NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

PROGRAM OF POSTGRADUATE STUDIES
”COMPUTER SYSTEMS: SOFTWARE AND HARDWARE”

Master’s Thesis

A Sample Index for Approximate Query Processing

Victor A. Giannakouris - Salalidis

ATHENS

JUNE 2021

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
”ΥΠΟΛΟΓΙΣΤΙΚΑ ΣΥΣΤΗΜΑΤΑ: ΛΟΓΙΣΜΙΚΟ ΚΑΙ ΥΛΙΚΟ”

Διπλωματική Εργασία

Ευρετήριο Δειγμάτων για την Επεξεργασία Προσεγγιστικών Ερωτημάτων

Βίκτωρ Α. Γιαννακούρης - Σαλαλίδης

ΑΘΗΝΑ

Ιούνιος 2021

M.Sc. THESIS

A Sample Index for Approximate Query Processing

Victor A. Giannakouris - Salalidis
Student ID: CS3180002

Examination Date: June 4, 2021

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Ευρετήριο Δειγμάτων για την Επεξεργασία Προσεγγιστικών Ερωτημάτων

Βίκτωρ Α. Γιαννακούρης - Σαλαλίδης
Α.Μ.:CS3180002

Ημερομηνία Εξέτασης: 4 Ιουνίου 2021

ABSTRACT

We introduce sample index, a novel index structure that aims to enhance the sampling performance in a database system. Our idea is based on the observation that the overheads resulting from the expensive sampling steps during query execution can be mitigated by leveraging an index that is created offline. Our index is able to serve *fresh* samples even when queries are issued during continuous inserts in the database, i.e., during an ETL process. Our experimental evaluation proves that our sample index implementation in MonetDB can achieve performance improvements ranging from *2x* to *4.5x* better query execution times.

SUBJECT AREA: Database Systems

KEYWORDS: Database System, Sampling, Index

ΠΕΡΙΛΗΨΗ

Σε αυτή την εργασία παρουσιάζεται το Ευρετήριο Δειγμάτων, μία δομή δεδομένων η οποία αποσκοπεί στην βελτίωση της απόδοσης της διαδικασίας της δειγματοληψίας σε ένα σύστημα διαχείρισης βάσης δεδομένων. Η κεντρική ιδέα βασίζεται στην παρατήρηση πως ένα σημαντικό ποσοστό του κόστους εκτέλεσης ενός προσεγγιστικού ερωτήματος, λόγω ορισμένων πολύπλοκων διαδικασιών που σχετίζονται με την δειγματοληψία, μπορεί να μειωθεί με την χρήση ενός ευρετηρίου που έχει δημιουργηθεί ενόσω το σύστημα βρίσκεται σε αδράνεια. Το Ευρετήριο Δειγμάτων που προτείνεται έχει την δυνατότητα να επιστρέφει σε αποδοτικό χρόνο τα πιο πρόσφατα δείγματα ενός πίνακα, ακόμη και κατά την διάρκεια συνεχών ενημερώσεων, όπως για παράδειγμα κατά την διάρκεια μιας ETL διαδικασίας. Στην πειραματική αξιολόγηση αποδεικνύεται πως με την δομή που προτείνουμε μπορεί να επιτευχθεί βελτίωση της απόδοσης η οποία κυμαίνεται από 2.4 έως και 4.5 ταχύτερους χρόνους εκτέλεσης.greek

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Συστήματα Βάσεων Δεδομένων

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Βάση Δεδομένων, Δειγματοληψία, Ευρετήριο

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Modern data analytics and scientific discovery rely on systems that can efficiently process large amounts of data so the user can extract useful information and draw conclusions. In most scenarios, the users, usually data scientists or business analysts, need to perform several exploratory steps to get a better understanding of the data before they proceed to the long-running analysis. Approximate Query Processing (AQP) has been proven a great tool for providing accurate answers in "human-time" when exploratory queries need to be evaluated.

The predominate approach to AQP is to construct samples [4, 22, 21] where queries can be tested for validity and usefulness before fired against large amounts of data. The majority of the systems that support queries against samples implement one of the two approaches: either pre-compute and store materialized fixed-size samples [20, 7, 12, 17], or construct the sample during the time-critical path of query execution by invoking a random number generator [1,2]. There exists a large amount of research in sampling techniques for database systems, however there are also some important shortcomings on these common sample construction techniques that we believe have not yet been clearly identified and addressed in the literature.

The following shortcomings are identified:

- Constructing and maintaining fixed-size samples is not efficient when many updates are performed. Even in the case of bulk updates, the existing samples have to be disregarded and new ones to be recomputed in order to provide an accurate representation of the latest data. This introduces huge update and maintenance costs, rendering these solutions inefficient for applications with continuous insertions, such as scientific discovery and business analysis. Moreover fixed-size samples do not provide true random samples for sample sizes that are not equal to the predefined fixed sizes.
- A plethora of commercial database engines, support arbitrary sample construction during query evaluation time. In most cases, this approach significantly reduces query processing performance due to the sampling operator overheads. More specifically, a sample operator needs to invoke multiple times a *rand()* function in order to generate a random value and identify which rows of a base table should be part of the sample. Clearly, such function calls significantly hinder the performance of query evaluation. In addition, every random number that is produced must be checked if it has been already drawn because in a correct data sample we cannot have the same row picked twice. Hence, duplicate elimination is also a sizeable overhead that reduces performance and can too significantly reduce query evaluation times when large enough samples are requested. We analyze further the overheads that result from this approach in the next subsection.

[1]`https://www.monetdb.org/Documentation/ServerAdministration/Sampling`

[2]`https://docs.mongodb.com/manual/reference/operator/aggregation/sample/`

1.1 Motivation

We present our findings of a preliminary analysis that we performed on the sampling operator of MonetDB. We focus on the overheads that hinder query performance when samples are constructed during query execution. The sampling approach adopted by most commercial database systems, including MonetDB, require the following steps.

1. Generate k unique random values and map each value to a row id by applying a modulo operator.
2. Materialize the corresponding k tuples according to the row IDs acquired from the first step.

We provide a pseudo-code version of the this sampling algorithm (Algorithm 1). Taking into account that this algorithm needs to be executed during query evaluation each time a query requires a sample, we observe that this approach has the following drawbacks. First, invoking k times a random value generator like `rand()` (line 5) can be expensive for large sample sizes, as each invocation needs to execute a number of calculations in order to produce a random row id. Furthermore, in order to construct a correct sample we need to ensure that it contains only unique tuples. As a result, in each iteration requires a lookup to check if the produced random value has been already drawn by some previous iteration (line 7). Usually, this is done by leveraging a data structure, like a binary search tree, that keeps track of the random values that have been drawn by past iterations. Thus, for each new random value, an extra $log(n)$ lookup is needed.

Algorithm 1 A naive sampling algorithm

```
1 int[] sample = new int[k];
2 for (i=0; i < k; i++)
3     do
4         // Generate a random row id
5         int rid = random(0, N-1);
6         // Check if that row id exists (log(k))
7         bool exists = search(rid, sample)
8     while (exists)
9     insert(sample, rid);
```

In order to analyze further the aforementioned overheads, we modified the code of the sampling operator of MonetDB in order to keep track of the execution metrics and analyze the execution time distribution. We are particularly interested in exploring the total time consumed by the sampling operator of MonetDB, the random number generator [3], and the time required for duplicate checks. Next, we run the following two experiments, in which we used the query ”`select avg(l_quantity) from (select * from lineitem sample k)`”, where `k` denotes the sample size.

[3]https://github.com/MonetDB/MonetDB/blob/master/gdk/xoshiro256starstar.h

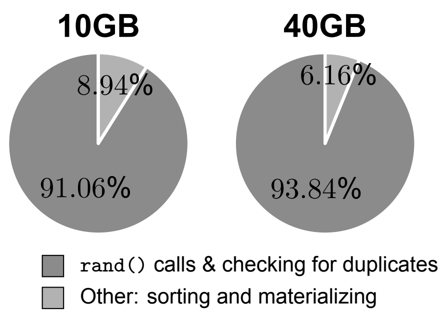


Figure 1.1: Sample operator: where time goes (increasing table size, 10% sample)

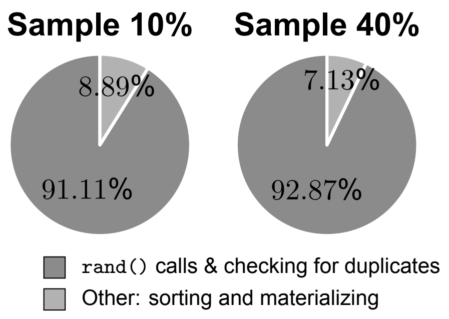


Figure 1.2: Sample operator: where time goes (increasing sample size, 10GB scale)

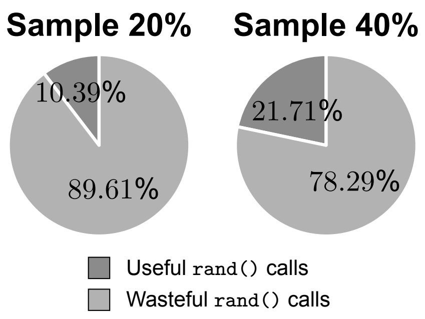


Figure 1.3: Sample operator: `rand()` call distribution (constant table size)

1.1.1 Experiment 1

We generated the *lineitem* table of the TPC-H benchmark using the 10, 20, 40 and 80 GB scale factors. We executed the aforementioned query over a 10% sample of the original table, for each particular scale. In other words, we increase the data size while we keep the sample size ratio constant (10%). Table 1.1 depicts the query time distribution. Figure 1.1 visualizes the time distribution of the sampling operator for the increasing table size scenario. It can be seen that in all cases more than 90% of the sampling time is consumed by the the random number generator and duplicate checks. Moreover, we can see in all cases there is an extra 5% of `rand()` invocations due to duplicates.

Table 1.1: Query Time Distribution - Constant k (%), Increasing Table Size

TPC-H Scale	Size (rows)	Sample Size (k)	Total (s)	Sampling (s)	`rand` (s) (% of sampling)	rand() calls	Unsuccessfull rand() calls
10	59986052	5998606 (10%)	9.54	7.995 (84%)	7.28 (91.06%)	6320956	322350 (5.1%)
20	119994608	11999461 (10%)	23.48	19.95 (85%)	18.31 (91.78%)	12643505	644044 (5.1%)
40	240012290	24001229 (10%)	55.25	44.43 (81%)	41.26 (92.87%)	25288844	1287615 (5.1%)
80	480025129	48002513 (10%)	162.97	100.22 (62%)	93.45 (93.25%)	50574196	2571683 (5.09%)

1.1.2 Experiment 2

In the second experiment, we used only the 10GB TPC-H scale of the *lineitem* table. We run the same query as in the previous experiment but now we increase the sample size ratio. We run the query using the sample sizes (k) of 10%, 20%, 40% and 50%. Table 1.2 depicts the query time distribution and figure 1.3 visualizes the distribution of the *useful* and *wasteful* `rand()` invocations. The results are similar to the first experiment. Again, we can see that more of 90% of the sampling time is spent by the `rand()` calls and the duplicate checks in all experiments. Furthermore, this experiment showcases that the number of false `rand()` invocations increases monotonically to the sample size. This happens because by increasing the sample size, the probability of generating the same a random number more than once is increased as well. Specifically, we observe that the ratio of the unsuccessful `rand()` calls is near the half of the sample size ratio in all cases.

Table 1.2: Query Time Distribution - Increasing k (%), Constant Table Size

TPC-H Scale	Size (rows)	Sample Size (k)	Total (s)	Sampling (s)	`rand` (s) % of sampling	rand() calls	Unsuccessfull rand() calls
10	59986052	5998605 (10%)	8.72	7.53 (86.36%)	6.86 (91.11%)	6320528	321923 (5.1%)
10	59986052	11997210 (20%)	21.6	20.11 (93.11%)	18.52 (92.1%)	13387174	1389964 (10.39%)
10	59986052	17995815 (30%)	37.12	35.39 (95.34%)	32.98 (93.2%)	21394070	3398255 (15.89%)
10	59986052	23994420 (40%)	57	54.98 (96.46%)	51.59 (93.84%)	30645731	6651311 (21.71%)
10	59986052	29993026 (50%)	75.92	73.65 (97.02%)	69.98 (95.02%)	41574955	11581929 (27.86%)

From this experiment is easy to come to the conclusion that if we were to remove the `rand()` and duplicate checks from the time-critical path of query execution and move it to an index that is created during loading and updates, then the sample queries will run much faster.

1.1.3 The Sample Index

In this thesis we introduce a novel index that provides a *variable-sized* sample without the overhead of computing any random values during the *time-critical* path of query evaluation. Our index is optimized for read-intensive databases such as column stores. In addition, the sample index can be updated during continuous insertions or bulk updates, a common scenario for read-optimized engines that support scientific and business analysis. Our index is capable of providing the user with “*fresh*” samples that include the latest data whilst are being inserted. This feature is very important for our target applications since data maybe injected for many hours (e.g., long-running experiments or overnight ETL processes) while the user (data scientist or business analyst) needs to start querying as soon as possible.

Our sample index is an offline index in the sense that it computes, orders, and stores row identifiers according to a uniform sample, supports online analysis since it will continuously update its uniform sample to include the latest inserted data. Our experimental evaluation over the open-source column store MonetDB proves that by using our sample index we can achieve up to *5x speedups* in running approximate queries.

1.1.4 Contributions

The contributions of this thesis are summarized as follows:

1. We introduce the offline sample index, an novel index scheme that enhances the performance of the sampling operator in a database system, by replacing the excessive overheads of random number generation and duplicate elimination in query time with a sequential index scan.

