{D} = \{a_1, a_2, \cdots, a_N\}$, a query template Q, and a threshold k, the query-template error of Q w.r.t P_{eq} is*

$$error(Q, P_{eq}) = \lambda N \sqrt{\frac{\sigma_{eq}^2}{n}},$$

where $\sigma_{eq}^2 = \frac{1}{k} E[\mathcal{D}^2] - \frac{1}{k^2}(E[\mathcal{D}])^2$.

Lemma 4 indicates that the query-template error decreases at a rate of $O(\frac{1}{\sqrt{k}})$. This is a very interesting result because it shows that with only a small k (i.e., the BP-Cube size), the query-template error can be dramatically reduced. For example, if $k = 100$, the error can be reduced by about 10 times.

Lemma 5 proves that $error(Q, P_{eq})$ is minimum.

Lemma 5. *Given $\mathcal{D} = \{a_1, a_2, \cdots, a_N\}$, a query template Q, and a threshold k, if $P \neq P_{eq}$, then $error(Q, P) \geq error(Q, P_{eq})$.*

It is easy to prove Theorem 1 based on Lemmas 4 and 5.

Theorem 1. *Given $\mathcal{D} = \{a_1, a_2, \cdots, a_N\}$, a query template Q, and a threshold k, P_{eq} is an optimal BP-Cube.*

For example, suppose $\mathcal{D} = \{a_1, a_2, \cdots, a_{12}\}$ and $k = 4$. Based on Theorem 1, we obtain the optimal BP-Cube $P_{eq} = \{\text{SUM}(1:3), \text{SUM}(1:6), \text{SUM}(1:9), \text{SUM}(1:12)\}$.

6.1.2 An Adaptive Approach Based on Hill Climbing. The optimal partition scheme requires two assumptions which may not hold in practice. In this section, we propose a hill-climbing based algorithm that can adaptively adjust the partition scheme based on the actual data distribution and attribute correlation.

Algorithm Overview. The algorithm starts with an initial BP-Cube and then attempts to improve it by moving a *single* partition point from one place to another. If the change leads to a better BP-Cube, the change is made and the iterative process is repeated; otherwise, the algorithm is terminated. To make the algorithm work, we need to address three problems: (1) how to find an initial BP-Cube; (2) how to evaluate the effectiveness of a BP-Cube (in order to know whether the change leads to a better BP-Cube); (3) how to adjust a BP-Cube (i.e., decide which partition point should be moved away and where it should move to).

(1) Initialization. A poor initialization may not only hurt the efficiency of an optimization algorithm, but also lead to a local optimum that is far from the global optimum. We use P_{eq} as an initialization because (1) it has been proved to be optimal in some situations and (2) it avoids ending up with a solution that is even worse than the naive equal partitioning. However, P_{eq} may not be always feasible (due to the skewed distribution of C). If a partition point (in P_{eq}) is not feasible, we will choose its closest feasible point to replace it. For example, in Figure 4(a), since the middle point (4th point) is not feasible, its closest feasible partition point (6th point) will be chosen. Accordingly, the initial BP-Cube is $P = \{\text{SUM}(1:2), \text{SUM}(1:3)\}$.

(2) Evaluation. The most naive way to evaluate the effectiveness of a BP-Cube is to adopt query-template error because this is the ultimate optimization objective. But, when the assumption that A and C are independent does not hold, we have not found an efficient way to compute it without enumerating $\binom{|\text{dom}(C)|+1}{2}$ possible user queries. To address this challenge, we seek to find an upper bound of query-template error. It turns out that the upper bound can not only be efficiently computed (in linear time) but also lead to a robust solution (since it bounds the worst case).

Recall that the query-template error is defined as $error(Q, P) = \max_{q \in Q} error(q, P)$. We first give the upper bound of $error(q, P)$, and then we present an efficient linear algorithm to compute the upper bound of $error(Q, P)$.

To get the upper bound of $error(q, P)$, consider a user query $q = \text{SUM}(x : y)$ in Figure 5. The red dots are the partition points near by x or y . We can see that the middle part of q has been precomputed, so we only need to estimate $L_x + L_y$. For L_x, since $L_x + \bar{L}_x$ has been precomputed, we can estimate L_x in two ways. One is to directly estimate L_x and the other is to estimate the complement $\bar{L}_x$. We try both ways and choose the one with a smaller error, i.e., $\min\left\{\frac{\lambda N}{\sqrt{n}} \cdot \sqrt{\text{Var}(A_{L_x})}, \frac{\lambda N}{\sqrt{n}} \cdot \sqrt{\text{Var}(A_{\bar{L}_x})}\right\}$, where $A_{L_x} = A \cdot \text{cond}(C \in L_x)$ and $A_{\bar{L}_x} = A \cdot \text{cond}(C \in \bar{L}_x)$. Similarly, for L_y, the estimation error is $\min\left\{\frac{\lambda N}{\sqrt{n}} \cdot \sqrt{\text{Var}(A_{L_y})}, \frac{\lambda N}{\sqrt{n}} \cdot \sqrt{\text{Var}(A_{\bar{L}_y})}\right\}$. By adding them up, we obtain the upper bound of $error(q, P)$.

Lemma 6. *Given $\mathcal{D}$, a query template Q, and a BP-Cube P, for any $q \in Q$, we have $error(q, P) \leq$*

$$\frac{\lambda N}{\sqrt{n}} \cdot \min\left\{\sqrt{Var(A_{L_x})}, \sqrt{Var(A_{\bar{L}_x})}\right\} + \frac{\lambda N}{\sqrt{n}} \cdot \min\left\{\sqrt{Var(A_{L_y})}, \sqrt{Var(A_{\bar{L}_y})}\right\}.$$

To get the upper bound of $error(Q, P)$, we first compute $error_i = \frac{\lambda N}{\sqrt{n}} \min\left\{\sqrt{\text{Var}(A_{L_i})}, \sqrt{\text{Var}(A_{\bar{L}_i})}\right\}$ for every point $i \in [1, N]$, and then pick up two points $i_1, i_2 \in [1, N]$ where $error_{i_1}$ has the maximum error and $error_{i_2}$ has the second maximum error. Note that all of these can be computed in linear time. Based on Lemma 6, we can deduce that $error(Q, P)$ cannot be larger than $error_{up}(Q, P) = error_{i_1} + error_{i_2}$. Our hill-climbing algorithm uses the upper bound, $error_{up}(Q, P)$, to evaluate the effectiveness of a BP-Cube P.

(3) Adjustment. To adjust the current BP-Cube, we try to move a single partition point from one place to another. The heuristic is to move the partition point to either i_1 or i_2. This is because that the goal is to reduce $error_{up}(Q, P)$ and moving to i_1 (or i_2) is very likely to reduce $error_{up}(Q, P)$. In order to decide which partition point should be moved away, we want to find the one such that moving it away will have the least *chance* to increase $error_{up}(Q, P)$. Imagine a partition point t is moved away. Only the points between the two partition points nearby t may have a larger $error_i$. Thus, the *chance* that $error_{up}(Q, P)$ will increase depends on the maximum error among the changed points. For example, suppose $P = \{\text{SUM}(1:3), \text{SUM}(1:6), \text{SUM}(1:9), \text{SUM}(1:12)\}$. There are four partition points: 3, 6, 9, and 12. If the partition point $t = 6$ is moved away, only the points in $(3, 9)$ may have a larger $error_i$, and for the others, $error_i$ keeps unchanged. We compute the maximum $error_i$ among the changed points, i.e., $\max_{i \in (3,9)} error_i$ (after moving 6). Similarly, we compute the maximum error for the

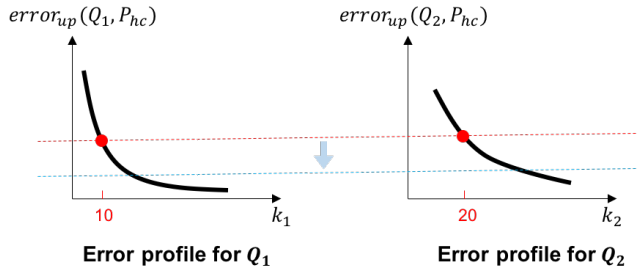


Figure 6: An illustration of the binary search algorithm to search for the BP-Cube's shape $k_1 \times k_2$ (suppose $k = 500$).

other three partition points, i.e., $\max_{i\in(1,6)} error_i$ (after moving 3), $\max_{i\in(6,12)} error_i$ (after moving 9), $\max_{i\in(9,12)}$ (after moving 12). Suppose $\max_{i\in(6,12)} error_i$ is minimal among the four values. Then, the partition point 9 will be moved away.

(4) Stop Condition. The hill-climbing algorithm will stop when $error_{up}(Q, P)$ cannot be decreased through the adjustment process.

Remark. Intuitively, there are two places that could cause the greedy approach to be not optimal. First, the approach aims to optimize the upper bound of the query-template error rather than the query-template error itself. Second, when putting a partition point to a new position, this new position (i.e., i_1 or i_2) is selected heuristically, which may not be the optimal position.

