THE UNIVERSITY OF CHICAGO

SYNOPSES FOR EFFICIENT AND RELIABLE APPROXIMATE QUERY PROCESSING

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCE
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
XI LIANG

CHICAGO, ILLINOIS
FEB 2022

Table of Contents

LIST OF FIGURES vi

LIST OF TABLES ix

ACKNOWLEDGMENTS x

ABSTRACT xi

1 INTRODUCTION 1
- 1.1 Explicit and Implicit Trade-offs Made by AQP Techniques 1
- 1.2 Designing Synopses for Better Trade-offs 4

2 COMBINING AGGREGATION AND SAMPLING (NEARLY) OPTIMALLY FOR APPROXIMATE QUERY PROCESSING 7
- 2.1 Introduction 7
- 2.2 Background and Problem Setup 11
  - 2.2.1 Uniform Sampling 12
  - 2.2.2 Stratified Sampling 14
  - 2.2.3 Stratified Aggregation 14
  - 2.2.4 Related Work 16
- 2.3 Overview and Query Processing 18
  - 2.3.1 Usage 19
  - 2.3.2 Partition Trees and Samples 20
  - 2.3.3 Query Processing 22
  - 2.3.4 Other Optimizations 23
- 2.4 Optimizing the Partitioning 24
  - 2.4.1 Objective and Search Space 24
  - 2.4.2 Partitioning Algorithm Intuition 25
  - 2.4.3 Algorithm in 1D 27
  - 2.4.4 Algorithm in Higher Dimensions 31
  - 2.4.5 Summary 32
- 2.5 Experiments 33
  - 2.5.1 Experiment Setup 33
  - 2.5.2 Accuracy Evaluation 37
  - 2.5.3 Approximated Dynamic Programming Partitioning vs. Equal Partition 39
  - 2.5.4 Multidimensional Query Templates 40
  - 2.5.5 End-to-End Comparison with Other Systems 43
- 2.6 Conclusion 45

3 JANUSAQP: EFFICIENT PARTITION TREE MAINTENANCE FOR DYNAMIC APPROXIMATE QUERY PROCESSING 46
- 3.1 Introduction 46
- 3.2 Background 49

3.2.1 Dynamic Approximate Query Processing 49
3.2.2 Related work 50
3.2.3 Partition Trees for AQP 52
3.3 System Architecture 55
3.3.1 Construction and Optimization API 55
3.3.2 Data and Query API 57
3.3.3 Summary and Algorithmic Contributions 57
3.4 Dynamic Partition Trees 58
3.4.1 Incrementally Maintaining Nodes 58
3.4.2 Maintaining Stratified Samples 60
3.4.3 Re-initialization and Catch-Up 61
3.4.4 Answering Queries With a DPT 62
3.5 Optimal DPT Partitioning 65
3.5.1 Preliminaries and Problem Setup 65
3.5.2 Partitioning for $d = 1$ 67
3.5.3 Partitioning in Higher Dimensions 72
3.5.4 Re-Partitioning Triggers 73
3.6 Experiments 76
3.6.1 Setup 76
3.6.2 Accuracy 78
3.6.3 Performance 79
3.6.4 Work with Real-life Dynamic Data 80
3.6.5 Partial Repartition 80
3.6.6 The Catch-up Phase 80
3.6.7 Multi-dimensional Query Templates 81
3.6.8 Re-partitioning 82
3.6.9 Comparing with PASS 83
3.7 Discussion and Conclusion 84

4 FAST AND RELIABLE MISSING DATA CONTINGENCY ANALYSIS WITH PREDICATE-CONSTRAINT 90
4.1 Introduction 90
4.2 Background 92
4.2.1 Example Application 92
4.2.2 Related Work 94
4.3 Predicate-Constraints 97
4.3.1 Predicate-Constraint 97
4.3.2 Predicate-Constraint Sets 100
4.4 Calculating result ranges 100
4.4.1 Cell Decomposition 101
4.4.2 Integer-Linear Program 104
4.4.3 Complexity 106
4.4.4 Numerical Example 107
4.5 Joins Over Predicate Constraints 109
4.5.1 Naive Method 109

4.5.2 A Better Bound For Natural Joins 110
4.6 Experiments 112
4.6.1 Experimental Setup 112
4.6.2 Intel Wireless Dataset 115
4.6.3 Detailed Sampling Comparison 116
4.6.4 Scalability 119
4.6.5 Handling MIN, MAX, and AVG Queries 121
4.6.6 Additional Datasets 122
4.6.7 Probabilistic Confidence Intervals are Often Unreliable on Real Data 126
4.7 Conclusion 127

5 OPPORTUNISTIC VIEW MATERIALIZATION WITH DEEP REINFORCEMENT LEARNING 129
5.1 Introduction 129
5.2 Background 132
5.2.1 Scope 133
5.2.2 Related Work 133
5.2.3 Problem Statement 135
5.2.4 What Does DQM Learn? 136
5.3 System Overview 138
5.3.1 View Manager 139
5.3.2 View Candidate Miner 139
5.3.3 Query Re-Writer 140
5.4 Learning Materialization 142
5.4.1 The One View Problem 143
5.4.2 Generalizing to N Views 148
5.5 View Eviction 150
5.5.1 Submissive Eviction 150
5.5.2 Algorithm 151
5.5.3 View Maintenance Through Eviction 152
5.6 Experiments 152
5.6.1 Join Order Benchmark 153
5.6.2 CubeLoad 160
5.6.3 TPC-DS 160
5.6.4 Micro-Benchmarks 161
5.7 Conclusion 165

6 CONCLUSION 166
6.1 Conclusion 166

REFERENCES 169

List of Figures

2.1 Temperatures collected over 20 time-steps and aggregated into 4 partitions. Partitioned aggregates can be used to decompose SUM/COUNT/AVG queries into an exact and approximate components. 8
2.2 PASS summarizes a dataset with a tree of aggregates at different levels of resolution (granularity of partitioning). Associated with the leaf nodes are stratified samples. We present an algorithm to optimize over such a structure for fast and accurate approximate query processing. 10
2.3 Median relative error of 2000 random SUM queries on the 3 real-life datasets using a varying number of partitions and a fixed sample rate of 0.5%. 35
2.4 Median relative error of 2000 random SUM queries on the 3 real-life datasets using a varying sample rate and a fixed number of partitions of 64. 38
2.5 Median confidence interval ratio of 2000 random SUM queries on the 3 real-life datasets using a varying sample rate and a fixed number of partitions of 64. . . 38
2.6 Median confidence interval ratio of Approximated Dynamic Programing partitioning (ADP) vs. Equal Partitioning (EQ) on a synthetic adversarial dataset. . 40
2.7 Median confidence interval ratio of ADP vs. EQ on challenging queries of the 3 real-life datasets. 41
2.8 Multidimensional predicates on the NYC Taxi dataset. (Left) The median confidence interval ratio of KD-PASS vs. KD-US. (Right) The skip rate of KD-PASS. 43
2.9 We use the aggregates constructed for a 2D query template to solve query templates of other dimensions. Left figure shows the Median confidence interval ratio of KD-PASS vs. KD-US on the NYC Taxi dataset. Right figure shows the percentage of tuples that are skipped by KD-PASS. 44

3.1 JanusAQP manages a collection of DPT synopses by maintaining them online while periodically re-optimizing partitioning and sample allocation. 47
3.2 The core data structure in JanusAQP is based on the PASS data structure [87] that summarizes a dataset with a tree of aggregates at different levels of resolution (granularity of partitioning). Associated with the leaf nodes are stratified samples. The two stage synopsis structure can be optimally partitioned to minimize error. 85
3.3 The DPT update process for an insertion or deletion. (1) A set of samples is maintained using a reservoir sampling algorithm. (2) The leaf node statistics are incrementally updated. (3) The updated statistics from the leaf node propagate to the parents. (4) Updated statistics from the parents propagate all the way to the root. 86
3.4 JanusAQP synopses can be re-initialized online using a multi-threaded implementation to minimize unavailability 87
3.5 We compare the accuracy (left plot) and re-optimization cost (right plot) of JanusAQP with DeepDB and reservoir sampling on the NYC Taxi dataset. . . 87
3.6 We evaluate the throughput (left plot) of JanusAQP when handling insertions and deletions in multi-thread mode. We also compare the query latency (right plot) with DeepDB. 88

3.7 Varying the catch-up goal from 1% to 10% of the data, we evaluate the accuracy of JanusAQP (left plot) and the time cost of the catch-up phase (right plot). . 88
3.8 We compare JanusAQP with DeepDB on multi-dimensional queries. Both DeepDB and JanusAQP perform re-optimization after every 10% insertion then evaluate the same set of queries. We record the median relative error on the left plot and the re-optimization cost on the right plot. 89
3.9 We compare the accuracy of JanusAQP and DPT in two scenarios that cause imbalanced partition trees. 89

4.1 Simple Extrapolation could introduce significant error when missing rows are correlated (e.g, tend to have the highest values). 94
4.2 Predicates in a predicate-constraint set are possibly overlapping. The first step of the algorithm is to decompose a set of predicate-constraints into disjoint cells. 102
4.3 Performance of baselines given different missing ratio, evaluated with 1000 COUNT(*) query on the Intel Wireless dataset. 116
4.4 Performance of baselines given different missing ratio, evaluated with 1000 SUM query on the Intel Wireless dataset. 117
4.5 Performance of the uniform sampling baseline with different sample size. 118
4.6 We investigate the sensitivity of Corr-PC, Overlapping-PC, and US-10n to different levels of noise. The failure rate of all approaches increases as we increase the noise level, but the PC baselines especially Overlapping-PC are more tolerable to the same level of noise. 119
4.7 Our optimizations reduce the number of cells evaluated during the cell decomposition phase by over a 1000x. 120
4.8 The run time needed to solve one query using partition of different size. 121
4.9 With PC, an optimal bound can be derived for MIN and MAX queries. PC also generates competitive result for AVG Queries. 122
4.10 Baseline performance on 1000 COUNT(*) and SUM queries with predicate attributes on *Latitude* and *Longitude* using the Airbnb NYC dataset. 123
4.11 Baseline performance on 1000 COUNT(*) and SUM queries with predicate attributes on *Port* and *Date* using the Border crossing dataset. 124
4.12 We compare the bound derived by our approach (Corr-PC) with state of the art baseline Elastic Sensitivity on the triangle counting problem of different table sizes (TOP) and an acyclic join (BOTTOM). 128

5.1 DQM runs as an independent process that issues view creation and deletion actions to Apache SparkSQL. A thin wrapper layer around SparkSQL manages the created views and returns any runtime results to DQM. DQM learns from these observations and issues creation and deletion events when appropriate. It also issues potential experiments to run. 138

5.2 We diagram the "rollout", or data collection, process used in DQM. At each time-step, DQM decides whether to create a view or not. A reward is received if a created view is used *AND* it improves a query runtime. Every time a created view is used, it queues up an experiment in (B) to run when the system is idle. Once the experiment is run, that observation of improvement is placed in (A) and can be used to improve the policy. 146
5.3 DQM maintains a table to prioritize which views to delete to enforce the storage constraint. This table is continuously updated with rewards accrued in the experience buffer. 151
5.4 We evaluate overhead caused by learning and exploration by comparing DQM with a TrainedDQM with no exploration. 153
5.5 We compare DQM to all of the baselines on the rzipf workload. Even including learning time, DQM is competitive with the best baselines. 154
5.6 We run DQM without true runtimes and an improvement metric derived from a cost model. While this version of DQM still performs reasonably well, the use of true runtimes is a strength of the RL-based algorithm. We also evaluate a more realistic implementation of Recycler by using the cost model estimated view costs instead of the true costs and this change drastically affects Recycler's performance.156
5.7 (A) We compare DQM to selected baselines on all workloads. (B) We compare DQM to selected baselines with periodic view maintenance (every 100 queries) on the JOB workloads. 158
5.8 We measure the performance of DQM as a function of the storage constraint on the JOB rzipf workload. The storage constraint is presented as normalized by a fraction of candidate views that could possibly be materialized at any time. . . . 159
5.9 We compare DQM and selected baselines on the CubeLoad workload. 160
5.10 We compare DQM to selected baselines on TPC-DS. DQM is competitive (or outperforms) the best heuristic on all the test scenarios. 161
5.11 DQM using different exploration terms on JOB with the rzipf workload. 161
5.12 We study the performance of DQM with different workload size. 162
5.13 We measure the performance of DQM using different delays of reward on JOB rzipf workload. 162
5.14 We combine our eviction policy with random view selection then compare its performance with DQM and selected baselines. 164

List of Tables

2.1 Controlling for worst-case query latency (total number of tuples processed), we demonstrate that it is possible to construct a synoposis that is highly accurate (less than .1% error) across 2000 random SUM/COUNT/AVG queries. The caveat is a high upfront optimal partitioning cost. 36
2.2 We compare the median relative error of three PASS variations with VerdictDB and DeepDB on workloads we used in previous experiments. We measure the average latency of query processing, the storage, and the construction time (training time for DeepDB) required by each approach. 42
2.3 Preprocessing cost, mean latency, max latency and accuracy given different number of partitions. 44

3.1 Table of basic notation 59
3.2 We compare our new binary search-based (BS) partitioning algorithm with the dynamic programming-based (DP) algorithm proposed by PASS[87] on the Intel Wireless dataset. 84

4.1 Trade-off between failure rate and accuracy of an uniform sampling baseline given different confidence interval vs. Corr-PC. 117
4.2 Over a 1000 randomly chosen predicates, we record the number of failure events of different "error bound" frameworks. A failure event is one where an observed outcome is outside the range of the returned bounds. For a 99% confidence interval, as used in our experiments, one would expect a 1% failure rate for the sampling-based frameworks—but is significantly higher in practice for small skewed datasets. PCs, and as a special case Histograms, are guaranteed not to fail if the assumptions are satisfied. 125

ACKNOWLEDGMENTS

TODO

ABSTRACT

Answering queries accurately at interactive speeds has become more challenging in modern data systems due to the massive growth of data. Such challenges lead to an increasing interest in Approximate Query Processing (AQP) techniques because they enable timely query execution in scenarios that can tolerate some degree of inaccuracy. While latency and accuracy have been the two main factors considered by many AQP systems, in our studies, we found other dimensions like applicability, reliability, robustness and data availability, etc. could also be the main considerations in certain scenarios and such demands call for the design of novel AQP techniques.

In this thesis, we propose novel AQP techniques of different characteristics for different scenarios where AQP can be useful. We first discuss PASS, a system that combines sampling and aggregation for better accuracy while keeping the latency and storage cost at a favorable level. As a follow-up of PASS, we present JanusAQP, a dynamic AQP system that extends the static partition tree proposed in PASS and addresses several challenges in a dynamic environment that make the system more practical. Thirdly, we propose PC, a novel missing-data analysis framework that not only enables a presentation of missing data but also the derivation of a tight hardbound for optimal reliability. Lastly, we discuss DQM, our effort in applying machine learning to manage materialized views in a robust manner.

CHAPTER 1

INTRODUCTION

The massive growth of data poses unprecedented challenges to efficient data system design. The data systems that rely on an in-memory query processing engine are risking of being eventually outpaced by the exponential growth of data and become the bottleneck of the analytical pipeline. With the potential to be the ultimate solution of handling arbitrarily large amount of data, approximate query processing (AQP) haven been studied for decades to leverage the fact that a perfect answer is not always needed.

An approximate answer is good enough as long as it can lead the user to the same optimal decision. This is the case in many scenarios, for example, in data visualization, the number of pixels caps the highest accuracy that can be displayed not to mention that small differences are often indistinguishable by human eyes; in feature selection, as long as the important features are ranked as the top ones (even in a different order), they will be selected to build the same optimal machine learning model; in non-critical sensor analysis like thermometer measures, users wouldn't mind if the average temperature of the past week is reported a bit higher than the actual value, etc. In these scenarios, good-enough but imperfect query results are favored if they can be generated with low latency and some guarantees of the error.

However, existing AQP techniques are still of unsatisfactory [25] in a sense that there isn't one single AQP technique that can fit all scenarios and it is necessary for developers to understand the trade-offs that are made by different AQP techniques in order to adopt them to solve real-life problems.

1.1 Explicit and Implicit Trade-offs Made by AQP Techniques

Despite the explicit trade-off between accuracy and lateny, there are a couple other dimensions that are in play. Some of the trade-offs are made implicitly but they should be considered when design or select an AQP technique. We summarize all the trade-offs as below:

- **Precomputation Cost**: Depending on whether a technique pays up-front computation cost before processing any query we can classify it into the online or the offline category [84]. Online techniques does not require precomputation because they select sample on-the-fly in query time. For offline techniques, such a precomputation can be workload-dependent, during which the technique utilize prior knowledge that is often derived from previous workload to compose a summary of the data that aligns with the workload [74, 60, 31] or even to optimize the sample selection process [6]. A workload-independent technique on the other hand does not rely on previous queries, they can derive a summary solely from the data [2, 24], or even build a machine learning model that learns from the data [56].

- **Latency**: Due to the fact that online techniques are selecting samples on-the-fly, it is often the case that they have a higher latency than the offline technique that can take advantage of pre-built sample and/or indices.

- **Applicability**: As a side-effect of being workload-dependent, techniques that relies on previous workload often implicitly trade-off its generality or applicability. I.e., they can only be used to solve queries that are a subset of the (combination of) previous queries. For example, queries that are from the same template[87] or queries that are from the same query column set (QCS) [6]. In these systems, ad-hoc queries are often handled by falling back to other methods or an answer of large error is returned. However, in the case of a recurring workload or if the future workload is highly correlated with previous workload, such a workload-dependent technique can be much more accurate and efficient.

- **Storage and Memory Cost**: Offline techniques usually store the synopses they pre-compute to avoid unnecessary re-computation and load these precomputed synopses into memory when initialize. Especially for the workload-dependent techniques, the storage cost could grow exponentially with the number of previous workloads (or query columns) that are considered. Selecting the subset of synopses to store can be treated as an optimization problem as shown in [6].

- **Reliability**: Many AQP techniques can compute an error estimation in posterior but not a prior, i.e., they can compute an error estimation only after a query is processed but not before, which could lead to a potential SLA violation. Sample+Seek [38] propose a method to compute query-independent accuracy guarantee in terms of a distribution error but it is still an open problem to compute accuracy guarantee in other error models preferred by user (e.g. mean square error). As such, we argue that reliability is also an implicit trade-off that is made by online techniques because for highly skewed data, there is a higher chance that the online techniques would fail with large error, while such a skewness could be identified and resolved in the offline precomputation phase.

- **Data Availability**: Another implicit trade-off made by online technique is that they assume all the data needed to process a query are readily available for online sampling, this is not always the case if part of the data is missing or corrupted (e.g., due to file system or network outage). Offline techniques have a better chance to study whatever is still available and provide a better solution for missing data analysis. However, existing techniques like m-table [126] only provides a system that can represent the missing data, but lack the ability to do any computation.

- **Maintenance and Robustness**: Lastly, for (offline) techniques that stores and (re-)use results or synopses for query answering, such synopses have to be maintained when the underlying data or the workload changes. Maintenance of such synopses are often non-trivial, and there are two main challenges: 1) when to perform the maintenance to make sure the answer is still good enough, should we do it eagerly or lazily? 2) how to perform the maintenance efficiently so that it won't compromise the performance of the system? Previous works propose techniques to clean staled materialized view [76] and to maintenance of a Count-Min sketch in a data streaming setting [97] but it is always an problem to be solved when design a new dynamic AQP technique.

1.2 Designing Synopses for Better Trade-offs

The fact that no single AQP technique can meet demands of different applications leads to opportunities and challenges. Our efforts in designing new synopses touch every dimension we mentioned in the previous section: 1) **PASS** [87] is a synopses that combines sampling and aggregates near optimally. It sacrifice applicability and precomputation for better accuracy and reasonable latency and storage cost; 2) **JanusAQP** is a AQP system that extends the static partition tree proposed in PASS and addresses several challenges introduced by a dynamic environment; 3) **PC** [86] is a framework that can be used for missing data analysis based on our novel synopses called a predicate-constraint set. PC is designed to compute a tight hard bound of query results under specified constraints, therefore offers a better trade-off between accuracy, reliability and data availability; 4) **DQM** [85] focus on maintaining materialized views with reinforcement learning to provide a better trade-off between accuracy, maintenance and robustness of the system. We now give each system a high level overview as following.

PASS[87]: Sample-based approximate query processing (AQP) suffers from many pitfalls such as the inability to answer very selective queries and unreliable confidence intervals when sample sizes are small. Recent research presented an intriguing solution of combining materialized, pre-computed aggregates with sampling for accurate and more reliable AQP. We explore this solution in detail in this work and propose an AQP physical design called PASS, or Precomputation-Assisted Stratified Sampling. PASS builds a tree of partial aggregates that cover different partitions of the dataset. The leaf nodes of this tree form the strata for stratified samples. Aggregate queries whose predicates align with the partitions (or unions of partitions) are exactly answered with a depth-first search, and any partial overlaps are approximated with the stratified samples. We propose an algorithm for optimally partitioning the data into such a data structure with various practical approximation techniques.

As a followup of PASS, **JanusAQP** propose a more efficient partitioning algorithm to reduce the main overhead of constructing a static partition tree data structure. Also, to overcome the limitation of the static partition tree of PASS that works only with static data, JanusAQP propose techniques that lead to a dynamic partition tree that can handle insertions and deletions of data. Furthermore, JanusAQP also address challenges when adopting an AQP system into an existing data system. To work with arbitrarily large amount of existing data, JanusAQP use a catch-up phase that let a user decide the cost to be paid to work with existing data. JanusAQP is also integrated with Apache Kafka as a consumer of the message queue to demonstrate how it can work with an existing system without changing its physical design.

PC[86]: Data analysts largely rely on intuition to determine whether missing or withheld rows of a dataset significantly affect their analyses. We propose a framework that can produce automatic contingency analysis, i.e., the range of values an aggregate SQL query could take, under formal constraints describing the variation and frequency of missing data tuples. We describe how to process SUM, COUNT, AVG, MIN, and MAX queries in these conditions resulting in hard error bounds with testable constraints. We propose an optimization algorithm based on an integer program that reconciles a set of such constraints, even if they are overlapping, conflicting, or unsatisfiable, into such bounds. We also present a novel formulation of the Fractional Edge Cover problem to account for cases where constraints span multiple tables. Our experiments on 4 datasets against several statistical imputation and inference baselines show that statistical techniques can have a deceptively high error rate that is often unpredictable. In contrast, our framework offers hard bounds that are guaranteed to hold if the constraints are not violated. In spite of these hard bounds, we show competitive accuracy to statistical baselines.

DQM[85]: We study using deep reinforcement learning to learn adaptive view materialization and eviction policies. Our insight is that such selection policies can be effectively trained with an asynchronous RL algorithm, that runs paired counterfactual experiments during system idle times to evaluate the incremental value of persisting certain views. Such a strategy obviates the need for accurate cardinality estimation or hand-designed scoring heuristics. We focus on views with inner-joins, predicates, aggregate functions and modeling effects in a main-memory, OLAP system. Our research prototype system, called DQM, is implemented in SparkSQL and we experiment on several workloads including the Join Order Benchmark, the TPC-DS workload and a workload that is generated by a generator that can simulate real-life usage of OLAP systems. Results suggest that: (1) DQM can outperform heuristic when their assumptions are not satisfied by the workload or there are temporal effects like period maintenance, (2) even with the cost of learning, DQM is more adaptive to changes in the workload, and (3) DQM is broadly applicable to different workloads and skews.

In the rest of this thesis, we present the aforementioned new synopses designed for efficient, reliable and robust AQP in Chapter 2 (PASS), 3 (JanusAQP), 4 (PC), and 5 (DQM). Lastly, we conclude in Chapter 6.

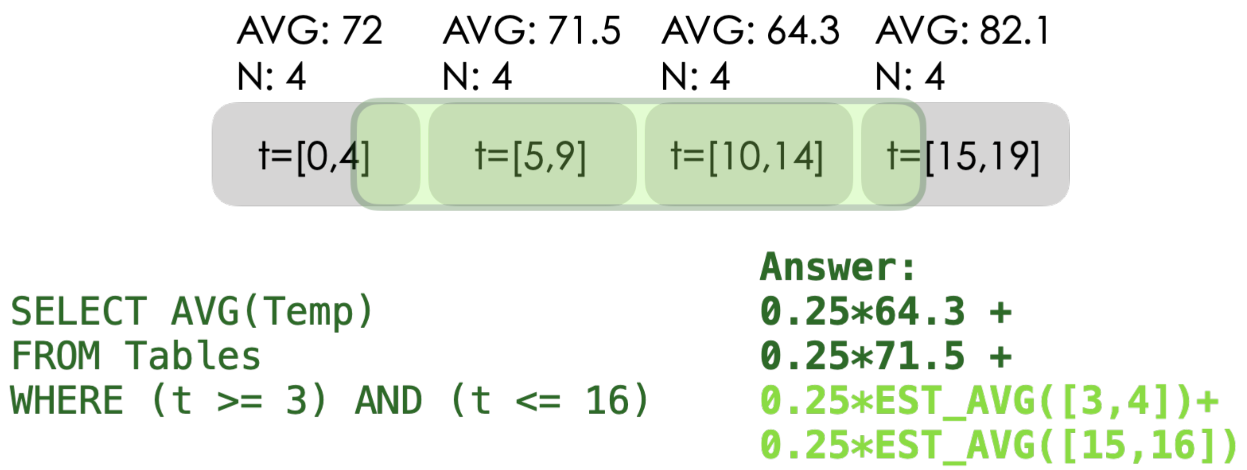
CHAPTER 2
COMBINING AGGREGATION AND SAMPLING (NEARLY) OPTIMALLY FOR APPROXIMATE QUERY PROCESSING

2.1 Introduction

There are a number of applications where exact query results are unnecessary. For example, visualizations only require precision up to screen and human perceptual resolutions [68, 89]. Similarly, in exploratory data analysis where users may be only looking for broad trends, exact numerical results are not needed [135]. In industrial workloads, such queries are not only highly prevalent, but they are also highly resource-intensive: Agarwal et al. found that roughly 30% of a Facebook workload consists of aggregate queries over tables larger than 1TB [6]. Examples such as these have motivated nearly 40 years of approximate query processing (AQP) research, where a database deliberately sacrifices accuracy for faster [32, 70] or more resource-efficient results [76] for aggregate queries.

Data sampling has been the primary approach used in AQP since the research area's conception [105]. To this day, there are new results in sampling techniques [131], implementation [69], and mechanisms [138]. The perennial research interest in data sampling stems from its generality as an approximation technique and the extensive body of literature in sampling statistics to quantify the error rate in such approximations. However, small samples may not contain the relevant data for highly selective queries, and this can lead to confusing or misleading results. Recent work mitigates this problem by using an anticipated query workload to prioritize sampling certain regions of the database to support such selective queries (called stratified sampling) [7, 24], or construct samples online during query execution [70, 131].

Consequently, pure uniform sampling is rarely used on its own, and most practical sampling systems leverage workload information to ensure that the system can reliably answer selective queries [3, 24, 7, 12, 22, 23, 47, 38]. These systems often run offline optimization



```
SELECT AVG(Temp)
FROM Tables
WHERE (t >= 3) AND (t <= 16)
```

```
Answer:
0.25*64.3 +
0.25*71.5 +
0.25*EST_AVG([3,4])+
0.25*EST_AVG([15,16])
```

Figure 2.1: Temperatures collected over 20 time-steps and aggregated into 4 partitions. Partitioned aggregates can be used to decompose SUM/COUNT/AVG queries into an exact and approximate components.

routines to materialize optimal samples to answer future queries. If we can tolerate expensive, up-front sample materialization costs, there is a natural question of whether it is also valuable to expend resources to compute helper “full dataset” query results. There have been a number of different proposals that do exactly this [61, 44, 113, 132, 76]; where systems leverage pre-computed exact aggregates to help estimate the result of a future query. Not too far off from such proposals are a number of recent approaches to use machine learning for query result estimation [56, 134].

Despite the handful of research papers on the subject, we find that the theory on how to best leverage both pre-computed aggregates and sampling is limited. Existing work often assumes one of the sides is fixed: Galakatos et al. optimize sampling given that they have cached previously computed query exact results [44], or Peng et al. optimize aggregate selection given a uniform sample of data [113]. Such piecewise optimization results in an incomplete understanding of the worst-case error of the data structure. Systems have to balance a number of complex, interlinked factors: (1) precomputation/optimization time, (2) storage space, (3) query latency, and (4) query accuracy. Tuning such structures for a desired accuracy SLO can be significantly challenging for a user.

The joint optimization, over both sampling *and* precomputation, is complex because one has to optimize over a combinatorial space of SQL aggregate queries, while accounting for the real-valued effects of sampling. *The core insight of this paper is to formalize a connection between pre-computed aggregates and stratified sampling.* We interpret pre-computed aggregates as a sort of index that can guide sampling rather than a simple materialized cache of query results. Figure 2.1 illustrates a motivating example, where temperature readings over 20 time-steps are aggregated into 4 partitions. Any new AVG query over a time range can be decomposed into two parts: an exact part where the range fully covers intersecting partitions, and an approximate part where the range partially covers a partition. Thus, we will show a hierarchy of partitioned aggregates that can act as an efficient sample selector—determining which stratified samples of data are relevant for answering a query. This formulation leads to a simple formula extension of stratified sampling variance, and a partitioning optimization objective that can control for the worst-case query result error from the synopsis.

We propose a new AQP data structure called PASS, or Precomputation Assisted Stratified Sampling (illustrated in Figure 2.2). PASS is given a precomputation time budget and a query latency constraint, and it generates a synopsis data structure constructed of both samples and aggregates. The more work that PASS is allowed to do upfront, the more accurate future queries are. PASS first generates a hierarchical partitioning of the dataset (a tree). For each partition (nodes in the tree), we calculate the SUM, COUNT, MIN, and MAX values of the partition. Associated with the leaf nodes is a uniform sample of data from that partition (effectively a stratified sample over the leaves). The tree-like structure acts as an index, allowing us to efficiently skip irrelevant or inconsequential partitions to the query results. Crucially, this lends to an analytic form for query result variance for SUM, COUNT, and AVG queries with predicates.

PASS gives the end user a stronger guarantee about the worst-case end-to-end accuracy than any recent "hybrid" AQP work [44, 113, 111, 56]. In practice, we find that the data structure is often empirically beneficial compared to alternatives. PASS is more general than the work proposed in Gan et al. and supports a wider set of queries [46]. We also find that

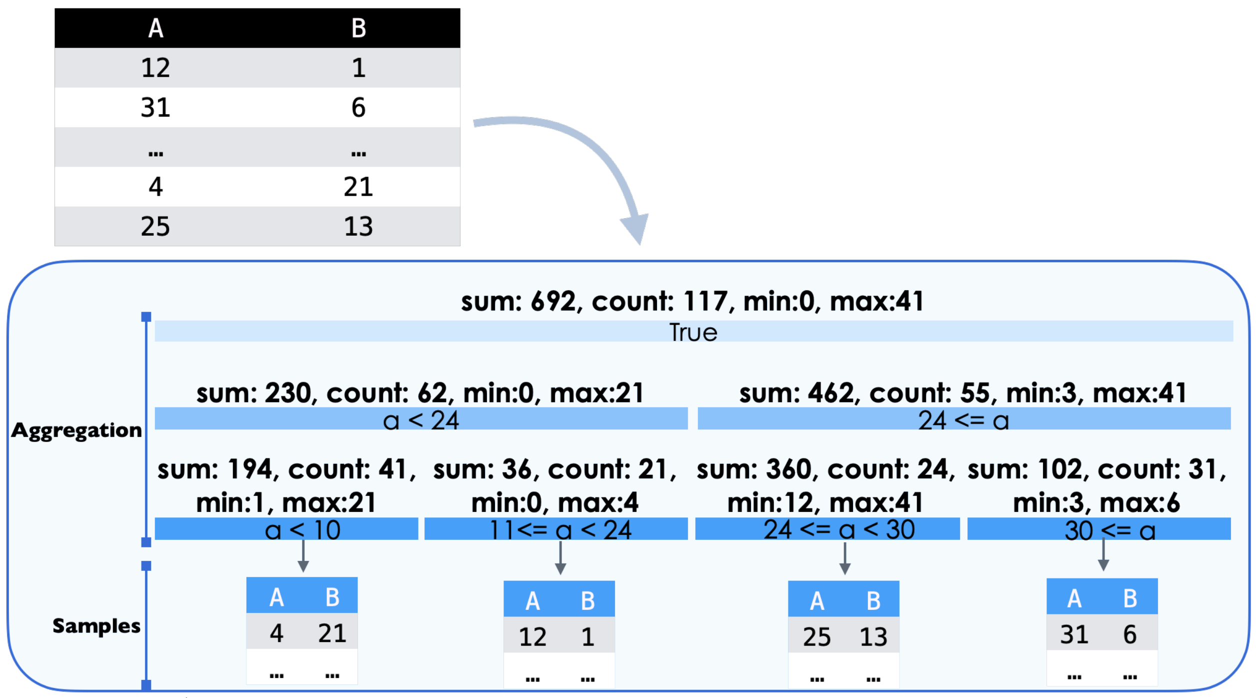

A
B
12
1
31
6
...
...
4
21
25
13
sum: 692, count: 117, min:0, max:41
True
Aggregation
sum: 230, count: 62, min:0, max:21
a < 24
sum: 462, count: 55, min:3, max:41
24 <= a
sum: 194, count: 41, min:1, max:21
a < 10
sum: 36, count: 21, min:0, max:4
11<= a < 24
sum: 360, count: 24, min:12, max:41
24 <= a < 30
sum: 102, count: 31, min:3, max:6
30 <= a
Samples
A B
4 21
A B
12 1
A B
25 13
A B
31 6

Figure 2.2: PASS summarizes a dataset with a tree of aggregates at different levels of resolution (granularity of partitioning). Associated with the leaf nodes are stratified samples. We present an algorithm to optimize over such a structure for fast and accurate approximate query processing.

PASS is empirically more accurate than AQP++ [113], and can provably scale to much larger pre-computed sets. A secondary benefit of PASS is that aggregate hierarchy can compute worst-case estimation error (a 100% confidence interval) for common queries since we know the true extrema and the true cardinality of each partition (similar to [80]). To the best of our knowledge, no other commonly used sample-based data structure offers this benefit.

Of course, the data structure is only as good as its optimization objective. We note that PASS is sensitive to the expected workload and is more expensive to construct. However, we contend that PASS is a step towards a comprehensive understanding of how sampling and precomputation can be combined for fast and accurate AQP and find that if we control for query latency and the amount of precomputation, our results are generally more accurate. In summary, we contribute:

1. A new data structure for AQP called PASS which supports SUM, COUNT, AVG, MIN, and MAX queries with predicates.

2. An optimization algorithm to generate the structure from real data that finds a partitioning that minimizes the maximum sampling variance of a set of possible expected queries.

3. Experiments that show that PASS is often more accurate than uniform sampling, stratified sampling, and AQP++.

2.2 Background and Problem Setup

We start with a simplified model: a "one-dimensional" approximate query processing problem. Consider a collection of N tuples representing numerical data $P = \{(c_i, a_i)\}_{i=1}^{N}$, where one collects numerical value measurements a_i and attributes about those measurements c_i (e.g., a description about what the measurement represents); generically denoted as A and C respectively, when we are not interested in a particular tuple.

For example, one could have a dataset of times (attributes) and temperature measurements (values):

```
(00:01, 70.1C), (00:02, 70.4C),..., (15:53, 69.9C)
```

Over such a collection of data, we would like to be able to answer the following "subpopulation-aggregate" queries: SUM, COUNT, and AVG aggregations of the numeric measurements A over subpopulations determined by filters (predicates) over the C.

However, in approximate query processing, we would like a sub-linear time (preferably constant time) answer with a tolerable approximation error. This problem is fundamentally a data structure question, namely, how to summarize the collection P into a synopsis that can approximately answer the desired queries. Leading to the following more precise problem statement: *derive a data structure from the collection P that occupies no more than $\tilde{\mathcal{O}}(K)$ space and can answer any subpopulation-aggregate query with approximation error provable guarantees in $\tilde{\mathcal{O}}(K)$ time where K is a user-defined parameter much less than N.*

2.2.1 Uniform Sampling

The simplest such data structure is a uniform sample. From P, we can first sample a subset S of size K uniformly; that is, every tuple is sampled with equal probability. Such uniform samples of numbers have the property that the averages within the sample approximate (tend towards with bounded error) the average of the population from which the sample is derived. So, we can approximate our desired queries by first re-formulating them as different average-value calculations. We first define some notation:

- t a tuple (c, a).
- $f(\cdot)$: a function representing any of the supported aggregates.
- $\mathsf{Predicate}(t)$: the predicate of the aggregate query, where $\mathsf{Predicate}(t) = 1$ or 0 denotes t satisfies or dissatisfies the predicate, respectively.
- K: the number of tuples in the sample.
- K_{pred}: the number of tuples that satisfy the predicate in the sample.
- a: the numerical value.

We can reformulate SUM, COUNT, and AVG queries as calculating an average over transformed attributes:

$$f(S) = \frac{1}{K} \sum_{t \in S} \phi(t) \tag{2.1}$$

where $\phi(\cdot)$ expresses all of the necessary scaling to translate the query into an average value calculation:

- COUNT: $\phi(t) = \mathsf{Predicate}(t) \cdot N$
- SUM: $\phi(t) = \mathsf{Predicate}(t) \cdot N \cdot a$
- AVG: $\phi(t) = \mathsf{Predicate}(t) \cdot \frac{K}{K_{pred}} \cdot a$

In order to represent $AVG(S)$ in the form of Equation 2.1, we rewrite it to the following equivalent Equation:

$$AVG(S) = \frac{1}{K} \sum_{t \in S} \mathsf{Predicate}(t) \cdot \frac{K}{K_{pred}} \cdot a. \tag{2.2}$$

Therefore, we have $\phi(t) = \mathsf{Predicate}(t) \cdot \frac{K}{K_{pred}} \cdot a$ for the AVG query.

Error Rate

Let us denote the $\phi(P)$ and $\phi(S)$ as taking the transformation functions above and applying them to every tuple in P or S respectively. The Central Limit Theorem (CLT) states that these empirical mean values tend towards a normal distribution centered around the population mean:

$$N(mean(\phi(P)), \frac{var(\phi(P))}{K}) \tag{2.3}$$

Since the estimate is normally distributed, we can define a confidence interval parametrized by λ (e.g., 95% indicates $\lambda = 1.96$)[1].

$$mean(\phi(S)) \pm \lambda \sqrt{\frac{var(\phi(S))}{K}}. \tag{2.4}$$

To understand the main pitfall of uniform sampling, notice the main scaling factor in the AVG queries is $\frac{K}{K_{pred}}$. The more selective a query is (i.e., smaller K_{pred}), the smaller the effective sample size is. If your sampling rate is 10% but your predicate matches with only 1% of the tuples in a database, then your effective sample size for that query is 0.1%!

Not only do selective queries increase the error in your result estimates, but they also make the confidence interval estimates less reliable. Accurately estimating the variance from a very small sample is often harder than estimating the result itself since variance is a measure of spread. Furthermore, the CLT holds asymptotically and is naturally less reliable at small sample sizes.

1. When estimating means of finite population there is a finite population correction factor of $FPC = \frac{N-K}{N-1}$ which scales the confidence interval.

2.2.2 Stratified Sampling

Stratified sampling is one way to mitigate the effects of selective predicates. Instead of directly sampling from P, we first partition P into B strata, which are mutually exclusive partitions defined by groupings over C. Within each stratum $P_1, ..., P_B$, we uniformly sample as before resulting in samples $S_1, ..., S_B$. So, instead of a single parameter K which controlled the accuracy in the uniform sampling case, we have a $K_1, ..., K_B$ for each stratum. The sum of all K_i can be equated to the uniform sampling size to compare efficiencies $K = \sum_{i=1}^{B} K_i$.

The results estimation scheme in the previous section can be applied to each of the strata treating it as a full dataset. We combine the estimates with a simple weighted average:

$$\sum_{i=1}^{B} est(S_i) \cdot w_i$$

For SUM/COUNT $w_i = 1$. For AVG $w_i = \frac{N_i}{N_q}$ in strata with at least one relevant tuple to the query and 0 otherwise; where N_i is the total number of tuples in the strata, and N_q is the total number of tuples in all relevant strata. Using the algebraic properties of variance, the confidence interval can be calculated as follows:

$$\pm\lambda \cdot \sqrt{\sum_{i=1}^{B} w_i^2 \cdot V_i(q)}$$

where $V_i(q)$ is $\frac{var(\phi(S_i))}{K_i}$. Stratified sampling is really powerful when the strata correlate with predicates the user may issue. The variance V_i within the strata might be much smaller than the variance globally.

2.2.3 Stratified Aggregation

Like we saw in the example in the introduction, partitioned aggregations can be used to approximate a query result. Suppose, as in stratified sampling, we first partition P into B

mutually exclusive partitions. But instead of sampling from these partitions $P_1, ..., P_B$, we compute the SUM, MAX, MIN, and COUNT for each partition [2]. This data structure, a collection of partitioned aggregates, has $\mathcal{O}(B)$ values. We can use this data structure to estimate query results for our desired subpopulation-aggregate queries.

For any predicate, there are three different sets of partitions:

- R_{cover} : it is known that every tuple in the partition satisfies the predicate
- $R_{partial}$: it is possible some tuple in the partition satisfies the predicate
- R_{none} : no tuple in the partition satisfies the predicate

Since each partition P_i has a $SUM(P_i)$, $MAX(P_i)$, $MIN(P_i)$, and $COUNT(P_i)$, can use these sets to estimate the maximum possible and minimum possible value the aggregate query of interest could take. For SUM and COUNT queries, this is easy due to their monotonic nature. We simply fully include the partial partitions in the upper bound, and omit them for the lower bound:

$$ub = \sum_{P_i \in R_{cover}} AGG(P_i) + \sum_{P \in R_{partial}} AGG(P_i)$$

$$lb = \sum_{P_i \in R_{cover}} AGG(P_i)$$

AVG queries are a little more complex to estimate since they are not monotonic. Let's define $MAX(R_{partial})$ to be the maximum of all of the max values of the partitions with partial overlap, and $MIN(R_{partial})$ to similarly be the minimum value. A bound for the AVG query is:

$$ub = \max\{\frac{\sum_{P_i \in R_{cover}} SUM(P_i)}{\sum_{P_i \in R_{cover}} COUNT(P)} \ , \ MAX(R_{partial})\}$$

$$lb = \min\{\frac{\sum_{P_i \in R_{cover}} SUM(P_i)}{\sum_{P_i \in R_{cover}} COUNT(P)} \ , \ MIN(R_{partial})\}$$

2. For technical reasons, we assume all a are positive (they can be shifted if not)

The average is at most the maximum of the average of fully covered partitions and the overall max of any potentially relevant partitions (and likewise for the lower bound). This scheme is fully deterministic and it is always guaranteed that the user's result lies within those confidence intervals.

We can characterize the estimation error as $ub - lb$, and notice that in all three queries the error is a function of $R_{partial}$. If a query predicate "aligns" with the partitioning (no partial overlaps), the query is answered exactly with 0 error. This property is not guaranteed with stratified sampling, which will always have sampling error in its estimates. However, the partial overlaps introduce ambiguity and error since we do not know how many relevant tuples match the predicate in those partitions. In those partial overlap cases, sampling is a far more accurate estimate because the deterministic bounds are very pessimistic.

2.2.4 Related Work

Uniform sampling, stratified sampling, and stratified aggregation have dominated the AQP literature dating back to the 1980s [105], and we refer the readers to recent taxonomy and critique of this work [25].

Optimizing Sampling. The pitfalls of uniform sampling are well-established, and several approaches have been proposed to optimize sampling [3, 24, 7, 12, 22, 23, 47, 38]. Almost all of this work relies on significant prior knowledge before query time. Either they leverage prior knowledge of a workload [7, 12, 3] or rely on auxiliary index structures [22, 47, 38]. The consequence is a substantial offline optimization component that takes at least one full pass through the dataset in these AQP systems. The proposal in this paper, PASS, is similar in that it constructs a synopsis data structure offline for accurate future query processing. While there are AQP settings where samples are constructed online or during query processing [70, 55, 76], we believe there are a large number of data warehousing use cases where expensive upfront creation costs can be tolerated. VerdictDB [110] is a recent AQP system that supports approximate query processing of general ad-hoc queries. It builds a new index *scramble* by drawing samples from the original data. Given a query it uses only the sampled items in the scramble to estimate the result. They achieve fast latency with error provable guarantees. PASS also uses samples to construct a data structure in the pre-processing phase, however, it combines aggregation and stratified sampling to build a tree-based index with very low space complexity to answer aggregation queries efficiently with minimum error. VerdictDB uses more space to answer queries with high accuracy, however, it can handle more types of queries, like equi-joins. In Section 2.5 we compare PASS with VerdictDB over different datasets.

Optimizing Aggregation. There are also similar studies of how to optimize "binned aggregates" [60, 74, 59]. In particular, there is a highly related concept to pass of V-Optimal histograms, which are histogram buckets placed in such a way to minimize the cumulative variance [60]. In contrast, PASS is designed for cases where the goal is to aggregate one column of data based on predicates on another set of columns. Accordingly, PASS constructs predicate partitions over the predicate columns to control the variance of the aggregation column. We further minimize the maximum variance (the worst-case error) unlike the V-Optimality condition. There are also multi-dimensional binned aggregation variants such as Lazaridis et al. [80] (essentially a data cube for approximate query processing). While Lazardis et al. do not contribute a variance optimization, they do organize their aggregates in a hierarchical structure like PASS.

Hybrid AQP. There are also a number of recent hybrid techniques that leverage pre-computed "full data" aggregates to make sampling-based AQP more reliable. For example, Galakatos et al. [44] cache previously computed results to augment previously constructed samples. In that way, they build an interactive scheme to handle ad-hoc queries efficiently. SampleClean materializes a full dataset aggregate over dirty data to mitigate sampling error [132]. AQP++ [113] precomputes a number of aggregate queries, determines query subsumption relationships to coarsely match a new query with one of those previously computed, and then, uses a uniform sample to approximate the gap. AQP++ runs a practical iterative hill-climbing heuristic to determine which aggregates to compute. We see AQP++ as the most similar proposal to PASS, but there are key differences in the two approaches. First, we propose an efficient dynamic programming algorithm with provable guarantees to find which aggregation queries (partitioning) to precompute so that the maximum error is minimized. We further organize these aggregates into a tree structure for efficient predicate evaluation. Second, instead of using uniform sampling to approximate the gap, we apply stratified sampling only on the strata that are partially intersected by the query. Our experimental results find that PASS is generally more accurate for the same sample size.