2. We present an architecture for continuous insertion support, that leverages a priority queue that handles update-intensive workloads, namely a min-heap priority queue. During insertions, users are still able to run their queries without facing any downtime, while they are also served with *fresh* samples that contain the most recently inserted data.

3. We present a detailed experimental evaluation using four datasets, and we prove that our sample index can achieve up to 5x speed-ups for sample-based approximate queries.

4. We present some interesting findings on how data are laid-out in a min-heap, when records are indexed according to random numbers drew from a uniform distribution. We identify a common layout pattern which allows us to reduce the cost of obtaining a k-size sample from a min-heap from $O(k \cdot log(n))$ to k, by simply fetching the first k elements from the heap array. We believe that this insight could also benefit other applications that depend on priority queues.

1.1.5 Outline

The rest of this thesis is organized as follows.

In chapter 2 we present some preliminaries required for the proper understanding of the rest of this thesis. We discuss topics including indexes in database systems, data structures like priority queues, as well as some statistics background on sampling and order statistics. If the reader is familiar with these concepts, this chapter can be skipped.

Chapter 3 presents the idea of sample index. We present the main architecture of sample index and how it is constructed offline over a database table. We also present an architecture that allows us to efficiently handle continuous insertions by employing an priority queue. This idea is inspired by an ETL use-case, where the users need to issue analytical queries and get updated results, while the database is being updated.

In chapter 4 we present the results of our experimental evaluation. We use four datasets, including TPC-H, Census, Abalone and Wine, obtained from UCI Machine Learning repository. We compare the performance when a sample index is being used and when it is not. We showcase that when a sample index is leveraged in the sampling process, queries can be run in up to 5x better execution times.

Chapter 5 presents the related work, where we classify previously presented ideas into three categories. These categories are *i)* sampling relational databases, *ii)* sampling-based approximate query processing and *iii)* online aggregation. We discuss what are the shortcomings when common techniques like fixed-size sample pre-computation is used, and how our proposed sample index can overcome these drawbacks.

Finally, we conclude this thesis by summarizing the highlights and describing the future work.

2. BACKGROUND

In this chapter we present the background needed for the understanding of the rest of this thesis. We will be discussing topics including the MonetDB column store, priority queues, indexes, order statistics and sampling.

2.1 The MonetDB Column Store

The idea of sample index presented in this thesis is implemented and evaluated over MonetDB. MonetDB is an open-source database management system developed at the Centrum Wiskunde & Informatica (CWI). It employs a columnar storage layout, which means that the data are physically organized on disk in a column-major fashion.

In MonetDB, each table is represented by a set of binary association tables, called *BAT*s. A BAT is an abstraction that represents a column as a mapping between an `oid` and an the corresponding attribute value. An `oid` represents the id of a specific record in the table, and it is analogous to the row id in a row-based database system. As a result, a table is decomposed into a set of BATs and all attribute values of a specific tuple are positioned at the same index of their corresponding BAT. BATs can be also thought of as parallel arrays representing a table, where each array holds the data of a specific column.

In a typical database system, the optimizer produces the optimal SQL query plan before execution, through some sophisticated optimization algorithm. The execution plan can be usually explored by the `explain` clause. MonetDB is quite different, in that it first converts an SQL query into an intermediate representation, called *MonetDB Assembly Language (MAL)* plan. Further execution decisions including access methods, are being taken at run-time.

The MAL intermediate representation of an SQL query is a sequence of operations between BATs. The required tables are first loaded from disk into the corresponding set of BATs, and each intermediate result is kept in a new BAT as well. For example, consider the following query.

```
select * from test sample 5
```

By invoking the `explain` clause we can easily get the MAL execution plan. Partially, the plan of this query should look like the following sequence of MAL expressions:

```
X_4:int := sql.mvc();
C_5:bat[:oid] := sql.tid(X_4:int, "sys":str, "test":str);
X_17:bat[:int] := algebra.projection(C_5:bat[:oid], X_8:bat[:int]);
C_20:bat[:oid] := sample.subuniform(X_17:bat[:int], 5:lng);
X_21:bat[:int] := algebra.projection(C_20:bat[:oid], X_17:bat[:int]);
```

We can easily observe that plan is a chain between BAT dependencies, that is, `C_5` depends on `X_4`, `C_17` depends on `X_C5` and so on. The `sample` clause is translated into the `sample.subuniform` MAL function. Finally, the set of `oid`s and the attributes are projected with `algebra.projection` function and stored into `X_21`.

2.2 Indexes

In this section we present some background on indexes in database systems and how operations like `scan`s and `selection`s can benefit from them.

An index is a data structure that is used to speedup data loading in query execution. The speed-up is at the cost of extra writing and updating overheads, as the index needs to be maintained each time the column that it refers to is updated. In general, an index speeds up either lookup, or range queries by pointing to specific positions of the columns and thus, when a scan needs to be executed lots of irrelevant data to the scan can be pruned. As a result, disk seek time and bandwidth is reduced significantly when one or more indices are involved in query execution.

2.2.1 The Basics

An index on a column c is a *non-clustered* index, when the data is physically organized with arbitrary order and not according to c. When a *clustered* index over one column is created, then the data pages are physically laid out ordered by the values of that column.

2.2.2 System Specific Indexes

In addition to the state-of-the-art index categories, there are also other forms of indexes provided by each database management system. In the next subsection we discuss about the *order index*, a particular index type provided by MonetDB. The sample index presented in this thesis, is also an index structure that speeds-up the sampling process in a database system.

2.2.2.1 The Order Index

An order index[1] is an index structure provided by MonetDB. The idea of an order index is similar to the concept of the order statistic that we discuss in a latter section. Given a table T that is *unsorted* on column A, an order index on A represents the position of each `rid` of T if T were sorted on A. In other words, an order index on a column represents the *relative order* of the table with respect to that column.

[1] https://www.monetdb.org/Documentation/Manuals/SQLreference/Indices

2.2.2.2 An Order Index Example

Let T a table with the column A with the following values: $A = \{5, 1, 3, 4\}$, and the set $T_{rid} = \{0, 1, 2, 3\}$ the `rid`s of table T. An order index on column A would look like $A_{oidx} = \{1, 2, 3, 0\}$. It should be clear that each element in A_{oidx} represents the relative position of each `rid` in T, that is, where each `rid` in T would be positioned if T were sorted according to A.

2.3 Data Structure Preliminaries

In this section we briefly discuss some preliminaries on specific data structures that we employ in the implementation of the sample index. We will focus on the priority queues, and especially the min heap.

2.3.1 Priority Queues

A priority queue is an abstract data structure similar to a regular queue or stack. The main difference is the following. In a priority queue, each record of the data is associated with a value called a *priority*. Taking into account the priority value, each element will be extracted from the queue according to its priority, and not solely its time of insertion. The priority definition depends on the use-case and the type of the priority queue.

2.3.1.1 Binary Heaps

A binary heap is a tree-based data structure similar to a binary search tree. In contrast to a binary-search tree, a binary heap is a complete binary tree, as when a new records needs to be indexed will be alternately inserted left and right. That is, the first element that will arrive will be the parent. The next one, will be inserted on the left, the next on the right, the next again on the left and so on. As a result, each level of the binary heap (probably except of the last one) will be always fully filled. The property of the binary heap is that it always holds either the minimum or the maximum level at the root level. This is guaranteed by a process called *heapify* that is invoked each time a new element is inserted. If the root node holds the maximum key, then the binary heap is called a *max heap* and otherwise a *min heap*.

A binary heap is usually backed up by an array. In general, an array-based binary tree can be traversed as follows. For a given node `i`, the corresponding left child is positioned at `2*i` and the right node by `2*1 + 1`. Figure 2.1 depicts a min heap in three concrete instances. On the left the key `3` is inserted into the empty heap and goes on the position `0` of the array. Next, `3` and `8` are inserted. Now `3` is the minimum element, so it is bubbled-up on the top of the tree through the *heapify* process. Finally, the key `5` is inserted. We can

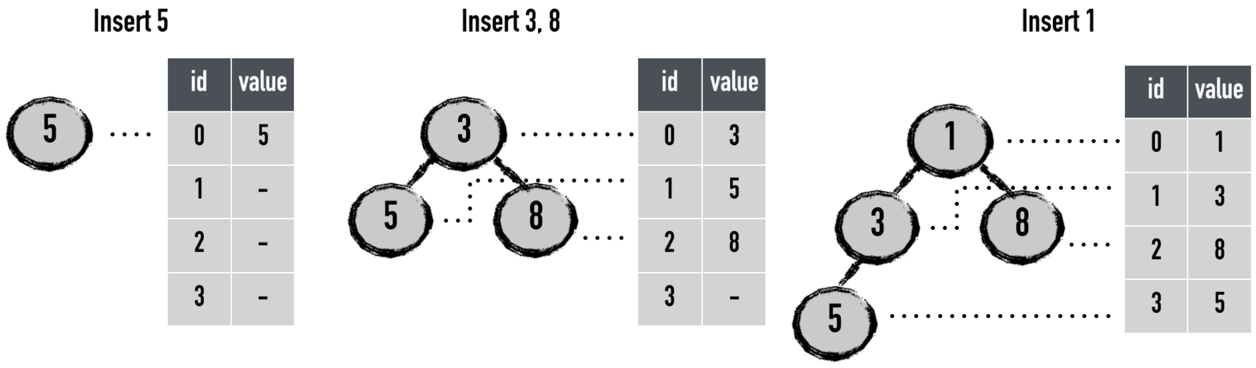


Figure 2.1: Min Heap Insertion

also see in the figure the mappings between the nodes in the tree and their positions the underlying array.

2.3.1.2 Accessing Array-Based Binary Heaps

Given a node `i`, its sibling nodes can be accessed by the following indexes, depicted in table 2.1.

Table 2.1: Binary Heap Traversal

Node	Index
Parent	`i/2`
Left Child	`2*i`
Right Child	`2*i +1`

2.3.1.3 Cost of Operations in Binary Heaps

The costs of operations are similar to any tree structure. Insertion of an element is logarithmic, as each time we insert a new element in the tree we need to make sure that either the minimum or the maximum is at the root level by the heapify process. As a result, the insertion cost in a min heap is $O(log(n))$, where n the total elements in the heap. Peeking the element with the highest priority needs a single operation, that is, $O(1)$. On the other hand, pop needs $O(\log n)$ steps, as each time an element is popped out of the heap, the heapify process is called to put the rightmost element at root-level, and then sink it down. Table 2.2 depicts the cost of each operation over a binary heap for the average and the worst cases.

Table 2.2: Binary Heap Operations Complexities

Operation	Average Case	Worst Case
Insert	$O(1)$	$O(\log n)$
Delete	$O(\log n)$	$O(\log n)$
Peek	$O(1)$	$O(1)$
Pop	$O(\log n)$	$O(\log n)$

2.4 Sampling and Order Statistics

Now, we will give a brief explanation of the statistical concepts that are related to the idea of the sample index.

2.4.1 Order Statistic

In a statistical sample, the k^{th} order statistic of an individual represents its order (rank) amongh the whole sample. For example, assume the sample $A = \{5, 1, 3, 4\}$. Then, the order statistic of the first value of the set is 4, as if we put A in increasing order, then 5 would be the fourth element. The order statistic of 1 is 1, as its rank is 1. The order statistic of 3 is 3 and so on. In fact, the set of the order statistics of a sample represents the *relative position* of each element if that sample were sorted in ascending order. We will use the notion of order statistic in order to formulate the problem that a sampling index addresses in the next sections.

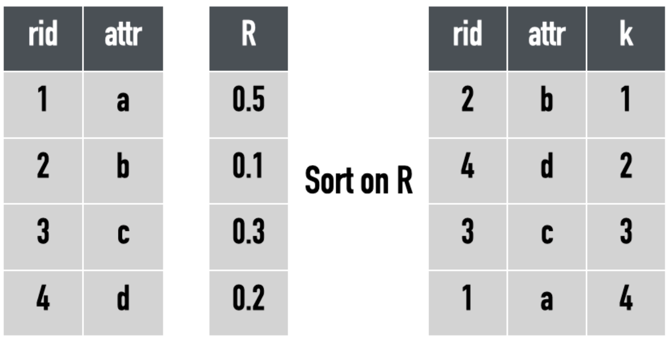
2.4.2 Random Sampling

We now give the formal definition of a random sample. Given a set A of size N, a random sample of A is a subset of k individuals. Each individual i is chosen with equal probability given by $P(i) = \frac{1}{N}$. When each element of A is chosen only once, then it is being said that the random sample is created *without replacement*.

2.4.3 A Random Sort Sampling Algorithm

2.4.3.1 Algorithm Description

We now describe a random sampling algorithm by A.B. Sunter [23], called *random sort*. The construction phase of the sample index described in the next chapter of this thesis is based on this algorithm. A random sample can be obtained with the following method. Given a set T with $|T| = N$ elements, we need to perform a random sort on T as follows. For each element of T, draw a random value from the rectangular distribution $R(0, 1)$. Next, sort T according to the random values. Let us call the resulting randomly sorted set



rid	attr
1	a
2	b
3	c
4	d

R
0.5
0.1
0.3
0.2

Sort on R

rid	attr	k
2	b	1
4	d	2
3	c	3
1	a	4

Figure 2.2: Random Sort Sampling Example

T_{random}. The first k elements (or any contiguous set of size k) of T_{random} will form a k-sized true random sample of table T. Next, we give a detailed example of the algorithm.

2.4.3.2 A Random Sampling Example

Let $T = \{a, b, c, d\}$ be a set with four elements, the corresponding random values $V = \{0.5, 0.1, 0.3, 0.2\}$ drew from $R(0, 1)$ and their order statistics $X = \{4, 1, 3, 2\}$. Recall that X represents the relative order of V. If we sort T according to X, then the output will be $T_{random} = \{b, d, c, a\}$. Then, for each $k \in [1, 4]$ a subset of T_{random} will be a k-size uniform sample of T. From a more technical perspective, we could think of X as an *ordered index* built on top of V.