6.2 Multidimensional Query Template

Consider a query template, $Q : [\text{SUM}(A), C_1, C_2, \cdots, C_d]$. Given a threshold k, we need to first assign k_i to each dimension C_i such that $\prod_{i=1}^{d} k_i \le k$. Once $k_1, k_2, \cdots, k_d$ are determined, we apply the above hill climbing algorithm to choosing the k_i partition points in each dimension.

Determine the BP-Cube's Shape. Without loss of generality, we use a 2-dimensional query template $Q : [\text{SUM}(A), C_1, C_2]$ to illustrate our idea. A naive solution is to assign the same value to C_1 and C_2, i.e., $k_1 = k_2 = \sqrt{k}$. But, this ignores the fact that the data distributions in C_1 and C_2 can be quite different. To better balance the values of k_1 and k_2, we first plot an *error profile* for $Q_1 : [\text{SUM}(A), C_1]$ and $Q_2 : [\text{SUM}(A), C_2]$, respectively. The error profile shows how $error_{up}(Q_i, P_{hc})$ decreases with the increase of k_i for $i = 1, 2$, where P_{hc} is the BP-Cube determined by the hill-climbing algorithm. Since it is expensive to compute every data point on the profile curves, we compute a small subset of them and approximate the remaining ones by interpolation. The function used for the interpolation is proportional to $\sim 1/\sqrt{k}$ (see Lemma 4). Once the error profiles are plotted, k_1 and k_2 can be efficiently determined. For example, consider the two error profiles in Figure 6. Suppose k = 500. We do a binary search on the y-axis of the error profiles. At each iteration, if $k_1 \cdot k_2 < k$ (or $k_1 \cdot k_2 > k$), it continues the search in the lower (or upper) half of the search range, eliminating the other half from consideration. In the example, suppose the red line is the current search position. Then, we obtain $k_1 = 10$ and $k_2 = 20$ from the error profiles. Since $k = 500$ and $k_1 \times k_2 = 200 < k$, it means that we have the additional budget to enlarge k_1 and k_2 for getting a smaller error. Thus, the next search position is the lower half of the search range (i.e., the blue line). The binary search repeats until $k_1 \times k_2 = k$ or the search range is empty.

Putting It All Together. Given a query template $Q = [\text{SUM}(A), C_1, C_2, \cdots, C_d]$ and a threshold k, the aggregate-precomputation contains two stages. The first stage is to determine which BP-Cube should be precomputed and the second stage is to precompute the BP-Cube. Please note that the first stage is based on a sample S. It consists of two steps: (1) Determine the BP-Cube's shape, $k_1 \times k_2 \times \cdots \times k_d$; (2) Run the hill-climbing based algorithm to get k_i partition points from each dimension ($i \in [1, d]$). In the second stage, we need to scan the full data. Ho et al. [34] proposed an efficient algorithm to compute a BP-Cube. Since a BP-Cube is typically several orders of magnitude smaller than the P-Cube, it incurs much less preprocessing cost (see B for a detailed cost analysis).

Extensions. We discuss how to extend AQP++ to handle other aggregation functions, group-by queries, data updates, multiple query templates, and space allocation in Appendix C.

7 EXPERIMENTAL RESULTS

We conduct extensive experiments to evaluate AQP++. The experiments aim to answer three major questions. (1) When does AQP++ give more accurate answers than AQP? (2) How does AQP++ compare with AQP and AggPre in terms of preprocessing cost, response time, and answer quality? (3) How well does the hill climbing based approach perform compared to the equal-partition scheme?

7.1 Experiment Setup

Experimental Settings. We implemented AQP and AQP++ using DBX, a commercial OLAP system with column-store indexes supported. The code was written in C++, compiled using Visual Studio 2015, and connected to DBX through ODBC. The experiments were run on a Windows machine with an Intel Core 8 i7-6700 3.40GHz processor, 16GB of RAM, and 1TB HDD.

Datasets. We conducted experiments on three datasets. (1) *TPCD-Skew* is a synthetic dataset generated from the TPCD-Skew benchmark [18]. We generated 100GB data with the skew parameter $z = 2$, and ran queries on the lineitem table, which contains 600 million rows. (2) *BigBench* is a synthetic dataset generated from the Big Data Benchmark [1]. We generated 100GB data, and ran queries on the UserVisits table, which contains 752 million rows. (3) *TLCTrip* is a real-world dataset from the NYC Taxi and Limousine Commission [2]. We used the yellow car data from year 2009 to 2016, which is of size 200GB and contains 1400 million rows.

Sampling. It is worth noting that AQP++ is a general framework that can connect any AQP engine with AggPre no matter which sampling approach the AQP engine adopts. To allow for a clear comparison between the cores of AQP and AQP++ frameworks (i.e., Equation 3 vs. Equation 4), we assume that both AQP and AQP++ only use a uniform sample by default. Specifically, we create a uniform sample from the full table and store the sample into DBX as a table (sample rate = 0.05% by default). The sample will be used by AQP and AQP++ to answer queries. To evaluate the performance of AQP++ on other forms of samples, we implemented another two sampling approaches, measure-biased sampling [24] and stratified sampling [6], used by the state-of-the-art AQP systems, and compared AQP with AQP++ on these samples.

Error Metrics. We adopted *relative error* to quantify query accuracy because it is easy to interpret. For an approximate query result $\hat{q} \pm \epsilon$, where ϵ is half the width of the 95% confidence interval, the relative error of the query is defined as $\frac{\epsilon}{q}$, where q is the true answer of the query. Given a collection of queries, when we say

Table 1: Comparison of the overall performance (TPCD-Skew 100GB, k=50000, 0.05% uniform sample).

	Preprocessing Cost		Response	Answer Quality	
	Space	Time	Time	Avg Err.	Mdn Err.
AQP	51.2 MB	4.3 min	0.60 sec	2.67%	2.48%
AggPre	> 10 TB	> 1 day	< 0.01 sec	0.00%	0.00%
AQP++	51.9 MB	11.7 min	0.67 sec	0.27%	0.19%

median (or average) error, it refers to the median (or average) value of the relative errors of the queries in the collection.

7.2 Overall Performance

Table 1 compares the overall performance (preprocessing cost, query response time, and answer quality) of AQP++ with alternative approaches on the TPCD-Skew dataset. We randomly generated 1000 queries using the template of [SUM(l_extendedprice), l_orderkey, l_suppkey], where the selectivity of each query is between $0.5\% - 5\%$.

Suppose the latency requirement is 1 second. We first examine whether the state-of-the-art OLAP solutions can meet the requirement. We first chose a commercial (M)OLAP system and created a data cube with l_extendedprice as the measure attribute, and l_orderkey and l_suppkey as the dimension attributes. The cube has a hierarchy structure of "l_suppkey →l_orderkey". We ran the 1000 queries over the cube and found that the cube size was around 4GB and the average query response time was more than 10 seconds, which is far from interactive. The reason is that the two dimensions, $\langle l\_orderkey, l\_suppkey \rangle$, have a large number of distinct values (i.e., 377 million), thus a range query still needs to scan a lot of cells in the cube. In addition, we tested the time of directly executing queries in DBX. DBX needed an average response time of 6 seconds and a maximum response time of 35 seconds to run all the queries, which did not meet the latency requirement either.

We now evaluate the performance of AQP, AggPre, and AQP++. AQP used a uniform sample to answer queries (sample rate = 0.05%); AggPre precomputed the complete P-Cube using the algorithm in [34]. In the lineitem table, l_orderkey and l_suppkey have 1.5×10^8 and 7.5×10^4 distinct values, respectively, so there are 1.1×10^{13} cells in P-Cube. Clearly, AQP and AggPre represented two extreme cases of AQP++, where one did not precompute any aggregate and the other precomputed all possible aggregates. For AQP++, we used the same sample as AQP, and precomputed a BP-Cube of size $k = 50,000$.

Table 1 compared its performance with AQP and AggPre. We can see that all of them met the latency requirement (< 1 sec), but they were quite different in terms of preprocessing cost and answer quality. AQP++ spent orders of magnitude less preprocessing time and space than AggPre since it only needs to precompute a small BP-Cube rather than the complete P-Cube. In comparison with AQP, AQP++ reduced the average error by 10× and the median error by 13× for almost the same preprocessing space and about 7.4 minutes more preprocessing time. Furthermore, the overhead added to the AQP++'s response time (due to the aggregate-identification step) is negligible. This is because the response time was dominated by the I/O time for reading data from the sample table, and the added CPU time was relatively very small.

To further compare AQP and AQP++, we set the sample rate of AQP to 4% such that it can reach approximately the same average error as AQP++. We call it AQP(large). Compared to AQP++, the AQP(large)'s sample size was about 80× larger, which significantly increased the preprocessing time and space. Furthermore, due to the increase of the sample size, its query response time was more than 1 second, which violated the latency requirement.

