SmartDataLake

DELIVERABLE D2.2

Data synopses for approximate analytics

PROJECT NUMBER: 825041
START DATE OF PROJECT: 01/01/2019
DURATION: 36 months

Horizon 2020

SmartDataLake is a Research and Innovation action funded by the Horizon 2020 Framework Programme of the European Union.

Dissemination Level	Public
Due Date of Deliverable	Month 16 (30/04/2020)
Actual Submission Date	27/04/2020
Work Package	WP2: Adaptive Data Virtualization and Storage Tiering
Tasks	Task 2.2: Data synopses and approximate query processing
Type	Report
Lead Beneficiary	Eindhoven University of Technology
Approval Status	Submitted for approval
Version	1.0
Number of Pages	41
Filename	SmartDataLake-D2.2-Data_synopses_for_approximate_analytics.pdf

Abstract

This report presents our work on approximate query analytics in SmartDataLake. In particular, we describe a query approximation layer (QAL) for OLAP queries built over Spark that is offered as a web service to all SmartDataLake components. We elaborate on the synopses that are already integrated in QAL (both samples and sketches) and explain how these are maintained and used for query answering. Our main innovation involves a novel self-tuning query engine that: (a) decides which synopses should be constructed to maximize the query throughput, and constructs them transparently from the user, and (b) integrates these synopses in the query plan, and uses them to estimate the answers. Finally, we describe our current directions for improving the construction of synopses, and the way the query plans are constructed.

History

Version	Date	Reason	Revised by
0.1	26 Feb 2020	First draft	Hamid Shahrivari Joghan
0.2	13 Mar 2020	Second draft	Hamid Shahrivari Joghan
0.3	31 Mar 2020	First version for internal review	Hamid Shahrivari Joghan
1.0	27 Apr 2020	Final version for submission	Hamid Shahrivari Joghan

Author List

Organization	Name	Contact information
TUE	Odysseas Papapetrou	o.papapetrou@tue.nl
TUE	Hamid Shahrivari Joghan	h.shahrivari.joghan@tue.nl
TUE	George Fletcher	g.h.l.fletcher@tue.nl
TUE	Nikolay Yakovets	n.yakovets@tue.nl
TUE	Larissa Shimomura	l.capobianco.shimomura@tue.nl
EPFL	Anastasia Ailamaki	anastasia.ailamaki@epfl.ch
EPFL	Bikash Chandra	bikash.chandra@epfl.ch
RAW-LABS	Benjamin Gaidioz	ben@raw-labs.com
ATHENA	Dimitris Skoutas	dskoutas@athenarc.gr

Executive Summary

Approximate query processing (AQP) enables applying a trade-off between accuracy and performance, in order to provide the user with a real-time response. State-of-the-art AQP engines rely on constructing compact summaries of data, named synopses, and approximating the query answers on these summaries. Examples of synopses include samples, histograms, and sketches. This report presents our work on constructing an AQP engine for SmartDataLake (SDL), where, due to the extreme data volume, data exploration and analytics become very slow.

In Section 1, we briefly motivate the need for AQP on SDL. We present an overview for the three different categories of AQP, and argue that none of them fulfils the requirements for data exploration in SDL.

In Section 2, we cover some basic concepts and related studies. Firstly, we study state-of-the-art AQP engines and outline their strengths and weaknesses. We then discuss synopses, the power horse of all AQP engines, and explain their functionality. We focus on both samples and sketches, which are exploited in our work. Next, we provide an overview of the Apache Spark platform, emphasizing on SparkSQL and the Catalyst optimizer.

The SmartDataLake AQP solution, called Query Approximation Layer (QAL), is introduced in Section 3. QAL is an adaptive approximate query processing engine inside SDL-Virt. Implemented on top of SparkSQL and Catalyst, QAL scales out on hundreds of machines. First, we present the big picture of QAL and how each component interacts with others. The main innovation behind QAL compared to other state-of-the-art AQP engines is its self-tuning nature: QAL automatically adapts to user behaviour by *constructing and maintaining* reusable synopses. To enable this adaptivity, the planner generates multiple *approximate physical plans* in which synopses are injected as physical operators, and then it executes the one that maximizes *the expected throughput for future queries*, as opposed to maximizing the performance of a single query, which is the approach of the state-of-the-art AQP engines. Leveraging a variety of synopses, the planner considers various approximate execution plans for each query. To preserve the accuracy of plans, it obtains rules for integrating synopses into the plans. We also discuss the two approaches of QAL for forecasting future queries, which is a critical part of estimating the expected throughput: (a) a window-based query prediction and (b) a novel ML model predicting approximate operators.

In Section 4, we focus on the SDL ecosystem, explaining how other components of SDL can connect to QAL through a web service interface. This setup enables each layer of SDL to rapidly import its data into QAL and submit approximate data analytics queries. The bridge opening the QAL functionality to the other SDL components is a REST API web service.

In Section 5, we present our preliminary experiments and results that demonstrate the efficiency of the proposed QAL.

Abbreviations and Acronyms

API	Application Programming Interface
QAL	SmartDataLake's Query Approximation Layer
AQP	Approximate Query Processing
BF	Bloom Filter
CMS	Count-min Sketch
CLT	Central Limit Theorem
CPU	Central Processing Unit
CSV	Comma-Separated Values
ELP	Error-Latency Profile
FM	Flajolet-Martin Sketch
HDFS	Hadoop Distributed File System
HIN	Heterogeneous Information Network
JSON	JavaScript Object Notation
ML	Machine Learning
MPP	Massively Parallel Processing
LSTM	Long Short-term Memory
OLA	Online Aggregation
OLAP	Online Analytical Processing
QCS	Query Column Set
RAM	Random-access Memory
RDD	Resilient Distributed Dataset
SDL	Smart Data Lake
SQL	Structured Query Language
URL	Uniform Resource Locator

Table of Contents

1. Introduction

SmartDataLake focuses on extreme-scale analytics over data lakes. The two main technical challenges for data management involve: (a) addressing data heterogeneity, which is inherent in data lakes (Task 2.1), and (b) addressing the ever-increasing data volume (Tasks 2.1 and 2.2).

Task 2.2 focuses on using query approximation strategies to increase querying performance, with a small, controllable loss of accuracy. The key motivation behind query approximation is that a slight relaxation in terms of accuracy can lead to tremendous improvements in scalability and response times. SmartDataLake leverages this idea by integrating a fully-fledged AQP engine over the data virtualization layer. The engine integrates various types of data synopses.

This document describes our progress with respect to SDL's AQP engine. We start with a high-level overview of approximate query processing (AQP) for big data. Then, we focus on the role of AQP in SmartDataLake, i.e., which project components can utilize AQP and what type of AQP is desired in the project. Section 2 describes the preliminaries and related work. Our scientific contribution, which comprises a self-tuning Query Approximation Layer (QAL), is presented in Section 3. Section 4 focuses on the technical aspects of integrating QAL with the other SmartDataLake components. We conclude with highlights of our experimental results (Section 5), and conclusions (Section 6).

1.1. Approximate query processing for big data analytics

Online analytical processing (OLAP) is today offered by all commercial database systems. Users can exploit OLAP either through direct SQL queries (cubes, slicing and dicing, etc.) or through third-party software that uses a data management system for accessing the data. It is a core functionality for data-driven decision making, e.g., in the context of a decision support system or a data analytics environment, and it is tightly integrated with data visualization platforms.

As such, performance and scalability of OLAP engines is imperative for facilitating near-real-time decisions. However, it is rather costly to support OLAP on big data. Even the state-of-the-art methods that compute exact answers cannot meet the high performance requirements for interactive analytics over big data. To alleviate this problem, approximate query processing (AQP) offers a way to trade accuracy with performance. AQP is particularly suited for aggregate queries, such as counts, sums, and averages. Existing AQP techniques can be broadly classified into three categories: (a) online AQP, (b) offline AQP, and, (c) online aggregation systems. Most of the state-of-the-art techniques rely on *synopses*, i.e., data structures that can compactly summarize the data and support estimation of aggregate queries. Examples of synopses include samples, histograms, and sketches.

Online AQP systems construct the synopses during the query execution stage, thereby optimizing them for the query at hand. The lifetime of these synopses is therefore typically limited to a single query. Instead, offline AQP rely on a priori knowledge (e.g., knowledge of the expected query workload), and use this to decide which synopses would be the most beneficial for increasing the overall query throughput. In this case, the synopses are constructed at a preparation phase, and are exploited for supporting more than one queries. Finally, online aggregation, which is mostly used for stream processing, provides continuous and real-time estimates of the query while

observing more data. In the remainder of this section, we elaborate on the main representatives of these three categories, and discuss their advantages and disadvantages.

Offline AQP: Offline AQP engines start by analysing the expected workload to identify the optimal set of synopses that should be generated to provide fast responses, subject to a predefined storage budget and error tolerance specification.

There are several algorithms and fully-fledged systems for offline AQP. Congressional sampling [1], STRAT [2] and BlinkDB [3] provide algorithms to compute the best set of 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 [4][5][6]. AQUA [7] and VerdictDB [8] instead act as a middleware between users and traditional database systems by rewriting user queries to take advantage of precomputed samples. Similarly, Sample+Seek [9] 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++ [10] blends AQP with aggregate precomputation, such as data cubes, to handle aggregate relational queries. Furthermore, the most recent offline AQP engines [2][8][11] are constructed over massively parallel processing platforms, e.g., Spark, combining horizontal scalability with fast estimates.

By enabling the query engine to drastically reduce I/O, offline AQP engines typically offer massive performance improvements for the queries that can be answered with the prepared synopses. However, the choice of which synopses should be constructed is not always easy, since it requires some kind of 'prediction' of which synopses will be useful for the future queries. Offline AQP engines build on the premise that the query workload is predictable, which makes them unsuitable for unpredictable and dynamically changing workloads. Data exploration is one such example, where future queries are determined based on the results obtained from past queries.

Online AQP: Online AQP engines address the challenge of the dynamic workload by introducing approximation at the time of query execution. The optimization goal of these query engines is to construct the optimal synopses for speeding-up the query at hand. State-of-the-art online AQP engines achieve this goal by introducing samplers within the query plan, in order to reduce the tuples that need to be considered [12][13]. However, unlike offline AQP, the samplers are introduced at the runtime and injected in the execution plan, after scanning the raw data and before intermediate operations. As a result, samplers improve the computational performance of the operators at higher levels of the query plan, e.g., expensive joins, but do not avoid going through the base relations for each input query.

Even though online AQP can, in principle, be applied to all queries and does not require a priori knowledge about the query workload, it brings substantially less performance boost compared to offline AQP, since it still requires at least one full access over the raw data, in order to generate the synopses. Query-time sampling is by definition limited in the scope of a single query, as the generated samples are not constructed with the purpose of reuse across queries. Instead, they are specific to the query at hand, and they are typically discarded after the query has been answered. A notable exception is the recent work of Galakatos et al. [13], which rewrites the incoming query so that it can reuse previously created samples; however, ignoring the nature of the workload, it does not adapt the query plans to future queries.

Online aggregation: Online aggregation was first introduced in the seminal work of Hellerstein et al. [14] and offers a fundamentally different way of approximating: instead of sampling over the data, it estimates the answer by looking at progressively increasing portions of the data, until a user determines that the answer is sufficiently accurate. Online aggregation techniques rely on the assumption that data is observed at a random order, in order to estimate the accuracy bounds of the answers. Recent query engines that support online aggregation [15][16][17] mainly differ on the way they produce the accuracy bounds. For example, ABS [16] uses bootstrapping to produce

multiple estimators from the same sample, whereas iOLAP [17] models online aggregation as incremental view maintenance with uncertainty propagation.

1.2. Approximate query processing in SDL

Before delving into the details of the query approximation layer (QAL), we briefly discuss how query approximations can be used to boost performance of each of the three main components of SmartDataLake: SDL-Virt, SDL-HIN and SDL-Vis (Figure 1). Furthermore, we summarize the challenges these requirements bring for the approximation layer.

AQP in SDL-Virt: SDL-Virt offers efficient data analytics over big heterogeneous data. The data analyst can execute SQL and SQL-like queries over different data formats and data sources that are possibly distributed across different systems and networks. However, big data analytics typically do not require answers that are accurate until the last decimal; in most cases, insights can be extracted even with approximate results, as long as the user can control the accuracy guarantees. The QAL is constructed within SDL-Virt, adding to the functionality of the query engine developed in Task 2.1 by offering the capability of approximate queries. It can be accessed by other components through a simple REST API. Furthermore, to enable transparent interoperability with other components that are already prepared to use SQL interfaces, it supports SQL queries, with a small extension for declaring the user's desired accuracy guarantees.

