Taster: Self-Tuning, Elastic and Online Approximate Query Processing

Matthaios Olma
EPFL

Odysseas Papapetrou
TU Eindhoven/EPFL

Raja Appuswamy
EURECOM

Anastasia Ailamaki
EPFL

***Abstract*—Current Approximate Query Processing (AQP) engines are far from silver-bullet solutions, as they adopt several static design decisions that target specific workloads and deployment scenarios. Offline AQP engines target deployments with large storage budget, and offer substantial performance improvement for predictable workloads, but fail when new query types appear, i.e., due to shifting user interests. To the other extreme, online AQP engines assume that query workloads are unpredictable, and therefore build all samples at query time, without reusing samples (or parts of them) across queries. Clearly, both extremes miss out on different opportunities for optimizing performance and cost. In this paper, we present Taster, a self-tuning, elastic, online AQP engine that synergistically combines the benefits of online and offline AQP. Taster performs online approximation by injecting synopses (samples and sketches) into the query plan, while at the same time it strategically materializes and reuses synopses across queries, and continuously adapts them to changes in the workload and to the available storage resources. Our experimental evaluation shows that Taster adapts to shifting workload and to varying storage budgets, and always matches or significantly outperforms the state-of-the-art performing AQP approaches (online or offline).**

I. Introduction

AQP engines trade accuracy for better response time and lower resource usage, by executing analytical queries over small data samples and providing an approximate result within a few percents of the actual value. State-of-the-art AQP engines are classified into two categories, depending on the assumptions they make about the query workload. *Offline AQP* engines (e.g. STRAT [10] and BlinkDB [4]) target applications where the query workload is known a priori, e.g., aggregate dashboards that compute summaries over a few fixed columns. Offline AQP engines analyse the expected workload to identify the optimal set of synopses (summaries of the data, such as samples, sketches, and histograms) that should be generated to provide fast responses to the queries at hand, subject to a predefined storage budget and error tolerance specification. Since this analysis is time-consuming, both due to the computational complexity of the analysis task, as well as the I/O overhead in generating the synopses, AQP engines perform the analysis offline each time the query workload or the storage budget changes.

While offline AQP engines substantially improve query execution time under predictable query workloads, their need for a priori knowledge of the queries makes them unsuitable for unpredictable workloads. Data exploration is one such example, where future queries are determined based on the results obtained from past queries. These workloads benefit from *online AQP* techniques, where approximation is introduced to query execution at runtime. State-of-the-art online AQP engines achieve this by introducing samplers during query execution. By reducing the input tuples, samplers improve performance of the operators higher in the query plan. In this way, online AQP techniques can boost unknown query workloads. However, query-time sampling is limited in the scope of a single query, as the generated samples are not constructed with the purpose of reuse across queries – they are specific to the query, and are not saved. Thus, online AQP engines offer substantially constrained performance gains compared to their offline counterparts for predictable workloads.

In summary, all state-of-the-art AQP engines force end-users to pick an extreme point in the generality–performance spectrum, as they make static, design-time decisions based on a fixed set of assumptions about the query workload and the available resources. However, workload in modern data analytics clusters is complex, far from homogeneous, and often contains a mix of queries that vary widely with respect to the degree of approximability [4]. Similarly, the available hardware resources are also non-static and time-varying. For instance, an administrator might elastically provision storage space for storing synopses based on the expected system load. Hence, in the ideal case, an AQP engine should be *self-tuning* and *adaptive*. It should automatically pick the right point in the design spectrum based on the workload, and adapt its decision on-the-fly with each change in workload or storage capacity.

In this work, we present Taster, a self-tuning, adaptive, online AQP engine. Taster inherits ideas from (adaptive) database systems, such as intermediate result materialization, query subsumption, materialized view tuning and index tuning, and adapts these in the context of AQP, enabling a combination and extension of the benefits of both offline and online approximation engines. First, by injecting approximation operators in the query plan, Taster supports a broad range of queries over unpredictable workloads. Taster extends prior work by showing that the technique of injecting sampling operators can also be generalized to sketch-based approximations. Second, by performing online materialization of synopses as a byproduct of query execution, Taster provides performance on-par with offline AQP engines under predictable workloads, yet without an expensive offline preparation phase. Taster also extends prior work by supporting synopses materialization at intermediate stages in the query plan, and not only for summa-

rizations of base tables. Third, by using a cost:utility greedy algorithm to determine the right set of synopses to maintain, Taster adapts to changes in the workload and available storage.

Contributions. This paper makes the following contributions:

- We present Taster, an online adaptive approximate query engine that enables materialization of synopses during query execution, and their reuse across queries. Taster uses synopses for summarizing both base tables and intermediary results of query subplans. We show how to integrate synopses as first-class citizens in query planning, which leads to better plans and improved performance.
- We formalize a utility metric to capture the performance benefit of a synopsis. The metric drives an online tuning algorithm that determines the optimal set of synopses to maintain.
- We consider the question: what other optimizations are possible if we have additional knowledge of user's intentions, e.g., some frequent queries, on which attributes, and on which files? We sketch the space of possible optimizations in the presense of additional user hints, and demonstrate how Taster integrates these hints by preconstructing some samples using state-of-the-art offline sampling algorithms (e.g., variational subsampling).
- We integrate our techniques into SparkSQL. We compare Taster to vanilla SparkSQL, a representative offline AQP approach called BlinkDB, and an online approximation approach called Quickr. Our experiments with industry-standard benchmarks demonstrate that Taster offers substantially improved performance compared to online AQP engines ($2.9\times$), and comparable performance to offline AQP engines without requiring the excessive sample pre-generation cost. Speed-up compared to the baseline is over $3\times$ on average ($20\times$ when additional hints are provided, reaching to $30\times$ for some queries).

The rest of this paper is organized as follows. Section II presents an overview of the used approximation methods. Section III discusses the architecture of Taster, along with an example of its execution workflow. Section IV details on the query planning process, whereas Section V discusses the self-tuning nature of the system. Section VI presents a thorough experimental evaluation. We discuss related work in Section VII, and conclude in Section VIII.

II. Approximation methods

Taster utilizes two types of synopses: sketches and samples. Both satisfy the following requirements, which are imperative for high performance: (a) they are partitionable, i.e., they can be constructed over massively parallel platforms such as Spark, and (b) they are pipelineable, such that they can be built with a single pass over the data.

Samples. Taster generates samples using two types of samplers – uniform and distinct. Each sampler goes over all input rows, and lets only a subset of them to pass through. Relational operators are then executed over the available sample(s), and the aggregates are scaled to get the final estimates. To be able to perform this scaling, each sampler appends an additional attribute that represents the weight associated with the row. For example, given a query calculating the SUM of a column, for every tuple of the sample with value t_i and weight w_i, Taster returns $t_i \times w_i$.

Uniform sampler. The uniform sampler Γ_p^{U} samples without replacement, letting a row pass through with probability p at random. For every row, the weight is set to $1/p$. This sampler is both pipelineable and partitionable, and its memory footprint during construction is approximately equal to the memory footprint of the desired sample size.

Distinct sampler. Even though the uniform sampler has low execution overhead, it does not have good statistical properties in more complex workloads, e.g., in join queries, it may miss an arbitrary large number of join keys. Prior works cope by generating stratified samples. Stratified sampling guarantees the existence of all groups for specific attributes (stratification attributes) and a minimum number of tuples per group. However, stratified sampling operators are blocking operators and require two passes over the data. The execution overhead of such an operator is prohibitive for online approximation. Therefore, Taster uses distinct sampler [12], [24], [25], which guarantees that at least a certain number of rows pass per distinct combination of values of a column set.

Distinct sampler works as follows: given a set of stratification attributes $\mathcal{A}$, a number δ, and probability p, the distinct sampler $\Gamma_{p,\mathcal{A},\delta}^{\mathsf{D}}$ passes at least δ rows for every distinct combination of values of the columns in $\mathcal{A}$. Subsequent rows with the same value are let through with probability p, uniformly-at-random. The weight of each row is set correspondingly: If the row passes because of the frequency check, its weight is set to 1, whereas if it passes due to the probability check, its weight is set to $1/p$. The Taster planner extracts the parameters $\{\mathcal{A}, \delta, p\}$ automatically for each query, such that user accuracy requirements are satisfied (details in Section IV-B). In terms of implementation, distinct sampling is implemented efficiently by using a heavy-hitters sketch that requires space logarithmic to the number of rows [12].