Mergeable Summaries and Partitioning. There is increasing discussion of data partitioning in AQP (outside of stratified sampling). Rong et al. define the PS^3 framework [122] to optimize sampling at a data partition level to avoid loading a large number of samples. We believe that the core tenets of the PS^3 framework are complementary to PASS and our optimization algorithm could be used as an inner routine in their framework. If our strata align with storage partitions, we could see similar benefits. We similarly see connections with Liang et al. who study constraint-based optimization for summarizing missing data [86]. Hierarchical aggregation is also related to the work on mergeable summaries, which are synopses that can be exactly combined at different levels of granularity [46, 4].

Learned AQP. There are also a number of techniques that leverage machine learning for AQP. Some of the initial work in using precomputation for AQP uses Maximum Likelihood Estimates to extrapolate results to unseen queries [61, 62]. There also are more comprehensive solutions that train from a past query workload [111] or directly build a probabilistic model of the entire database [56, 134]. There are also middle grounds that learn weights to direct stratified sampling [131].

2.3 Overview and Query Processing

These strengths and weaknesses of stratified sampling and stratified aggregation suggest an intriguing middle ground. In the simplest version, one can create stratified samples and annotate them with precomputed partition aggregates. To answer a supported query, we can skip all strata that are fully covered and only use the samples to estimate those partially covered strata leading to our contribution, PASS: Precomputation-Assisted Stratified Sampling.

2.3.1 Usage

PASS is a synopsis data structure used for answering aggregate queries over relational data. The user defines an *aggregation column* (numerical attribute to aggregate) and a *set of predicate columns* (columns over which filters will be applied). The system returns an optimized data structure that can answer SUM, COUNT, AVG, MIN, and MAX aggregates over the aggregation column filtered by the predicate columns.

```
SELECT SUM/COUNT/AVG/MIN/MAX(A)
FROM P
WHERE Predicate(C1,...,Cd)
```

Conceptually, this is the same problem setup as described in the previous section with a dataset of (c, a) tuples; where the aggregation column corresponds to a and the predicate columns correspond to c. A PASS data-structure is one-dimensional when there is a single predicate column and is multi-dimensional when there are a set of predicate columns.

The user specifies the following parameters: (τ_c) a time limit for constructing the data structure, and (τ_q) a time limit for querying the data structure. Then, using a cost-model, our framework minimizes the maximum query error while satisfying those constraints. Let $\mathcal{T}$ be the set of all PASS data structures that satisfy the above constraints, and Q be the set of all relevant queries to the user. We define the following optimization problem:

$$T^* = \arg\min_{T \in \mathcal{T}} \max_{q \in Q} error(q, T). \tag{2.5}$$

The details of this optimization problem are described in the next section, but for simplicity, we will only consider tree structures with a fixed fanout and "rectangular" partitioning conditions $x_i \leq C_i \leq y_i$ for $1 \leq i \leq d$.

2.3.2 Partition Trees and Samples

A preliminary concept to understanding PASS is an understanding of multi-resolution partitioning. A *partition* of a dataset P is a decomposition of P into disjoint parts $P_1, ..., P_B$. Each P_i has an associated partitioning condition ψ_i, a predicate that when applied to the full dataset as a filter retrieves the full partition. This definition is recursive as each partition is itself another dataset. Partitions can be further subdivided into even more partitions, which can then be subdivided further. This type of recursive subdivision leads to the definition of a *partition tree.*

Definition 2.3.1 (Partition Tree)**.** Let $\{P_i\}_1^B$ be subsets of a dataset P. A partition tree is a tree with B nodes (one corresponding to each subset) with the following invariants: (1) every child is contained in its parent, (2) all siblings are disjoint, and (3) the union of the siblings equals the parent.

Given this definition, the root of this tree is necessarily the full dataset, which we can think of as a degenerate partitioning with the condition $\psi = True$. Siblings' conditions can be combined together with a disjunction to derive the parent, and children can be derived with a conjunction with the parent's condition. Thus, each layer of the tree completely spans the entire dataset, but is subdivided at finer and finer granularities.

For a target query predicate q and a corresponding subset of tuples that satisfy q denoted by $P(q)$, we can define the *coverage frontier* or a minimal set of partitioning conditions that fully covers a query. Let $\{P_i\}_{i=1}^B$ be nodes in a partition tree. A subset $P_1, ..., P_l$ of these nodes *covers* a predicate q if $P(q) \subset \bigcup_{i=1}^l P_i$. A covering subset is minimal if it is the smallest subset (in terms of the number of partitions) that covers q.

The nature of the invariants, where disjoint children completely span their parents, described above allows us to find such a subset of nodes efficiently. Consider the following recursive algorithm:

Algorithm 1: Minimal Coverage Frontier Algorithm

```
1 MCF($P_i, q$):
2     if $P_i \subseteq P(q)$ or $P_i$ is a leaf: return $\{P_i\}$
3     if $P_i \cap P(q) = \emptyset$: return $\{\}$
4     $\gamma = \{\}$
5     for all children $P_i'$ of $P_i$: $\gamma = \gamma \cup$ MCF($P_i', q$)
6     return $\gamma$
```

Note that there are two types of nodes returned by the MCF algorithm above. Either we return *leaf* nodes or we return nodes that are fully contained by the query predicate. These two types of nodes exactly correspond to the two scenarios we described in Section 2.2.3: partial coverage and total coverage. The leaf nodes correspond to the partial overlap case.

Such a data structure gives us a practical algorithm to scale up stratified aggregation to a large number of nodes. Instead of a tuple-wise containment test, the base case in line 2 can be evaluated from the partitioning conditions ψ_i and a description of the query predicate. Normally, for stratified aggregation, we would have to test each of the B partitions. However, a tree facilitates faster evaluation time for selective queries.

Suppose, we have a partition tree of B nodes where every parent has a fixed number of children. Let q be a query that overlaps with γ of the leaf nodes in the partition tree. In the worst case, for computing MCF we need to visit $O(\gamma)$ nodes in each level of the partition tree. In this setting, if the partition tree is balanced the time-complexity for computing the MCF is $\mathcal{O}(\gamma \log B)$. This result can be shown by noting that the number of overlaps with leaves bounds the number of relevant nodes in any of the layers (due to the invariants) and there are $\log B$ such layers. For selective queries, this approach is far more efficient than a linear search through all partitions.

For each of the partitions in the partition tree, we compute four aggregate statistics over all the tuples within the partition: SUM, COUNT, MIN, MAX. Note, the AVG value is implicitly calculated with SUM and COUNT. This data structure forms the backbone of PASS: a partition tree annotated with aggregate statistics (as seen in the top half of Figure

2.2). The MCF algorithm returns those partitions that are completely covered, within which we can directly leverage the pre-computed aggregate, and those that are partially covered where we will need a more sophisticated estimation scheme.

It is natural to compare PASS to existing data skipping frameworks that skip irrelevant partitions of data (e.g., [125]). However, it is worth noting that PASS further skips partitions that are completely covered by a query predicate. This is due to the composable structure of SUM/COUNT/AVG/MIN/MAX aggregate queries supported, where they can be safely computed from partial aggregates.

Sampling: The challenge with partial coverage is that we do not exactly know the selectivity of a query within the partition. Thus, it makes sense to leverage sampling for this estimation. Due to the partitioning invariants, partially covered nodes will only be leaf nodes and all retrieved nodes are disjoint. We associate with each of the leaf nodes a uniform sample of tuples *within that partition.* This per-partition sampling plan differs from other proposals such as [113], which use uniform sampling. The entire structure is summarized in Figure 2.2, where the partition tree of B nodes lies above and stratified samples associated with each of leaf nodes lie below.

2.3.3 Query Processing

PASS leads to the following query processing algorithm. We present SUM, COUNT, AVG for brevity, but it is also possible to get estimations for MIN and MAX.

Index Lookup. Apply the MCF algorithm above to retrieve two sets of partitions: R_{cover} and $R_{partial}$.

Partial Aggregation. For each partition in R_{cover}, we can compute an exact "partial aggregate" for the tuples in those partitions. For a SUM/COUNT query q: $agg = \sum_{P_i \in R_{cover}} P_i(q)$, for an AVG query, we weight the average by the relative size of the partition: $agg = \sum_{P_i \in R_{cover}} P_i(q) \frac{N_i}{N_q}$, where N_i is the size of the partition P_i and N_q is the total size in all relevant partitions of query q.

Sample Estimation. Each partition in $R_{partial}$ is a leaf node with an associated stratified sample. We use the formula in Section 2.2.2 to estimate their contribution. Let S_i denote the sample associated with partition $P_i \in R_{partial}$: $samp = \sum_{P_i \in R_{partial}} f(S_i) \cdot w_i$

Results. The results can be found by taking a sum of the two parts: $result = samp + agg$.

Confidence Intervals. Since the *agg* result is fully deterministic, the only part that needs uncertainty quanitification is *samp*. We can use the formula in Section 2.2.2 to compute this result:

$$\pm\lambda \cdot \sqrt{\sum_{P_i \in R_{partial}} w_i^2 \cdot V_i(q)}$$

where $V_i(q)$ is $\frac{var(\phi(S_i))}{K_i}$ where K_i is the sample size of the sample associated with partition P_i.

Hard Bounds. The data structure allows us to compute deterministic hard bounds on query results using formulas in Sec. 2.2.3.

2.3.4 Other Optimizations

To summarize, PASS associates a hierarchy of aggregates with stratified samples. This allows us to create sampling plans with a large number of strata and efficiently skip irrelevant ones. The hierarchy further allows us to bound estimates deterministically since the extrema of each stratum are known. The design of PASS allows for a few important optimizations that can significantly improve performance in special cases. First, the data structure has an important special case where we can skip processing the samples even when there is a partial overlap.

0 Variance Rule: For AVG queries, we can add an additional base case to the MCF algorithm: *if the node in question has 0 variance (i.e., the min value is equal to the max value), return the current node.* When answering AVG queries, 0 variance nodes (where all numerical values are the same) are equivalent to covered nodes.

Next, the data structure can also effectively compress the samples using *delta encoding*. Every sampled tuple can be expressed as a *delta* from its partition average. Ideally, the variance within a partition would be smaller than the variance over the whole dataset.

2.4 Optimizing the Partitioning

This section describes how we optimize PASS to meet the desired error rate.

2.4.1 Objective and Search Space

The first step is to process the user-specified time constraints τ_c and τ_q into internal parameters. As before, we consider a dataset P, where there is an aggregation column A, and a collection of predicate columns $(C_1, .., C_d)$. We calculate the maximum number of leaf nodes k (which governs the construction time as we show later) allowable in the time limit τ_c. Each leaf will define rectangular partitioning condition $x_i \leq C_i \leq y_i$ for $1 \leq i \leq d$. Then, we calculate the maximum number of samples allowable in the time limit τ_q. The search space $\mathcal{T}$ is all PASS data structures with k leaves with a fixed fanout of 2^d.

Next, we have to define a class of queries Q that we care about. While in general SQL predicates can be arbitrary, we restrict ourselves to a large class of "sensible" predicates. Over this schema, we define Q to be AVG/SUM/COUNT queries in a "rectangular region", which returns the average value with respect to the attribute A among all tuples with $x_i \leq C_i \leq y_i$ for $1 \leq i \leq d$. Our optimization framework can also support other definitions for Q (hereafter called templates), but for brevity, we will focus on rectangular averages.

Given this definition of Q and $\mathcal{T}$, now we describe the optimization objective. Let $T = \{b_1, \ldots, b_k\}$ be the set of leaf nodes. For a query $q \in Q$ let T_q be the set of leaf nodes that partially intersect with the query predicate, we get an error formula as follows:

$$error(q,T) = \lambda \sqrt{\sum_{b_i \in T_q} w_i^2 \cdot V_i(q)}.$$

Note from the previous section, the structure of the leaf nodes governs the estimation error of the data structure. The shape of the tree (height and fanout) only affects construction time and query latency. Thus, it is sufficient to optimize PASS in two steps to control for worst-case query error: first, choose an optimal partitioning of the leaf nodes, and then construct the full tree with a bottom-up aggregation. The core of the algorithm is then to optimize the following "flat" partitioning:

$$R^* = \arg\min_{T \in \mathcal{T}_{leaves}} \max_{q \in Q} error(q,T),$$

where $\mathcal{T}_{leaves}$ is the family of all possible k leaf nodes.

2.4.2 Partitioning Algorithm Intuition

It is worth noting that we do not have to search over all possible rectangular partitions and queries. Consider the 1D case (where the rectangles are simply intervals): all meaningful rectangular conditions will have predicate intervals defined by the attribute values of the tuples c_i (any others are equivalent), and thus, there are $\binom{N}{k-1} = \mathcal{O}(N^{k-1})$ possible partitions and $\mathcal{O}(N^2)$ possible query intervals.

Thus, we make the following simplifying but practical assumptions. Note that the error in a query result is governed by the variance of items where there is partial overlap. So, we approximate the problem to the following search condition: control the variance of every query's single worst partial overlap. Next, we further assume that when there is a partial overlap, this overlap is non-trivial where at least δN tuples are relevant to the query (avoids degenerate results that could result in empty partitions).

Even in the 1D case, we want to avoid exhaustive enumeration of all the $\mathcal{O}(N^{k-1})$ partitions. In addition, we want to avoid precomputing and storing the results of all possible $\mathcal{O}(N^2)$ interval queries due to the quadratic space dependency; if we could simply execute all $\mathcal{O}(N^2)$ queries, there is no need for a synopsis structure in the first place.

Technical Details

The following contains technical details about the assumptions that can be skipped for brevity. Let $b_i \in R$ be a partition of partitioning R (notice that the set of leaves T corresponds to a partitioning R). Let N_i be the number of items in partition (i.e. leaf) b_i. Furthermore, let R_q be the partitions of partitioning R that intersect the query q (either fully or partially). Finally, let $P_i(q) = P \cap b_i \cap q$, be the set of items in bucket i that are contained in query q, as we had in the previous section. We assume that the valid queries of Q with respect to a partitioning R, are the queries that intersect sufficiently many items in each partition they intersect. Precisely, for a partition $b_i \in R_q$, if $N_{i,q} = |P_i(q)|$ is the number of items in b_i that are contained in q, then we assume that $N_{i,q} \geq \delta N$.

Basically, the assumption states that we only care about queries that meaningfully overlap when they do partially overlap. Accordingly, we can define the set of "meaningful" queries with respect to a partitioning R as $Q' = \{q \in Q \mid N_{i,q} \geq \delta N, \forall b_i \in R_q\}$. One can think of this set as rectangles whose boundaries are grounded with actual tuples in the dataset.

From Section 4.2 we can define the "single-partition" b_i variance of AVG queries as $V_i(q) = \frac{1}{N_i} \cdot \frac{1}{N_{i,q}^2} \left[N_i \sum_{h \in P_i(q)} t_h^2 - \left(\sum_{h \in P_i(q)} t_h\right)^2\right]$, while for SUM queries $V_i(q) = \frac{1}{N_i} \cdot \left[N_i \sum_{h \in P_i(q)} t_h^2 - \left(\sum_{h \in P_i(q)} t_h\right)^2\right]$. For COUNT queries the formula for $V_i(q)$ is identical to the formula for SUM queries with $t_h = 1$ or $t_h = 0$.

Based on these formulas and the above assumptions, we approximate Equation 2.5 with a simpler problem. The high-level idea is that we focus on the problem of minimizing the maximum variance of queries that are fully contained in only one partition. We show that solving this simpler problem leads to efficient approximations for our original problem, where queries intersect multiple partitions.

Let $Q_R^1 \subseteq Q$ be the set of meaningful queries in Q that intersect only one partition in partitioning R, and let i_q be that partition. We define a new problem, as: $R' = \arg\min_{R\in\mathcal{R}} \max_{q\in Q_R^1} V_{i_q}(q)$, where $\mathcal{R}$ is the family of all possible valid partitionings.

Lemma 1. It holds that

$$\max_{q\in Q} error(q, R') \leq \sqrt{k} \min_{R\in\mathcal{R}} \max_{q\in Q} error(q, R),$$

for SUM and COUNT queries and

$$\max_{q\in Q} error(q, R') = \min_{R\in\mathcal{R}} \max_{q\in Q} error(q, R),$$

for AVG queries.

From Lemma 1 it follows that any α-approximation algorithm for the newly defined problem is also a α-approximation for our original problem for AVG queries and a $\sqrt{k} \cdot \alpha$-approximation for SUM and COUNT queries. Notice that it is much easier to handle the newly defined problem since we can find the query with the maximum error with respect to partitioning R by looking only on queries that are fully contained in partitions $b \in R$.

Finally, we note that the approximation ratio for SUM and COUNT queries in Lemma 1 is stated in the worst case. In particular, if we guarantee that a query can partially intersect at most $\mathcal{K}$ partitions then the approximation factor is $\sqrt{\mathcal{K}}$ instead of $\sqrt{k}$. For example, in 1D a query interval can partially intersect at most 2 partitions, so the approximation factor is $\sqrt{2}$.

2.4.3 Algorithm in 1D

Considering the variance function for COUNT, we can show that the optimum partitioning for COUNT queries in 1D consists of equal size partitions and hence we can construct it in linear time. Next, we mostly focus on SUM and AVG queries.

First, we consider the 1-d case where we have a collection of tuples $P = \{(c_i, a_i)\}_{i=1}^{N}$. We have a defined query type, i.e., SUM, AVG, and we want to find the partitioning that minimizes the maximum estimation error for that query type. In this first case, we start by developing a strawman algorithm: one where we essentially enumerate all possible 1d aggregate queries over the full data.

To do so, we first sort the tuples with respect to the predicate values c_i. Then we define the dynamic programming table A of N rows and k columns, where $A[i,j]$ is the optimal solution among the first i data items (with respect to the predicate values) with at most j partitions. Let $\mathcal{M}$ be function that takes as input an interval $[i_1, i_2]$ and returns the maximum variance of a query $q \in Q$ that lie completely inside $[i_1, i_2]$.

```
$\mathcal{M}(i_1, i_2)$ :
1. $\mu = 0$
2. For all meaningful subintervals $[g, w] \subset [i_1, i_2]$:
    (a) Let $\phi$ be appropriately defined for the query type and the predicate $g \le C \le w$.
    (b) $\mu = \max\{\mu, var(\phi(P \cap [g, w]))\}$
3. return $\mu$
```

Using the function $\mathcal{M}$ we can define the recursion

$$A[i,j] = \min_{h<i} \max\{A[h, j-1], \mathcal{M}([h+1, i])\}.$$

Notice that we can easily solve the base cases $A[i,1]$, $A[1,j]$ using the $\mathcal{M}$ function. In an efficient implementation of $\mathcal{M}$ the subquery variances are computed with pre-computed prefix sums. Since $\mathcal{M}$ function considers $\mathcal{O}(|Q|)$ queries ($\mathcal{O}(N^2)$ assuming all possible queries). Hence, the total running time of the basic DP algorithm is $\mathcal{O}(kN^2|Q|) = \mathcal{O}(kN^4)$. This algorithm achieves an optimal partitioning for AVG queries and $\sqrt{2}$ approximation for SUM queries.

Faster Algorithm With Monotonicity. The first insight ignores the query enumeration problem but focuses on the structure of the DP itself. Suppose, we have a query q completely inside a partition b_x and let b_y be another partition such that $b_x \subseteq b_y$. Then, we can show that $V_x(q) \leq V_y(q)$. This statement is very intuitive: adding irrelevant data to a query can only make the estimate worse.

The exact proof of this statement depends on the type of the query. For example for the AVG query, we have that $P_x(q) = P_y(q)$, $N_{x,q} = N_{y,q}$, and $N_x < N_y$. Hence,

$$V_x(q) = \frac{1}{N_{x,q}^2}\left[\sum_{h \in P_x(q)} t_h^2 - \frac{\left(\sum_{h \in P_x(q)} t_h\right)^2}{N_x}\right] \leq \frac{1}{N_{y,q}^2}\left[\sum_{h \in P_y(q)} t_h^2 - \frac{\left(\sum_{h \in P_y(q)} t_h\right)^2}{N_y}\right] = V_y(q).$$

Similar arithmetic shows that the same statement holds for SUM and COUNT queries (with a caveat in the next section).

Then, we can argue that for $h_1 \leq h_2$ it holds that $A[h_1, j-1] \leq A[h_2, j-1]$ and $\mathcal{M}([h_1+1, i]) \geq \mathcal{M}(h_2+1, i)$. The second inequality holds because of our last observation. The first inequality holds because we can use the partition of $A[h_2, j-1]$ in the first h_1 items as a valid partition, so $A[h_1, j-1]$ can only be smaller. Because of this property, a binary search over the values of h can return the value $\hat{h}$ such that $\max\{A[\hat{h}, j-1], \mathcal{M}([\hat{h}+1, i])\}$ is minimized. The running time of this DP algorithm is $\mathcal{O}(kN^3 \log(N))$.

Faster Approximate Dynamic Program

Next, we want to avoid having to evaluate every possible query exactly during construction. In this step, we can leverage a uniform sample of data to estimate the query variance; we can take a uniform sample of m tuples $P = \{c_i, a_i\}_{i=1}^m$ to perform the optimization. This approximation would create partitioning rules, which we could then resample from to construct our stratified sampling. With sampling, the complexity of the optimization algorithm would be $\mathcal{O}(kN^2 m \log(m))$. There is an interplay between sampling and the proofs above, where we require that every query interval receives roughly the same fraction of samples. Sampling avoids the circular logic in our strawman algorithm, and the optimal partitions can be found with far less computation than the exact evaluation of every possible query over the original dataset.

Next, instead of considering all possible query intervals in a partition, we consider a subset of such intervals to improve the running time. For SUM (and COUNT) queries, given an interval $[i_1, i_2]$ that contains m' sampled items, we consider picking only $\mathcal{O}(1)$ items $L \subseteq P \cap [i_1, i_2]$ such that there exists an item $x \in L$ where the intervals $[i_1, x]$ and $[x, i_2]$ contain the same number of samples. Then we consider only the interval queries whose endpoints are defined by the selected items L. Hence, the number of query intervals we consider in a partition is still $\mathcal{O}(1)$. This change affects line (2) in the algorithm above. Instead of considering all meaningful subintervals $[g, w] \subset [i_1, i_2]$, we consider only the intervals defined by the items in L leading to a new running time of $\mathcal{O}(km \log m)$. We show that the maximum variance we get by checking only this subset of queries is not smaller than the maximum variance over all meaningful queries in $[i_1, i_2]$ divided by 4. Our new algorithm finds a partitioning where the maximum error of a SUM (or COUNT) query is at most $2\sqrt{2}$ times the maximum error of the optimum partitioning. For AVG queries, we show that the query with the maximum variance in a partition has length at most $2\delta m$. We precompute the variance of all possible length δm queries (there are only $\mathcal{O}(m)$ of them) and store them in a binary search tree. Given an interval $[i_1, i_2]$ we can return the length δm query with the maximum variance in $\mathcal{O}(\log m)$ time. Overall the algorithm runs in $\mathcal{O}(km \log^2 m)$ time and finds a partitioning where the maximum error of an AVG query is at most 2 times the maximum error of the optimum partition.

To summarize, let N be the total size of the dataset, k be the desired number of partitions, m be the number of samples [3]:

3. ** Indicates the approximation algorithm used in the experiments

Naive DP	$\mathcal{O}(kN^4)$
Faster DP	$\mathcal{O}(kN^3 \log N)$
Approx Sampling	$\mathcal{O}(kN^2 m \log m)$
Sampling + Discretization (**)	$\mathcal{O}(km \log m)$

2.4.4 Algorithm in Higher Dimensions

While the DP algorithm gives an optimum partitioning in a single dimension, it is not clear how to extend this to multiple dimensions.

For higher dimensions, we have to consider a space of partition trees that each layer defines rectangular partitioning. Such a space is well-parameterized by the class of balanced k-d trees. A k-d tree is a binary tree in which leaf nodes represent d-dimensional points. Every non-leaf node can be thought of as a partitioning plane that divides the parent space into two parts with the same number of items. Points to the left of this hyperplane are represented by the left subtree of that node and points to the right of the hyperplane are represented by the right subtree. We note that we can also design k-d trees with fanout 2^d by splitting a node over all dimensions simultaneously.

1. Construct a balanced k-d tree U over $\{c_i\}_{i=1}^N$.
2. Start with an empty tree U' initialize as the root of U.
3. While the number of leaf nodes is less than k:
   (a) For all new leaf nodes v in U':
      i. Apply $\mathcal{M}$ on the items in v.
   (b) For the leaf node that contained the query with the maximum variance, add its children (from the corresponding node in U) to U'.

The precomputation time is $\mathcal{O}(N \log N)$ to construct U. After the precomputation, the algorithm runs in $\mathcal{O}(kN^{1-1/d}|Q|)$ time. The tree U' we return has $\mathcal{O}(k)$ space and it gives an optimum partition with at most k leaf nodes with respect to the k-d tree U. Hence, U' is optimum with respect to AVG queries and has a $\sqrt{k}$-approximation for SUM and COUNT queries.

A naive way to find the leaf that contains the query with the maximum variance is to consider all possible rectangular queries in the leaf, $\mathcal{O}(|Q|) = \mathcal{O}(N^{2d})$.

2.4.5 Summary

To summarize our analysis, we propose an optimization framework that returns partitions that control the maximum query error over a workload of hypothetical queries. The key approximation parameters in this framework avoid having to enumerate all possible queries. Our analysis provides a synopsis data structure and an approximation framework that has the following parameters. We summarize the effects when these parameters are increased:

Knob	**Effect**	**Tradeoff**
Sample Size K	+ Accuracy	+ Query Latency
Partitions k [4]	+ Accuracy; - Query Latency	+ Init Time; + Update Cost
Apx factors m, L	- Worst-Case Error	+ Init Time

In our experiments, we control K and k across all baselines. We ensure that the query latency of the queries is roughly the same by budgeting the same amount of precomputation and samples.

Extensions PASS can be extended to handle multiple predicates, group-by's, and categorical queries. To handle multiple predicate column sets, we construct different trees based on statistics from the workload (see notes on statistics from Facebook [7]). In the full version, we demonstrated the scenario of 'workload-shifting' in which PASS can use a synopsis that is built for one query template to solve other query templates that share one or more attributes.

4. Notice that B is related to the number of leaves k.

Furthermore, by applying any dictionary encoding we can handle queries over categorical variables. Finally, PASS can handle group-bys over categorical columns, i.e. each group-by condition can be rewritten as an equality predicate condition. Then we can aggregate answers for all the selection queries to generate a final answer.

Dynamic updates PASS can easily handle new insertions (or deletions) while maintaining the statistical consistency of the estimates for COUNT, SUM, and AVG queries. In particular, we can maintain samples using Reservoir sampling [130]. Each time that a new item t_i is inserted, Reservoir sampling might choose to replace a sample t_j with t_i. Assume that t_j belongs in partition P_j and t_i belongs in partition P_i. We remove t_j from P_j and we insert t_i in P_i. Furthermore, we update all the statistics in the nodes from the leaf P_i to the root and from the leaf P_j to the root of the partition tree. In each node, we can update the statistics in $\mathcal{O}(1)$ time so the total update time depends on the height of the tree (for $d = 1$ we have $\mathcal{O}(\log k)$). However, if there are enough updates to the structure, re-optimization of the partitioning may be needed. In that case Split and Merge technique [40, 49] might help to get efficient update time. We leave this part as an interesting future problem.

2.5 Experiments

We evaluate PASS on a number of different datasets and workloads. We run our experiments on a Linux machine with an Intel Core i7-8700 3.20GHz CPU and 16G RAM.

2.5.1 Experiment Setup

We follow the problem setup described in Section 3.1. Given an aggregation column and a set of predicate columns, we construct a pass data structure with a specified construction time (number of leaf nodes) and query time (sampling rate).

Datasets

Intel Wireless Dataset: The Intel wireless dataset [115] is an Internet of Things dataset with data from 54 sensors deployed in the Intel Berkeley Research lab in 2004. It contains 3 million rows, 8 columns including humidity, temperature, light, voltage as well as date and time that are measured by different sensors. In our experiments, we use the *time* column for predicates and the *light* column for aggregation.

Instacart Online Grocery Shopping Dataset 2017: The Instacart Online Grocery Shopping dataset 2017 [58] is released by the grocery delivery service Instacart. We use the *order_product* table of 1.4 million entries. Each entry has 4 columns: the *order_id*, *product_id*, *add_to_cart_order* and *reordered*. We use the *product_id* column for predicate and *reordered* column for aggregation.

New York City Taxi Trip Records Dataset: The New York City Taxi Trip Records dataset [129] is published by the NYC Taxi and Limousine Commission (TLC). The dataset contains the yellow and green taxi trip records including fields capturing pick-up and drop-off dates/times, pick-up and drop-off locations, trip distances, itemized fares, rate types, payment types, and driver-reported passenger counts. In our experiments, we use the 7.7 million records collected in January 2019, and unless otherwise specified, we use the *pickup_datetime* column for predicate and the *trip_distance* column for aggregation.

Metrics

Our primary metric is **relative error** which is the difference of estimated query result and the ground truth divided by the ground truth–for fixed sample size and precomputation budget. In all the experiments, we evaluate the median relative error over randomly selected queries. We also measure the **confidence interval ratio** (CI Ratio) which is the ratio between the half of estimated confidence interval and the ground truth. This quantifies the accuracy of the confidence intervals found with each framework. Since one advantage of PASS is that it enables aggressive and reliable data skipping, we measure the **skip rate**, which is the ratio of the tuples that are safely skipped during query processing.

Baselines

Every baseline gets a sampling budget of K and a query precomputation budget of B.

- **Uniform Sampling (US)** Sample K records from the database uniformly at random.
- **Stratified Sampling (ST)** Create B strata, and uniformly sample $\frac{K}{B}$ records from each one. *We use equal depth partitioning to construct the strata.*
- **AQP++**[113]. We implemented the hill-climbing algorithm described in the AQP++ paper. For the 1-D experiments, instead of using a BP-cube, we partition the dataset with the hill-climbing algorithm then pre-compute aggregations on the partitions to combine with the sampling results. For multi-dimensional experiments, we construct a KD-Tree which we describe in detail in Section 2.5.4.

Unless further specified, we use a sample rate of 0.5%, λ=2.576 for a 99% confidence interval and a precomputation budget of 64 queries. The sample size is much larger than the precomputation size. Thus, the sample size is a good proxy for query latency.

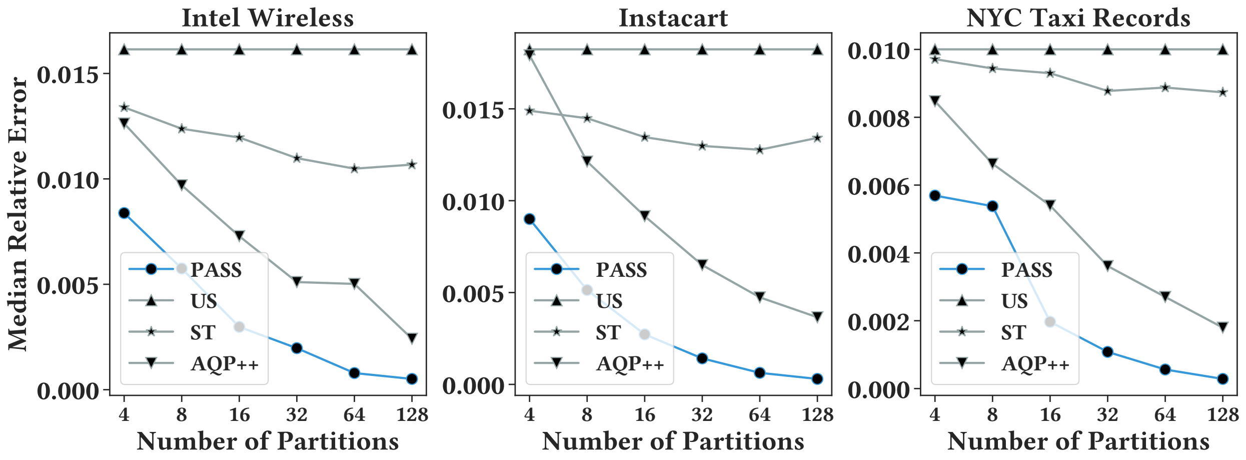


Figure 2.3: Median relative error of 2000 random SUM queries on the 3 real-life datasets using a varying number of partitions and a fixed sample rate of 0.5%.

————		COUNT			SUM			AVG		
Approach	Mean Cost	Intel	Insta	NYC	Intel	Insta	NYC	Intel	Insta	NYC
US	0.09s	0.94%	1.20%	0.50%	1.61%	1.82%	1.0%	1.21%	1.25%	0.87%
ST	0.35s	0.16%	0.22%	0.08	1.0%	1.27%	0.8%	1.0%	1.22%	0.89%
AQP++	0.8s	0.33%	0.37%	0.16%	0.5%	0.47%	0.2%	0.4%	0.31%	0.22%
PASS-ESS	23s	0.03%	0.038%	0.02%	0.05%	0.07%	0.044%	0.04%	0.057%	0.04%
PASS-BSS2x	23s	0.12%	0.17%	0.07%	0.23%	0.3%	0.16%	0.2%	0.23%	0.15%
PASS-BSS10x	23s	0.06%	0.06%	0.02%	0.1%	0.11%	0.07%	0.08%	0.09%	0.07%

Table 2.1: Controlling for worst-case query latency (total number of tuples processed), we demonstrate that it is possible to construct a synoposis that is highly accurate (less than .1% error) across 2000 random SUM/COUNT/AVG queries. The caveat is a high upfront optimal partitioning cost.

Comparing Baselines

In AQP, one usually evaluates accuracy for a fixed number of sample data points processed. Since PASS couples result estimation with data skipping, controlling for the exact sample size is less intuitive. The most straightforward way to compare sampling rates across techniques is to use the effective sampling size (**ESS**) (average number of data points processed per query divided by total data points). ESS is a good metric when the main objective is to control for query latency because it basically measures the IO cost of answering an aggregate query. However, solely comparing techniques w.r.t ESS can be misleading if one is concerned about the size of the synopsis structure. Data skipping could allow one to include more samples into the synopsis if not all of them are likely to be used for any given query. Thus, we additionally include a bounded sampling size (**BSS**) comparison where techniques are restricted to a maximum number of samples.

2.5.2 Accuracy Evaluation

Table 2.1 illustrates the key premise of the paper: with PASS, a user pays an upfront cost for increased accuracy over future queries. As described in the previous section, we include both a comparison in terms of ESS and BSS variants of PASS. In the ESS case, for three datasets and randomly generated queries, a PASS synopsis (0.5% sampling rate and 64 partitions) achieves less than a 0.1% median relative error. PASS is more accurate than the baselines across datasets, but does require a larger upfront optimization cost. However, ESS is an optimistic setting in certain environments with memory constraints. Thus, we additionally present the BSS versions of PASS. In Table 2.1, we include PASS-BSS2x and PASS-BSS10x which are bounded to 2 times and 10 times of the online storage of uniform sampling. Due to the data skipping, PASS-BSS2x evaluates results about 13% faster than US/ST. Even with bounded storage, they still outperform other baselines significantly in terms of accuracy. For the rest of the experiments, we will focus on the ESS setting unless explicitly mentioned as it is the most intuitive.

As a Function of Precomputation

This construction cost is controlled by the number of partitions, which is also the amount of space allocated for aggregate precomputation. Figure 2.3 illustrates the accuracy on these three datasets for a fixed sample size of 0.5% and a varying degree of partitions (or strata in stratified sampling). As the number of partitions decreases the benefits of PASS also decrease. *PASS gives the user a new axis for control in AQP, where she can not only trade-off query latency but also data structure construction time for additional accuracy.*

As a Function of Sample Size

To better understand how each baseline performs, for each dataset, we first fix the partition size to 64 and vary the sample rate from 10% to 100%. Figure 2.4 shows the median relative

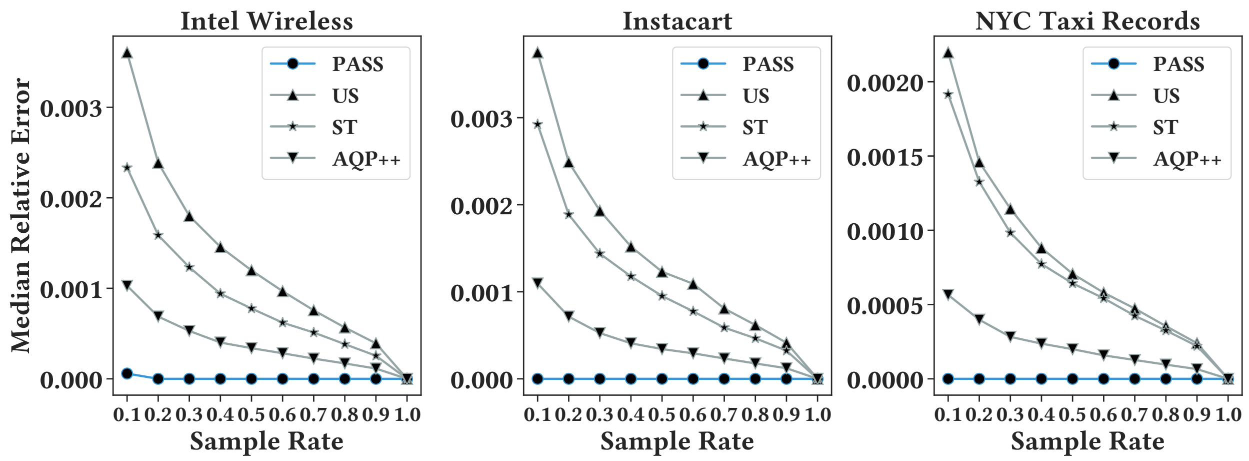


Figure 2.4: Median relative error of 2000 random SUM queries on the 3 real-life datasets using a varying sample rate and a fixed number of partitions of 64.

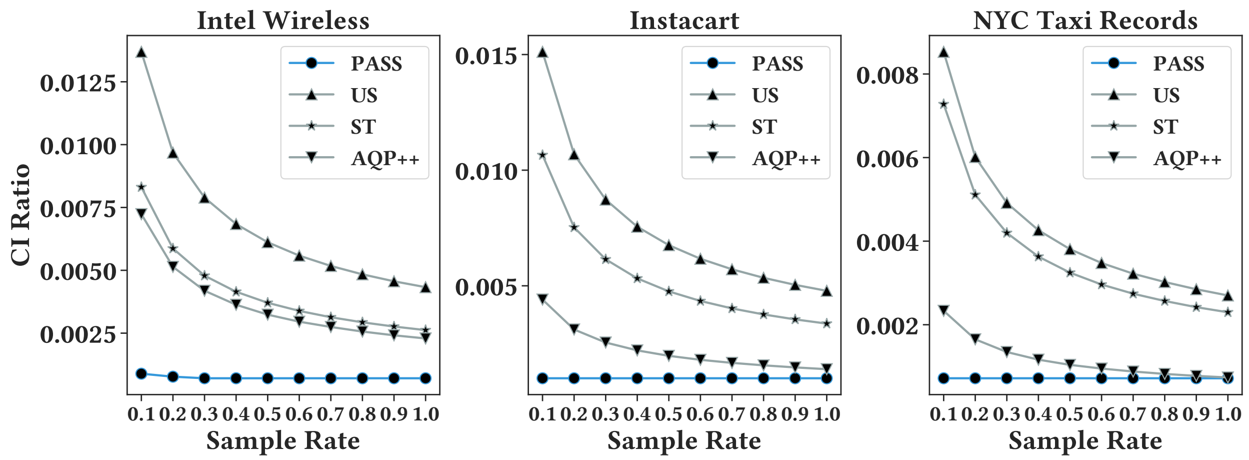


Figure 2.5: Median confidence interval ratio of 2000 random SUM queries on the 3 real-life datasets using a varying sample rate and a fixed number of partitions of 64.

error for 2000 random SUM queries on the Intel Wireless dataset, Instacart dataset, and the New York City Taxi Trip Records dataset. PASS outperforms other baselines starting with a 10% sample rate. PASS not only returns an accurate result it also accurately quantifies this result with a confidence interval. Figure 2.5 shows the median confidence interval ratio on the three real-life datasets under the same experimental setting. *PASS is a reliable alternative to pure sampling-based synopses when expensive upfront optimization times can be tolerated.*

2.5.3 Approximated Dynamic Programming Partitioning vs. Equal Partition

In this experiment, we use different partitioning algorithms to partition the dataset, then we build a balanced binary tree bottom-up as our partition tree. The partition is later used as the strata for stratified sampling and is combined with the partition tree to solve a query as described in previously in Section 2.3. We evaluate the approximated dynamic programming partitioning algorithm (ADP) and the equal partitioning (or equal depth, equal frequency) algorithm (EQ). We found our implementation of the hill-climbing algorithm performs very similar to the equal partitioning algorithm, so it is omitted in this experiment.

We construct a synthetic adversarial dataset of 1 million tuples and 2 attributes. The predicate attribute contains 1 million unique values. The first 875K tuples have 0 as the value of their aggregate attribute and the last 125K tuples are generated by a normal distribution. The left plot of Figure 2.6 shows the result on 2000 random queries generated on the entire dataset and the right plot shows the result on 2000 random queries generated on the last 125K tuples. The results show that our approximated dynamic programming partitioning algorithm outperforms the EQ on the challenging queries and performs similarly on the trivial random queries.

Similarly, we evaluate the 3 real-life datasets by two sets of queries. For each dataset, we first randomly generate 2000 queries, then we randomly generate another 2000 challenging queries from the interval with the maximum variance identified using the fast discretization method we discussed in Section 2.4.3. Figure 2.7 shows the median CI ratio on the challenging queries generated on the Intel Wireless dataset, Instacart dataset, and the New York City Taxi Records dataset. The results suggest that in most cases, ADP outperforms EQ on challenging queries.

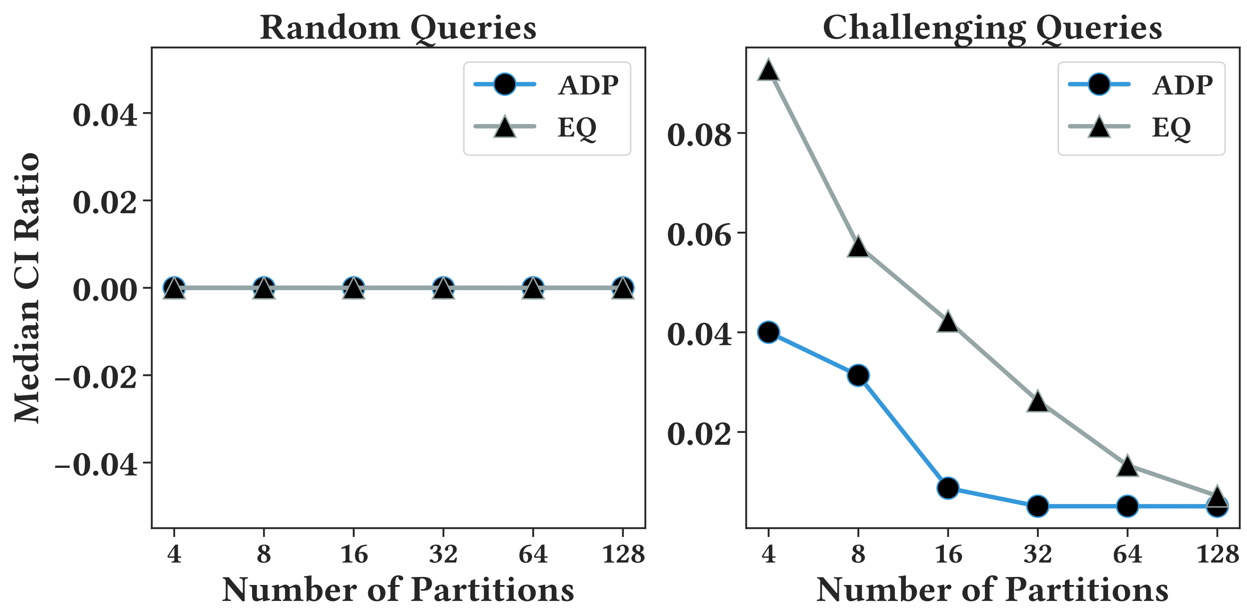


Figure 2.6: Median confidence interval ratio of Approximated Dynamic Programing partitioning (ADP) vs. Equal Partitioning (EQ) on a synthetic adversarial dataset.

2.5.4 Multidimensional Query Templates

In this section, we evaluate the performance of PASS on multi-dimensional queries on the NYC Taxi dataset. Using the trip_distance attribute as the aggregate attribute and pickup_time, pickup_date, PULocationID, dropoff_date, dropoff_time as the predicates attributes, we build 5 query templates of different dimensions where the i_{th} template uses the first i attribute(s) as predicate attribute(s).

The PASS variation used in this experiment is called KD-PASS. As described in Section 2.4.4, we build a KD-Tree that uses the fast discretization method to select the leaf node with the maximum variance for expansion until we reach the maximum leaf count of 1024. Also to make sure the tree is relatively balanced we limit the difference of the depth of leaf nodes to be no more than 2. At each expansion of a node, we find the median of each attribute so the fan-out factor is 2^d. The leaf nodes of the KD-Tree forms a partition of the dataset which is used by ST for sampling and data skipping.

The baseline in this experiment is called KD-US. KD-US also uses a KD-Tree that always expands the node with the smallest depth and breaks tie randomly until we reach the

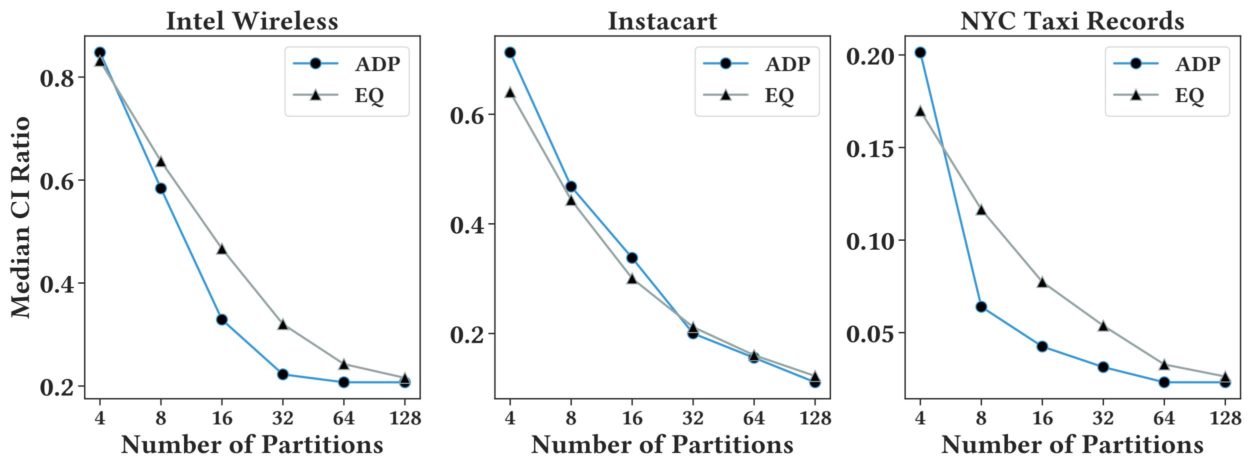


Figure 2.7: Median confidence interval ratio of ADP vs. EQ on challenging queries of the 3 real-life datasets.

maximum leaf count. The baseline then constructs a set of pre-computed aggregations based on the partition formed by the leaf nodes which is later combined with uniform sampling to generate the final answer.