AQP in SDL-HIN: SDL brings forth novel and scalable methods and algorithms to explore and mine data lake's content with the help of analytics and mining over a dynamic HIN. SDL-HIN component can use the QAL to access data stored in a relational format. Typical queries (in natural language) that can be used for enriching and exploring the HIN are, for example:

- find the number of connections between two nodes, in order to determine the weight of the nodes
- find the number of nodes in the HIN satisfying a particular property
- find the number of edges between two selected HIN subgraphs

AQP in SDL-Vis: Data scientists rely on the guidance of interactive data visualization engines, in order to analyze big data efficiently and effectively. Clearly, response time is critical to maintain the user connected to the analysis task: users get distracted easily and lose focus when the answer delays for more than a few seconds. Accordingly, most visualization engines strive to provide results within a few seconds, or explicitly handle the latency in a way that does not negatively affect user's cognition[18][19]. However, as the data size grows, systems providing exact results simply fail to respond with a reasonable latency. Several recent studies improve this latency by exploiting AQP [20]–[23], yet on more controlled configurations, i.e., not in a data lake, and not in a self-tuning nature.

Our approach in SDL is to use the query approximation layer to provide near-interactive answers to the user. SDL-Vis can execute SQL-like queries and receive approximate results in a few seconds, if suitable synopses are already constructed. Furthermore, QAL continuously adapts to the arriving queries by constructing new synopses on previously unexplored data, thereby optimizing the future queries.

Novel challenges for the Query Approximation Layer: The highly dynamic and distributed nature of SmartDataLake leads to unique challenges related to Approximate Query Processing, some of which we already addressed in the first part of the project:

a) Automating synopses creation process, i.e. choosing which synopses should be constructed to speed-up future analytics. Such a self-tuning nature is critical for a multi-user system, and particularly for enabling data exploration, where the query types and the data ranges of interest can change significantly within a data exploration task.

b) Offering a layered synopses storage, i.e., distinguishing between different storage devices/locations, and deciding which synopses to store on each layer. Current AQP engines rely on the assumption that synopses are sufficiently small to fit on either RAM or the local hard disk, which however no longer holds in the era of data deluge. On the other hand, advances in hardware, software, and networking bring different data storage layers, each with different properties (e.g., RAM, local hard disk of each node, distributed storages). Following recent results [24], QAL will carefully utilize these storage layers to bring a substantial performance boost.

c) Combining multiple synopses to answer complex analytics tasks. Existing AQP engines are restricted to the use of a single synopsis (or single type of synopses) per query plan. By enabling the combination of multiple synopses and synopsis types in the same query plan (potentially even to approximate the same operator) QAL can produce more efficient query plans and can better reuse synopses.

d) Identifying the changing part of the data and reflecting the changes to synopses. Even though most synopses are focused on summarizing data streams, they assume an append-only model: the new data is added on the old data. Since QAL will be reading the data from disk, it needs to be able to identify the data updates and reconstruct the synopses only for the updated data segments.

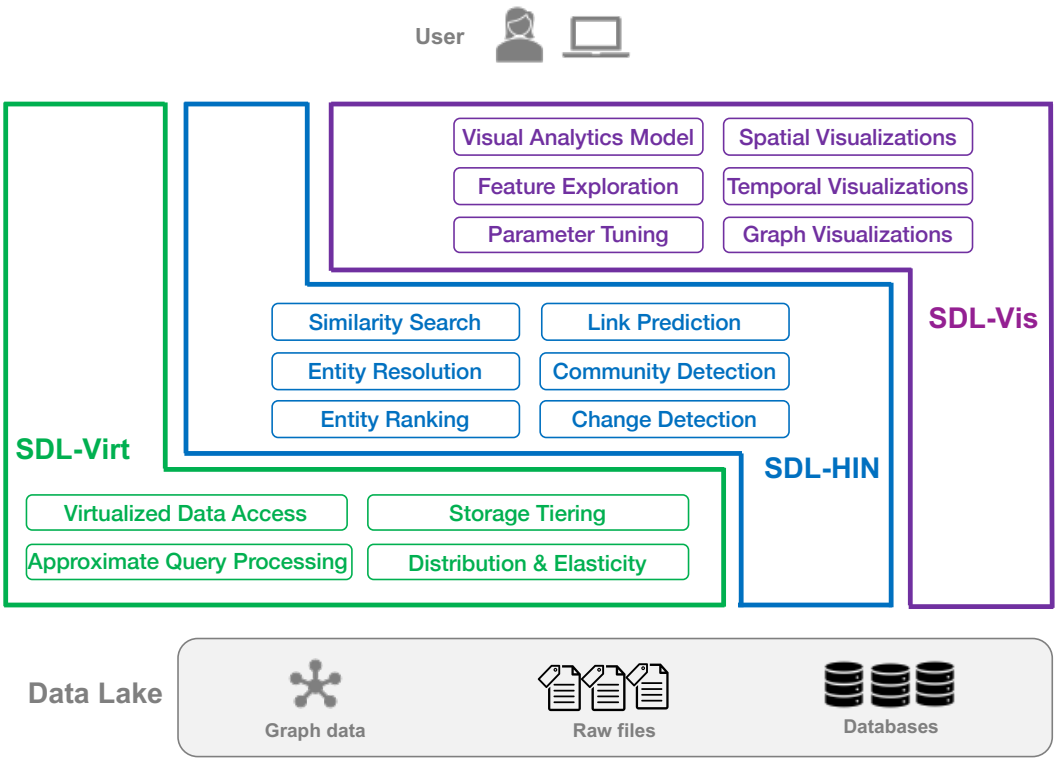


Figure 1: The three layers of SDL: SDL-Virt, SDL-HIN, and SDL-Vis.

1.3. The silver bullet: adaptive approximate query processing

The SDL Query Approximation Layer (QAL) is an elastic, adaptive AQP engine that *synergistically combines the benefits of online and offline AQP*. QAL 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. QAL is implemented over Apache SparkSQL and extends Catalyst query optimizer and SparkSQL query engine with the aforementioned functionality.

To illustrate the utility of QAL, let us consider an example use case. Visual analytics is a core process in data exploration, facilitating extraction of useful insights out of big data. A data scientist typically starts by running simple exploratory queries over the data and visualizing the results, formulating and validating hypotheses. Queries are not known a priori, since each query typically depends on the results of the previous queries. For example, the results of one query may hint the user to zoom in, or to analyze further a region of the data as the next query. In this case, offline AQP engines cannot be used, since they require a priori knowledge of the query load in order to prepare the synopses. On the other hand, online AQP engines offer a substantial performance improvement compared to not using approximation at all; however, online AQP engines do not support reusability of approximations across queries (e.g., if two queries have an overlapping sub-plan).

The ideal situation is to start building the synopses as byproducts of the queries and save these synopses such that they can be reused in future queries. A synopsis can be built on a base relation (a table), or even on intermediary result, e.g., the results of a join, or even the results of a filter. Clearly, creating and saving a synopsis incurs a cost, so the decision of constructing synopses bounded to space quota is taken by the query engine, considering the utility of each synopsis and the frequency of use at the future queries. Furthermore, the storage budget for synopses can be increased or reduced in order to anticipate an increase in data, query load (number of users), and available hardware.

2. Prerequisites and Related Work

This section covers some basic concepts related to QAL and summarizes the current state of the art. We start with a discussion of related work, in Section 2.1, going over state-of-the-art approximate query processing engines and outlining their strengths and weaknesses. We then continue with the prerequisites. Specifically, in Sections 2.2 and 2.3 we discuss synopses (samples and sketches, respectively), focusing on the ones that are already integrated (or planned to be integrated) in QAL, and explain their functionality. In Section 2.4 we provide an overview of the Apache Spark platform, which is the platform of choice for scaling out QAL. In this context, we discuss SparkSQL and the Catalyst optimizer, both of which we extend for QAL.

2.1. State of the art in approximate query processing

We now look at three representative AQP engines: (a) BlinkDB (offline AQP), (b) Quickr (online AQP), and (c) iOLAP (online aggregation). We explain the key decisions of each engine and how these influence its scalability and performance.

2.1.1. BlinkDB

BlinkDB [3] is a distributed sampling-based approximate query processing engine that supports SQL-based aggregation queries over stored data. It enables users to fine-tune the trade-off between performance (the query response time) and accuracy constraints (the error bound of the approximate result). BlinkDB relies on Apache Spark for distributed execution of the query, and on Hadoop HDFS for robust and scalable data storage.

As a result, queries over multiple terabytes of data can be answered in seconds, accompanied by meaningful error bounds relative to the answer that would be obtained if the query ran on the whole data. Compared to other offline AQP, BlinkDB supports more general queries as it makes no assumptions about the attribute values in the WHERE, GROUP BY, and HAVING clauses, or the distribution of the values used by aggregation functions. The key observation is that BlinkDB only assumes that query column sets (QCS) used by queries in WHERE, GROUP BY, and HAVING clauses, are stable over time.

Fundamentally, BlinkDB comprises two main modules:

1. Sample creation: a stratified sampling strategy that builds and maintains a variety of samples.
2. Sample selection: a run-time sample selection strategy that leverages parts of a sample to estimate query selectivity and chooses the best samples for satisfying query constraints.

The sample creation module creates stratified samples on the most frequently used QCSs so that BlinkDB can answer queries about any subgroup, regardless of its representation in the underlying data. The frequent QCSs are identified by abstracting workload information so that it can get a meaningful estimation of the workload distribution. However, since BlinkDB generates offline samples based on the assumption that the workload is stable over time, it does not perform well for queries whose QCS is not covered by the prebuilt samples. If the distribution of QCSs remains stable over time, BlinkDB creates samples that are neither over- nor under-specialized for the query workload. Additionally, allocating the available space to the constructed samples is formulated as an optimization problem that minimizes the loss of accuracy from samples for the overall workload distribution. Based on the collection of past QCS and their historical frequencies, it chooses a collection of stratified samples limited to user-configurable space quota. These samples are chosen to efficiently answer queries with the same QCSs as past queries and to provide good coverage for future queries over similar QCS.

The second module, the sample selection, selects samples to answer the arriving queries. By executing the query on multiple smaller sub-samples, BlinkDB quickly chooses the best set of samples to satisfy specified response time and error bounds. It leverages Error-Latency Profile (ELP) heuristic, the accuracy and latency of a query executed on pre-computed samples, to efficiently choose the sample that will best satisfy the user-specified error or time bounds. BlinkDB has proposed an efficient mechanism to consult the ELPs to find an appropriate sample when a new query arrives with an accuracy and performance target. It also tries to share samples among

different QCSs. For example, a sample for columns (Age, Salary) can also cover the queries for column (Salary).

In a nutshell, BlinkDB offers a distributed approximate query processing engine that constructs samples based on user hints, e.g. the past query workload. In practice, since it relies on pre-computed samples, it enables fast and error-bounded query answers, as long as the arriving queries are of the same type as the past query workload. However, this engine becomes inefficient as the nature of input queries dynamically changes, which is very common in data exploration tasks.

2.1.2. Quickr

Quickr [12] is an online AQP engine that offers a new way to lazily approximate complex SQL queries, without a priori knowledge. The main idea in Quickr is to modify the sampling process such that the samples are pushed down in the execution plan as deep as possible below joins and filters. Consequently, it decreases the number of records sent to upper levels of the execution plan. For example, a pair join requires two passes over data and one shuffle across the network. If data were sampled in the first pass, all subsequent computations could be sped up. At a high level, Quickr strives to achieve the following goals:

1. Given an SQL query, automatically decide whether it can be answered using samples, and output an appropriate query plan with samplers.
2. Support complex queries.
3. Ensure that answers will be accurate and that all groups will be included (i.e., if a group by clause is present).

Quickr utilizes three types of samples: the standard uniform sampler, the distinct sampler that guarantees no groups will be missed, and a new sampler called the universe sampler. The universe sampler is particularly important for joins, as it enables Quickr to sample both join relations in parallel, such that joins performed with these samples yield an approximate result with negligible degradation in accuracy. The universe sampler supports equi-joins, as long as the group-by columns and the value of the aggregates are uncorrelated with the join keys. All samplers operate in a single pass over the data, with bounded memory, and can be run in parallel. Furthermore, Quickr guarantees that query plans with samplers do not miss groups and the estimated aggregates are within a small ratio of their true value.

Quickr introduces an algorithm for deciding which samplers to construct, and where to inject the sampler in the query plan. Quickr starts by placing a sampler below every aggregation (which are typically at high positions in the query plan). Next, by pushing samplers closer to the raw data and before other database operators such as joins, selects, and projects, it generates multiple approximate plans. In the end, Quickr picks the best performing plan among all generated plans and sends it for execution.

Without any a priori knowledge of the past workload or user's hints, Quickr provides approximate answers based on real-time constructed samples. However, Quickr fails to store and reuse the samples across queries, thereby suffering from high I/O for creating the optimal samples from scratch at each query.