Distinct sampler is pipelinable by design, as it requires only a single pass over the data. To make it partitionable, given the sampler operator distribution factor D (the number of operator instances), we adjust the minimum number of rows required from each operator instance from δ to $\delta + D\epsilon$ with ϵ being a variable addressing variations in data distribution. As per [25], ϵ is set to δ/D, which builds on the assumption that data is distributed uniformly across instances.

Sketches. Some queries are amenable to sketch-driven approximations. Such queries are, e.g., nested queries containing EXISTs which can be approximated with Bloom filters [8], distinct counts and join size estimations with Bloom filters [34], FM-sketches [17], and AMS-sketches [6].

Taster exploits count-min sketches [13] to answer several types of these queries. A count-min sketch consists of a 2-dimensional array ($w \times d$) of counters (integers), accompanied with d pairwise independent hash functions that uniformly map each item from the domain space (each potential key) to one counter per array row. Let counter i at row j be denoted as

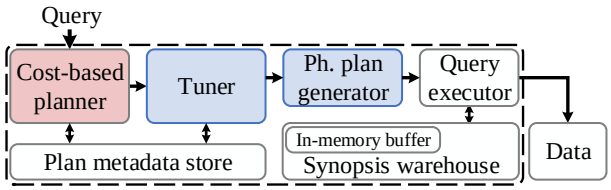


Fig. 1: The overview of Taster.

$A[i][j]$, and hash function corresponding to row j be denoted as h_j. Adding an item x to a sketch is achieved by finding the corresponding counters from the sketch, i.e., $A[h_j(x)][j]$ for $j = \{1, \dots d\}$, and increasing these by one (or by the frequency of x). The sketch has a memory footprint of a few MB and can be constructed on-the-fly during query answering. After construction, the sketch is used as an *approximate key-value store* for estimating the frequency $\hat{f}(x)$ of any item x, as follows: $\hat{f}(x) = \min(A[h_j(x)][j] | j = \{1, \dots d\})$. When $d = O(1/\delta)$ and the number of columns of the sketch is set to $O(1/\epsilon)$, the estimate is within range ϵN of the real answer, with probability at least $1 - \delta$ (variable N represents the L1 norm of the frequencies). Construction of count-min sketches is fully partitionable. Therefore, each node in the cluster builds sketches for its own data, and all sketches for one dataset are added pair-wise to get a sketch representing the whole RDD.

Sketch-join. Besides simple aggregations, the sketch also supports aggregations over joins. The Sketch-Join operator builds a sketch on the relation over which the aggregation takes place and uses as key the join key and as a value the executed aggregation for the tuple. This sketch is subsequently used in a similar fashion as a hash index in the hash-join algorithm. The reduced size of the sketch (a few MB as opposed to possibly several GB for a sample of a large table, or a hash index) makes sketches ideal for materialization and re-use in subsequent queries.

III. Architecture of Taster

Figure 1 presents Taster's high-level architecture. Taster is implemented over SparkSQL and extends Apache Catalyst query optimizer and SparkSQL query engine with online approximation techniques, combined with synopsis materialization and self-tuning. The techniques presented are not limited to SparkSQL, and are applicable to any query processing system – even centralized ones. In the following we present a high-level overview of the core concepts of Taster.

Synopses and synopsis warehouse. Taster uses a set of automatically-constructed and tuned synopses to summarize both the raw data (the base relations) and intermediary results of subplans (e.g., join results). Currently, it exploits two types of synopses, samples and sketches, each being appropriate for answering different query families (cf. Section II). All synopses are constructed as byproducts of query answering, and are saved in the synopsis warehouse, in HDFS. Along with synopses, Taster stores statistics of the dataset (distribution of values, number of distinct values), which are calculated on-the-fly during the first access to any table. To control monetary cost, the synopsis warehouse is subject to space quota, which is set at initialization and can also be modified at runtime from the administrator. More details for the process of selecting synopses for the synopsis warehouse will be presented in Section V.

Synopsis buffer. The plan chosen for execution may require generation of a new synopsis (i.e., if the synopsis is not already in the synopsis warehouse). Generation of a new synopsis on-the-fly may still be beneficial for the query at hand, in order to reduce CPU usage of operators higher in the plan. In this case, the new synopsis will be temporarily stored in the *synopsis buffer* – a fixed-size buffer implemented as a sequence of in-memory RDDs in Spark. The buffer offers two main benefits: (a) it serves as a fast main-memory cache, which offers significant boost for workloads exhibiting temporal locality, and, (b) it decouples the decision of writing the synopsis in the HDFS-based synopsis warehouse – an I/O expensive operation – with the process of query answering which needs to be executed with a very small latency. When the buffer is full, the tuner decides which synopses should be permanently stored in the synopsis warehouse (cf. Section V).

Cost-based planner. Taster's query engine decides automatically on the exploitation of supported synopses to speed-up user queries. This automation relies on a *cost-based planner*, which is currently built into the Catalyst optimizer. Upon receiving the query, the planner generates a set of approximate execution plans. These plans utilize synopses that may, or may not yet exist, and they all satisfy the approximation requirements of the query. The next step is to estimate the cost of each plan and the performance gain by the use of synopses, compared to the best plan without synopses that will return exact answers. The plans and their costs are then passed to the tuner, for further optimizations and the final execution. The cost-based planner is discussed in Section IV.

Tuner. The primary purpose of the tuner is to choose the best plan out of the ones proposed by the planner. However, when ranking the plans, the tuner focuses on maximizing long-term throughput, i.e., over the future workload, as opposed to minimizing the cost for the query at hand. This *holistic optimization* translates to decisions in two levels: (a) promoting the plans that generate reusable synopses, pertinent to many different queries, and, (b) deciding which of the generated synopses will be stored in the synopsis warehouse, and which will be deleted, to satisfy the space quota. Tuning involves two major challenges: (a) holistic optimization can be CPU-intensive, and (b) the future queries, over which the tuner needs to optimize, are of course not yet known. We explain how these issues are addressed in Section V.

Physical plan generator. The plan chosen by the tuner is subsequently passed to the physical plan generator, for extraction of the physical plan and execution over Spark. The physical plan generator is now implemented within the tuner to avoid additional synchronization overhead. Fault tolerance, distribution, partitioning-related details, and the actual task execution are handled transparently by Spark.

Metadata store. Effectiveness of both the planner and tuner

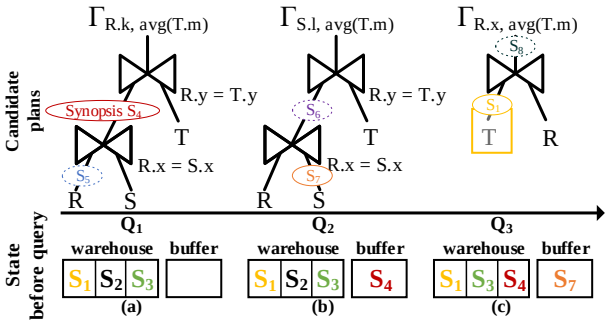


Fig. 2: Example execution of Taster. The dotted ellipses depict prospective synopses, the normal-line ellipses depict chosen synopses. Yellow boxes are used to denote reuse of existing synopses. $\Gamma_{group,aggr}$ denotes aggregation operator with $group$ grouping attributes and $aggr$ aggregation function.

depends on the existence of metadata that characterizes the past workload and the synopses that could speed-up this workload. The metadata store is a main-memory, *synopses-centric* metadata repository that keeps rich statistics about the properties, impact, and popularity of each synopsis. In particular, the store keeps details for all synopses contained in all plans generated by the planner – even the ones that are not chosen for execution. These details include: (a) the logical definition of the synopsis (the logical subplan whose results are summarized by this synopsis), (b) stratification and accuracy requirements of the synopsis, (c) whether the synopsis is saved in the synopsis warehouse or not, and, (d) the list of recent queries that could utilize this synopsis to improve performance, their estimated cost when this synopsis exists, and their cost if an exact query plan (without synopses) would be chosen instead. The purpose of this metadata is twofold: (a) to assist the planner to estimate the cost of each candidate plan (cf. Section IV-A), and (b) to enable the tuner to decide which synopses will maximize throughput, i.e., because they will improve many different subplans (cf. Section V).