Figure 2.2 depicts an example of the random sort algorithm over an example MonetDB relation. The leftmost table represents the relation consisting of the `oid` and `attr` BATs, the middle table represents the BAT with the random values while the rightmost one is the final randomly sorted BAT. The k column is a helper column that represents the sample size. Now it should be clear that the first k rows of the rightmost table form a k-size random sample.

2.4.3.3 Cost Analysis

The cost analysis of the algorithm described above is relatively simple. The algorithm can be described in two discrete steps. First, given a table T of size N we need to generate a set V that contains N random values, drew by the rectangular distribution $R(0, 1)$. The asymptotic complexity is obviously $O(N)$, while the total cost can be described by the following formula:

$$C_r = N \cdot C_{rand} \tag{2.1}$$

where C_{rand} the cost of generating a single random number from $R(0, 1)$. Next, we need to sort the elements in table T according to V. On average, using any well-known sorting algorithm this would take $O(n \log n)$ steps. Finally, k more steps are needed to get the first

k elements of the sample. As a result, the total cost for obtaining a k-size sample from T will be

$$C = N \cdot C_{rand} + N \cdot \log N + k \tag{2.2}$$

where $N = |T|$.

2.5 Summary

In this section we briefly presented the background of some concepts used for the development of the sample index scheme. We discussed topics including the MonetDB column store, sampling, order statistics and indexes. Our cost analysis, as well as the experiment that we presented in section 1.1 showcases that sampling can be a slow process when the table or sample size k (or both) are high. Clearly, the construction phase which is order of N can be frustrating in real-time applications, especially when the sampling algorithm needs to be invoked multiple times.

3. THE SAMPLE INDEX

3.1 Introduction

In this section we present a novel sample index structure that aims at speeding up the performance of the sampling operator by pre-computing and pre-ordering all *random row identifiers (row ids)* to avoid the significant overhead of these operations during query time. In addition, with the help of a priority queue we can update our sample index whilst new data are been inserted. Consequently, the user will continuously have access to *fresh uniform samples* that contain – with the same probability – a mix of old and newly inserted rows.

To construct the sample index we use the *random sort algorithm* that was described by Sunter in 1977 [23]. The algorithm simply assigns a random number between $(0, 1)$ as a key to each data point, and then by sorting using that key and selecting any contiguous set of k points we can obtain a truly uniform sample of size k.

The main idea behind our sample index is to use the random sort algorithm to create and store pairs of row ids and random values to be used to construct a sample of any size. For that we keep on persistent storage a mapping between row ids and random values sorted on the random value. This mapping is simply a two column structure. In order to obtain a sample of size k we simply retrieve any contiguous set of k pairs of this mapping. However, the pairs are sorted on the random value, so an extra sorting on row ids has to be performed if the database engine is expecting always sorted row ids, e.g., for column alignment and late materialization.

3.1.1 Problem Formulation

Let $Q = \{q_0, q_1, ..., q_N\}$ be a set of N queries that run over samples of the table T, and $S = \{150, 200, 300, ..., s_N\}$ the corresponding sample sizes. That is, query q_0 runs on a sample of T of size $k = 150$, while q_1 a sample of size $k = 200$, q_2 a sample of size $k = 300$ and so on. The sample index addresses the following shortcomings of the state-of-the-art sampling approach adopted by most commercial database engines that construct samples during query execution.

3.1.1.1 Random Number Generation and Deduplication Overheads

It should be clear by section 1 that sample query performance suffer by constructing samples during execution. The main reasons are the multiple invocations of the `rand()` function and the *lookups* required to ensure that the sample contains only unique row IDs. For example, to construct the sample required for query q_0, the sampler will invoke `rand()` 300 times plus the number of duplicate values occurred. In section 1 we also presented some results that show that more that 90% of the sampling time is being spent on computing random values and checking for duplicates.

3.1.1.2 Overlapping Samples

As long as all queries run on the same table T, but over samples of different sizes, they could benefit by re-using a single, bigger sample, instead of re-running the sampling algorithm each time a sample is requested. For example, q_0 uses a smaller sample than q_1 and could reuse the first 150 rows of the larger, 200-row sample of q_1. However, the current approach needs to execute the sampling algorithm from scratch for each new sample request.

3.1.1.3 The Sample Index Abstraction

A sample index can speedup the sampling process by constructing offline a secondary structure that represents a *relative random order* of the table. The construction steps are two. First, each row ID of the table is assigned a random value, drew from the uniform distribution $[0, 1]$. Next, the row IDs are sorted according to this random value. For example, we assume a table A, its row IDs $\{1, 2, 3, 4, 5\}$ and the corresponding random values $R = \{0.2, 0.5, 0.3, 0.1, 0.4\}$. The random values are aligned with the row IDs, i.e., 0.2 is assigned to the row ID 1, 0.5 to the row ID 2 and so on. The relative order of the table according to R will be $S = \{4, 1, 3, 5, 2\}$, because 4 is paired with 0.1 so it goes first, 1 is paired with 0.2 and it will be placed after 4, and so on. The set of pairs $\{(0.1, 4), (0.2, 1), (0.3, 3), (0.4, 5), (0.5, 2)\}$ is the resulting sample index. We need to keep the random values associated with the rows in order to be able to update the sample index when new rows are inserted. In order to construct a k-sized sample we simply need to take any contiguous set of size k from the sample index. As a result, both of the aforementioned shortcomings of the naive sampling approach are addressed, as random number generation and deduplication overheads are replaced with a sequential scan on sample index. Both processes are performed only once as part of the random sort algorithm, during index construction. Thus, running any sampling algorithm during query execution is not required anymore.

Figure 3.1 depicts a sample index on the relation A. The leftmost table depicts the row identifiers (`rid`) and the attributes of table A, and the middle column (R) the associated random values. The right table shows the final sample index. On the right we can also see the bounds of queries Q_0, Q_1 and Q_2 and where they point to the sample index. Clearly, there is a discrete overlap on the sample sizes of these three queries, as $Q_0 \subset Q_1 \subset Q_2$. The benefit gained from the sample index is that none of these queries need to execute any complex processing in order to decide which rows will be included in the sample. The sample is obtained just by fetching any contiguous set of size k from the index.

Through this example, it should be clear that by leveraging an access method like thesample index, the sampling algorithm (random sort) needs to be run only once during the

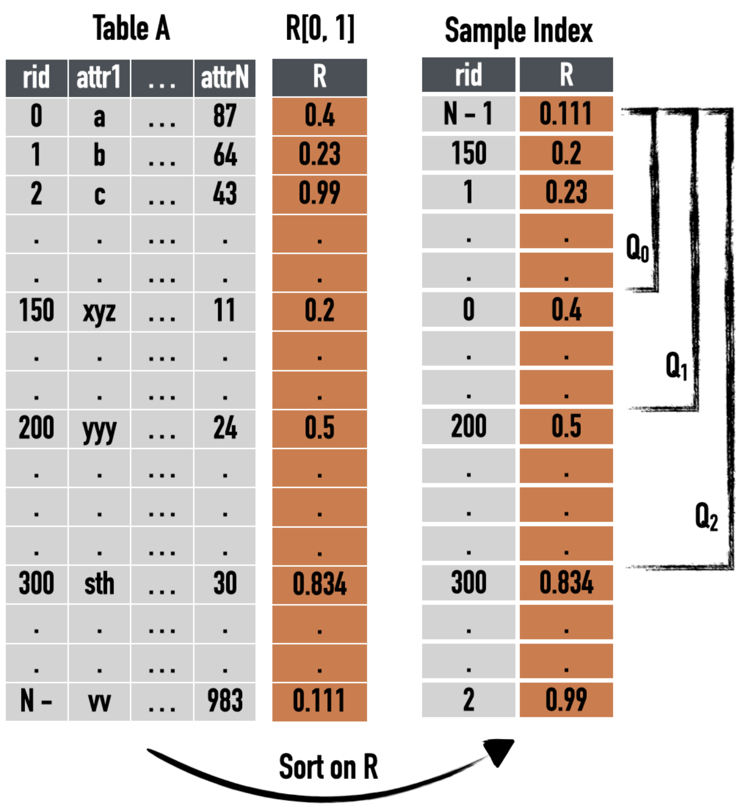


Figure 3.1: A sample index on Table A

index construction phase. Then, all requested samples of any size k can be served by the same sample index.

3.2 Offline Construction

The offline construction of an index is typical in any database system. Usually, in a database an index can be created offline by issuing a query like `create index idx on T (col)`. This query will create an index named *idx* on column *col* of table *T*. The index will be created on top of the existing rows of the table, while the index is updated when new rows are inserted into the table. In this section we explain how the sample index is constructed offline.

3.2.1 Creating sample indexes

A sample index can be created in two ways.

- First, similarly with any other index, a sample index can be created by executing a query of the form `create sample index on T`. In contrast with a traditional index, a sample index is created on the full table table, and not a specific column.

- A sample index can be also created when the first query that requires the sample is executed. That is, when a query of the form `select avg(c) from A sample 1000000` arrives, the index creation is triggered and executed as part of query processing. That is, the system generates first the sample index to obtain the requested sample. Next, that sample index can be persisted in order to serve future sample queries.

In both scenarios, the index is stored to persistent storage and can be reused to speed-up subsequent queries.

3.2.2 Initial Construction

First, given a table T with N rows, a column of size N and type `double` is initialized. Next, N random values from the rectangular distribution $R(0, 1)$ are generated and inserted to the newly generated column. What is left, is to create an *ordered index* on that column, and the resulting ordered index will be our sample index on table T. Recalling section 2, an ordered index is an index type in MonetDB that can be created over any column of the table and represents the relative order of the row identifiers according to that column. Algorithm 2 is a pseudo-code version of the construction phase of sample index.

Algorithm 2 Sample index construction

```
int[] sample_idx = new int[N];

for (rid=0; i < N; i++)
    do
        // Draw a random value
        r = random(0, 1);
        // Check if that random value exists
        exists = search(r, sample_idx)
    while (exists)
    insert_sorted(sample_idx, (r, rid));
```

3.2.3 Usage

When a sample query is issued on table T, we first need to check whether T has a sample index or not. If a sample index is available, then we need to load the index in memory. For a sample query of size k, we finally return a contiguous set of k values that starts in a random position of the index. This set contains the set of `rid`s of the tuples that will form our final sample.

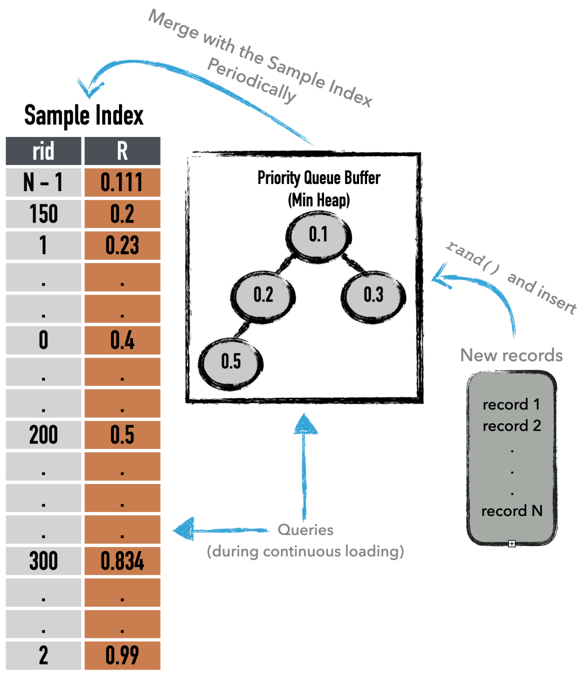


Figure 3.2: Continuous insertion and querying

3.3 Continuous Insertions

The sample index can support continuous insertions with the help of an intermediate structure, namely a priority queue. The scenario that inspires such a feature is the demand from the users to start their approximate query evaluations while data are being inserted either because of a long running ETL process or a scientific experiment that produces vast amounts of results. An important constrain is that each approximate query must be run against fresh samples that contain with the same probability the latest inserted rows. This idea is not included in our implementation yet, however, we present in detail the architecture and some high-level experiments conducted on the algorithms and data structures that we plan to use.

The approach to support such “online approximate queries” is to compute for each newly inserted row a unique random value but instead of updating the sample index – which would be costly at this point – insert the new pair in a priority queue, implemented here as a min-heap. The priority queue would always return the smallest random number and its row id of the newly inserted row. Now, in order to return a sample of size k we would have to consult both the existing sample index, and the priority queue. The goal is to merge the two structures and return the k row ids that have the smallest random values from either the sample index or the priority queue. Figure 3.2 demonstrates this algorithm when new records are inserted in the priority queue while queries are run against both structures. Finally, when the bulk insertion is finished, we can merge the existing sample index with the entire priority queue in order to create a new sample index, ready to be used until new updates are performed.

3.3.1 Leveraging a Priority-Queue to the Sample Index

As previously mentioned, to insert a new element to the sample index we need to generate a new random value, and based on that value, locate the correct position in the index and perform the insertion. The intermediate min-heap structure has an $O(1)$ insertion cost on the average-case, offering a significantly better insertion cost than the direct insertion to the sample index which requires to either sort the index with the new records from scratch, or, find the correct position with a binary search. As a result, a user can issue sample queries to the database while the table is being updated, without any delays due to index reorganization or slow insertions.