2.1.3. iOLAP

iOLAP [17] is an incremental OLAP query engine that provides a smooth trade-off between query accuracy and latency. iOLAP covers a full range of user requirements, from approximate but timely

query execution to exact query execution. iOLAP offers interactive incremental query processing built on a novel mini-batch execution model. In the beginning, iOLAP randomly partitions the input dataset into smaller sets called mini-batches. Then, the system presents the user an approximate result with an associated error estimate as soon as it has processed the input query on a mini-batch. Concurrently, the system keeps reading larger and larger numbers of mini-batch, refining the approximate query results and updating the user. This process continues until either the user is satisfied with the accuracy of the query results and stops the query, or the system has processed all the data.

Given an OLAP query, iOLAP automatically rewrites the query into an enhanced delta query and executes the delta query on each mini-batch of data. However, instead of executing the query on all mini-batches, iOLAP applies a delta query on each new batch and updates the previous results. The key idea behind iOLAP is a new delta update algorithm that tracks uncertainties in the partial results of operators in a query. The uncertainty is categorized to two levels: tuple uncertainty and attribute uncertainty. At each batch, all the uncertain values from the previous batch need to be recomputed and brought up-to-date with the new data. Based on this, iOLAP focuses on minimizing the recomputation on uncertainties that would change. Lastly, iOLAP delta update algorithm relies on an efficient bootstrap-based error estimation, which can be applied to arbitrary user-defined aggregates.

Functionality of the iOLAP delta update algorithm is limited to positive relational algebra queries, i.e., any query that can be composed using relational operators SELECT, PROJECT, JOIN, UNION, and AGGREGATE. However, it does not consider queries that have approximate join/group-by keys under sampling.

2.2. Samples

Execution of OLAP queries over very big relations is destined to take too much time, simply due to the necessary iteration over all records for computing the aggregates. To speed-up these queries, most AQP engines rely on samples. Precisely, small samples are taken over all tables participating in a query plan, either in a preparatory –offline– phase, or when the query arrives. Then, the query plan is executed over the samples which are much smaller than the full data. Therefore, the query execution cost is drastically reduced.

The literature includes several generic sampling algorithms, each coming with its own properties and ways to provide accuracy guarantees. In the following, we go over the two most frequently used sampling algorithms: (a) uniform sampler, (b) distinct sampler.

Uniform sampler (without replacement): The uniform sampler selects rows (without replacement), letting a row pass through with probability p at random. This sampler is both pipeline-able and partition-able, and its memory footprint during construction is approximately equal to the memory footprint of the desired sample size. Even though the uniform sampler has low execution overhead, it does not have good statistical properties in more complex workloads (e.g. join queries) since it may miss an arbitrarily large number of join keys.

Distinct sampler: Distinct sampler 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 A, a number δ, and probability p, the distinct sampler passes at least δ rows for every distinct combination of values of the columns in A. Subsequent rows with the same value are let through with probability p. 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 $1/p$.

2.3. Sketches

Sketches are small-size summaries of data designed for massive, rapid-rate data streams processed either in a centralized or distributed environment. Due to their attractive space and computational complexity, in the last decade, sketches have found their way to AQP, and they evolved as the premier approximation technique for aggregate queries. Each sketch is optimized to support a family of queries. Sketches are parameterized with one or more parameters, which determine the accuracy of the sketch – and in effect, also its size. To execute a query on a sketch, we perform a query-specific procedure on the sketch in order to obtain an approximate answer.

In the following, we describe the basic sketching techniques: Count-min sketches and Bloom filters. Count-min sketches are already integrated in the QAL, whereas Bloom filters will be integrated in the second part of the project. We also briefly summarize other synopses that we are considering for integration in SmartDataLake's query approximation layer.

2.3.1. Count-min sketch

Count-Min sketches are a widely applied sketching technique for *data streams*. A Count-Min sketch is composed of a set of d hash functions, $h_1(\cdot)$, $h_2(\cdot)$, . . ., $h_d(\cdot)$, and a 2-dimensional array of counters of width w and depth d. Hash function h_j corresponds to row j of the array, mapping stream items to the range of $[1...w]$.

Let CM$[i, j]$ denote the counter at position (i, j) in the array. To add an item z of value v_z in the Count-Min sketch, we increase the counters located at CM $[j, h_j(z)]$ by v_z, for $j \in [1 \ldots d]$ (see Figure 2). Similarly, to remove an item from the sketch, we hash the item and decrease the corresponding counters for the item. A point query for an item q is answered by hashing the item in each of the d rows and getting the minimum value of the corresponding cells, i.e.,

$$\min_{j=1:d} CM[j, h_j(q)]$$

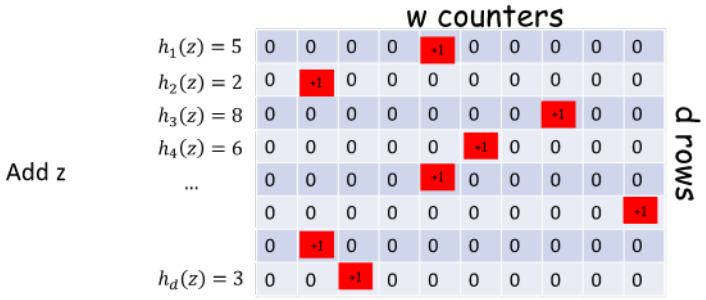


Figure 2: Adding item z to a Count-min sketch.

Note that hash collisions may cause estimation inaccuracies - only overestimations. By setting $d = \lceil \ln(1/\delta) \rceil$ and $w = \lceil e/\varepsilon \rceil$, where e is the base of the natural logarithm, the structure enables point queries to be answered with an error of less than $\varepsilon \|a\|_1$, with a probability of at least $1 - \delta$, where $\|a\|_1$ denotes the number of items seen in the stream.

The Count-min sketch can be used to estimate heavy hitters and to identify the most frequent items. More formally, we define the set of heavy hitters as those items whose frequency exceeds a fraction φ ($0 < \varphi < 1$) of the stream length. To keep track of heavy hitters, the frequent item is stored in a data structure separate to the sketch, such as a heap or list sorted by the estimated frequency [23]. When the frequency of an item increases, at the same time the sketch can be queried to obtain the current estimated frequency. If the item exceeds the current threshold for being a heavy hitter, it can be added to the data structure. At any time, the currently set of (approximate) heavy hitters can be found by probing this data structure.

Another functionality of Count-min sketches is for estimating the size of the join. Precisely, consider two tables *A* and *B*, which need to be joined on attributes *x* and *y* respectively. We can estimate the size of the join as follows: (a) create a count-min sketch for *A* summarizing the distribution of *x*, (b) create a count-min sketch for *B* summarizing the distribution of *y*, (c) computing the inner product of each row of the first sketch with the corresponding row of the second sketch, and (d) returning the minimum value as an estimate. Due to the compactness of the sketches, this approach is both space- and computationally-efficient.

Count-min sketches also support range queries. To answer range queries with an acceptable error, the sketch is combined with dyadic ranges. An interval is a dyadic range if its length is a power-of-two length, and its start index is 1 (mod *l*). Any arbitrary range can be canonically partitioned into dyadic ranges with a simple procedure: greedily find the longest possible dyadic range from the start of the range and repeat on what remains. So for example, the range [18...38] can be broken into the dyadic ranges [18...18], [19...20], [21...24],[25...32],[33...36],[37...38]. Therefore, a range query (length *m*) can be broken up into O(log(*m*)) pieces, and each of these can be posed to an appropriate sketch over the hierarchy of dyadic ranges.

To combine Count-min sketches with dyadic ranges, we maintain a hierarchical structure comprising log(*n*) sketches (see Figure 3). The first level covers the total range of input elements[1]. For any subsequent level *l*, we divide each of the intervals of level *l-1* to 2 intervals, and use the sketch to summarize the distribution of the intervals. As an example, consider that the input data is non-negative integers. Therefore, the sketch at level 1 keeps the count of all integers from the range [0, 2^{32}-1]. At level 2, the sketch keeps the count of all integers from two separate intervals, i.e., [0, 2^{31}-1] and [2^{31}, 2^{32}-1], and so on. Continuing this process, the sketch at level 32 summarizes the frequency of each individual integer. Then, at each level, the upper Intervals are divided by half, and the frequency of element in each interval is stored in a Count-min sketch. For each level, we maintain a separate CMS. When a new element arrives, we update the sketches at each level by increasing the frequency of the interval that covers the element. Range queries are then executed as follows: (a) we break the range into dyadic intervals as discussed above, and (b) we get an estimate for each dyadic interval and sum our estimates. Inserting and removing an item requires an update of log(n).

Count-min sketches combined with dyadic ranges are also used to estimate quantiles, which are important in data analytics. A quantile divides a frequency distribution to equal groups, each containing the same fraction of the total population. For example, the median is the 2-quantile, dividing the frequency distribution to two groups of equal size. The *k*-quantile problem can be formulated as a sequence of range queries for finding the range that covers the $1/k$ fraction of the distribution. By executing a binary search on dyadic ranges, we can identify the corresponding quantile points.

[1] In practice, the first i levels (where i ~ 10, depending on the available RAM) are not stored as sketches but as small, one-dimensional arrays of size 2^{i-1}. Furthermore, additional optimizations are possible to reduce the approximation error.

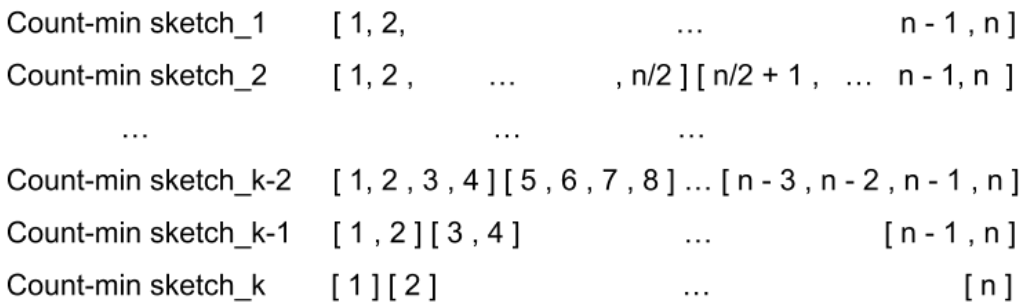


Figure 3: Hierarchical count-min sketch.

Notice that the above techniques are all limited to one-dimensional data. However, SDL also contains high-dimensional data, e.g., spatial data which are two-dimensional, or spatiotemporal data which cover three dimensions. A natural way to execute range queries on two or more dimensions with count-min sketches is to decompose the d dimensions to *d*-dimensional dyadic ranges, i.e., to dyadic *d*-orthotopes. However, this suffers from the curse of dimensionality, quickly becoming a non-viable solution. A common approach is to form groups of dimensions and sketch the pairwise distributions of these groups [25].

Even though Count-min sketches were originally proposed for data streams, their compact nature, probabilistic guarantees, and performance, makes them very appealing also for summarizing stored data. To the best of our knowledge, QAL is the first AQP engine that integrates Count-min sketches for query execution over stored data, and handles them as first-class citizens in terms of query planning.

2.3.2. Bloom filter

Bloom Filters [26] are popular synopses for testing set membership in an efficient way. A Bloom filter consists of an array of m bits and a set of k independent hash functions $F = \{f_1, f_2 \ldots f_k\}$, which hash elements of a universe U to an integer in the range of [1, m]. The m bits are initially set to 0 in an empty Bloom filter. An element e is inserted into the Bloom filter by setting all positions $f_i(e)$ of the bit array to 1 (see Figure 4). For any given element $e \in U$, we conclude that e is not present in the original set if at least one of the positions computed by the hash functions of the Bloom filter points to a bit still set to 0. However, Bloom filters allow false positives; due to hash collisions, it is possible that all bits representing a certain element have been set to 1 by the insertion of other elements. Given that r elements are hashed in the filter, the probability that a membership test yields a false positive is $p \approx \left(1 - e^{-kr/m}\right)^k$. The false positive probability is minimized by setting the number of hash functions to $k \approx \frac{m}{r} \ln(2)$.

Removing items from a Bloom filter is not possible: simply rehashing the item and setting the corresponding bits to 0 can accidentally introduce false negatives, since other items that are already added to the Bloom filter might have hashed in the same bits. An approach for handling deletions is by replacing bits with small counts which can be increased and reduced accordingly[26].

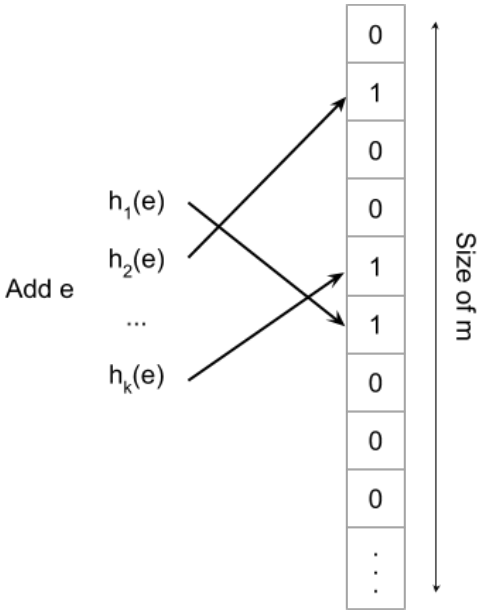


Figure 4: Adding item e to Bloom Filter.