Example. Figure 2 presents an overview of Taster running three queries over three relations R, S, T. For simplicity, we assume that the synopsis buffer fits one synopsis, and the warehouse fits three synopses.[1] Just before arrival of Q_1, the synopsis warehouse already contains synopses S_1, S_2, and S_3. S_1 is a sample of relation T. Synopses S_2 and S_3 refer to another table W, not relevant to the three queries.

During Q_1, the planner proposes two candidate plans (cf., Fig. 2a). The first one contains synopsis S_4, which summarizes $R \bowtie S$, and the second contains synopsis S_5 of R. Notice that neither of the two synopses exist. The two plans are costed, and the metadata store is updated with the corresponding properties of S_4 and S_5. Then, the plans are sent to the tuner. The tuner identifies the best plan (in this case, the one with S_4), and sends it for execution. During execution, S_4 is generated and saved in the in-memory synopsis buffer. When Q_2 arrives, the planner identifies two candidate plans (cf., Fig. 2b), which rely on the nonexistent synopses S_6 and S_7 respectively (synopses S_1 and S_4 cannot be used because of different grouping attributes). Again, the planner updates the metadata store with the corresponding properties of the two candidate synopses, and the plans are sent to the tuner. Now, as the synopsis buffer is full, the tuner first needs to free up space, so that either of the candidate synopses can be generated. The synopsis warehouse is also full. By estimating the long-term benefit of each synopsis, the tuner decides to keep S_1, S_3, and S_4 in the warehouse, and to execute the plan that requires S_5. The plan is executed, and S_7 is stored in the synopsis buffer. During Q_3, the planner proposes two plans (cf., Fig. 2c), the first replacing the scanning of relation T with S_1 which is already saved in the warehouse (the yellow box), and the second utilizing a non-existent synopsis S_8. The plans are sent to the tuner, where the first one is chosen and sent for execution.

Supported Queries. Taster accepts and answers all SQL queries supported by Spark SQL. Similar to prior work, e.g., [4], [25], it improves performance for queries containing aggregates (e.g., COUNT, AVG, SUM). The query format for approximate queries follows the standard syntax: "ERROR WITHIN x% AT CONFIDENCE y%", which corresponds to aggregate results with relative error of at most x% at a y% confidence level.

IV. Query Planning with Synopses

Taster automatically decides which synopses to create, store, and use for answering each query. Synopses are used for summarizing both raw data (base relations) and query subplans (e.g., the results of an aggregator over a join). Due to their small size compared to the original data, synopses improve both computational complexity and I/O cost during query processing. All synopses are created on-the-fly, as byproducts of query answering, thereby inducing no additional I/O.

Synopses in Taster are promoted to first-class citizens: they are included as approximate operators in the logical query plans, costed as all other logical operators, and transformed to fully pipelined and distributable code during the physical plan generation. This enables the planner to produce more efficient plans, and the tuner to promote reusability of synopses by matching synopses across different queries. In the remainder of this section we will explain how the Taster planner integrates synopses into planning. The discussion explains the plan generation process, how synopses are configured to satisfy the query's accuracy requirements, and how they are matched to existing synopses from the synopses warehouse.

A. Query Planning

The planner generates auxiliary logical plans, replacing the aggregator operators with approximate aggregators (whenever these are beneficial for performance), costing the plans, and, passing them to the tuner for further optimizations. In the following, we describe this process in detail.

Upon receiving query q_i, the planner generates candidate logical plans $\mathbb{P}(q_i) = \{p_1, p_2, \ldots\}$, which integrate synopses.

[1] This is only for illustration purposes. Since synopses have different sizes, quotas are determined in GB, and not in number of synopses.

The key observation to limit the search space is that prospective synopses are used for approximating aggregators and joins. Focusing on aggregations, the planner first identifies all query subplans rooted on (partial/eager) aggregators. For each, it injects a generic synopsis operator just below the aggregator operator, and modifies the aggregator to account for the synopsis (e.g., a SUM over a sample would require scaling to account for the full dataset). The synopsis operator represents the potential to efficiently approximate the underlying subplan by the use of a (possibly not yet existent) synopsis. Subsequently, Taster tweaks the query plan to achieve two goals, (i) maximize the re-use of existing synopses and (ii) satisfy the user's accuracy requirements. All resulting plans are annotated with cost estimates based on their expected I/O, and analyzed to extract all synopses, along with the subplans they summarize. The collected data is used to update the metadata store with the appearances of these synopses. Following, all plans are passed to the tuner for further optimizations.

The above process entails several challenges. First, the process of generating candidate plans is different compared to traditional planners. Unlike traditional query planning, the planner now also needs to take into account the required approximation guarantees and stratification requirements while constructing the plans. Furthermore, when pushing down a synopsis in the plan, the synopsis, as well as its corresponding approximate aggregation operator, may require modifications. Second, the approximate aggregators in the plan need to be configured. This boils down to choosing between the supported types of sampling and sketches, and configuring the selected synopses (e.g., for uniform sampling, setting the sampling probability). Third, the candidate synopses contained in the plan need to be mapped to existing synopses (if any), so that the planner can replace the subplan with the synopses, and estimate the execution cost. In the following we describe how the planner handles these three challenges.

Generating the candidate plans. The planner generates the first set of plans by injecting synopsis operators below the aggregations. Particularly, given aggregation operator $\Gamma_{G,AGG(A)}(c)$, which computes aggregation function AGG over the data produced by operator c (the child operator in the logical plan) by grouping over attributes A, the synopsis operator $\Gamma^{\mathsf{S}}_{state}$ is injected and the aggregation operator is updated (now denoted as $\Gamma'_{G,AGG(A)}(\Gamma^{\mathsf{S}}_{G}(c))$) to use the synopsis as input. Subsequently, Taster starts pushing the synopses down in the plan, closer to the raw data, as an effort to enable executing the plan with existing synopses, or to generate more re-usable synopses. For these, it relies on the push-down rules for synopses introduced in [25], and adapted to enable sketch synopses. Briefly, whenever Taster pushes a synopsis operator under a filter σ_p, it needs to account for two possibilities. If the distribution of values of predicate p is uniform, the new operator is moved under the filter unaltered, since a uniform sample over that attribute will not reduce the number of groups appearing in the final result [29]. However, if the distribution of the values of p is skewed (some groups appear infrequently), Taster needs to stratify the underlying output on p. Thus, Taster adds the attributes appearing in p which follow a skewed distribution into the stratification set.

Considering pushing synopses under the joins, given a join $R \bowtie_{jp} S$ with join predicates jp, the planner pushes the synopsis below the join, to the side of the join on which the aggregation takes place (say, the side of R), and modifies the stratification attributes of the synopsis to include the attributes from jp that are contained in R (i.e., $\Gamma^{\mathsf{S}}_{(A\cup jp)\cap R}(R) \bowtie_{jp} S$). Finally, if the join predicate is not a grouping attribute, Taster introduces a partial aggregation after the join.

The above push-down process guarantees that (i) the generated physical query plan will gather sufficient samples from each of the groups to satisfy user's accuracy requirements, and (ii) the overall sampling process overhead will not exceed the performance gains. We discuss how result accuracy is estimated efficiently and reused across different queries in Section IV-B. In terms of implementation, the push-down strategies are implemented as rules in the Catalyst optimizer, and are executed at every query. Since Catalyst default implementation returns only a single plan at the end, we intervene the planning process in order to store all intermediate plans.

Choosing and configuring the synopses. The synopsis operators contained in the logical plans up to now were parameterized with stratification and accuracy requirements, but omitted configuration details, e.g., which synopsis to use, and how to configure it for satisfying user's accuracy requirements.

Due to the immense ratio of performance gain to storage requirement of sketches, Taster prioritizes the use of sketch-join when appropriate: Let R and T be two relations joined over attributes jp and subsequently passed through aggregator $\Gamma_{grp,agg}$, with grp being the grouping attributes and agg the attributes taking part in the aggregation. With $attrs(R)$ we denote the attributes of R which are given as input to the join. Sketch-join can boost join queries with aggregates, when the projected attributes from one side of the join are either join attributes, or they are used in the aggregate function. Formally, the following requirements must be satisfied:

- $attrs(T) - jp = agg$
- $attrs(T) \cap grp = \emptyset$ OR
  $attrs(T) \cap grp = attrs(T) \cap jp$

Then, the synopsis operator injected between the aggregation and the join can be pushed under the join operator, and transformed into a sketch-join operator.