3.3.1.1 Serving fresh samples during continuous updates

Our goal is to make sample index able to offer fresh samples that contain the most recently inserted data during continuous insertions. If a sample is requested during insertions, the query must mix data both from the sample index and the min-heap, in order to provide a valid, up-to-date result. To achieve that, the sample will be constructed as follows. For each new record that needs to be appended to the sample, we look at the next record of the sample index, as well as the root record of the min-heap. We compare the random values of the two records, and we append the record with the minimum random value to our sample, following the property of the random sort sampling algorithm. We provide a detailed algorithm in pseudo-code of this process (Algorithm 3).

Algorithm 3 Online sample construction of a k-size sample

```
 1  int[] sample = new int[k];
 2  a = sample_idx.iterator.next();
 3  b = min_heap.iterator.next();
 4
 5  for (i=0; i < k; i++)
 6      if (b == null or a.key < b.key)
 7          insert(sample, a.value);
 8          a = sample_idx.iterator.next();
 9      else
10          insert(sample, b.value);
11          b = min_heap.iterator.next();
```

The cost of line 2 is $O(1)$, as it only requires to move to the next position of the sample index and fetch the element. On the other hand, the cost of line 3 is $O(log(n))$, as the `next()` call will fetch the next element of the min-heap and it will trigger the `heapify` process in order to move the next minimum element to the root. The cost of this algorithm is summarized in equation 3.1:

$$C_{smpl} = S + P\log(n) \tag{3.1}$$

In equation 3.1, S is the number of elements that will be pulled from the sample index, P the number of elements that will be pulled from the min-heap and n the min-heap size. Obviously, for a sample size k, $S + P = k$, while the values of S and P depend to the random value distribution. We can easily observe that the cost of fetching S elements is independent to the sample index size. However, the cost of fetching P elements from the min-heap is logarithmic to the size of the min-heap (n). As a result, in case that either P, n or both are large, our query performance will probably suffer. It can be seen in the equation that the most expensive part of the algorithm, is to obtain the P-size part of the sample from the min-heap. In the next section we explain our attempt to make cost of reading that P-size set from the min-heap independent of the heap size n.

3.3.2 Scanning the Min-Heap Sequentially

We performed an analysis on how data are laid out in the array when we create a random min-heap. As mentioned earlier in section 2.3.1, a min-heap uses an in-memory array. Given a parent node `i`, the left child is positioned at `2*i` and the right child at `2*i + 1`. The order of the incoming keys is unknown. In the best case scenario, if the keys come in a sorted order, then the array of the min-heap will be sorted too. In practice, this is unlikely to happen, especially when the keys are generated randomly. As a result, getting the set of the top-P minimum values from a min-heap requires $P\log(n)$ steps, that is, P key `pop()`s, while each `pop()` requires $log(n)$ additional steps due to the `heapify` process.

As previously mentioned, $Plog(n)$ steps is not the desirable performance for obtaining the values from the min-heap, especially for large P and n values. In the ideal case, we would like to obtain the P-sized set from the min-heap both in linear time, and with sequential memory access. Those two objectives led us to study how the elements are laid-out in the array of a random min-heap.

3.3.2.1 Getting the P-size set of the min-heap faster

We attempt to address the following question. To get the top-P minimum elements of the min-heap, we simply need to pop P elements from the heap, performing $Plog(n)$ steps. Instead, if we get the first P elements of the array as they are (randomly) laid out in memory through the heap creation, *how many of these elements are part of the actual top-P element set*? In other words, if we compare the top-P elements popped of the heap using `pop()`, and the top-P elements from the array of that heap, what will be the intersection of these two sets? An important point on that, is that when we construct a k-size sample we are interested in obtaining the rows that are associated to the top-k minimum random

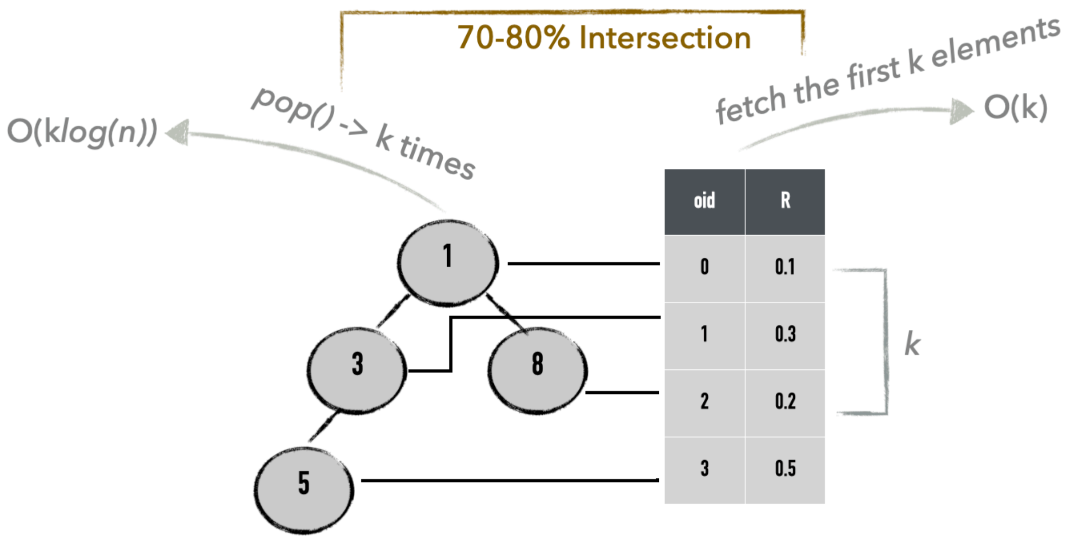


Figure 3.3: Intersection of Elements

values, while *we do not care if these random values are sorted or not*. The only requirement is to obtain the set of rows with the top-k minimum random values. Thus, we are only interested in the intersection between the two sets.

3.3.2.2 Experimental Intersection Comparison

After running a significant number of experiments, we observed that this intersection ratio is ranges from *70% to 80%*. Figure 3.4 depicts this result. We concluded to this result after multiple experiments, by generating a number of *uniformly random binary heaps* of various sizes. A uniformly random binary heap is either a min or max heap, generated by repeatedly inserting random values in it. This structure is also known as a random binary tree [1]. Each of our experiments consisted of the following steps. First, we initialize a min-heap and insert n random values. Next, we take k-size samples by obtaining the top-k elements of the min-heap, where $k = \frac{n}{10}$. Next, we get the top-k elements from the array. Finally, we compute the intersection ratio of these two sets. For each heap of size n, we perform the same experiment 100 times and we keep the average intersection ratio. The results lead to an interesting observation, showcasing that the ratio falls in the range $(0.7, 0.8)$ for the heap size ranging from 50 to 200000. Figure 3.3 depicts the results from these experiments. Each bar shows the average intersection ratio per heap size.

3.3.2.3 Performance Gains

Clearly, getting the top-k elements from the array in k steps has an obvious performance benefit than pulling k elements using `pop()` which takes $klog(n)$ time. However, on the average case the 20-30% of the elements are not part of the real top-k elements. The critical

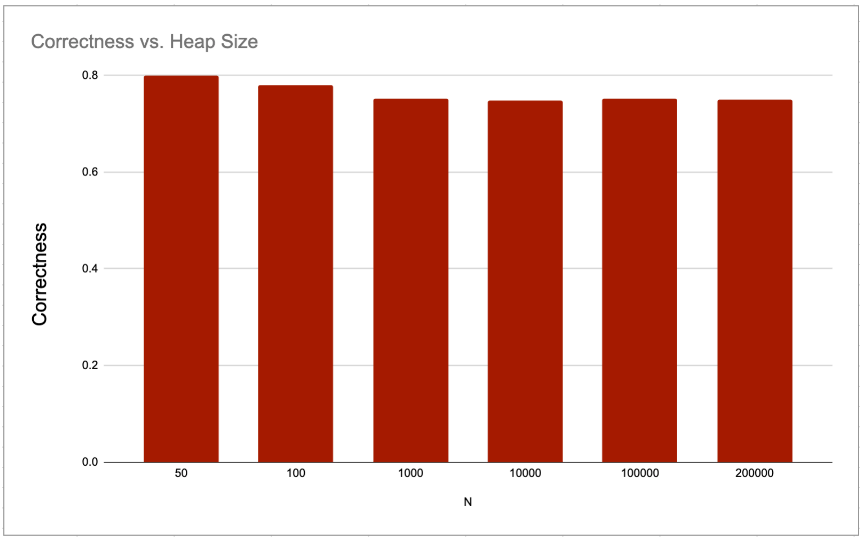


Figure 3.4: Intersection Ratio (Correctness) / Heap Size (N)

question is how this deviation from the real top-k elements will impact the randomness of our sample, and consequently, the quality of the final result. Our assumption is that this 20-30% of inaccuracy will not impact the randomness that much, so the final answer of a sample-based approximate will be ”good enough”, taking into account the performance gains that include both sequential memory access and complexity reduction from $P \log n$ to P. To investigate that, we perform a series of experiments in that we compare the qualities of the approximate answers of specific queries. We measure the error of the approximate answer that includes the real top-k elements that are extracted from the heap and the top-k elements of the underlying array. The results presented section 4 prove that both methods provide answers with comparable errors.

3.4 Summary

In this section we presented the main idea and implementation details of sample index, a novel index scheme that enhances the sampling procedure in a database system. We explained how a sample index can eliminate the overheads of the multiple `rand()` invocations and duplicate checks that hinder query performance, when samples are created during query execution. Furthermore, we presented an architecture that leverages a priority queue, that allows users to access fresh samples during continuous insertions and without delays due to the updates.

4. EXPERIMENTAL EVALUATION

This section presents a thorough performance evaluation of sample index. We implemented our sample index prototype in MonetDB, a commercial column store for high performance analytical queries. We compare the execution times of the queries included in our benchmarks with, and without using a sample index. The following experiments demonstrate that by leveraging a sample index we can achieve up to *4.5x speed-ups* compared to MonetDB's vanilla sampler that constructs the samples during query execution.

4.1 Setup

4.1.1 Hardware

All the experiments were run on a Macbook Pro with the following specifications:

- CPU: 2.9 GHz Quad-Core Intel Core i7
- Memory: 16 GB 2133 MHz LPDDR3
- Storage: 500GB SSD

4.1.2 Workload

We run four individual benchmark setups. In the first setup, we used TPC-H [2] from which we generated data of 10 and 20 GB scales. For the other three benchmarks, we used three datasets obtained from UCI Machine Learning repository [3], namely Census, Abalone and Wine. The TPC-H benchmark consists of eight tables, from which we sample only the lineitem table in each query. Each of the other three benchmarks (Census, Abalone, and Wine) consist of a series of queries but a single table, so in each query we sample the base table of the benchmark.

4.1.2.1 Sample Query Transformation

To transform each query into a sample query, we rewrite the queries as follows. For each table that we need to sample, we replace the table name ”`t`” in the query with a subquery of the form ”`select * from t sample X`”. For example, assume the following query:

```
select avg(l_quantity) from lineitem
```

If we want to run the query over a 10%-size sample instead of the full *lineitem* table, the query will be transformed as:

```
select avg(l_quantity) from (select * from lineitem sample 0.1) as tmp
```

We do the same transformation in all queries that we used in our experiments, for the varying sample size ratios (0.1, 0.2, 0.3 and 0.4).

4.2 Results

4.2.1 Proof of Concept

We begin by presenting the results of the evaluation a simple aggregation query over the lineitem table of TPC-H. We evaluate the `avg()` aggregate on the `l_quantity` column of the lineitem table, similarly to the experiment of section 1.1. Recalling the results from this introductory experiment, we showed that the sampling step of a query can occupy up to 94% of the execution time in certain cases. We showcase how a sample index can significantly reduce the time occupied by MonetDB's vanilla sampler that constructs the samples during query execution. We will be using the following query in this experiment:

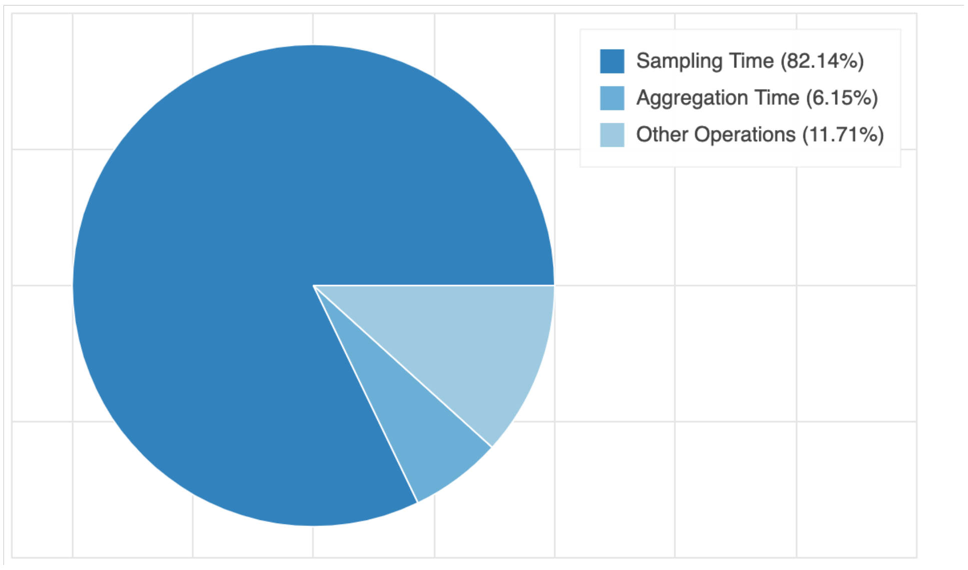
```
select avg(l_quantity) from (select * from lineitem sample $k) as tmp
```

4.2.1.1 Excessive Sampling Overhead

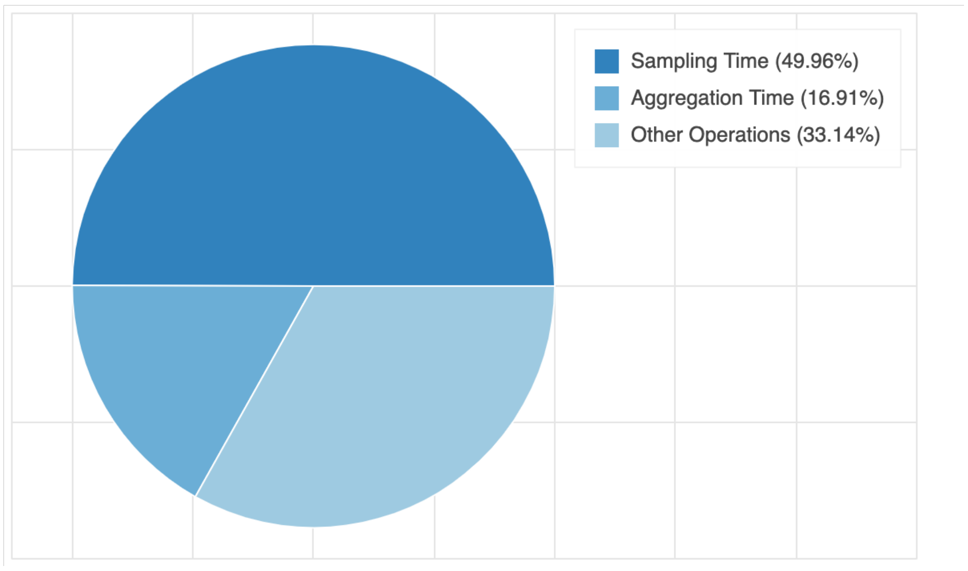
We execute this query for the sampling factors of 0.1, 0.2, 0.3 and 0.4 on the lineitem table of the 10GB TPC-H dataset. For each sampling factor we run the query both with and without using a sample index. Figure 4.1 depicts the average distribution of the query execution time of all sampling factors. We can see that 82% of the execution time is occupied by the sampling operator, while only 6.15% is occupied by the `avg()` operator and 11.71% by the rest of the operations.