We generate 1000 queries on each query template for evaluation and results can be found in Figure 2.8. On the left plot, we show the median CI ratio of the two approaches which indicates KD-PASS outperforms KD-US. On the right figure, we plot the average skip rate of KD-PASS. We note that as we increase the dimensions of the query, the skip rate decreases. This is expected because as we increase the dimension, the partitions that are relevant to a query (thus no skipping) increase exponentially in the worst case.

Due to the sizes of datasets, we use a maximum leaf count (i.e. number of partitions) of 1024 and a dimension of 5. Theoretically, there is no limitation in applying our framework to higher dimensions and partition sizes with proper engineering efforts.

Workload Shift

In this experiment, we extend the previous experiment on multi-dimensional query templates and evaluate the performance of KD-PASS and KD-US when the workload does not align perfectly with the attributes used to generate the pre-computed aggregates. We use the aggregates generated from Q2, i.e. the 2D query template, to solve all 5 templates. In this

—	Mean Cost			Median Relative Error						
Approach	Latency(ms)	Storage(MB)	Time(s)	Intel	Insta	NYC	NYC-2D	NYC-3D	NYC-4D	NYC-5D
PASS-BSS1x	24.8	0.5	20.7	0.34%	0.4%	0.2%	0.68%	2.9%	3.4%	3.6%
PASS-BSS2x	25.7	1.4	20.9	0.14%	0.29%	0.17%	0.48%	2%	2.1%	2.26%
PASS-BSS10x	29	5.9	21.1	0.09%	0.12%	0.08%	0.24%	0.97%	0.9%	1.2%
VerdictDB-10%	31	17.8	17	90.8%	90.8%	90.7%	90.9%	90.6%	90.7%	90.7%
VerdictDB-100%	842	176.8	49	0.09%	0.01%	0.07%	0.27%	0.46%	0.47%	0.48%
DeepDB-10%	21	21.2	86	0.9%	65.8%	0.9%	5.2%	24.6%	24.8%	25.6%
DeepDB-100%	22	61.5	154	1.1%	66.1%	1.1%	5.4%	24.7%	24.8%	25.4%

Table 2.2: We compare the median relative error of three PASS variations with VerdictDB and DeepDB on workloads we used in previous experiments. We measure the average latency of query processing, the storage, and the construction time (training time for DeepDB) required by each approach.

setting, the aggregates match Q2 perfectly, but it only shares 1 common attribute with Q1, 2 common attributes with Q3, Q4, and Q5.

The results shown in Figure 2.9 is quite encouraging. In a design like AQP++ that is without data skipping, as the dimension increases, the pre-computed aggregates will be less effective in solving a query because more 'hyper-rectangles' will be intersecting with the query thus increases the area that needs to be solved by sampling, therefore, the error (variance) will increase. However, due to the unique design of PASS, as long as the query template shares one or more common attributes, even the pre-computed aggregates that are not perfectly aligned with the target query can still be used for aggressive and reliable data skipping thus increase the accuracy of the sampling and leads to an overall better result. This can be a favorable feature in exploratory and interactive data analysis.

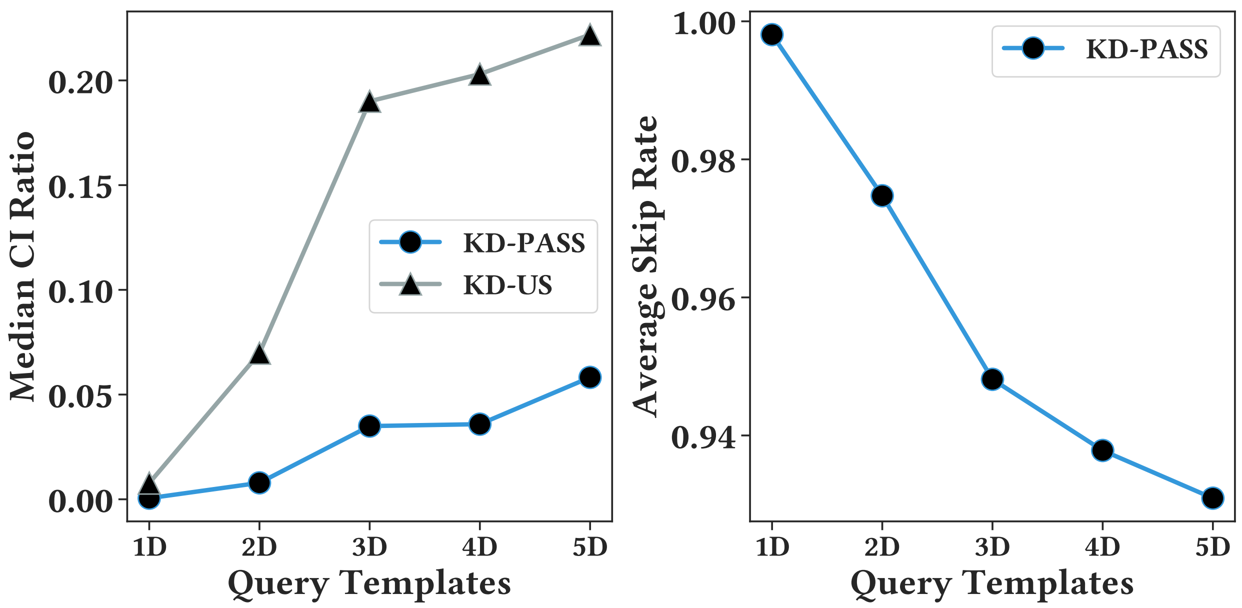


Figure 2.8: Multidimensional predicates on the NYC Taxi dataset. (Left) The median confidence interval ratio of KD-PASS vs. KD-US. (Right) The skip rate of KD-PASS.

Preprocessing Cost

Table 2.3 shows the preprocessing cost in seconds required by PASS given different numbers of partitions (k in the table) on the NYC Taxi dataset. We use an optimization sample rate of 0.0025% for the ADP algorithm to partition the dataset. As expected, the cost increases as we increase k but not significantly. This is because in our implementation we cache the results of the discretization method, therefore the partitioning cost of k=4 does not differ a lot with k=64 and the difference in preprocessing cost is mostly due to the partition tree construction. The cost of k=128 increases because more samples are used for a larger partition size. The results suggest that as we increase k, the latency decreases and the accuracy increases, this is because a fine grain partitioning can lead to more aggressive data skipping and more efficient sampling.

2.5.5 End-to-End Comparison with Other Systems

We run extensive experiments comparing PASS to VerdictDB[110] and DeepDB[56] on the 3 real datasets. In all of these experiments, we use the BSS mode of PASS and explicitly

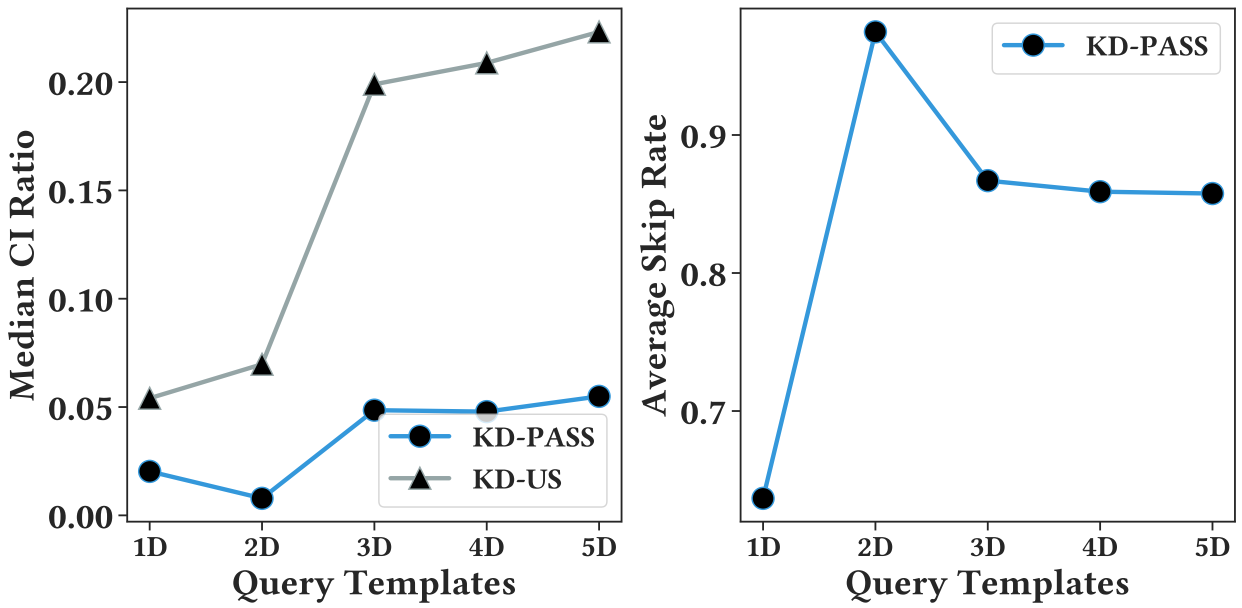


Figure 2.9: We use the aggregates constructed for a 2D query template to solve query templates of other dimensions. Left figure shows the Median confidence interval ratio of KD-PASS vs. KD-US on the NYC Taxi dataset. Right figure shows the percentage of tuples that are skipped by KD-PASS.

k	Cost(s)	Latency(ms)	MaxLatency(ms)	MedianRE
4	16	14.6	29.2	0.55%
8	18	13	26	0.32%
16	20	11.6	23.3	0.18%
32	22	10.7	21.4	0.11%
64	25	8.9	17.8	0.04%
128	50	6.4	12.9	0.03%

Table 2.3: Preprocessing cost, mean latency, max latency and accuracy given different number of partitions.

bound the storage size. We record the mean cost in terms of query latency, storage and construction/optimization time across different workloads, and we measure the median relative error of each approach on different workloads. For PASS, we build three variations using storage costs of 0.5MB, 1.4MB and 6MB; for VerdictDB, we use scrambles of a ratio of 10% and 100%; for DeepDB, we train models using 10% and 100% samples of the datasets.

The results in Table 2.2 show that VerdictDB-100% generates overall the most accurate results but its storage is about the same size of the original datasets, and its latency of 842ms — while 60% less than MySQL — is still much higher than PASS and DeepDB. On the other hand, the cost of VerdictDB-10% is more favorable but the accuracy drops a lot. DeepDB has the lowest latency among the three, its accuracy on the Intel Wireless dataset and the NYC 1D workload is at the same magnitude as the other two, but much worse on the Instacart dataset and the higher dimensional queries. And we also noticed that increasing the size of the training data and the storage cost of DeepDB does not necessarily improve the results. The three PASS variations demonstrate a trade-off between the costs and the accuracy: as we increase the storage, the latency increase slightly and the accuracy improves. The overall performance of PASS is slightly worse than VerdictDB-100% but we believe it is the most favorable approach given the accuracy and the costs.

2.6 Conclusion

While it has been proposed in previous work, we found theory around the joint use of pre-computation and sampling in synopsis data structures to be limited. The joint optimization, over both sampling *and* precomputation, is complex because one has to optimize over a combinatorial space of SQL aggregate queries while accounting for the real-valued effects of sampling. We propose an algorithmic framework that formalizes a connection between pre-computed aggregates and stratified sampling and optimizes over the joint structure. Our results are very promising, where we see clear accuracy benefits but with the cost of initial data structure construction. We further show how to tradeoff cost for accuracy.

As future work, we believe AQP needs to be examined in terms of synopsis construction and maintenance costs. If expensive up-front costs can be tolerated then accurate results can be found. Such a result is related to recent interest in learned models for AQP and cardinality estimation [56, 110, 134].

CHAPTER 3

JANUSAQP: EFFICIENT PARTITION TREE MAINTENANCE FOR DYNAMIC APPROXIMATE QUERY PROCESSING

3.1 Introduction

Approximate query processing (AQP) studies principled ways to sacrifice query result accuracy for faster or more resource-efficient execution [25, 48]. AQP systems generally employ reduced-size summaries, or "synopses", of large datasets that are faster to process. The simplest of such synopsis structures are histograms and samples [32, 87, 7, 80], but many others have been proposed in the literature. More complex synopses are more accurate for specific types of queries [131], specific data settings [118], or even are learned with machine learning models [134, 56, 93]. AQP is particularly interesting and challenging in a **dynamic data setting**, where a dataset is continuously modified with insertions and deletions [48, 106, 2]. In this setting, hereafter denoted as DAQP, any synopsis data structures created by the system will have to be continuously maintained online.

As an example use-case, consider a database aggregating per-stock order data for the NASDAQ exchange [1]. Suppose, that we would like to build a low-latency SQL interface for approximate aggregate queries over the past seven days of order data. On a typical day, there are 25M new orders that correspond to trades that are placed by brokers (up to 70,000 orders in any given second). A decent fraction of these orders are eventually canceled or prematurely terminated, for a variety of financial reasons. Thus, this database is highly dynamic with a large volume of new insertions (new orders) and a small but significant number of deletions (canceled orders). This paper explores such scenarios with similar motivating applications in internet-of-things monitoring and enterprise stream processing.

Simple synopses like 1D histograms and uniform samples are easy to maintain dynamically. However, such structures are often inaccurate in high-dimensional data and selective query workloads. More complex synopses structures, e.g, [134, 87] can be optimized for

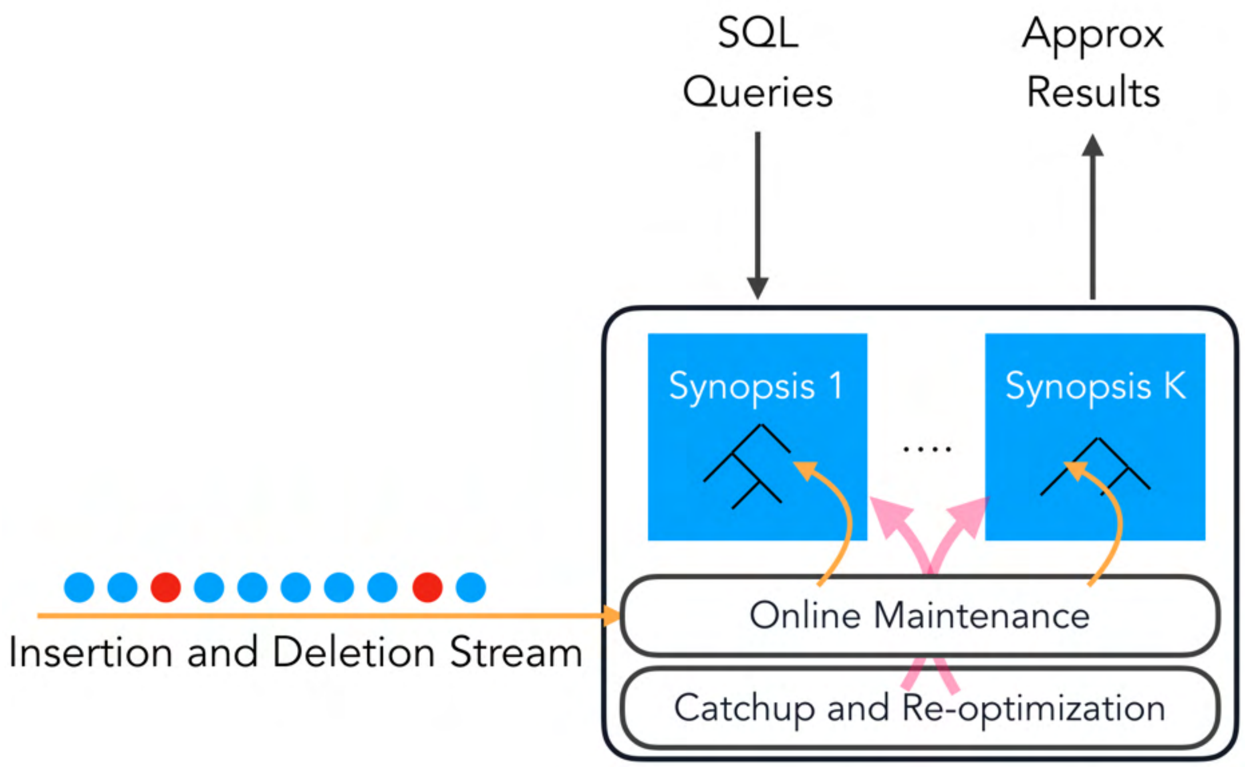


Figure 3.1: JanusAQP manages a collection of DPT synopses by maintaining them online while periodically re-optimizing partitioning and sample allocation.

a particular instance (dataset and query workload), but are generally harder to maintain online. For example, recently proposed learned synopses require expensive retraining procedures which limit insertion/deletion throughput [134, 56, 93]. Even classical stratified samples may have to be periodically re-optimized and re-balanced based on query and workload shifts [7]. These, expensive (re-)initialization procedures can significantly hurt insertion throughput, and accordingly, almost all existing AQP systems focus on the static data warehousing setting[1]. Unfortunately, the existing techniques that *are* designed for dynamic data, such as sketches and mergeable summaries [45, 4, 118], often cannot handle arbitrary deletions or aggregation queries with arbitrary predicates easily. Thus, it is understood that most synopsis data structures have at least one of the following pitfalls in our desired dynamic setting: throughput, drift, or generality [25].

This paper explores the DAQP problem and studies ways that we can mitigate the

1. A notable exception being the AQUA project [2] from 20 years ago.

pitfalls of prior approaches with a flexible synopsis data structure that can continuously re-optimize itself. We present JanusAQP, a new DAQP system, which supports SUM, COUNT, AVG, MIN, and MAX queries with predicates under arbitrary insertions and deletions to the dataset. The main data structure in JanusAQP is a dynamic extension of recently published work [86, 87], which we call a Dynamic Partition Tree (DPT). DPT is a two-layer synopsis structure that consists of a: (1) hierarchical partitioning of a dataset into a tree, and (2) a uniform sample of data for each of the leaf partitions (effectively a stratified sample over the leaves). An optimizer determines the best partitioning conditions and sample allocations to meet a user's performance goals. For each partition (nodes in the tree), we calculate the SUM, COUNT, MIN, and MAX values of the partition. Any desired SUM, COUNT, AVG, MIN, and MAX query can be efficiently decomposed into two parts with the structure: a combination of the partial aggregates where the predicate fully covers a partition in the tree, and an approximate part where the predicate partially covers a leaf node (and can be estimated with a sample). More importantly, this structure is essentially a collection of materialized views and samples, which can be maintained incrementally.

A core contribution of JanusAQP is online synopsis optimization. JanusAQP continuously monitors the accuracy of all of its DPT synopses to account for data and workload drift. When a synopsis is no longer accurate, it triggers a re-optimization procedure that resamples and repartitions the data. This re-optimization problem is both a significant algorithmic and systems challenge. From an algorithmic perspective, JanusAQP needs an efficient way to determine the optimal partitioning conditions in dynamic data. We propose an efficient algorithm based on a dynamic range tree index that finds a partitioning that controls the minimax query error (up to an approximation factor). From a systems perspective, re-optimization poses a bit of a logistical challenge. New data will arrive as the new synopsis data structure is being constructed. We design an efficient multi-threaded catch-up processing algorithm synchronizes new data and historical data without sacrificing the statistical rigor of the estimates.

Our prototype version of JanusAQP is integrated with the message-broker framework, Apache Kafka. Insertions, deletions, and user queries are processed as Kafka topics allowing for a multi-threaded DAQP server. In our experiments, we show that DPT is significantly more accurate than a reservoir sampling method and the DeepDB system [56]. It also achieves a throughput of processing nearly 200k records per second, while serving sub-millisecond query latencies.

3.2 Background

We find that previous discussions of DAQP have not been particularly formal (especially, in comparison to the streaming setting), and we make the problem setting more precise in this section. We also introduce the core concepts behind the synopses used in this work.

3.2.1 Dynamic Approximate Query Processing

We assume an initial database table $\mathcal{D}^{(0)}$. This initial table $\mathcal{D}^{(0)}$ is continuously modified through a stream of insertions and deletions of tuples. As a design principle, we assume that insertions are common but deletions are rare. With each insertion or deletion operation, the table evolves over time with a new **state** at each time step i:

$$\mathcal{D}^{(0)}, \mathcal{D}^{(1)}, ..., \mathcal{D}^{(i)}, \mathcal{D}^{(i+1)}, ...$$

A **synopsis** is a data structure that summarizes the evolving table. For each $\mathcal{D}^{(i)}$, there is a corresponding synopsis $\Sigma^{(i)}$:

$$\Sigma^{(0)}, \Sigma^{(1)}, ..., \Sigma^{(i)}, \Sigma^{(i+1)}, ...$$

In DAQP, the problem is to answer queries as best as possible from only the $\Sigma^{(i)}$. For a query q, the estimation error is defined as the difference between the estimated result (using the synopsis) and the true result (using the current database state):

$$\mathsf{Error}(q, \Sigma^{(i)}) = |q(\mathcal{D}^{(i)}) \quad - \quad q(\Sigma^{(i)})|$$

We further assume that there is sufficient cold/archival storage to store the current state of the table $\mathcal{D}^{(i)}$. This data can be accessed in an offline way for initialization, re-optimization, and logging purposes but not for query processing.

There are a few notable differences from the "streaming" setting. First, most data streaming models do not support arbitrary record deletion, i.e., as studied in [118]. We find that in many use-cases limited support for deletion is needed due to records that are invalidated through an out-of-band, asynchronous data process like fraud detection or financial auditing. Next, most streaming settings enforce a single pass over the data with limited overall memory. We do not make this assumption and allow for archival storage and slow access to old data. This is a more realistic AQP setting where all data are stored, however, there is limited working memory for a fast, approximate query answering service.

3.2.2 Related work

There is significant research in histograms and their variants that is highly relevant to this project [60, 74, 59]. V-Optimal histograms construct buckets to minimize the cumulative variance [60]. There are works on multi-dimensional histograms [80], and histograms on the streaming/dynamic setting [52, 50]. Like histograms, JanusAQP constructs partitions over attribute domains and aggregates within the partition. However, we contribute different partition optimization criteria than typically used in histograms and novel techniques based on geometric data structures to scale partitioning into higher dimensions. Furthermore, our system works in the general dynamic setting, unlike [50] where the number of total items must remain the same. Another related area of research is into mergeable summaries that compute a partition of the data and optimize sampling at a data partition level [122, 86, 4, 46, 118].

The DPT used in JanusAQP very much behaves like a mergeable summary but a far greater breadth of downstream queries. Furthermore, some prior work mostly focuses on a streaming setting without support for deletion [118]. Similarly, sketches [30, 32] have been used to find a summary of data to answer approximately a variety of queries efficiently. However, they also do not handle arbitrary range queries using space independent of the size of the full database. Generally, mergeable summaries and sketches usually focus on optimizing different types of problems such that frequency queries, percentile queries, etc. This paper shows how to operationalize a general DAQP system for aggregation queries with both systems and algorithmic contributions relating to the design of dynamic synopses and their continuous optimization. Our system can handle arbitrary insertions and deletions and can estimate any arbitrary predicate query with provable confidence intervals.

In databases, a number of tree-based indexes, such as the improved R^* tree [67], have been used for supporting range aggregation queries efficiently. The space of such indexes is super-linear with respect to the input items so they cannot be used for high volume of data, which is the main focus in this paper. In another line of work, tree-based data structured are used for returning a set of k uniform samples in a query range. More specifically, in [66, 133] the authors construct indexes such that given a query range Q and a parameter k, they return k uniform samples from the input items that lie in Q efficiently. These samples can be used to estimate any aggregation query in the range query Q. There are several issues with these indexes in our setting. First, the design of the index in [66] makes their structure inherently static and it cannot be maintained efficiently. Furthermore, the estimation error in both indexes is the same as the error in the simple uniform random sampling schema. In Section 3.6, we show that the error of our new index in real data sets is always less than half of the error in uniform random sampling, so our new index always outperforms these range sampling indexes. Finally, the space and the query time of these indexes depend on N, i.e., the size of the input set, so they cannot be used on big data.

Dynamic AQP problems have been discussed in prior work [48], however, most existing systems have focused on a static data warehousing setting [7]. The Aqua system [2] did consider the maintenance of its synopsis data structures under updates. However, these synopses were relatively simple and only samples and histograms. Furthermore, we discuss systems issues such as catch-up processing that was not discussed in [2] or any subsequent work [49].

Many new AQP techniques use machine learning. The basic ideas exist for a while, e.g., [61, 62]. Recently, there are more comprehensive solutions that train from a past query workload [111] or directly build a probabilistic model of the entire database [56, 134]. We show that these systems are not optimized for a dynamic setting. Even when they can be updated efficiently with warm-start training, their throughput is much lower than JanusAQP.

3.2.3 Partition Trees for AQP

We propose a new dynamic data synopsis and optimization strategy that is an extension of the work in [87]. Liang et al. proposed a synopsis structure called PASS (which we call SPT for "static partition tree"). SPT synopses are related to works such as [80] in the data cube literature and hybrid AQP techniques [113]. Liang et al. showed that with appropriate optimization of the partitioning conditions, an SPT could achieve state-of-the-art accuracy in AQP problems.

Construction

An SPT is a synopsis data structure used for answering aggregate queries over relational data. To use SPT, the user defines an *aggregation column* (numerical attribute to aggregate) and a *set of predicate columns* (columns over which filters will be applied). An SPT consists of two pieces: (1) a hierarchical aggregation of a dataset, and (2) a uniform sample of data for each of the leaf partitions (effectively a stratified sample over the leaves). The system returns a synopsis that can answer SUM, COUNT, AVG, MIN, and MAX aggregates over the aggregation column filtered by the predicate columns. Figure 3.2 illustrates a partition tree synopsis over toy stock-order data.

To understand how this structure is useful, let us overview some of its formal properties. A *partition* of a dataset $\mathcal{D}$ is a decomposition of $\mathcal{D}$ into disjoint parts $\mathcal{D}_1, ..., \mathcal{D}_B$. Each $\mathcal{D}_i$ has an associated partitioning condition, a predicate that when applied to the full dataset as a filter retrieves the full partition. Partitions naturally form a hierarchy and can be further subdivided into even more partitions, which can then be subdivided further. A *static partition tree* $\mathcal{T}$ is a tree with B nodes (where each node corresponds to a partition) with the following invariants: (1) every child is a subset of its parent, (2) all siblings are disjoint, and (3) the union of all siblings equals the parent.

In an SPT, each node of the tree is associated with SUM, COUNT, MIN, and MAX statistics over the items in $\mathcal{D}$ that lie inside the node. SPT synopses have a flexible height to tradeoff accuracy v.s. storage. In shorter trees, the leaf nodes of an SPT can cover large subsets of data and vice versa in deeper trees. Note how each layer of the tree in Figure 3.2 aggregates the lower layer over coarser-and-coarser aggregation conditions (e.g., first by "sector" and then by "order type").

This structure works well when the queries align with partition boundaries. For example, a user aggregating total orders by "order type" in Figure 3.2 would get an exact answer with no approximation. The challenge is to answer queries with predicates that partially intersect partitions. Due to the tree invariants, the set of partial intersections can be fully determined at the leaf nodes. To estimate the contributions of these partial intersections, an SPT associates a uniform sample of tuples *within that partition* for each leaf node.

Query Processing

Using an SPT, a user can estimate the result of a query as follows. Essentially, the query processing algorithm identifies "fully covered" nodes that are contained in the query predicate and "partially covered" ones that overlap in some way. Exact statistics from the "fully covered" nodes can be used, while estimates can be used to determine the contribution of "partially covered" ones. We present SUM, COUNT, AVG for brevity, but it is also possible to get estimations for MIN and MAX.

Step 1: Frontier Lookup. Given a query predicate q, traverse the tree top-down to retrieve two sets of nodes partitions: R_{cover} (nodes that fully cover the predicate) and $R_{partial}$ (nodes that partially intersect the predicate). Nodes that do not intersect the predicate can be ignored.

Step 2: Partial Aggregation For each partition in R_{cover}, we can compute an exact "partial aggregate" for the tuples in those partitions. For a SUM/COUNT query q: $agg = \sum_{R_i \in R_{cover}} SUM(R_i)$, for an AVG query, we weight the average by the relative size of the partition: $agg = \sum_{R_i \in R_{cover}} SUM(R_i) \frac{N_i}{N_q}$, where N_i is the size of the partition R_i, N_q is the total size in all relevant partitions of query q, and $SUM(R_i) = \sum_{t \in R_i \cap \mathcal{D}} t.a$ is the sum of the aggregation values of all tuples in the partition R_i.

Step 3: Sample Estimation. Each partition in $R_{partial}$ is a leaf node with an associated stratified sample. Within each stratified sample, we use standard AQP techniques to estimate that partition's contribution to the final query result [7]. For completeness, we include those calculations here. Suppose a partition R_i has a set S_i of m_i samples and there are N_i total tuples in R_i. We can formulate COUNT, SUM, AVG as calculating an average over transformed attributes:

$$f(S_i) = \frac{1}{m_i} \sum_{t \in S_i} \phi_q(t),$$

where $\phi_q(\cdot)$ expresses all the necessary scaling to translate the samples in query q into an average query population. In particular, if we define $Predicate(t, q) = 1$ if tuple t satisfies the predicate of query q, and 0 otherwise, we have

- COUNT: $\phi_q(t) = Predicate(t, q) \cdot N_i$
- SUM: $\phi_q(t) = Predicate(t, q) \cdot N_i \cdot t.a$

- AVG: $\phi_q(t) = Predicate(t,q) \cdot \frac{m_i}{\sum_{t \in S_i} Predicate(t,q)} \cdot t.a$

We run such a calculation for each partition that is partially covered. These results are combined with a weighted combination like before. For SUM/COUNT queries it is: $samp = \sum_{R_i \in R_{partial}} f(S_i)$. And for AVG queries, it is: $samp = \sum_{R_i \in R_{partial}} f(S_i) \cdot \frac{N_i}{N_q}$. N_i and N_q can be exactly retrieved from the statistics computed for each partition.

Step 4: Final Estimate. The results can be found by taking a sum of the two parts: $result = samp + agg$. For this result estimate, confidence intervals can be calculated using standard stratified sampling formulas.

3.3 System Architecture

In this section, we describe the JanusAQP architecture.

3.3.1 Construction and Optimization API

First, we overview how users construct synopsis data structures in JanusAQP. Unlike systems like BlinkDB [7], JanusAQP does not use a single synopsis to answer all queries. Much like index construction in a database, users choose which attributes to include in the synopsis structure. Each synopsis can answer query templates of the following form:

```
SELECT SUM/COUNT/AVG/MIN/MAX(A)
FROM D
WHERE Rectangle(D.c1,...,D.cd)
```

where A is an aggregation attribute and $c_1, ..., c_d$ are predicate attributes used in some rectangular predicate region (a conjunction of $>, <, =$ clauses). The **dimensionality** of a synopsis is the number of predicate attributes d. To construct a synopsis, the user must define the following basic inputs:

- **Aggregation Attribute.** An attribute A that is the primary metric for aggregation.

- **Predicate Attributes.** A collection of d columns $c_1, ..., c_d$ that are used to filter the data prior to aggregation.
- **Memory Constraint.** The maximum amount of space that the synopsis can take.
- **Query Processing Constraint.** The maximum bytes of data that the system should process in answering a query.
- **Historical Data Limit.** How much historical data to include in the synopsis, i.e., the earliest time-step of data included in the system.

JanusAQP contains an optimizer that integrates these constraints into a solver that produces an optimized synopsis (one with low error). Beyond these basic knobs that are relevant to most AQP systems, there are two other considerations discussed in this paper: **Catch-Up Processing.** Constructing a synopsis will require some amount of computational time. While incremental maintenance might be efficient, constructing the initial synopsis $S^{(0)}$ from the initial database state $\mathcal{D}^{(0)}$ might be very expensive if there is a significant amount of initial data. However, as the initial $S^{(0)}$ is being constructed new data will arrive at the system, and the system will require additional processing to catch up. JanusAQP optimizes the catch-up process using a multi-threaded system and approximate internal statistics for the partition tree. This process minimizes the amount of time where the system is unable to process new data or queries. The user decides how much processing to expend during catch up, the quicker the system is ready, the higher the error will be.

Throughput. The maximum data throughput is the maximum rate of insertions and deletions that the system can support. Throughput depends on the complexity of the synopsis used.

3.3.2 Data and Query API

For processing queries and data, we adopt the PSoup architecture where both queries and data are streams [21]. JanusAQP supports three types of requests: insertion of a new tuple, deletion of an existing tuple and querying of the existing tuples. Thus, there are three Kafka topics insert(tuple), delete(tuple), and execute(query).

The use of Kafka, with its timing and delivery guarantees, simplifies the query processing semantics. The system will process the incoming stream of queries in order. Each query will have an arrival time i, which is the current database state at the time at which the query is issued. Therefore, we define $q_j^{(i)}$ as the jth query in the sequence that arrives at database state i. Query results should reflect all of the data that has arrived until the time point i.

3.3.3 Summary and Algorithmic Contributions

To summarize, the usage of JanusAQP can be thought of as a life cycle. (1. Initialization) The user triggers synopsis construction through an API call. (2. Catch-Up) The system will online construct the synopsis while managing new data arriving into the system. (3. Query/-Data Processing) Then, JanusAQP is ready to process requests of insertions, deletions, and queries. (4. Re-Initialization) As JanusAQP processes more updates, the data or query workload could drift requiring re-partitioning. This procedure re-enters the initialization phase.

Throughout the rest of the paper, we present technical contributions throughout this synopsis life cycle:

- **Dynamic Partition Trees (Section 3.4)** The core data structure in JanusAQP is called a dynamic partition tree (DPT). This data structure aggregates data at multiple levels of resolutions and associates some of the aggregates with stratified samples. The size of the tree (i.e., depth and width) can be changed to tradeoff accuracy at the cost of increased storage and throughput. The sampling rate at the leaf nodes can be adjusted to tradeoff query latency with accuracy.

- **Warm-Start Deployment (Section 3.4.3)** Next, we present a multi-threaded technique that allows DPT synopses to be deployed in dynamic environments accounting for new data that might arrive during their construction. This component crucially allows for online re-partitioning (either partially or entirely) and re-balancing of the data structure.

- **Minimum Variance Partitioning (Section 3.5)** We describe a new, efficient algorithm for selecting the DPT tree structure that minimizes the worst-case estimation error.

3.4 Dynamic Partition Trees

Now, we describe the main data structure in JanusAQP which extends the work described in Section 2 (SPT synopses). We discuss how Dynamic Partition Trees (DPT) are constructed, how they answer queries, and how they are maintained under updates. Structurally, a DPT is essentially the same data structure as an SPT with layers of partitioned aggregated and leaves that are associated with samples. However, the way that the partition statistics and samples are represented differ to allow for incremental maintenance. Figure 3.3 summarizes the basic update process whose details are described in the subsequent sub-sections.

3.4.1 Incrementally Maintaining Nodes

As in an SPT, each node defines a partition and contains statistics (the SUM, COUNT, MIN, and MAX aggregates) of the data contained in that partition. Moving to the dynamic setting, the key challenge is to keep these statistics up-to-date in the presence of insertions and deletions. When an insertion or deletion arrives, an entire path of nodes from the leaf to the root will have to be updated.

Table 3.1: Table of basic notation

$\mathcal{D}$	Full database
N	$\|\mathcal{D}\|$
S	Set of reservoir samples
H	Set of catch-up samples
R_i	Partition/bucket/rectangle
$\|R_i\|$	$\|R_i \cap S\|$
N_i	$\mathcal{D} \cap R_i$
S_i	$S \cap R_i$
H_i	$H \cap R_i$
m_i	$\|S_i\|$
h_i	$\|H_i\|$
m	$\|S\|$
t	Tuple in $\mathcal{D}$
$t.a$	Aggregation value of tuple t
$\mathcal{T}$	Partition tree in DPT

DPT Nodes: First, we discuss how we represent the statistics in a DPT node. Since the SUM and COUNT are easy to incrementally maintain under both insertions and deletions, we simply store a single SUM and COUNT value for each aggregation attribute. The MIN and MAX values are harder to incrementally maintain. To store the MIN and MAX values, we store the top-k values and the bottom-k values in a MIN/MAX heap respectively [2]. The top value of these heaps is equal to the MIN and MAX of all the data in the node.

Insert New Record: When a new record is inserted, we first test each leaf node to see if the record is contained in the node. Once find the appropriate leaf node, we then increment the SUM and COUNT statistics accordingly. Finally, we push the new aggregation values onto the heap. If the heap exceeds the size limit k, then the bottom value on the heap is removed.

Delete Existing Record: When an existing record is deleted, we first test each leaf node to see if the record is contained in the node. Once find the appropriate leaf node, we then decrement the SUM and COUNT statistics accordingly. Finally, if that aggregation value is contained in the heap it is removed from the heap. This might make the heap smaller than

2. $k = 10$ in our implementation

k. Repeated deletes from the same node might fully empty the heap. We stop removing values from the heap when there is only one value left. When the heap reaches a single element the MIN/MAX estimates received from the nodes are outer approximations where the estimated value is larger than the MAX and smaller than the MIN.

3.4.2 Maintaining Stratified Samples

Next, we describe how to maintain the samples associated with leaf nodes. We use a modified version of the well-known technique of *reservoir-sampling* [130] under updates [49]. The details of how we implement this are interesting. Conceptually, each leaf node is associated with a physically disjoint sample of just that partition, i.e., a stratified sample. Instead of physical strata, we implement virtual partitions of a single global sample. This global sample can be maintained using a reservoir sampling algorithm and makes it easier to control the overall size of the synopsis under insertions/deletions as well as simplifies concurrency control.

Sample Representation: The DPT maintains a "pooled" sample (all the relevant samples in a single data structure). This set of samples has a target size of $2m$ tuples. At the construction time, we choose a set S of $2m$ uniform random samples from $\mathcal{D}$. The update procedure ensures that there are always between $m \leq |S| \leq 2m$ samples. The leaf nodes index into this "pooled" sample selecting only the relevant data to their corresponding partitions.

Insert New Record: Suppose that we insert a new tuple t. If $|S| < 2m$ we add t in S. If $|S| = 2m$, then we choose t with probability $\frac{|S|}{|\mathcal{D}|}$. If it is selected then we replace t with a point from S uniformly at random.

Delete Existing Record: Next, suppose that we delete a tuple t from $\mathcal{D}$. If $t \notin S$ we do not do anything. If $t \in S$ then we check the cardinality of S. If $|S| > m$ then we only remove t from S. If $|S| = m$ then we skip the set S and we re-sample $2m$ items from $\mathcal{D}$. As shown in [49] this procedure always maintain a set of uniform random samples. Using a simple dynamic search binary tree of space $O(m)$ we can update the samples S stored in $\mathcal{T}$ in $O(\text{height}(\mathcal{T}))$ time.

3.4.3 Re-initialization and Catch-Up

As we noted before, repeated deletes on the same leaf partition can degrade the accuracy of the synopsis. As we will see in the next section, it is also possible for repeated insertions to degrade the accuracy as well. In such cases, re-initialization of the DPT may be needed where the data structure is re-built and re-optimized over existing data.

Enabling periodic re-initialization is crucial for reliable long-term deployment but is challenging because new data will not simply stop arriving during the re-initialization period. As the dataset size grows, the amount of time needed for re-initialization will grow as well. We employ a multi-threaded approach to minimize any period unavailability for processing new data arrival as well as new queries (Figure 3.4). When re-initialization is triggered, the main processing thread initiates the construction of a new DPT synopsis (the algorithm for determining the optimal partitions is in the next section), and the following steps are performed:

1. Optimization Phase (In Parallel)

    - The partition optimization algorithm analyzes the data in the pooled reservoir sample to determine the optimal new partitioning criteria. It returns a new empty DPT with no node statistics.

    - In parallel with (Step 1), the old synopsis is maintained under all insertions and deletions that happen during the optimization algorithm. Queries can still be answered with the old synopsis.

2. (Blocking) Approximate node statistics are populated into the new synopsis using the pooled reservoir sample S (note, that this will reflect any data that arrived during the

optimization phase). This is the only blocking step in the re-initialization routine and new data and queries will have to wait until completion.

3. The old synopsis is discarded.

4. The system resamples a uniform sample of data from archival storage to be the new pooled reservoir sample. Queries and results can still be processed on the new synopsis even without a sample.

5. Random samples of historical data are used to improve the node statistics in the background until a user-specified "catch-up" time.

This process is the key difference between an SPT and an DPT, where after catch-up the node statistics may be inexact. However, this old data is propagated in a random order, which means that the SUM,COUNT,AVG values in each node will be unbiased estimates of their full data statistics. The duration of the catch-up phase can be chosen by the user. For example, in our experiments, the catch-up phase does not stop until we get $0.2 \cdot |\mathcal{D}|$ samples. It is worth noting that queries close to the beginning of the catch-up phase will have a higher error, however queries towards the middle or the end of the catch-up phase will have a smaller error. In the later section 3.5.4, we describe how to trigger re-initialization. Furthermore, there is only one step (2) where the synopsis is unavailable to process queries and data (has the duration of 100s of milliseconds in our experiments).

This multi-threaded re-initialization is also valuable in a warm-start setting. Imagine a scenario where JanusAQP is deployed onto existing infrastructure with historical data and continuously arriving new data. In this case, even the initial construction of the first DPT synopsis will require a catch-up process.

3.4.4 Answering Queries With a DPT

Next, we discuss how to answer SUM, COUNT, AVG queries with a DPT (we can also answer MIN/MAX queries but without giving confidence intervals). The basic structure of the result estimator is the same as before, especially for the $R_{partial}$ partitions. However, there are a few key changes due to the nature of the catch-up phase. In SPT, for each partition in R_{cover}, we can compute an exact "partial aggregate" for the tuples in those partitions and combine the partial aggregates. In a DPT, this process changes considering the estimations we get from the catch-up samples. Overall, the estimation of a partition $R_i \in R_{cover}$ consists of i) estimation using the catch-up samples H and the formulas of Section 3.2.3, ii) the exact statistics of the new inserted tuples in R_i, and iii) the exact statistics of the deleted tuples in R_i (recall that the quantities in ii), iii) are stored and maintained as described in the Incrementally Maintaining Statistics in Section 3.4.1). By taking the sum of i), ii) and subtracting iii) we get the unbiased estimation in partition R_i.

Let H be the set of catch-up samples and $H_i \subseteq H$ be the subset of H that lie in a partition R_i, and $h_i = |H_i|$. All basic notations are defined in Table 3.1. The formulas for estimating COUNT and SUM queries in both R_{cover}, $R_{partial}$ from Section 3.2.3 contain the factor $\frac{N_i}{m_i}$ or $\frac{N_i}{h_i}$, while the formulas for estimating the AVG contain the factor $\frac{N_i}{N_q}$. In DPT we do not have the exact values for N_i. Instead, we use an estimate of the size of the partition R_i denoted by $\hat{N}_i$. In particular we use the catch-up samples H to estimate $\hat{N}_i = \frac{h_i}{h} N$.

Confidence Intervals

While the estimators do not significantly change from an SPT to a DPT, the confidence intervals are calculated very differently. This is because there are now two sources of errors: estimation errors due to the stratified samples and estimation errors in the node statistics. Both these sources of errors have to be integrated into a single measure of uncertainty. To be able to analytically calculate these errors, we make a simplifying assumption to avoid having to deal with products of random variables.

Large partition assumption. We assume that the partition sizes are enough ($\Omega(\frac{m}{\alpha})$, where α is the reservoir-sampling rate) such that $\frac{N_i}{N_q}, \frac{N_i}{m_i}$ can be treated as deterministic values $\frac{\hat{N}_i}{\hat{N}_q}, \frac{\hat{N}_i}{m_i}$, as we had in the unbiased estimation above. This is an assumption similar to those made in previous AQP work [55].

Once we make this assumption, the central limit theorem can be used to asymptotically bound the estimation error for SUM/COUNT/AVG queries. Informally, the central limit theorem states that this asymptomatic error is proportional to the square-root of the ratio of estimate variance and the amount of samples used $\propto \sqrt{\frac{var(est_i)}{m_i}}$. We simply have to match terms to this formula for all sample estimates and all node estimates because both are derived from samples.

Error in Node Estimates. First, let's account for all the uncertainty due to catch-up. Recall that H is the set of catch-up samples we have considered so far and $H_i \subseteq H$ is the samples in partition R_i with $h_i = |H_i|$. We note that we do not store the set H or the subsets H_i, instead we only use the new catch-up samples to continuously improve the statistics we store in the nodes. Using the notation in the previous section, we can calculate the catch-up variance ν_c:

$$\nu_c(q) = \sum_{R_i \in R_{cover}} w_i^2 \frac{var(\phi_q(H_i))}{h_i}$$

where $w_i = \frac{\hat{N}_i}{\hat{N}_q}$ for AVG queries and $w_i = 1$ for SUM/COUNT queries. Calculating $\phi_q(H_i)$ is straight-forward. We simple store additional information that allows us to efficiently calculate the variance. For any node i of $\mathcal{T}$ we store h_i, $\sum_{t \in H_i} t.a^2$, $\sum_{t \in H_i} t.a$.

Error in Sample Estimates. For a partition $R_i \in R_{partial}$, let $S_i \subseteq \mathcal{D}$ be the set of samples in S that lie in partition R_i and let $m_i = |S_i|$. Like the catch-up variance, we can calculate the sample estimate variance ν_s:

$$\nu_s(q) = \sum_{R_i \in R_{partial}} w_i^2 \frac{var(\phi_q(P_i))}{m_i}$$

We can calculate an overall confidence interval as:

$$\pm z \cdot \sqrt{\nu_c(q) + \nu_s(q)}$$

Where z is a normal scaling factor corresponding to the desired confidence level, e.g., $z = 1.96$ for 95%. As before, $w_i = \frac{\hat{N}_i}{\hat{N}_q}$ for AVG queries and $w_i = 1$ for SUM/COUNT queries.

3.5 Optimal DPT Partitioning

We next describe a new dynamic partitioning algorithm designed for the dynamic setting.