2.3.3. Other synopses

The problem of estimating the cardinality of distinct items that appeared in a sequence have been heavily studied in the last two decades. The Flajolet-Martin sketch (FM) is probably the earliest and best-known method to approximate the distinct count in a small space [25]. Given a stream of *N* integers, the FM algorithm answers how many distinct integers have appeared in the stream.

In addition to samples and sketches, there are a few other data structures that focus on data summarization. Histogram summarizes the distribution of an attribute in a dataset by dividing the attribute range into multiple non-overlapping buckets, and counting the occurrences in each bucket [26]. Wavelet is another type of synopses that is conceptually close to histograms. Wavelet transform the data with an aim to compress the most expressive features in a wavelet domain, whereas histograms simply produces buckets on the original –non-transformed– data. Several variants of wavelets have been studied in recent years, e.g., see [25].

2.4. Massively parallel processing platforms

Due to the mere size of data expected in SmartDataLake, we require scaling out of both the storage and processing. Therefore, we emigrate from a centralized system to a distributed environment of individual machines connected over a high-bandwidth, low-latency LAN. This naturally brings the need of a programming model that supports distributed processing over this distributed data. To attain this goal, the QAL is implemented over one of the state-of-the-art Massively Parallel Processing (MPP) platforms, Apache Spark, by extending with approximation techniques both SparkSQL (Spark's query execution engine) and Catalyst query optimizer. In the remainder of this section, we briefly describe Apache Spark and its programming model, and then focus on SparkSQL and Catalyst.

2.4.1. Apache Spark

Apache Spark™ is an open-source unified analytics engine for large-scale data processing [27]. It is a fast and general-purpose cluster computing system that offers high-level APIs in Java, Scala, Python and R for distributed programming. In practice, it provides an interface for programming entire clusters with implicit data parallelism and fault tolerance. Spark facilitates the rapid implementation of analytic applications by offering pre-built machine learning algorithms, graph analysis techniques, interactive SQL query processing, and real-time streaming analytics. Unlike Hadoop MapReduce that relies heavily on disk storage, Spark mainly works in memory, making it much faster at processing data.

The core data structure for storing data in Spark is called Resilient Distributed Dataset (RDD) [28]. RDDs are read-only multisets of data items distributed over a cluster of machines, that is maintained in a fault-tolerant way with user-configurable replication level. Spark and its RDDs were designed to allow the programmer to develop applications as a sequence of MapReduce tasks over a cluster of machines. Simply, MapReduce programs read input data as RDD, map a function on each record of RDD, and then reduce the results e.g., by aggregating. Spark's RDD is the primary data abstraction: an immutable collection of data sets or partitioned records distributed across cluster. Additionally, RDDs can contain any type of Java or Scala objects. As later versions of RDD API, the Dataframe API was released as an abstraction on top of the RDD, followed by the Dataset API.

Spark Core, the foundation of the project, provides distributed task dispatching, scheduling, and basic I/O functionalities, exposed through an application programming interface. This interface provides a functional model of programming: a "driver" program invokes parallel operations such as map, filter or reduces on an RDD by passing a function to Spark, which then schedules the function's execution in parallel on "executors" [29]. Executors are worker nodes' processes in charge of running individual tasks. They are launched at the beginning of a Spark application and typically run for the entire lifetime of an application. Once they have run the task, they send the results to the driver. All these operations, and additional ones such as joins, take RDDs as input and produce new RDDs. RDDs are immutable and their operations are lazy; fault-tolerance is achieved by keeping track of the lineage of each RDD so that it can be reconstructed in the case of data loss.

Apache Spark strength could be summarized as follows:

1. **Performance**: By exploiting in-memory computing and other optimizations. Additionally, Spark is also fast when data is stored on disk.
2. **Ease of use**: Spark has easy-to-use APIs for operating on large datasets. This includes a collection of operators for transforming data and familiar data frame APIs for manipulating semi-structured data.
3. **A unified Engine**: Spark continuously releases high-level libraries, including support for SQL queries, streaming data, machine learning and graph processing.

2.4.2. SparkSQL

SparkSQL brings native support for SQL to Apache Spark and streamlines the process of querying data. In practice, SparkSQL conveniently blurs the lines between RDDs and relational tables. Concretely, SparkSQL will allow developers to import relational data, run SQL queries over imported data and existing RDDs, and easily save the output RDD over HDFS. In terms of implementation, SparkSQL defines SQL operations as native Spark operations like RDD filter, join, sort, union, etc. Additionally, SparkSQL includes a cost-based optimizer, columnar storage, and

code generation to make queries faster. SparkSQL has the following four libraries which are used to interact with relational and procedural processing:

1. **Data source API**: This is a universal API for loading and storing structured data. It processes the data on the size of Kilobytes to Petabytes on a single-node cluster to multi-node clusters.
2. **DataFrame API**: A DataFrame is a distributed collection of data organized into named columns. It is equivalent to a relational table in SQL used for storing data into tables.
3. **SQL interpreter and optimizer**: SQL interpreter and optimizer is based on functional programming constructed in Scala. It is the newest and most technically evolved component of Spark SQL. It provides a general framework for transforming trees, which is used to perform analysis, evaluation, optimization, and planning. This supports cost-based optimization and rule-based optimization, making queries run much faster than executing unrefined sequence operations on RDDs. At the core of SparkSQL is the Catalyst optimizer, which leverages advanced programming language features (e.g. Scala's pattern matching) in a novel way to build an extensible query optimizer.
4. **SQL service**: SQL service is the entry point for working along with structured data in Spark. It allows the creation of DataFrame objects as well as the execution of SQL queries.

2.4.3. Catalyst optimizer

Catalyst is SparkSQL query planner, and it is based on functional programming Scala constructs. It is designed to let the developers easily extend the optimizer by adding their own optimization rules (e.g. adding data source-specific rules, or support for new data types). The terminology used by the optimizer is the following:

Tree: the main data type in the catalyst. A tree contains node objects, and a node can have one or more children.

Rule: define as a transform function from one tree to another tree, and it is a common approach to use a pattern-matching function and replace sub-tree with a specific structure. We can recursively apply pattern matching on all the nodes of a tree. Catalyst applies rules until a fixed point is achieved, after which tree stops changing.

Logical plan: series of algebraic or language constructs, as for example SELECT, GROUP BY or UNION keywords in SQL. It is represented as a tree where nodes are the constructs, but without defining how to conduct computation.

Physical plan: like logical, physical plan is represented as a tree but the difference is that the physical plan concerns low-level operations.

As shown in Figure 5, Catalyst optimization starts from relations to be computed which may contain unresolved attribute references or relations. An attribute is unresolved when we do not know its type, or have not matched it yet to an input table. SparkSQL makes use of Catalyst rules and the Catalog (standard API for accessing metadata in SparkSQL) to resolve these attributes. Then, Catalyst propagates and pushes the types through expressions to generate an unoptimized logical plan.

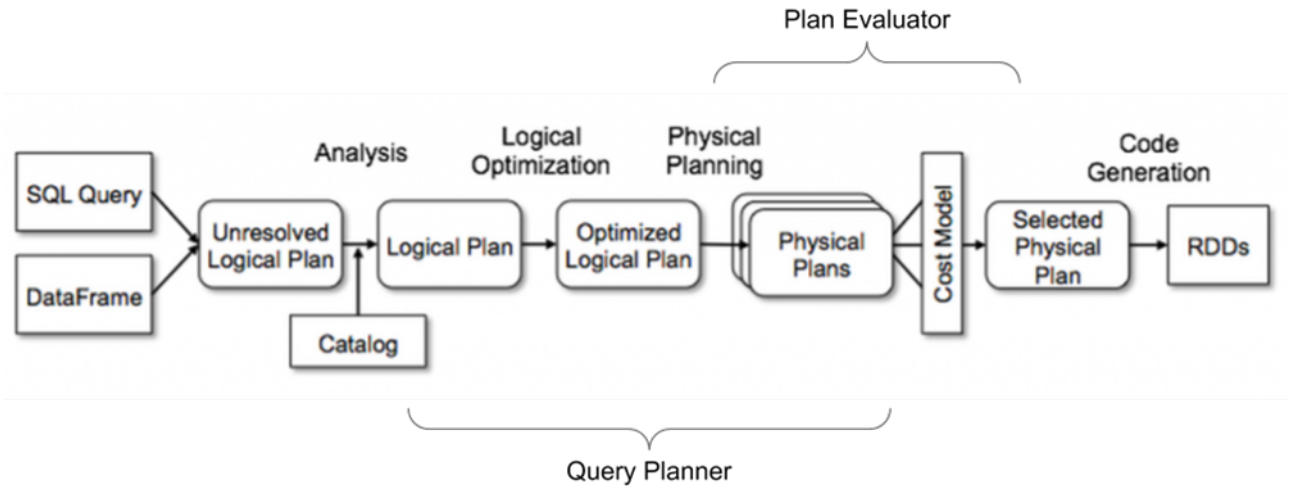


Figure 5: Catalyst's general query transformation framework in four phases.

After Catalyst generates the unoptimized logical plan, it applies a standard rule-based optimization on the plan. Examples for the extensible set of optimization rules include predicate pushdown, Boolean expression simplification, and projection pruning. For the next phase, SparkSQL takes the optimized logical plan and generates one or more physical plans, using physical operators that are implemented in the Spark execution engine. Furthermore, a final round of rule-based physical optimizations is executed, e.g., pipelining projections or filters into one Spark map operation. Finally, Catalyst estimates the cost of each physical plan and selects the plan with the smallest estimated cost.

3. SDL Query Approximation Layer

We now introduce Query Approximation (QAL), an adaptive approximate query processing engine inside SDL-Virt. QAL is constructed over Spark by extending SparkSQL and Catalyst, thereby exploiting the robustness, performance, and scalability of a state-of-the-art platform. The innovative nature of QAL is that it provides automated creation and adaptation of synopses (samples and sketches) for summarizing big data, as well as transparent integration of these synopses during query execution. The data analyst only needs to specify the query requirements as an SQL query, along with the desired accuracy. Then, QAL automatically creates and executes query plans for approximating the query results over big data.

Self-tuning and adaptive nature: QAL provides an automated synopses creation process that adapts to the query workload, along with a mechanism for selecting and storing frequent synopses in distributed RAM (as RDDs) for future usage. In particular, for each query, QAL generates multiple approximate plans leveraging a variety of synopses. Then, it chooses to execute the plan and to materialize the required synopses that are expected to maximize the throughput of future queries, thereby decreasing the overall I/O cost. Preliminary results of QAL were published in [30].

Figure 6 demonstrates the overall architecture of QAL. The input of QAL comprises of a stream of SQL queries annotated with the desired accuracy guarantees, and the input data for analysis – possibly distributed over a Hadoop distributed file system for scalability. The desired output is an approximate answer for each query in the stream. The core components of QAL are the query

planner, the plan evaluator, the plan executor, the synopsis catalogue, and finally the change detector. Precisely, as QAL receives an approximate query, the query planner analyses the query to extract a set of logical plans, chooses the most efficient ones, and out of them generates a set of physical plans. The difference between these logical and physical plans lies on the use of synopses – the logical plans are synopsis-unaware plans, analogous to the logical plans in relational databases, whereas the physical plans have some access paths or plan sub-trees replaced by synopses. The involved synopses in these physical plans may or may not be already materialized. We explain how the query planner works in Section 3.1.

After the candidate plans are generated, the choice of which plan should be executed, and, subsequently, which synopses should be materialized, is taken by the cost-based evaluation component. Contrary to the traditional approaches in relational databases which choose the plan that reduces execution time for the query at hand, the cost-based evaluator in QAL chooses the plan that will also optimize the throughput over the next few queries (see Section 3.2). The idea behind this choice is that the decision as to which plan is executed influences the query at hand, but also the next queries that may be able to use the synopses materialized for the current query. The chosen best plan is finally forwarded to the plan executor.

All synopses generated in this process are stored in the synopsis catalogue. When the input files are updated (updating happens, in principle, only with appends) the synopses stored in the synopsis catalogue are also updated to reflect the new data.