4.2.1.2 Minimizing Sampling Overheads

We can see in the second pie chart that the execution time distribution changes when a sample index is used. The sampling overheads are mitigated, the time occupied by the sampler is reduced and takes 50% of the total query time. Figure 4.2 depicts the execution time per sampling size ratio. We can observe that the execution time of the query is reduced significantly when the sample index is used. As the sample size increases, the sample index can achieve up to 4.5x better execution time.



(a) Query Time Distribution without sample index



(b) Query Time Distribution with sample index

Figure 4.1: Average Query Time Distribution - TPCH 10GB

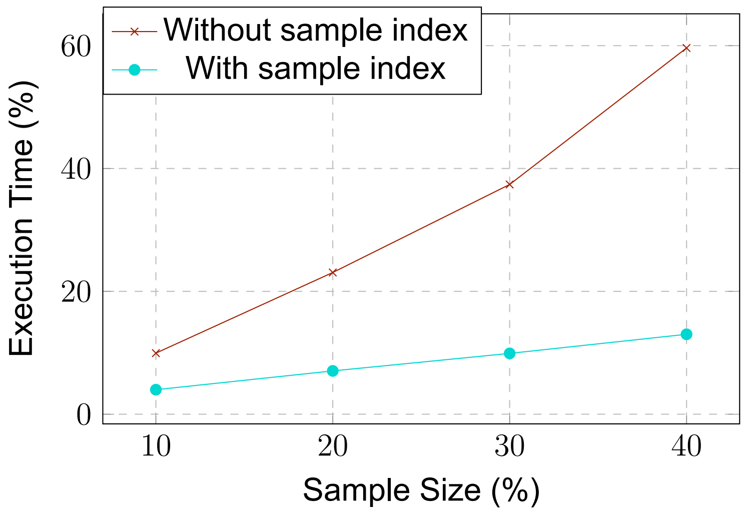


Figure 4.2: Execution Time per Sampling Size Ratio

4.2.2 TPC-H

4.2.2.1 Average Query Execution Time

We begin by comparing the average query execution time per sample size ratio in the TPC-H benchmark, both for the 10 and 20 GB scales. We summarize the results in figure 4.3. The left line charts depict the average execution time per sample size ratio. The right line charts depict the performance improvement (speed-up) achieved by sample index, that is, the fraction of the execution times (with a sample index / without a sample index).

We can observe that by using a sample index we always achieve at least x2 speedup. Moreover, it can be seen that as the sample size grows, the performance gain increases, and we can achieve more than 4x speedup in both TPC-H scales.

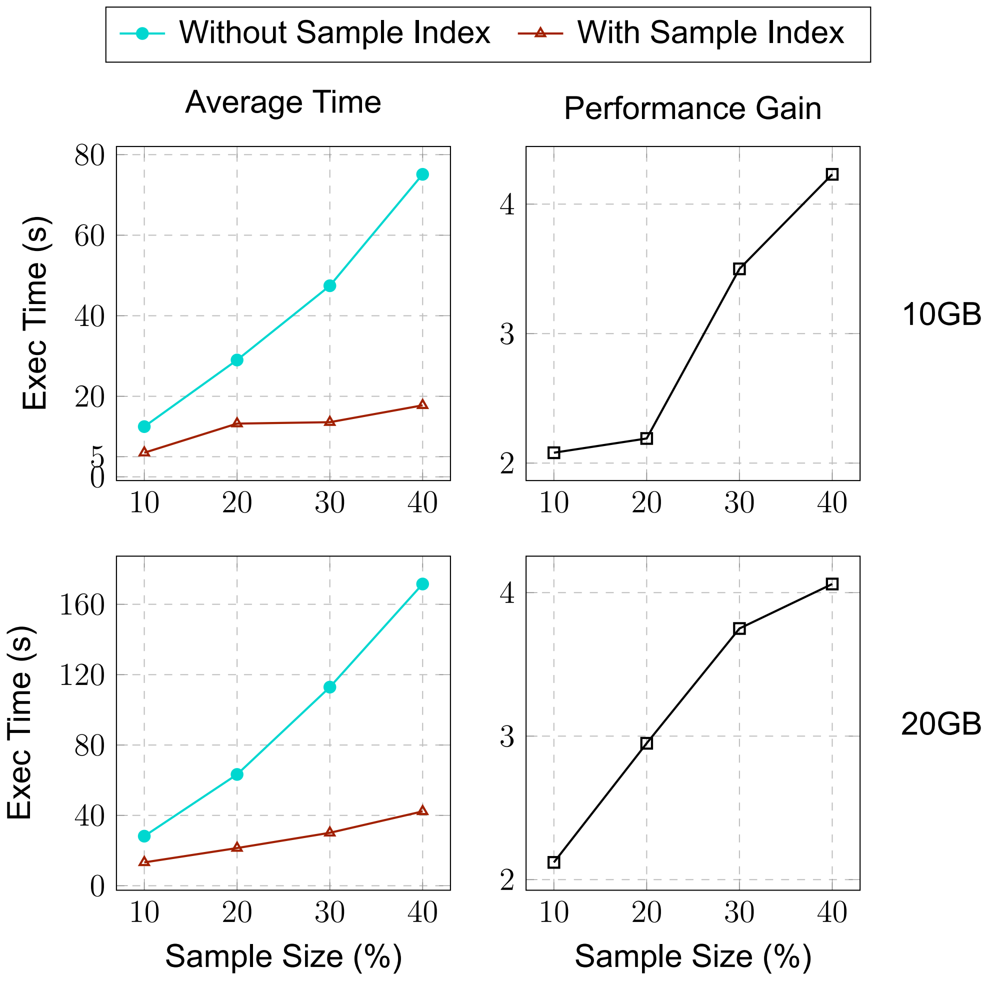


Figure 4.3: Average Execution Time and Speedup

4.2.2.2 Individual Query Performance

Next, we present the detailed execution times of all queries. The bar charts in figures 4.4 and 4.5 depict the execution time of each individual TPC-H query, both for the 10GB and 20GB scales. By observing the bars, we can see that when the sample index is used we always achieve significantly better execution time for all queries.

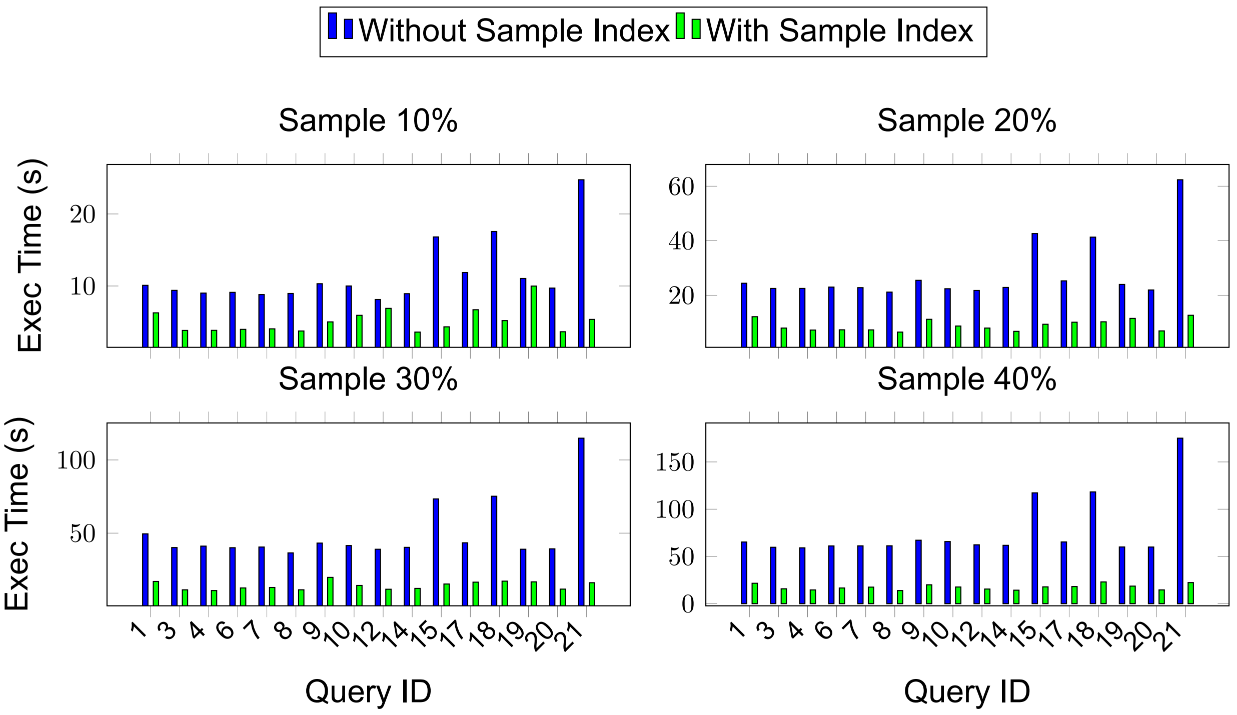


Figure 4.4: TPC-H 10GB

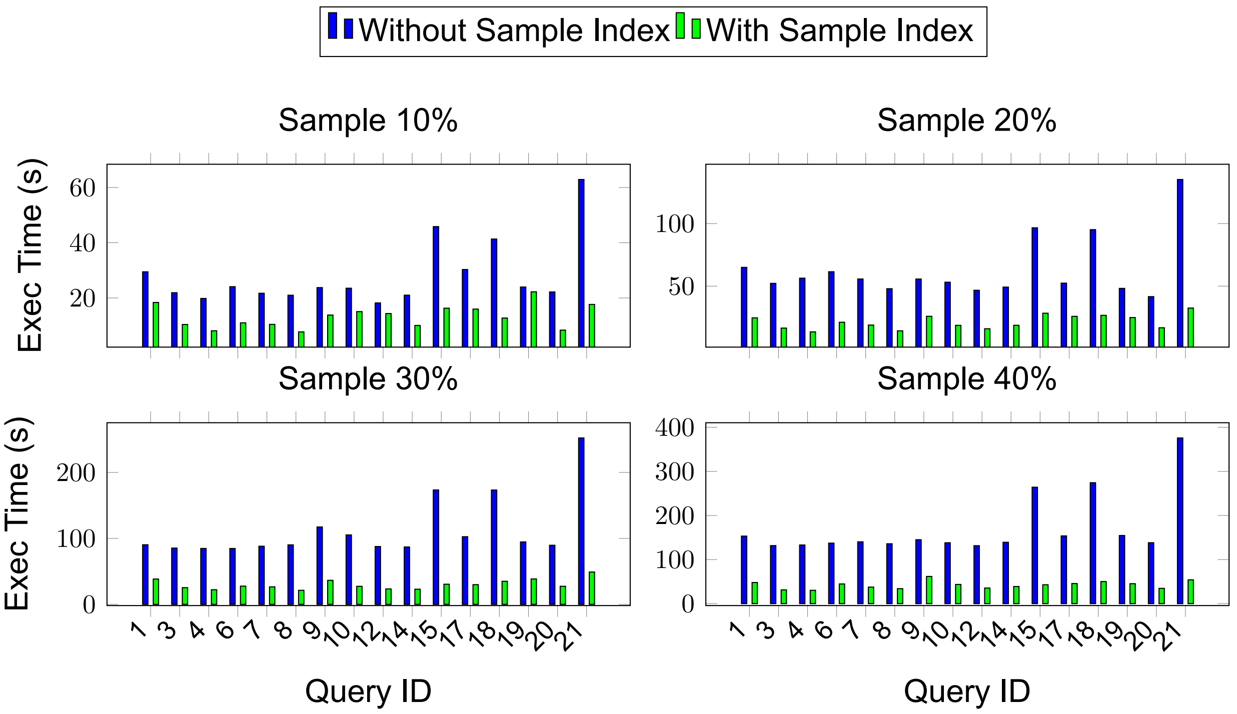


Figure 4.5: TPC-H 20GB

4.2.3 Census, Abalone & Wine

Next, we evaluate sample index over three datasets, namely Census, Abalone and Wine obtained from UCI Machine Learning Repository [3]. Because all these three datasets are relatively small, we scale up the data size of each dataset to fifty million rows, by randomly replicating the initial records. Each of these benchmarks consists of a single table. In each benchmark, we create a sample index for the base table of the benchmark. Similarly with the TPC-H benchmark, we compare the execution times of each query when the sample index is used and when it is not.

