DISTRIBUTED SAMPLING IN A BIG DATA MANAGEMENT SYSTEM

DAN RADIUS

University of Washington
Department of Computer Science and Engineering
Undergraduate Departmental Honors Thesis

Advised by Dan Suciu
## Contents

1 Introduction .......................... 2
   1.1 Overview of Myria .................. 2
   1.2 Query Languages ..................... 3
   1.3 Sampling Processes ................... 3
   1.4 Sampling Operations in MyriaL ........ 4

2 Relational Algebra COmpiler (RACO) 6
   2.1 Relational Algebra ................. 6
   2.2 Overview of RACO .................... 6

3 Myria Operator Implementation 8
   3.1 Physical Query Plans ............... 8
   3.2 Bernoulli & Poisson Sampling ........ 9
   3.3 New Operators ....................... 9
   3.4 Distributed Sampling Algorithm ..... 10
      3.4.1 Sampling a Base Relation ........ 10
      3.4.2 Sampling an Arbitrary Expression ... 10

4 Query Plan Optimization 13
   4.1 Flip .................................. 13
   4.2 Sample ................................. 14

5 Results 15
   5.1 Test Setup ............................. 15
   5.2 Runtime Performance ................... 15
   5.3 Tuples per Second Performance ...... 17

6 Related Work 19

7 Conclusion 20
   7.1 Future Work ............................ 20
Abstract

With the advent of Big Data, databases are becoming responsible for managing increasingly larger datasets. Despite the advantage of having more data, this often results in slow query runtimes and overwhelmingly-large resulting datasets. It is often beneficial for a database system to be able to extract a smaller subset of the data. Random samples of data can be used to return results more quickly or allow a dataset to be more manageable for the end user. In this work, we discuss our query language extension for sampling operators, our algorithms to efficiently sample in a distributed setting, and the implementation of these sampling techniques on Myria, a distributed shared-nothing parallel database management system developed by the University of Washington.

1 Introduction

1.1 Overview of Myria

Myria is a distributed, shared-nothing big data management system. A public demo of the service can be found at:

demo.myria.cs.washington.edu

Figure 1: The architecture of the Myria system. Users can write interactive queries in a number of query languages. The user query is converted into a logical plan and then into a physical plan, which is sent to the master node through the REST API.
1.2 Query Languages

Almost every database system has a declarative query language that end-users interact with. Myria accepts queries in three such languages: a modified version of SQL, a variation of Datalog, and a novel language called MyriaL (Myria Language).

```
T = SELECT $0, COUNT($1) FROM SCAN(TwitterK) AS Twitter;
store(T, InDegree);
```

Figure 2: An example query in Myria’s version of SQL. This query will return the in-degree value of each node in the TwitterK dataset.

```
Triangles(x,y,z) :- TwitterK(x,y), TwitterK(y,z), TwitterK(z,x)
```

Figure 3: An example query in Datalog. This query will return every ‘triangular-cycle’ in the TwitterK dataset.

```
T1 = scan(TwitterK);
T2 = scan(TwitterK);
Joined = [from T1, T2
        where T1.$1 = T2.$0
        emit T1.$0 as src, T1.$1 as link, T2.$1 as dst];
store(Joined, TwoHopsInTwitter);
```

Figure 4: An example query in MyriaL. This query will return the second-degree-follower network in the TwitterK dataset.

In this paper, we discuss our work on extending MyriaL to express the new sampling operations.

1.3 Sampling Processes

There are a number of different sampling techniques. In this paper, we focus on unbounded-size sampling (Bernoulli & Poisson) and bounded-size sampling (Simple With-Replacement & Simple Without-Replacement) as they relate to relational databases.

Bernoulli & Poisson Sampling

In the Bernoulli sampling process, every element in the population goes through an independent weighted coin flip trial that determines whether that element will be included in the sample. Because each element is considered independently, the final sample size follows
a binomial distribution.
The Poisson sampling process is very similar to the Bernoulli process - except that a different-weight coin can be used for each element. This allows each element to have a different probability of being included in the sample. Bernoulli sampling is a special case of the Poisson sampling process in which all elements share the same probability of being selected.

**SampleWR & SampleWoR**

Sampling With-Replacement (SampleWR) and Sampling Without-Replacement (SampleWoR) are bounded-size sampling processes in which every element has an equal probability of being included in the final sample. The final resulting sample is of a fixed size. For the SampleWR case, a single element can be included multiple times in the final sample.