When sketch-join is not applicable, Taster falls back to sampling. In this case, the planner needs to decide which sampling strategy will be used. A key input for this decision is the cardinality estimates per relational expression, and the number of distinct values in each column (both these statistics are computed during the first access to the table). In particular, Taster checks (i) if the set of stratified attributes C is empty, and, (ii) if some sampling probability $p \leq 0.1$ can ensure that, each distinct value of the columns in C receives at least k rows w.h.p.. If both these checks are true, the sampler is implemented using the uniform sampler. Otherwise, if $C \neq \emptyset$, Taster chooses a distinct sampler. Finally, Taster generates a plan without samplers if stratification and accuracy

requirements are so restrictive that they cannot be satisfied with a reasonable sampling probability.

Matching subplans to materialized synopses. Costing of the logical plans requires efficiently matching the synopses contained in the query's logical plans to the synopses stored in the synopsis warehouse and buffer. This matching is enabled through the metadata store.

Particularly, each synopsis (candidate or materialized) corresponds to a unique logical subplan – the one of which the results it summarizes. Therefore, the subplans for the query at hand are compared to the subplans of the synopses contained in the metadata store. We say that a query subplan matches a synopsis when: (i) the accuracy guarantees of the synopsis satisfy the query requirements, and (ii) the synopsis subplan subsumes the query subplan. For the latter, Taster ensures that the query subplan is covered by the synopsis regarding join and filtering predicates as well as the projected columns. Particularly, the synopsis subplan must have identical join predicates, its filtering predicates must be weaker than, or equal to the filtering predicates of the query, and its output attributes must be a superset of the corresponding parameters of the query subplan [19]. Some mismatches are addressed by adding filtering and projection operators directly above the query subplan, to remove extraneous tuples and attributes. Considering accuracy, a synopsis is a candidate for a subplan if (i) the set of stratification attributes of the stored synopsis is a superset of the stratification attributes of the subplan, and (ii) the aggregation function and the aggregate columns are identical to those of the synopsis and the accuracy requirement of the query generating the synopsis is equal or weaker than of the current query. By ensuring the former, Taster guarantees group coverage i.e., Taster results will contain all groups, whereas the latter ensures that the aggregates will have constrained error [4]. For example Q1: "SELECT $dept, AVG(salary)$ FROM $Employees$ GROUP BY $dept$" will generate a sample over $Employees$ stratified on $dept$. Subsequent query Q2: "SELECT $dept, AVG(salary)$ FROM $Employees$ WHERE $gender$ = 'male' GROUP BY $dept$" will be able to use the previous sample, since, the created sample is more general and Taster can put an additional filter in the query plan. However to use this sample, salaries should be uniformly distributed, irrespective of gender.

Subplan matching is expensive. Therefore, Taster utilizes an index to speed-up this process. Specifically, all candidate synopses contained in the metadata store are indexed using their base relations as the key. In the case of joins, the join attribute(s) are also included in the key. This index, although simple, effectively limits the search space and the lookup time to find suitable synopses for each subplan.

B. Accuracy guarantees

While generating and exploring the potential plans, the planner needs to ensure that the user's accuracy requirements are satisfied. For this, Taster relies on previous analytical results [13], [25], which we outline below.

When using sampling, Taster uses the Horvitz-Thompson (HT) estimator [29] to calculate unbiased estimators of the true aggregate values. Confidence intervals are computed using the CLT. Due to the distance of the samplers to the aggregation operators, we use the notion of dominance between query expressions as defined in Quickr [24], which ensures that plans resulting from transformation rules used by the optimizer have no worse variance of estimators and no higher probability of missing groups than the plan with only one sampler before the aggregation operator. In terms of implementation, a naive way to compute the HT estimator squared error requires a self-join and can take quadratic time since it checks all pairs of tuples in the sample [29]. However, for stratified and uniform sampling, Taster calculates the error in a single pass by utilizing the observation of [25] that to compute the standard error for each group we only need to take into account the tuples with the same stratification key (resp. grouping key). Therefore, we estimate the expected error for each group by building a distributed hash table, using as a key the values of the stratification (resp. grouping) attribute, as as value the running estimated error for that group and the corresponding list of sampled tuples. For every sampled tuple, Taster updates the error of that tuple's group by using the HT estimator error formula, leading to a single-pass, linear complexity algorithm.

CM-sketches offer error guarantees relative to the L1 norm of the summarized relation [13]. Particularly, let $f(x)$ denote the real frequency of key x, and $\hat{f}(x)$ the frequency estimated from the sketch. Then, the sketch is configured such that $\hat{f}(x) - f(x) < \epsilon N$ w.h.p., where N represents the L1 norm of the frequencies for all keys (cf., Section II).

V. Continuous synopsis tuning

Taster's self-tuning nature and ability to adapt to shifting user interests stems from a lightweight synopsis tuner. The tuner is invoked just after the planner, and has a goal to select the candidate plan that will *maximize the throughput over a window of the next w queries* (we will discuss about the value of w later). That is, in contrast to the planner which generates plans with a short term outlook (per-query performance), the tuner looks into overlaps between queries and query subplans in order to increase the long-term performance. The tuner's decisions are driven by a cost:utility model, which leads to a formalization of the task as an optimization challenge. Notice that the decisions made by the tuner affect solely query performance, and not the required accuracy. Even though the tuner has the final decision on which synopsis to build, the considered synopses are proposed by the planner, and thus satisfy user's accuracy requirements (cf., Section IV-A).

The cost:utility model. Tuning is an iterative process. At every invocation, the tuner is presented with a set of candidate plans for query q, denoted with $\mathbb{P}(q) = \{p_1, p_2, \ldots\}$, and needs to choose one for execution in order to maximize throughput. Intuitively, the tuner will solve two problems concurrently: (a) select the best plan and corresponding synopses for answering the query, and (b) choose the best set of synopses to keep, which will speed-up Taster over

a horizon of the next w queries, denoted with $\mathbb{Q}_i^+$, i.e., $\mathbb{Q}_i^+ = \{q_i, q_{i+1}, \ldots, q_{i+w-1}\}$.

It is useful to define the synopsis gain metric, i.e., how much does each set of synopses $\mathbb{S}$ contribute to the performance of each query. Formally, $gain(q, \mathbb{S}) = cost(q, \emptyset) - cost(q, \mathbb{S})$, where $cost(q, \mathbb{S})$ denotes the *minimum cost of any plan* in $\mathbb{P}(q)$ for answering q, given only the synopses in $\mathbb{S}$. In the case of $\mathbb{S} = \emptyset$, this will be the cost of the most efficient plan that does not utilize synopses and returns the exact answers. For a given $\mathbb{Q}_i^+$ we maximize the query throughput by minimizing the total cost of these queries, i.e., $\texttt{minimize} \sum_{q \in \mathbb{Q}_i^+} cost(q, \mathbb{S})$, or equivalently, by maximizing their corresponding gain: $\texttt{maximize} \sum_{q \in \mathbb{Q}_i^+} gain(q, \mathbb{S})$. For convenience, we slightly overload the notation by using $gain(\mathbb{Q}_i^+, \mathbb{S})$ to denote the gain over all queries using synopses in $\mathbb{S}$. Notice that the problem contains two variables. The first one, which is latent, is the set of plans $\mathbb{P}(q)$ for each query $q \in \mathbb{Q}_i^+$. The second is the set of synopses $\mathbb{S}$. Formally, the optimization problem is as follows:

$$\begin{aligned} \underset{\mathbb{S}}{\texttt{maximize}} \quad & gain(\mathbb{Q}_i^+, \mathbb{S}) \\ \text{subject to} \quad & \sum_{s \in \mathbb{S}} |s| \leq \texttt{maxSpace} \end{aligned}$$

where `maxSpace` denotes the space quota for synopses, and $\mathbb{S}$ denotes the set of synopses that will maximize the objective function. Therefore, the tuner needs to select the set of plans (one per query) and synopses that will maximize the total gain.

Even though the problem is well-defined, it involves two challenges. First, it turns out that the problem can be reduced to the NP-hard knapsack constraint problem. This happens because of correlations between synopses, i.e., each synopsis can be used for answering more than one queries, and some queries are answered by more than one synopses. Therefore, we cannot hope for a tractable exact solution. Luckily, we can approximate the solution within a constant factor, by noticing that the objective function is a monotone submodular function, i.e., the gain provided by each single synopsis is only reduced as the set of synopses in $\mathbb{S}$ increases. For this special case, there exist several efficient approximation algorithms. We employ the efficient greedy algorithm of [27], which guarantees that the gain of the constructed set will be within a factor $(1 - 1/e)/2$ of the maximum gain. In a nutshell, the algorithm builds $\mathbb{S}$ gradually by starting from an empty set and adding synopses one-by-one until the quota is filled. At each step, synopses are chosen based on their marginal gain, i.e., how much is the additional gain each synopsis brings when added in $\mathbb{S}$. After $\mathbb{S}$ is created, the tuner checks all synopses that are already stored in the synopsis buffer and warehouse, and updates them accordingly: all synopses not contained in the newly-computed $\mathbb{S}$ are deleted.