We implemented APA+ [38] and compared its performance with AQP++. Since our query template was 2-dimensional, we assumed that *1-dimensional facts* (i.e., a set of statistics defined by APA+) are available in the system for each query. To process a query, APA+ first gets the related facts and then combines them with a sample to estimate the answer to the query. We used the gurobi library to solve the quadratic programming problem in APA+ such that it can minimize the estimation error. The experimental result showed that APA+ achieved an median error of 1.69% while the median error for AQP++ was only 0.19%. The reason is that APA+ does not use BP-Cubes for result estimation while AQP++ can identify the best BP-Cube to precompute and use it for result estimation.

7.3 Detailed Performance

In this section, we evaluate the performance of AQP++ by varying the number of dimensions and the set of condition attributes, aiming to gain a deeper understanding of various trade-offs. We also examine the effectiveness of the hill climbing algorithm for aggregate precomputation. If not specified, we use the same dataset and queries as the previous section, and set the sample rate to 0.05%, k=50000, and the number of dimensions to 2 by default.

Number of Dimensions. We compare the preprocessing time, response time, and answer quality of AQP and AQP++ by varying the number of dimensions. We chose ten columns from the lineitem table, and constructed ten query templates accordingly:

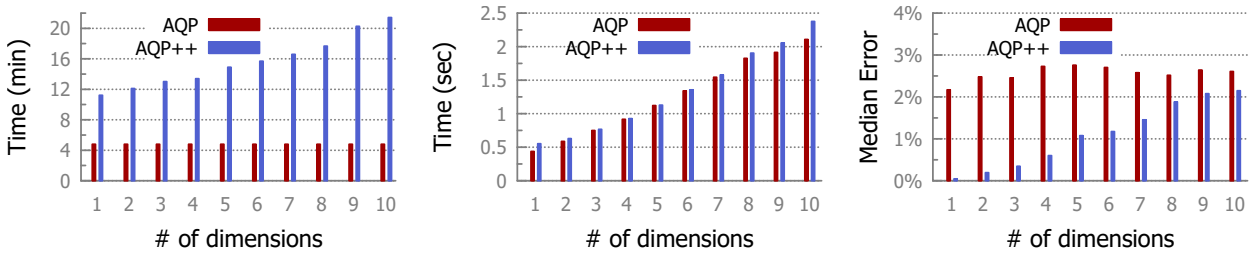
[SUM (l_extendedprice), l_orderkey],
[SUM (l_extendedprice), l_orderkey, l_partkey],
...
[SUM (l_extendedprice), l_orderkey, l_partkey, l_suppkey, l_linenumber, l_quantity, l_discount, l_tax, l_shipdate, l_commitdate, l_receiptdate],

where each of them has a different number of dimensions. We compared AQP with AQP++ w.r.t. each query template, and reported the result in Figure 7.

Figure 7(a) compares the preprocessing time of AQP and AQP++. Since AQP only needs to create a random sample, the number of dimensions had no impact on its preprocessing time. In comparison, AQP++ requires a little more preprocessing time when the number of dimensions increased since it needs to generates an error profile for each dimension. The larger the number of dimensions, the more time spent in generating error profiles.

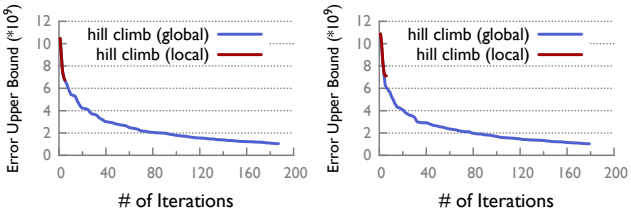
Figure 7(b) shows how the response time changed w.r.t. the number of dimensions. To identify the best precomputed aggregate value, AQP++ first generates $4^d + 1$ candidate values and then uses a subsample to estimate which one will lead to the smallest error. As the number of dimensions increased, the number of candidate values increased exponentially. However, we can see from the figure that the difference between the response times of AQP and AQP++ did not increase exponentially. This is because that if the number of candidate values is increased by 4 times, AQP++ will decrease the subsampling rate by 4 times as well, which helps to reduce the aggregate-identification time.

Figure 7(c) compares the answer quality between AQP and AQP++ in terms of median error. The figure shows that the median error of AQP++ got bigger as the number of dimensions increased. This is because that the space budget $k = 50000$ was fixed. If there is only



(a) Preprocessing Time (b) Response Time (c) Answer Quality

Figure 7: Comparison of the performance of AQP and AQP++ by varying the number of dimensions (TPCD-Skew 100GB, k=50000, 0.05% uniform sample).



(a) l_shipdate (b) l_commitdate

Figure 8: Evaluation of the adjustment approach in our hill climbing algorithm (TPCD-Skew 100GB, $k_1 = 200$, $k_2 = 200$, and 0.05% sample).

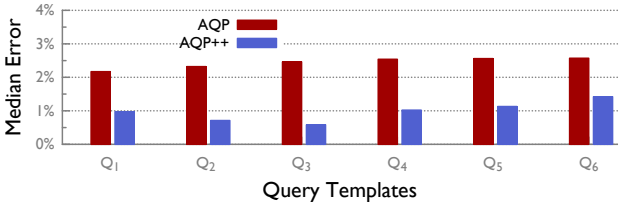


Figure 9: Evaluation of the changes of the set of condition attributes in user queries. Note that only Q_3 has a precomputed BP-Cube (TPCD-Skew 100GB, k=50000, 0.05% sample).

one dimension, this dimension can be assigned a budget of 50000 partition points, but if there are two dimensions, each dimension (on average) can only be assigned a budget of $\sqrt{50000} = 224$ partition points, which is less effective than 1D. Nevertheless, as shown in Figure 7(c), AQP++ outperformed AQP by 12.8× for 2D. As the number of dimensions increased, the improvement of AQP++ over AQP decreased. The result indicates that AQP++ can scale up to 10 dimensions but is hard to scale to a very large number of dimensions (e.g., 20) due to the limitation of prefix cubes.

Hill Climbing. We evaluate the adjustment approach of our hill climbing algorithm. Recall that at each iteration, our approach considers all partition points and picks up the best one to move. One may ask that why not only consider the four partition points next to i_1 and i_2. We compared the two adjustment approaches.

As discussed in Section 6, an equal partitioning scheme turns to be ineffective when attributes are highly correlated. Thus, we picked up two attributes, l_shipdate and l_commitdate, that have strong correlations with l_extendedprice, and constructed the query template: [SUM (l_extendedprice), l_shipdate, l_commitdate]. Figure 8 compares the upper bound of the query template error (i.e., $error_{up}(Q, P)$) of Hill Climb (global) and Hill Climb (local) on each dimension, where the former used our adjustment approach while the latter adopted the alternative. We set $k_1 = k_2 = 200$. We can see that Hill Climb (local) converged to a local optimum with less than 10 iterations while Hill Climb (global) can continue the iterative process and finally reached a much better result. The reason is that Hill Climb (local) only considers the four partition points next to i_1 and i_2. If moving them away cannot lead to a better solution, the algorithm will stop.

Changes of Condition Attributes. In exploratory workloads, a user may frequently change the set of condition attributes in her queries. We discuss how AQP++ handles this situation below.

Suppose a user may issue a collection of queries generated from the three query templates: Q_1 : [SUM(A), C_1], Q_2 : [SUM(A), C_1, C_2], Q_3 : [SUM(A), C_1, C_2, C_3], but only Q_2 has a precomputed BP-Cube P_2. We next show how AQP++ can use P_2 to answer the queries from Q_1 and Q_3.

If a user query q is from Q_1, we can rewrite q as an equivalent query q' from Q_2 where q' does not enforce any restriction on C_2, thus AQP++ can still use P_2 to answer q. For example, consider q : [SUM(A), 1 : 2]. It can be rewritten as q' : [SUM(A), 1 : 2, 1 : |dom(C_2)|] (where C_2 can be any value in its domain), and then be processed by AQP++ using P_2.

If a user query q is from Q_3, we can consider P_2 as a 3-dimensional BP-Cube P_2'. For example, suppose the shape of P_2 is $k_1 \times k_2$. Then, it can be seen as a 3-dimensional BP-Cube P_2' with the shape of $k_1 \times k_2 \times 1$. Thus, AQP++ can still use P_2 to answer q.

To evaluate this approach, we constructed six query templates Q_1 : [SUM(A), C_1], $\cdots$, Q_6 : [SUM(A), C_1, C_2, C_3, C_4, C_5, C_6] on the TPCD-Skew dataset, where A is l_extendedprice, and $C_1, C_2, \cdots, C_6$ are l_orderkey, l_partkey, l_suppkey, l_linenumber, l_quantity, l_discount, respectively. We assumed that only Q_3 had a precomputed BP-Cube (with size $k = 50000$). We randomly generated 1000 queries from each query template with the selectivity of 0.5%-5%. Figure 9 compares the median error of AQP and AQP++ for these queries w.r.t. each Q_i ($i \in [1, 6]$). We can see that AQP++ kept outperforming AQP when changing the set of condition attributes from Q_3 to Q_1 or from Q_3 to Q_6, but with the improvement being smaller as more changes are made. An interesting future work is to study how to detect this situation and how to trigger the computation of more suitable BP-Cubes in an automatic way.