3.5.1 Preliminaries and Problem Setup

The partitioning algorithm analyzes the pooled reservoir sample of data to determine how best to partition the dataset. The goal of the partitioning algorithm is to find a partitioning such that the subsequent queries issued to the DPT have low-error. Surprisingly enough, the partitioning algorithm does not need an exact query workload to perform this optimization. It simply needs a focus aggregation function (e.g., SUM, COUNT, AVG) and finds a partitioning that minimizes the worst-case query error for sufficiently large predicates. We do find that the choice of focus function is a bit academic due to differences in their error calculations, and optimizing for one of them usually results in good partitioning for all.

More formally, let Q be a set of possible aggregate queries with a predicate. And, let Θ be the set of all DPT synopses defined by rectangular partitioning conditions over k partitions. The main optimization objective is to minimize the maximum error over the query workload:

$$\min_{\mathcal{T} \in \Theta} \max_{q \in Q} \mathsf{Error}(q, \mathcal{T}) \tag{3.1}$$

As a proxy for this objective, one can minimize the maximum confidence interval length, which is equivalent to minimizing the maximum variance $\nu_s(q)$.

The above problem still seems challenging because queries can intersect partitions and cut a tree in arbitrary ways, but recent work shows an important simplification [87] (under mild technical conditions about the size of partitions). Instead of looking over all possible queries to minimize the maximum error, one only needs to focus on single partitions to ensure they do not have "high-variance" sub-partitions. Indeed by considering only these sub-partitions we can still get a $\sqrt{k}$-approximation for COUNT and SUM queries over the optimum partition considering all queries (for k leaves). The approximation factor improves to $\sqrt{2}$ for $d = 1$. For AVG queries the error of the optimum partition of this simplification is the same with the maximum error considering every possible query.

Thus, the optimization problem reduces to finding partitions that do not contain a high-variance "rectangle" of data. We next show that a special indexing structure called a can be used for this purpose. The core procedure of our partitioning algorithm is to find what is the maximum error of a query that lies completely inside a rectangular region—once that can be done efficiently a simple search procedure can produce the partitioning condition.

Indexing To Find Maximum Variance

Now, we describe the core subroutine of all of our partitioning algorithms. [3] A range tree [34] is a tree data structure that is built over a set of n points in $\mathbb{R}^d$ and it is used for aggregation or reporting rectangle range queries. Every point is stored in the leaf nodes and every inner node stores an interval defined over the maximum and the minimum coordinate of the points stored in the leaf nodes of the tree. Given a query rectangle, it returns the result of an aggregation query, such as a sum, in $O(\log^{d-1} n)$ time. The range tree can be made dynamic supporting insertions and deletions in $O(\log^{d} n)$ time [20, 100].

The overall data structure for our variance calculation is a modified dynamic range tree. We note that any variance can essentially be represented as the difference of two sums: sum

3. Due to space limitations, we only provide a high-level description. All the details and technical proofs will be shown in the full version of the paper.

of squares and the square of the sum. For particular focus functions, the technical details differ. In particular, for COUNT queries we construct a data structure of $O(m \log^d m)$ space that can be updated in $O(\log^d m)$ time such that given a query rectangle it finds the maximum variance query in $O(\log^d m)$ time. For SUM queries we construct a data structure with the same guarantees as in the COUNT data structure but it returns a $\frac{1}{4}$-approximation of the maximum variance. For AVG queries we construct a data structure of $O(m \log^{3d} m)$ space that can be updated in $O(\log^{3d+1} m)$ time such that given a query rectangle it finds a $\frac{1}{4 \log^{d+1} m}$-approximation of the the maximum variance query in $O(\log^{2d+1} m)$ time. In the next subsections we use the notation $\mathcal{M}(R)$ for the value of the variance returned by our approximation algorithm in a rectangle R.

3.5.2 Partitioning for $d = 1$

Now, we discuss how to solve the partitioning optimization problem in one dimension. We present results for SUM and AVG queries. COUNT can be thought of as a special case of SUM with binary data. The basic trick is to search over a discretized set of possible variance values. For each value e, we try to construct a partitioning of k partitions such that in each bucket the length of the longest confidence interval of a query is at most e. By systematically reducing e in each iteration, we control for the worst-case error.

Bounding the Error. The first step is to calculate the bounds for the maximum length of the largest possible confidence interval among queries that intersect one partition.. We assume that the aggregation value of any item in $\mathcal{D}$ is bounded by a maximum value $\mathcal{U}$ and a minimum non-zero value $\mathcal{L}$. We allow items to take zero values since this is often the case in real datasets but no item with positive value less than $\mathcal{L}$ or larger than $\mathcal{U}$ exists. We assume that $\mathcal{U} = O(\text{poly}(N))$ and $\mathcal{L} = \Omega(1/\text{poly}(N))$. In the full paper we show that the length of the longest confidence interval is also bounded by $O(\text{poly}(N))$ and $\Omega(1/\text{poly}(N))$.

Lemma 2. Let R be any rectangle and let $\mathcal{V}_S(R), \mathcal{V}_A(R) > 0$ be the variance of the SUM and AVG query respectively with the maximum variance in R. Then it holds that $\frac{\mathcal{L}}{\sqrt{2}} \leq \sqrt{\frac{\mathcal{V}_S(R)}{|R|}} \leq N\mathcal{U}$ and $\frac{\mathcal{L}}{\sqrt{2}N} \leq \sqrt{\frac{\mathcal{V}_A(R)}{|R|}} \leq \sqrt{N}\mathcal{U}$.

Proof. Without loss of generality let q be the SUM or AVG query with the maximum variance in R.

First we focus on SUM queries. Let $N_R = |R \cap \mathcal{D}|$ be the number of total tuples in R. Unless $V(q) = 0$, from [87], we know that there exists a query q' with $|q'| = |R|/2$ such that $\frac{V(q)}{|R|} \geq \frac{N_R^2}{|R|^3}\frac{|R|}{2}\sum_{t\in q'} t.a^2$, where $\sum_{t\in q'} t.a^2 > 0$. We also have $\sum_{t\in q'} t.a^2 \geq \mathcal{L}^2$ and $w_u = 1$ leading to $\sqrt{w_u\frac{\mathcal{V}_S(R)}{|R|}} \geq \frac{N_R}{\sqrt{2}|R|}\mathcal{L} \geq \frac{\mathcal{L}}{\sqrt{2}}$. Furthermore, we have $V(q) \leq \frac{N_R^2}{|R|^2}|R|^2\mathcal{U}^2 \leq N^2\mathcal{U}^2$ leading to $\sqrt{w_u\frac{\mathcal{V}_S(R)}{|R|}} \leq N\mathcal{U}$.

Next, we consider AVG queries. Unless $V(q) = 0$, from [87], we know that there exists a query q' with $|q'| = \delta m \leq |R|/2$ such that $\frac{V(q')}{|R|} \geq \frac{1}{|R|\delta^2 m^2}\frac{|R|}{2}\sum_{t\in q'} t.a^2$, where $\sum_{t\in q'} t.a^2 > 0$. We also have $\sum_{t\in q'} t.a^2 \geq \mathcal{L}^2$ and $w_u = 1$ leading to $\sqrt{w_u\frac{\mathcal{V}_A(R)}{|R|}} \geq \frac{1}{\sqrt{2}\delta m}\mathcal{L} \geq \frac{\mathcal{L}}{\sqrt{2}N}$. Furthermore, we have $V(q) \leq \frac{|R|}{|q|^2}\mathcal{U}^2 \leq |R|\mathcal{U}^2 \leq N\mathcal{H}^2$ leading to $\sqrt{w_u\frac{\mathcal{V}_A(R)}{|R|}} \leq \sqrt{N}\mathcal{U}$. $\square$

Description of Algorithm. We describe the partitioning algorithm for SUM queries. The procedure is identical for AVG queries. For a parameter $\rho \in \mathbb{R}$ with $\rho > 1$, let $E = \{\rho^t \mid t \in \mathbb{Z}, \frac{\mathcal{L}}{\sqrt{2}} \leq \rho^t \leq N\mathcal{U}\} \cup \{0\}$, be the discretization of the range $[\frac{\mathcal{L}}{\sqrt{2}}, N\mathcal{U}]$, i.e., the lower and upper bound of the longest confidence interval (assuming queries completely inside one bucket), by the multiplicative parameter ρ. For an interval b, let $\mathcal{M}(b)$ be the approximation of the query with the maximum variance in bucket b (supporting updates) as described in Section 3.5.1. We run a binary search on the values of E. For each value $e \in E$ we consider, we try to construct a partitioning of k partitions such that in each partition the length of the longest confidence interval of a query is at most e. If there exists such a partitioning we continue the binary search with values $e' < e$. If there is no such a partitioning we continue the binary search with values $e' > e$. In the end of the binary search we return the last partitioning that we were able to compute.

It remains to describe how to check if a partitioning with k buckets (intervals) with maximum length confidence interval at most e exists. A high level description of the algorithm is the following.

1. For $i = 1$ to k
   (a) Let b_i be the i-th bucket with left endpoint t_a
   (b) Binary search on samples t_j to find the maximum bucket b_i with error at most e
   (c) If $\sqrt{\frac{\mathcal{M}([t_a, t_j])}{j-a+1}} \leq e$
      i. Continue search for values $> j$
      ii. Else Continue search for values $< j$
2. If we find a partitioning where all samples belong to a bucket construct and return $\mathcal{T}$ using b_i as its leaf nodes. Otherwise we return $\mathcal{T} = \emptyset$

We start with the leftmost sample, say t_1, which is the left boundary of the first bucket. In order to find its right boundary we run a binary search on the samples S. Let t_j be one of the right boundaries we check in the binary search, and let $b_1 = [t_1, t_j]$. If $\sqrt{\frac{\mathcal{M}(b_1)}{|b_1|}} \leq e$ then we continue the binary search with a sample at the right side of t_j (larger bucket). Otherwise, we continue the binary search with a sample at the left side of t_j (smaller bucket). When we find the maximal bucket with longest confidence interval at most e we continue with the second bucket repeating the same process for at most k buckets. In the end, if all samples in S are contained in k buckets then we return that there exists a partitioning (with k buckets) with maximum variance at most e. If we cannot cover all samples in k buckets then we return that there is no partitioning (with k buckets) with maximum variance at most e.

Correctness. We define the notation $\mathcal{V}(R)$ as the variance of the maximum variance query that lies completely inside the partition/rectangle R. Notice that $\mathcal{M}(R)$ returns an approximation of $\mathcal{V}(R)$. Before we start with the correctness proof of our algorithm we recall that in [87] the authors showed that under a mild assumption, for two buckets b_i, b_j if $b_i \subseteq b_j$

Algorithm 2:

```
Data: $e, S, k, \eta$
Result: $\mathcal{T}$
1  $j \leftarrow 0$;
2  for $i = 1$ to $k$ do
3      $i_1 \leftarrow j + 1$, $i_2 \leftarrow |S| - 1$;
4      if $i_1 > |S|$ then
5          break;
6      while $i_1 \leq i_2$ do
7          $i_3 \leftarrow \text{floor}((i_1 + i_2)/2)$;
8          if $i_3 - i_2 + 1 \geq \eta$ then
9              $W \leftarrow \mathcal{M}([t_a, t_{i_3}])$;
10             if $\sqrt{\frac{W}{i_3 - i_2 + 1}} \leq e$ then
11                 $j \leftarrow i_3$;
12                 $i_1 \leftarrow i_3 + 1$;
13             else
14                 $i_2 \leftarrow i_3 - 1$;
15             end
16         else
17             $i_1 \leftarrow i_3 + 1$;
18         end
19     end
20     $b_i \leftarrow [t_a, t_j]$;
21 end
22 if $(\bigcup_i b_i) \cap S = S$ then
23     Construct search binary tree $\mathcal{T}$ using $b_i$ as leaf nodes;
24     return $\mathcal{T}$;
25 else
26     $\mathcal{T} \leftarrow \emptyset$;
27 end
```

then $\sqrt{\frac{\mathcal{V}(b_i)}{|b_i|}} \leq \sqrt{\frac{\mathcal{V}(b_j)}{|b_j|}}$, namely the length of the longest confidence interval in b_i is smaller than the length of the longest confidence interval in b_j. We call it the monotonic property of the longest confidence interval.

We assume that $\mathcal{M}(b_i)$ computes a $\frac{1}{\gamma}$-approximation of the maximum variance in b_i, i.e., $\mathcal{M}(b_i) \geq \frac{1}{\gamma}\mathcal{V}(b_i)$. Let $\mathcal{R}^*$ be the optimum partitioning and let b^* be the bucket that contains the query with the longest confidence interval in $\mathcal{R}^*$. First, we notice that if $e \geq \sqrt{\frac{\mathcal{V}(b^*)}{|b^*|}}$ then we always find a partitioning with longest confidence interval at most e. We can show it by induction on the right boundaries of the buckets (intervals) and the monotonic property of confidence intervals. We show all details in the full version of the paper. For the base case, let $b_1^* = [t_1, t_2]$ be the first bucket of partitioning $\mathcal{R}^*$. The procedure $\mathcal{M}$ always underestimates the maximum variance in an interval so the binary search in our procedure will consider the right boundary to be greater than t_2. Let t_i be the right boundary of the i-th bucket in $\mathcal{R}^*$ and let assume that the i-th bucket in our procedure has a right boundary $t_j \geq t_i$. We consider the $(i+1)$-th bucket in $\mathcal{R}^*$ with boundaries $[t_{i+1}, t_r]$. We show that the $(i+1)$-th bucket in our procedure has a right boundary at least t_r. Let $[t_a, t_b]$ be the boundaries of the $(i+1)$-th bucket in our procedure. We have $t_a \geq t_{i+1}$. If $t_a = t_{i+1}$ then $t_b \geq t_r$ as in the basis case. If $t_a > t_{i+1}$ then because of the monotonic property of the confidence intervals and the fact that the $\mathcal{M}$ procedure underestimates the maximum variance we also have that $t_b \geq t_r$. Let e' be the smallest value in E such that $\sqrt{\frac{\mathcal{V}(b^*)}{|b^*|}} \leq e'$. Because of the previous observation our algorithm always returns at least a valid partitioning for an $e \leq e'$. For every bucket b of this partitioning, $\sqrt{\frac{\mathcal{M}(b)}{|b|}} \leq e$. Let b' be the bucket in the returned partitioning containing the query with the longest confidence interval. We have, $\sqrt{\frac{\mathcal{V}(b')}{|b'|}} \leq \sqrt{\gamma\frac{\mathcal{M}(b')}{|b'|}} \leq \sqrt{\gamma}e \leq \sqrt{\gamma}e' \leq \rho\sqrt{\gamma}\sqrt{\frac{\mathcal{V}(b^*)}{|b^*|}}$.

From Section 3.5.1 we have that $\gamma = 4$ for SUM and AVG queries queries. So we get a partition where the maximum error is within $2\rho\sqrt{2}$ of the optimum error for SUM queries and within 2ρ of the optimum error for AVG queries.

Running time. We assume that $\mathcal{M}(\cdot)$ can be computed in M time. Since, $\mathcal{L}$,$\mathcal{U}$ are polynomially bounded on N we have that $|E| = O(\log_\rho N)$ and it can be constructed in $O(\log_\rho N)$ time. The binary search over E takes at most $O(\log \log_\rho N)$ steps. We can decide if there exists a partitioning with confidence interval e in $O(kM \log m)$ time. Overall, the running time of our algorithm is $O(kM \log m \log \log_\rho N)$. If ρ is a constant, for example $\rho = 2$, then the running time is $O(kM \log m \log \log N)$. From Section 3.5.1 we have that in 1-dimension $M = O(\log m)$ for SUM and $M = O(\log^2 m)$ for AVG queries. Notice that if we skip the log factors the running time depends only linearly on the number of buckets k and the approximation factor is constant.

3.5.3 Partitioning in Higher Dimensions

We construct a partitioning by building a k-d tree using the dynamic procedures $\mathcal{M}$ as shown in Section 3.5.1. Using the results of [87] we could construct a near optimum k-d tree in time $O(km)$ skipping the log factors. Here, we use our new results from Section 3.5.1 to construct a k-d tree faster (in roughly $O(k)$ time) with better approximation approximation guarantees.

The high level description of the algorithm is the following.

1. Max Heap C containing partition R_1 covering all items in $\mathcal{D}$
2. For $j = 2$ to k
   (a) Extract the partition R_i with maximum $\mathcal{M}(R_i)$ from C
   (b) Create a partitioning of R_i of two partitions R_{i_1}, R_{i_2} by splitting on the median of R_i
   (c) Insert $\mathcal{M}(R_{i_1})$, $\mathcal{M}(R_{i_2})$ in C
   (d) Set R_{i_1}, R_{i_2} as children of R_i in $\mathcal{T}$

We start by constructing a dynamic data structure from Section 3.5.1 over the initial set of samples S. Assume that after a number of updates in S we want to (re)construct the tree structure $\mathcal{T}$ over S. We pre-define an ordering of the dimensions. Each node i of the tree is associated with a partition/rectangle R_i. We build the tree in k iterations in a top-down manner starting from the root which is associated with a partition R_1 such that $\mathcal{D} \subset R_1$. In any iteration we store and maintain the approximate maximum variance queries of every leaf node in a max heap C. In the end of the j-th iteration we have a tree of j leaf nodes. Let R_i be the partition of the leaf node with the maximum $\mathcal{M}(R_i)$ value in C. We remove its value from C. We find the medium coordinate with respect to the next dimension in the ordering (in this branch of the tree) among the samples $R_i \cap S$. We split R_i on the median into two partitions R_{i_1}, R_{i_2} and we add them as children to the node of R_i. Using the algorithms from Section 3.5.1 we compute $\mathcal{M}(R_{i_1}), \mathcal{M}(R_{i_2})$ and we insert their values in the max-heap C. We continue with the same way until we construct a tree $\mathcal{T}$ with k leaf nodes (buckets).

We can show that such a tree construction returns a partitioning which is near optimal with respect to the optimum partition tree construction following the same splitting criterion: split on the median of the leaf node with the largest maximum variance query. For any query our data structure from Section 3.5.1 can be updated in $O(\text{polylog} m)$ time. For a (re-)partition activation over a set S of m samples we can construct a new $\mathcal{T}$ with the following guarantees: For COUNT/SUM queries, $\mathcal{T}$ can be constructed in $O(k \log^d m)$ time with approximation factor $2\sqrt{k}$. For AVG queries, $\mathcal{T}$ can be constructed in $O(k \log^{2d} m)$ time with approximation factor $2 \log^{(d+1)/2} m$.

3.5.4 Re-Partitioning Triggers

A key contribution of JanusAQP is continuous re-optimization of the partitioning. We describe how JanusAQP tracks the variances of the current partitions and decides when to re-partition. In addition, we propose two ways to re-partition our index, either partially or entirely.

Assume that the current partitioning is $\mathcal{R}$ and let $\mathcal{M}(\mathcal{R})$ be the (approximate) maximum variance query with respect to the current set of samples S. The automatic procedure first checks the number of samples in each bucket (leaf node) of the current $\mathcal{T}$. If there is a leaf node i with associated with partition R_i such that $|S_i| << \frac{1}{\alpha}\log m$ (recall that α is the sampling rate) then there are not enough samples in u to make robust estimators. Hence, we need to find a new re-partition of S. Even if the number of samples in each bucket is large our system might enable a re-partition: For a partition R_i in the leaf node layer of $\mathcal{T}$ let $\mathcal{M}_i = \mathcal{M}(R_i)$ be the (approximate) maximum variance at the moment we constructed $\mathcal{T}$. Let $\beta > 1$ be a parameter that controls the maximum allowable change on the variance. It can either be decided by the user or we can set it to $\beta = 10$. Assume that an update occurred in the leaf node associated with the partition R_i. After the update we run the function $\mathcal{M}_i' = \mathcal{M}(R_i)$ and we update $\mathcal{M}(\mathcal{R})$ if needed. If $\frac{1}{\beta}\mathcal{M}_i \leq \mathcal{M}_i' \leq \beta\mathcal{M}_i$ then the new maximum variance in partition b_i is not very different than before so we do not trigger a re-partition. Otherwise, the maximum variance in bucket b_i changed by a factor larger than β from the initial variance $\mathcal{M}_i$. In this case a re-partition might find a new tree with smaller maximum error. As we describe in the next paragraph, we compute a new partitioning $\mathcal{R}'$ and hence a new tree $\mathcal{T}$. If $\mathcal{M}(\mathcal{R}') < \frac{1}{\beta}\mathcal{M}(\mathcal{R})$ then we activate a re-partition restarting the catch-up phase over the new tree $\mathcal{T}$. On the other hand, if $\mathcal{M}(\mathcal{R}') \geq \frac{1}{\beta}\mathcal{M}(\mathcal{R})$ then our current partitioning $\mathcal{R}$ is good enough (its worst error is close to the optimum one) so we can still use it. Of course, the user can also manually trigger re-partitioning. For example, the user can choose to re-partition once every hour, day, or after τ insertions and deletions have occurred.

Next, we propose two ways to re-partition the index. In particular, the user can select either partial re-partitioning or entire re-partitioning. Entire re-partitioning is easy; using the algorithms from the previous subsections we can construct a new partitioning and a new tree structure in near-linear time with respect to the samples. Hence, we focus on partial re-partitioning. Instead of re-partitioning the entire space we can only re-partition the area around the "problematic" leaf node. Let b_i be this leaf node. In order to define the neighboring area around b_i we propose either a predefined way or an automatic way. In both cases, the neighboring area is defined by a parameter ψ, which defines the level of the tree above b_i that the tree needs to change. In the predefined way, the parameter ψ is a known parameter. We find the node v which is defined as an ancestor of the leaf node b_i, ψ levels above b_i. Let $\mathcal{T}_u$ is the subtree with root node u and let l_u be the number of leaf nodes in $\mathcal{T}_u$. Using the algorithms from the previous subsections we find a near optimum partition starting from node u with l_u leaf nodes. The running is near-linear with respect to the samples stored in the leaf nodes of $\mathcal{T}$. In the automatic way, we do not know the parameter ψ upfront so we try different values of ψ running a binary search on the levels of the tree until we find a partition with low enough error. For each different value of ψ we try, we run the same partial re-partitioning algorithm as in the static case starting from the node u we are considering in the binary search. Notice that if the automatic partial re-partitioning does not find a good enough partitioning, it might reconstruct the entire tree. In the worst case the running time a=of the automatic partial partitioning is $\tilde{O}(m \log(\text{height}(\mathcal{T})))$, however in practice it usually runs much faster than the entire re-partitioning.

Generally, partial re-partitioning is faster than the entire re-partitioning since we only suffices to find a better partitioning around one leaf node. Furthermore, in partial re-partitioning we can still keep all the current estimations in all nodes of $\mathcal{T} \setminus \mathcal{T}_u$, i.e., the nodes of the tree we do not change. Hence, the error of queries after a partial re-partitioning is also lower than the error of the queries immediately after an entire re-partitioning. However, in both cases we need to restart the catch-up phase over the new tree. Recall that we cannot get samples from a particular area (ideally samples that stored in the leaf nodes of $\mathcal{T}_u$) hence we run the catch-up phase getting samples from the entire space. We finally note that while the catch-up phase considers samples from the entire space, we only use these samples to improve the estimators in the nodes that are still under-represented, i.e., the catch-up phase time threshold for these nodes has not been completed.

3.6 Experiments

Next, we evaluate JanusAQP on different real-world datasets and query workloads. We run our experiments on a Linux machine with an Intel Core i7-8700 3.2GHz CPU and 16GB RAM.

3.6.1 Setup

To set up each experiment, we select a single aggregate attribute and one or more predicate attributes. We generate query workloads of 2000 queries by uniformly sampling from rectangular range queries over the predicates. We then initialize a JanusAQP instance with a user-specified sample rate, a catch-up ratio and a number of leaf nodes of the partition tree to compare with other baselines (these parameters directly control the Throughput, Query Latency, and Storage Size).

Datasets

Intel Wireless dataset. The Intel Wireless dataset [116] contains 3 million rows of sensor data collected in the Berkeley Research lab in 2004. Each row contains measurements like humidity, temperature, light, voltage as well as the date and time each record was collected.

New York Taxi Records dataset. The New York City Taxi Trip Records dataset [129] contains 7.7 million rows of yellow and green taxi trip records collected in January 2019. Each record contains information about the trip including `pickUpDateTime`, `dropOffDateTime`, `tripDistance`, `dropOffLocation`, `passengerCount`, etc.

NASDAQ ETF Prices dataset. The NASDAQ Exchange Traded Fund (ETF) Prices dataset[108] contains 2166 ETFs traded in the NASDAQ exchange from April 1986 to April 2020. There are 4 million entries in the dataset and each entry contains the date, the volume of transactions of an ETF on the date, and 4 prices: the price of an ETF when the market opens and closes; the highest and the lowest of its daily price range.

Metrics and ground truth

In terms of performance, we report the wall-clock latency and the throughput, i.e. number of requests (query/data) processed per second. To measure the accuracy of the system, unless otherwise specified, we report the 95 percentile of the relative error which is the difference between ground truth and estimated query result divided by the ground truth. We define the ground truth to be w.r.t all the tuples available when the query arrives, i.e. the true results reflect all insertions and deletions up to its arrival point. With this setup, the results indeed depend on the sequence of requests that are processed by the system. To make sure our experiments are deterministic, we fix this sequence up-front and ensure they are the same for each baseline.

Baselines

We evaluate the following baselines. All of these baselines are tuned to roughly control for query latency.

- **Reservoir Sampling (RS).** We construct a uniform sample of the entire data set which is maintained using the reservoir sampling algorithm [130]. We use a variant of RS first designed for the AQUA system that handles both insertions and deletions [49][4]. Unless otherwise noted, we use a 1% sample of data.
- **DeepDB.** We also compare with a machine learning-based baseline called DeepDB[56]. DeepDB achieves state-of-the-art AQP results in the static setting, and we chose it as a baseline since it has limited support for dynamic data. In our baseline, DeepDB trains on 10% of the data. We set this to be equivalent to the "catch-up" sampling in the DPT construction.
- **Dynamic Partition Tree-Only (DPT).** We also compare with a baseline of only using a single DPT synopsis without online optimization. This synopsis is constructed

4. Due to its age, a direct comparison with AQUA was not feasible

once and then used for the duration of the experiment. Unless otherwise noted there are 128 leaf nodes in a balanced binary tree, the leaf nodes as associated with 1% samples of their respective strata, and the catch-up sampling rate 10% of the data.

- **JanusAQP.** Finally, we evaluate the full-featured JanusAQP system. This includes a DPT and also performs re-partitioning if needed. Unless otherwise noted there are 128 leaf nodes in a balanced binary tree, the leaf nodes as associated with 1% samples of their respective strata, and the catch-up sampling rate 10% of the data.

The storage costs of the baselines on the NYC Taxi dataset given the typical setting (128 leaf-nodes, 10% catch-up rate, and 1% sample rate) are the following, reservoir sampling baseline takes about 5MB, JanusAQP and DPT takes about 6MB, a DeepDB baseline trained with 10% of the data is about 60MB.

3.6.2 Accuracy

We first evaluate the end-to-end performance of JanusAQP and the baselines on the NYC Taxi dataset on a 1d problem (1 predicate attribute). We use the `pickUpTime` attribute as the predicate attribute and the `tripDistance` attribute as the aggregate attribute. We start with 10% of the data in Kafka which is used by the baselines for initialization (simulating historical data). We incrementally add 10% more data in increments (simulating new data arrival). After every 10% increment, we re-train the model for DeepDB and re-initialize the DPT used by JanusAQP.

Figure 3.5(left) plots the accuracy as a function of the insertions. We can see that JanusAQP has the overall best accuracy while controlling for latency. We note that the accuracy of DeepDB is stable as a function of progress. This is because as a learned model DeepDB has a roughly fixed resolution of the data (it does not increase the number of parameters as more data is inserted). These findings are consistent with results from [87]. The accuracy of RS improves because of the increasing reservoir size (1% sample of the entire dataset). In the right plot of Figure 3.5, we show the re-optimization time cost in seconds by JanusAQP and DeepDB. The cost to initialize JanusAQP increases with the number of tuples stored in Kafka but it is still much cheaper than DeepDB. It is worth noting that the re-optimization cost of DeepDB is the cost of re-training instead of incremental training. This is mostly due to the constraint of the API exposed by DeepDB, and we observe that re-train a model with $2n$ samples is faster than train a model with n samples then incrementally train another n samples. The results suggest that complex, learned synopses are not ideal in the dynamic setting.

3.6.3 Performance

Next, we evaluate the throughput and query latency of JanusAQP. We populate Kafka with the first p percent of the NYC Taxi dataset (p varies from 10 to 90). Like before, we initialize JanusAQP on the first 10% of data and then incrementally add increments of 10% more. In this experiment, we construct a mixed update workload of both insertions and deletions. Results can be found in Figure 3.6, on the left plot we show the throughput of handling insertions and deletions using a pool of 12 threads. We can see the performance of JanusAQP is quite stable and does not change with the size of existing data or the amount of data that have been processed. For each insertion and deletion, we just need to find the target node in $O(\log(k))$ to modify the summary. Even though a larger reservoir size increases the overhead of manipulating the samples for reservoir sampling, the increased overhead is unnoticeable. This is because the stratum stored in each node is $\frac{1}{k}$ of the reservoir, and each stratum is independent with others and race condition only happens if two workers are working on the same node. On the right plot Figure 3.6, we compare the query latency of JanusAQP with DeepDB, we can see the query latency of JanusAQP is lower than DeepDB, this is because JanusAQP combines aggregates with sampling. When possible, JanusAQP uses the aggregates that are more accurate instead of the samples which are the main sources of overhead and error.

3.6.4 Work with Real-life Dynamic Data

With YCSB benchmark. TODO

3.6.5 Partial Repartition

Instead of repartition from scratch, we find the smallest subtree that needs to be repartitioned and re-optimize the subtree. TODO. Idea is to reconstruct the subtree then perform catch-up to improve the subtree stats of the internal node.

3.6.6 The Catch-up Phase

One of the unique designs of JanusAQP is the 'catch-up' phase which enables the system to work with an arbitrarily large amount of existing data. In this experiment, we want to understand how the catch-up phase can impact the accuracy and performance of the entire system.

Accuracy

We use the entire Intel wireless dataset as the existing data. We compare a set of JanusAQP (128, c, 1%) instances where the catch-up goal c varies from 1% to 10% with a step of 1%. When each JanusAQP instance reaches the catch-up goal, we use it to evaluate the same set of 2000 random queries generated using the light attribute as the aggregate attribute and the time attribute as the predicate attribute.

The results can be found in the left plot of Figure 3.7. As a reference, we also show the accuracy of an RS baseline with 1% sample rate. We notice that JanusAQP (128,1%,1%) has no advantage against the RS baseline because neither the samples nor the summaries built during catch-up could provide better accuracy. As we increase the catch-up ratio, we can see an improvement in accuracy because the quality of the summaries built by the catch-up phase improved. Comparing with the expensive offline pre-processing used in [87], we believe the catch-up phase is a better alternative that provides another knob to tune the tradeoff between accuracy and cost.

Overhead

The overhead of the catch-up phase comes from two sources: the loading and processing of the samples. We distinguish and measure the two types of overhead in terms of their time cost. Data loading time measures the time spent on calling the Kafka `poll()` API, transferring the data, and ETL operations that are necessary to prepare the data for JanusAQP to process. It is worth noting that the data loading cost is part of the essential cost that occurs in all systems and is usually less relevant to the core design of the system but more relevant to the design of interfaces. For example, with a different interface, instead of dealing with the strings from Kafka that can be expensive to parse, the system could use Protocol Buffers[51] for more efficient data exchanging or even offload some of the ETL duties to the client-side as described in [39]. On the other hand, the data processing time stands for the time taken by JanusAQP to analyze the data then accordingly modify internal data structures that will be used for query processing.

Results can be found in the right plot of Figure 3.7, we can see that the data processing with a single thread takes less than 1.5 seconds for a catch-up ratio of 10%, which is equivalent to a throughput of processing 160,000 tuples per second. Furthermore, we can see that the data loading cost is much higher than the data processing cost and we believe the data loading cost can be further improved by more engineering efforts and techniques such as client-assisted data loading[39].

3.6.7 Multi-dimensional Query Templates

In this experiment, we investigate the performance of JanusAQP with multi-dimensional queries on the NASDAQ ETF Prices dataset. We randomly generate 2000 queries from a 5-D query template that uses the volume attribute as the target attribute, the date attribute and the 4 price attributes as predicate attributes. We perform the same workflow as we did in Section 3.6.2. We first compare the median relative error of JanusAQP (256,10%,1%) with DeepDB and the results can be found in the left plot of Figure 3.8. We notice that the accuracy of JanusAQP is better than DeepDB but the relative error increases for both. This is because multi-dimensional queries are usually more selective. Also, because the queries are generated using the entire dataset, we notice that many of the ground truths generated using the first 20% of the data are 0s. Therefore, in the experiment, we start with 30% of the data. On the right plot of Figure 3.8, we can find the re-optimization cost of JanusAQP is lower than DeepDB but is more expensive than in the 1D setting. While the increase of dimensions can indeed make it more expensive to process the samples we fetched during catch-up, we believe the re-optimization cost can be further improved with more engineering efforts.

3.6.8 Re-partitioning

Next, we consider microbenchmarks that evaluate the re-partitioning optimizations in JanusAQP. In the first experiment, using the NYC Taxi dataset, JanusAQP performs a periodic re-partitioning after every 10% insertions. For comparison, the DPT baseline does not perform any re-partitioning and we evaluate the accuracy. We deliberately skew the insertions by sorting on `pickUpDateTime` so that new insertions would hit a small number of partitions. The results are illustrated in Figure 3.9(left), we can see the relative error of DPT increases drastically due to a partition tree that becomes more and more imbalanced with new insertions. With periodic re-partition, JanusAQP keeps the accuracy at a controlled level.

In the second experiment, we use the `pickupTimeOfDay` as the predicate attribute. Because the dataset is randomly distributed over the `pickupTimeOfDay` attribute, the insertions are not skewed as in the previous setting. To demonstrate a situation where a re-partition is triggered by deletions, we randomly choose 10% of the nodes and we randomly delete half of the samples that belong to these nodes then we insert the next 10% data. After the insertion, the re-partition will be triggered for JanusAQP. For comparison, we use a DPT baseline that does not perform any re-partition. We perform the same operations to the leaf nodes of the DPT baseline then we evaluate the same set of queries. The results can be found in the right plot of Figure 3.9, we can see the relative error of DPT increases due to the imbalanced partition tree while the error of JanusAQP drops because of re-partition.

3.6.9 Comparing with PASS

As mentioned earlier, we borrow the idea of combining aggregates with sampling from PASS[87] and propose a couple novel contributions that enable JanusAQP to handle insertions and deletions. Because PASS is not designed to work with dynamic data, it is not end-to-end comparable to JanusAQP and we study the main contributions that enable JanusAQP to work with dynamic data in previous experiments.

In this experiment, we focus on the partitioning algorithms of the two systems. In Section 3.5, we propose a binary search-based partitioning algorithm for 1 dimension that is much more efficient. We compare the accuracy and time cost of the BS-based algorithm with the dynamic programming-based partitioning algorithm used by PASS on the Intel Wireless dataset. We implement the BS-based algorithm in Python in our code base of PASS for a fair comparison. We measure the time cost in seconds of each partitioning algorithm given different number of partitions, we also compare the median relative error of the PASS variation over 2000 randomly generate queries.

The result can be found in Table 3.2, we vary the number of partitions from 16 to 128, as we increase the number of partitions, the sample size used by the algorithms also increase. We notice that the time cost of the DP-based algorithm increase drastically[5] with the number of partitions while the time cost of the BS-based algorithm increase slightly. On the accuracy side, the DP-based algorithm does lead to a lower error but the BS-based

5. Because we use a larger sample size than what we used in [87], the time cost of the DP algorithm increases and the accuracy improves from what we reported in [87].

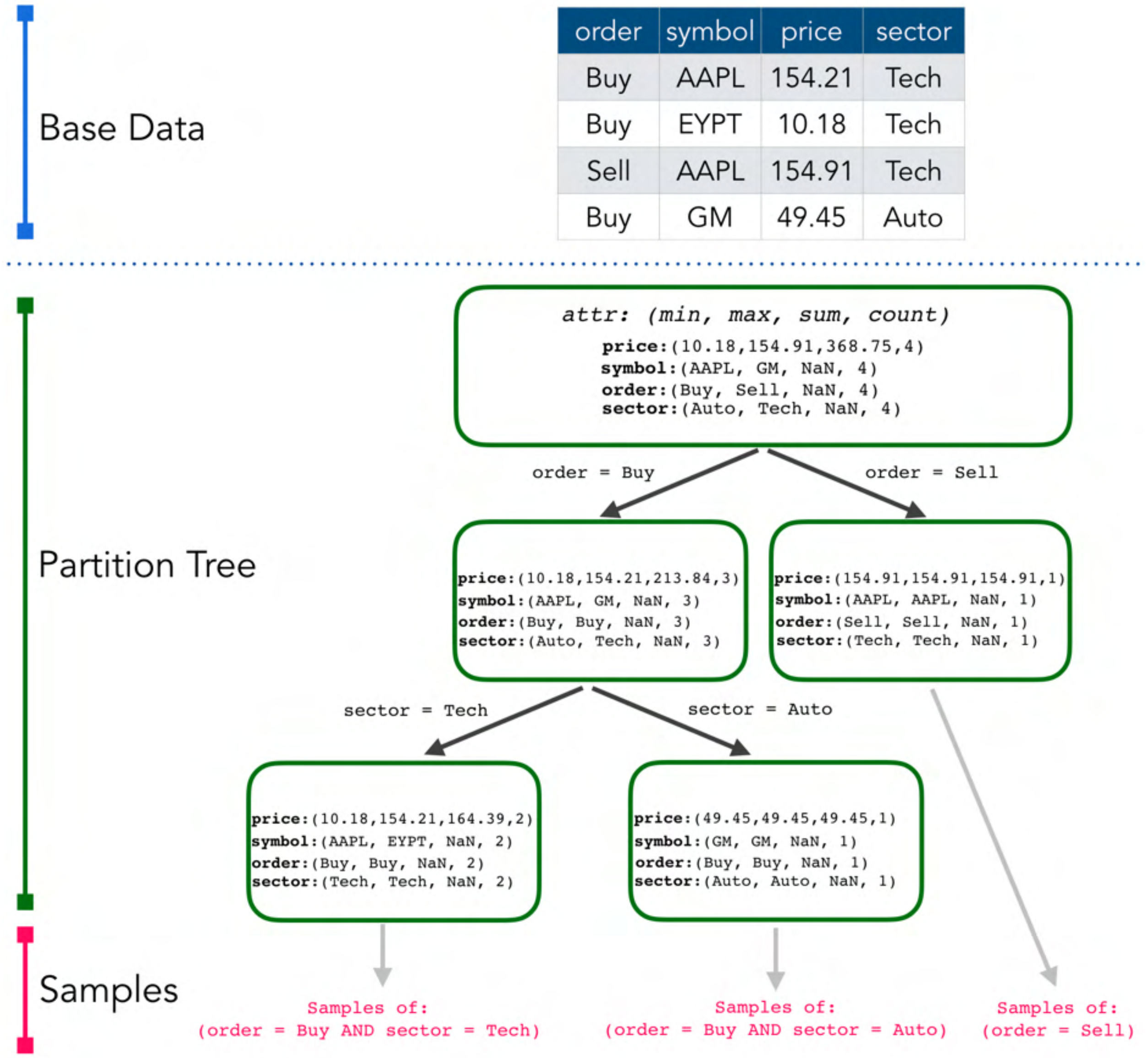
		16	32	64	128
Partition Time (s)	DP	16	22	382	6349
	BS	0.3	0.3	0.4	1.6
Median RE (CNT)	DP	0.2%	0.1%	0.05%	0.04%
	BS	0.6%	0.4%	0.1%	0.1%
Median RE (SUM)	DP	0.2%	0.1%	0.07%	0.05%
	BS	1%	0.9%	0.2%	0.2%
Median RE (AVG)	DP	0.2%	0.1%	0.08%	0.05%
	BS	1%	0.7%	0.2%	0.15%

Table 3.2: We compare our new binary search-based (BS) partitioning algorithm with the dynamic programming-based (DP) algorithm proposed by PASS[87] on the Intel Wireless dataset.

algorithm also introduce good accuracy. Overall, we believe the BS-based algorithm is more scalable than the DP-based algorithm and it provides favorable trade-off between cost and accuracy.

3.7 Discussion and Conclusion

Approximate query processing over dynamic databases has applications ranging from high frequency trading to internet-of-things analytics. We present JanusAQP, a new dynamic AQP system, which supports SUM, COUNT, AVG, MIN, and MAX queries under insertions and deletions to the dataset. The initial results are highly promising, and this basic system has many interesting avenues for future work. We are particularly interested in physically partitioned DPT synopses. This is possible because the leaf nodes of the dynamic partition tree are disjoint and we can further split a tree into two trees which are in charge of different domains. Each DPT can be run in a different node as a standalone service and we can use a router to forward each request to the right tree for processing. We leave the implementation of a JanusAQP cluster as a future work.


Base Data
Partition Tree
Samples
attr: (min, max, sum, count)
price:(10.18,154.91,368.75,4)
symbol:(AAPL, GM, NaN, 4)
order:(Buy, Sell, NaN, 4)
sector:(Auto, Tech, NaN, 4)
order = Buy
order = Sell
price:(10.18,154.21,213.84,3)
symbol:(AAPL, GM, NaN, 3)
order:(Buy, Buy, NaN, 3)
sector:(Auto, Tech, NaN, 3)
price:(154.91,154.91,154.91,1)
symbol:(AAPL, AAPL, NaN, 1)
order:(Sell, Sell, NaN, 1)
sector:(Tech, Tech, NaN, 1)
sector = Tech
sector = Auto
price:(10.18,154.21,164.39,2)
symbol:(AAPL, EYPT, NaN, 2)
order:(Buy, Buy, NaN, 2)
sector:(Tech, Tech, NaN, 2)
price:(49.45,49.45,49.45,1)
symbol:(GM, GM, NaN, 1)
order:(Buy, Buy, NaN, 1)
sector:(Auto, Auto, NaN, 1)
Samples of: (order = Buy AND sector = Tech)
Samples of: (order = Buy AND sector = Auto)
Samples of: (order = Sell)

order	symbol	price	sector
Buy	AAPL	154.21	Tech
Buy	EYPT	10.18	Tech
Sell	AAPL	154.91	Tech
Buy	GM	49.45	Auto

Figure 3.2: The core data structure in JanusAQP is based on the PASS data structure [87] that summarizes a dataset with a tree of aggregates at different levels of resolution (granularity of partitioning). Associated with the leaf nodes are stratified samples. The two stage synopsis structure can be optimally partitioned to minimize error.

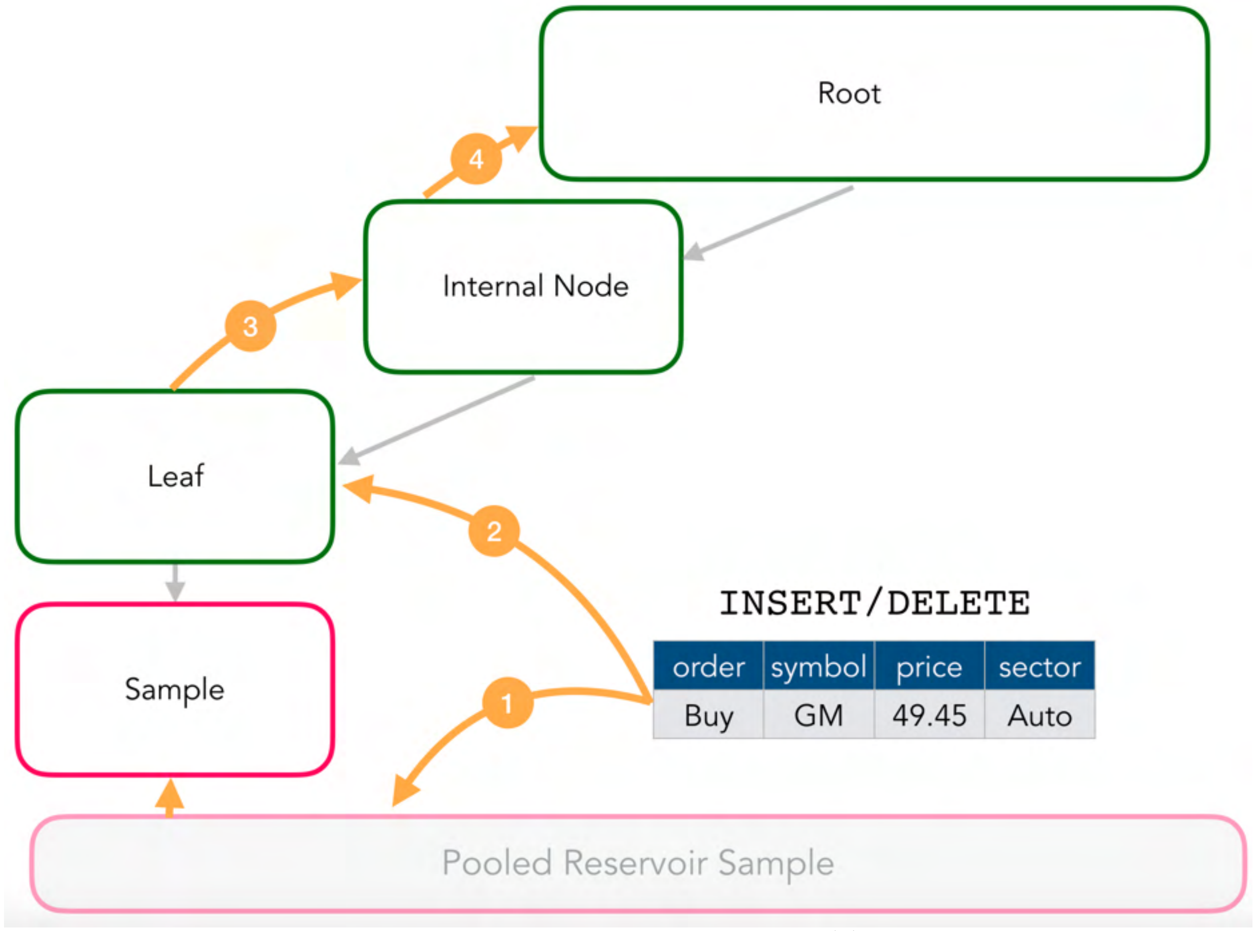


Figure 3.3: The DPT update process for an insertion or deletion. (1) A set of samples is maintained using a reservoir sampling algorithm. (2) The leaf node statistics are incrementally updated. (3) The updated statistics from the leaf node propagate to the parents. (4) Updated statistics from the parents propagate all the way to the root.