The second challenge concerns the definition of the tuner's horizon, $\mathbb{Q}_i^+$. In practice, we cannot expect to know the queries contained in $\mathbb{Q}_i^+$ during the tuning. We therefore employ the standard assumption that recent queries are a good representation of the following queries [9]. For this, we keep track of the last w queries, denoted as $\mathbb{Q}_i^- = \{q_{i-w+1}, q_{i-w+2}, \ldots q_i\}$, and use their proposed plans to estimate $gain(\mathbb{Q}_i^+, \mathbb{S})$.

Storage elasticity. This cost:utility model is also used for adapting to the available storage budget. Taster's administrator can modify the space quota of the synopses warehouse online. This action will automatically invoke the tuner to re-evaluate all synopses, and decide which ones need to be discarded, or created at future queries.

Physical plan generation. The above algorithm will choose both set $\mathbb{S}$, and the plan that minimizes the cost for q. This plan is then used for generating the physical plan. If the plan refers to creation of a new synopsis, then this step is injected in the physical plan as a new operator. The new synopsis is then stored in the in-memory synopsis buffer. In this case, the tuner has already freed up the required space in the buffer, during the tuning phase.

Computational overhead of the tuner. The cost estimates for each subplan (with and without each synopsis) are already computed by the planner and stored in the metadata store, i.e., they do not need to be recomputed from the tuner. The tuner also knows which of the synopses are already stored in the synopsis warehouse or the synopsis buffer, in order to account for the need to create synopses that do not yet exist. Therefore, computation of marginal gain per synopsis is very efficient. In practice, our single-threaded/centralized implementation of the tuner takes ~ 2 seconds per query.

Adapting the tuner's horizon length. To predict usefulness of each synopsis, Taster uses a sliding window of the previous w queries as a good approximation of the next, unseen w queries. The best value for w depends on the task at hand – data exploration, verification of hypotheses, finding outliers, etc. – which determines the *repetitiveness* in the query workload. Therefore, Taster dynamically adapts w.

Initially, w is set to a small value. The tuner also identifies (without building) the set of best synopses using a slightly larger and a slightly smaller w value, i.e., $w^+ = \lceil (1+\alpha) \times w \rceil$ and $w^- = \lfloor (1-\alpha) \times w \rfloor$, with $\alpha \in (0, 1)$. At the next invocation, the tuner examines which of w^-, w, or w^+ would minimize execution time for the queries that arrived since the last invokation, and sets w to that value for the next tuning round. Since all necessary statistics for estimating execution time are already contained in the metadata store, this computation is very efficient.

Our experimental results signify the need to dynamically adapt w. In our tests, we start with default values $w = 10$, and $\alpha = 0.25$. The results show that, for the tested query workloads, the optimal w varies between 12 and 17. Compared to a fixed w, adaptive configuration shows performance improvement that exceeds $1.5\times$. A too large or too small value of w annihilates the predictive nature of the tuner, leading to bad choice of synopses. Value of α is also important on the adaptation speed, and part of our current work is to vary α.

User hints. Our discussion up to now assumed that the user is not required to (and, in most cases, cannot) offer hints/advice to the system. This is the typical case in many data science and data exploration scenarios, where the query load is unpredictable – hence the importance of the online tuner. However, several past works frequently required that the

user provides different types of hints for the optimizer. This information includes, e.g., the whole query workload [4], or the synopses to be constructed, such that they can be build in a pre-processing step [35]. The natural question that arises is: how can Taster utilize such additional knowledge and hints?

A priori knowledge of the full query workload can be utilized from Taster, for accurate computation of the gain of each synopsis – since the full $\mathbb{Q}_i^+$ will be known at every invocation of the tuner, we do not need to revert to the past queries $\mathbb{Q}_i^-$ in order to estimate $gain(\mathbb{Q}_i^+, \mathbb{S})$. The user can also request some synopses to be pre-built offline, and pinned in the synopsis warehouse. In this case, Taster will generate these synopses off-line, and the tuner will never delete them. Still, tuner will keep optimizing the use of the remaining available space, filling it with synopses according to the observed queries. As we show experimentally, pre-computed synopses can lead to significant speed-up (up to $20\times$ compared to baseline), since the synopsis generation time will not be included in the query execution time.

VI. Evaluation

We compare Taster against three state-of-the-art systems: Quickr [25], BlinkDB [4][2], and vanilla SparkSQL which we refer to as *Baseline*. We compare the systems using industry standard benchmarks and a micro-benchmark. Specifically, we use TPC-H with scale factor 300 (300GB before compression) along with the TPC-H queries[3], and TPC-DS with scale factor 200 (200GB before compression) along with a set of 20 TPC-DS queries. To examine suitability of Taster under various workloads we also use a synthetic benchmark of an online grocery store (*instacart*) [1], scaled $100\times$ ($\sim$ 120GB before compression). The query templates used for the instacart benchmark are shown in Table I. All datasets were stored in the Parquet-compressed data format.

Experimental Setup. The experiments are conducted on a cluster of 11 nodes. Each node has a Westmere processor with a dual socket Intel(R) Xeon(R) X5660 CPU (6 cores per socket @ 2.80GHz), equipped with 64KB of L1 cache and 256KB L2 cache per core, 12MB of L3 cache shared, 48GB of RAM, and a RAID-0 of seven 250GB 7500 RPM SATA disks. The cluster runs Spark 2.1.0 and Hadoop HDFS 3.0.1. Spark launches 11 workers, each using 24 cores and 40GB of memory. We distribute all data across the 11 nodes with replication factor 3. All queries are configured to return relative aggregation error per group less than 10%, and no missing groups. Finally, all queries are run from cold OS caches.

Implementation. To have a fair comparison, we integrated all systems to SparkSQL 2.1.0, and extended the Catalyst built-in optimizer accordingly. For Quickr, we implemented the three sampler operators (Distinct, Uniform, Universe) and added all rules described in [25] to Catalyst. For BlinkDB, we followed the algorithms described in [4] to choose the same set of samples that the mixed integer linear program would select for the different workloads. We then generated the samples and executed the queries over that set of samples. Taster was implemented in Scala, over SparkSQL. We integrated Taster's tuner and optimization rules, as well as rudimentary costing capabilities into Spark Catalyst. Both query planner and tuner are centralized and run locally on the driver node of the Spark cluster. We implemented Taster's sketch-join algorithm using the serializable implementation of count-min sketch native to Spark 2.1.0. The uniform sampler is also native to Spark 2.1.0. The distinct sampler operator was implemented as an additional operator over DataFrames, using the algorithm described in Section II. The error estimator for samplers was estimated as described in Section IV-B. For robustness and scalability, all data, metadata, and materialized intermediate summaries of Taster were stored in HDFS, except of the in-memory buffer, which was implemented as persisted RDDs.

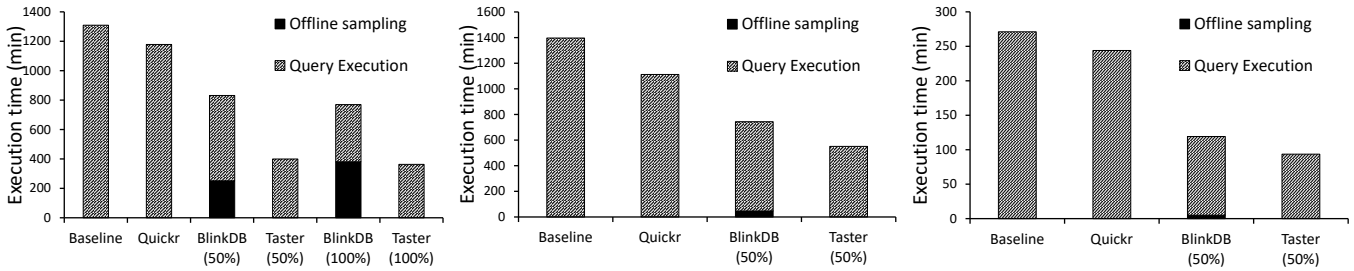
A. Evaluation with different benchmarks and comparison to state-of-the-art AQP engines

Methodology. To compare all systems in a variety of workloads, we execute query sequences over all three datasets. To emulate workload shifts and examine system adaptivity, we instantiate 200 queries from the benchmark templates and issue them in random order. For each benchmark we randomly choose one of the available templates with equal probability (uniformly) and generate a new query by randomly choosing the predicate value. For TPC-H, both Taster and BlinkDB are tested with storage budgets 50% and 100% of the size of the compressed dataset. For TPC-DS and instacart, the queries have fewer prospective stratification attribute sets and require less space for samples. Therefore, we present results only for the 50% storage budget.