4.2.3.1 Queries

We borrowed the SQL queries from a work of Galakatos et. al. [7]. We list the SQL queries for the Census, Abalone and Wine datasets in tables 4.1, 4.2 and 4.3 respectively.

Table 4.1: Census Queries

ID	Query
1	`select count(*) from adult as tmp group by sex`
2	`select count(*) from adult as tmp group by education`
3	`select count(*) from adult as tmp where sex = 'Female' group by education`
4	`select count(*) from adult as tmp where sex = 'Male' group by education`
5	`select count(*) from adult as tmp group by education, sex`
6	`select count(*) from adult as tmp where education = 'Doctorate' group by sex`
7	`select count(*) from adult as tmp group by capital_gain`
8	`select count(*) from adult as tmp where education = 'Doctorate' group by capital_gain`
9	`select count(*) from adult as tmp group by sex, capital_gain`
10	`select count(*) from adult as tmp where sex = 'Female' group by capital_gain`
11	`select count(*) from adult as tmp where education = 'Doctorate' group by capital_gain`
13	`select count(*) from adult as tmp where sex <> 'Female' and education = 'Doctorate' group by capital_gain`
14	`select count(*) from adult as tmp group by age`
15	`select count(*) from adult as tmp where age >= 20 and age < 40 and sex <> 'Female' and education = 'Doctorate' group by capital_gain`

Table 4.2: Abalone Queries

ID	Query
1	`select count(*) from abalone as tmp group by age`
2	`select count(*) from abalone as tmp group by whole_weight`
3	`select count(*) from abalone as tmp where age >= 16`
4	`select count(*) from abalone as tmp where age < 8 group by whole_weight`
5	`select count(*) from abalone as tmp group by sex`
6	`select count(*) from abalone as tmp where age >= 16 group by sex`
7	`select count(*) from abalone as tmp group by whole_weight, sex`
8	`select count(*) from abalone as tmp where whole_weight >= 2 group by age`
9	`select count(*) from abalone as tmp where whole_weight >= 2 and sex = 'I' group by age`
10	`select count(*) from abalone as tmp where whole_weight >= 2 and sex <> 'I' group by age`
11	`select count(*) from abalone as tmp where whole_weight < 0.4 group by age`
12	`select count(*) from abalone as tmp where whole_weight < 0.4 and sex = 'I' group by age`

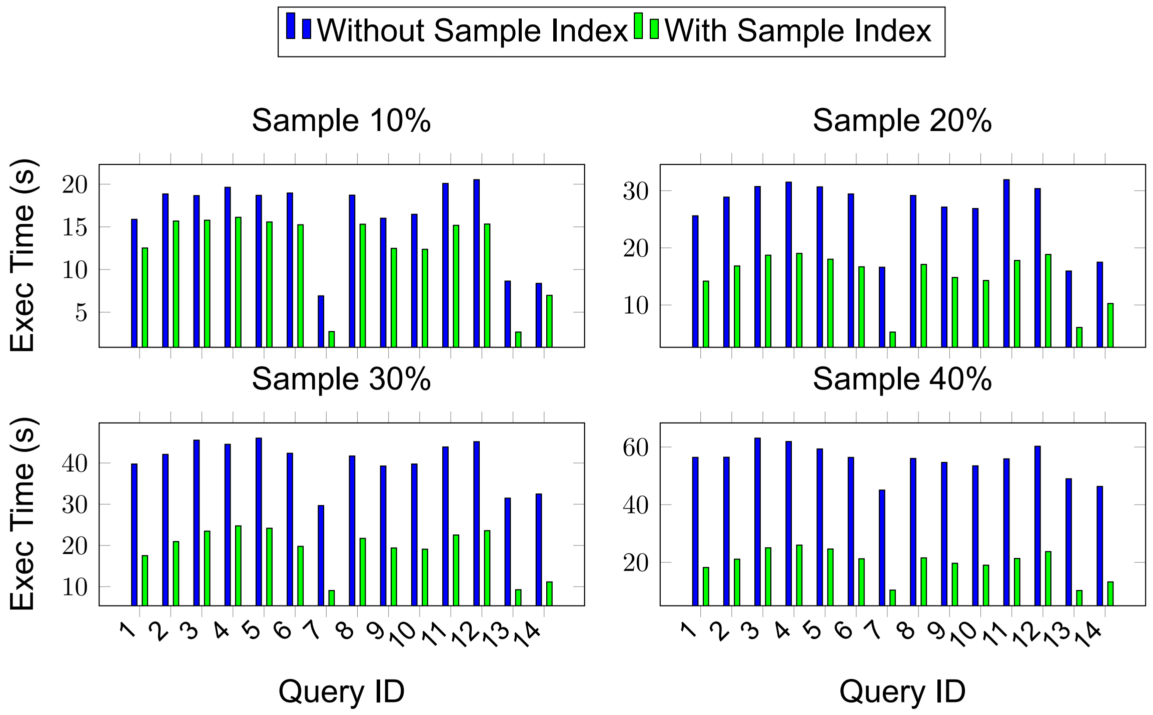


Figure 4.6: Census Execution Times

Table 4.3: Wine Queries

ID	Query
1	`select count(*) from wine as tmp group by score`
2	`select count(*) from wine as tmp group by abv`
3	`select count(*) from wine as tmp where score >= 8 group by abv`
4	`select count(*) from wine as tmp where score < 8 group by abv`
5	`select count(*) from wine as tmp group by so2`
6	`select count(*) from wine as tmp where score >= 8 group by so2`
7	`select count(*) from wine as tmp where score < 8 group by so2`
8	`select count(*) from wine as tmp group by so2, abv`
9	`select count(*) from wine as tmp where abv >= 13 group by score`
10	`select count(*) from wine as tmp where so2 >= 100 and so2 <= 200 and abv >= 8 group by score`
11	`select count(*) from wine as tmp where abv >= 8 group by score`
13	`select count(*) from wine as tmp where so2 >= 100 and so2 <= 200 group by score`

4.2.3.2 Results

Similarly to the TPC-H experiments, we present the speedups achieved and the execution time of each individual query. Figure 4.9 depict the speedups, while figures 4.6,4.7 and 4.8 depict the individual execution times for each query. Again, we can see that in all cases the sample index outperforms the vanilla sampling implementation of MonetDB, with similar speedups that increase to the sample size ratio as in TPC-H experiments.

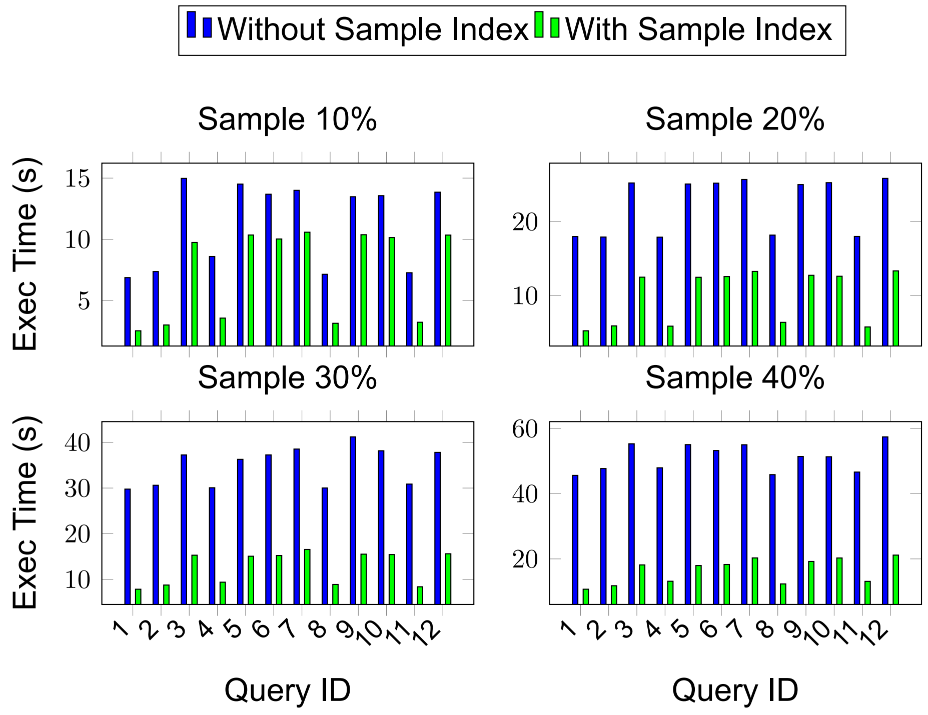


Figure 4.7: Abalone Execution Times

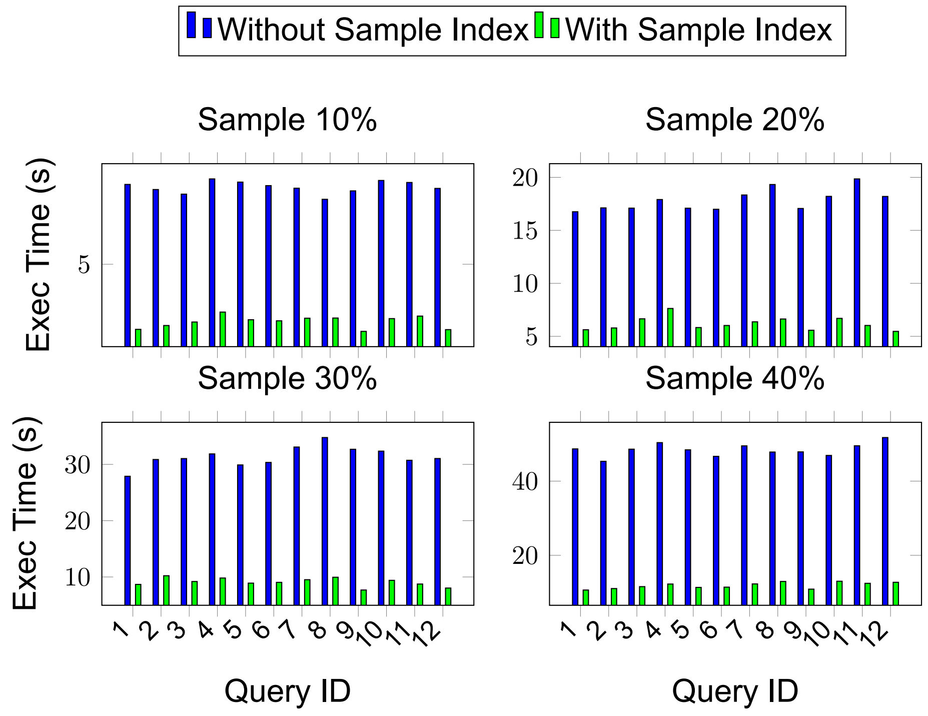


Figure 4.8: Wine Execution Times

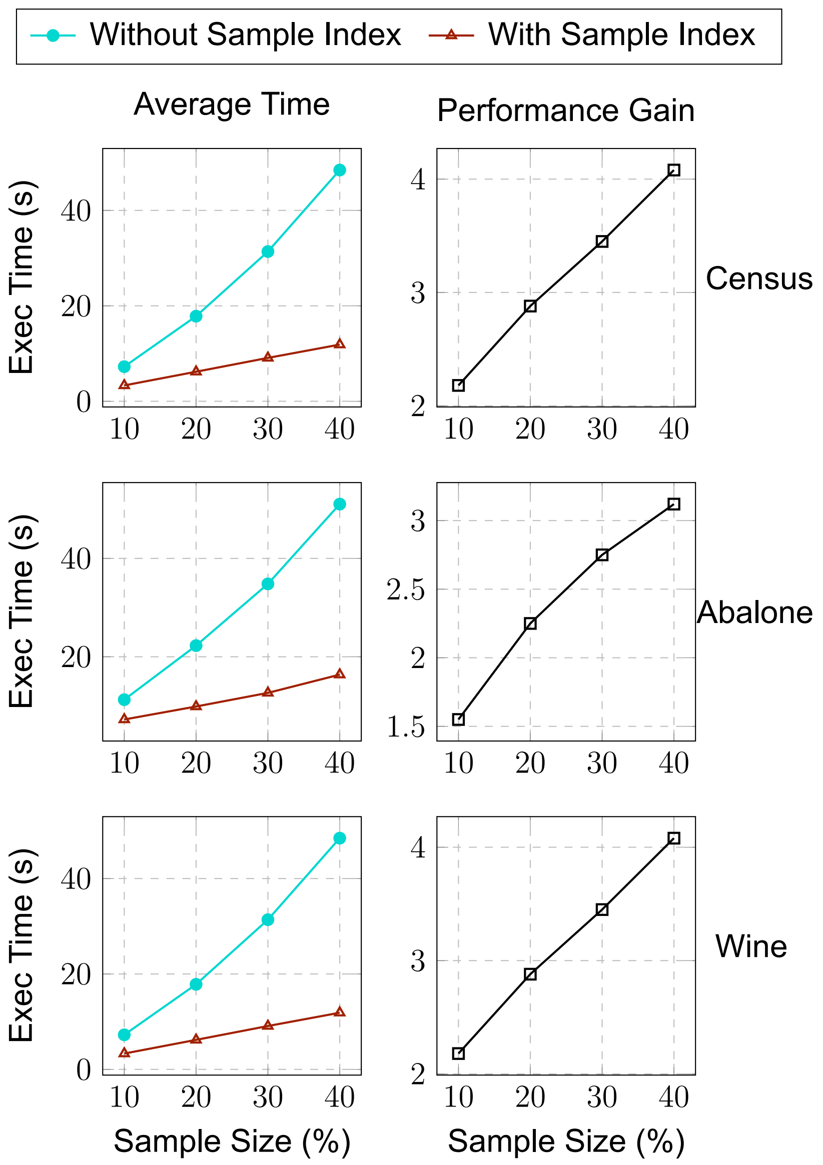


Figure 4.9: Average Execution Time and Speedup

4.3 Online Sample Index Evaluation

In section 3.3, we presented an architecture that employs a min-heap priority queue in order to handle continuous insertions efficiently. We explained how this architecture is able to provide fresh samples to the user by mixing data both from the persisted sample index and the min-heap. We highlighted that pulling data from the min-heap is expensive, as for each single record that we need to fetch, we need $log(n)$ steps due to the `heapify` process that min-heap runs in order to move the next minimum element to the root. In order to mitigate this logarithmic overhead, we presented an approach in that instead of pulling k elements from the min-heap using the `pop()` method, we simply pick the top-k

elements from the array of the heap sequentially. Our preliminary analysis lead us to the conclusion that by picking the top-k elements of the array of the min-heap, the result is $70-80$% correct compared to the 100% correct result that we could get using the `pop()` method, which always return the next minimum element.