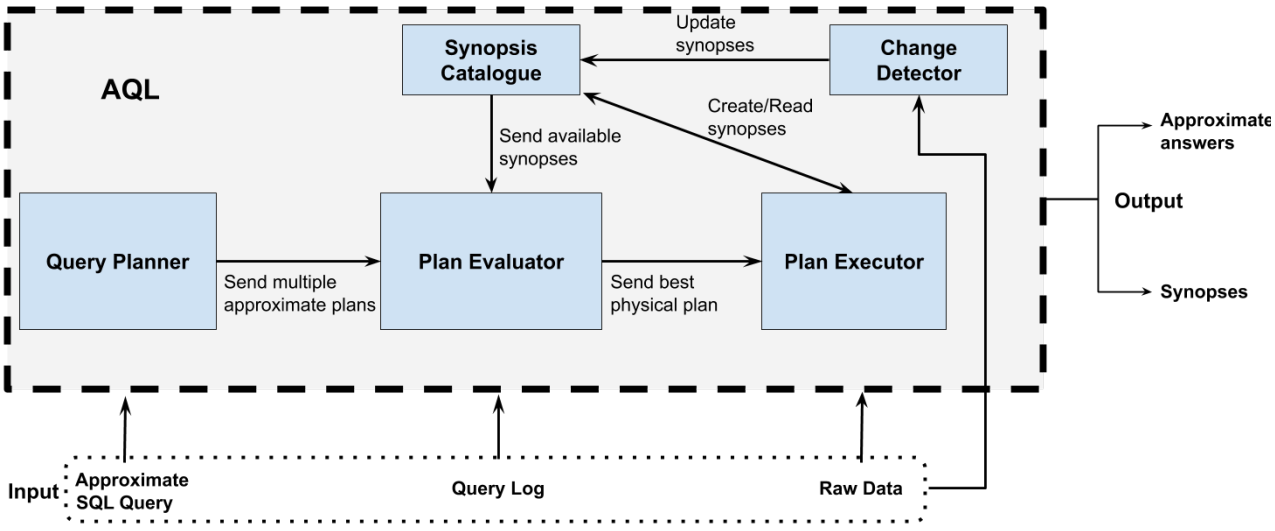


Figure 6: QAL input/output, internal components and their interactions.

In term of implementation, QAL is integrated in SparkSQL by extending both the physical planner and SparkSQL cost model to support our requirements. Recall that SparkSQL and Catalyst answer the SQL queries by generating and executing a sequence of optimized physical operations (e.g. aggregations, scans, projects, joins) as MapReduce actions on raw data. To enable Catalyst to support approximate query processing, we define synopses as approximate physical operators and extend the rules for generating the physical plans, such that they integrate and optimize approximate physical plans. Additionally, we extend the cost model to adapt itself to the workload and to reuse stored synopses for answering coming queries.

In the rest of this section, we continue the discussion for the internals of QAL. This includes integrating synopses as physical operators inside the SparkSQL physical planner and introducing an adaptive cost-based planner. As discussed before, approximate physical planning requires extra

effort because we have to generate more than one executable plans, and they should all satisfy user's desired accuracy. Additionally, we detail the QAL adaptivity to the future workload which includes predicting future queries, identifying reusable physical plans, and storing proper synopses.

3.1. Approximate query planning

In this part, we focus on integrating QAL into SparkSQL by extending both the query planner and cost model to exploit synopses. Due to their small size compared to the original data, synopses improve both computational complexity and I/O cost for future queries. QAL promises negligible overhead for creating and maintaining synopses, by parallelizing the synopses construction phase with the exact query execution procedure. In practice, all synopses are created on-the-fly, as by-products of query answering, thereby inducing no additional I/O.

QAL accepts and answers all SQL queries supported by Spark SQL. Similar to prior work, e.g., [3][12], it improves performance for queries containing aggregates (e.g., COUNT, AVG, SUM). The query format for approximate queries follows the syntax: "with ERROR x% AT CONFIDENCE y%", which corresponds to aggregate results with a relative error of at most x% at a y% confidence level. QAL adapts the query plan accordingly such that the accuracy guarantees are satisfied, and all groups are included in the results, e.g., when a group-by is requested.

In the following sub-sections, we present a high-level overview of the core concepts of QAL.

3.1.1. Approximate physical operators

Synopses in QAL are promoted to first-class citizens, and are used for summarizing both raw data (base relations) and query sub-plans (e.g., the results of an aggregator over a join). To achieve this goal, synopses are included as approximate operators in the execution plans, cost as all other physical operators, and are transformed to fully pipelined and distributable code during the physical plan generation. As an example, an approximate answer for point query can be estimated via Count-min sketch, dyadic ranges, uniform sample, or distinct sample; thus, the physical planner proposes various execution plans for the approximate query. Each generated plan comprises various synopses in different levels of the execution tree so that the planner can produce more efficient plans, and to promote the reusability of synopses by matching a variety of synopses across different queries.

Upon receiving the input query, QAL generates the optimal logical plan for the query by exploiting Catalyst's standard optimization rules. Then, it generates multiple physical plans for the optimized logical plan by replacing the aggregator operators with (possibly approximate) aggregators. Focusing on aggregations, the planner first identifies all query sub-plans rooted on (partial/eager) aggregators. For each, it injects a synopsis operator satisfying the user's accuracy requirements, 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 sub-plan using a (*possibly not yet existent*) synopsis.

The above process entails several challenges. First, the process of generating candidate plans is different compared to planners found in traditional relational databases. Unlike traditional query planning, the planner now also needs to consider 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, in order to reuse the synopses in future queries, the planner must store metadata representing characteristics of each synopsis so that it can map constructed synopses to the next plans. In the following, we describe how the planner handles these challenges.

3.1.2. Generating candidate physical plans

The planner generates the set of approximate physical plans by injecting synopses as physical operators inside the execution tree. Subsequently, QAL 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. The planner must preserve the accuracy and functionality of synopses while it is injecting and pushing down approximate operators, so we define specific rules for each type of operator to correctly and effectively integrate it inside the plans.

For samples, the planner relies on the push-down rules for samples introduced in [12]. Whenever QAL pushes a sample operator under a filter on *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. On the other hand, if the distribution of the values of *p* is skewed (some groups appear infrequently), QAL needs to stratify the underlying output on *p*. Thus, QAL adds the attributes appearing in *p* which follow a skewed distribution into the stratification set.

Considering pushing samples under the joins, given a join *R* with *S* with join predicates *j*, the planner pushes the sample 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 sample to include the attributes from *j* that are contained in. Finally, if the join predicate is not a grouping attribute, QAL introduces a partial aggregation after the join. The above push-down process guarantees that (i) the generated physical query plan will gather enough samples from each of the groups to satisfy the 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 the next section. In terms of implementation, the push-down strategies are implemented as rules in the Catalyst optimizer and are executed at every query.

Example: Assume that we have a table *companies* with attribute *province*, and QAL receives input query "SELECT COUNT (*) FROM companies WHERE province= 'Taranto' WITH CONFIDENCE 95 AND ERROR 5". The first step is generating an optimized logical plan and passing the logical plan to the physical planner. As an instance, the purple tree in Figure 7 depicts the optimized logical plan of the input query. The planner can approximately answer the input query utilizing different execution strategies such as exact execution, sampling data, or constructing sketches. As shown in Figure 7, the red tree represents the exact execution of input query that reads the table as an RDD, filters the rows, and counts the number of remaining rows.

Since the query allows a margin of error, along with the exact plan, the planner produces approximate plans that can leverage synopses. In our example, the input query is a simple point query that can be answered by samples (uniform or distinct), or sketches (count-min sketches, and count-min sketches with dyadic ranges – see Figure 7). The planner injects potential approximate operators and changes the exact aggregation operator (AggregateExec) to approximate one which scale the answer of samples. Both samples are pushed down, just above the RDD scans to maximize the re-usability of samples for future queries and to decrease computation cost of higher levels. For plans with sketches, we eliminate filters and construct count-min sketches from

attribute *province*, and then estimate the aggregation operators by querying the sketches instead of the tables.

3.1.3. Accuracy guarantees

While generating and exploring the potential plans, the planner needs to ensure that the user's accuracy requirements are satisfied. For this, QAL relies on previous analytical results [12][31], which we outline below. When using sampling, QAL uses the Horvitz-Thompson (HT) estimator [32] to calculate unbiased estimators of the true aggregate values. Confidence intervals are computed using the Central Limit Theorem (CLT) [33]. Due to the distance of the samplers to the aggregation operators, we use the notion of dominance between query expressions as defined in Quickr [12], 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 [32]. However, for stratified and uniform sampling, QAL calculates the error in a single pass by utilizing the observation of [12] that to compute the standard error for each group, we only need to take into account the tuples with the same stratification key (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 (grouping) attribute, as value the running estimated error for that group and the corresponding list of sampled tuples. For every sampled tuple, QAL updates the error of that tuple's group by using the HT estimator error formula, leading to a single-pass, linear complexity algorithm.

Count-min sketches offer error guarantees relative to the $L1$ norm of the summarized relation [31]. Particularly, let $f(x)$ denote the real frequency of key x, and $f(x)$ the frequency estimated from the sketch. Then, the sketch is configured such that $f(x) - f(x) < \varepsilon N$ where N represents the $L1$ norm of the frequencies for all keys.

3.2. Adaptivity to query workload

The remaining question is selecting and executing the most beneficial plan to maximize gain, which is the throughput of AQP for future queries. To alleviate this problem, we proposed an adaptive cost-based model that evaluates generated approximate physical plans based on their synopses re-usability for a window of future queries. That means QAL automatically decides which synopses to create, store, and use for answering each query while it is maximizing the gain. In other words, the cost-based model combines two main goals: (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.

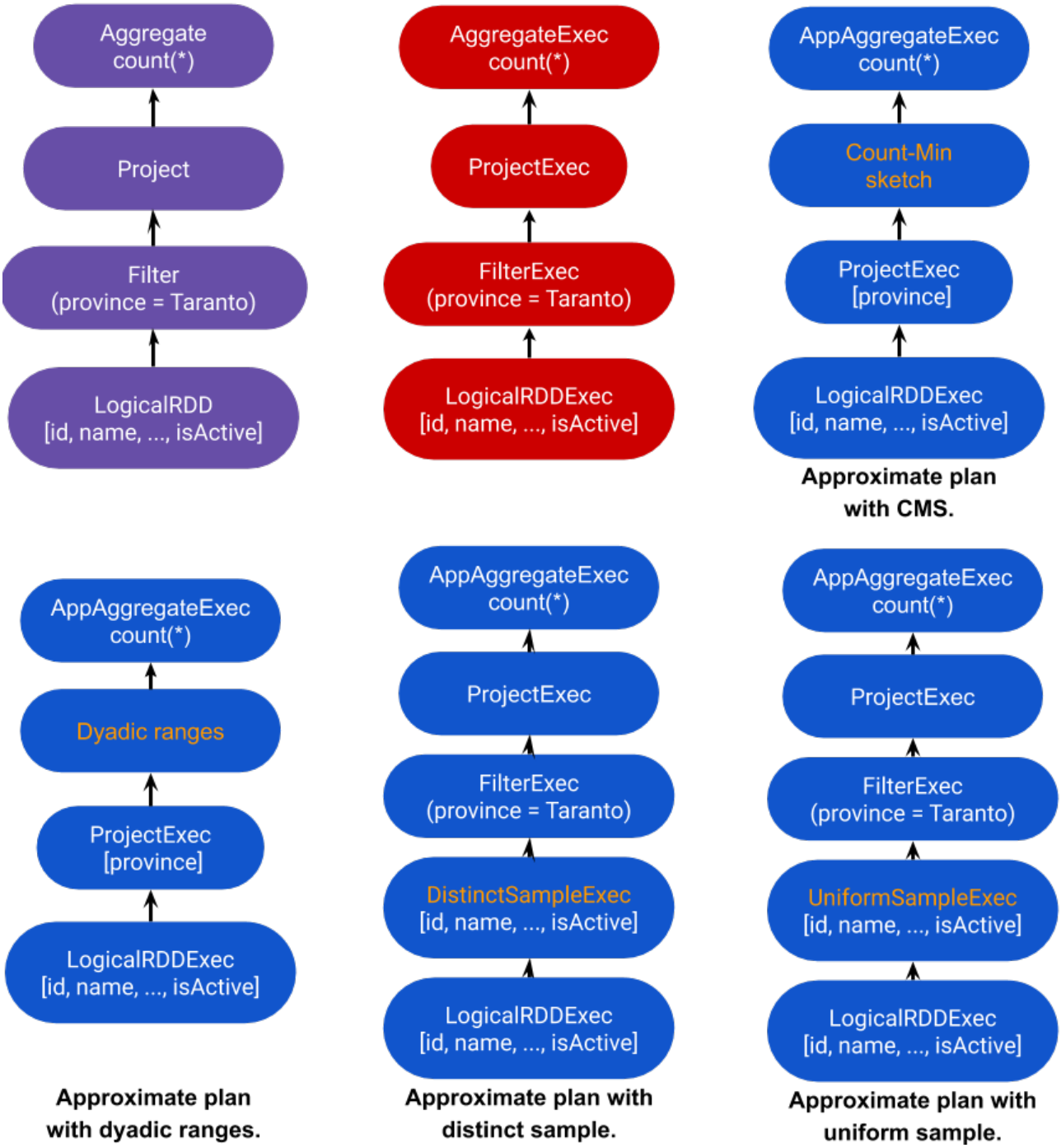


Figure 7: Logical plan (purple) and its candidate physical plans (exact plan is depicted as red, and approximate plans are blue, with the approximate operators noted with orange fonts).