End-to-end execution time. Figure 3 presents the required time for executing all 200 queries for each of the workloads. The reported time includes initialization time (i.e., the creation of the samples for BlinkDB). As expected, BlinkDB with only 50% budget requires less time for constructing the samples, but incurs a higher execution time since less queries are approximable by the set of available samples. Specifically, for TPC-H (Figure 3a), BlinkDB 50% offers $2.25\times$ speed-up compared to the Baseline, and requires 251 seconds for pre-computing the sample, whereas BlinkDB 100% offers $3.36\times$ performance increase but spends 380 seconds on sampling. Quickr requires no preprocessing, but offers a smaller performance boost ($1.2\times$). This is attributed mainly to the relevantly shallow queries of TPC-H, as well as the small network congestion of the cluster. Finally, Taster achieves low response time and $\sim 3\times$ speed-up without pre-computing the samples, by adapting to the query workload. We also see that Taster with 50% and 100% storage budget have a similar performance (difference is less than 10%), precisely because the system adapts to the workload and does not require all synopses to be present at all times.

[2]BlinkDB requires all queries to be known a priori, in order to decide on the samples. Therefore, we assumed the existence of an oracle that provides all queries to BlinkDB at initialization time. Clearly, this assumption strongly favors BlinkDB in the comparison.

[3]We used 18 out of the 22 TPC-H templates (Q_2 is not approximable, Q_4, Q_{21} and Q_{22} include `EXISTS` statement which require key of dimension relation thus no gain from approximation).



(a) TPC-H workload (b) TPC-DS workload (c) instacart workload

Fig. 3: End-to-end execution time for different workloads.

The results with TPC-DS and instacart workloads (Figures 3b-c) were qualitatively similar, confirming the applicability of Taster to different data and workload characteristics. In particular, Taster has slightly better performance from BlinkDB, yet without requiring any initialization time. For TPC-DS, this performance improvement is attributed mainly to the capability of Taster to summarize also intermediate results (specifically, the join between tables $store\_sales$ and $date\_dim$, which appears frequently in the workload), rather than only base relations. For instacart, the increased performance of Taster comes from the extensive use of sketches.

Individual performance gains for TPC-H queries. Fig. 4 presents a CDF of the speed-up of Taster for TPC-H queries. Taster slows down less than 10% ($\sim 0.8\times$) of the queries, mostly due to the planning and tuning overhead. However, more than 50% of the queries are being sped-up more than $6\times$. The maximum speed-up ($13\times$) is achieved using sketches.

Approximation error for TPC-H queries. We also verified that the approximations of Taster are within the desired accuracy requirements, with high probability. Figure 5 presents a CDF of the observed aggregation error, for the TPC-H queries. The user requirements for these experiments are: (a) all groups should be detected, and (b) aggregate error should be less than 10%. By employing distinct sampling with stratification guarantees, Taster misses no groups. Furthermore, more than 93% of the queries have error less than 10%, and all queries have error less than 12%. These numbers are very close to the accuracy achieved from BlinkDB with offline sampling.

Summary. Taster substantially outperforms Quickr and offers comparable performance to BlinkDB, yet without requiring a priori knowledge of the workload, and without an offline sample pre-computation. Hence, Taster enables instant access to data while adhering to user accuracy requirements.

B. Adapting to query workload

Methodology. In this experiment, we evaluate the robustness of Taster to workload shifts, i.e., changes in query stratification attributes, the accessed tables, and query predicates. To emulate a real world scenario, we execute a sequence of 80 TPC-H queries, generated from the 18 used query templates by varying the filtering predicates. We split the queries into 4 *epochs* of 20 queries each, based solely on the query execution time, i.e., queries in each group have similar execution time when executed using Baseline. The following templates are used per epoch: (1): q6, q14, q17 (2): q5, q8, q11, q12 (3): q1, q3, q16, q19 (4): q7, q9, q13, q18. As the grouping relies only on query execution time, the queries within each epoch may use different synopses. For example, in epoch (2) template of q5 requires a synopsis with stratification on $orderkey$ whereas template of q8 requires stratification on $partkey$. The storage budget for Taster is set to 35GB.

Figure 6 presents the execution time and storage requirements of Taster at each query. Taster's tuner continuously re-evaluates the synopses stored in the synopsis warehouse, and it frequently drops and build some synopses while executing the queries. At the beginning of each epoch, Taster quickly recognizes the new useful synopses, and makes space for them by evicting the older ones. During the last epoch, the tuner decides to materialize the synopses earlier, since the new synopses provide a higher prospective gain.

Summary. Taster adapts the available synopses to the evolving workload. This enables better space utilization with performance comparable to state-of-the-art offline AQP systems.

C. Adapting the sliding window length to query workload

Methodology. We now evaluate the adaptivity of the tuner in terms of the sliding window length w used for predicting the future queries. We execute a sequence of 200 TPC-H queries, generated by using the 18 query templates. The queries are executed in random order. To evaluate the impact of the adaptive sliding window, the same query workload is executed using three static configurations ($w = 5$, $w = 10$, and $w = 50$), and the adaptive configuration where w changes according to the queries. Storage budget is fixed to 35GB.

Figure 8 presents the cumulative execution time for all queries, for the considered configurations. Taster with adaptive sliding window length starts with window size 5 and increases/decreases according to the correctness of prior predictions. During this experiment the window size fluctuates between 12 and 17, but never converges. This exemplifies the need for an adaptive sliding window length. Among the static window configurations, Taster with window size 10 performs the best, but it is still noticeably slower than the adaptive version. Window sizes 5 and 50 lead to fairly bad performance, i.e., the predictive power of the tuner for future queries is annihilated.

D. Storage elasticity

Methodology. We now investigate how Taster adapts to changing storage budget. We run a sequence of 250 TPC-H queries

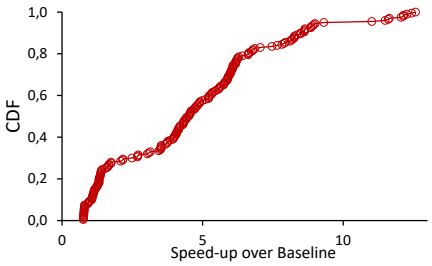


Fig. 4: Individual performance gains

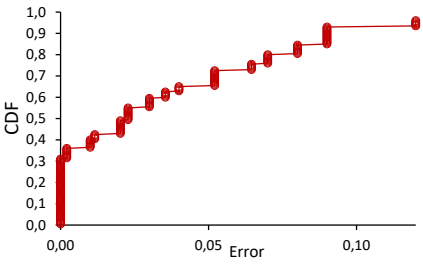


Fig. 5: Approximation error

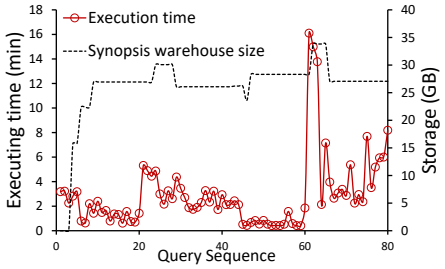


Fig. 6: Taster adapting to query workload

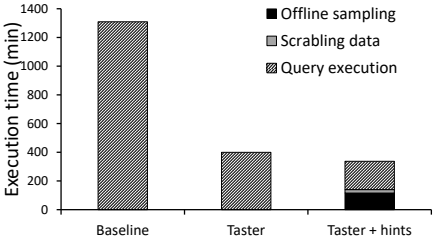


Fig. 7: Performance with user hints

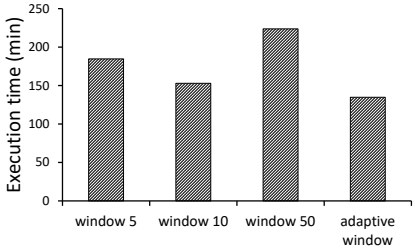


Fig. 8: Varying the horizon size

Fig. 9: Varying the storage budget

in random order, progressively changing the storage budget configuration. To emulate a real world scenario (e.g., adapting the budget to workload), we fluctuate storage budget a lot, to correspond to $20\%, 50\%, 100\%, 50\%, 100\%$ of the dataset.