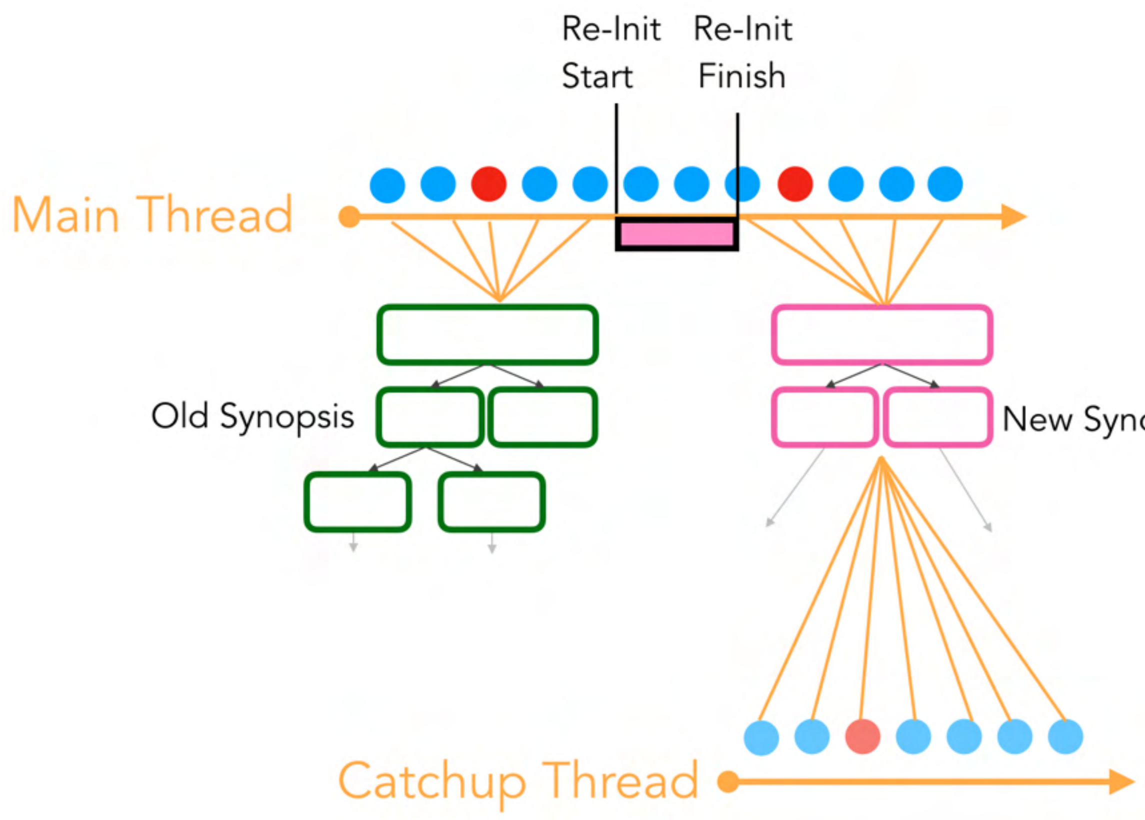


Figure 3.4: JanusAQP synopses can be re-initialized online using a multi-threaded implementation to minimize unavailability

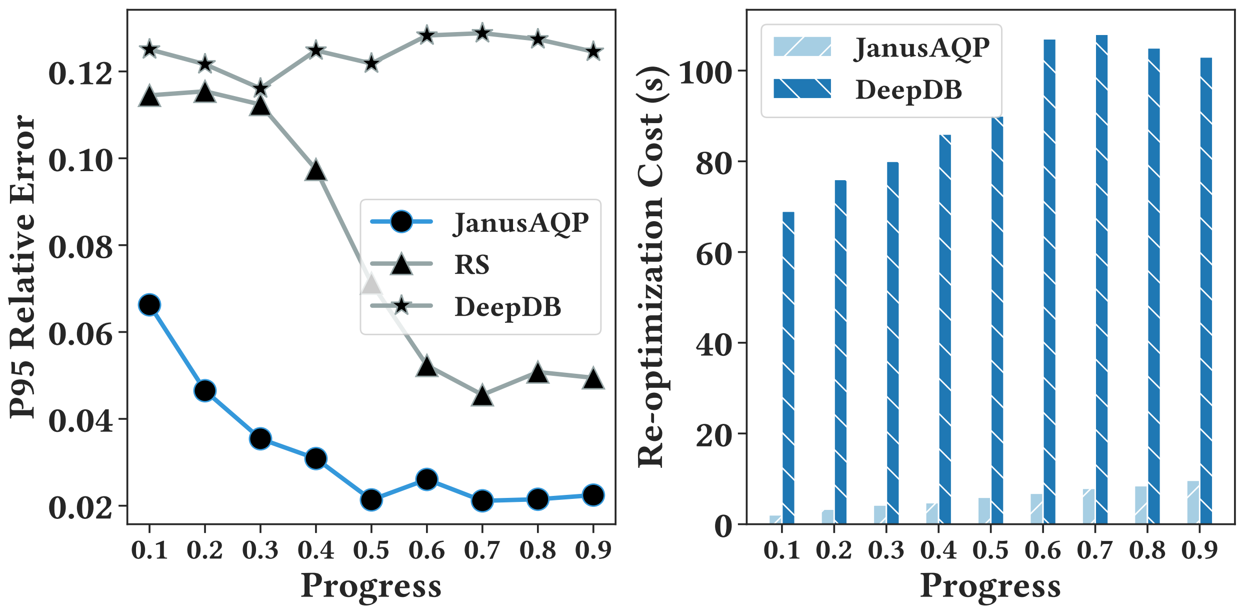


Figure 3.5: We compare the accuracy (left plot) and re-optimization cost (right plot) of JanusAQP with DeepDB and reservoir sampling on the NYC Taxi dataset.

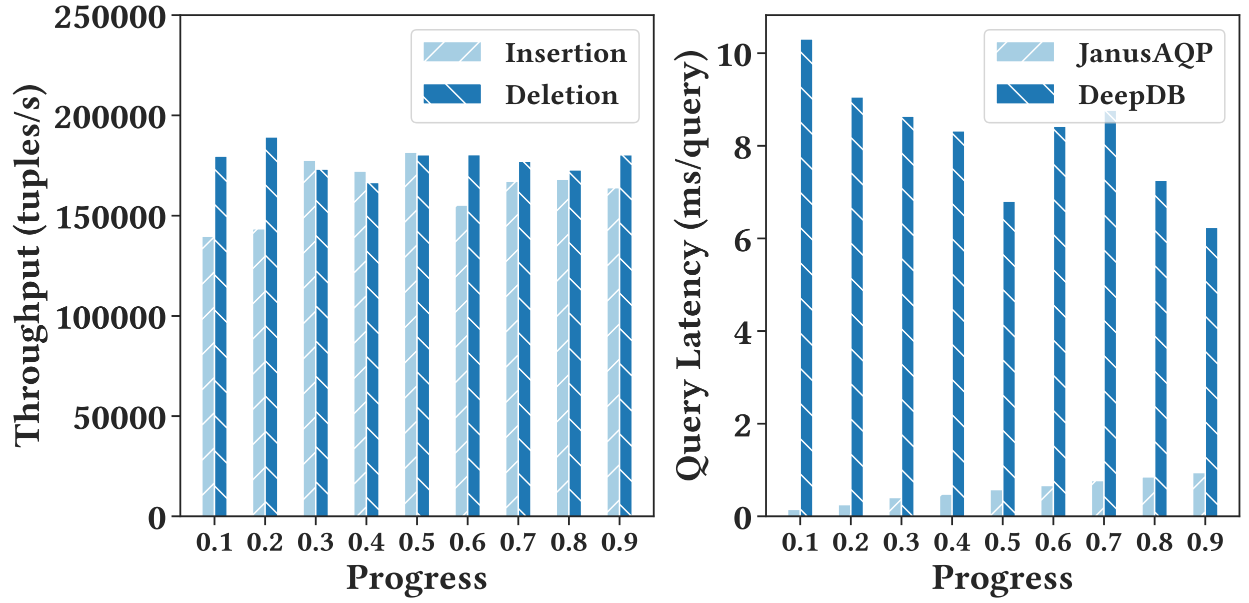


Figure 3.6: We evaluate the throughput (left plot) of JanusAQP when handling insertions and deletions in multi-thread mode. We also compare the query latency (right plot) with DeepDB.

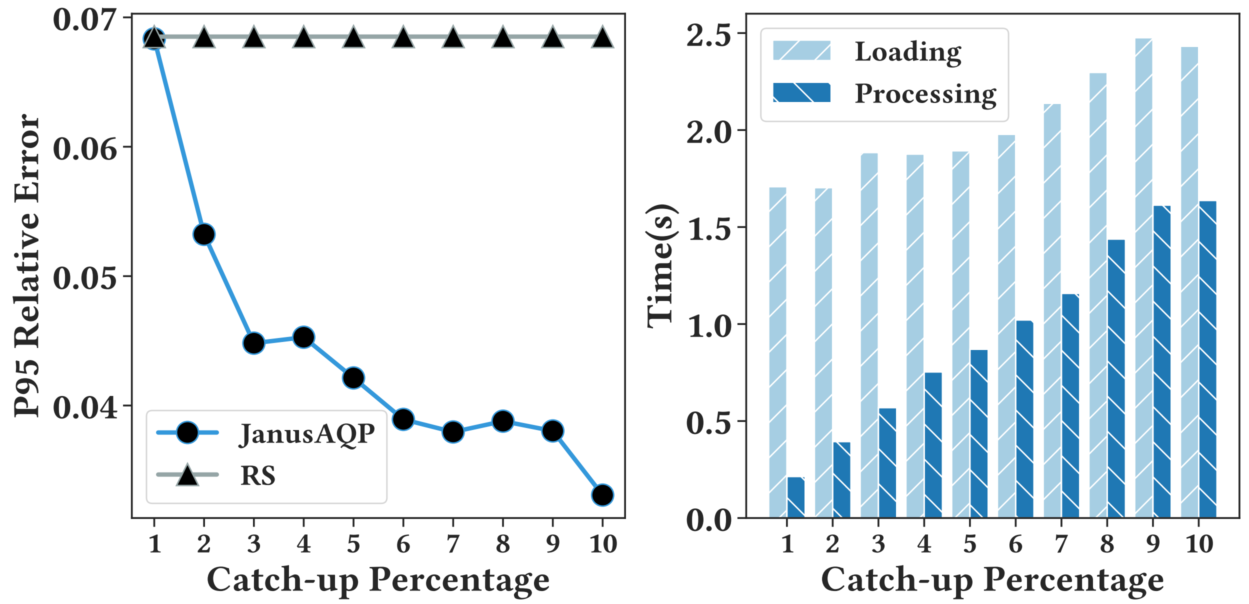


Figure 3.7: Varying the catch-up goal from 1% to 10% of the data, we evaluate the accuracy of JanusAQP (left plot) and the time cost of the catch-up phase (right plot).

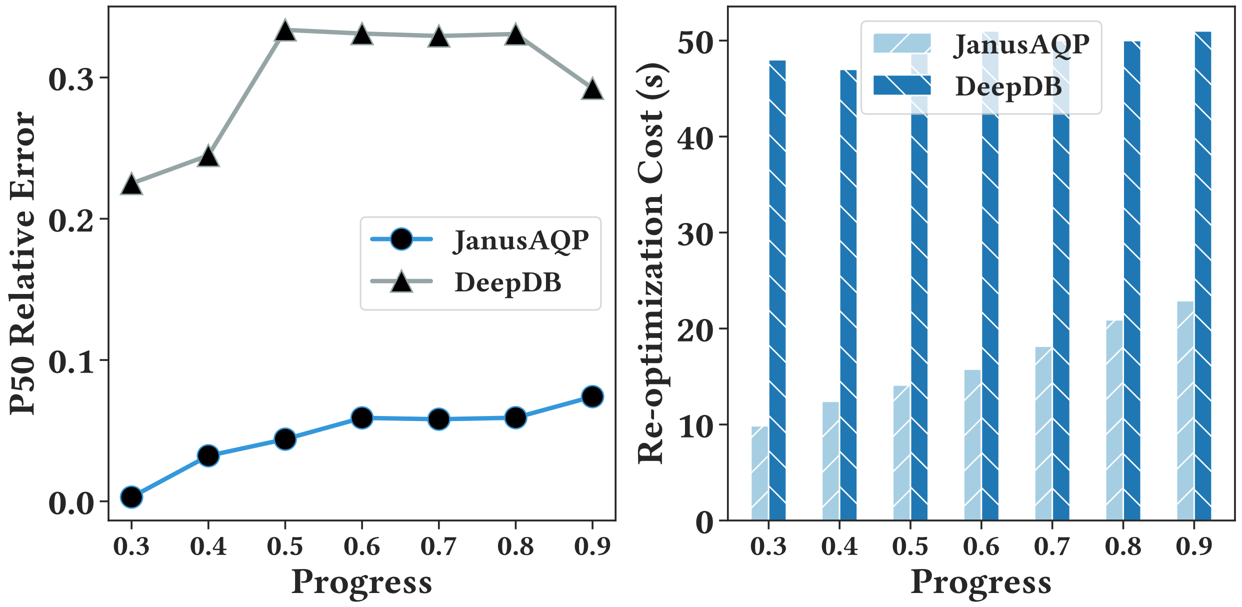


Figure 3.8: We compare JanusAQP with DeepDB on multi-dimensional queries. Both DeepDB and JanusAQP perform re-optimization after every 10% insertion then evaluate the same set of queries. We record the median relative error on the left plot and the re-optimization cost on the right plot.

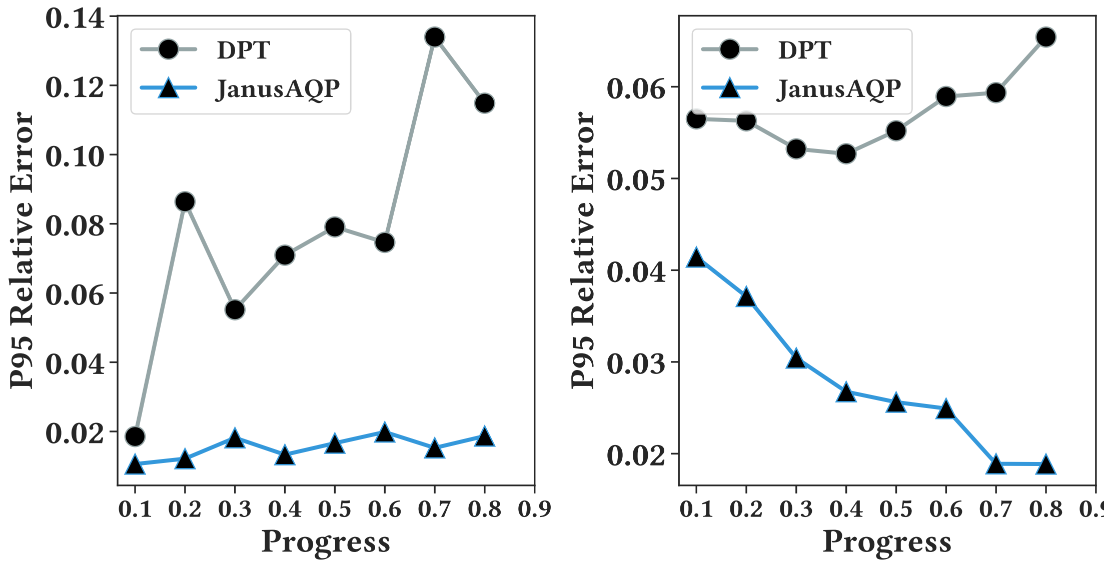


Figure 3.9: We compare the accuracy of JanusAQP and DPT in two scenarios that cause imbalanced partition trees.

CHAPTER 4
FAST AND RELIABLE MISSING DATA CONTINGENCY ANALYSIS WITH PREDICATE-CONSTRAINT

4.1 Introduction

The data stored in a database may differ from real-world truth in terms of both completeness and content. Such issues can arise due to data entry errors, inexact data integration, or software bugs [28]. As real-world data are rarely perfectly clean or complete, data scientists have to reason how potential sources of error may affect their analyses. Communicating these error modes and quantifying the uncertainty they introduce into a particular analysis is arguably as important as timely execution [75].

For example, suppose a data analyst has collected data from a temperature sensor over the span of several days. She is interested in computing the number of times that the sensor exceeded a temperature threshold. The data are stored in 10 partitions; one of which failed to load into the database due to parsing errors. The analyst can still run her query on the 9 available partitions, however, she needs to determine whether the loss of that partition may affect her conclusions.

Today, analysts largely rely on intuition to reason about such scenarios. The analyst in our example needs to make a judgment about whether the lost partition correlates with the attributes of interest, such as temperature, in any way. Such intuitive judgments, while commonplace, are very problematic because they are based on assumptions that are often not formally encoded in any code or documentation. Simply reporting an extrapolated result does not convey any measure of confidence in how (in)accurate the result might be, and could hide the fact that some of the data were not used.

This paper defines a framework for specifying beliefs about the missing rows in a dataset in a logical constraint language and an algorithm for computing a range of values an aggregate query can take under those constraints (hereafter called a result range).[1] This framework, which we call the Predicate-Constraint (PC) framework, facilitates several desirable outcomes: (1) the constraints are efficiently testable on historical data to determine whether or not they held true in the past, (2) the result range is calculated deterministically and guaranteed to bound the results if the constraints hold true in the future, (3) the framework can reconcile interacting, overlapping, or conflicting constraints by enforcing the most restrictive ones, and (4) the framework makes no distributional assumptions about past data resembling future data other than what is specified in the constraints. With this framework, an analyst can automatically produce a contingency analysis, i.e., the range of values the aggregate could take, under formally described assumptions about the nature of the unseen data. Since the assumptions are formally described and completely determine the result ranges, they can be checked, versioned, and tested just like any other analysis code—ultimately facilitating a more reproducible analysis methodology. The constraints themselves, called Predicate-Constraints, are logical statements that constrain the range of values that a set of rows can take and the number of such rows within a predicate. We show that deriving the result ranges for a single "closed" predicate-constraint set can be posed as a mixed-integer linear program (MILP). The solver itself contains a number of novel optimizations, which we contribute such as pruning of unsatisfiable search paths.

To the best of our knowledge, a direct competitor framework does not exist. While there is a rich history of what-if analysis [37] and how-to analysis [98], which characterize a database's behavior under hypothetical updates, analyzing the effects of constrained missing rows on aggregate queries has been not been extensively studied. The closest framework is the m-table framework [126], which has a similar expressiveness but no algorithm for computing aggregate result ranges. Likewise, some of the work in data privacy solves a simplified version of the problem where there are no overlapping constraints [137]. In summary, we contribute:

1. A formal framework for contingency analysis over missing or withheld rows of data,

1. We use this term to differentiate a deterministic range with probabilistic confidence intervals.

where users specify constraints about the frequency and variation of the missing rows.

2. An optimization algorithm that reconciles a set of such constraints, even if they are overlapping, conflicting, or unsatisfiable, into a range of possible values that SUM, COUNT, AVG, MIN, and MAX SQL queries can take.

3. Optimization that improve accuracy and/or optimization performance such as pruning unsatisfiable constraint paths.

4.2 Background

In this paper, we consider the following user interface. The system is asked to answer SQL aggregate queries over a table with a number of missing rows. The user provides a set of constraints (called predicate-constraints) that describes how many such rows could be missing and a range of possible attribute values those missing rows could take. The system should integrate these constraints into its query processing and compute the maximal range of results (aggregate values) consistent with those constraints.

All of our queries are of the form:

```
SELECT agg(attr)
FROM R
WHERE ....
GROUP BY ....
```

We consider SUM, COUNT, AVG, MIN, and MAX aggregates with predicates. Because GROUP-BY clause can be considered as a union of such queries without GROUP-BY. In the rest of the paper, we focus on queries without GROUP-BY clause.

4.2.1 Example Application

Consider a simplified sales transaction table of just three attributes:

```
Sales(utc, branch, price)
Nov-01 10:20,New York,3.02
Nov-01 10:21,Chicago,6.71
...
Nov-16 6:42,Trenton,18.99
```

Over this dataset, a data analyst is interested in calculating the total number of sales:

```sql
SELECT SUM(price)
FROM Order
```

Suppose that between November 10 and November 13 there was a network outage that caused data from the New York and Chicago branches to be lost. How can we assess the effect of the missing data on the query result?

Simple Extrapolation: One option is to simply extrapolate the SUM for the missing days based on the data that is available. While this solution is simple to implement, it leads to subtle assumptions that can make the analysis very misleading. Extrapolation assumes that the missing data comes from roughly the same distribution as the data that is present. Figure 4.1 shows an experiment on one of our experimental datasets. We vary the mount of missing data in a way that is correlated with a SUM query. Even if the exact amount of missing data is known, the estimate become increasingly error prone. More subtly, extrapolation returns a single result without a good measure of uncertainty—there is no way to know how wrong an answer might be.

Better Extrapolation: A smarter approach might be to build a probabilistic model that identifies trends and correlations in the data (e.g., a Gaussian Mixture Model that identifies weekly patterns) and use that model to extrapolate. If a user mis-specifies her belief in the data distribution or sampling process, any inference would be equally fallible as simple extrapolation. The probabilistic nature of the inference also makes potential failure models hard to interpret—errors could arise due to modeling, sampling, or even approximation error

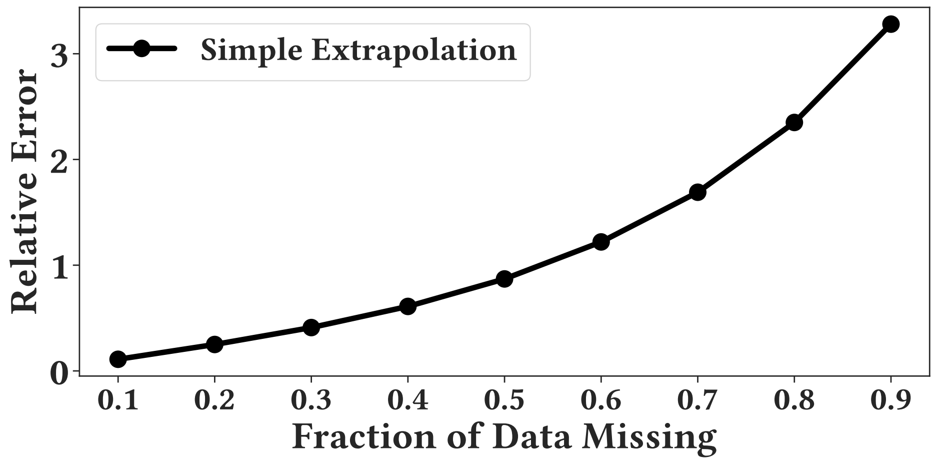


Figure 4.1: Simple Extrapolation could introduce significant error when missing rows are correlated (e.g, tend to have the highest values).

in the model fitting process.

Our Approach: In light of these issues, we propose a fully deterministic model for quantifying the uncertainty in a query result due to missing data. Like the probabilistic approach, we require that the user specify her belief about the distribution of missing data. Rather than specifying these beliefs in terms of probability distributions, she specifies the beliefs in terms of hard constraints.

For example, there are no more than 300 sales each day in Chicago. Or, the most expensive product costs 149.99 and no more than 5 are sold each day. We collect a set of such constraints and solve an optimization problem to find the maximal sum possible for all missing data instances that satisfy the constraints. This formalism acts as a programming framework that can be used to test the effects of different scenarios. It crucially enforces that there are testable constraints that are recorded during the decision making process. We will use this as an example throughout the paper.

4.2.2 Related Work

The overarching challenge addressed in the PC framework is related to the concept of "reverse data management" proposed by Meliou et al. [98, 99]. Meliou et al. argue that as data grow in complexity, analysts will increasingly want to know not what their data currently says but what changes have to happen to the dataset to force a certain outcome. Such *how-to* analyses are useful in debugging, understanding sensitivity, as well as planning for future data. Meliou et al. build on a long line of *what-if* analysis and data provenance research, which study simulating hypothetical updates to a database and understanding how query results might change [37, 17]. While we address similar concerns to this line of work in spirit, our focus on aggregate queries and confidence intervals leads to a very different set of technical contributions. The PC framework should be evaluated much more like a synopsis data structure than a data provenance reasoning systems.

Therefore, our experiments largely focus on evaluations against other synopsis structures and how to extract confidence intervals from them [32]. While Approximate Query Processing (AQP) has been studied for decades [107], it is well-known that the confidence intervals produced can be hard to interpret [71]. This is because estimating the spread of high dimensional data from a small sample is fundamentally hard, and the most commonly used Central-Limit Theorem-based confidence intervals rely on estimated sample variance. Especially for selective queries, these estimates can be highly fallible—a 95% confidence interval may "fail" significantly more than 5% of the time [5]. Unfortunately, as confidence intervals become more conservative, e.g., using more general statistical bounding techniques, their utility drops [55]. In a sense, our optimization algorithm automatically navigates this tradeoff. The algorithm optimizes the tightest bound given available information in the form of PCs. We interpret PCs as generalized histograms with overlapping buckets and uncertain bucket counts. Despite these differences with AQP, we do believe that the connections between uncertainty estimation and dirty data (like missing rows) are under-studied [76, 77]. We also believe that in future work mixed systems with both PCs and samples can have the best of both worlds, e.g., augmenting Quickr with PC estimation [70].

Deterministic alternatives to AQP have been studied in some prior work. Potti et al. propose a paradigm called DAQ [120] that does reason about hard ranges instead of confidence intervals. DAQ models uncertainty at relation-level instead of predicate-level like in PCs and DAQ does not handle cardinality variation. In the limited scenario of windowed queries over time-series data, deterministic bounds have been studied [15]. The technical challenge arises with overlapping constraints and complex query structures (like join conditions and arbitrary predicates). Similarly, we believe that classical variance reduction techniques for histograms could be useful for PC generation in future work [119], since histograms are a dense 1-D special case of our work.

c-tables are one of the classical approaches for representing missing data in a relation [57]. Due to the frequency constraints in Predicate-Constraint sets, we can represent cases that go beyond the typical closed-world assumption (CWA) is required in c-tables, where all records are known in advance and null cells are specifically annotated. There is also recent work that studies missing rows from databases. m-tables study variable cardinality representations to go beyond the CWA. In m-tables, cardinality constraints are specified per-relation. We specify frequency constraints per predicate. However, Sundarmurthy et al. [126] do not consider the problem of aggregate query processing on uncertain relations. There is similarly related work that studies intentionally withholding partitioned data for improved approximate query performance [127]. We believe that the novelty of our framework is the efficient estimation of aggregate query confidence intervals. Similarly, the work by Burdik et al. is highly related where they study databases with certain “imprecise” regions instead of realized tuples [18]. It is important to note, that our objective is not to build the most expressive language to represent uncertain data but rather one that we can pragmatically use to bound aggregate queries.

The privacy literature has studied a version of this problem: bounding aggregate queries on uncertain data [137, 65]. In fact, Zhang et al. can be seen as solving the partitioned version of our problem [137]. However, they do not need to consider the overlapping case in the way that our work does.

4.3 Predicate-Constraints

The formal problem setting is defined as follows. Let R be relation with a known "certain" partition denoted by R^* and unknown "missing" partition $R^?$, where $R = R^* \cup R^?$. The user defines a set of constraints $\pi_1, ..., \pi_n$ over the possible tuples that could be in $R^?$ and their multiplicity. The computational problem is to derive a result range, namely, the min and max value that one of the supported aggregate queries given all possible instances of $R^?$ that are valid under the constraints. This section describes our language for expressing constraints. Suppose, this relation is over the attributes $A = \{a_1, ..., a_p\}$. The domain of each attribute a_i is a set denoted by $\mathsf{dom}(a_i)$.

4.3.1 Predicate-Constraint

If $R^?$ could be arbitrary, there is clearly no way to bound its effect on an aggregate query. Predicate-constraints restrict the values of tuples contained in $R^?$ and the total cardinality of such tuples. A single predicate-constraint defines a condition over the $R^?$, for tuples that satisfy the condition a range of attribute values, and the minimum and maximum occurrence of this predicate. As an example predicate-constraint in the sales dataset described in the previous section, "the most expensive product in Chicago costs 149.99 and no more than 5 are sold". This statement has a predicate ("in Chicago"), a constraint over the values ("cost at most 149.99"), and occurrence constraint ("no more than 5"). We will show how to express systems of such constraints in our framework.

Predicate: A predicate ψ is a Boolean function that maps each possible rows to a True and False value $\psi : \mathcal{D} \mapsto \mathbb{Z}_2$. For efficient implementation, we focus on predicates that are conjunctions of ranges and inequalities. This restriction simplifies satisfiability testing, which is an important step in our algorithms introduced in Section 4.4.1.

Value Constraint: A value constraint specifies a set of ranges that each attribute can take on. A range of the attribute a_i is defined as a row of two elements $(l, h) \in \mathsf{dom}(a_i)$

where $l \leq h$. A value constraint ν is a set of ranges for each of the p attributes:

$$\nu = \{(l_1, h_1), ..., (l_p, h_p)\}$$

ν defines a Boolean function as above $\nu : \mathcal{D} \mapsto \mathbb{Z}_2$ that checks whether a row satisfies all the specified ranges. Since we focus on bounding aggregates it is sufficient to assume that the attribute ranges are over numerical attributes.

Frequency Constraint: Associated with each predicate is additionally a frequency constraint. This bounds the number of times that rows with the predicate appear. The frequency constraint is denoted as $\kappa = (k_l, k_u)$. k_l and k_u specify a range of the frequency constraint, i.e., there are at least k_l rows and at most k_u rows that satisfy the predicates in $R^?$. Of course, k_l, k_u must be non-negative numbers and $k_l \leq k_u$.

Predicate Constraint: A predicate-constraint is a three-tuple of these constituent pieces $\pi = (\psi, \nu, \kappa)$, a predicate, a set of value constraints, and a frequency constraint. The goal of π is to define constraints on relations that satisfy the above schema. Essentially a predicate constraint says if R is a relational instance that satisfies the predicate-constraint π **"For all rows that satisfy the predicate ψ, the values are bounded by ν and the number of such rows is bounded by κ"**. Formally, we get the definition below.

Definition 4.3.1 (Predicate Constraint)**.** A predicate constraint is a three-tuple consisting of a predicate, a value constraint, and a frequency constraint $\pi = (\psi, \nu, \kappa)$. Let R be a relational instance over the attributes A. R satisfies a predicate constraint denoted by $R \models \pi$ if:

$$(\forall r \in R : \psi(r) \implies \nu(r)) \quad \wedge k_l \leq |\{r \in R : \psi(r)\}| \leq k_u$$

Let us now consider several examples of predicate constraints using the sales data example in the previous section on two days worth of missing rows. For example, if we want to express the constraint "the most expensive product in Chicago costs 149.99 and no more than 5 are sold":

$$c_1 : (\text{branch} = \text{`Chicago'}) \implies (0.00 \leq \text{price} \leq 149.99),\ (0, 5)$$

In this example, the predicate is (branch = 'Chicago'), the value constraint is $(0.00 \leq \text{price} \leq 149.99)$ and the frequency constraint is (0, 5). If the aforementioned predicate constraint is describing a sales dataset, then it specifies that there are at most 5 tuples in the dataset with value of the *branch* attribute equals to 'Chicago' and the range of the *price* attribute of these tuples are between 0.0 and 149.99 (inclusive).

We could tweak this constraint to be "the most expensive product in ALL branches costs 149.99 and no more than 100 are sold":

$$c_2 : \text{TRUE} \implies (0.00 \leq \text{price} \leq 149.99),\ (0, 100)$$

Suppose one wanted to define a simple histogram over a single attribute based the number of sales in each branch, we could express that with a tautology:

$$(\text{branch} = \text{`Chicago'}) \implies (\text{branch} = \text{`Chicago'}),\ (100, 100)$$

$$(\text{branch} = \text{`New York'}) \implies (\text{branch} = \text{`New York'}),\ (20, 20)$$

$$(\text{branch} = \text{`Trenton'}) \implies (\text{branch} = \text{`Trenton'}),\ (10, 10)$$

Observe how $c1$ and $c2$ interact with each other. Some of the missing data instances allowed by only $c2$ are disallowed by c_1 (Chicago cannot have more than 5 sales at 149.99). This interaction will be the main source of difficulty in computing result ranges based on a set of PCs.

4.3.2 Predicate-Constraint Sets

Users specify their assumptions about missing data using a set of predicate constraints A predicate-constraint set is defined as follows:

$$\mathbf{S} = \{\pi_1, ..., \pi_n\}$$

S gives us enough information to bound the results of common aggregate queries when there is *closure*: every possible missing row satisfies at least one of the predicates.

Definition 4.3.2 (Closure)**.** Let **S** be a predicate constraint set with the elements $\pi_i = (\psi_i, \nu_i, \kappa_i)$ **S** is closed over an attribute domain $\mathcal{D}$ if for every $t \in \mathcal{D}$:

$$\exists \pi_i \in \mathbf{S} : \psi_i(t)$$

Closure is akin to the traditional closed world assumption in logical systems, namely, the predicate constraints completely characterize the behavior of the missing rows over the domain.

To understand closure, let us add a new constraint that says "the most expensive product in "New York" is 100.00 and no more than 10 of them are sold:

$$c_3 : (\text{branch} = \text{'New York'}) \implies (0.00 \leq \text{price} \leq 100.00),\ (0, 10)$$

Closure over c_1, c_3 means that all of the missing rows come from either New York or Chicago.

4.4 Calculating result ranges

This section focuses on a simplified version of the bounding problem. We consider a single table and a single attribute aggregates. Let q denote such an aggregate query. The problem to calculate the upper bound is:

$$u = \max_{R} q(R) \tag{4.1}$$

$$\text{subject to: } R \models \mathbf{S}$$

We will show that our bounds are tight–meaning that the bound found by the optimization problem is a valid relation that satisfies the constraints.

Throughout the rest of the paper, we only consider the maximal problem. Unless otherwise noted, our approach also solves the lower bound problem:

$$l = \min_{R \in \mathcal{R}} q(R)$$

$$\text{subject to: } R \models \mathbf{S}$$

Specifically, we solve the lower bound problem in two settings. In a general setting where there is no additional constraint, the minimal problem can be solved by maximizing the negated problem: we first negate the value constraints, solve the maximizing problem with negated weights, and negate the final result.

In a special but also common setting, all the frequency constraints' lower bounds are 0 (i.e., each relation has no minimum number of missing rows) and the value constraints' lower bounds are 0 (i.e., all attributes are non-negative), the lower bound is easily attained by the absent of missing row.

4.4.1 Cell Decomposition

Intuitively, we can think about the optimization problem as an allocation. We strategically assign rows in each of the predicates to maximize the aggregate query. However, the first challenge is that a row may fall in multiple Predicate-Constraints' predicates, so this row may "count towards" multiple Predicate-Constraints. As the first step of the solution, we decompose the potential overlapping Predicate-Constraints' predicates into disjoint cells.

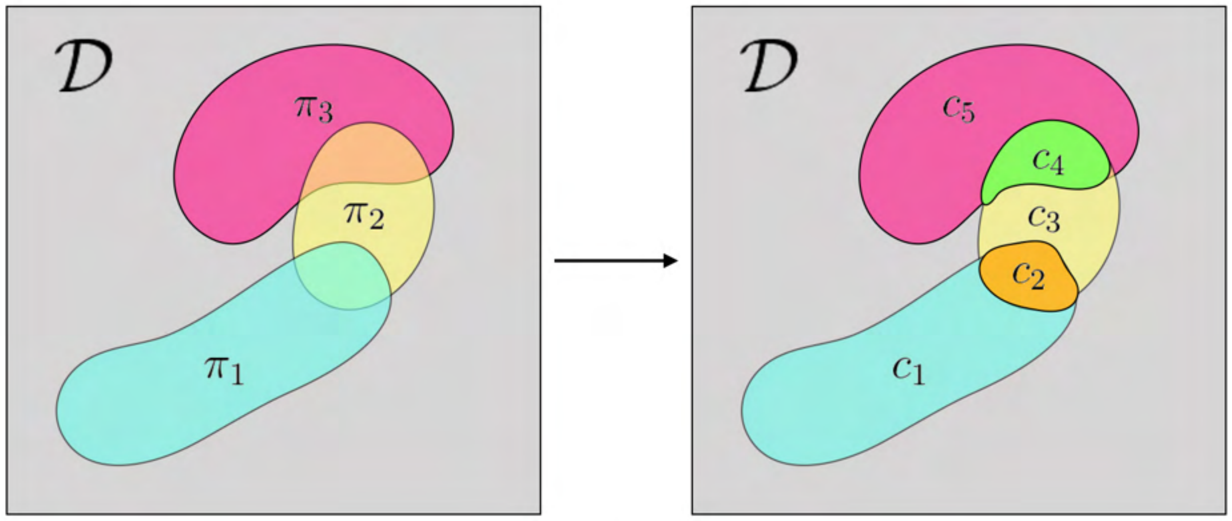


Figure 4.2: Predicates in a predicate-constraint set are possibly overlapping. The first step of the algorithm is to decompose a set of predicate-constraints into disjoint cells.

An example that illustrates the decomposition is depicted in Figure 4.2. Each predicate represents a sub-domain. For each subset of predicates, a cell is a domain that only belongs to these predicates and not others. Thus, for n predicates in a predicate-constraint set there are at most $O(2^n)$ cells. The cells take the form of conjunctions and negations of the predicates of each of the n predicate constraints:

$$c_0 = \psi_1 \wedge ... \wedge \psi_n$$

$$c_1 = \psi_1 \wedge ... \wedge \neg\psi_n$$

$$c_2 = \psi_1 \wedge ... \wedge \neg\psi_{n-1} \wedge \psi_n$$

$$...$$

$$c_{2^n-1} = \neg\psi_1 \wedge ... \wedge \neg\psi_{n-1} \wedge \neg\psi_n$$

For each c_i, we have to reconcile the active predicate constraints (not negated above). Each cell is thus assigned the most restrictive upper and lower value bounds, and upper and lower cardinality bounds in the set of active constraints. Not all possible cells will be satisfiable–

where there exists a row $t \in \mathcal{D}$ that satisfies the new predicate-constraint. As in Figure 4.2, there are 7 possible subsets, but there are only 5 satisfiable cells. We use the Z3 [35] solver to prune all the cells that are not satisfiable.

Optimization 1. Predicate Pushdown: Cell decomposition is a costly process for two reasons. First, there is a potentially exponential number of cells. Second, determining whether a cell is satisfiable is not always easy (each check is on the order of 10's of ms). One obvious optimization is to push down the predicates of the target query into the decomposition process. When the target query has predicates, we exclude all cells that do not overlap with the query's predicate.

Optimization 2. DFS Pruning: The naive solution is to simply sequentially iterate through all of the cells and test each for satisfiability. Note that for a problem with n PCs, each logical expression describing a cell is a conjunction of n predicates. Conjunctions can be short-circuited if any of their constituent elements evaluate to false. Therefore, the process of evaluating the satisfiability of the cells can be improved by using a Depth First Search that evaluates prefixes of growing length rather than a sequential evaluation. With DFS, we can start from the root node of all expressions of length 1, add new PCs to the expression as we traverse deeper until we reach the leaf nodes which represent expressions of length n. As we traverse the tree, if a sub-expression is verified by Z3 to be unsatisfiable, then we can perform pruning because any expression contains the sub-expression is unsatisfiable.

Optimization 3. Expression Re-writing: To further improve the DFS process, we can re-write the logical expressions to have an even higher pruning rate. There is one simple re-writing heuristic we apply:

$$(X \wedge \neg(X \wedge Y)) = True \implies X \wedge \neg Y = True$$

It means if we have verified a sub-expression X to be satisfiable, and we also verified that after adding a new PC Y, the expression $X \wedge Y$ becomes unsatisfiable. We can conclude that

$X \wedge \neg Y$ is satisfiable without calling Z3 to verify. As shown in the experiment section, the DFS pruning technique combined with the rewriting can prune over 99.9% cells in real-world problems.

Optimization 4. Approximate Early Stopping: With the DFS pruning technique, we always get the correct cell decomposition result because we only prune cells that are verified as unsatisfiable. We also propose an approximation that can trade range tightness for a decreased run time. The idea is to introduce 'False-Positives', i.e., after we have used DFS to handle the first K layers (sub-expressions of size K), we stop the verification and consider all cells that have not been pruned as satisfiable. These cells are then admitted to the next phase of processing where we use them to formulate a MILP problem that can be solved to get our bound. Admitting unsatisfiable cells introduces 'False-Positives' that would make our bound loose, but it will not violate the correctness of the result (i.e. the result is still a bound) because: (1) the 'true-problem' with correctly verified cells is now a sub-problem of the approximation and (2) the false-positive cells does not add new constraints to the 'true-problem'.

4.4.2 Integer-Linear Program

We assume that the cells in $\mathcal{C}$ are ordered in some way and indexed by i. Based on the cell decomposition, we denote C_i as the (sub-)set of Predicate-Constraints that cover the cell i. Then for each cell i, we can define its maximal feasible value $U_i(a) = \min_{p \in C_i} p.\nu.h_a$, i.e., the minimum of all C_i's value constraints' upper bounds on attribute a.

A single general optimization program can be used to bound all the aggregates of interest. Suppose we are interested in the `SUM` over attribute a, we slightly abuse the terms and define a vector U where $U_i = U_i(a)$. Then, we can define another vector X which represents the decision variable. Each component is an integer that represents how many rows are allocated to the cell.

The optimization problem is as follows. To calculate the upper bound:

$$\max_{X} \quad U^T X \tag{4.2}$$

$$\text{subject to:} \quad \forall j, k_l^{(j)} \leq \sum_{i:j \in C_i} X[i] \leq k_u^{(j)}$$

$$\forall i \ , \ X[i] \text{ is integer}$$

As an example, the first constraint in Figure 4.2 is $k_l^1 <= x_1 + x_2 <= k_u^1$ for π_1, so on and so forth. This MILP can be solved with a commercial optimization solver such as Gurobi or CPLEX.

Given the output of the above optimization problem, we can get bounds on the following aggregates:

COUNT: The count of cardinality can be calculated by setting U as the unit vector ($U_i = 1$ for all i).

AVG: We binary search the average result: to testify whether r is a feasible average, we disallow all rows to take values smaller than r and invoke the above solution for the maximum `SUM` and the corresponding `COUNT`. If the overall average is above r, then we test $r' > r$, otherwise we continue testing $r' < r$.

MAX/MIN: Assuming all cells are feasible, the max is the largest of all cells' upper bound. Min can be handled in a similar way.

Faster Algorithm in Special Cases In the case that the Predicate-Constraints have disjoint predicates, the cell decomposition problem becomes trivial as each predicate is a cell. The MILP problem also degenerates since all the constraints, besides each variable being integer, do not exist at all. Thus, if we take the `SUM` problem as an example, the solution is simply the sum of each Predicate-Constraints' maximum sum, which is the product of the maximum value and the maximum cardinality. This disjoint case is related to the work in data privacy by Zhang et al. [137].

4.4.3 Complexity

Next, we analyze the complexity of the predicate-constraint optimization problem. Suppose each predicate-constraint is expressed in Disjunctive Normal Form with at most p clauses. Suppose, we are given a set of N such predicate-constraints. The data complexity of the predicate-constraint optimization problem is the number of computational steps required to solve the optimization problem for a fixed p and a variable N. For $p >= 2$, we show that the problem is computationally hard in N based on a reduction:

Proposition 4.4.1. Determining the maximal sum of a relation constrained by a predicate-constraint is NP-Hard.

Sketch. We prove this proposition by reduction to the maximal independent set problem, which is NP-Hard. An independent set is a set of vertices in a graph, no two of which are adjacent. We can show that every independent set problem can be described as a predicate-constraint maximal sum problem. Let $G = (V, E)$ be a graph. We can think of this graph as a relational table of "vertex" rows. For example: (V1, V2). For each vertex $v \in V$, we define a single Predicate-Constraint with a max value of 1 and a max frequency of 1 $(x = v, [0, 1], [0, 1])$. For each edge between v and v', we define another predicate constraint that is equality to either of the vertices also with a max frequency of 1:

$$(x = v \vee x = v', [0, 1], [0, 1]).$$

This predicate constraint exactly contains the two vertex constraints. Therefore, the cells after the cell-decomposition step perfectly align with those vertices. The optimization problem allocates rows to each of those cells but since the edge constraint has a max frequency is 1 only one of the vertex cells can get an allocation. Since all of the vertex constraints have the same max value, the optimization problem finds the most number of such allocations that are consistent along the edges (no two neighboring vertices both are allocated). This is exactly the definition of a maximal independent set. Since the maximal sum problem is more expressive than the maximal independent set its complexity is greater than NP-Hard in the number of cells. □

The hardness of this problem comes from the number of predicate constraints (i.e., data complexity). While, we could analyze the problem for a fixed N and variable p (i.e., query complexity), we find that this result would not be informative and is highly problem specific with very large constant-factors dependent on N.

4.4.4 Numerical Example

To understand how this optimization problem works, let's consider a few simple numerical examples using our example dataset. Suppose, we have a query that wants to calculate the total number of sales over a period:

```
SELECT SUM(price)
FROM Order
WHERE utc >= Nov-11 0:00 AND
      utc <= Nov-13 0:00
```

For the sake of simplicity, let's assume that all of this data was missing and the only information we have is what is described in the PCs. Suppose, we define two PCs that describe the price of the least/most expensive items sold on the day, and that between 50 and 100 items were sold:

$$t_1{:}Nov{-}11{<=}utc{<}Nov{-}12 \implies 0.99{<=}\text{price}{<=}129.99,\ (50{,}100)$$

$$t_2{:}Nov{-}12{<=}utc{<}Nov{-}13 \implies 0.99{<=}\text{price}{<=}149.99,\ (50{,}100)$$

Since the PCs are disjoint, we can trivially compute the result range for the total sales:

$$[50 \times 0.99 + 50 \times 0.99, 100 \times 129.99 + 100 \times 149.99]$$

$$= [99.00, 27998.00]\blacksquare$$

Now, suppose, we had overlapping PCs. This makes the solution much harder to manually reason about since we have to account for the interaction between the constraints.

$$t_1{:}Nov{-}11{<=}utc{<}Nov{-}12 \implies 0.99{<=}\text{price}{<=}129.99,\ (50{,}100)$$

$$t_2{:}Nov{-}11{<=}utc{<}Nov{-}13 \implies 0.99{<=}\text{price}{<=}149.99,\ (75{,}125)$$

This requires a cell decomposition to solve. There are 3 possible cells:

$$c_1{:=}(Nov{-}11{<=}utc{<}Nov{-}12){\wedge}(Nov{-}11{<=}utc{<}Nov{-}13)$$

$$c_2{:=}\neg(Nov{-}11{<=}utc{<}Nov{-}12){\wedge}(Nov{-}11{<=}utc{<}Nov{-}13)$$

$$c_3{:=}(Nov{-}11{<=}utc{<}Nov{-}12){\wedge}\neg(Nov{-}11{<=}utc{<}Nov{-}13)$$

c_3 is clearly not satisfiable, so we can discount this cell. Then, we need to figure out how much to allocate to c_1 and c_2. The lower bound can be achieved by an allocation of 50 tuples to c_1 and 25 to c_2. The upper bound can be achieved by an allocation of 50 tuples to c_1 and 75 tuples to c_2. Note that the optimal allocation does not maximize tuples in c_1.

$$[50 \times 0.99 + 25 \times 0.99, 50 \times 129.99 + 75 \times 149.99] =$$

$$[74.25, 17748.75]\blacksquare$$

4.5 Joins Over Predicate Constraints

In the previous section, we considered converting single table predicate-constraints into result ranges for a query result. In this section, we extend this model to consider aggregate queries with inner join conditions, namely, there are predicate constraints describing missing data in each of the base tables and we have to understand how these constraints combine across tables.