In this section, we explore the accuracy of the aforementioned approach. We run same series of approximate queries as in the previous experiments over the Census, Abalone and Wine datasets. Now, instead of measuring the performance, we evaluate the result accuracy. To do that, we index each of the aforementioned datasets into a min-heap using a random value generator to create a random key for each record. Then, we run each of our queries in each benchmark over a 10% sample of the dataset. We obtain the sample with two ways: *i.* by fetching the k samples using the `pop()` method which ensures that we get the correct k minimum elements and *ii.* by fetching the top-k elements of the array of the min-heap. We then execute the queries over the samples and we compare the result accuracy over these two methods.

4.3.0.1 Workload and Dataset

We used the queries presented in [7], and we executed them over the Census, Abalone and Wine datasets. Similar to [7], we treat each query as a categorical random random variable θ. For example, if a query contains the selection `WHERE sex = 'Male'` on a table T and the number of rows that satisfy the selection are s, then the theta value of that query will be $\theta_{Male} = \frac{s}{|T|}$. For multiple categories, we compute theta for all distinct categories.

Now, in order to produce approximate answers for each distinct category x we use a *maximum likelihood estimator (MLE)*. Thus, the approximate answer for a sample of size n, will be estimated as $\hat{\theta}_x^{MLE} = \frac{n_x}{n}$, where n_x the number of instances in the sample of the category x.

4.3.0.2 Error Metric

In order to quantify the result error, we use the *standard error* formula, normalized by the MLE $\hat{\theta_x}$. Thus, the normalized error formula is $error(x) = \frac{1}{\hat{\theta_x}}\sqrt{\frac{\theta_x(1-\theta_x)}{n}}$.

4.3.0.3 Results

We compare the errors between answering queries using samples generated by fetching the real top k values from the heap using the `pop()` versus the top k values obtained by fetching the first k elements of the array. As aforementioned, the latter includes a 20-30% of falsely selected samples, in terms of randomness. In this experiment we do not expect to achieve low error in all cases, as we use uniform/random sampling and we do not take into account corner cases like rare sub-populations, which affects the quality of the result.

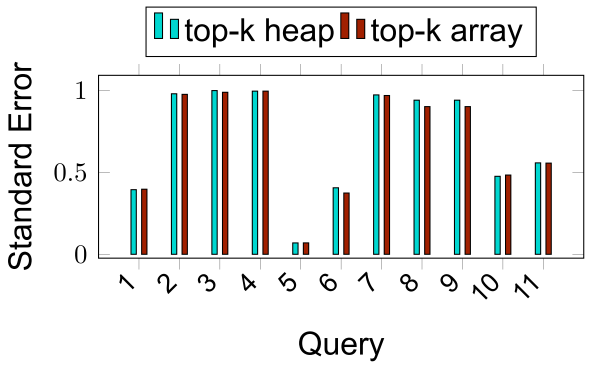


Figure 4.11: Standard Error - Abalone

However, we showcase that the 20-30% randomness error has negligible impact on the error final result, as both approaches result into similar errors. The bar charts 4.10, 4.11 and 4.12 depicts the results. We can see that in all cases the sample obtained with the first k elements from the heap's array results into a similar error with the samples obtained by obtaining the true random samples, using the `pop()` operation of the heap. The conclusion of this experiment, is that even if the sample is not a 100% random sample, the quality of the result will be not affected, compared to a true random sample.

As a result, in the continuous insertion scenario in which we combine data both from the sample index and the min-heap, this finding allows us to replace the expensive logarithmic read of the min-heap with a sequential scan in the array, without worrying about the quality of the sample.

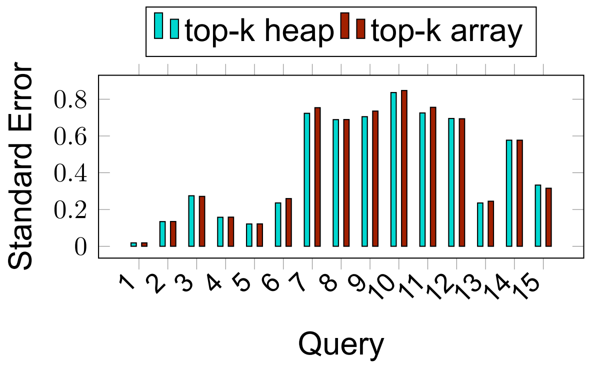


Figure 4.10: Standard Error - Census

4.4 Summary

In this chapter we presented the results of our experimental evaluation. We used queries and data generated from the TPC-H benchmark using the scales of 10GB and 20GB. Furthermore, we used three additional datasets, namely Census, Abalone and Wine. By

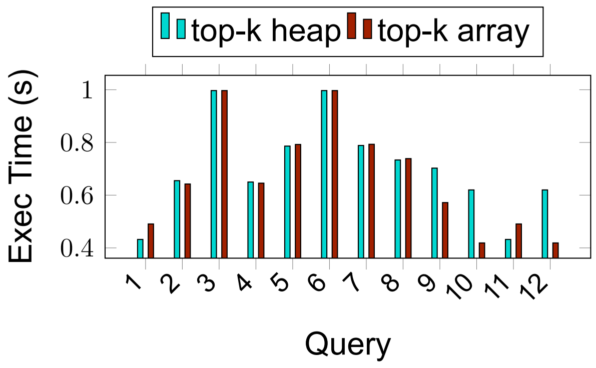


Figure 4.12: Standard Error - Wine

mitigating the overheads of the naive sampling approach, our results showcase that our MonetDB implementation with sample index can achieve up to 4.5x speedups in sample queries, by enhancing the sampling operator performance.

First, we analyzed the distribution of the execution time of a sample query, and showed that on average, 87% of the time is occupied by the sampler, and that most of the sampling time is taken by the excessive `rand()` invocations. We showed how a sample index can eliminate these overheads by precomputing the random values only once, during index construction. Next, we evaluated our implementation over four benchmarks: TPC-H, Census, Abalone and Wine. Our results proved that when a sample index is used, we can always achieve better execution time than the vanilla MonetDB sampler implementation.

5. RELATED WORK

In this section we present the related work on sampling in database systems. We classify the related works into three categories. The first category is called *sampling relational databases* and presents the state-of-the art methods and early works on how sampling methods are integrated with core components of a relational database system, like relational operators and materialized views. The next section namely *sampling-based approximate query processing* describes works that focus on how sampling methods, like uniform and stratified sampling, can be leveraged in approximate query processing systems. Finally, in the last section entitled *online aggregation* we discuss about approaches that focus on how the user can be periodically informed about the query result during query execution. Most of these works are based on uniform or stratified sampling, where fixed-sized samples are usually pre-computed offline, or, created during query execution.

5.1 Sampling Relational Databases

5.1.1 Random Sampling from Databases

Random sampling has a long history in database systems research. One of the first attempts was made by Frank Olken back at 1986 in one of his works entitled *Simple Random Sampling from Relational Databases* [18] and his Ph.D. dissertation [17]. His work can be described as a framework that consists of methods for obtaining random samples from relational operators like `scan`, `select` and `join`. The goal his work is to achieve execution time proportional to the sample size, when sample-based queries are issued to the system. Leveraging a sampling operator in query evaluation can reduce data access costs, i.e. disk page I/O, in this type of queries. Of course, all these techniques are suitable for use-cases in that the full data is not required, and an fast approximate answer is more preferable.

5.1.1.1 Sampling Methods

Various sampling methods are presented in the paper including sampling with and without replacement, as well as weighted random sampling techniques like acceptance/rejection sampling and partial sum trees. For example, in the acceptance/rejection approach, the j^{th} record (r_j) of a N-size population is selected with *inclusion probability* $p_j = \frac{w_j}{w_{max}}$, where w_j the weight of the j^{th} row and w_{max} the maximum weight. Then, another uniform value u_j is drawn and r_j is accepted if $u_j < p_j$.

5.1.1.2 Sampling Relational Operators

As aforementioned, Olken's work focuses on integrating sampling algorithms with relational operators and construct random samples from the output of these operators. He presents a formal framework that can be used to describe various sampling methods over relational operators with the following notations. First, the sampling operator is represented as ψ, while an expression like $\psi_{SRS,100}(R)$ represents a simple random sample without replacement of the relation R. Next, more complex schemes are represented by iteration operators like $WR(s, < expr >)$. WR stands for *with replacement*, while the expression $< expr >$ will be applied repeatedly until a sample of size s is obtained.

5.1.1.3 Operator-Specific Sampling

- **Selection:** First, selection (a.k.a. `where` clause) naturally integrates with sampling, as sampling from the result of a selection is equivalent to selecting from a sample. All records that do not satisfy the selection predicate have zero inclusion probability, while the rest of the records have equal inclusion probability of being included in the sample $p = \frac{s}{n_{pred}}$, where s the sample size and n_{pred} the number of records that satisfy the predicate. Multiple sampling algorithms can be utilized for the selection operator depending on weather there is an index or not. In the latter case case, a full scan is issued first.

- **Projection:** Projection (a.k.a. `select` clause) is more complex, if the projected attribute A is not a primary key of the relation, then the projection cannot commute with simple random sampling. The main reason is that projection removes duplicates. As a result, each value α of attribute A appears only once when A is projected. Formally, this can be represented by the relation $\psi_s(\pi_A(R)) \not\Leftrightarrow \pi_A(\psi_s(R))$. In other words, projecting a sample is not the same as sampling a projection.

- **Join:** Similarly to selection, sampling a join operator depends on a number of factors. For example, we should question when the join attribute is a key in one or more relations, if the relations are indexed or hashed by the join attribute and also, what is the cardinality of the join output. Given two relations R and T, a simple random sampling algorithm over that join is the following. First, sample one element from R, join it with T and produce the intermediate result V. Finally, sample one element from V with accept/reject sampling, with acceptance probability proportional to the cardinality of V. This procedure should be repeated until a sample of the desirable size is obtained. This is the *join without key* algorithm. The join *with key* algorithm is much simpler. If the join attribute X is a key on one relation, let's say T, then we can obtain an s-size simple random sample from the join operator by iteratively obtaining a random record from R and then joining it with T until we get a the s-size sample.

5.1.1.4 Maintaining Sample Views

In the final chapter of his dissertation, Olken presents the idea of maintaining what he calls *Materialized Sample Views*. This is one of the early attempts to reuse fixed-sized, pre-computed samples to enhance sampling query performance.

Briefly, a view is a relation derived from a query. It can be either *virtual* or *materialized*. If the view is virtual, each time the view is referred the associated query needs to be run in order to produce the view result. If it is materialized, then the query is executed once and the result is stored as a materialized view. In the latter case, when a table that somehow relates with that view (base relation) is updated, the materialized view needs to be updated as well. Subsequently, a sample view is an analogous scenario to the traditional view, but the underlying query is a sample-based query.

Olken's dissertation focuses on strategies for updating materialized sample views. A change in the base relation can be either an `insert`, an `update` or a `delete`. The maintenance strategy depends on the change and the view type. For example, a view can be selection on a single relation, or a join between two or more relations.

For each update type, there is a corresponding strategy that is followed. For a view that results from a selection on one relation the strategy is simple. For example, in case of insertion the possible paths are two. If the newly inserted record satisfies the selection predicate, then an insertion algorithm is invoked which decides if the record is going to be included in the sample or not. If the new record does not qualify the selection predicate, no action is required. In case of deletion, if the deleted record is included in the sample, then another sample of size 1 needs to be obtained to replace the deleted record. Finally, in case of update, three paths are possible. If the updated record satisfies the selection predicate after the update, an insertion algorithm should be invoked, similarly to what happens in case of the insertion. If the update changes the record so it does not qualify the predicate, then it analogous to the deletion scenario. Finally, if the change does not affect the predicate state, no action is required. Similar strategies are followed for project views, while more complex ones for joins.

To conclude, Frank Olken's work on Simple Random Sampling on Relational Databases introduces a complete framework that includes algorithms, mathematical formulas and their corresponding proofs for obtaining random samples from relational operators. His methods are also integrated with the use-case of materialized views.