Figure 9 presents the average speed-up for these configurations compared to Baseline. With 20% of storage, Taster fits only one sample and a sketch, thereby providing very limited approximation potentials. When given 50%, Taster has sufficient space to keep *almost* all synopses, whereas a budget of 100% enables Taster to keep all synopses. When storage allowance is reduced, Taster automatically invokes the tuner to keep the synopses that will maximize the gain, thereby minimizing the performance impact.

E. Utilizing user hints

The final experiment focuses on examining how Taster utilizes user hints to improve performance. The experiment simulates the following scenario: the user already has an idea on the analysis that will be conducted *on one part* of the database (on a subset of the tables) and she advices Taster on the samples that need to be taken, e.g., by listing representative queries, or even by explicitly stating the required samples. In this case, Taster constructs and pins the synopses in the synopsis warehouse offline, and manages the remaining quota online for storing new synopses. We demonstrate this setup by generating two databases – two instances of TPC-H (scale factor 300) – and using Taster to query both, with intervening queries. For the first database, db_{off}, we instruct Taster at initialization for the synopses that need to be created offline (in this case, samples on the $lineitem$ table). For the second, db_{onl}, we let Taster generate and handle the synopses online. Taster is also free to create additional samples for db_{off}, if the precomputed samples do not cover all queries.

Offline sampling in db_{off} follows the state-of-the-art variational subsampling approach of VerdictDB [35]. Notice that this approach requires the following offline steps: (a) creating a shuffled clone of the $lineitem$ table (the scrambled copy), and (b) extracting the samples. We also alter the query execution process to apply variational subsampling. Both databases are queried with 100 queries of TPC-H, i.e., a total of 200 queries, in mixed order. For this experiment, Taster is given a total of 50 GB for synopsis quota.

Figure 7 presents the time spent for answering all 200 queries (denoted as Taster + hints), as well as the time spent in the offline phase. For comparison, the figure also includes the elapsed time for getting exact results (Baseline), and the time for executing the same workload in Taster without hints (Taster). Clearly, hints help Taster to increase query performance, by taking the sampling phase offline. Furthermore, the use of variational subsampling enables the use of smaller samples. In particular, the average speed-up over all queries was $12.6\times$ compared to the baseline, and $4.98\times$ compared to Taster without hints. The speed-up over the queries on db_{off} only (these are the queries using the pre-computed samples) was $20.43\times$ and $9.24\times$ compared to Taster without hints. The construction of the samples using variational subsampling, however, takes a non-negligible amount of pre-processing (116 minutes), delaying the first insights from the dataset. Therefore, a hints-driven offline phase is beneficial when the user knows that a database/table will be frequently queried in the near future; it reduces both query execution time and the size of the generated samples.

VII. Related Work

We identify two lines of work related to Taster: (i) Approximate query processing, (ii) DBMS physical design.

Approximate Query processing. There is a large collection of recent work on approximate query processing. In this context, we separate the work into three categories: (i) offline sampling, (ii) online sampling, and (iii) online aggregation.

sketch-1	$order\_id$, $count(*)$ FROM $orderproducts$ JOIN $orders$ WHERE $o\_order\_dow = \_day\_$ AND $o\_order\_hod > \_hour\_$
sketch-2	$product\_id$, $count(*)$ FROM $orderproducts$ JOIN $products$ WHERE $p\_product\_name = \_productname\_$
sketch-3	$product\_id$, $count(*)$ FROM $orderproducts$ JOIN $products$ JOIN $departments$ WHERE $d\_department = \_department\_$
sketch-4	$product\_id$, $count(*)$ FROM $orderproducts$ JOIN $products$ JOIN $aisles$ WHERE $a\_aisle = \_aislename\_$
sample-1	$product\_id$, $count(*)$ FROM $orderproducts$ JOIN $orders$ WHERE $o\_order\_dow = \_day\_$ AND $o\_order\_hod > \_hour\_$
sample-2	$order\_id$, $count(*)$ FROM $orderproducts$ JOIN $products$ WHERE $p\_product\_name = \_productname\_$
sample-3	$order\_id$, $count(*)$ FROM $orderproducts$ JOIN $products$ JOIN $departments$ WHERE $d\_department = \_department\_$
sample-4	$order\_id$, $count(*)$ FROM $orderproducts$ JOIN $products$ JOIN $aisles$ WHERE $a\_aisle = \_aislename\_$

TABLE I: Instacart micro-benchmark queries. Variables starting and ending with _ are randomly set for query variation.

Offline Sampling. A rich vein of literature exists on sampling strategies for approximate query processing (see, e.g., [12] for an extensive overview). Most approaches assume some degree of knowledge over the upcoming queries, and differ primarily on the nature of samples that they maintain. Congressional sampling, STRAT and BlinkDB [2], [4], [10] provide algorithms to compute the best set of uniform and stratified samples, subject to a storage budget. In the same line, other works maintain additional data structures to better support skewed datasets and to reduce the size of samples [7], [14], [37]. AQUA [3] and VerdictDB [35] instead act as a middleware between users and traditional database systems, by rewriting user queries to take advantage of precomputed samples. VerdictDB is particularly interesting as it proposes a novel error estimation technique called variational subsampling, which enables smaller samples. Similarly, Sample+Seek [15] introduces measure-biased sampling which takes advantage of indexes to create more efficient samples and provide error guarantees for GROUP BY queries with many groups. AQP++ [36] blends AQP with aggregate precomputation, such as data cubes, to handle aggregate relational queries. Such a unified approach balances preprocessing time and query runtime.

However, all offline approaches are designed based on static assumptions about future queries. Thus, they require workload knowledge, and they undergo a time-consuming preprocessing operation for sample preparation. Both are limiting properties of these methods, since in modern data analytics setups (e.g., data exploration), the analyst typically starts with little knowledge about data. Hence, she can hardly predict the future workload, or the time that she will spend analyzing the new data, in order to decide whether an extensive sampling preparation will be beneficial. Furthermore, these methods fail when the actual queries diverge from the predicted workload that was used for constructing the synopses. Taster does not suffer from these constraints since it constructs and adapts the synopses online, during query execution. Still, as we have shown, by integrating with VerdictDB, Taster can capitalize on user hints – when there is such a possibility – to construct some samples in an offline phase, using different sampling strategies. Techniques presented in Sample+Seek [15] and AQP++ [36] are also prime candidates for integration into Taster, to further speed-up query execution by taking advantage state-of-the-art sampling and precomputed aggregates.

Online Sampling. To address the limitations that source from the uncertainty of the future query workload, Quickr [25] follows an online sampling approach, where samples are taken during query execution. In particular, samplers are injected into the query plan to reduce network and computation load. However, Quickr performance gains are constrained by the I/O cost since the system still needs to read the full input for every query. Taster extends the online approximation techniques of Quickr in several non-trivial ways. First, it materializes/stores some of these synopses for re-use in future queries. The decision as to which synopses should be stored relies on a formal model, which enables adaptivity to the workload and to the shifting user interests, and is amenable to efficient approximations. Second, it incorporates additional types of synopses, beyond samples. Finally, it incorporates hints for offline synopsis construction, thereby exhibiting the best properties of both online and offline AQP. Galakatos et al. [18] build and re-use samples for a data exploration scenario. This work, however, assumes that the user builds queries incrementally, allowing the system to generate samples while the user is further expanding his query (e.g., adding a filter). Furthermore, samples are built only over base relations, not taking advantage of intermediary results.

Online aggregation (OLA) [21], [23], [33], [38] offer a different way to approximate. Instead of sampling over data, they estimate the answer by looking at progressively increasing portions of the data, until a user determines that the answer quality is sufficient. EARL [26] and ABS [40] use bootstrapping to produce multiple estimators from the same sample. Finally, iOLAP [39] models online aggregation as incremental view maintenance with uncertainty propagation.