4.5.1 Naive Method

One way to handle multi-relation predicate constraints is to treat a join as a Cartesian product. Let's consider two tables R and S, and let $\mathcal{P}_R$ and $\mathcal{P}_S$ denote their predicate constraints sets respectively. For two predicate constraints from each set π_s and π_r, let's define a direct-product operation as:

$$\pi_s \times \pi_r = (\psi_s \wedge \psi_r, [\nu_s \nu_r], \kappa_s \otimes \kappa_r)$$

that takes a conjunction of their predicates, concatenates their attribute ranges, and multiplies their cardinalities. If we do this for all pairs of predicate constraints, we can derive a set of constraints that account for the join:

$$\mathcal{P} = \{\pi_i \times \pi_j : \forall \pi_i, \pi_j \in \mathcal{P}_R, \mathcal{P}_T\}$$

This approach will produce a bound for all inner-joins since any satisfying tuple in the output has to be satisfying either $\mathcal{P}_R$ or $\mathcal{P}_T$.

While this approach will produce a bound, it may be very loose in certain cases. It is particularly loose in the case of inner equality joins. Most obviously, it does not consider the effects of equality conditions and how those may affect cardinalities and ranges. For conditions that span more than 2 relations, we run into another interesting problem. The

Worst-Case Optimal Join (WCOJ) results are some of the most important results in modern database theory [43, Lemma 3.3]. Informally, they show that solving an n-way join with a cascade of two-way joins can create exponentially more work due to very large intermediate results. An optimal algorithm would only do work proportional to the number of output rows and no more.

A very similar issue arises with this bounding problem. Consider the exemplary triangle counting query:

$$q = |R(a,b)S(b,c)T(c,a)|$$

Suppose, each relation has a size of N. If we apply the naive technique to bound q for a predicate-constraint set defined for each relation, the bound would be $O(N^3)$. However, from WCOJ results, we know that the maximum value of q is $O(N^{\frac{3}{2}})$ [103]. We can perpetuate this logic to the 4-clique counting query, 5-clique, and so on:

$$q = |R(a,b,c)S(b,c,d)T(c,d,e)U(e,a,b)|$$

and the gap between the theoretical bound and the one computed by our framework grows exponentially.

4.5.2 A Better Bound For Natural Joins

We can leverage some of the theoretical machinery used to analyze WCOJ algorithms to produce a tighter bound. We first introduce an important result for this problem, Friedgut's Generalized Weighted Entropy (GWE) inequality [43, Lemma 3.3].

There are r relations, indexed by i as R_i. The joined relation is R. For each row t in R, its projection in the relation R_i is t^i. For each relation R_i, this is an arbitrary non-negative weight function $w_i(\cdot)$ defined on all rows from R_i. Fractional edge cover (FEC) is a vector c that assigns a non-negative value c_i to each relation R_i, and also satisfies that for each attribute s,

$$\sum_{R_i \oplus s} c_i \geq 1$$

$R_i \oplus s$ if the relation R_i contains attribute s (when multiple relations join on one attribute, we consider the attribute indistinguishable, i.e., they all contain the same attribute).

When c is FEC, GWE states

$$\sum_{t \in R} \prod_i \left(w_i(t^i)\right)^{c_i} \leq \prod_i \Big(\sum_{t_i \in R_i} w_i(t_i)\Big)^{c_i} \tag{*}$$

What GWE implies depends on the choice of weight functions $w_i(\cdot)$. In a query with `SUM(A)` aggregation, without loss of generality we assume attribute A comes from the relation R_a. Let $w_a(t_a) = t_a.A$, and $w_i(t_i) = 1$ for $i \neq a$. Then the left hand of $(*)$ becomes

$$\sum_{t \in R} w_a(t^a)^{c_a} = \sum_{t \in R} t.A^{c_a}$$

and the right hand of $(*)$ becomes

$$\prod_{i \neq a} |R_i|^{c_i} \times \Big(\sum_{t_a \in R_a} t_a.A\Big)^{c_a}$$

We only consider the FEC c such that $c_a = 1$, then (*) becomes

$$\sum_{t \in R} t.A \leq \prod_{i \neq a} |R_i|^{c_i} \times \Big(\sum_{t_a \in R_a} t_a.A\Big) \tag{**}$$

Now given a set of PC π_i for each relation R_i, and we only consider those R_i that conform to corresponding PCs. According to $(**)$, we have

$$\underset{\substack{t_1,\ldots,t_r \text{ natural join} \\ R_i \models \pi_i}}{\mathtt{SUM}(A)} \leq \underset{R_a \models \pi_a}{\mathtt{SUM}(A)} \times \prod_{i \neq a} \Big(\underset{R_i \models \pi_i}{\mathtt{COUNT}(*)}\Big)^{c_i}$$

Here the left hand side is our expected result, and the right hand side can be solved on each relation individually using the approaches discussed in the previous section.

Note that we use the solution to the FEC problem to compute c, and this solution derives an upper bound of the left-hand side. In order to get the tightest upper bound, we consider an optimization problem: minimize the right-hand side subject to that c is an FEC. We take a log of the target function (i.e., right-hand side) so both the target function and constraints are in linear form. The optimization problem becomes a linear programming problem, which can be solved by a standard linear programming solver. See Section 6.4 for two cases where this solution significantly improves on the naive Cartesian product solution described above.

4.6 Experiments

Now, we evaluate the PC framework in terms of accuracy and efficiency at modeling missing data rows.

4.6.1 Experimental Setup

We set up each experiment as follows: (1) summarize a dataset with n Predicate-Constraints, (2) each competitor framework gets a similar amount of information about the dataset (e.g., a statistical model with $O(n)$ statistical parameters), (3) we estimate a query result using each framework with calibrated error bounds. Our comparative evaluation focuses on SUM, COUNT queries as those are what the baselines support. We show similar results on MIN, MAX, AVG queries, but only within our framework.

It is important to note that each of these frameworks will return a confidence interval and not a single result. Thus, we have to measure two quantities: the **failure rate** (how often the true result is outside the interval) and the tightness of the estimated ranges. We measure tightness by the ratio between the upper bound and the result (we denote this as the **over estimation rate**). A ratio closer to 1 is better on this metric, but is only meaningful if the failure rate is low.

Sampling

To construct a sampling baseline, we assume that the user provides actual unbiased example missing data records. While this might be arguably much more difficult for a user than simply describing the attribute ranges as in a predicate constraint, we still use this baseline as a measure of accuracy. In our estimates, we use *only* these examples to extrapolate a range of values that the missing rows could take.

Uniform Sampling We randomly draw n samples (US-1) and $10n$ samples (US-10) from the set of missing rows.

Stratified Sampling We also use a stratified sampling method which performs a weighted sampling from partitions defined by the PCs that we use for a given problem. Similarly, we denote n samples as (ST-1) and $10n$ samples (ST-10).

The goal is to estimate the result range using a statistical confidence interval. Commonly, approximate query processing uses the Central Limit Theorem to derive bounds. These confidence intervals are parametric as they assume that the error in estimation is Normally distributed. Alternatively, one could use a non-parametric method, which does not make the Normal assumption, to estimate confidence intervals like those described in [55] (we use this formula unless otherwise noted). We denote confidence interval schemes in the following way: US-1p (1x sample using a parametric confidence interval), US-10n (10x sample using a non-parametric confidence interval), ST-10p (10x stratified sample using a parametric confidence interval), etc.

Generative Model

Another approach is to fit a generative model to the missing data. We use the whole dataset as training data for a Gaussian Mixture Model (GMM). The trained GMM is used to generate the missing data. A query result that is evaluated on the generated data is returned. This is simulating a scenario where there is a generative model that can describe what data is missing. If we run this process several times, we can determine a range of likely values that the query could take.

Equiwidth Histogram

We build a histogram on all of the missing data on the aggregate attribute with N buckets and use it to answer queries. We use standard independent assumptions to handle queries across multiple attributes.

PCs

The accuracy of the PC framework is dependent on the particular PCs that are used for a given task. In our “macro-benchmarks”, we try to rule out the effects of overly tuned PCs. We consider two general schemes: **Corr-PC**, even partitions of attributes correlated with the aggregate of interest, and **Rand-PC**, randomly generated PCs.

For **Corr-PC**, we identify the (other than the aggregation attribute) most correlated attributes with the aggregate to partition on. We divide the combined space into equi-cardinality buckets where each partition contains roughly the same number of tuples. We omit the details of this scheme for the sake of brevity. For **Rand-PC**, we generate random overlapping predicate constraints over the same attributes (not necessarily equi-cardinality). We take extra care to ensure they adequately cover the space to be able to answer the desired queries. We see these as two extremes: the reasonably best performance one could expect out of the PC framework and the worst performance. We envision that natural use cases where a user is manually defining PCs will be somewhere in the middle in terms of performance.

4.6.2 Intel Wireless Dataset

The Intel wireless dataset [115] contains data collected from 54 sensors deployed in the Intel Berkeley Research lab in 2004. This dataset contains 3 million rows, 8 columns including different measurements like humidity, temperature, light, voltage as well as date and time of each record. We consider aggregation queries over the `light` attribute. For this dataset, Corr-PC is defined as $n = 2000$ predicate constraints over the attributes `device_id` and `time`. Rand-PC defines random PCs over those same attributes. Missing rows are generated from the dataset in a correlated way—removing those rows maximum values of the `light` attribute.

We compare Corr-PC and Rand-PC with 3 baselines we mentioned in Section 4.6.1: US-1n, ST-1n, and Histogram. We vary the fraction of missing rows r and evaluate the accuracy and failure rate of each technique. Figure 4.3 and Figure 4.4 illustrate the experimental results: (1) as per our formal guarantees, both Corr-PC, Rand-PC (and Histograms) do not fail if they have accurate constraints, (2) despite the hard guarantee, the confidence intervals are not too "loose" and are competitive or better than those produced by a 99.99% interval, (3) informed PCs are an order of magnitude more accurate than randomly generated ones. There are a couple of trends that are worth noting. First, the failure rate for sampling techniques on the SUM queries is higher than the expected 1 in 10000 failures stipulated by the confidence intervals. In this missing data setting, a small number of example rows fail to accurately capture the "spread" of a distribution (the extremal values), which govern failures in estimation. Queries that require values like SUM and AVG are very sensitive to these extrema.

It is important to note that these experiments are idealized in the sense that all the baselines get true information about the missing data and have to summarize this information into $O(n)$ space and measure how useful that stored information is for computing the minimal and maximal value a workload of aggregate queries could take. There is a subtle experimental point to note. If a query is fully covered by the missing data, we solve the query with each baseline and record the results; if a query is partially covered by the missing data, we solve the part that is missing with each baseline then combine the result with a ‘partial ground truth’ that is derived from the existing data; finally, if a query is not overlapping with the missing data then we can get an accurate answer for such a query. Such issues are common to all the baselines, in the upcoming experiments we will only consider the accuracy of representing the missing data (and not partially missing data) to simplify questions of correlation and accuracy.

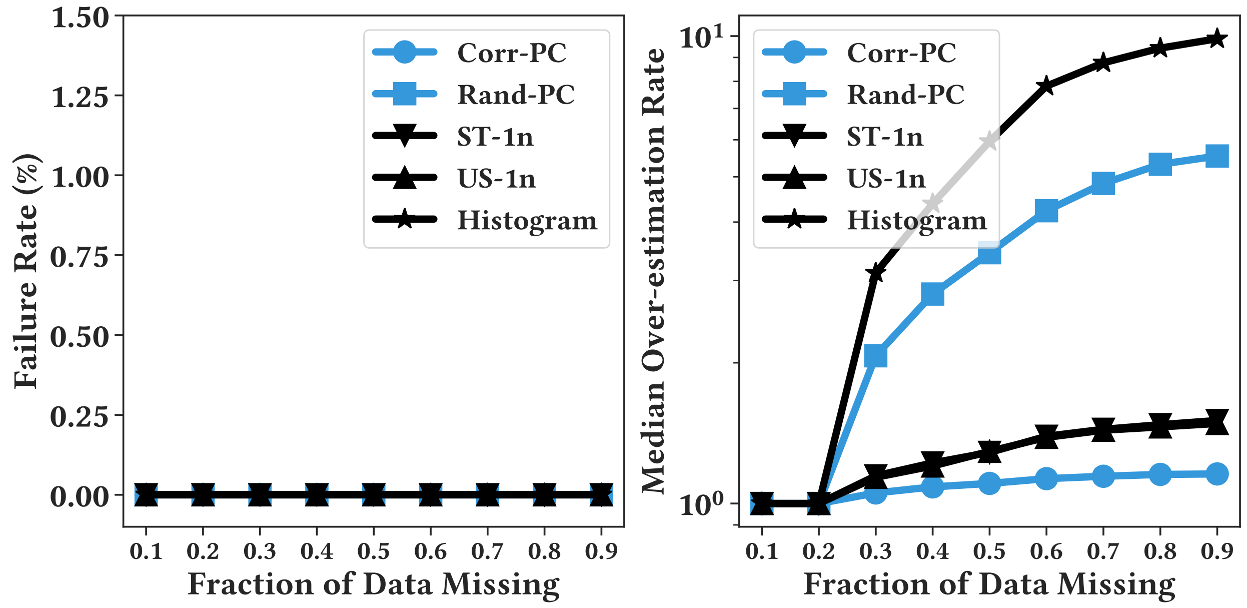


Figure 4.3: Performance of baselines given different missing ratio, evaluated with 1000 COUNT(*) query on the Intel Wireless dataset.

4.6.3 Detailed Sampling Comparison

One might argue that 99.99% is an overly conservative confidence interval. In this experiment, we evaluate the performance of the uniform sampling baseline in terms of failure rate and accuracy as function of that confidence interval setting. Our results show there is not a clear way to calibrate the confidence intervals to either minimize failures or avoid inaccuracy. Results in Table 4.1 show a clear trade-off between failure rate and accuracy: when we increase the confidence interval, the over-estimation ratio increases, and the failure rate

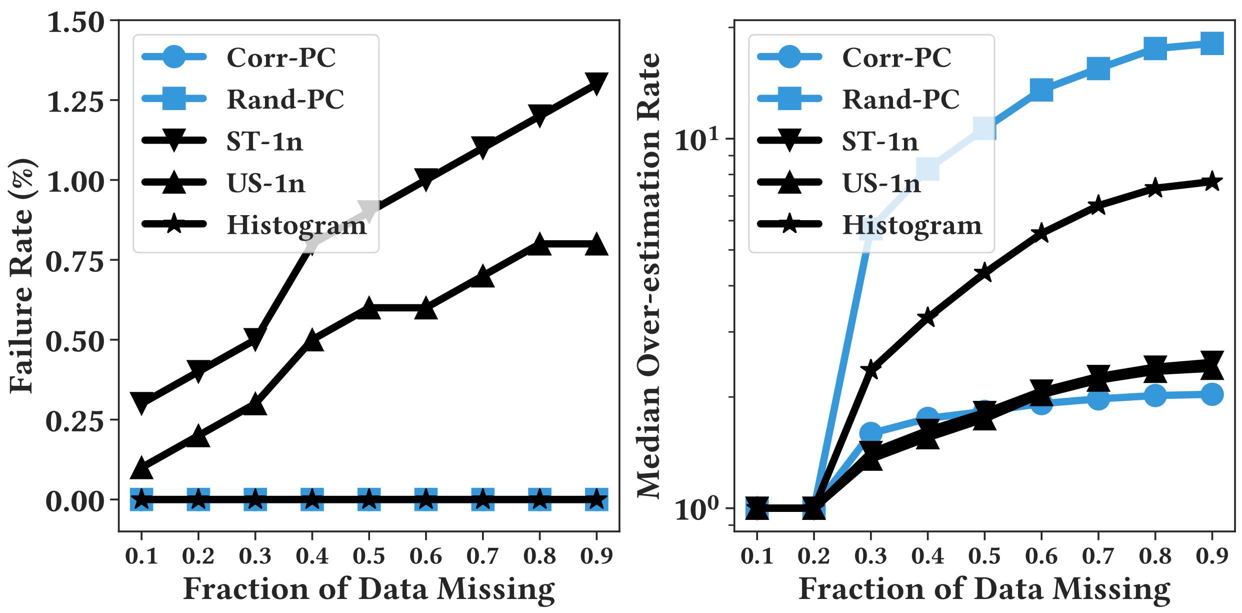


Figure 4.4: Performance of baselines given different missing ratio, evaluated with 1000 SUM query on the Intel Wireless dataset.

decreases. However, even with a 99.99% confidence interval derived from Chernoff Bound, the failure rate is non-zero. We believe that the PC framework provides the user with a competitive accuracy but the guarantee of no failures.

Failure Rate %							
Conf (%)	80	85	90	95	99	99.9	99.99
US-1n	20.1	15.6	11.4	6.9	3.4	2.4	0.8
Corr-PC				–0–			
Over Estimation Rate							
US-1n	1.07	1.08	1.11	1.13	1.2	1.27	3.13
Corr-PC				–2.23–			

Table 4.1: Trade-off between failure rate and accuracy of an uniform sampling baseline given different confidence interval vs. Corr-PC.

Sampling More Data

In all of our experiments above, we consider a 1x random sample. Where the baseline is given the same amount of data compared to the number of PCs. PCs are clearly a more accurate estimate in the "small data" regime, what if the sample size was larger. In Figure

4.5, we use the Intel Wireless dataset to demonstrate the performance of the non-parametric bounds using different sample sizes. And the results demonstrate a clear trend of convergence as we increase the sample size. If we consider data parity (1x), the confidence interval is significantly less accurate than a well-designed PC. One requires 10x the amount of data to cross over in terms of accuracy. We envision that PCs will be designed by a data analyst by hand, and thus the key challenge is to evaluate the accuracy of the estimation with limited information about the distribution of missing values.

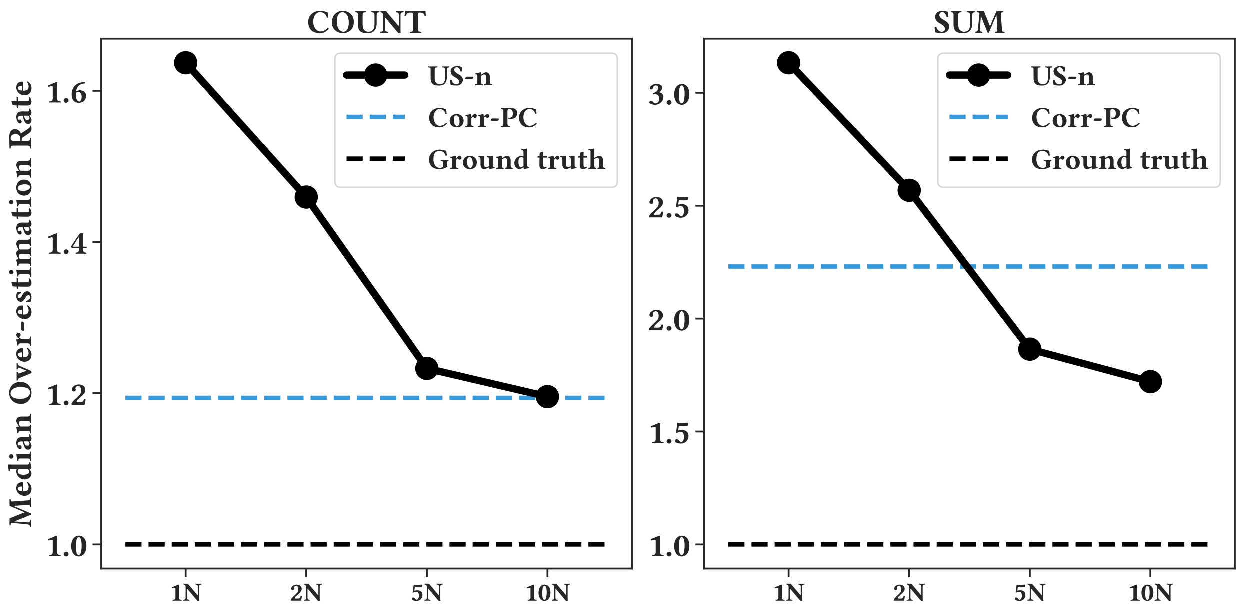


Figure 4.5: Performance of the uniform sampling baseline with different sample size.

Robustness To Noise

Of course, these results depend on receiving PCs that accurately model the missing data. We introduce noise into the PCs to understand how incorrectly defined PCs affect the estimated ranges. When the PCs are noisy, there are failures because the ranges could be incorrect. We add independent noise to the minimum and maximum values for each attribute in each PCs. Figure 4.6 plots the results for Corr-PC, a set of 10 overlapping PCs (Overlapping-PC), and US-10n (a 10x sample from the previous experiment). We corrupt the sampling bound by mis-estimating the spread of values (which is functionally equivalent to an inaccurate

PC). All experiments are on the SUM query for the Intel Wireless dataset as in our previous experiment. For corrupting noise that is drawn from a Gaussian distribution of 1, 2 and 3 standard deviation, we plot the failure rate.

Our results show that PCs are not any more sensitive than statistical baselines. In fact, US-10n has the greatest increase in failures due to the noise. Corr-PC is significantly more robust. This experiment illustrates the benefits of overlapping PCs. When one such overlapping PC is incorrect, our framework automatically applies the most restrictive overlapping component. This allows the framework to reject some amount of mis-specification errors.

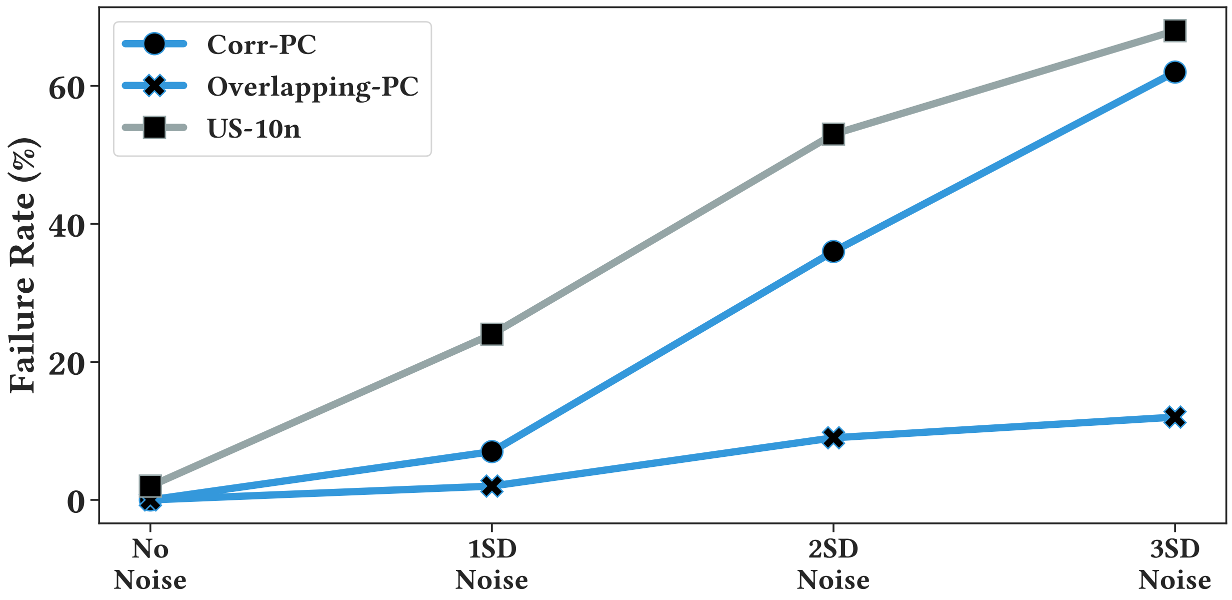


Figure 4.6: We investigate the sensitivity of Corr-PC, Overlapping-PC, and US-10n to different levels of noise. The failure rate of all approaches increases as we increase the noise level, but the PC baselines especially Overlapping-PC are more tolerable to the same level of noise.

4.6.4 Scalability

In this subsection, we evaluate the scalability of PCs and optimization techniques. As mentioned in earlier sections, the complete process of solving a PC problem including two parts. First, we need to perform cell decomposition to find out which cells are valid and second, we need to formalize a MILP problem with the valid cells that can be solved to find the optimal bound. The vanilla version described above is intractable because the number of

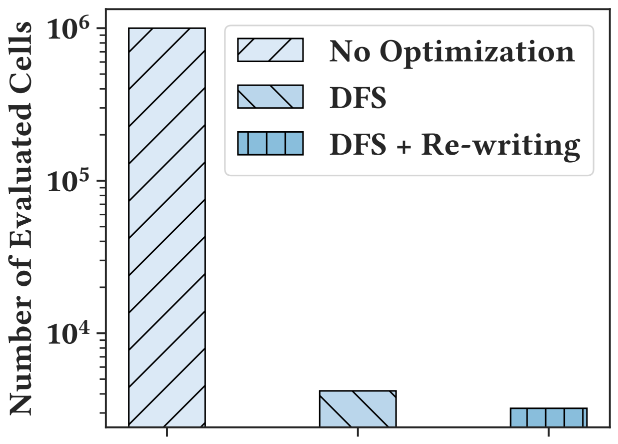


Figure 4.7: Our optimizations reduce the number of cells evaluated during the cell decomposition phase by over a 1000x.

sub-problems we need to solve in the cell decomposition phase is exponential to the number of PCs.

We presented a number of optimizations in the paper to improve this time. We will show that naive processing of the PCs leads to impractical running time. We generate 20 random PCs that are very significantly overlapping. Figure 4.7 plots the number of cells evaluated as well as the run time of the process. The naive algorithm evaluates the SAT solver on more than 1000x more cells than our optimized approach.

Since cell decomposition is really the most expensive step We can prune and save about 99.9% of the solving time by using DFS (early termination for cell decomposition) and the rewriting heuristic. Without these optimizations, PCs are simply impractical at large scales.

Non-Overlapping PCs Scale Effectively

PCs can be solved significantly faster in the special case of partitioned PCs (non-overlapping). The process of answering a query with PC partitions is much simpler than using overlapping PCs. Because partitions are disjoint with each other, we can skip the cell decomposition, and the optimization problem can be solved by a greedy algorithm. As shown in Figure 4.8, the average time cost to solve one query with a partition of size 2000 is 50ms, and the time cost is linear to the partition size. We can scale up to 1000s of PCs in this case.

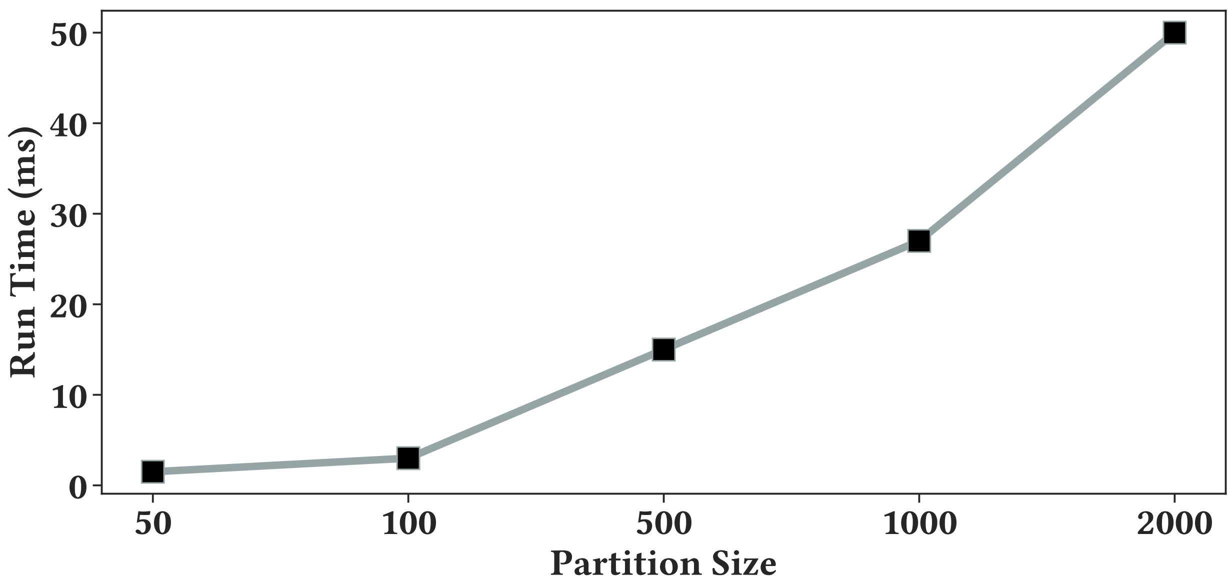


Figure 4.8: The run time needed to solve one query using partition of different size.

4.6.5 Handling MIN, MAX, and AVG Queries

As mentioned in earlier sections, besides COUNT and SUM queries, PCs can also handle MIN, MAX and AVG queries. In this experiment, we use the Intel Wireless dataset for demonstration, we partition the dataset on `DeviceID` and `Time`. For each type of query, we randomly generate 1000 queries and use PC to solve them. Results can be found in Figure 4.9. First, note how PC can always generate the optimal bound for MIN and MAX queries. PCs are a very good representation of the spread of the data, more so than a sample.

We show similar performance for AVG queries to the COUNT and SUM queries studied before. AVG queries are an interesting case that we chose not to focus on. While sampling is a very accurate estimate of AVG queries without predicates, with predicates the story becomes more complicated. Since averages are normalized by the number of records that satisfy the predicate, you get a “ratio of estimators” problem and the estimate is not exactly unbiased. So for small sample sizes, standard bounding approaches can have a high failure rate despite seemingly accurate average-case performance.

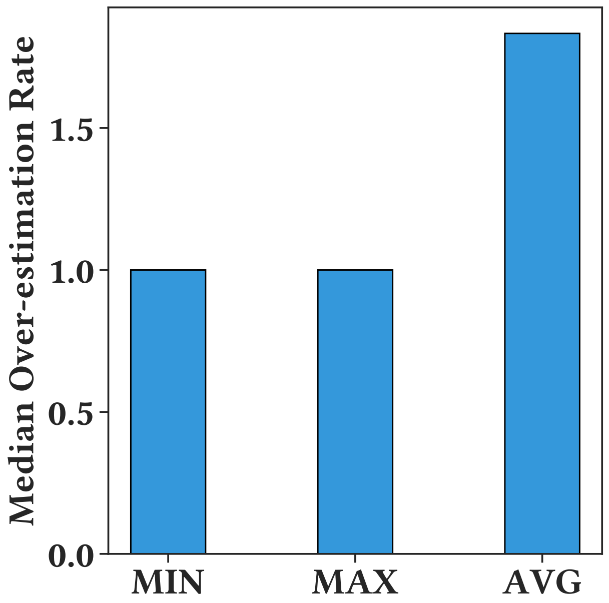


Figure 4.9: With PC, an optimal bound can be derived for MIN and MAX queries. PC also generates competitive result for AVG Queries.

4.6.6 Additional Datasets

We evaluate the accuracy of the framework using two other datasets.

Airbnb at New York City Dataset

The Airbnb dataset [10] contains open data from Airbnb listings and metrics in the city of New York, 2019. This dataset contains 50 thousand rows, 19 columns that describe different properties of listings like location (latitude, longitude), type of room, price, number of reviews, etc. Corr-PC and Rand-PC are defined as $n = 1500$ constraints over `latitude` and `longitude`.

Figure 4.10 replicates the same experiment on a different dataset. This dataset is significantly skewed compared to the Intel Wireless dataset, so the estimates are naturally harder to produce. As before, we find that well-designed PCs are just as tight as sampling-based bounds. However, randomly chosen PCs are significantly looser (more than 10x). PCs fail conservatively, a loose bound is still a bound, it might just not be that informative. In skewed data such as this one, we advise that users design simple PCs that are more similar

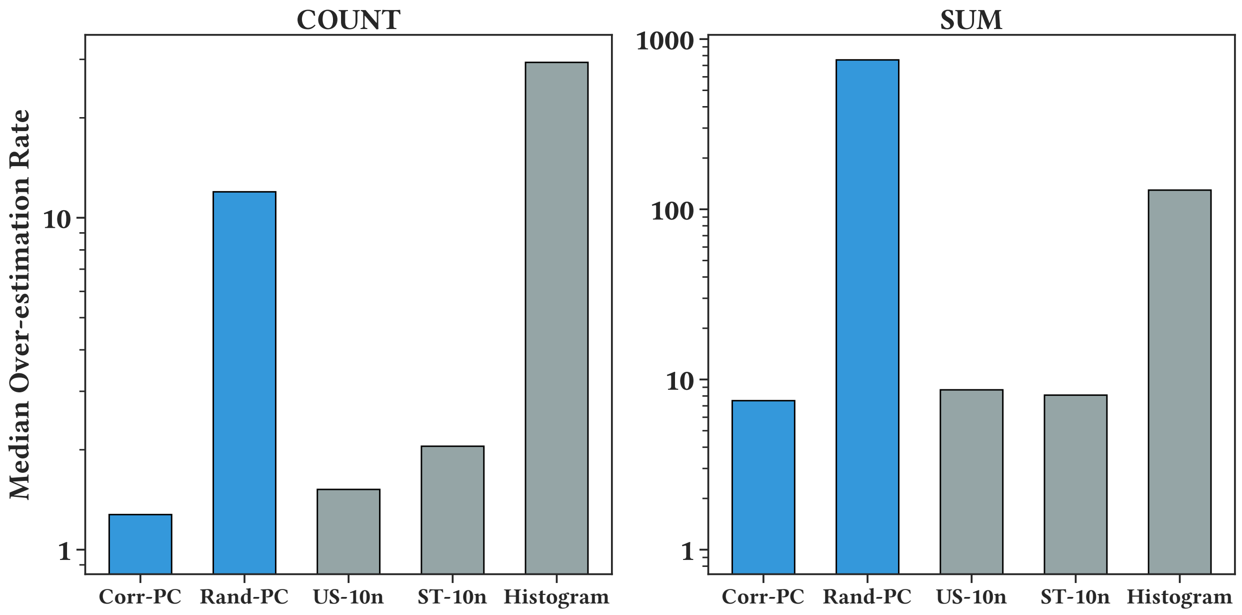


Figure 4.10: Baseline performance on 1000 COUNT(*) and SUM queries with predicate attributes on *Latitude* and *Longitude* using the Airbnb NYC dataset.

to histograms (partition the skewed attribute).

Border Crossing Dataset

The Border Crossing dataset [104] from The Bureau of Transportation Statistics (BTS) summary statistics for inbound crossings at the U.S.-Canada and the U.S.-Mexico border at the port level. This dataset contains 300 thousand rows, 8 columns that describe the summary of border crossing (the type of vehicles and the count) that happen at a port (port code, port location, state, etc) on a specific date. We compare the hard bound baselines with PCs using different partitions with three groups of randomly generated queries. Corr-PC and Rand-PC are defined as $n = 1600$ constraints over `port` and `date`.

Results in Figure 4.11 show results on another skewed dataset. As before, informed PCs are very accurate (in fact more accurate than sampling). Randomly chosen PCs over-estimate the result range by about 10x compared to the other approaches. Again, the advantage of the PC framework is that unless the assumptions are violated, there are no random failures. On this dataset, over 1000 queries, we observed one bound failure for the sampling approach.

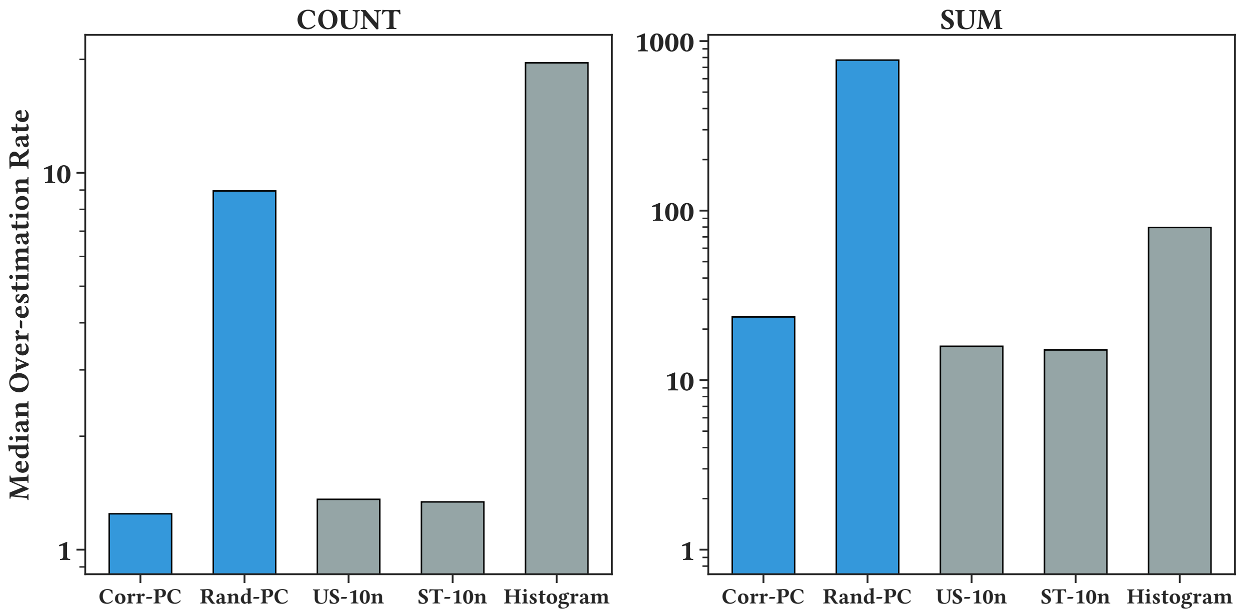


Figure 4.11: Baseline performance on 1000 COUNT(*) and SUM queries with predicate attributes on *Port* and *Date* using the Border crossing dataset.

This failure is included in the results.

Join Datasets

We also evaluate the PC framework on a number of synthetic join examples on randomly generated data. The statistical approaches do not generalize well to estimates for queries with inner equal joins, and we found the bounds produced were too fallible for meaningful comparison. To evaluate PCs on such queries, we compare to another class of bounding techniques that have been proposed in the privacy literature. These bounds estimate how much a query might change for a single hypothetical point update. Our insight connecting the bounding problem to worst-case optimal join results leads to far tighter bounds in those settings. Johnson et al. [65] proposed a technique named elastic sensitivity that can bound the maximum difference between the query's result on two instances of a database.

Counting Triangles. In this example, which is also studied by Johnson et al. [65], we analyze a query that is used to count triangles in a directed graph. In Figure 4.12 (TOP), we show the results of the two approaches on the counting triangle problem using randomly populated *edges* tables of different sizes. And the results confirm that our approach drives a bound that is much tighter in this case—in fact by multiple orders of magnitude.

Acyclic Joins. We also consider the following join query:

$$R1(x_1, x_2) \bowtie R2(x_2, x_3) ... \bowtie R5(x_5, x_6)$$

We generate 5 tables, each with K rows and use the two approaches to evaluate the size of the join results. We vary the value of K to understand how the bounds change accordingly. The results are shown in Figure 4.12 (BOTTOM), we can see that elastic sensitivity always assumes the worst-case scenario thus generates the bound for a Cartesian product of the tables that is several magnitudes looser than our approach.

Dataset	Query	PredAttr	PC	Hist	US-1p	US-10p	US-1n	US-10n	ST-1n	ST-10n
		Time	0	0	0	0	0	0	0	0
	COUNT(*)	DevID	0	0	2	3	0	0	0	0
Intel Wireless		DevID, Time	0	0	12	3	0	0	0	0
		Time	0	0	2	0	0	0	0	0
	SUM(Light)	DevID	0	0	3	0	0	0	0	0
		DevID, Time	0	0	24	4	8	2	9	1
		Latitude	0	0	15	0	0	0	0	0
	COUNT(*)	Longitude	0	0	36	0	0	0	0	0
		Lat, Lon	0	0	39	0	0	0	0	0
Airbnb@NYC		Latitude	0	0	16	0	13	0	11	6
	SUM(Price)	Longitude	0	0	36	0	35	0	24	16
		Lat, Lon	0	0	45	19	39	0	39	13
		Port	0	0	1	1	0	0	0	0
	COUNT(*)	Date	0	0	1	0	0	0	0	0
		Port, Date	0	0	13	1	0	0	0	0
Border Cross		Port	0	0	173	1	1	1	1	1
	SUM(Value)	Date	0	0	33	1	1	0	0	0
		Port, Date	0	0	192	12	20	0	15	0

Table 4.2: Over a 1000 randomly chosen predicates, we record the number of failure events of different "error bound" frameworks. A failure event is one where an observed outcome is outside the range of the returned bounds. For a 99% confidence interval, as used in our experiments, one would expect a 1% failure rate for the sampling-based frameworks—but is significantly higher in practice for small skewed datasets. PCs, and as a special case Histograms, are guaranteed not to fail if the assumptions are satisfied.

4.6.7 Probabilistic Confidence Intervals are Often Unreliable on Real Data

Table 4.2 presents different techniques and their "failure rate" over 1000 queries, which is the number of queries for which the true value exceeded what was produced in a bound. The most common technique by far is to rely on the Central Limit Theorem (US-1p, US-10p). Estimating this standard error from a sample is often far more unreliable than one would normally expect. We use a 99% confidence interval for a CLT bound given N samples and $10N$ samples, and observe that the failure rate is far higher than 1%. In this missing data setting, a small number of example rows fail to accurately capture the "spread" of a distribution.

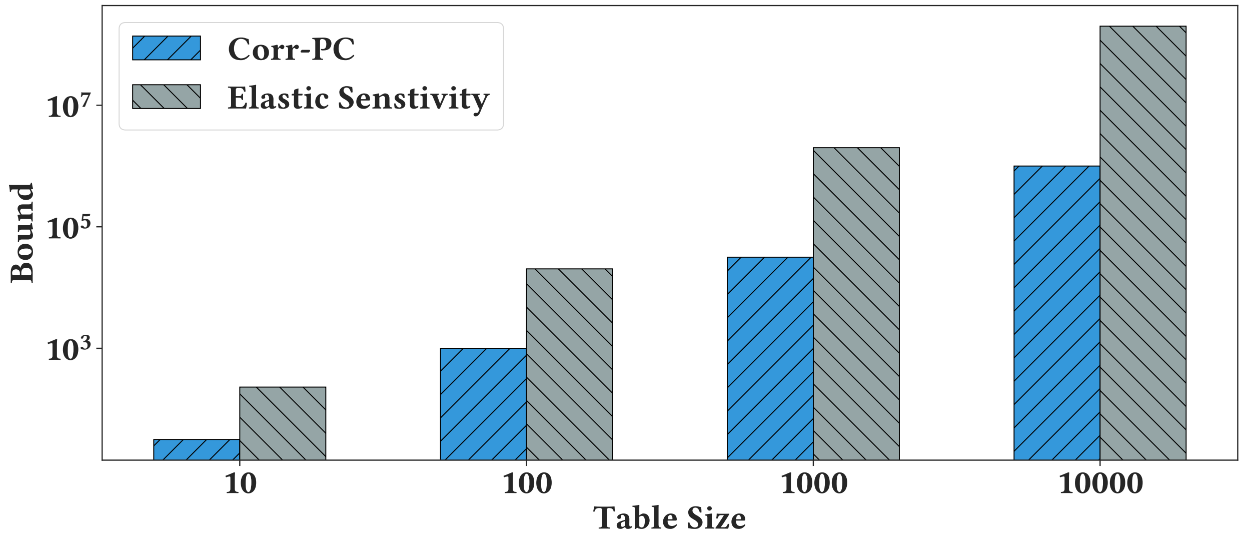
Next, we can make the sample-based confidence intervals a much more conservative non-parametric model (US-1n, US-10n), which holds under milder assumptions. Such a bound relies on an estimate of the min and max values and not an accurate estimate of the standard error. Predictably, this performs much better than the CLT approach. However, as we can see in the table, non-parametric bound baselines still fail more often than one would expect over 1000 queries. Small samples and selective queries pose a fundamental challenge to these approaches. Stratified samples do not solve this problem either. While, they cover the space more evenly, for any given strata, they can have a high failure rate.

One could intuitively fix this problem by annotating the strata in a stratified sample with metadata that accurately depicts min and max values. This is exactly the definition of PCs. The PC technique and Histograms always generate hard bounds for queries because for the same number of "bits" of information they capture the entire spread of values much more accurately. For the purposes of bounding, the example tuples provided by a sample are not as useful as the ranges.

Finally, we use the generative approach to model the joint data distribution. We draw samples from this model and use that to produce a confidence interval. Such an approach works very well on some datasets/queries but not others. These experiments illustrate how important a guaranteed "0 failure rate" is for real-world decision making. Statistical confidence intervals can give a false sense of security in real-world data.

4.7 Conclusion

We proposed a framework that can produce automatic contingency analysis, i.e., the range of values an aggregate SQL query could take, under formal constraints describing the variation and frequency of missing data tuples. There are several interesting avenues for future work. First, we are interested in studying these constraints in detail to model dirty or corrupted data. Rather than considering completely missing or dirty rows, we want to consider rows with some good and some faulty information. From a statistical inference perspective, this new problem statement likely constitutes a middle ground between sampling and Predicate-Constraints. Second, we would like to further understand the robustness properties of result ranges computed by Predicate-Constraints as well as other techniques. Understanding when result ranges are meaningful for real-world analytics will be an interesting study. Finally, we would like to extend the Predicate-Constraint framework to be more expressive and handle a broader set of queries.