7.4 Evaluation With Other Sampling Methods

The previous experiments validate the effectiveness of AQP++ on uniform samples. In this section, we implement two other sampling approaches used by the state-of-the-art AQP systems [6, 24] and examine the performance of AQP++ on these samples.

AQP (measure-biased) vs. AQP++ (measure-biased). Measure-biased sampling selects each record with a probability proportional to the value in the measure attribute. That is, the larger the value in the measure attribute, the more likely the record

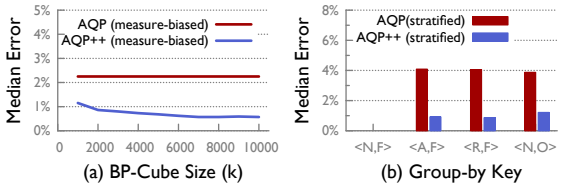


Figure 10: Comparing AQP++ with AQP using measure-based sampling and stratified sampling (TPCD-Skew 100GB).

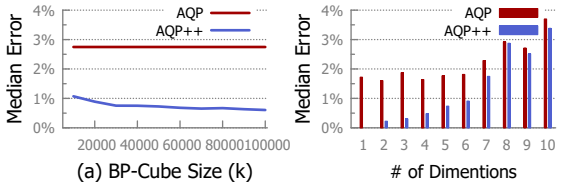


Figure 11: Comparing AQP++ with AQP on the BigBench (100 GB) and TLCTrip (200GB) datasets.

will appear in the sample. This has shown be to a very effective sampling approach to mitigate the negative impact of outliers on estimated answers. We randomly generated 1000 queries with the selectivity of $0.5\% - 5\%$ using the default template. Since measure-biased sampling is designed for handling outliers, we only chose the queries that can cover (at least) one outlier, where a value is defined as an outlier if l_extendedprice is larger than median(l_extendedprice) + 3 * SD(l_extendedprice). We created a 0.05% measure-biased sample of the dataset, leading to a sample of size $|S| = 0.3$ million, and then compared AQP with AQP++ on the sample by varying BP-Cube size from k=1000 to k=10,000. Figure 10(a) plots the median error. We can see that with a very small BP-Cube (e.g., k=5000), AQP++ reduced the median error of AQP by 3.3×, which validated the effectiveness of AQP++ for measure-biased sampling.

AQP (stratified) vs. AQP++ (stratified). Stratified sampling divides data into different groups and then applies a different sampling ratio to each group. The sampling ratio of each group is disproportional to its group size. This is to ensure that there are enough records being sampled from small groups. Since stratified sampling is designed for optimizing group-by queries, we randomly generated 1000 group-by queries of the following form:

```
SELECT SUM(l_extendedprice) FROM lineitem
WHERE  l_orderkey, l_suppkey
GROUP BY l_returnflag, l_linestatus,
```

where the selectivity of each query is between 0.5% - 5%. We then created a 0.05% stratified sample of the dataset w.r.t. the group-by attributes, l_returnflag and l_linestatus, leading to a sample of size $|S| = 0.3$ million. AQP used the sample to estimate the answers to the group-by queries; AQP++ used the same sample along with a small BP-Cube of size $k = 50,000$ to estimate the answers. Figure 10(b) reports the median error w.r.t. each group. We can see that AQP++ achieved $3\times - 4\times$ more accurate answers than AQP, which validated the effectiveness of AQP++ for stratified sampling. Interestingly, both AQP++ and AQP returned true answers for the group-by key of "<N,F>" because the group size was very small and all its records were included into the sample due to the use of stratified sampling.

7.5 Evaluation on More Datasets

We compared the performance of AQP++ and AQP on two other datasets, BigBench and TLCTrip.

For the BigBench dataset, we want to examine the performance of AQP++ for different BP-Cube size. We created a 0.05% uniform sample of the dataset, and randomly generated 1000 queries using the template of [SUM (adRevenue), visitDate, duration, sourceIP] with the selectivity of 0.5%-5%. Figure 11(a) compares the median error of AQP++ and AQP by varying k. We can see that even with a small BP-Cube, AQP++ can still outperform AQP by a lot. For example, when $k = 50,000$, AQP++ reduced the median error by 3.8×. As k grows, AQP++ can achieve better and better performance, finally reached a median error of 0.60% when $k = 100,000$.

For the TLCTrip dataset, we want to examine the performance of AQP++ for different number of dimensions. We chose ten columns from the table, and constructed ten query templates accordingly: [SUM (Distance), Pickup_Date], ⋯, [SUM (Distance),Pickup_Date, Pickup_Time,vendor_name,Fare_Amt,Rate_Code, Passenger_Count, Dropoff_Date, Dropoff_Time, surcharge, Tip_Amt]. We created a 0.1% uniform sample of the dataset; for each query template, we randomly generated 1000 queries with the selectivity of 0.5%-5%, and precomputed a BP-Cube of size $k = 300,000$ for it. Figure 11(b) compares the median error of AQP++ and AQP w.r.t. each query template. Similar to Figure 7(c), we found that AQP++ significantly outperformed AQP when the number of dimensions is small and marginally improved the median error of AQP when the number of dimensions was increased to 10.

8 CONCLUSION

In this paper, we studied how to enable database systems to answer aggregation queries within interactive response times. We found that the two separate ideas for interactive analytics, AQP and AggPre, can be connected together using the AQP++ framework. We presented the unification and generality of the framework, and demonstrated (analytically and empirically) why AQP++ can return a more accurate answer than AQP. After that, an in-depth study of the framework was conducted for range queries. In the study, we formally defined the aggregate-identification and aggregate-precomputation problems, and proposed both optimal solutions (under certain assumptions) as well as effective heuristic approaches (for general settings). We implemented AQP++ on a commercial OLAP system, and evaluated them on three datasets. Experimental results showed that AQP++ can improve the answer quality of AQP by up to 10× and reduce the preprocessing cost (both time and space) of AggPre by several orders of magnitude.

Our work is a first attempt to provide a general framework to connect AQP and AggPre together. Since both AQP and AggPre have been extensively studied in the past, we believe there are many future research directions to explore. First, various techniques have been proposed to optimize AQP (e.g., workload-driven sample creation) as well as AggPre (e.g., cube approximation). It would be interesting to revisit these techniques under the AQP++ framework. Second, there are some aggregation functions that AQP cannot handle well, such as min and max. However, they are easy for AggPre. Since AQP++ connects AQP with AggPre, it would be interesting to explore whether AQP++ can be extended to support these aggregation functions. Third, it might be hard for some users to decide which query templates should be specified. Thus, user-guided query template design is another interesting topic to explore.