Physical Design. Taster's adaptive nature is influenced by the vast bibliography in adaptive database systems. There exist many recent and influential works on tuning data structures, intermediate result recycling and query matching, as well as eager/lazy aggregation. Specifically, there has been a plethora of work on automatic selection of materialized views [16], [22], [28], [31], indices [30], [41], or both [5] and most modern databases come with tuning tools. All these works, however, assume known workloads. Since future queries are often unknown in advance, these tools optimize for past queries as approximations of future ones. As a result, they may produce designs that are sub-optimal in practice. To mitigate some of these problems, a few heuristics have been proposed to summarize the workload [11] as well as adaptive approaches [9], [20], [32] which adjust indices and views incrementally, on demand. Taster follows a similar *adaptive* approach in the context of deciding and materializing synopses. It exploits constant monitoring, forecasting using a sliding window approach, and a formal model to evaluate subplans, decide on the best subplan for each query, and

materialize the required synopses.

VIII. Conclusions

Approximate query processing engines – both offline and online – gained significant interest in the last years, as they offer low latency data analytics in return to an acceptable, slightly relaxed precision of the results. However, the ever-growing data sizes combined with the need of today's data scientists to get immediate insights out of big data, introduce a new set of challenges to these systems. On the one hand, offline approximation approaches require long pre-processing and knowledge of the expected workload, and have high storage requirements. On the other hand, online approaches require reading all data for each query in order to collect samples, hence offering much smaller performance gains. In this paper, we demonstrate Taster, a system that adaptively combines the two approaches. Synopses in Taster summarize both base relations and intermediary results (frequent sub-plans). They are generated in an online fashion, as byproducts of the queries, but they can also be saved and reused across several queries similar to offline AQP engines. Importantly, the stored synopses transparently adapt to the ever-shifting user workload, without user intervention, and without requiring a priori knowledge of the query workload. Finally, Taster can also integrate hints, e.g., for creating some samples offline, thereby further reducing query latency. A thorough evaluation of Taster using three industry-standard benchmarks demonstrates that it adapts to variations in workload and storage, and it outperforms both online and offline AQP approaches, without requiring a priori knowledge of the query workload.

Acknowledgments. We would like to thank the reviewers for their valuable comments. This work has received funding from the EU Horizon 2020 research and innovation programme under grant agreement No 825041, the EU FP7 (ERC-2013-CoG), Grant 617508 (ViDa), the EU Horizon 2020 Research and Innovation Grants 650003 (Human Brain project) and the Marie Sklodowska-Curie grant agreement 665667 (MSCA-COFUND-2017, AQuViDa).

References

[1] The Instacart Online Grocery Shopping Dataset 2017. https://www.instacart.com/datasets/grocery-shopping-2017. accessed 28-May-2018.

[2] S. Acharya, P. B. Gibbons, and V. Poosala. Congressional Samples for Approximate Answering of Group-by Queries. In *SIGMOD*, 2000.

[3] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. The Aqua Approximate Query Answering System. In *SIGMOD*, 1999.

[4] 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.

[5] S. Agrawal, S. Chaudhuri, and V. Narasayya. Materialized View and Index Selection Tool for Microsoft SQL Server 2000. In *SIGMOD*, 2001.

[6] N. Alon, Y. Matias, and M. Szegedy. The Space Complexity of Approximating the Frequency Moments. *J. Comput. Syst. Sci.*, 1996.

[7] B. Babcock, S. Chaudhuri, and G. Das. Dynamic Sample Selection for Approximate Query Processing. In *SIGMOD*, 2003.

[8] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM*, 1970.

[9] S. Chaudhuri and N. Bruno. An Online Approach to Physical Design Tuning. In *ICDE*, 2007.

[10] S. Chaudhuri, G. Das, and V. Narasayya. A robust, optimization-based approach for approximate answering of aggregate queries. In *SIGMOD*, 2001.

[11] S. Chaudhuri, A. K. Gupta, and V. Narasayya. Compressing SQL Workloads. In *SIGMOD*, 2002.

[12] G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches. *Found. Trends databases*, 2012.

[13] G. Cormode, S. Muthukrishnan, and I. Rozenbaum. Summarizing and mining inverse distributions on data streams via dynamic inverse sampling. In *VLDB*, 2005.

[14] M. Datar, R. Motwani, V. Narasayya, G. Das, and S. Chaudhuri. Overcoming Limitations of Sampling for Aggregation Queries. In *ICDE*, 2001.

[15] B. Ding, S. Huang, S. Chaudhuri, K. Chakrabarti, and C. Wang. Sample + Seek: Approximating Aggregates with Distribution Precision Guarantee. In *SIGMOD*, 2016.

[16] X. Ding and J. Le. Adaptive projection in Column-stores. In *FSKD*, 2011.

[17] P. Flajolet and G. N. Martin. Probabilistic Counting Algorithms for Data Base Applications. *J. Comput. Syst. Sci.*, 1985.

[18] A. Galakatos, A. Crotty, E. Zgraggen, C. Binnig, and T. Kraska. Revisiting Reuse for Approximate Query Processing. *VLDB*, 2017.

[19] J. Goldstein and P.-A. Larson. Optimizing Queries Using Materialized Views: A Practical, Scalable Solution. In *SIGMOD*, 2001.

[20] G. Graefe and H. Kuno. Adaptive indexing for relational keys. In *ICDEW*, 2010.

[21] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online Aggregation. In *SIGMOD*, 1997.

[22] R. Huang, R. Chirkova, and Y. Fathi. Two-stage stochastic view selection for data-analysis queries. In *ADBIS*, 2013.

[23] C. Jermaine, S. Arumugam, A. Pol, and A. Dobra. Scalable Approximate Query Processing with the DBO Engine. *Trans. Database Syst.*, 2008.

[24] S. Kandula. Errata and Proofs for Quickr. Technical report, 2016.

[25] S. Kandula, A. Shanbhag, A. Vitorovic, M. Olma, R. Grandl, S. Chaudhuri, and B. Ding. Quickr: Lazily Approximating Complex Ad-Hoc Queries in Big Data Clusters. In *SIGMOD*, 2016.

[26] N. Laptev, K. Zeng, and C. Zaniolo. Early Accurate Results for Advanced Analytics on MapReduce. In *VLDB*, 2012.

[27] J. Leskovec, A. Krause, C. Guestrin, C. Faloutsos, J. M. VanBriesen, and N. S. Glance. Cost-effective outbreak detection in networks. In *SIGKDD*, 2007.

[28] W. Liang, H. Wang, and M. E. Orlowska. Materialized view selection under the maintenance time constraint. *DKE*, 2001.

[29] S. Lohr. *Sampling: Design and Analysis*. 2009.

[30] C. Maier, D. Dash, I. Alagiannis, A. Ailamaki, and T. Heinis. PARINDA: An Interactive Physical Designer for PostgreSQL. In *EDBT*, 2010.

[31] B. Mozafari, E. Z. Y. Goh, and D. Y. Yoon. CliffGuard: A Principled Framework for Finding Robust Database Designs. In *SIGMOD*, 2015.

[32] M. Olma, M. Karpathiotakis, I. Alagiannis, M. Athanassoulis, and A. Ailamaki. Slalom: Coasting through raw data via adaptive partitioning and indexing. *VLDB*, 2017.

[33] N. Pansare, V. Borkar, C. Jermaine, and T. Condie. Online aggregation for large mapreduce jobs. In *VLDB*, 2011.

[34] O. Papapetrou, W. Siberski, and W. Nejdl. Cardinality estimation and dynamic length adaptation for bloom filters. *DAPD*, 2010.

[35] Y. Park, B. Mozafari, J. Sorenson, and J. Wang. VerdictDB: Universalizing Approximate Query Processing. In *SIGMOD*, 2018.

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

[37] L. Sidirourgos, M. Kersten, and P. Boncz. SciBORQ: Scientific data management with Bounds On Runtime and Quality. In *CIDR*, 2011.

[38] K. Zeng, S. Agarwal, A. Dave, M. Armbrust, and I. Stoica. G-OLA: Generalized On-Line Aggregation for Interactive Analysis on Big Data. In *SIGMOD Demo*, 2015.

[39] K. Zeng, S. Agarwal, and I. Stoica. iOLAP: Managing Uncertainty for Efficient Incremental OLAP. In *SIGMOD*, 2016.

[40] 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*, 2014.

[41] D. C. Zilio, J. Rao, S. Lightstone, G. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden. DB2 Design Advisor: Integrated Automatic Physical Database Design. In *PVLDB*, 2004.