3.2.1. Cost-based planner

Upon receiving the query q, the planner generates a set of approximate physical plans, denoted by $P(q) = \{p_1, p_2, ...\}$. These plans utilize synopses as physical operators 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 its performance gain, consequently, when ranking the plans, the cost-based model focuses on maximizing long-term throughput, i.e., over the future workload, as opposed to minimizing only the cost of the query at hand. Clearly, we have limited available space to store synopses for future queries, denoted *maxSpace*, so that the objective function is selecting the set of plans and respective synopses S that will maximize the total gain (future throughput). Formally, the optimization problem is as follows:

maximize gain(Q^+_i, S)

subject to $\sum_{s \in S} |s| \leq maxSpace$

It is useful to define the synopsis gain metric, i.e., how much does each set of synopses S contribute to the performance of each query. Formally, gain(q, S) = cost(q, φ) – cost(q, S), where the cost(q, S) denotes the minimum cost of any plan in $P(q)$ for answering q, given only the synopses in S. In the case of $S = \varphi$, this will be the cost of the most efficient plan that does not utilize synopses and returns the exact answers. For a given Q^+_i we maximize the query throughput by minimizing the total cost of executing these queries, i.e., minimize $q \in Q^+_i$ cost (q, S), or equivalently, by maximizing their corresponding gain: maximize $q \in Q^+_i$ gain (q, S). For convenience, we slightly overload the notation by using gain (Q^+_i, S) to denote the gain overall queries using synopses in S.

Figure 8 represents a simple cost-based planning of QAL for input query q_{20} based on expected future queries q_{21}, q_{22}, q_{23}. For this example, we skip the mechanism of predicting future queries and postpone it to the next section. As it was discussed, the planner generates multiple approximate physical plans for the q_{20} named P_1, P_2, and P_3 including their corresponding synopses and execution time. In order to evaluate each candidate plan, we must look to potential future plans; thus, QAL feeds expected future queries to the planner and generates possible physical plans named FP_1 to FP_{10} with their related synopses. Now, the cost-based planner can estimate re-usability of P_1, P_2, and P_3 synopses and the future throughput for each plan. Based on execution time, P_1 provides fastest answer for the current query, but It has the least future throughput. On the other hand, by executing P_2, we decrease the current response time (around 0.4 second), but we have created set of synopses that covers most of future queries. Consequently, future queries will be executed faster due to reusing synopses and less I/O.

Even though the problem is well-defined, we face two major challenges: (a) holistic optimization can be CPU-intensive, and (b) the future queries and their potential synopses are not yet known. We explain how these issues are addressed.

Formally, the first problem can be reduced to a variant of 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 query, and some queries are answered by more than one synopsis. 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 sub-modular function, i.e., the gain provided by every single synopsis is only reduced as the set of synopses in S increases. For this special case, there exist several efficient approximation algorithms. We employ the efficient greedy algorithm of [34], 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 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 S. After S is created, the QAL 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 S are deleted.

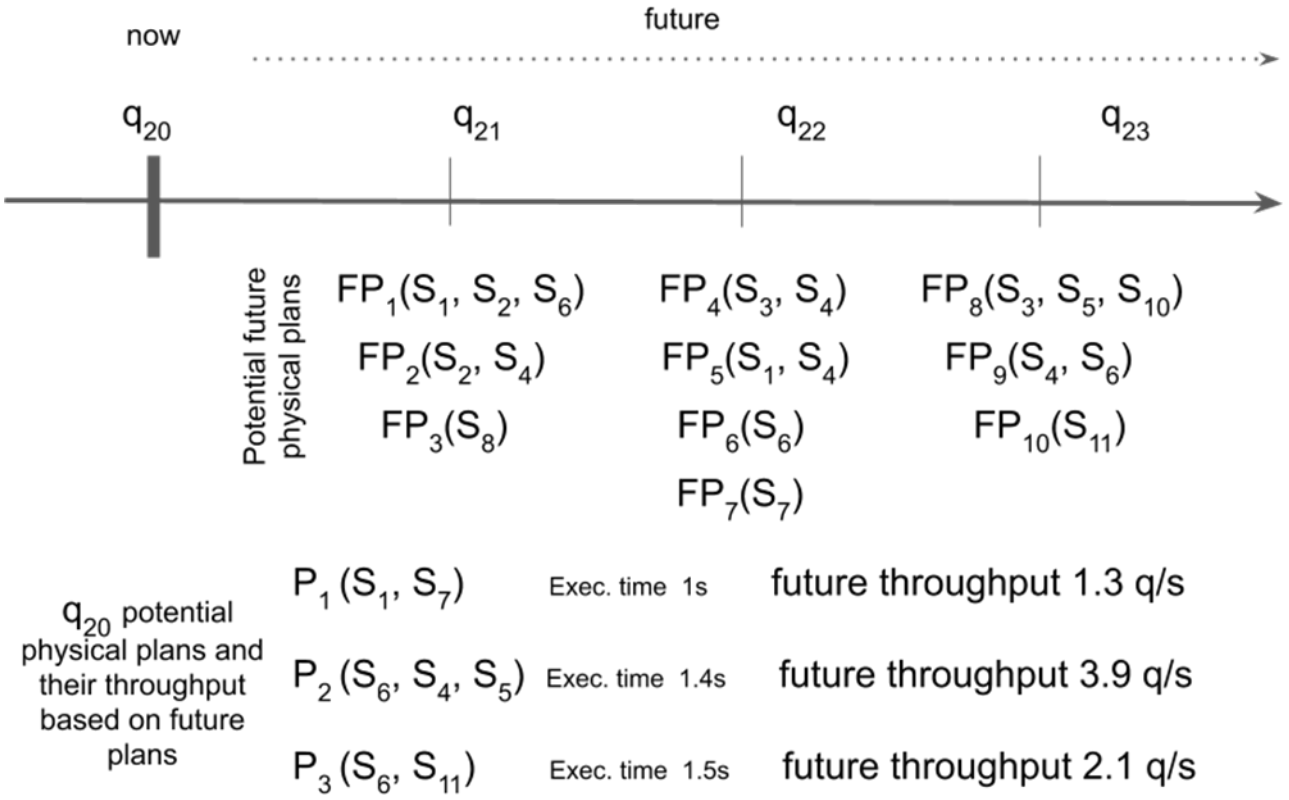


Figure 8: The cost-based planner evaluates q_{20} candidate physical plans based on future throughput.

The second challenge concerns the definition of the future query, Q^+_i. In practice, we cannot expect to know the queries contained in Q^+_i during the cost planning. We, therefore, have proposed two solution: (a) rely on the standard assumption that recent queries are a good representation of the following queries [26] (b) leverage Machine Learning (ML) predictive models to forecast future physical operators (indirectly, synopses). We detail both solutions in Section 3.3.

3.2.2. Synopsis warehouse

QAL uses a set of automatically constructed and tuned synopses to summarize both the raw data (the base relations) and intermediary results of sub-plans (e.g., join results). Currently, it exploits two types of synopses (samples and sketches) each being appropriate for answering different query families. All synopses are constructed as byproducts of query answering and are saved in the synopsis warehouse, in HDFS (Hadoop Distributed File System). Along with synopses, QAL 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.

3.2.3. Synopsis buffer

The plan chosen for execution may require the 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 a 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.

3.2.4. Metadata store

Effectiveness of the planner depends on the existence of metadata that characterizes the past workload and the synopses that could speed-up future 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 sub-plan 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, and (b) decide which synopses will maximize throughput, i.e., because they will improve many different sub-plans.

Figure 9 presents an overview of QAL 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. Just before the arrival of a 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, not relevant to the three queries. During Q_1, the planner proposes two candidate plans (cf., Figure 9a). The first one contains synopsis S_4, which summarizes R on S, and the second contains synopsis S_5 of R. Notice that neither of the two synopses exists. The two plans are costed, and the metadata store is updated with the corresponding properties of S_4 and S_5. The cost-based planner 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., Figure 9b), 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 stored with the corresponding properties of the two candidate synopses. Now, as the synopsis buffer is full, QAL 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, QAL 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., Figure 9c), the first replacing the scanning of relation T with S_1 which is already saved in the warehouse, and the second utilizing a non-existent synopsis S_8. The plans are sent to the cost-based planner, where the first one is chosen and sent for execution.

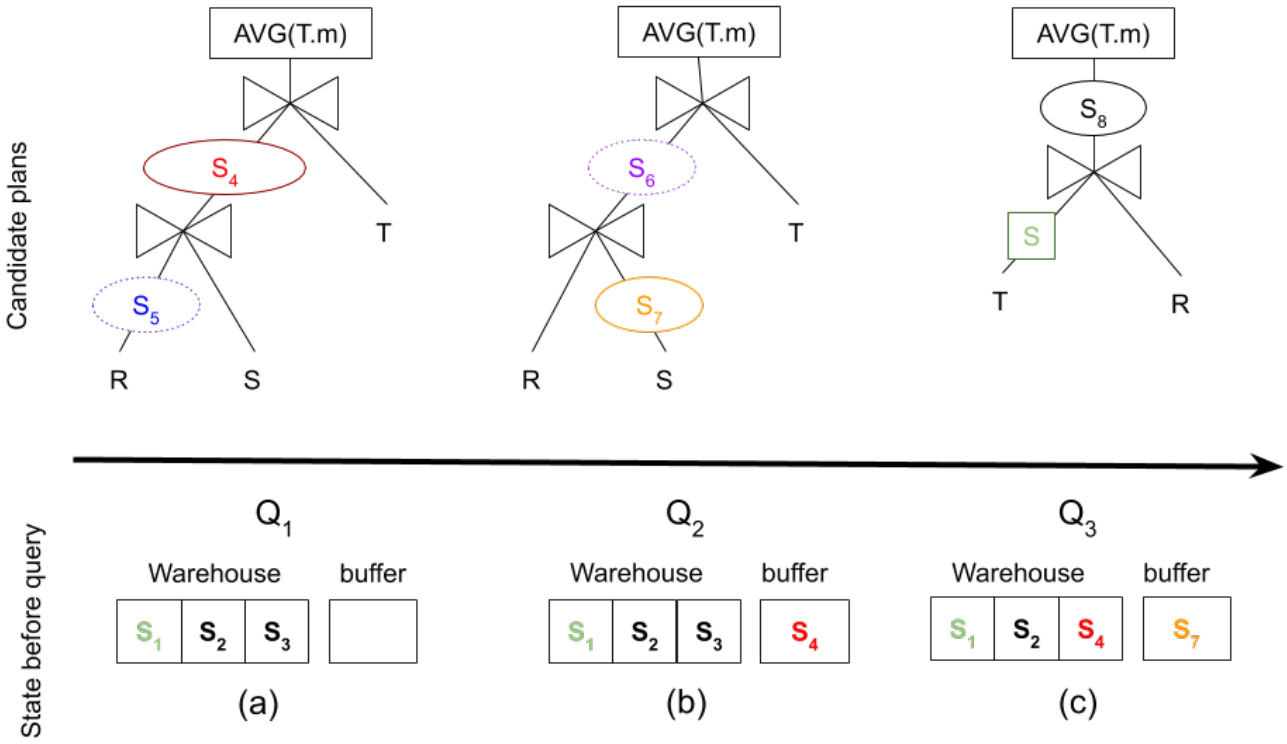


Figure 9: Example of warehouse and buffer management. The dotted ellipses are prospective synopses, the normal-line ellipses are executed synopses, and square is reused synopses.

3.2.5. Matching approximate query operators to synopses

Costing of the physical plans requires efficiently matching the synopses contained in the query's 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 physical sub-plan – the one of which the results it summarizes. Therefore, the sub-plans for the query at hand are compared to the sub-plans of the synopses contained in the metadata store. We say that a query sub-plan matches a synopsis when: (i) the accuracy guarantees of the synopsis satisfy the query requirements, and (ii) the synopsis sub-plan subsumes the query sub-plan. For the latter, QAL ensures that the query sub-plan is covered by the synopsis regarding join and filtering predicates as well as the projected columns.

Particularly, QAL compares the input relations, the join and filtering predicates as well as the output attribute set. The synopsis sub-plan 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 sub-plan [35]. Some mismatches are addressed by adding filtering and projection operators directly above the query sub-plan, to remove extraneous tuples and attributes.