REFERENCES

[1] Big Data Benchmark. http://amplab.cs.berkeley.edu/benchmark.
[2] TLC Trip Record Data. http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml.
[3] S. Acharya, P. B. Gibbons, and V. Poosala. Congressional samples for approximate answering of group-by queries. In *SIGMOD*, 2000.
[4] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. The aqua approximate query answering system. In *SIGMOD*, 1999.
[5] S. Agarwal, H. Milner, A. Kleiner, A. Talwalkar, M. I. Jordan, S. Madden, B. Mozafari, and I. Stoica. Knowing when you're wrong: building fast and reliable approximate query processing systems. In *SIGMOD*, 2014.
[6] 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*, 2013.
[7] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: relational data processing in spark. In *SIGMOD*, 2015.
[8] B. Babcock, S. Chaudhuri, and G. Das. Dynamic sample selection for approximate query processing. In *SIGMOD*, 2003.
[9] P. Bailis, J. M. Hellerstein, and M. Stonebraker. *Readings in Database Systems, 5th Edition*, chapter "Interactive Analytics". 2016.
[10] D. Barbará and M. Sullivan. Quasi-cubes: Exploiting approximations in multidimensional databases. *SIGMOD Record*, 1997.
[11] Y. Cao and W. Fan. Data driven approximation with bounded resources. *PVLDB*, 10(9):973–984, 2017.
[12] C. Y. Chan and Y. E. Ioannidis. Hierarchical prefix cubes for range-sum queries. In *VLDB*, 1999.
[13] S. Chaudhuri, G. Das, M. Datar, R. Motwani, and V. R. Narasayya. Overcoming limitations of sampling for aggregation queries. In *ICDE*, 2001.
[14] S. Chaudhuri, G. Das, and V. R. Narasayya. A robust, optimization-based approach for approximate answering of aggregate queries. In *SIGMOD*, 2001.
[15] S. Chaudhuri, G. Das, and V. R. Narasayya. Optimized stratified sampling for approximate query processing. *ACM Trans. Database Syst.*, 32(2):9, 2007.
[16] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Record*, 1997.
[17] S. Chaudhuri, B. Ding, and S. Kandula. Approximate query processing: No silver bullet. In *SIGMOD*, 2017.
[18] S. Chaudhuri and V. Narasayya. TPC-D data generation with skew. ftp.research.microsoft.com/users/viveknar/tpcdskew.
[19] Y. Chen and K. Yi. Two-level sampling for join size estimation. In *SIGMOD*, 2017.
[20] R. Chirkova and J. Yang. Materialized views. *Foundations and Trends in Databases*, 4(4):295–405, 2012.
[21] S. Chun, C. Chung, J. Lee, and S. Lee. Dynamic update cube for range-sum queries. In *VLDB*, 2001.
[22] S. Cohen, W. Nutt, and Y. Sagiv. Rewriting queries with arbitrary aggregation functions using views. *ACM Trans. Database Syst.*, 31(2):672–715, 2006.
[23] G. Cormode, M. N. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends in Databases*, 4(1-3):1–294, 2012.
[24] B. Ding, S. Huang, S. Chaudhuri, K. Chakrabarti, and C. Wang. Sample + Seek: approximating aggregates with distribution precision guarantee. In *SIGMOD*, pages 679–694, 2016.
[25] C. E. Dyreson. Information retrieval from an incomplete data cube. In *VLDB*, 1996.
[26] A. Galakatos, A. Crotty, E. Zgraggen, C. Binnig, and T. Kraska. Revisiting reuse for approximate query processing. *PVLDB*, 10(10):1142–1153, 2017.
[27] V. Ganti, M. Lee, and R. Ramakrishnan. ICICLES: self-tuning samples for approximate query answering. In *VLDB*, 2000.
[28] S. Geffner, D. Agrawal, A. El Abbadi, and T. R. Smith. Relative prefix sums: An efficient approach for querying dynamic OLAP data cubes. In *ICDE*, 1999.
[29] P. B. Gibbons and Y. Matias. New sampling-based summary statistics for improving approximate query answers. In *SIGMOD*, 1998.
[30] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub totals. *Data Min. Knowl. Discov.*, 1(1):29–53, 1997.
[31] A. Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4):270–294, 2001.
[32] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *SIGMOD*, 1996.
[33] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *SIGMOD*, 1997.
[34] C. Ho, R. Agrawal, N. Megiddo, and R. Srikant. Range queries in OLAP data cubes. In *SIGMOD*, 1997.
[35] C. Jermaine. Robust estimation with sampling and approximate pre-aggregation. In *VLDB*, pages 886–897, 2003.
[36] C. Jermaine and R. J. Miller. Approximate query answering in high-dimensional data cubes. In *SIGMOD*, 2000.
[37] C. M. Jermaine, S. Arumugam, A. Pol, and A. Dobra. Scalable approximate query processing with the DBO engine. In *SIGMOD*, 2007.
[38] R. Jin, L. Glimcher, C. Jermaine, and G. Agrawal. New sampling-based estimators for OLAP queries. In *ICDE*, 2006.
[39] S. Joshi and C. Jermaine. Materialized sample views for database approximation. In *ICDE*, 2006.
[40] N. Kamat, P. Jayachandran, K. Tunga, and A. Nandi. Distributed and interactive cube exploration. In *ICDE*, pages 472–483, 2014.
[41] N. Kamat and A. Nandi. A session-based approach to fast-but-approximate interactive data cube exploration. *ACM Trans. Knowl. Discov. Data*, 12(1):9:1–9:26, Feb. 2018.
[42] 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*, 2016.
[43] A. Kemper, T. Neumann, J. Finis, F. Funke, V. Leis, H. Mühe, T. Mühlbauer, and W. Rödiger. Processing in the hybrid OLTP & OLAP main-memory database system hyper. *IEEE Data Eng. Bull.*, 36(2):41–47, 2013.
[44] S. Krishnan, J. Wang, M. J. Franklin, K. Goldberg, and T. Kraska. Stale view cleaning: Getting fresh answers from stale materialized views. *PVLDB*, 8(12):1370–1381, 2015.
[45] F. Li, B. Wu, K. Yi, and Z. Zhao. Wander join: Online aggregation via random walks. In *SIGMOD*, 2016.
[46] X. Li, J. Han, Z. Yin, J. Lee, and Y. Sun. Sampling cube: a framework for statistical olap over sampling data. In *SIGMOD*, 2008.
[47] W. Liang, H. Wang, and M. E. Orlowska. Range queries in dynamic OLAP data cubes. *Data Knowl. Eng.*, 34(1):21–38, 2000.
[48] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. *PVLDB*, 3(1):330–339, 2010.
[49] G. Moerkotte. Small materialized aggregates: A light weight index structure for data warehousing. In *VLDB*, 1998.
[50] D. Moritz, D. Fisher, B. Ding, and C. Wang. Trust, but verify: Optimistic visualizations of approximate queries for exploring big data. In *CHI*, 2017.
[51] B. Mozafari and N. Niu. A handbook for building an approximate query engine. *IEEE Data Eng. Bull.*, 38(3):3–29, 2015.
[52] B. Mozafari, J. Ramnarayan, S. Menon, Y. Mahajan, S. Chakraborty, H. Bhanawat, and K. Bachhav. Snappydata: A unified cluster for streaming, transactions and interactice analytics. In *CIDR*, 2017.
[53] I. S. Mumick, D. Quass, and B. S. Mumick. Maintenance of data cubes and summary tables in a warehouse. In *SIGMOD*, 1997.
[54] A. Nandi, A. Fekete, and C. Binnig. HILDA 2016 workshop: A report. *IEEE Data Eng. Bull.*, 39(4):85–86, 2016.
[55] S. Nirkhiwale, A. Dobra, and C. M. Jermaine. A sampling algebra for aggregate estimation. *PVLDB*, 6(14):1798–1809, 2013.
[56] F. Olken. *Random sampling from databases.* PhD thesis, University of California at Berkeley, 1993.
[57] F. Olken and D. Rotem. Simple random sampling from relational databases. In *VLDB*, 1986.
[58] N. Pansare, V. R. Borkar, C. Jermaine, and T. Condie. Online aggregation for large mapreduce jobs. *PVLDB*, 4(11):1135–1145, 2011.
[59] Y. Park, A. S. Tajik, M. J. Cafarella, and B. Mozafari. Database learning: Toward a database that becomes smarter every time. In *SIGMOD*, pages 587–602, 2017.
[60] A. Pol and C. Jermaine. Relational confidence bounds are easy with the bootstrap. In *SIGMOD*, pages 587–598, 2005.
[61] N. Potti and J. M. Patel. DAQ: A new paradigm for approximate query processing. *PVLDB*, 8(9):898–909, 2015.
[62] F. Rusu, C. Qin, and M. Torres. Scalable analytics model calibration with online aggregation. *IEEE Data Eng. Bull.*, 38(3):30–43, 2015.
[63] F. Rusu, F. Xu, L. L. Perez, M. Wu, R. Jampani, C. Jermaine, and A. Dobra. The DBO database system. In *SIGMOD*, pages 1223–1226, 2008.
[64] A. Shukla, P. Deshpande, and J. F. Naughton. Materialized view selection for multidimensional datasets. In *VLDB*, 1998.
[65] A. Shukla, P. Deshpande, J. F. Naughton, and K. Ramasamy. Storage estimation for multidimensional aggregates in the presence of hierarchies. In *VLDB*, 1996.
[66] L. Sidirourgos, M. L. Kersten, and P. A. Boncz. SciBORQ: Scientific data management with bounds on runtime and quality. In *CIDR*, 2011.
[67] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-Store: A column-oriented DBMS. In *PVLDB*, 2005.
[68] J. S. Vitter and M. Wang. Approximate computation of multidimensional aggregates of sparse data using wavelets. In *SIGMOD*, 1999.
[69] J. Wang, S. Krishnan, M. J. Franklin, K. Goldberg, T. Kraska, and T. Milo. A sample-and-clean framework for fast and accurate query processing on dirty data. In *SIGMOD*, pages 469–480, 2014.
[70] S. Wu, B. C. Ooi, and K. Tan. Continuous sampling for online aggregation over multiple queries. In *SIGMOD*, 2010.
[71] K. Zeng, S. Agarwal, and I. Stoica. iOLAP: managing uncertainty for efficient incremental OLAP. In *SIGMOD*, 2016.
[72] K. Zeng, S. Gao, B. Mozafari, and C. Zaniolo. The analytical bootstrap: a new method for fast error estimation in approximate query processing. In *SIGMOD*, pages 277–288, 2014.

APPENDIX

A PROOFS

Proof of Lemma 1

Consider a user query q:

`SELECT` $f(A)$ `FROM` $\mathcal{D}$ `WHERE Condition_1`,

and a precomputed aggregate query pre:

`SELECT` $f(A)$ `FROM` $\mathcal{D}$ `WHERE Condition_2`.

If AQP supports the aggregation function f, query q and pre can be estimated using AQP, i.e., $\hat{q}(\mathcal{S})$ and $\hat{pre}(\mathcal{S})$. Since $pre(\mathcal{D})$ is a constant, AQP++ can use Equation 4 to get the estimatation of q.

Proof of Lemma 2

Consider a user query q:

`SELECT` $f(A)$ `FROM` $\mathcal{D}$ `WHERE Condition_1`,

and a precomputed aggregate query pre:

`SELECT` $f(A)$ `FROM` $\mathcal{D}$ `WHERE Condition_2`.