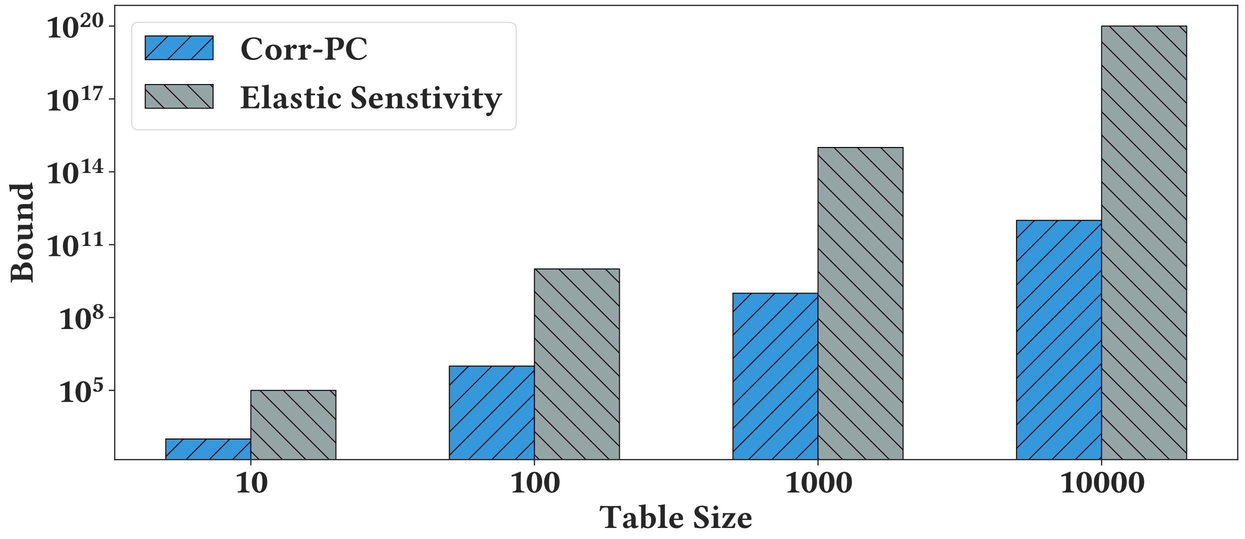


Figure 4.12: We compare the bound derived by our approach (Corr-PC) with state of the art baseline Elastic Sensitivity on the triangle counting problem of different table sizes (TOP) and an acyclic join (BOTTOM).

CHAPTER 5
OPPORTUNISTIC VIEW MATERIALIZATION WITH DEEP REINFORCEMENT LEARNING

5.1 Introduction

It is common for a database to receive a large number of closely-related queries in a short period of time. For example, during data exploration, a data scientist might execute a query with several different filters to test different contingencies [81]. Or, a recent news report may trigger a hot-spot in an e-commerce database in records relating to a featured product and its related product recommendations [29, 88]. These bursty, short-lived events often have a natural redundancy where successive queries may be able to reuse previously computed, expensive subquery structures. Ideally, a database system should be able to respond to such workload events by automatically identifying and persisting useful intermediate results for future use without explicit intervention.

This idea motivates a framework that "opportunistically" caches useful intermediate results. By opportunistic, we mean that while processing a query, intermediate states that are generated as a part of its execution are selectively persisted for future reuse [82]. In its purest form, the database starts with a "blank slate" with only the base tables and empty extra storage space. As queries are executed, the opportunistic view materialization (OVM) framework should identify commonly occurring subqueries and store their results into the extra storage as materialized views that can be used by future queries. The basic optimization problem is to determine whether materializing an intermediate result benefits future query processing more than the future storage and maintenance costs that it might incur. The framework may need to evict existing materialized views to free up enough space to persist a new view. Thus, the core technical challenge is reasoning about the long term opportunity costs of choosing to materialize a particular intermediate state.

This opportunistic nature is what differentiates such a framework from extensive prior work on view recommendation [114, 33, 16, 109, 9, 139, 64], since opportunism couples view creation with query execution. This coupling leads to a complex interplay between the database query optimizer and the materialization decisions. For example, it might be beneficial to force a sub-optimal query plan that generates an intermediate result that is useful for future query processing. Furthermore, this setting induces a sequential problem where the decision is stateful; the choice of whether to persist a view depends on those views currently persisted. There might be two different views that independently benefit a future query, but the query cannot be rewritten to use both. While similar to the problem of page caching, OVM brings further challenges due to overlapping intermediate results and a non-uniform benefit and cost for persisting results.

In part due to these nuances, we find that existing heuristics cannot be applied as a general-purpose solution. Such heuristics range from the simplest Least-Recently-Used or Least-Frequently-Used approaches [36, 73, 123] to more sophisticated cost-model based approaches dynamic view selection approaches [114, 101]. We find that some heuristics [101] perform well in join-dominated workloads and others [114] perform better in data-cubing problems, but not on both. Not surprisingly, our experiments further suggest that the performance of these approaches dramatically degrades if there are cardinality estimation errors or unmodeled costs, such as the effects of background view maintenance.

A machine learning approach that incorporates feedback from observed execution times can ameliorate this brittleness. Rather than relying on a heuristic, we can observe whether the creation of a view has a net positive or negative impact on subsequent query latencies. Beneficial decisions should be remembered for the future and adverse decisions should be avoided. This broad idea is inspired by recent works that apply Reinforcement Learning (RL) to query optimization, where actual runtimes are used to inform/optimize future plans [96, 26, 95, 78]. A working definition of RL is “learning by doing”; the algorithm takes actions and observes feedback via a performance metric (e.g., query runtime). It assigns credit or blame to actions based on the feedback, and can even account for delayed effects. As more feedback is observed, the learned behavior is increasingly informed. It does not require a hard-coded heuristic nor does it need to explicitly generate an anticipated workload—its predictive model is simply a "means-to-an-end" in terms of minimizing overall query latency.

The RL algorithm has to be able to assign credit or blame purely from how the queries execute. This is the crux of the machine learning challenge in applying RL in OVM systems—ascertaining the net benefit of a view is difficult. Any system either makes a choice to use a view or not during query optimization, and the learning agent only observes the final runtime of one of these choices–and does not know the marginal effect with respect to the other choice. We lack the "paired" experiment, where we observe the same query with and without the view, thereby quantifying the *reward* of creating a view [14]. If the same queries (or similar queries) do not frequently repeat, the amount of time needed to learn an effective and adaptive materialization policy will be prohibitive.

Our insight is that OVM systems need a new type of asynchronous RL algorithm that runs such paired experiments in the background. For every query, the system identifies a set of eligible views that can be opportunistically created. The scope of the current work covers views with inner joins, predicates and aggregate functions, but the technique is more general. The system proactively takes the decision it thinks is best at the time using its query optimizer (possibly using no views). The counterfactual decision(s), the ones that the system did not take, are queued into an experiment buffer. We simplify the experimentation problem by assuming an in-memory database with no extraneous unobserved state (e.g., the buffer pool state or caching effects). Therefore, we can independently schedule and run these experiments during idle times producing retroactive marginal utility metrics for each view. Our system can further model view refresh costs but is not optimized for OLTP systems where these refresh events might be very frequent.

We implement this model in a prototype OVM system called Deep Q-Materialization (DQM). DQM contains three main components: (1) An online query miner that analyzes a trace of SQL queries to identify candidate views for the current query to opportunistically materialize, (2) an RL agent that selects from the set of candidates, and (3) an eviction policy that selects views to delete. DQM is integrated with SparkSQL. The adaptive policy interacts with the Spark environment through a RESTful API and can easily be ported to other SQL-based data processing systems. Over workloads of 10000 queries, DQM is competitive with the best of the heuristics on each workload in terms of cumulative query latency. This is even including the time needed to learn the selection model. Further experiments find that DQM can match or outperform standard heuristics policies across 5 different temporal query patterns on several different workloads. DQM further maximizes utilization of available storage for the given workload and query processing engine.

In summary, this paper makes the following contributions:

- We formalize online view selection in opportunistic materialization systems as a Markov Decision Process (MDP).
- We propose a new asynchronous reinforcement learning algorithm, based on the Double DQN model, to optimize this MDP objective online.
- We propose a new credit-based eviction model that can enforce a hard storage constraint on views created by the learned selection policy.
- We compare our approach to classical and state-of-the-art baselines to demonstrate DQM's adaptivity, latency, and robustness.

5.2 Background

In this section, we overview the related literature, review reinforcement learning, and motivate our system DQM.

5.2.1 Scope

We borrow the term "opportunistic materialization" from [82], which describes automatic persistence in large-scale data processing systems like Hive and Pig. We use the term *opportunistic* to describe any materialization that is an artifact of execution and not explicitly defined by a human database administrator. All queries given to DQM are specified in a standard SQL dialect. DQM identifies select, inner join, and aggregate views that can be opportunistically materialized for future reuse. DQM learns which of these views to materialize completely *online* by making decisions. Learning from observations requires making occasional suboptimal decisions (also called exploration), and all of our performance numbers include this overhead unless otherwise noted. We optimize DQM for in-memory analytics scenarios, and thus do not account for disk caching or buffer pool state in our learned model. While DQM does account for maintenance costs, we assume that maintenance events are infrequent.

5.2.2 Related Work

Classical view selection methods recommend the best views to create based on a query workload and storage constraints [114, 33, 16, 109, 9, 139]. The downside is that one only searches over "static" strategies, where the views are created upfront. Even if we were to periodically run such view recommendation tools, we would have a number of difficult, unresolved questions: how to window the workload, how to penalize view creation costs, and how frequently to re-run a recommendation tool. An intriguing variant of these ideas is to form a predictive model that forecasts the type and distribution of queries one may encounter [92]. As far as we can tell, such a predictive approach has not yet been truly applied to materialized view selection (Ma et al. only study index creation) and defer a detailed exploration of workload forecasting for future work.

Online materialization has been less extensively studied in SQL analytics. DynaMat [73] and WATCHMAN [123] were seminal projects in dynamic materialized view management.

Systems in this space have to solve multiple problems: what views to materialize [94, 117], when to evict views [36], and how to select which views to use [27]. Older systems borrowed strategies from database paging (e.g., LRU), and state-of-the-art systems apply more sophisticated scoring heuristics that account for creation and usage costs. HAWC [114] scores views based on a cost-model and maintains a table of such scores for persisted views. New views that have a higher score than those in the table force an eviction event. The scores in the table are windowed to consider only the latest K queries to ensure adaptivity. Recycler [102] prioritizes the most expensive views (in terms of creation cost). The reasoning is that these views are harder to re-create if they are evicted. However, all of these heuristic-based approaches are limited. Our experiments suggest that there is no "one-size fits all" heuristic. It is challenging to decide *a priori* whether a heuristic-based approach will even work for a workload. More subtly, these heuristics can be at the mercy of the DBMS's cost estimation and query optimizer and actually hurt performance. We believe this is an opportunity for adaptive online approaches like DQM that use the actual observed query latencies as feedback.

Similar problems have been studied in the context of big data analytics systems. There are a number of other recent works that reuse computation in MapReduce jobs [42], Spark RDDs [79], and Pig [19]. The Nectar system [53] caches important subroutines for DryadLINQ programs. Kodiak uses an optimization algorithm to anticipate hotspots and proactively compute certain results in their online advertising database. The CloudViews system [64], which identifies overlapping DAG subgraphs across different jobs and persists them as materialized views, is the most relevant to our work. CloudViews proposes and implements many of the core components in DQM including online materialization and updating cost estimates from feedback from actual executions. We replace this entire architecture with a learned end-to-end reinforcement learning model. RL greatly simplifies the design by obviating the need for a separate cost-estimation and view selection module. This design choice is building off a recent trend in AI-powered database systems [112, 83, 11, 136, 95, 78]. Experiments suggest competitive performance with the CloudViews algorithm even when CloudViews is given a perfect cost estimate. It is worth noting that the scope of DQM is narrower and allows for a tighter coupling between the query optimizer and the materialization system, which we discuss later in the paper. The idea is inspired by similar tight couplings of query optimization and data partitioning/placement in related work [72, 91].

5.2.3 Problem Statement

Every database D is a collection of base relations (tables) and derived relations (materialized views). Let $W = [q_0, q_1...]$ be a sequence of read-only queries. These queries are issued to D in the order that they arrive. As views are materialized and deleted, the database state s_i changes. The system automatically chooses to re-write queries given the views that are materialized. Therefore, every query has a latency associated with it, given the current state of the database, and the overall runtime of the workload is defined as:

$$\mathsf{Runtime} = \sum_{i=0}^{\infty} \mathsf{Latency}(q_i)$$

Associated with each query q_i is a set of new views V_i that can be persisted opportunistically (as the query is executed) for the future. Any of these views can be persisted when processing the query:

$$s_{i+1} \leftarrow v$$

There is a storage cost to persisting such views, which is a function of the current database state:

$$\mathsf{Storage} = \sum_{i=0}^{\infty} C(s_i),$$

and there is further a cap on the amount of storage used at any given time:

$$C(s_i) \leq \mathsf{Capacity}$$

Therefore, our system possibly needs to evict a view:

$$s_{i+1} \not\leftarrow v$$

At each time-step, the state of the database increments based on the view creation and eviction actions taken by the materialization policy (function defined by π):

$$s_{i+1} = \pi(s_i, q_i)$$

Problem 1 (Opportunistic Materialization)**.** Given a database instance s_0 and a stream of queries Q, plan a set of view creation and eviction operations to:

$$\min_{\pi \in \Pi} \sum_{i=0}^{\infty} \mathsf{Latency}(q_i)$$

$$\text{subject to: } s_{i+1} = \pi(s_i, q_i)$$

$$C(s_i) \leq \mathsf{Capacity}$$

5.2.4 *What Does DQM Learn?*

Materialization is not like paging: there is a complex interplay between immediate effects (use) and long-term effects (the opportunity cost of storing a view). Even discounting other uncertainty in the DBMS, like errors in query optimizer cost estimation, these are effects that are fundamentally hard to encode as fixed heuristics. Thus, we advocate for an end-to-end machine learning approach that is grounded in real run-times. Rather than building a separate view selection optimizer and cost model, we couple them together in a single policy. *DQM learns a scoring function that evaluates the marginal utility of materializing and adding a new view to the set of already materialized views.*

DQM learns this scoring function by observing query latencies in the database after taking a certain action, and this learning procedure can be cast as Reinforcement Learning (RL) problem [128]. RL models a learning setting where an agent takes decisions to affect the state of the system. After each decision, the system updates its state, and then, the agent observes a "reward", or a score of how good that decision is. The agent's goal is to learn a strategy that maximizes its long-term cumulative reward. Mathematically, the interaction between the agent and the system is described by a 6-tuple $\langle \mathcal{S}, \mathcal{A}, p_0, p, R, \gamma \rangle$, where $\mathcal{S}$ denotes the state space (the set of all possible states), $\mathcal{A}$ the action space (set of all possible decisions), p_0 the initial state distribution (how the system starts out), $p(s_{t+1} \mid s_t, a_t)$ the state transition distribution (how the state changes given a decision), $R(s_t, a_t) \in \mathbb{R}$ is the reward function, and $\gamma \in [0, 1)$ the discount factor (a weight to discount future rewards). Many popular RL algorithms define a *Q-function*, which is similar to the cost-to-go function in dynamic programming and join-order optimization, that returns the maximum possible cumulative reward after taking an action in a specific state:

$$Q(s, a) = R(s, a) + \max_{a'} Q(S', a') \tag{5.1}$$

Intuitively, this function quantifies the long-term benefit of taking a specific action. These RL algorithms learn by estimating this Q function from data, i.e., observe how the system state evolves given actions and then pose a regression problem that relates states and actions to the long-term outcome. The estimated Q-function acts as the desired scoring metric. In our problem, the state is the set of all views current materialized, the action space is all feasible views that can be opportunistically generated, and the reward is the negative latency of the current query. The Q-function quantifies the long-term benefit of a particular materialization action in a given database state. At each decision step, we can develop a materialization policy by taking the highest scored action:

$$\pi(s) = \arg\max_{a} Q(s, a) \tag{5.2}$$

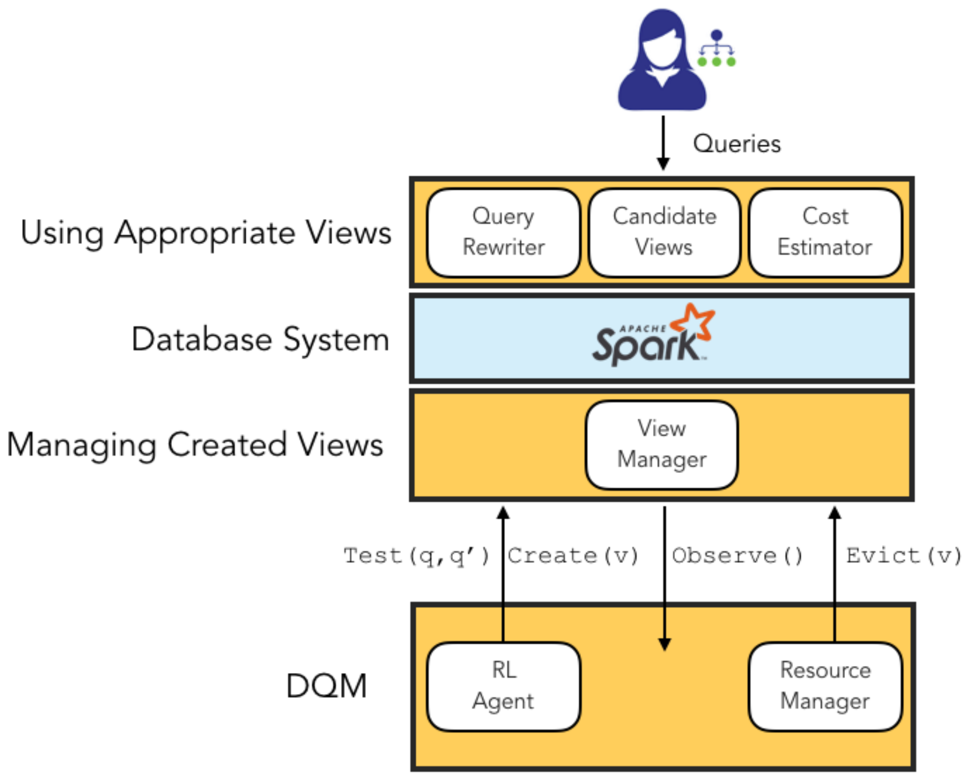


Figure 5.1: DQM runs as an independent process that issues view creation and deletion actions to Apache SparkSQL. A thin wrapper layer around SparkSQL manages the created views and returns any runtime results to DQM. DQM learns from these observations and issues creation and deletion events when appropriate. It also issues potential experiments to run.

5.3 System Overview

In our initial implementation, DQM considers select, inner join, and aggregate (SJA) views. DQM accepts general SparkSQL queries and decomposes each of the queries into Select-Project-Inner Join-Aggregate blocks:

```
SELECT a1,...,ak [agg(...)]
FROM T1,...,Tn
WHERE [pred(...)]
GROUP BY [g1,...,gm]
```

All rewriting, optimization, and materialization decisions are made at the granularity of the query blocks, and all other blocks types are ignored and passed through. In this section, we describe how DQM processes each block (thus, we will use query and block synonymously henceforth). The subsequent sections will describe the learning procedure in detail, but this section will treat it as a black box and assume we have a scoring function $Q()$ that can evaluate the benefit of persisting a view.

5.3.1 View Manager

Figure 5.1 illustrates the architecture of DQM. We found that it was convenient to run the OVM system in a separate process outside of Spark that issues the creation and deletion actions. All model training occurs in another Python process which interacts with the Spark environment via a RESTful API.

DQM stores all materialized views in the Parquet file format on an in-memory file system. The view manager maintains statistics about each of the views such as its size. The system scores each of these views with the learned scoring function $Q()$. The view manager also maintains logical descriptions of the views to avoid logically equivalent duplicates. The view manager stores a log that includes view usage and the latency of the query using the view. This log provides the training data for the learning module. The view manager is also responsible for evicting under-utilized views, we discuss this further in Section 5.

5.3.2 View Candidate Miner

When a SQL query arrives, the view candidate miner analyzes the query for potential OVM candidates–intermediate results that could be generated by executing the SQL query. Crucially, this step happens before query optimization. These candidates are logical do not necessarily come from a single physical query plan. In principle, any technique could be used to mine view candidates. We describe a small set of heuristics to construct reasonable candidates for the query types studied in this paper.

Join Candidates: We identify join views of all subsets of relations up-to a join_limit [1],

1. = 6 in our experiments

ignoring those joins that contain require a Cartesian product.

Filter Candidates: We also consider all single table filters as potential candidate views.

Aggregate Candidates: Any group by aggregate is included as a candidate view.

After identifying Join, Filter, and Aggregate candidates, the system scores each of these view candidates with the learned scoring function $Q()$. This creates a ranked list of view candidates by their score. We prune all views that are already materialized and those views with a lower score than the lowest scored materialized view. The remaining ranked list consists of potential views that are estimated to be more beneficial than those already stored. If this list is empty then no new views will be materialized.

5.3.3 Query Re-Writer

The query re-writer modifies the given query block in two passes. In Pass 1, it re-writes the query to force the materialization of the best view candidates. In Pass 2, it re-writes the query to best leverage previously computed views.

Pass 1. Opportunistic Optimization

In the first pass, DQM ensures that the given query creates the desired intermediate results. The view candidate miner sends the re-writer a ranked list of potential views to materialize that are more beneficial than at least one of the currently stored views. The first pass iterates over the ranked list in descending order. It determines whether it can rewrite the given query using each view. [2] If it can, then it rewrites the query:

```
pass1(query):
    for view in candidates_desc:
        if can_rewrite(query, view):
```

2. We use standard rewriting techniques that compose join, filter, and aggregate views. We do not handle cases of filter subsumption.

```
            query = rewrite(query, view)
```

We have to iteratively test the validity of each rewritten query because two views might be jointly incompatible: while the query can be rewritten to use both individually, together they do not form a valid rewritten plan. This iterative rewriting procedure ensures that the best view candidate is always materialized.

Pass 2. Reuse Optimization

The second pass is designed to maximize the reuse of persisted materialized views. The goal of this pass is different from the previous one. Pass 1 is meant to generate the best possible view candidates, Pass 2 is designed to execute the given query as efficiently as possible. In this pass, we leverage the query optimizer's cost model to only apply view rewriting that improve estimated performance.

Just because a valid rewriting exists doesn't mean it will improve query performance. This is counter-intuitive because one might think that pre-computation is guaranteed to save time. Consider the case of materializing a join view, and a future query that can utilize this view but also has a filter on one of the tables. If that filter is highly selective, forcing the query to use a join view may preclude a very effective filter push down optimization.

To deal with situations like this, assume a subroutine find_best(), which finds a view in the manager with the lowest query optimizer cost rewrite and has a lower cost than the cost of the original query. If a valid rewrite doesn't exist or the cost is high, then it returns `None`. Then, the second pass can be written as this iterative optimization routine.

```
pass2(query):
    view = find_best(view_mgr, query)
    while view != None:
        query = rewrite(query, view)
        view_mgr -= view
        view = find_best(view_mgr, query)
    return query
```

Discussion

In our implementation without much optimization effort, depends on the complexity of the query, the rewrite() function can take less than 1ms (for queries generated by CubeLoad) to up to 30ms (for some queries in JOB and TPC-DS) to complete. This two-pass query rewriting strategy ensures that views that can be generated opportunistically are planned for first before planning the query. An interesting consequence is that the system may take an instantaneously sub-optimal query planning decision for long-term benefit.

DQM learns this behavior by observing the consequences of its actions. It initially starts with a randomized scoring function $Q()$. As it materializes views, it observes rewards (the utility of these views to future queries). It then correlates rewards with materialization decisions. The process of learning from a randomized scoring function is called “exploration”. We will see that it will be beneficial to explore even after observing many such view, reward pairs. Exploration allows the system to hedge for changing, dynamic, or otherwise uncertain environments.

5.4 Learning Materialization

In this section, we discuss the core technical contribution of the paper: a reinforcement learning approach for adaptive view creation. In particular, this section describes how to learn $Q()$ described in the previous section. This section ignores concerns around eviction and assumes that there is an automated black-box system process that garbage collects old views when they fall into disuse. The next section will describe how to implement eviction.

5.4.1 The One View Problem

We start with a simplified problem: consider the decision of whether to materialize a single view candidate v. Our database has a one-bit state–whether v is currently materialized or not. Our system must simply decide when to apply the unary action to create a view if it is not materialized.

How do we quantify the "benefit" of this decision? For a single query q, let $\mathsf{Improvement}$(q,v) denote the difference in latency when using the view or not (the improvement is 0 if the view is not used):

$$\mathsf{Improvement}(q, v) = \mathsf{Latency}(q) - \mathsf{Latency}(q(v))$$

Over the entire future workload Q, the total improvement is:

$$\mathsf{Total\_Imp}(v) = \sum_{q \in Q} \mathsf{Improvement}(q, v)$$

$\mathsf{Improvement}$(q,v) hides a potential overhead on the first query (the query during which the view was created). Recall from the previous section, the two-pass rewriter may take a suboptimal query plan to materialize the view in the first place. $\mathsf{Total\_Imp}$(q,v) does not account for this overhead because it is a relative quantity.

Therefore, we introduce another term $\mathsf{Cost}(q_0$,v), which is an estimate of the creation overhead of the query (also includes any memory copy/persistence costs). The creation overhead is the difference between the execution time of an optimal query plan and that of the query that actually created the view:

$$\mathsf{Benefit}(v) = (\sum_{q \in Q} \mathsf{Improvement}(q, v)) - \mathsf{Cost}(q_0, v)$$

By explicitly decomposing the problem in this way, we can account for scenarios where the creation of a view may force an instantaneously suboptimal query plan but creates a view that benefits the cohort of other queries in the long run.

Towards A Reward Function

The broad goal of DQM is to create those views that optimize $\mathsf{Benefit}$(v). We make view usage decisions at "time-steps" that are synchronized with the query workload. So each query q_i defines a discrete-time decision point of whether to use the view or not. To be able to apply RL, we need a per-timestep (per-query) reward that quantifies the instantaneous benefit or harm of this action. So, we define the instantaneous benefit as the improvement for a given query and an amortized creation overhead over those queries that actually use the view:

$$R(q, v) = \mathsf{Improvement}(q, v) - \mathsf{Cost}(q_0, v) \cdot \frac{\delta(v, q)}{N_v},$$

where $\delta(v, q)$ is an indicator function determining whether q uses the view or not, and N_v is the number of times the view was used in the past. This means that each relevant query incurs a fractional creation cost. It can be verified that[3]:

$$\mathsf{Benefit}(v) = \sum_{q \in Q} R(q, v)$$

The per-query reward allows us to model the decision process as an MDP:

State: $M = \{0, 1\}$ view status, Q workload until q

Action: $\{\emptyset, +\}$: create the view or do nothing

Reward: $R(q, v)$: improvement minus amortized creation

Policy: $\pi(Q, M) \mapsto \{\emptyset, +\}$ decision to create view

Our objective is to find a view creation policy: given the current system state (i.e., whether the view is materialized or not and the query workload until the current point),

3. Additionally, we can scale $\mathsf{Cost}(q_0, v)$ by a hyperparameter to adjust unit differences.

decide the right time to create the view. RL is a framework that learns this policy through trial and error (explore random creation strategies) to optimize the cumulative reward, or $R(v)$ in our case.

Learning from Counterfacutal Experiments

The improvement metric compares two queries one that uses the view and one that doesn't. The latter is a query that the system would not ordinarily run. A counterfactual scenario is one that is contrary to what actually happened. There is a hypothetical (counterfactual) world in which v was not created and q was answered without using v (with a counterfactual runtime of $\mathsf{Latency}$(q)). We are really interested in the marginal improvement caused by an action we took in the system:

$$\mathsf{Improvement}(q, v) = \mathsf{Latency}(q) - \mathsf{Latency}(q(v))$$

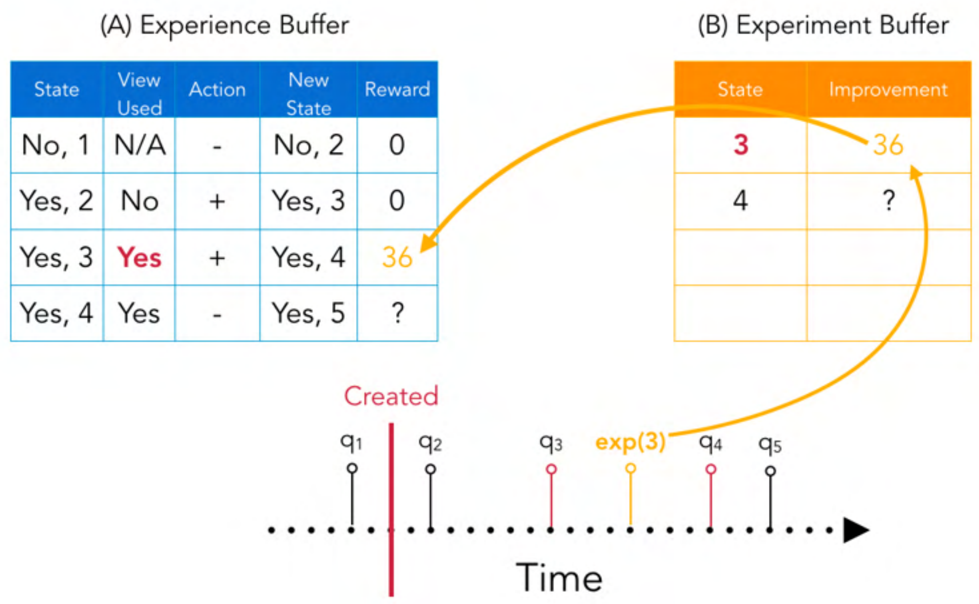
However, we cannot run both queries (with and without the view) in real-time as it would expose additional latency to the user. Our system maintains a running buffer of paired experiments to run. In idle times, it executes these experiments and stores the marginal improvement for each query.

Asynchronous RL Algorithm

The typical anatomy of an RL algorithm is to start with a randomly initialized policy and take decisions to affect the system. It observes the outcomes of its decisions. It periodically retrains the model based on these outcomes making the policy increasingly informed.

In our toy 1-view problem, the policy is simply to create the view or not at the current query. We now highlight the different parts of our algorithmic framework.

Rolling out (Data Collection): The core component of an RL algorithm is the "rollout" procedure. The algorithm starts off with a random policy (randomly choose whether to create



State	View Used	Action	New State	Reward
No, 1	N/A	-	No, 2	0
Yes, 2	No	+	Yes, 3	0
Yes, 3	Yes	+	Yes, 4	36
Yes, 4	Yes	-	Yes, 5	?

State	Improvement
3	36
4	?

Figure 5.2: We diagram the "rollout", or data collection, process used in DQM. At each time-step, DQM decides whether to create a view or not. A reward is received if a created view is used *AND* it improves a query runtime. Every time a created view is used, it queues up an experiment in (B) to run when the system is idle. Once the experiment is run, that observation of improvement is placed in (A) and can be used to improve the policy.

the view), then it observes its effects by applying it to the system. These observations need to be of the form:

$$(\text{state}, \text{action}, \text{reward}, \text{new state})$$

As the system observes the rewards accrued by different actions, it learns to assign credit to beneficial actions and incrementally builds a long-term strategy. As mentioned in the previous section, DQM collects this data in an asynchronous way by maintaining two buffers: an experience buffer and an experiment buffer. Figure 5.2 diagrams this process. At each time-step, DQM decides whether to create a view or not. If the cache is full, DQM will first evict a view (details of the eviction policy can be found in Section 5.6.4) before creating a new view. Every time a created view is used, it queues up a counterfactual experiment in (B) to run when the system is idle; run the query with and without the view. Once the experiment is run, the observed improvement is placed in (A) and can be used to improve the policy. In short, the experience buffer maintains a set of complete observations.

Policy Update: Our system continuously collects data and periodically updates the policy. Our RL algorithm is based on the Deep Q Neural Networks (DQN); which as an off-policy algorithm, is robust to asynchronous data collection. The DQN algorithm defines a *Q-function* (similar to the cost-to-go function):

$$Q(s,a) = R(s,a) + \max_{a'} Q(S',a') \tag{5.3}$$

Given the current state and action, what is the value of this action assume future optimal behavior. Of course, this function is hypothetical since having this function would imply having an optimal policy. If we had such a function, we would simply score each action and take the best valid action to make a decision:

$$\arg\max_{a} Q(s,a) \tag{5.4}$$

DQN iteratively approximates this function from data. Let Q_θ be a parametrized function (e.g., represented by a neural network):

$$Q_\theta(f_s, f_a) \approx Q(s,a)$$

where f_s is a *feature vector* representing the state and f_a is a feature vector representing the creation decision. In the 1-view problem, the state is simply a single binary variable of whether the view is materialized or not, and the action is another binary variable to create the view. In other words, *the DQN algorithm learns how to score actions in given states.*

θ is the model parameters that represent this function and is randomly initialized at the start. For each training tuple i in the experience buffer, one can calculate the following label, or the "estimated" Q-value:

$$y_i = R_i + \min_{a'} Q_\theta(s',a')$$

The $\{y_i\}$ can then be used as labels in a regression problem. If Q were the true Q-function, then the following recurrence would hold:

$$Q(s, a) = R_i + \min_{a'} Q_\theta(s', a')$$

So, the learning process, or *Q-learning*, defines a loss at each iteration:

$$L(Q) = \sum_i \|y_i - Q_\theta(s, a)\|_2^2$$

Then parameters of the Q-function can be optimized with gradient descent until convergence. The description above outlines the main theory behind Q-Learning. We also applied the tricks commonly used in practice like Experience Replay and Double DQNs [54].

5.4.2 Generalizing to N Views

For simplicity, we introduced the algorithm with a single view to create. The multiple view case when there is a pool of possible views to create is not that much harder. In principle, we can think of it as N-independent versions of the above algorithm. In fact, this extension is just a problem of featurization (how to define f_a and f_s).

When there is a single view featurization is trivial. There is a one-bit action variable and a one-bit state variable. When we move to N-views, we have to define a featurization scheme that can account for the particular views that are stored. This means that the set of views materialized are summarized as a feature vector and the particular view that you want to materialize is also summarized as a feature vector.

Let us start with the easier problem of describing the action (a particular view to materialize). To encode an action, we take a featurization approach that is similar to [78]. For each action, i.e a view candidate proposed by the view miner, our featurization focus on three components of its definition: (1) the join conditions, (2) the predicates and (3) the group by conditions.

For join conditions, to simplify our implementation without a loss of generalization, we assume there is only one way to join any two tables. Therefore, the join conditions can be encoded by applying one-hot encoding to encode the relations involved in the view. If there are multiple ways to join two tables, we can simply extend the featurization to the column level. For group by conditions, we apply one-hot encoding on the columns that are used in the group by clause of the view. The feature vector of a view is then the concatenation of the three vectors mentioned above. For the predicates defined in the view, we use a vector of size m, m equals to the number of columns in the database. The vector is initialized to be all 0s, for a predicate that is related to column c, we replace the 1 corresponding to column c with an estimated selectivity ϵ of the predicate generated by the query optimizer. Formally, for view v, let VJ, VG, VP represent the encoding of its join conditions, group-by conditions and predicates accordingly:

$$VJ_i = \begin{cases} 1 & \textsf{if table } i \textsf{ belongs to a join condition of } v \\ 0 & \textsf{otherwise} \end{cases}$$

$$VG_i = \begin{cases} 1 & \textsf{if table } i \textsf{ belongs to a group-by condition of } v \\ 0 & \textsf{otherwise} \end{cases}$$

$$VP_i = \begin{cases} \epsilon & \textsf{if column } i \textsf{ belongs to a predicate of } v \\ 0 & \textsf{otherwise} \end{cases}$$

And the final encoding of the view is simply the concatenation of VJ, VG, VP:

$$V = [VJ, VG, VP]$$

To encode the state which is a set of actions (i.e., views that are alive), we merge the action feature vectors together by two steps. We first take the disjunction of each column of the one-hot encoded features, i.e. features representing the join conditions and group-by conditions. Secondly, for the features representing predicates, for each column we take the maximum of the predicate selectivity estimations:

$$SJ_i = \begin{cases} 1 & \textsf{if table } i \textsf{ belongs to a join of an alive view} \\ 0 & \textsf{otherwise} \end{cases}$$

$$SG_i = \begin{cases} 1 & \textsf{if table } i \textsf{ belongs to a group-by of an alive view} \\ 0 & \textsf{otherwise} \end{cases}$$

$$SP_i = max(VP_{ji}|\ j \in \textsf{alive views}, i \in \textsf{columns})$$

Merging the action feature vectors results in a state feature vector of the same size as an action vector. An action vector and a state vector are concatenated together to represent an action state tuple and feed to the Q function as input.

5.5 View Eviction

In a pure RL setting, it is difficult to enforce a hard constraint, such as a storage limit, with a learned model. Therefore, we have to decouple the creation policy from the eviction policy, which independently enforces this constraint.

5.5.1 Submissive Eviction

Our eviction policy "submissive" to the RL algorithm in the sense that it's objective is to allow the RL algorithm to act as optimally as possible while enforcing the storage constraint. Whenever there's room to materialize the desired view, it allows the RL algorithm to make the decision. But if a certain decision exceeds the allotted space, it attempts to free up space such that the constraint can be enforced. We need an algorithm that is an inverse of the previous RL algorithm which maintains an estimate of the negative effects of evicting a

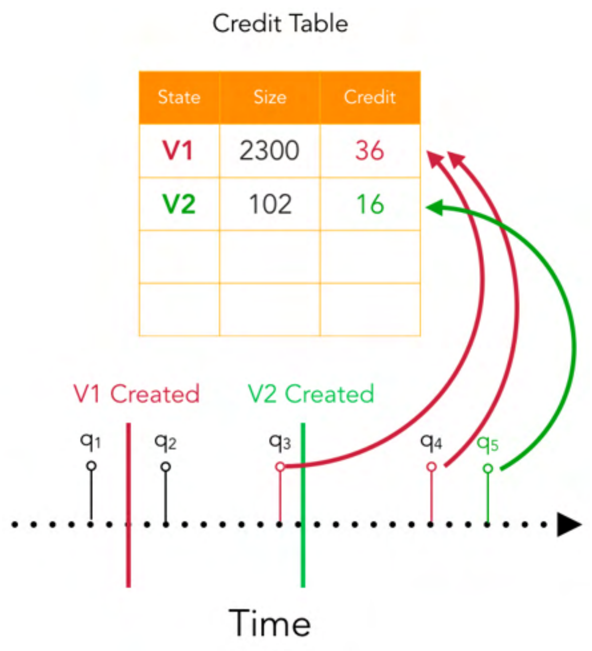


Figure 5.3: DQM maintains a table to prioritize which views to delete to enforce the storage constraint. This table is continuously updated with rewards accrued in the experience buffer.

view.

5.5.2 Algorithm

For each created view, the observed value of keeping it materialized is:

$$R^{-1}(v) = \sum_{q \in Q} \mathsf{Improvement}(q, v) + \mathsf{Cost}(q_0, v),$$

or the cumulative improvement so far plus the overhead cost of re-materializing the view. Unlike during creation, where Cost amortizes over each query, there is no such amortization. If we delete the view there is always a fixed cost of re-creating it. We maintain a running estimate of its current value as a member of the table (Figure 5.3), each time a view is used by a query:

$$C_{T+1}(v) = C_T(v) + R^{-1}(v).$$

Again, as with the view creation policy, we may have to tune hyper-parameters that scale the sum to account for different units or different preferences $\mathsf{Improvement}(q, v) + \gamma * \mathsf{Cost(v)}$.

One challenge is modeling dynamic workloads. If a view was very valuable in the early stage of a workload but then falls into disuse the credit table might have an inflated score. In practice, we decay the credits of each view by a rate of $\mu \in (0, 1]$[4]:

$$C_{T+1}(v) = \mu \cdot C_T(v) + R^{-1}(v).$$

Given the credit table, our eviction policy is to simply evict the view of lowest credit until sufficient space is freed up for the new view.

5.5.3 View Maintenance Through Eviction

We consider an OLAP setting where the materialized views are maintained infrequently. In this problem setting, it is sufficient to treat view maintenance as an automatic eviction event. For each view currently materialized, if one of its base tables have been updated, we evict it from the pool. After an eviction, we additionally flush the experiment buffer of any queued up experiments that use the view since the paired experimental results are now stale. Since we explicitly model $\mathsf{Cost}(v)$ and how it amortizes, our reward function is consistent under maintenance events, as creating a view that is repeatedly evicted will force the view to incur high creation costs that do not amortize well. This model is sufficient to capture maintenance through re-computation and not incremental view maintenance. We hope to explore modeling incremental view maintenance in detail in future work.

5.6 Experiments

Next, we present an experimental evaluation of DQM. All experiments are run on a cluster of machines running Scientific Linux 7.2 each with 2 Intel E5-2680 2.40 GHz CPUs and 64G memory. We run each experiment 3 times and present the average of the 3 runs. We

4. In practice, it is possible for a view to cause negative improvement, we do not decay a negative credit and the hyper-parameter we use to scale the cost will also be negative so that the cost became a penalty instead of a reward to its credit.

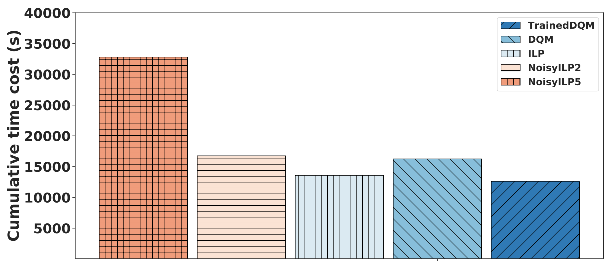


Figure 5.4: We evaluate overhead caused by learning and exploration by comparing DQM with a TrainedDQM with no exploration.

experiment with three datasets and workloads: Join Order Benchmark (JOB), TPC-DS, and CubeLoad. We run all of our microbenchmarks on JOB because it has a mix of join views and single-table filters.

5.6.1 Join Order Benchmark

The queries and data in this experiment are derived from the Join Order Benchmark (JOB). JOB is based on the IMDB dataset and consists of 113 aggregate queries with joins of up-to 16 tables. The workloads contain a sequence of 10,000 such queries and the queries are submitted and served in a sequential manner. Queries arrive at regular intervals, and the asynchronous experiments can be run in a single time-step (we evaluate these effects explicitly later). We normalize the storage constraint across all experiments. The available storage for opportunistic materialization is set to 200MB (roughly 20% the size of the largest base table in the experiments). This allows us to compare results across workloads in an apples-to-apples way.

We generate different scenarios from these benchmarks. Our default is called rzipf in which we draw queries randomly from the JOB benchmark based on a Zipf distribution, some queries appear much more frequently than others.

Comparison to Optimal View Selection

First, it is natural to compare DQM to optimal view selection algorithms. Jindal et al. formulate the problem with Integer Linear Programming (ILP) [63, 64]. They applied this

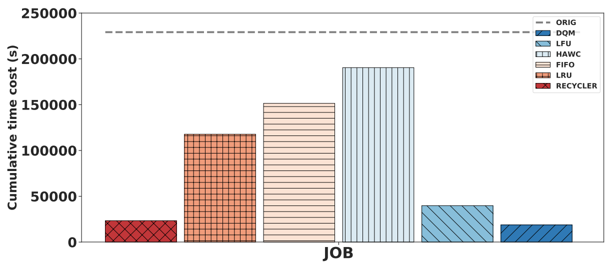


Figure 5.5: We compare DQM to all of the baselines on the rzipf workload. Even including learning time, DQM is competitive with the best baselines.

algorithm to identify reusable views in recurrent workloads.

Our JOB-based workload is stationary, i.e., 10000 queries are drawn from the same distribution, therefore it is highly likely that queries repeat. We construct an idealized setting for the ILP by collecting a perfect statistic for every (*view*, *query*) pair offline, then we use a solver to find the set of views that maximizes the overall utility under the given storage constraint. *The ILP technique is given perfect foresight of the workload.* However, such a perfect foresight might not be available which means the statistics could be noisy. To study how the noise affects ILP's performance we add random noise to the statistics. NoisyILP2 (NoisyIPL5) represents the situation where the noise is capped by a 2x (5x) of the true statistics accordingly. Figure 5.4 shows that the performance of ILP changes drastically as the noise level increases.

We first train DQM with 10000 queries from the workload from scratch (no prior knowledge) and compare the cumulative latency over the whole workload (Figure 5.4). DQM learns by proactively creating views (some of which are suboptimal, i.e., exploration). By definition, there will be an overhead to this process. Even including this overhead and the idealized ILP setting, DQM is actually competitive over the 10000 query workload. Now, if we were to run DQM for the same 10000 queries without any learning and exploration, DQM outperforms the ILP approach. This performance improvement is due to reasoning about the opportunistic creation overheads and handling eviction. This result suggests that DQM is able to learn a strategy that approaches an optimal allocation in a stationary workload.

Comparison to Caching Heuristics

Next, we compare DQM to different caching heuristics that range from conventional cache algorithms, like Least Recently Used (LRU), to sophisticated heuristic-based approaches from previous work. Unlike the previous experiment, we evaluate these algorithms *online*. All of the baselines benefit from the other components of DQM such as the candidate view miner. DQM proposes relevant views that can be opportunistically generated by the current query and relevant to the past workload. The baselines have to select which of these views to persist and evict existing views if necessary.

- **LRU:** Randomly select one of the candidates to materialize, evict the least recently used view in the store if full.
- **LFU:** Randomly select one of the candidates to materialize, evict the least frequently used view in the store if full.
- **FIFO:** Randomly select one of the candidates to materialize, evict the earliest view persisted in the store if full.
- **HAWC [114]:** Select the best view candidate based on the Spark query optimizer cost model. For each materialized view, maintain a "credit table" based on subsequent query cost that uses the view (cost difference of using vs. not using the view). The credit table is windowed to take the latest K queries and the lowest credit view is evicted.
- **RECYCLER [102]:** Select the most expensive view (in terms of creation cost). Materialize a new view if its cost is higher than existing views. Evict the lowest cost view otherwise. For views in the cache, the cost is scaled up when used, scaled down when not used. Our default implementation of Recycler makes use of the true costs of views, we further study a more practical alternative using cost model estimated view costs in Section 5.6.1.

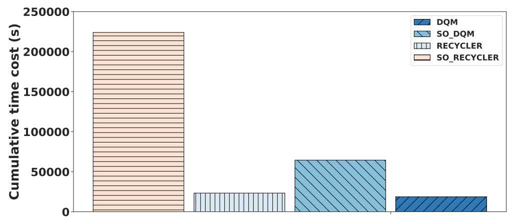


Figure 5.6: We run DQM without true runtimes and an improvement metric derived from a cost model. While this version of DQM still performs reasonably well, the use of true runtimes is a strength of the RL-based algorithm. We also evaluate a more realistic implementation of Recycler by using the cost model estimated view costs instead of the true costs and this change drastically affects Recycler's performance.