One shortcoming in the aforementioned methods, is that there is no an appropriate access method for obtaining samples efficiently during query execution. The algorithms presented need to execute part of the relational operator and sample from it, which can be expensive for some operators. For example, in the case of joins, the full join should be executed before sampling. Furthermore, the idea of materialized sample views require extra storage for maintaining redundant information that already exists in the base relations, while they also pose a significant maintenance overhead in case that frequent updates are issued. In that case, the materialized views might become invalid frequently and will either *i)* return outdated results or *ii)* result into a downtime before queries can be issued until the sample view is updated. In contrast, our proposed sample index requires negligible update cost due to the underlying priority queue structure, while it can serve fresh samples to the user by combining data both from the sample index and the recently inserted data in the priority queue.

5.1.2 Materialized Sample Views for Database Approximation

We now continue our discussion on materialized sample views. In this paper, Shantanu Joshi et al. [12] introduce a concept similar to the one of F. Olken, but more efficient for range queries. Briefly, this work presents materialized views backed up by an indexing scheme called ACE Tree, such as a materialized view is indexed on a specific column. As a result, the proposed indexing abstraction provides good performance for range sampling queries.

We will give a brief description of the ACE tree. A leaf node L is associated with a set of h ranges, where h the tree size. Each range $L.R_i$ is also associated with a section $L.S_i$ that contains a simple random sample of all records in the table within this range. In fact, each leaf node contains sample sets for all internal nodes in its path from the root. An internal node is associated with a range R, a value that splits the range to the left and right children, the pointers, and the count of each child. As a result, given a range query $Q = [l, r]$ where l the left bound and r the right bounds of the range, the query will traverse all nodes that overlap with $[l, r]$ until the corresponding leaf nodes that contain the samples are reached.

The ACE tree is characterized by three properties. First, *Appendability* means that the union of the i^{th} sections of two leaves, that is, $L_j.S_i \cup L_k.S_i$ is always a true random samples of all records in the range $L_j.R_i \cup L_k.R_i$. Next, the *Combinability* property lets samples retrieved from different leaves to be combined in order to produce an ever increasing sample. Finally, *Exponentiality* guarantees all records that fall in $L.R_i$ are twice than the records that fall in $L.R_{i+1}$.

The ACE tree has a key drawback. It is developed upon the assumption of static data and bulk creation, while incremental updates are not supported. The lack of incremental updates is very restricting for real-time applications where lots of updates are issued, and the downtime due to the ACE tree reconstruction would be unacceptable. In contrast, the sample index supports efficient incremental/batch updates by leveraging an in-memory priority queue structure that is merged with the persisted sample index periodically.

5.2 Sampling-Based Approximate Query Processing

5.2.1 Revisiting Reuse for Approximate Query Processing

Galakatos et al. [7] presented an approximate query processing framework with low error bounds that takes advantage of previously computed results and reuses them. The approach treats each query as a random variable. For example, consider following query:

```
select sex, count(*)
from census
group by sex
```

In this example, the `sex` attribute is considered as a categorical random variable that belongs in the set of sub-populations $\{Male, Female\}$. If $1/3$ of the samples are females, it is represented as $\theta_{Female} \approx 1/3$. The system aims at providing approximate answers in terms of *maximum likelihood estimation*, described as $\hat{\theta}_x^{MLE}$. The architecture consists of a *Sample Store* that is populated by a stream of tuples from an integrated data source through some random sampling method. The generated approximate results are stored in the *Result Cache* of the system. The cached results can be used by the Query Engine for future query evaluation. In addition, the system provides a hash-based data structure called *Tail Index* that keeps track of rare sub-populations in the Sample Store. By utilizing Tail Indexes, the system can maintain a low approximation error while preserving the required randomness.

However, the Tail Index requires pre-computation of fixed-size samples that might become invalid frequently in case of vast updates are issued, and the system will need to reconstruct the samples.

5.2.2 BlinkDB

BlinkDB [4] is a system that aims in achieving balance between efficiency and generality of the query workload by providing time or error bound constraints. It assumes that queries are predictable based on the column set (QCS) frequencies of past queries. In order to reduce sampling error due to rare sub-populations when a group-aggregate query is issued, BlinkDB employs stratified sampling [16]. Given an input query, for each stratified sample kept, BlinkDB creates a latency-error profile (ELP). Based on ELP and the contstraint's defined by the user, BlinkDB picks the appropriate sample to serve the query. Depending on the query and objective, BlinkDB will select the appropriate stratified samples in order to meet the error and execution time requirements provided by the user.

However, BlinkDB assumes that the query column sets (QCS) used in aggregation functions remain stable over time, which can be restricting in case that the query workload changes. Moreover, it depends on historical data of past QCS frequencies in order to pre-compute fixed-sized samples and serve future queries. As a result, fixed-sized pre-computed samples need to be recomputed again in case that the workload or the underlying data changes.

5.2.3 VerdictDB

Another, yet more commercial project is VerdictDB [20], that introduces a database-agnostic technology called *Universal Approximate Query Processing (UAQP)*. VerdictDB introduces a universal solution that can be plugged in any relational database without any further modification to the internals of the attached system. As a result, in can connect through the appropriate JDBC/ODBC driver to any SQL-based database engine (e.g. Spark SQL, Impala or Azure SQL). Its architecture contains an Query Parser that parses a raw-text SQL query and an AQP Rewriter that rewrites the SQL query accordingly to perform AQP to the underlying database. VerdictDB prepares the samples for specific tables offline, using either uniform, hashed [9] or stratified sampling. VerdictDB paper also introduces a novel error estimator technique, called variational subsampling, a variation of subsampling, but with smaller time complexity.

While VerdictDB is a sampling-based AQP engine, differs from our sample index approach, as its main goal is to support multiple execution engines without modifying them. Furthermore, VerdictDB also needs to pre-compute fixed-sized samples for each table in the database.

5.3 Online Aggregation

Online Aggregation [11] is a query processing technique that can be used to inform the user about the query process during execution. For example, consider an aggregation (like `AVG`) function over a table scan or a group-by clause. Traditionally, this is executed in a batch manner, that is, the user has to wait until completion. With online aggregation the user is able to get an approximate answer as query executes, with some confidence interval and probability. Key objectives in online aggregation are the time required until a useful result is presented, the time required to produce the final query answer, as well as the rate in that the result is updated during execution. The access method is critical of a correct online approximation, as the records should be accessed in a random order. As a result, heap files are more desirable, or index scans in case that the aggregation attribute is not indexed.

As aforementioned, providing a useful result in a short time is a key objective for an online aggregation system. Joins are one of the most expensive operations in a database system, but are usually optimized to run in a batch manner. Ripple joins [8] address this problem in online aggregation systems, by providing a set of join algorithms for providing a precise approximate result in reasonable time. Usually, the sampling method for a join operator requires a number of sampling steps of one relation, and for each sampling step a full join with the other relation is required. This is completely avoided by Ripple joins, as both relations are sampled at each steps, and the joins is performed between the samples. Again, all these algorithms work under the assumption that the records of a table are retrieved in a random order. The reason is that the data being processed is helpful to be viewed as a random sample. In a follow-up paper entitled ”Informix under

CONTROL", Hellerstein et al. [10] present a set of access methods and database physical design suitable for randomized data access. The main solution is to store records of a table in random order, ensuring that the records fetched always constitute a random sample. Furthermore, if the database system supports some kind of `rand()` function and *functional indices*, a secondary index can be created to preserve random ordering. However, a secondary index requires a random I/O per record. Moreover, as random ordering is not always desirable, a user-defined ordering stage is provided where the user can define a custom reordering of the data. The reordering process takes place during query execution.

Join optimization for online aggregation has been also studied in [25], a method that aims to optimize random sampling over multi-way joins. Furthermore, Wander Join [14, 13, 15] presents another method for optimizing joins for online aggregation by employing random walks instead of sampling or random ordering. Online aggregation has also gained significant attention in the domain of Big Data and distributed data processing paradigms. Notably, [6, 19, 5] introduce how online aggregation can be integrated with MapReduce frameworks.

Most of these works require the data to retrieved in a random order. However, none of these methods provides some access method to efficiently fetch the data in random order, and the randomization process takes place during query execution. Our experimental evaluation showcases that adding randomization during the critical path of the query execution usually takes over the 90% of the sampling time.

5.3.1 Continuous Sampling for Online Aggregation Over Multiple Queries

Next, COSMOS [24] is another sampling-based system for online aggregation, optimized for processing efficiently multiple queries. The key component is the data *Scrambler*. Traditionally, the samples are generated during query execution, or they are pre-computed in fixed-sizes. Instead, with the Scrambler feature the initial dataset in shuffled, by performing multiple passes over the whole dataset and putting each element into some random position. Updates are collected as batches, and the whole table needs to re-scrambled in order for the updates to get integrated. The system treats a set of multiple queries as a dissemination graph, where each edge represents a dependency between two queries. Thus, results from parent nodes can be reused from descendant nodes.

COSMOS has the following major drawbacks. First, the scramble algorithm needs to be executed multiple times over the full dataset. For large tables, this will lead in excessive pre-processing time. Furthermore, the scrambling algorithm does not guarantee the output uniformity, as simple random sampling does [23]. Updates are also handled in a batch-manner, which means that the full dataset needs to be re-scrambled, while no queries can be executed while updating. For real-time applications, this downtime is unacceptable.

6. CONCLUSIONS AND FUTURE WORK

We presented sample index, a new index structure that enhances the sampling operator's performance in a database system. We showed that the state-of-the art approach that constructs samples during query execution can significantly hinder query performance, due to the costly sampling steps involved in the process. We explained how we can eliminate the overheads by replacing the expensive sampling steps in query execution with a sequential scan on our sample index. Through our experimental evaluation, we demonstrated that we can achieve up to 5x performance improvements.

There are several ongoing extension ideas that we are working on in order to enrich the abilities of the sample index. One of our main future goals is to support horizontal partitioning of the sample index for multi-threaded execution. This can be achieved by merging multiple priority queues, each corresponding to a different partition. In addition, we are interested in improving the expensive logarithmic read cost of the min heap priority queue by replacing it with a sequential scan. Our current solution slightly sacrifices the randomness of the sample. However, as our experimental evaluation proves, our first findings show that this slight ”loss” in the randomness of the sample has negligible impact in the quality of the final approximate query results, which implies that we could afford to exchange this loss with a significant read performance improvement.

BIBLIOGRAPHY

[1] Random Binary Trees. Wikipedia. `https://en.wikipedia.org/wiki/Random_binary_tree`.

[2] TPC-H. `http://www.tpc.org/tpch/`.

[3] UCI Machine Learning Repository. `https://archive.ics.uci.edu/ml/index.php`.

[4] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion 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.

[5] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *Nsdi*, volume 10, page 20, 2010.

[6] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M Hellerstein, John Gerth, Justin Talbot, Khaled Elmeleegy, and Russell Sears. Online aggregation and continuous query support in mapreduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 1115–1118, 2010.

[7] Alex Galakatos, Andrew Crotty, Emanuel Zgraggen, Carsten Binnig, and Tim Kraska. Revisiting reuse for approximate query processing. *Proceedings of the VLDB Endowment*, 10(10):1142–1153, 2017.

[8] Peter J Haas and Joseph M Hellerstein. Ripple joins for online aggregation. *ACM SIGMOD Record*, 28(2):287–298, 1999.

[9] Marios Hadjieleftheriou, Xiaohui Yu, Nick Koudas, and Divesh Srivastava. Hashed samples: selectivity estimators for set similarity selection queries. *Proceedings of the VLDB Endowment*, 1(1):201–212, 2008.

[10] Joseph M Hellerstein, Ron Avnur, and Vijayshankar Raman. Informix under control: Online query processing. *Data Mining and Knowledge Discovery*, 4(4):281–314, 2000.

[11] Joseph M Hellerstein, Peter J Haas, and Helen J Wang. Online aggregation. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 171–182, 1997.

[12] Shantanu Joshi and Christopher Jermaine. Materialized sample views for database approximation. *IEEE Transactions on Knowledge and Data Engineering*, 20(3):337–351, 2008.

[13] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. Wander join: Online aggregation for joins. In *Proceedings of the 2016 International Conference on Management of Data*, pages 2121–2124, 2016.

[14] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. Wander join: Online aggregation via random walks. In *Proceedings of the 2016 International Conference on Management of Data*, pages 615–629, 2016.

[15] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. Wander join and xdb: online aggregation via random walks. *ACM Transactions on Database Systems (TODS)*, 44(1):1–41, 2019.

[16] Sharon L Lohr. *Sampling: design and analysis*. Nelson Education, 2009.

[17] Frank Olken. *Random sampling from databases*. PhD thesis, University of California, Berkeley, 1993.

[18] Frank Olken and Doron Rotem. Simple random sampling from relational databases. 1986.

[19] Niketan Pansare, Vinayak Borkar, Chris Jermaine, and Tyson Condie. Online aggregation for large mapreduce jobs. *Proceedings of the VLDB Endowment*, 4(11):1135–1145, 2011.

[20] Yongjoo Park, Barzan Mozafari, Joseph Sorenson, and Junhao Wang. Verdictdb: Universalizing approximate query processing. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1461–1476, 2018.

[21] Lefteris Sidirourgos, Martin Kersten, and Peter Boncz. Scientific discovery through weighted sampling. In *2013 IEEE International Conference on Big Data*, pages 300–306. IEEE, 2013.

[22] Lefteris Sidirourgos, Martin L Kersten, and Peter Boncz. Sciborq: Scientific data management with bounds on runtime and quality. In *CIDR*, volume 11, pages 296–301, 2011.

[23] AB Sunter. List sequential sampling with equal or unequal probabilities without replacement. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, 26(3):261–268, 1977.

[24] Sai Wu, Beng Chin Ooi, and Kian-Lee Tan. Continuous sampling for online aggregation over multiple queries. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 651–662, 2010.

[25] Zhuoyue Zhao, Robert Christensen, Feifei Li, Xiao Hu, and Ke Yi. Random sampling over joins revisited. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1525–1539, 2018.