If AQP can estimate their answers unbiasedly, then we have $q(\mathcal{D}) = \mathbf{E}[\hat{q}(\mathcal{S})]$ and $pre(\mathcal{D}) = \mathbf{E}[\hat{pre}(\mathcal{S})]$. Based on Equation 4, AQP++'s estimator returns $pre(\mathcal{D}) + \left(\hat{q}(\mathcal{S}) - \hat{pre}(\mathcal{S})\right)$. We can prove that its expect value is equal to the true value:

$$\begin{aligned}
&\mathbf{E}\Big[pre(\mathcal{D}) + \left(\hat{q}(\mathcal{S}) - \hat{pre}(\mathcal{S})\right)\Big] \\
&= \mathbf{E}[pre(D)] + \left(\mathbf{E}[\hat{q}(\mathcal{S})] - \mathbf{E}[\hat{pre}(\mathcal{S})]\right) \\
&= pre(\mathcal{D}) + \left(q(\mathcal{D}) - pre(\mathcal{D})\right) \\
&= q(\mathcal{D})
\end{aligned}$$

Proof of Lemma 3

Consider a sequence of i.i.d values $\mathcal{D} = \{a_1, a_2, \cdots, a_n\}$. We define $\mathcal{D}_{pre} = \{G_{pre}(a_i) \mid i \in [1, n]\}$, where $G_{pre}(a_i)$ take a_i as input and returns a_i if a_i satisfies the pre's condition; otherwise, 0. We define $\mathcal{D}_q = \{G_q(a_i) \mid i \in [1, n]\}$, where $G_q(a_i)$ take a_i as input and returns a_i if a_i satisfies the q's condition; otherwise, 0. We define $\mathcal{D}_d = \mathcal{D}_{pre} - D_{pre} = \{x_i - y_i | x_i \in \mathcal{D}_{pre}, y_i \in \mathcal{D}_q, i \in [1, n].\}$ Based on the confidence interval for a SUM query in Example 3, we can deduce that $error(q, pre) = \lambda N\sqrt{\frac{\mathsf{VAR}(\mathcal{D}_d)}{n}}$. Since $\mathsf{VAR}(\mathcal{D}_d) = E[\mathcal{D}_d^2] - E[\mathcal{D}_d]^2$, let us compute $E[\mathcal{D}_d^2]$ and $E[\mathcal{D}_d]^2$ separately.

Firstly, we will compute $E[\mathcal{D}_d]^2$. We have $E[\mathcal{D}_d] = E[\mathcal{D}_{pre} - D_q] = E[\mathcal{D}_{pre}] - E[\mathcal{D}_q]$. Since each value in $\mathcal{D}$ is i.i.d, the sum of the values within any range is proportional to the length of the range. Let α be the percentage of the values in $\mathcal{D}$ satisfying pre's condition but not q's, β be the percentage of the values in $\mathcal{D}$ satisfying q's condition but not pre's, and γ be the percentage of the values in $\mathcal{D}$ satisfying both pre's and q's condition. Then, we have $E[\mathcal{D}_{pre}] = (\alpha + \gamma)E[\mathcal{D}]$ and $E[\mathcal{D}_q] = (\beta + \gamma)E[\mathcal{D}]$. Hence,

$$E[\mathcal{D}_d]^2 = (\alpha - \beta)^2 E[\mathcal{D}]^2 \tag{8}$$

Secondly, we will compute $E[\mathcal{D}_d^2]$. We have $\mathcal{D}_{preq} = \{G_{preq}(a_i) \mid i \in [1, n]\}$, where $G(a_i)$ take a_i as input and returns a_i if a_i satisfies the pre's condition as well as the q's condition; otherwise, 0. Similar to the idea of computing $E[\mathcal{D}_d]^2$, we obtain $E[\mathcal{D}_{pre}^2] = (\alpha + \gamma)E[\mathcal{D}^2]$, $E[\mathcal{D}_q^2] = (\beta + \gamma)E[\mathcal{D}^2]$ and $E[\mathcal{D}_{preq}^2] = \gamma E[\mathcal{D}^2]$. Hence, we can get:

$$E[\mathcal{D}_d^2] = (\alpha + \beta)E[\mathcal{D}^2] \tag{9}$$

Combining Equation 8 and Equation 9, we can derive $error(q, pre) = \lambda N\sqrt{\frac{\sigma^2}{n}}$, where $\sigma^2 = (\alpha + \beta)E[\mathcal{D}^2] - (\alpha - \beta)^2(E[\mathcal{D}])^2$.

Given q, our goal is to find pre with the minimal error, which is equivalent to find the minimal σ^2. Denote the selectivity of q as θ. The following two equations always hold:

$$\theta \geq \beta \tag{10}$$

$$0 \leq \alpha + \theta \leq 1 \tag{11}$$

Let us consider pre in P^+ in the following three cases:

(1) Case 1: pre satisfies $\alpha \geq \beta + \frac{E[\mathcal{D}^2]}{2E[\mathcal{D}]^2}$. we will prove that $\sigma^2 - error(q, \phi) \geq 0$, which means $\phi \in P^-$ is the optimal pre query. Actually, we have $\sigma^2 \geq (\alpha - \beta)E[\mathcal{D}^2] - (\alpha - \beta)^2 E[\mathcal{D}]^2$. Hence, $\sigma^2 - error(q, \phi) \geq (\alpha - \beta - \theta)E[\mathcal{D}^2] - (\alpha - \beta - \theta)(\alpha - \beta + \theta)E[\mathcal{D}]^2$.

Since $\alpha - \beta \geq \frac{E[\mathcal{D}^2]}{2E[\mathcal{D}]^2} \geq \frac{1}{2}$, and $\theta \leq 1 - \alpha \leq \frac{1}{2}$ (based on Equation 11), we have $\alpha - \beta - \theta \geq 0$. Since $\alpha - \beta + \theta \leq 1 - \beta \leq 1$, we can get $E[\mathcal{D}^2] - (\alpha - \beta + \theta)E[\mathcal{D}]^2 \geq E[\mathcal{D}^2] - E[\mathcal{D}]^2 \geq 0$. Combine it with $\alpha - \beta - \theta \geq 0$, we can derive $\sigma^2 - error(q, \phi) \geq 0$.

(2) Case 2: pre satisfies $\beta \geq \alpha + \frac{E[\mathcal{D}^2]}{2E[\mathcal{D}]^2}$. We will prove that $\phi \in P^-$ is the optimal pre, i.e., $\sigma^2 - error(q, \phi) \geq 0$. Since $\beta \geq \alpha$, we can get $(\alpha - \beta)^2 \leq \beta^2$. Besides, we also have $\alpha + \beta \geq \beta$. Then, we can derive $\sigma^2 = (\alpha + \beta)E[\mathcal{D}^2] - (\alpha - \beta)^2 E[\mathcal{D}]^2 \geq \beta E[\mathcal{D}^2] - \beta^2 E[\mathcal{D}]^2$.

Hence, $\sigma^2 - error(q, \phi) \geq (\beta - \theta)E[\mathcal{D}^2] - (\beta - \theta)(\beta + \theta)E[\mathcal{D}]^2$.

Since we have $\beta \geq \alpha + \frac{E[\mathcal{D}^2]}{2E[\mathcal{D}]^2} \geq \frac{E[\mathcal{D}^2]}{2E[\mathcal{D}]^2}$ and $\theta \geq \beta$ (Equation 10), we can get $\beta + \theta \geq 2\beta \geq \frac{E[\mathcal{D}^2]}{E[\mathcal{D}]^2}$. Hence, we have $E[\mathcal{D}^2] - (\beta + \theta)E[\mathcal{D}]^2 \leq E[\mathcal{D}^2] - E[\mathcal{D}]^2 \frac{E[\mathcal{D}^2]}{E[\mathcal{D}]^2} = 0$. Combine it with $\beta - \theta \leq 0$, we could derive $\sigma^2 - error(q, \phi) \geq 0$.

(3) Case 3: pre satisfies $\beta < \alpha + \frac{E[\mathcal{D}^2]}{2E[\mathcal{D}]^2}$ and $\alpha < \beta + \frac{E[\mathcal{D}^2]}{2E[\mathcal{D}]^2}$. We will prove that a query $pre \in P^-$ will have the smallest error. If regard σ^2 as a quadratic function of α, then the turn point is $\beta + \frac{E[\mathcal{D}^2]}{2E[\mathcal{D}]^2}$. Since $\alpha < \beta + \frac{E[\mathcal{D}^2]}{2E[\mathcal{D}]^2}$, we can get that for a fixed β, the error is monotonically increasing w.r.t. α. Similarly, for a fixed α, the error is monotonically increasing w.r.t. β. Now given a query q, there are five possible positions that pre query can be. Let x and y denote the lowest point and the highest point of q query. Let pre_x and pre_y denote the lowest point and the highest point of pre query, respectively. Let pre_{opt} denote the optimal pre with smallest error.

(a) position 1: $pre_l \geq x$ and $pre_h \leq y$. In this case, $alpha = 0$ and $\beta = pre_x - x + y - pre_y$. Then the smallest β will get when $pre_x = h_x$ and $pre_y = l_y$. Hence, $pre_{opt} \in P^-$.