We evaluate DQM and the baselines on rzipf (Figure 5.5). We measure the cumulative runtime of the entire 10000-query workload. The neural network of DQM is initialized randomly and has to learn the creation and deletion policy online. *This exploration time for DQM is included in the overall runtime.*

Recycler works well when its creation cost heuristic correlates with improvements in runtime. Recycler speeds up query latency in JOB by about 10x. One caveat is that we provide Recycler with an *exact* cardinality estimate for the size of the views. While this is possible to know in hindsight after the views are created in order to prioritize deletions; it is impossible to know this exactly during creation time (i.e., a join cardinality estimation problem). Nonetheless, we are generous to Recycler as future experiments show that a faulty cardinality estimate very significantly affects results. HAWC uses a selection policy based on Spark's query optimizer but it has poor performance on this workload.

DQM is competitive with all baselines even when it has to learn. By leveraging real runtime observations, it is robust to cost estimation issues in the query optimizer. To us, this is a very surprising insight. There is overhead in the exploration process as the system has to learn from suboptimal actions. Even so, it is competitive with the best baselines during this learning phase.

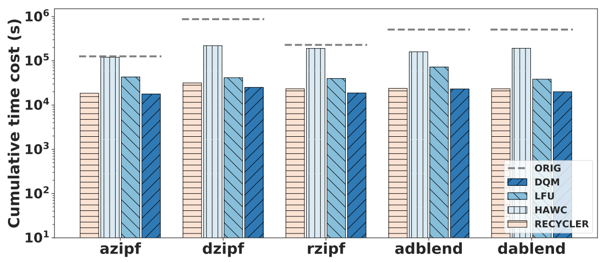
Cost Estimation Errors

In our previous results, the cost-aware baselines benefit from the exact view cardinality estimates (DQM does not use this as it learns purely from observed runtimes). Recycler is most sensitive to the accuracy of cardinality estimation. We implemented a more realistic alternative of Recycler called SO_Recycler. The only difference between the two is that SO_Recycler uses SparkSQL's query optimizer to estimate the cost of a view instead of using the true cost. As we can see in Figure 5.6, this change significantly affects Recycler's performance because the costs of views play an important rule in Recycler's heuristic: it assumes that a more expensive view will bring more benefits. Therefore, when the costs of views are inaccurate its performance drops drastically (about 10x).

We could do the opposite with DQM and examine the effect if we use Spark's optimizer for an inexpensive cost estimate rather than the counterfactual experiments. The difference in query cost using or not using the view can be used to determine the improvement. In this experiment, we modify DQM to use estimated reward from SparkSQL's query optimizer to investigate how it would affect the performance of DQM. We call the SparkSQL optimizer based version SO_DQM and results can be found in Figure 5.6. As mentioned in earlier sections, reward functions play an important role in RL systems, as it is designed to guide the model towards the direction of the highest long term value. Therefore, it is not surprising an RL system underperforms when its reward function is inaccurate or even wrong. SO_DQM still performs reasonably well but we believe that the power of RL is to feedback true execution times. Directly optimizing the true reward function explains much of the power of DQM.

Skew Considerations

We dig deeper on these baselines and consider different query skews and query distributions. We generate different scenarios. Our default is called rzipf in which we draw queries randomly from the JOB benchmark then apply the Zipf distribution to skew the frequency of the



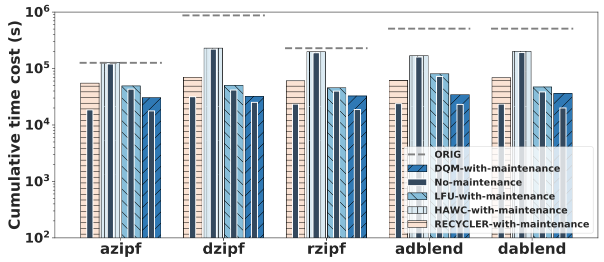


Figure 5.7: (A) We compare DQM to selected baselines on all workloads. (B) We compare DQM to selected baselines with periodic view maintenance (every 100 queries) on the JOB workloads.

queries. We go beyond rzipf and apply the power law to skew the frequency of queries as previous work has done [41, 90]. dzipf skews towards more expensive queries with a much higher frequency, and azipf skews towards the least expensive queries with a much higher frequency. dablend is a 10000-query workload that starts of executing the most expensive queries then switches to executing the least expensive queries, and adblend does the opposite. We evaluate DQM, LFU and the two heuristic-based approaches with 5 different workloads. Results are shown in Figure 5.7 A. We find that the results from the previous experiment broadly hold across all of the different skews.

Maintenance Considerations

The heuristics break down when there are costs that they do not model or anticipate. Maintenance costs in OLAP systems are infrequent but are significant. In this experiment, we study how view maintenance could affect DQM and the baselines. Because Spark does not support an incremental update of views, every time the base tables are modified we have to re-compute and re-materialize the views that are affected. To simulate periodic maintenance, our system will randomly select a base table of the workload and evict all views using the table.

We perform a controlled eviction routine at every 100 queries so all approaches have the same maintenance routine. Even with the same maintenance routine, different approaches will introduce different maintenance cost because different views are materialized.

Results can be found in Figure 5.7B. Maintenance certainly adds overhead to all tech-

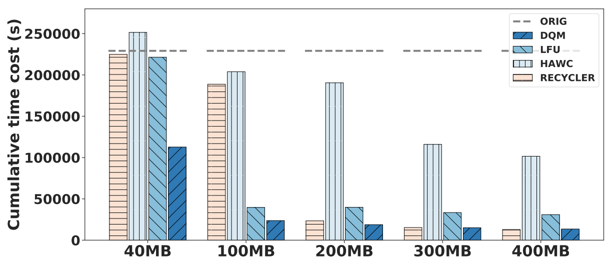


Figure 5.8: We measure the performance of DQM as a function of the storage constraint on the JOB rzipf workload. The storage constraint is presented as normalized by a fraction of candidate views that could possibly be materialized at any time.

niques, but the results demonstrate that DQM is more efficient and robust to maintenance. In the previous experiment, we found that Recycler was very effective on this workload. But after maintenance, we found DQM now outperforms Recycler non-trivially on all 5 workloads because Recycler's heuristic favors expensive views thus selecting views that incur a higher maintenance overhead than DQM. Again, the benefit of DQM is direct optimization of observed query latencies. Views that have to be constantly recreated because they are maintained fall out of favor of the learning algorithm quickly.

Storage Constraints

To study how DQM reacts to changes in the storage constraint, we use the same rzipf workload from the previous experiments. The result is shown in Figure 5.8, where the storage constraint is normalized by a fraction of candidate views that could possibly be materialized. OVM is most valuable when there is a substantial amount of spare storage in the system. The power of OVM is trading off this spare storage for future query latency. As expected, the performance of DQM and baselines improve as we increase the storage constraint. We note that the most significant increase is between 40MB and 100MB[5].

5. For reference the 200MB datapoint is the level used for DQM and all baselines in the previous experiments.

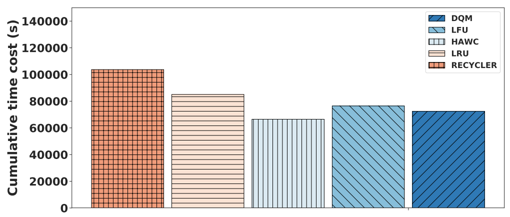


Figure 5.9: We compare DQM and selected baselines on the CubeLoad workload.

5.6.2 CubeLoad

CubeLoad [121] is a data-cubing workload that is designed to simulate user sessions exploring data. It simulates drilling down, augmenting queries with filters, and aggregating along different attributes. We use an IPUMS[124] data set that contains the census data. The workload contains 1000 sessions, each session of 5 to 10 queries with predicates and group-by conditions drawn from different dimensions of the data set.

We compare DQM with other baselines and the result is shown in Figure 5.9. The result shows that HAWC slightly outperforms DQM and other baselines. This is because the queries generated by CubeLoad are less complicated compared to the queries in the Join Order Benchmark, therefore the query optimizer provides a reliable estimation for HAWC to find the best views and its credit-based eviction policy also fits the workload well. On the other hand, we can see that even with the overhead of learning and exploration DQM is competitive to HAWC and outperforms other baselines.

5.6.3 TPC-DS

Next, we compare the selected baselines on the TPC-DS benchmark. We use a scale factor of 1 and generate queries from the TPC-DS query list. We use the same sampling distributions described before and present cumulative performance on 10000 queries. The results are shown in Figure 5.10. Most notably Recycler, which was the best technique on JOB, is no longer performant. On the other hand, HAWC performs much better. But its drastic performance shift on the two workloads indicates optimizer based selection policy is not reliable. DQM is competitive with all baselines on both benchmarks even when it has to

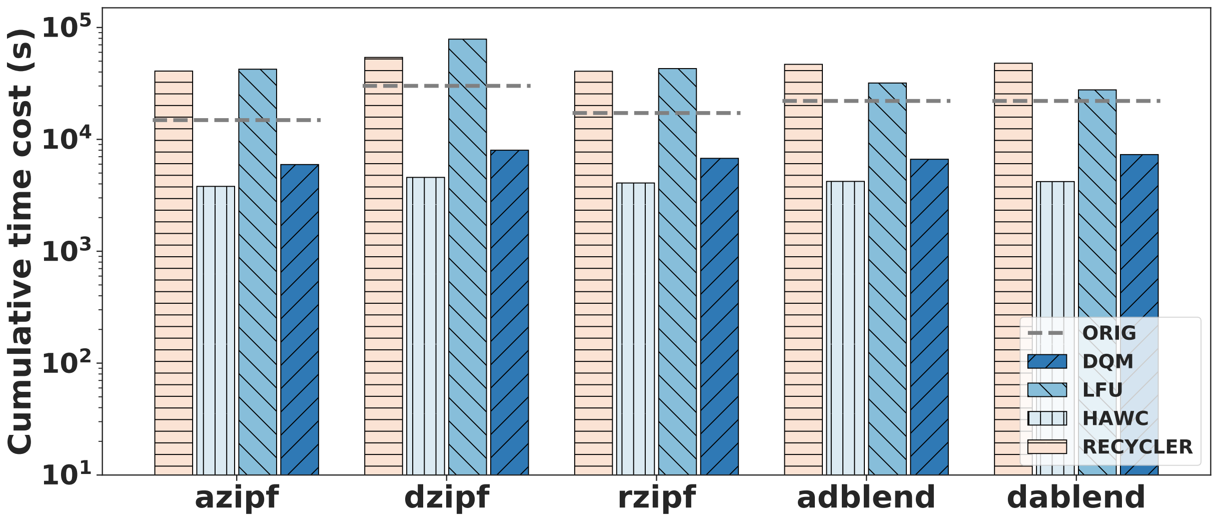


Figure 5.10: We compare DQM to selected baselines on TPC-DS. DQM is competitive (or outperforms) the best heuristic on all the test scenarios.

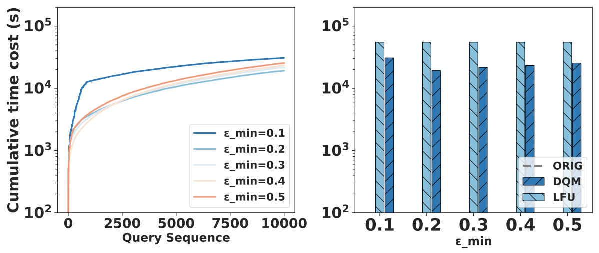


Figure 5.11: DQM using different exploration terms on JOB with the rzipf workload.

learn. As before the learning overheads are included in the cumulative runtimes.

5.6.4 Micro-Benchmarks

Next, we summarize a number of important results pertaining to different parameter choices in DQM. Unless otherwise noted, all of these experiments run on the JOB rzipf workload.

Exploration vs. Exploitation

DQM starts with random selection to explore until there is enough experience to start training and as observations come it, it periodically re-trains its model. As we collect more observations, we become more confident about DQM, and then we start to explore less with random actions. The exploration parameter ϵ represents the exploration rate and $1 - \epsilon$

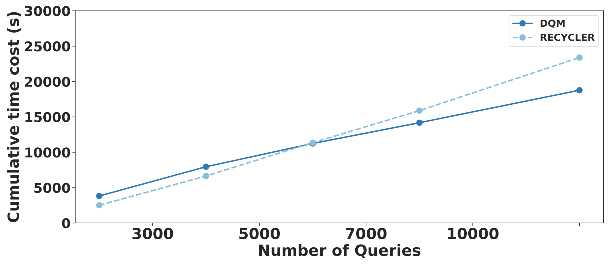


Figure 5.12: We study the performance of DQM with different workload size.

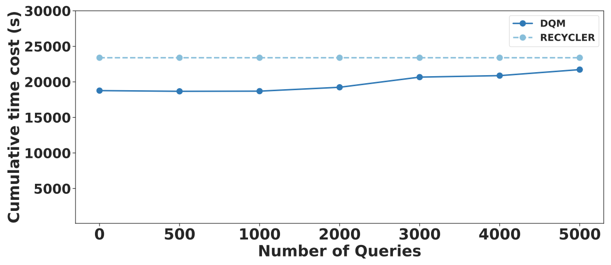


Figure 5.13: We measure the performance of DQM using different delays of reward on JOB rzipf workload.

represents the probability of exploiting what we have learned. We must always have some degree of random creation to ensure that DQM is adaptive to changes. Our system starts from an $\epsilon = 1$ (always take random actions) and decays this value to ϵ_min. We evaluate DQM using different ϵ_min.

We set ϵ_min to 0.1, 0.2, 0.3, 0.4, 0.5 and results are shown in Figure 5.11[6]. The figure on the left shows the trend of processing the JOB-based rzipf workload with different ϵ_min and we compare with LFU on the right figure. As expected, exploration is necessary for DQM to avoid settling on a local optimal, but as we learn more we should prefer more exploitation and a high ϵ_min hurts the performance. The results indicate that an ϵ_min of 0.2 is a good compromise between exploration and exploitation. Depends on the size of the action space, a smaller ϵ_min would work better for a smaller action space and a larger ϵ_min have the potential to find better views in larger action spaces.

6. Other experiments use a fixed ϵ_min of 0.1.

Number of Training Queries

The performance of DQM relies on the experiences it can learn from and the number of experiences increases as DQM process more queries. In this experiment, we demonstrate how the performance of DQM changes with different numbers of training queries. As shown in Figure 5.12, initially, Recycler outperforms DQM from a cold start with an empty experience set. As DQM starts to take actions and collect feedback from the experiments, it starts to learn and becomes better at selecting views. We can see that as the number of experiences increases, the gap between the two shrinks. At around the 5000th query, DQM starts to outperforms Recycler and eventually outperforms Recycler by 20% after 10,000 queries.

Delayed Rewards

In this experiment, we explore how delays in the asynchronous experimentation affect DQM. DQM relies on system idle time to execute paired experiments, what if this idle time is contended? We simulate this in the following way: given a delay of K, an experience that is generated at time step T will only be available to DQM for learning at time step $T + K$. As an extreme example, if K equals 10000, DQM will select views completely randomly for our 10000 query workloads.

This is a worst-case simulation of delayed reward. In practice, the system idle time will likely be more randomly distributed and some queries might get earlier observations. However, by pushing all the experiences to the end of the process, we are simulating the worst case of delayed rewards, i.e. if the same amount of experiences were thrown away but system idle time was more randomly distributed, then more experiences will be available earlier for the agent to learn from thus benefiting the system.

We use the rzipf workload from JOB in this experiment and test a delay of 500, 1000, 2000, 3000, 4000 and 5000. The results can be found in Figure 5.13. We see that the performance of DQM deteriorates as we increase the delay, but even with a 5000 query delay DQM still outperforms Recycler. A further investigation shows that Recycler outperforms

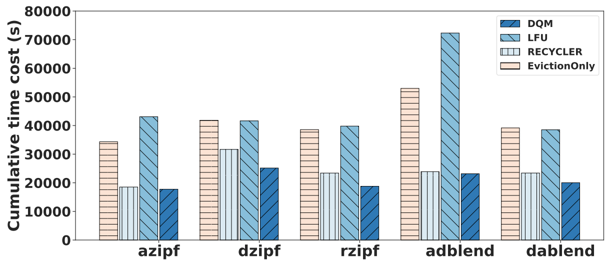


Figure 5.14: We combine our eviction policy with random view selection then compare its performance with DQM and selected baselines.

DQM by 25% at query 5000, but the performance of DQM start to improve 500 queries after the rewards are available (around query 5500), and after another 3000 queries (around query 8500) DQM outperforms Recycler.

Ablation Study

To understand how much the RL-based view selection and the eviction policy contribute to the overall performance, we build another baseline which uses random view selection with our eviction policy (called Eviction-Only). This baseline differs from DQM only in its view selection thus the performance difference is due to different view selection policy. The results in Figure 5.14 show that our eviction policy which takes both the benefit and the cost into account outperforms or is competitive with LFU (which also use random view selection but with the LFU eviction policy) on all 5 workloads. However, DQM is almost 2x more effective than the eviction-only baseline, which indicates the RL-based view selection is the key to our overall framework to outperform other heuristic-based approaches.

Comparison to Optimal Usage Heuristics

Belady* is a hypothetical baseline whose eviction policy is based on the Belady's algorithm[13], which is hypothetical because it relies on hindsight to evict a view that will not be needed for the longest time in the future. Belady* is also hypothetical because it uses a perfect foresight so that it always selects the most beneficial view to materialize every time a query arrives. Belady*'s policy is optimal in terms of "usage", however, views have different costs and utility. It might evict a more beneficial view that was materialized earlier to make room for a less beneficial one that is optimal for the current query. It could also frequently evict and re-materialize an expensive view and cause extra overhead.

We evaluate Belady* on the JOB, TPC-DS, and CubeLoad workloads and found DQM outperforms Belady* on all 5 JOB workloads by an average improvement of 45%. Belady* outperforms DQM on the TPC-DS workloads by an average improvement of 49%. DQM outperforms Belady* on CubeLoad by 1%. Further investigation shows that on JOB workloads, initially, Belady* works better than DQM but with more training and exploration DQM eventually outperforms Belady* after a couple thousands of queries. On TPC-DS workloads, it happens to be the case that the best views are also cheap, therefore, due to Belady*'s hypothetical optimal view selection policy it only takes 4% of the view creation cost of DQM and outperforms DQM. Overall, DQM is competitive with Belady* and we believe this result again testifies the robustness of DQM even when compared with a baseline using the best hypothetical heuristics.

5.7 Conclusion

There are numerous opportunities for reusing computation and intermediate query state in OLAP workloads, and we believe that machine learning will be an important part of future OLAP systems. We see DQM as a first step towards a View-Oriented database, one that aggressively anticipates future queries and materializes anything that could be useful. Such an architecture shifts the query optimization burden from planning a query to efficiently reusing past computation. New algorithms and theory will have to develop to understand the new problem setting. We believe machine learning will be an important part of this discussion. In the short-term, extending DQM to consider OLTP settings and more complex reward functions is certainly a priority. We also want to explore dynamic or periodic workloads.

CHAPTER 6
CONCLUSION

6.1 Conclusion

The massive growth of data provides opportunities for better decision-making processes that are data-driven instead of heuristic/experience-driven. However, data-driven decision-making should not be taken for granted: from storing to accessing and analyzing the data, every step in the data processing pipeline has to be fast, efficient, and scalable in order to deliver the final result.

On one hand, we have techniques like columnar storage, compression that makes data processing more efficient. On the other hand, in scenarios where some inaccuracy can be tolerated, there is also an increase of interest in approximated query processing because AQP can be used to provide a tradeoff between accuracy and cost.

In this thesis, we propose 4 techniques for approximate query processing of different characteristics.

First, we propose Precomputation Assisted Stratified Sampling (PASS), a framework for combining precomputed aggregates with stratified sampling. PASS provides a novel approach to bridge the two main categories of AQP techniques: sampling-based and precomputation-based techniques. PASS proposes a data structure called a static partition tree (SPT) that makes it possible for the AQP engine to perform aggressive yet reliable sample-skipping during query processing. Because dealing with samples is the main source of both inaccuracy and overhead, being able to reduce the usage of samples without sacrificing accuracy drastically improves both the accuracy and efficiency of the system at the cost of some offline processing time.

PASS works great in a static setting where data stays unchanged, however, that is not always the case. In real life, the data we are dealing with is constantly growing, being modified, or even getting deleted. To address this limitation of PASS, we propose JanusAQP, a dynamic AQP system that is designed to work in a dynamic environment. JanusAQP includes several techniques that lead to a dynamic partition tree (DPT) that can work with the insertion and deletion of data (implicitly also modification, i.e. a deletion followed by an insertion). A new partitioning algorithm is designed to improve the partitioning cost which is the main overhead for building an SPT. Furthermore, a catch-up phase is used for JanusAQP to work with an arbitrarily large amount of historical data in an existing system and JanusAQP is integrated with Apache Kafka as a consumer of the message queue to demonstrate how it could be used in real-life scenarios – without changing the existing physical design of the system.

Thirdly, we explore another scenario where AQP can be useful – missing data analysis. We propose a framework called Predicate Constraint (PC) that not only enables the formalization of our beliefs on the missing data but can also be used to answer queries. As a missing data analysis framework, PC is optimal in terms of reliability, i.e. the ability to quantify the error, which is probably the most important metric for missing data analysis. This is because the ability to quantify the error determines the quality of the decision that is made based on the estimated result. By modeling the problem with integer linear programming, PC can generate the best answer which is the optimal hard-bound that leads to a 0 failure rate which is a clear advantage against all the baselines. It is worth mentioning that the idea of predicate constraint also plays a role in the design of SPT and DPT.

Lastly, in the DQM project, we study the management of materialized views using deep reinforcement learning. As our first trial to apply machine learning techniques in database systems, we propose to use a deep reinforcement learning model to learn from the rewards and penalties of managing the materialized views. The model gets to decide which views to keep or get rid of under a storage constraint, and based on the consequences of the past decisions, the model improves and adapts to potential changes (e.g. change of the workload patterns, etc.) with an objective of improving the overall query processing cost. We integrate DQM in Apache Spark and the experiment results show that DQM is more robust than other heuristic-based techniques.

AQP has been studied by academia for decades but there's still a long way to go for it to be widely utilized in industry. Not only do we need to come up with better AQP techniques but we also need a better way to represent the 'inaccuracy' of the answer.

Confidence intervals used by most AQP techniques quantify the error but it is not intuitive and it does not help an average user to put trust in the AQP technique that generates the answer. To this end, a recent paper by Google [8] proposes to trade instead of the accuracy but the freshness of the answer for a lower cost. In a sense, the answer is inaccurate with regard to the most up-to-date data, but it is accurate for an older version of the data. In my opinion, this is probably the best way to present the inaccuracy to a user that is easy to understand – but Google might not agree with me that their technique should be categorized as AQP.

References

[1] Nasdaq bookviewer, 2021.

[2] S. Acharya, P. B. Gibbons, and V. Poosala. Aqua: A fast decision support system using approximate query answers. In *In Proc. of 25th Intl. Conf. on Very Large Data Bases*. Citeseer, 1999.

[3] 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*, pages 487–498, 2000.

[4] P. K. Agarwal, G. Cormode, Z. Huang, J. Phillips, Z. Wei, and K. Yi. Mergeable summaries. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI symposium on Principles of Database Systems*, pages 23–34, 2012.

[5] S. Agarwal, H. Milner, A. Kleiner, A. Talwalkar, M. Jordan, S. Madden, B. Mozafari, and I. Stoica. Knowing when you're wrong: building fast and reliable approximate query processing systems. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 481–492. ACM, 2014.

[6] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 29–42. ACM, 2013.

[7] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 29–42, 2013.

[8] A. Agiwal, K. Lai, G. N. B. Manoharan, I. Roy, J. Sankaranarayanan, H. Zhang, T. Zou, M. Chen, J. Chen, M. Dai, et al. Napa: Powering scalable data warehousing with robust query performance at google. 2021.

[9] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in sql databases. In *VLDB*, volume 2000, pages 496–505, 2000.

[10] Airbnb. Airbnb new york city open data 2019. `https://www.kaggle.com/dgomonov/new-york-city-airbnb-open-data`, 2019.

[11] J. Arulraj, R. Xian, L. Ma, and A. Pavlo. Predictive indexing. *arXiv preprint arXiv:1901.07064*, 2019.

[12] B. Babcock, S. Chaudhuri, and G. Das. Dynamic sample selection for approximate query processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 539–550, 2003.

[13] L. A. Belady, R. A. Nelson, and G. S. Shedler. An anomaly in space-time characteristics of certain programs running in a paging machine. *Commun. ACM*, 12(6):349–353, June 1969.

[14] G. E. Box, W. G. Hunter, J. S. Hunter, et al. Statistics for experimenters. 1978.

[15] J. Brito, K. Demirkaya, B. Etienne, Y. Katsis, C. Lin, and Y. Papakonstantinou. Efficient approximate query answering over sensor data with deterministic error guarantees. *arXiv preprint arXiv:1707.01414*, 2017.

[16] N. Bruno and S. Chaudhuri. Automatic physical database tuning: a relaxation-based approach. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 227–238. ACM, 2005.

[17] P. Buneman, S. Khanna, and T. Wang-Chiew. Why and where: A characterization of data provenance. In *International conference on database theory*, pages 316–330. Springer, 2001.

[18] D. Burdick, P. M. Deshpande, T. Jayram, R. Ramakrishnan, and S. Vaithyanathan. Efficient allocation algorithms for OLAP over imprecise data. In *Proceedings of the 32nd international conference on Very large data bases*, pages 391–402. VLDB Endowment, 2006.

[19] J. Camacho-Rodríguez, D. Colazzo, M. Herschel, I. Manolescu, and S. Roy Chowdhury. Reuse-based optimization for pig latin. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, pages 2215–2220. ACM, 2016.

[20] T. M. Chan and K. Tsakalidis. Dynamic orthogonal range searching on the ram, revisited. *Leibniz International Proceedings in Informatics, LIPIcs*, 77:281–2813, 2017.

[21] S. Chandrasekaran and M. J. Franklin. Psoup: a system for streaming queries over streaming data. *The VLDB Journal*, 12(2):140–156, 2003.

[22] S. Chaudhuri, G. Das, M. Datar, R. Motwani, and V. Narasayya. Overcoming limitations of sampling for aggregation queries. In *Proceedings 17th International Conference on Data Engineering*, pages 534–542. IEEE, 2001.

[23] S. Chaudhuri, G. Das, and V. Narasayya. A robust, optimization-based approach for approximate answering of aggregate queries. *ACM SIGMOD Record*, 30(2):295–306, 2001.

[24] S. Chaudhuri, G. Das, and V. Narasayya. Optimized stratified sampling for approximate query processing. *ACM Transactions on Database Systems (TODS)*, 32(2):9–es, 2007.

[25] S. Chaudhuri, B. Ding, and S. Kandula. Approximate query processing: No silver bullet. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 511–519, 2017.

[26] S. Chaudhuri, V. Narasayya, and R. Ramamurthy. A pay-as-you-go framework for query execution feedback. *Proceedings of the VLDB Endowment*, 1(1):1141–1152, 2008.

[27] C. M. Chen and N. Roussopoulos. The implementation and performance evaluation of the adms query optimizer: Integrating query result caching and matching. In *International Conference on Extending Database Technology*, pages 323–336. Springer, 1994.

[28] X. Chu, I. F. Ilyas, S. Krishnan, and J. Wang. Data cleaning: Overview and emerging challenges. In *Proceedings of the 2016 international conference on Management of Data*, pages 2201–2206. ACM, 2016.

[29] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[30] G. Cormode. Sketch techniques for approximate query processing. *Foundations and Trends in Databases. NOW publishers*, 2011.

[31] G. Cormode, M. Garofalakis, P. J. Haas, C. Jermaine, et al. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends® in Databases*, 4(1–3):1–294, 2011.

[32] G. Cormode, M. Garofalakis, P. J. Haas, C. Jermaine, et al. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends® in Databases*, 4(1–3):1–294, 2011.

[33] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zait, and M. Ziauddin. Automatic sql tuning in oracle 10g. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 1098–1109. VLDB Endowment, 2004.

[34] M. De Berg, M. Van Kreveld, M. Overmars, and O. C. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, 3rd edition, 2008.

[35] L. de Moura and N. Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[36] P. M. Deshpande, K. Ramasamy, A. Shukla, and J. F. Naughton. Caching multidimensional queries using chunks. In *ACM SIGMOD Record*, volume 27, pages 259–270. ACM, 1998.

[37] D. Deutch, Z. G. Ives, T. Milo, and V. Tannen. Caravan: Provisioning for what-if analysis. 2013.

[38] B. Ding, S. Huang, S. Chaudhuri, K. Chakrabarti, and C. Wang. Sample+ seek: Approximating aggregates with distribution precision guarantee. In *Proceedings of the 2016 International Conference on Management of Data*, pages 679–694, 2016.

[39] C. Ding, D. Tang, X. Liang, A. J. Elmore, and S. Krishnan. Ciao: An optimization framework for client-assisted data loading. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 1979–1984. IEEE, 2021.

[40] D. Donjerkovic, R. Ramakrishnan, and Y. Ioannidis. Dynamic histograms: Capturing evolving data sets. In *Proceedings of 16th International Conference on Data Engineering*, pages 86–86. IEEE Computer Society, 2000.

[41] A. L. Drapeau, D. A. Patterson, and R. H. Katz. Toward workload characterization of video server and digital library applications. *ACM SIGMETRICS Performance Evaluation Review*, 22(1):274–275, 1994.

[42] I. Elghandour and A. Aboulnaga. Restore: reusing results of mapreduce jobs. *Proceedings of the VLDB Endowment*, 5(6):586–597, 2012.

[43] E. Friedgut. Hypergraphs, entropy, and inequalities. *The American Mathematical Monthly*, 111(9):749–760, 2004.

[44] A. Galakatos, A. Crotty, E. Zgraggen, C. Binnig, and T. Kraska. Revisiting reuse for approximate query processing. *Proceedings of the VLDB Endowment*, 10(10):1142–1153, 2017.

[45] E. Gan, P. Bailis, and M. Charikar. Coopstore: Optimizing precomputed summaries for aggregation. *Proceedings of the VLDB Endowment*, 13(11).

[46] E. Gan, P. Bailis, and M. Charikar. Coopstore: Optimizing precomputed summaries for aggregation. *Proceedings of the VLDB Endowment*, 13(12):2174–2187, 2020.

[47] V. Ganti, M.-L. Lee, and R. Ramakrishnan. Icicles: Self-tuning samples for approximate query answering. In *VLDB*, volume 176. Citeseer, 2000.

[48] M. N. Garofalakis and P. B. Gibbons. Approximate query processing: Taming the terabytes. In *VLDB*, volume 10, pages 645927–672356, 2001.

[49] P. B. Gibbons, Y. Matias, and V. Poosala. Fast incremental maintenance of approximate histograms. *ACM Transactions on Database Systems (TODS)*, 27(3):261–298, 2002.

[50] A. C. Gilbert, S. Guha, P. Indyk, Y. Kotidis, S. Muthukrishnan, and M. J. Strauss. Fast, small-space algorithms for approximate histogram maintenance. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 389–398, 2002.

[51] Google. Google protocol buffer, 2021.

[52] S. Guha, N. Koudas, and K. Shim. Approximation and streaming algorithms for histogram construction problems. *Trans. on Datab. Syst.*, 31(1):396–438, 2006.

[53] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: Automatic management of data and computation in datacenters.

[54] H. V. Hasselt. Double q-learning. In J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems 23*, pages 2613–2621. Curran Associates, Inc., 2010.

[55] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *Acm Sigmod Record*, volume 26, pages 171–182. ACM, 1997.

[56] B. Hilprecht, A. Schmidt, M. Kulessa, A. Molina, K. Kersting, and C. Binnig. Deepdb: Learn from data, not from queries! *VLDB Endowment*, 2019.

[57] T. Imieliński and W. Lipski Jr. Incomplete information in relational databases. In *Readings in Artificial Intelligence and Databases*, pages 342–360. Elsevier, 1989.

[58] I. Inc. Instacart dataset 2017. `https://www.instacart.com/datasets/grocery-shopping-2017`, 2017.

[59] H. Jagadish, H. Jin, B. C. Ooi, and K.-L. Tan. Global optimization of histograms. *ACM SIGMOD Record*, 30(2):223–234, 2001.

[60] H. V. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K. C. Sevcik, and T. Suel. Optimal histograms with quality guarantees. In *VLDB*, volume 98, pages 24–27, 1998.

[61] C. Jermaine. Robust estimation with sampling and approximate pre-aggregation. In *Proceedings 2003 VLDB Conference*, pages 886–897. Elsevier, 2003.

[62] R. Jin, L. Glimcher, C. Jermaine, and G. Agrawal. New sampling-based estimators for OLAP queries. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 18–18. IEEE, 2006.

[63] A. Jindal, K. Karanasos, S. Rao, and H. Patel. Selecting subexpressions to materialize at datacenter scale. *Proceedings of the VLDB Endowment*, 11(7):800–812, 2018.

[64] A. Jindal, S. Qiao, H. Patel, Z. Yin, J. Di, M. Bag, M. Friedman, Y. Lin, K. Karanasos, and S. Rao. Computation reuse in analytics job service at microsoft. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 191–203, New York, NY, USA, 2018. ACM.

[65] N. Johnson, J. P. Near, and D. Song. Towards practical differential privacy for sql queries. *Proceedings of the VLDB Endowment*, 11(5):526–539, 2018.

[66] S. Joshi and C. Jermaine. Materialized sample views for database approximation. *IEEE Transactions on Knowledge and Data Engineering*, 20(3):337–351, 2008.

[67] M. Jurgens and H.-J. Lenz. The r/sub a/*-tree: an improved r*-tree with materialized data for supporting range queries on olap-data. In *Proceedings Ninth International Workshop on Database and Expert Systems Applications (Cat. No. 98EX130)*, pages 186–191. IEEE, 1998.

[68] N. Kamat, P. Jayachandran, K. Tunga, and A. Nandi. Distributed and interactive cube exploration. In *2014 IEEE 30th International Conference on Data Engineering*, pages 472–483. IEEE, 2014.

[69] S. Kandula, K. Lee, S. Chaudhuri, and M. Friedman. Experiences with approximating queries in microsoft's production big-data clusters. *Proceedings of the VLDB Endowment*, 12(12):2131–2142, 2019.

[70] S. Kandula, A. Shanbhag, A. Vitorovic, M. Olma, R. Grandl, S. Chaudhuri, and B. Ding. Quickr: Lazily approximating complex adhoc queries in bigdata clusters. In *Proceedings of the 2016 international conference on management of data*, pages 631–646. ACM, 2016.

[71] O. Kennedy and B. Glavic. Analyzing uncertain tabular data. In *Information Quality in Information Fusion and Decision Making*, pages 243–277. Springer, 2019.

[72] D. Kossmann, M. J. Franklin, G. Drasch, and W. Ag. Cache investment: integrating query optimization and distributed data placement. *ACM Transactions on Database Systems (TODS)*, 25(4):517–558, 2000.

[73] Y. Kotidis and N. Roussopoulos. Dynamat: a dynamic view management system for data warehouses. In *ACM Sigmod record*, volume 28, pages 371–382. ACM, 1999.

[74] N. Koudas, S. Muthukrishnan, and D. Srivastava. Optimal histograms for hierarchical range queries. In *Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 196–204, 2000.

[75] T. Kraska. Northstar: An interactive data science system. *Proceedings of the VLDB Endowment*, 11(12):2150–2164, 2018.

[76] S. Krishnan, J. Wang, M. J. Franklin, K. Goldberg, and T. Kraska. Stale view cleaning: Getting fresh answers from stale materialized views. *Proceedings of the VLDB Endowment*, 8(12):1370–1381, 2015.

[77] S. Krishnan, J. Wang, M. J. Franklin, K. Goldberg, T. Kraska, T. Milo, and E. Wu. Sampleclean: Fast and reliable analytics on dirty data. 2015.

[78] S. Krishnan, Z. Yang, K. Goldberg, J. Hellerstein, and I. Stoica. Learning to optimize join queries with deep reinforcement learning. *arXiv preprint arXiv:1808.03196*, 2018.

[79] M. Kunjir, B. Fain, K. Munagala, and S. Babu. Robus: Fair cache allocation for data-parallel workloads. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 219–234. ACM, 2017.

[80] I. Lazaridis and S. Mehrotra. Progressive approximate aggregate queries with a multi-resolution tree structure. *Acm sigmod record*, 30(2):401–412, 2001.

[81] J. LeFevre, J. Sankaranarayanan, H. Hacigümüş, J. Tatemura, and N. Polyzotis. Towards a workload for evolutionary analytics. In *Proceedings of the Second Workshop on Data Analytics in the Cloud*, pages 26–30. ACM, 2013.

[82] J. LeFevre, J. Sankaranarayanan, H. Hacigumus, J. Tatemura, N. Polyzotis, and M. J. Carey. Opportunistic physical design for big data analytics. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 851–862. ACM, 2014.

[83] G. Li, X. Zhou, and S. Li. Xuanyuan: An ai-native database. *Data Engineering*, page 70, 2019.

[84] K. Li and G. Li. Approximate query processing: What is new and where to go? *Data Science and Engineering*, 3(4):379–397, 2018.

[85] X. Liang, A. J. Elmore, and S. Krishnan. Opportunistic view materialization with deep reinforcement learning, 2019.

[86] X. Liang, Z. Shang, S. Krishnan, A. J. Elmore, and M. J. Franklin. Fast and reliable missing data contingency analysis with predicate-constraints. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 285–295, 2020.

[87] X. Liang, S. Sintos, Z. Shang, and S. Krishnan. Combining aggregation and sampling (nearly) optimally for approximate query processing. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1129–1141, 2021.

[88] S. Liu, B. Song, S. Gangam, L. Lo, and K. Elmeleegy. Kodiak: leveraging materialized views for very low-latency analytics over high-dimensional web-scale data. *Proceedings of the VLDB Endowment*, 9(13):1269–1280, 2016.

[89] Z. Liu, B. Jiang, and J. Heer. immens: Real-time visual querying of big data. In *Computer Graphics Forum*, volume 32, pages 421–430. Wiley Online Library, 2013.

[90] E. Lo, N. Cheng, and W.-K. Hon. Generating databases for query workloads. *Proceedings of the VLDB Endowment*, 3(1-2):848–859, 2010.

[91] Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, H. Woo, B. G. Lindsay, and J. F. Naughton. Middle-tier database caching for e-business. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 600–611. ACM, 2002.

[92] 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. In *Proceedings of the 2018 International Conference on Management of Data*, pages 631–645. ACM, 2018.

[93] Q. Ma, A. M. Shanghooshabad, M. Almasi, M. Kurmanji, and P. Triantafillou. Learned approximate query processing: Make it light, accurate and fast. In *CIDR*, 2021.

[94] I. Mami and Z. Bellahsene. A survey of view selection methods. *ACM SIGMOD Record*, 41(1):20–29, 2012.

[95] R. Marcus and O. Papaemmanouil. Deep reinforcement learning for join order enumeration. *arXiv preprint arXiv:1803.00055*, 2018.

[96] V. Markl, G. M. Lohman, and V. Raman. LEO: An autonomic query optimizer for DB2. *IBM Systems Journal*, 42(1):98–106, 2003.

[97] S. Matusevych, A. Smola, and A. Ahmed. Hokusai - sketching streams in real time, 2012.

[98] A. Meliou, W. Gatterbauer, and D. Suciu. Reverse data management. *Proceedings of the VLDB Endowment*, 4(12), 2011.

[99] A. Meliou and D. Suciu. Tiresias: the database oracle for how-to queries. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 337–348. ACM, 2012.

[100] C. W. Mortensen. Fully dynamic orthogonal range reporting on ram. *SIAM Journal on Computing*, 35(6):1494–1525, 2006.

[101] F. Nagel. *Recycling intermediate results in pipelined query evaluation.* PhD thesis, Citeseer.

[102] F. Nagel, P. Boncz, and S. D. Viglas. Recycling in pipelined query evaluation. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 338–349, Apr. 2013.

[103] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms. *Journal of the ACM (JACM)*, 65(3):16, 2018.

[104] T. B. of Transportation Statistics. Border crossing dataset. `https://www.kaggle.com/akhilv11/border-crossing-entry-data`, 2019.

[105] F. Olken and D. Rotem. Simple random sampling from relational databases. 1986.

[106] M. Olma, O. Papapetrou, R. Appuswamy, and A. Ailamaki. Taster: self-tuning, elastic and online approximate query processing. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 482–493. IEEE, 2019.

[107] C. Olston, B. T. Loo, and J. Widom. Adaptive precision setting for cached approximate values. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, SIGMOD '01, pages 355–366, New York, NY, USA, 2001. ACM.

[108] O. Onyshchak. Stock market dataset, 2020.

[109] S. Papadomanolakis, D. Dash, and A. Ailamaki. Efficient use of the query optimizer for automated physical design. In *Proceedings of the 33rd international conference on Very large data bases*, pages 1093–1104. VLDB Endowment, 2007.

[110] Y. Park, B. Mozafari, J. Sorenson, and J. Wang. Verdictdb: Universalizing approximate query processing. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1461–1476, 2018.

[111] Y. Park, A. S. Tajik, M. Cafarella, and B. Mozafari. Database learning: Toward a database that becomes smarter every time. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 587–602, 2017.

[112] A. Pavlo, M. Butrovich, A. Joshi, L. Ma, P. Menon, D. Van Aken, L. Lee, and R. Salakhutdinov. External vs. internal: An essay on machine learning agents for autonomous database management systems. *Data Engineering*, page 31, 2019.

[113] J. Peng, D. Zhang, J. Wang, and J. Pei. Aqp++ connecting approximate query processing with aggregate precomputation for interactive analytics. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1477–1492, 2018.

[114] L. L. Perez and C. M. Jermaine. History-aware query optimization with materialized intermediate views. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 520–531. IEEE, 2014.

[115] W. H. Peter Bodik et al. Intel wireless dataset. `http://db.csail.mit.edu/labdata/labdata.html`, 2004.

[116] W. H. Peter Bodik et al. Intel wireless dataset. `http://db.csail.mit.edu/labdata/labdata.html`, 2004.

[117] T. Phan and W.-S. Li. Dynamic materialization of query views for data warehouse workloads. In *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*, pages 436–445. IEEE, 2008.

[118] R. Poepsel-Lemaitre, M. Kiefer, J. von Hein, J.-A. Quiané-Ruiz, and V. Markl. In the land of data streams where synopses are missing, one framework to bring them all. 2021.

[119] V. Poosala, P. J. Haas, Y. E. Ioannidis, and E. J. Shekita. Improved histograms for selectivity estimation of range predicates. In *ACM Sigmod Record*, volume 25, pages 294–305. ACM, 1996.

[120] N. Potti and J. M. Patel. Daq: a new paradigm for approximate query processing. *Proceedings of the VLDB Endowment*, 8(9):898–909, 2015.

[121] S. Rizzi and E. Gallinucci. Cubeload: a parametric generator of realistic OLAP workloads. In *International Conference on Advanced Information Systems Engineering*, pages 610–624. Springer, 2014.

[122] K. Rong, Y. Lu, P. Bailis, S. Kandula, and P. Levis. Approximate partition selection for big-data workloads using summary statistics. *arXiv preprint arXiv:2008.10569*, 2020.

[123] P. Scheuermann, J. Shim, and R. Vingralek. Watchman: A data warehouse intelligent cache manager. 1996.

[124] R. STEVEN, F. SARAH, and G. RONALD. Ipums usa: Version 9.0 [dataset], 2019.

[125] L. Sun, M. J. Franklin, S. Krishnan, and R. S. Xin. Fine-grained partitioning for aggressive data skipping. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1115–1126. ACM, 2014.

[126] B. Sundarmurthy, P. Koutris, W. Lang, J. Naughton, and V. Tannen. m-tables: Representing missing data. In *20th International Conference on Database Theory (ICDT 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[127] B. Sundarmurthy, P. Koutris, and J. Naughton. Exploiting data partitioning to provide approximate results. In *Proceedings of the 5th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond*, BeyondMR'18, pages 5:1–5:5, New York, NY, USA, 2018. ACM.

[128] R. S. Sutton, A. G. Barto, and R. J. Williams. Reinforcement learning is direct adaptive optimal control. *IEEE Control Systems*, 12(2):19–22, 1992.

[129] N. Taxi and L. Commission. New york city taxi trip records dataset. `https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page`, 2019.

[130] J. S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57, 1985.

[131] B. Walenz, S. Sintos, S. Roy, and J. Yang. Learning to sample: Counting with complex queries. *Proceedings of the VLDB Endowment*, 13(3):390–402, 2019.

[132] J. Wang, S. Krishnan, M. J. Franklin, K. Goldberg, T. Kraska, and T. Milo. A sample-and-clean framework for fast and accurate query processing on dirty data. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 469–480. ACM, 2014.

[133] L. Wang, R. Christensen, F. Li, and K. Yi. Spatial online sampling and aggregation. *Proceedings of the VLDB Endowment*, 9(3):84–95, 2015.

[134] Z. Yang, E. Liang, A. Kamsetty, C. Wu, Y. Duan, X. Chen, P. Abbeel, J. M. Hellerstein, S. Krishnan, and I. Stoica. Deep unsupervised cardinality estimation. *arXiv preprint arXiv:1905.04278*, 2019.

[135] E. Zgraggen, A. Galakatos, A. Crotty, J.-D. Fekete, and T. Kraska. How progressive visualizations affect exploratory analysis. *IEEE transactions on visualization and computer graphics*, 23(8):1977–1987, 2016.

[136] J. Zhang, Y. Liu, K. Zhou, G. Li, Z. Xiao, B. Cheng, J. Xing, Y. Wang, T. Cheng, L. Liu, et al. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data*, pages 415–432. ACM, 2019.

[137] Q. Zhang, N. Koudas, D. Srivastava, and T. Yu. Aggregate query answering on anonymized tables. In *2007 IEEE 23rd international conference on data engineering*, pages 116–125. IEEE, 2007.

[138] Z. Zhao, R. Christensen, F. Li, X. Hu, and K. Yi. Random sampling over joins revisited. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1525–1539, 2018.

[139] D. C. Zilio, C. Zuzarte, S. Lightstone, W. Ma, G. M. Lohman, R. J. Cochrane, H. Pirahesh, L. Colby, J. Gryz, E. Alton, et al. Recommending materialized views and indexes with the ibm db2 design advisor. In *Autonomic Computing, 2004. Proceedings. International Conference on*, pages 180–187. IEEE, 2004.