Considering accuracy, a synopsis is a candidate for a sub-plan if (i) the set of stratification attributes of the stored synopsis is a superset of the stratification attributes of the sub-plan, 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, guarantees group coverage i.e., QAL results will contain all groups, whereas the latter ensures that the aggregates will have constrained error [3]. 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 can put an additional filter in the query plan. However, to use this sample, salaries should be uniformly distributed, irrespective of gender. Sub-plan matching is expensive. Therefore, 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 sub-plan.

3.3. Predicting the future queries

The performance of QAL strongly depends on the choice of synopses to construct. In general, synopses need to be as reusable as possible, i.e., have a very small cost/utility ratio, where cost correspond to the space they occupy and utility to the expected performance boost they can yield for future queries. However, the future workload is unknown. Therefore, it is vital for QAL to be able to predict the future queries, with reasonable accuracy. In this section, we detail two different approaches to predict future queries. First, we look at a simple, yet effective technique based on window-based forecasting. Next, we introduce a more advanced technique that relies on a predictive ML model and enables identifying more complex query patterns. Notice that the latter model is not yet integrated in QAL.

3.3.1. Window-based query prediction

A simple approach for forecasting the future is looking into the past. We could maintain a sliding window w over previous queries to have a good approximation of the next, unseen queries. In practice, we cannot expect to know the queries contained in Q^+_i during answering queries. Therefore, we rely on the standard assumption that recent queries are a good representation of the following queries [29]. For this, we keep track of the last w queries, denoted as $Q^-_i = \{q_{i-w+1}, q_{i-w+2}, ..., q_i\}$, and use their proposed plans to estimate gain (Q^+_i, S).
The best value for w depends on the task at hand which determines the repetitiveness in the query workload. Therefore, the planner dynamically adapts w starting from a small value. For tuning size of the window, the planner identifies (without building) the set of best synopses using a slightly larger and a slightly smaller w value, i.e., $w^+ = (1 + \alpha) * w$ and $w^- = (1-\alpha) * w$, with $\alpha \in (0,1)$. At the next invocation, the planner examines which of w^-, w, or w^+ would minimize execution time for the queries that arrived since the last invocation 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.

3.3.2. Predicting approximate operators

Window-based query prediction is effective, as long as the query workload is quasi-static, i.e., it changes very slowly. However, it fails to identify more complex query patterns. For example, consider the typical scenario, where the data analyst needs to process a huge number of datasets

that satisfy a condition, as follows: (a) first, the dataset is loaded, (b) the data analyst executes a couple of queries for checking the quality of the dataset, (c) the data analyst executes a sequence of data exploration queries. The described query load has the following critical property: the next query depends on the results of the previous few queries. In such scenarios, a sliding window approach fails to successfully predict the future workload.

For such workloads, we can leverage more complex predictive ML models that learn the nature of the workload. There are a plenty of works exploiting ML for query optimization, e.g., for SQL queries clustering [36], predicting the query resource usage [37], query response time [38], database monitoring [39], and estimating output cardinality [40][41]. However, predicting the next SQL queries is difficult, and, to the best of our knowledge, it is not yet addressed. For QAL, we instead address the simpler problem of predicting the frequency of each physical operator in the next – unknown – queries. This offers two advantages: (a) it is easier to address with existing models, (b) the physical operators have a simpler representation (vectorized form) compared to SQL queries, and therefore they enable decent prediction with less training data. Since all synopses are defined as physical operators, prediction of physical operators directly translates to estimation of the cost: utility function of each candidate synopsis.

Example: We use notation q_i to represent the ith SQL query, and o_j to represent physical operator j contained in one or more of the considered physical plans for q_i. Consider the following workload:

Query	Operators	Query	Operators	Query	Operators
q_1 on dataset1	o_1, o_2, o_4, o_5	q_5 on dataset2	o_1, o_2	q_9 on dataset5	o_1, o_2, o_4, o_5
q_2 on dataset1	o_1, o_2	q_6 on dataset2	o_2	q_{10} on dataset6	o_1, o_2, o_4, o_5
q_3 on dataset1	o_2	q_7 on dataset3	o_1, o_2, o_4, o_5	q_{11} on dataset6	o_1, o_2
q_4 on dataset2	o_1, o_2, o_4, o_5	q_8 on dataset4	o_1, o_2, o_4, o_5	q_{12} on dataset6	o_2

Assume that we are using the sliding-window prediction model, and the data analyst executes the first 10 queries on the corresponding datasets. As soon as q_{11} is observed, based on the frequencies of operators observed in the sliding window, QAL will predict that the next – unseen – query (q_{12}) will be similar to q_{10}, but on a new dataset. Therefore, it will not construct or save synopses while answering q_{11}, unless these can also optimize q_{11}. Instead, our goal with using an ML-based predictive model is to enable QAL to identify that the next unseen query, q_{12}, will continue the data exploration process that started with q_{10}, and therefore QAL will optimize for the physical operator o_2 while answering q_{11}.

3.3.3. Approximate operator vectorization

Typically, ML models assume numeric values (vectors) as input and expected output. Therefore, our query plans should be converted to a numerical vector that preserves the characteristics of operators. This procedure is called vectorization, and it plays a vital role in the efficiency of ML models. In our case, the feature that is meaningful for our model is the type of operator, target table attributes, and approximation properties of the operator.

There exist a few recent studies for vectorizing the SQL queries, which are however not directly applicable to our case. Join cardinality can be predicted by vectorizing each query as a collection of a set of tables, a set of join columns, and a set of predicates [40]. A similar vectorization is used for predicting group-by cardinality [40], this time however including the group-by columns instead of the join columns. Another study [42] encodes the query's join graph as an adjacency matrix, and converts the execution plan into a tree vector. [38] instead stores the type of operator, the number of output records, and response time of input query. Marcus et al. [41] encodes the join relations and the position of the join in the execution tree. Lastly, [37] encodes both the SQL query text (e.g., number of nested subqueries, selection predicates, join predicates, sort columns, and aggregation columns), and the query plan by vectorizing cardinality of operator instances and their output records.

We define approximate physical operators as a binary vector that encodes the type of synopses, the offered accuracy, and table attributes. Therefore, the total size of the vector is equal to the number of all attributes plus the different types of synopses for various levels of accuracy. Assume a database comprises two tables each with 5 attributes, and it supports two types of samples (uniform and distinct samples) with varying accuracy 80, 85, 90, 95; consequently, the size of the vector is 22 bits (10 bits for encoding attributes and 8 bits for enumerating any possible synopses with different level of accuracy). To convert an approximate operator to a vector, we set bits of the attributes that are summarized to 1, and based on the type and accuracy of the synopsis, one of 8 bits of operator types is set to 1.

3.3.4. Predictive model

Recurrent neural networks are a type of artificial neural network designed to recognize patterns in sequences of data, such as numerical times series, data emanating from sensors, stock markets, etc. What differentiates RNNs from other neural networks is that they have a temporal dimension, which enables them to take time and sequence into account. In particular, LSTM [43] is a kind of recurrent neural network that is good at learning dependencies between two points in a sequence that are separated very far in time. For example, learning to predict a word in a long sentence where the word strongly depends on some other word that occurred much before in the same sentence. A similar scenario occurs in query workloads, where the current queries may exhibit similar patterns sequences of queries issued at some point in the past. This cannot be captured and predicted by the window-based approach which only learns from recent queries. Additionally, when the user starts to shift his analysis, the future queries will become unrelated to the previous ones, hence the window-based approach will hinder the adaptivity of the planner. To alleviate these problems, the LSTM model maintains memory and goes over the past workload for multiple times to learn the possible patterns (e.g., query shifting) regardless of their occurrence time. This capability is not present in the window-based approach, thus making LSTM a suitable technique to fulfil QAL requirements.

4. Integration with other SDL components and deployment

QAL provides adaptive approximate query processing over distributed large datasets. Each layer of SDL can rapidly import its data into QAL and submit approximate queries on-the-fly. QAL

functionality is exposed through a REST API. The API enables users to import data and to submit approximate queries. In the following, we detail on this API.

Input Query

QAL accepts SQL queries over 'relational-type' tables. The SQL queries are annotated with the desired approximation guarantees, e.g.,

SELECT COUNT (*) FROM Table1 WITH CONFIDENCE 95 AND ERROR 5

Input Data

Data can be directly uploaded to QAL via a REST API, or imported from the Proteus API. QAL accepts various input formats (e.g., CSV, Parquet, ORC, Avro, or JSON).

Output Data

The results are output as tables in CSV format.

Service API

The following is offered by a REST API:

Submit a query: http://x.x.x.x:port/query?q=your_query

Get existing tables name and their attributes in QAL: http://x.x.x.x:port/tablesName

Remove a table from QAL: http://x.x.x.x:port/removeTable?TableName=table1

Add a table to QAL: http://x.x.x.x:port/addTable?TableName=table1&format=csv&path=location

Download a table from QAL: http://x.x.x.x:port/downloadTable?TableName=table1&format=csv

Connect to an external database DB via JDBC (e.g., connect to RAW or Proteus), so that the raw data is read from DB and is not replicated in QAL:

http://x.x.x.x:port/connectDatabase?DBurl=somewhere&user=tue&pass=***

5. Experimental results

In this section we present our preliminary experiments and results that demonstrate the efficiency of QAL and its superiority on dynamic workloads compared to the state-of-the-art AQP engines. The experiments are executed with a preliminary implementation of QAL, which we refer to as *Taster* [32]. Taster utilizes two types of samples (uniform and distinct samples) and Count-min sketches. Furthermore, it uses only the naïve, sliding-window based model for predicting the future operators.

We start by comparing Taster with the state-of-the-art AQP engines over different benchmarks. Then, we detail individual performance gains and approximation error for TPC-H queries. Furthermore, we evaluate the robustness of QAL to workload shifts, i.e., changes in query stratification attributes, the accessed tables, and query predicates. Lastly, we examine the impact of the different size of sliding window and various storage budgets on the adaptivity of the cost-based planner to the future workload.

We compare Taster against three state-of-the-art systems: Quickr [12], BlinkDB [3] [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) [44], scaled 100× (~ 120GB before compression). All datasets were stored in the Parquet-compressed data format.

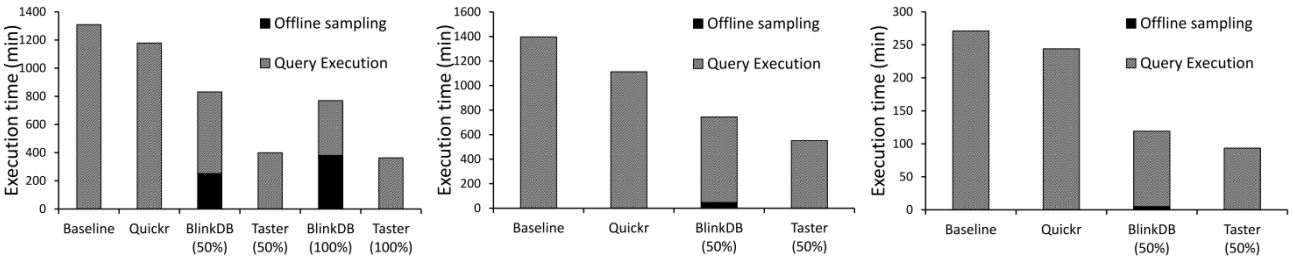
Implementation. To have a fair comparison, we integrated all systems to SparkSQL 2.1.0, and extended the Catalyst built-in optimizer accordingly. For Quickr, we implemented the three sampler operators (Distinct, Uniform, Universe) and added all rules described in [12] to Catalyst. For BlinkDB, we followed the algorithms described in [3] 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. 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.

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.

Figure 10 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 (Fig. 5.1 a), BlinkDB 50% offers 2.25× speed-up compared to the Baseline, and requires 251 seconds for pre- computing the sample, whereas BlinkDB 100% offers 3.36× performance increase but spends 380 seconds on sampling. Quickr requires no preprocessing, but offers a smaller performance boost (1.2×). 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 ~ 3× 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 (Q2 cannot be approximated, Q4, Q21 and Q22 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

Figure 10: End-to-end execution time for different workloads.

The results with TPC-DS and instacart workloads (Figure 10 b-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.