(b) position 2: $pre_x < x$ and $pre_y \leq y$. In this case, $\alpha = x - pre_x$ and $\beta = y - pre_y$. Then the smallest α and β is got when $pre_x = l_x$ and $pre_y = l_y$. Hence, $pre_{opt} \in P^-$.

(c) position 3: $pre_x \geq x$ and $pre_y > y$. In this case, $\alpha = pre_y - y$ and $\beta = pre_x - x$. Then the smallest α and β is got when $pre_x = h_x$ and $pre_y = h_y$. Hence, $pre_{opt} \in P^-$.

(d) position 4: $pre_x < x$ and $pre_y > y$. In this case, $\beta = 0$ and $\alpha = x - pre_x + pre_y - y$. Then the smallest α is got when $pre_x = l_x$ and $pre_y = h_y$. Hence, $pre_{opt} \in P^-$.

(e) position 5: $pre = \phi$. Since $\phi \in P^-$, we also have $pre_{opt} \in P^-$.

Proof of Lemma 4

According to Lemma 3, we only need to prove

$$\max_{q \in Q} \min_{pre \in P_{eq}^-} error(q, pre) = \lambda N\sqrt{\frac{\sigma_{eq}^2}{n}}.$$

Let θ denote q's selectivity.

(1) If $\theta > \frac{1}{k}$. In this case, there exists two points h_x and l_y inside query q. We will prove that when x and y are the middle points of $l_x h_x$ and $l_y h_y$, we can get $\max_{q \in Q} \min_{pre \in P^-_{eq}} error(q, pre) = \lambda N \sqrt{\frac{\sigma^2_{eq}}{n}}$, where $\sigma^2_{eq} = \frac{1}{k} E[\mathcal{D}^2] - \frac{1}{k^2}(E[\mathcal{D}])^2$.

First we will prove that when x and y are the middle points, we have $\sigma^2_{opt} = \sigma^2_{eq}$. Actually, there are five *pre* we can choose: $pre = AGG(l_x : h_y)$, $pre = AGG(h_x : l_y)$, $pre = AGG(l_x : l_y)$, $pre = AGG(h_x : h_y)$, and $pre = \phi$.

(a) If $pre = AGG(l_x : h_y)$, then $\alpha = \frac{1}{k}$ and $\beta = 0$, thus $\sigma^2 = \sigma^2_{eq}$.

(b) If $pre = AGG(h_x : l_y)$, similar to (a) we can get $\sigma^2 = \sigma^2_{eq}$.

(c) If $pre = AGG(l_x : l_y)$, we have $\alpha = \frac{1}{2k}$ and $\beta = \frac{1}{2k}$, hence $\sigma^2 = \frac{1}{k} E[\mathcal{D}^2] \geq \sigma^2_{eq}$.

(d) If $pre = AGG(h_x : h_y)$, similar to (c) we can get $\sigma^2 = \frac{1}{k} E[\mathcal{D}^2] \geq \sigma^2_{eq}$.

(e) If $pre = \phi$, we have $\sigma^2 = \theta E[\mathcal{D}^2] - \theta^2 E[\mathcal{D}]^2$. Since $\frac{1}{k} \leq \theta \leq 1 - \frac{1}{k}$, we can get $\sigma^2 \geq \sigma^2_{eq}$.

Hence, $\sigma^2_{opt} = \sigma^2_{eq}$.

Now we will prove that for any query q, we have $\sigma^2_{opt} \leq \sigma^2_{eq}$. Suppose $l = |h_x l_y|$ and $L = |l_x h_y|$.

(a) When $\theta - l \leq \frac{1}{k}$, for $pre = h_x l_y$, we have $\alpha = 0$ and $\beta = \theta - l \leq \frac{1}{k}$. Then, $\sigma^2 = \beta E[\mathcal{D}^2] - \beta^2 E[\mathcal{D}]^2 \leq \sigma^2_{eq}$. Hence, we can get $\sigma^2_{opt} \leq \sigma^2 \leq \sigma^2_{eq}$.

(b) When $\theta - l > \frac{1}{k}$, for $pre = l_x h_y$, we have $\alpha = L - \theta = l + \frac{2}{k} - \theta < \frac{1}{k}$ and $\beta = 0$. Then $\sigma^2 = \alpha E[\mathcal{D}^2] - \alpha^2 E[\mathcal{D}]^2 < \sigma^2_{eq}$. Hence, we can get $\sigma^2_{opt} \leq \sigma^2 < \sigma^2_{eq}$.

(2) If $\theta \leq \frac{1}{k}$, we will prove that the query-template error cannot be larger than $\lambda N \sqrt{\frac{\sigma^2_{eq}}{n}}$. When using ϕ to answer the query, the query's variance is $\sigma^2 = \theta E[\mathcal{D}^2] - \theta^2 (E[\mathcal{D}])^2$. Since $\theta \leq \frac{1}{k}$ and σ^2 is monotonically increasing w.r.t θ, we have $\sigma^2 \leq \sigma^2_{eq}$. Thus, $error(q, \phi) = \lambda N \sqrt{\frac{\sigma^2}{n}} \leq \lambda N \sqrt{\frac{\sigma^2_{eq}}{n}}$. Since $\phi \in P^-_{eq}$, we obtain $\min_{p \in P^-_{eq}} error(q, p) \leq error(q, \phi) \leq \lambda N \sqrt{\frac{\sigma^2_{eq}}{n}}$.

Proof of Lemma 5

In order to prove the lemma, we only need to construct a single bad query $q' \in Q$ such that $error(q', P) \geq \lambda N \sqrt{\frac{\sigma^2_{eq}}{n}}$. Since the precomputed queries are not evenly chosen, there must exist two intervals such that the sum of their lengths is larger than $\frac{2N}{k}$. Construct a query $q' = \text{SUM}(x : y)$, where x and y are the middle points of the two intervals, respectively. Suppose $|l_x h_x| = 2a$ and $|l_y h_y| = 2b$. Then we have $\frac{1}{k} \leq a + b \leq 1 - \theta$ and $\theta \geq a + b \geq \frac{1}{k}$. There are five possible *pre* queries for q':

(a) $pre = AGG(l_x : h_y)$: we have $\beta = 0$ and $\alpha = a + b$. Since $\frac{1}{k} \leq a + b \leq 1 - \theta \leq 1 - \frac{1}{k}$, we have $\sigma^2 = (a+b)E[\mathcal{D}^2] - (a+b)^2 E[\mathcal{D}]^2 \geq \frac{1}{k} E[\mathcal{D}^2] - \frac{1}{k^2} E[\mathcal{D}]^2 = \sigma^2_{eq}$.

(b) $pre = AGG(h_x : l_y)$: similar to (a), we have $\sigma^2 = (a+b)E[\mathcal{D}^2] - (a+b)^2 E[\mathcal{D}]^2 \geq \sigma^2_{eq}$.

(c) $pre = AGG(l_x : l_y)$: we have $\alpha = a$ and $\beta = b$. Then $\sigma^2 = (a+b)E[\mathcal{D}^2] - (a-b)^2 E[\mathcal{D}]^2 \geq (a+b)E[\mathcal{D}^2] - (a+b)^2 E[\mathcal{D}]^2 > \sigma^2_{eq}$.

(d) $pre = AGG(h_x : h_y)$: similar to (c), we can get $\sigma^2 \geq \sigma^2_{eq}$.

(e) $pre = \phi$: we have $\alpha = 0$ and $\beta = \theta$. Since $\frac{1}{k} \leq \theta \leq 1 - \frac{1}{k}$, we have $\sigma^2 = \theta E[\mathcal{D}^2] - \theta^2 E[\mathcal{D}]^2 \geq \sigma^2_{eq}$.

Now, for all five *pre* queries, we have $\sigma^2 \geq \sigma^2_{eq}$. Hence the query error of q' is $\min_{pre \in P^-} error(q', pre) \geq \lambda N \sqrt{\frac{\sigma^2_{eq}}{n}}$.

Proof of Theorem 1

Based on Lemmas 4 and 5, we can easily deduce that the query-template error of Q w.r.t. P_{eq} is minimum. That is,

$$P_{eq} = \operatorname*{argmin}_{P} \max_{\substack{q=\text{SUM}(x:y) \\ 1 \leq x < y \leq N}} error(q, P).$$

Hence, P_{eq} is an optimal BP-Cube.

Proof of Lemma 6

Due to the space limit, we just give the proof sketch. $P^- \setminus \{\phi\} \subseteq P^+$, then $error(q, P) = \min_{p \in P^+} error(q, p) \leq \min_{p \in P^- \setminus \{\phi\}} error(q, p)$. We can see that $P^- \setminus \{\phi\}$ consists of four precomputed queries that lead to four ways to estimate the sum. As shown in Section 6.1.2, we can choose the *pre* query that leads to min $\left\{ \frac{\lambda N}{\sqrt{n}} \cdot \sqrt{\text{Var}(A_{L_x})}, \frac{\lambda N}{\sqrt{n}} \cdot \sqrt{\text{Var}(A_{\bar{L}_x})} \right\}$ and min $\left\{ \frac{\lambda N}{\sqrt{n}} \cdot \sqrt{\text{Var}(A_{L_y})}, \frac{\lambda N}{\sqrt{n}} \cdot \sqrt{\text{Var}(A_{\bar{L}_y})} \right\}$. Then $error(q, P)$ would be $\frac{\lambda N}{\sqrt{n}} \cdot \sqrt{\text{Var}(X + Y)}$ or $\frac{\lambda N}{\sqrt{n}} \cdot \sqrt{\text{Var}(X - Y)}$, where $X = A_{L_x}$ if $\text{Var}(A_{L_x}) \leq \text{Var}(A_{\bar{L}_x})$, otherwise $X = A_{\bar{L}_x}$. Y has a similar meaning with X. Then, based on Cauchy-Schwarz inequality $\sqrt{\text{Var}(X \pm Y)} \leq \sqrt{\text{Var}(X)} + \sqrt{\text{Var}(Y)}$, we can derive the lemma.

B PREPROCESSING COST ANALYSIS

We analyze the time complexity of our aggregate-precomputation technique in this part.

In the first stage, we need to determine which BP-Cube needs to be precomputed. This stage is only executed on a sample $\mathcal{S}$. The total time complexity of this stage is dominated by determining the BP-Cube's shape because it needs to run hill climbing algorithms for multiple times (denote the number of the times by m) in order to plot an error profile for each dimension. The hill climbing algorithm is an iterative algorithm. Let $iter$ denote the number of iterations. Each iteration takes a linear time of $O(n)$. Thus, the time complexity of plotting a single error profile is $O(m \cdot iter \cdot n)$. Since we need to construct d error profiles, the total time complexity is $O(d \cdot m \cdot iter \cdot n)$. Note that here n is the sample size, which is orders of magnitude smaller than the data size. In the experiments, we set $m = 20$ by default, and found that $iter$ is on average smaller than 20.

In the second stage, we need to precompute the BP-Cube obtained from the first stage. Ho et al. [34] proposed an efficient algorithm to do so. The algorithm needs to scan the full data once to initialize a d-dimensional array of the size of $\prod_{i=1}^{d} k_i$. The time complexity of this step is $O(N \cdot \log k)$ and the I/O cost is $O(\mathcal{D})$. Next, the algorithm scans the array for d times and the final d-dimensional array is the BP-Cube that we want to precompute. The time complexity of this step is $O(d \cdot k)$. Since BP-Cube is often small, we assume that it can be put in memory, thus this step does not involve any I/O cost. To sum up, the total time complexity is $O(N \cdot \log k + d \cdot k)$ and the total I/O cost is $O(\mathcal{D})$. Since the entire P-Cube consists of $\prod_{i=1}^{d} |\text{dom}(C_i)|$ cells and a BP-Cube only contains $k (\ll \prod_{i=1}^{d} |\text{dom}(C_i)|)$ cells, AQP++ incurs much less preprocessing cost than AggPre in terms of both space usage and running time.

C EXTENSIONS

Aggregation Functions. As mentioned in Section 4.2, AQP++ has an estimator (Equation 5) that works for any aggregation function that AQP can support. For each aggregation function, however, it may require a different way to construct a precomputed aggregate query set. In this paper, we propose an aggregate-precomputation technique for SUM queries. The technique can be extended to COUNT and AVG with small changes. For COUNT queries, we add a virtual attribute to the table with all the values equal to 1. Any COUNT query can be rewritten as a SUM query on the virtual attribute. For AVG queries, since $\text{AVG}(A) = \frac{\text{SUM}(A)}{\text{COUNT}(A)}$, to construct a BP-Cube for it, we need to consider the accuracy of both SUM(A) and COUNT(A). We employ a simple heuristic approach [15] to combine them together, $\alpha \cdot \text{SUM}(A) + (1-\alpha) \cdot \text{COUNT}(A)$, where $\alpha \in [0,1]$ ($\alpha = 0.5$ by default). Given an AVG query template $[\text{AVG}(A), C_1, \cdots, C_d]$, we add a virtual attribute A' to the table with each value equal to $\alpha \cdot A + (1-\alpha)$, and then apply the aggregate-precomputation technique to $[\text{SUM}(A'), C_1, \cdots, C_d]$ to determine the BP-Cube that need to be precomputed.

For holistic aggregation functions such as median and percentile, BP-Cubes cannot support them well. The main reason is that their query answers cannot be easily combined. For instance, it is easy to add the answers to SUM(1 : 5) and SUM(6 : 10) to get the answer to SUM(1 : 10), but such idea will not work for median or percentile. If there is a sophisticated method to handle precomputed percentile results (like BP-Cubes for SUM), one can still benefit from AQP++ by using bootstrap to compute the confidence interval of $q - pre$ and using Equation 4 to get the estimation. In the future, we will explore other forms of cubes to handle holistic aggregation functions.

Group-by Queries. We now discuss how to extend AQP++ to support group-by queries. In the aggregate precomputation stage, we can treat group-by attributes as condition attributes and then apply the same hill-climbing algorithm to generate BP-Cubes. For example, suppose a user wants to run queries in the following form:

```
SELECT SUM(sales) FROM table
WHERE age GROUP BY country
```

The user can specify a query template [SUM(sale), age, country], and then use our algorithm to generate a BP-Cube for the template.

In the aggregate identification stage, given a group-by query, e.g.,

```
SELECT SUM(sales) FROM table
WHERE 19<age<31 GROUP BY country
```

we need to identify a precomputed aggregate value for each group. One idea is to apply our aggregate-identification approach to each group one by one. However, this may be costly when the number of groups is large. To make this process more efficient, we can adopt a heuristic approach, where we consider all groups as the same and only apply our approach to the following query (which is obtained by removing the group-by clause from the user query):

```
SELECT SUM(sales) FROM table WHERE 19<age<31.
```

Suppose the identified range condition is "20<age<30". Then, we use it for all groups, and construct the following query

```
SELECT SUM(sales) FROM table
WHERE 20<age<30 GROUP BY country,
```

where we can identify a precomputed aggregate for each group.

For both stages, it is clear to see that the proposed extension may not be the most effective solution to enable AQP++ to support group-by queries. For example, in the aggregate precomputation stage, it ignores the fact that *country* is a categorical attribute and its range condition can only use an equal sign, i.e., "*country* = x". We will explore these opportunities and further enhance the extension to group-by queries in future work.

Data Updates. When the underlying data is updated, AQP++ does not only need to update sample data (like AQP), but also needs to maintain precomputed query results. The latter is essentially a materialized view maintenance problem. Many techniques have been proposed to solve the problem [20]. In particular, for SUM, COUNT, AVG queries, their query results can be maintained more efficiently due to the availability of incremental algorithms. Furthermore, since AQP++ only needs to maintain a BP-Cube, it incurs much less maintenance cost than AggPre.

There are many interesting problems in this space, such as how to develop an incremental hill-climbing algorithm, how to achieve a better trade-off between maintenance cost, query response time, and answer quality, and how to combine stale prefix cubes with data updates to answer queries. However, addressing these problems is beyond the scope of this paper. We will systematically study these problems in future work and propose a comprehensive solution to data updates.

Multiple Query Templates. The paper studies how to decide which BP-Cube should be precomputed for a single query template. When multiple query templates are given, we need to decide how to allocate the space budget to each query template. Suppose there are two query templates Q_1 : [SUM(A_1), C_1, C_2] and Q_2 : [SUM(A_2), C_3], and the space budget is k. A simple approach is to allocate $k/2$ to each one. To better balance the allocation, we can adopt a similar idea with the binary-search algorithm in Section 6.2, which tunes the space-budget allocation iteratively. At the beginning, both Q_1 and Q_2 will be allocated to have the space budget of $k/2$. Then, we use error profile curves to estimate which query template has a larger error, e.g., Q_1 has a larger error. In this situation, Q_1 needs more space budget. We adjust the allocation by assigning the budget of $3k/4$ to Q_1 and the budget of $1/k$ to Q_1. The iterative process will continue until the search range is empty.

Space Allocation. In AQP++, part of the space is used for sampling while the other part is used for storing BP-Cubes. One natural question is how to allocate this budget between the two in order to achieve the best performance given a fixed budget. We adopted a simple approach in the paper. This approach is based on the observation that sample size has a big impact on query response time but the size of BP-Cubes does not. Therefore, we can first select the maximum sample size that meets the user's requirement for response time (e.g., less than 0.5 s), and then use the remaining space for storing BP-Cubes. In the future, we will study how to leverage query workloads to develop a more sophisticated approach.

D MORE RELATED WORK

Materialized Views. Precomputed query results are also known as materialized views (see [20] for a survey). If we look at AQP++ in a materialized view context, AQP++ essentially materializes aggregation views [64] as well as sample views [39], and studies how to answer queries using the materialized views. The two problems (aggregate identification and aggregate precomputation) that the paper delves into are known as answering queries using views [22, 31] and selecting views to materialize [32, 64]. But, existing approaches cannot be used to solve our problems because none of them has considered the connection between AQP and AggPre.