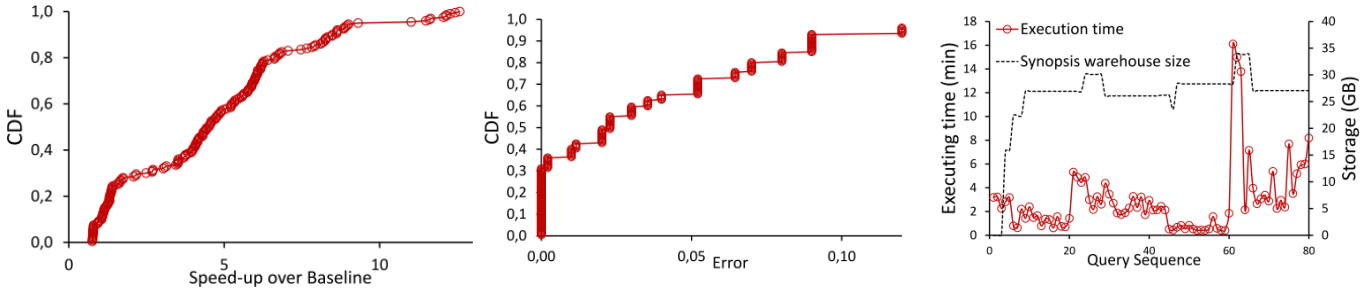
Figure 11a presents a CDF of the speed-up of Taster for TPC-H queries. Taster slows down less than 10% (~ 0.8×) of the queries, mostly due to the planning and tuning overhead. However, more than 50% of the queries are being sped-up more than 6×. The maximum speed-up (13×) is achieved using sketches.

We also verified that the approximations of Taster are within the desired accuracy requirements, with high probability. Figure 11b 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.

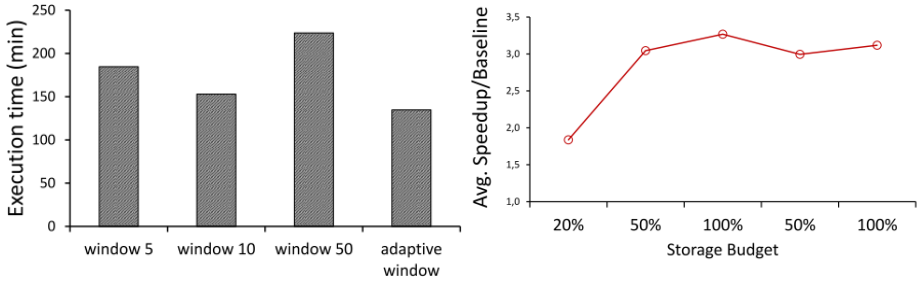
Figure 11c presents the execution time and storage requirements of Taster at each query. Taster's tuner continuously revaluates the synopses stored in the synopsis warehouse, and it frequently drops and build some synopses while executing the queries. At the beginning of each epoch, Taster quickly recognizes 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. 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 order key whereas template of q8 requires stratification on part key. The storage budget for Taster is set to 35GB.

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 11d 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.

We now investigate how Taster adapts to changing storage budget. We run a sequence of 250 TPC-H queries 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 11e 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.



(a) individual performance gains (b) approximation error (c) adaptation to query workload



(d) varying the horizon size (e) varying the storage budget

Figure 11: Experimental results for Taster.

6. Conclusion

The exponential increase of data no longer allows real-time data analytics with exact answers, even with state-of-the-art hardware and parallel implementations. Approximate query processing allows users to get approximate results with error guarantees.

In this report, we have presented QAL, the Query Approximation Layer of SmartDataLake. QAL introduces a novel adaptive approximate processing engine that constructs the synopses which maximize the future throughput. More specifically, QAL proposes multiple approximate plans for an individual SQL query, and then chooses and executes the plan whose synopses will be mostly used in future queries, and stores the generated synopses in a warehouse. The three main modules of QAL are the query planner for generating the logical and physical plans, the query evaluator for choosing between the plans, and the synopsis warehouse for storing the synopses. Unlike the other state-of-the-art AQP engines that only utilize samples, QAL leverages sketches (e.g., Count-min sketch with dyadic ranges) so that it can propose various execution plans for the approximate queries. The synopses are defined as a physical operator, and they are injected into the execution plan. QAL pushes down the synopses in the execution plan that not only decreases the number of records to be calculated but also Increases the reusability of synopses.

The efficiency of QAL heavily depends on the quality of predicted future queries. To address this problem, QAL uses two different approaches: a window-based query prediction and an ML predictive model. The second approach is a novel technique that forecasts the potential approximate operators rather than the future SQL queries. Our preliminary results demonstrate a significantly better performance of the proposed adaptive AQP compared to state-of-the-art AQP.

References

[1] S. Acharya, P. B. Gibbons, and V. Poosala, “Congressional samples for approximate answering of group-by queries,” in *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, 2000, pp. 487–498.

[2] S. Chaudhuri, G. Das, and V. Narasayya, “A robust, optimization-based approach for approximate answering of aggregate queries,” *SIGMOD Rec. (ACM Spec. Interes. Gr. Manag. Data)*, vol. 30, no. 2, pp. 295–306, 2001.

[3] S. Agarwal, A. Panda, B. Mozafari, S. Madden, and I. Stoica, “BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data,” 2012.

[4] B. Babcock, S. Chaudhuri, and G. Das, “Dynamic Sample Selection for Approximate Query Processing,” *Proc. ACM SIGMOD Int. Conf. Manag. Data*, pp. 539–550, 2003.

[5] S. Chaudhuri, G. Das, M. Datar, R. Motwani, and V. Narasayya, “Overcoming limitations of sampling for aggregation queries,” *Proc. - Int. Conf. Data Eng.*, pp. 534–542, 2001.

[6] L. Sidirourgos, M. Kersten, and P. Boncz, “SciBORQ: Scientific data management with bounds on runtime and quality,” *CIDR 2011 - 5th Bienn. Conf. Innov. Data Syst. Res. Conf. Proc.*, pp. 296–301, 2011.

[7] B. Gibbons, M. Avenue, and M. H. Nj, “The ALqua Approximate,” pp. 574–576.

[8] Y. Park, B. Mozafari, J. Sorenson, and J. Wang, “VerdictDB: Universalizing Approximate Query Processing,” 2018.

[9] B. Ding, S. Huang, S. Chaudhuri, K. Chakrabarti, and C. Wang, “Sample + Seek : Approximating Aggregates with Distribution Precision Guarantee,” *Proc. 2016 Int. Conf. Manag. Data*, pp. 679–694, 2016.
[10] J. Peng, D. Zhang, J. Wang, and J. Pei, “AQP++: Connecting approximate query processing with aggregate precomputation for interactive analytics,” *Proc. ACM SIGMOD Int. Conf. Manag. Data*, pp. 1477–1492, 2018.
[11] B. Mozafari *et al.*, “SnappyData : A Unified Cluster for Streaming, Transactions, and Interactive Analytics,” *Cidr*, pp. 1–8, 2017.
[12] S. Kandula *et al.*, “Quickr: Lazily approximating complex AdHoc queries in BigData clusters,” *Sigmod*, pp. 631–646, 2016.
[13] A. Galakatos, A. Crotty, E. Zgraggen, C. Binnig, and T. Kraska, “Revisiting reuse for approximate query processing,” *Proc. VLDB Endow.*, vol. 10, no. 10, pp. 1142–1153, 2017.
[14] J. M. Hellerstein, P. J. Haas, and H. J. Wang, “Online Aggregation,” *SIGMOD Rec. (ACM Spec. Interes. Gr. Manag. Data)*, vol. 26, no. 2, pp. 171–181, 1997.
[15] K. Zeng, S. Agarwal, A. Dave, M. Armbrust, and I. Stoica, “G-OLA: Generalized on-line aggregation for interactive analysis on big data,” *Proc. ACM SIGMOD Int. Conf. Manag. Data*, vol. 2015-May, pp. 913–918, 2015.
[16] K. Zeng, S. Gao, B. Mozafari, and C. Zaniolo, “The Analytical Bootstrap : a New Method for Fast Error Estimation in Approximate Query Processing,” pp. 277–288.
[17] K. Zeng, S. Agarwal, and I. Stoica, “iOLAP: Managing Uncertainty for Efficient Incremental OLAP,” *Proc. SIGMOD*, pp. 1347–1361, 2016.
[18] Z. Liu and J. Heer, “The effects of interactive latency on exploratory visual analysis,” *IEEE Trans. Vis. Comput. Graph.*, vol. 20, no. 12, pp. 2122–2131, 2014.
[19] Y. Wu, L. Xu, R. Chang, J. M. Hellerstein, and E. Wu, “Making Sense of Asynchrony in Interactive Data Visualizations,” vol. 14, no. 8, pp. 1–14, 2015.
[20] A. Kim, E. Blais, P. Indyk, and S. Madden, “Rapid Sampling for Visualizations with Ordering Guarantees,” pp. 521–532.
[21] D. Moritz, D. Fisher, B. Ding, and C. Wang, “Trust, but verify: Optimistic visualizations of approximate queries for exploring big data,” in *Proceedings of the 2017 CHI conference on human factors in computing systems*, 2017, pp. 2904–2915.
[22] S. Rahman *et al.*, “I’ve seen" enough" incrementally improving visualizations to support rapid decision making,” *Proc. VLDB Endow.*, vol. 10, no. 11, pp. 1262–1273, 2017.
[23] Y. Wu, B. Harb, J. Yang, and C. Yu, “Efficient evaluation of object-centric exploration queries for visualization,” *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1752–1763, 2015.
[24] R. Borovica-Gajić, R. Appuswamy, and A. Ailamaki, “Cheap data analytics using cold storage devices,” *Proc. VLDB Endow.*, vol. 9, no. 12, pp. 1029–1040, 2016.
[25] G. Cormode, “Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches,” *Found. Trends Databases*, vol. 4, no. 1–3, pp. 1–294, 2012.
[26] B. H. Bloom, “Space/time trade-offs in hash coding with allowable errors,” *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.
[27] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark: Cluster computing with working sets.,” *HotCloud*, vol. 10, no. 10–10, p. 95, 2010.
[28] M. Zaharia *et al.*, “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing,” in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, 2012, p. 2.
[29] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. S. Spark, “Cluster computing with working sets,” in *HotCloud’10 Proceedings of the 2nd USENIX conference on Hot*

topics in cloud computing, 2010, p. 10.
[30] M. Olma, O. Papapetrou, R. Appuswamy, and A. Ailamaki, "Taster : Self-Tuning , Elastic and Online Approximate Query Processing," *2019 IEEE 35th Int. Conf. Data Eng.*, pp. 482–493, 2019.
[31] G. Cormode, S. Muthukrishnan, and I. Rozenbaum, "Summarizing and mining inverse distributions on data streams via dynamic inverse sampling," in *Proceedings of the 31st international conference on Very large data bases*, 2005, pp. 25–36.
[32] S. L. Lohr, *Sampling: design and analysis*. Nelson Education, 2009.
[33] D. C. Montgomery and G. C. Runger, *Applied statistics and probability for engineers*. John Wiley and Sons, 2014.
[34] A. Krause, C. Guestrin, C. Faloutsos, and J. Vanbriesen, "Cost-effective Outbreak Detection in Networks," pp. 420–429, 2007.
[35] J. Goldstein and P.-åke Larson, "Optimizing Queries Using Materialized Views : A Practical , Scalable Solution," pp. 331–342, 2001.
[36] L. Ma, D. Van Aken, A. Hefny, G. Mezerhane, A. Pavlo, and G. J. Gordon, "Query-based workload forecasting for self-driving database management systems," *Proc. ACM SIGMOD Int. Conf. Manag. Data*, pp. 631–645, 2018.
[37] A. Ganapathi *et al.*, "Predicting multiple metrics for queries: Better decisions enabled by machine learning," *Proc. - Int. Conf. Data Eng.*, pp. 592–603, 2009.
[38] R. Marcus and O. Papaemmanouil, "Plan-structured deep neural network models for query performance prediction," *Proc. VLDB Endow.*, vol. 12, no. 11, pp. 1733–1746, 2019.
[39] Y. Fang, J. Peng, L. Liu, and C. Huang, "WOVSQLI: Detection of SQL injection behaviors using word vector and LSTM," *ACM Int. Conf. Proceeding Ser.*, pp. 170–174, 2018.
[40] A. Kipf, M. Freitag, D. Vorona, P. Boncz, T. Neumann, and A. Kemper, "Estimating Filtered Group-By Queries is Hard: Deep Learning to the Rescue," *1st Int. Work. Appl. AI Database Syst. Appli- cations*, 2019.
[41] R. Marcus and O. Papaemmanouil, "Deep reinforcement learning for join order enumeration," *Proc. 1st Int. Work. Exploit. Artif. Intell. Tech. Data Manag. aiDM 2018*, pp. 0–3, 2018.
[42] R. Marcus *et al.*, "Neo: A learned query optimizer," *Proc. VLDB Endow.*, vol. 12, no. 11, pp. 1705–1718, 2019.
[43] R. Hayek and O. Shmueli, "Improved Cardinality Estimation by Learning Queries Containment Rates," *arXiv Prepr. arXiv1908.07723*, 2019.
[44] N. Potti and J. M. Patel, "DAQ: A new paradigm for approximate query processing," *Vldb*, vol. 8, no. 9, pp. 898–909, 2015.