### 1.4 Sampling Operations in MyriaL

The new sampling functionality can be accessed by end-users through the new MyriaL functions we have implemented.

**Bernoulli & Poisson Sampling**

Bernoulli sampling can be performed by using ‘flip(p)’ anywhere an expression is allowed:

```plaintext
1. T1 = scan(TwitterFollowers);
2. T2 = [from T1 where flip(.6) emit *];
3. store(T2, SampledTwitterFollowers);
```

*Figure 5:* A MyriaL query involving Bernoulli sampling. The expression ‘flip(.6)’ will subject every tuple in T1 to a Bernoulli trial where the inclusion probability is 60%.

Poisson sampling can be performed by using ‘flip(p)’ where p is a variable or a relation’s attribute:

```plaintext
1. T1 = scan(R);
2. T2 = [from T1 where flip(R.weight) emit *];
3. store(T2, SampledR);
```

*Figure 6:* A MyriaL query involving Poisson sampling. The relation R has an attribute ‘weight’ that is real-numbered between 0.0 and 1.0

In both cases of Bernoulli and Poisson sampling, the value of p is bounded by 0.0 and 1.0. Any value less than 0.0, is treated as a 0.0 and any value above 1.0 is treated as a 1.0.
Sampling a Base Relation

MyriaL uses the ‘scan’ function to read data from a stored relation. A new ‘samplescan’ function allows the user to scan the relation with a specified number of tuples or a percentage of the total relation size. An example of a runnable query:

```plaintext
-- Sample from relation with-replacement
T1 = samplescan(TwitterFollowers, 1, WR);
T2 = samplescan(TwitterFollowers, 1.5%, WR);

-- Sample from relation without-replacement
T3 = samplescan(TwitterFollowers, 1, WoR);
T4 = samplescan(TwitterFollowers, .5%, WoR);

-- Defaults to WR sampling if no type specified
T5 = samplescan(TwitterFollowers, 1);
T6 = samplescan(TwitterFollowers, 1%);

T = unionall(T1, T2);
T = unionall(T, T3);
T = unionall(T, T4);
T = unionall(T, T5);
T = unionall(T, T6);
store(T, samplescanquery);
```

**Figure 7:** A MyriaL query demonstrating the various syntaxes available for sampling from a relation stored on disk.

Sampling an Arbitrary Expression

A user may want to sample from an intermediate result instead of a relation that is already stored on disk. To do this, we introduce another function ‘sample’ that is similar in syntax to the ‘samplescan’ function. Currently, ‘sample’ does not support sampling on a percentage of tuples.

```plaintext
T1 = scan(TwitterK);
T2 = scan(TwitterK);
CrossProduct = [from T1, T2 emit *];
Sampled = sample(CrossProduct, 500);
store(Sampled, SampledCrossProduct);
```

**Figure 8:** A MyriaL query that extracts 500 samples from the cross product of the TwitterK dataset. The user may optionally add a third argument to ‘sample’ that specifies the type of sampling to perform.
2 Relational Algebra COpiler (RACO)

2.1 Relational Algebra

Every query in a relational database is logically represented in Relational Algebra - a mathematical formulation of the operations that the database will use to output a result set. It is typically visualized as a directed graph of database operators.

![Diagram of relational algebra plan](image)

**Figure 9:** A logical relational algebra plan for the query from Figure 4. Scan is a leaf operator - it accepts no inputs from other operators. Store is a root operator - it does not output values for another operator.

2.2 Overview of RACO

RACO (Relational Algebra COpiler) is a framework for scanning, parsing, and optimizing a number of declarative query languages. It is developed and maintained by the University of Washington. Myria uses RACO as a bridge between the front-end web interface and the underlying database engine. A query in any of Myria’s query languages is converted into a relational algebra plan, transformed through a number of optimization rules, and ultimately outputted in a JSON format that the Myria database engine can deserialize into a chain of database operators.
Bernoulli & Poisson Sampling

**Figure 10:** A Poisson sampling MyriaL query converted into a relational algebra plan. Logically, ‘flip’ is viewed as a wrapper around the expression ‘random’ which returns a real-valued number between 0.0 and 1.0

**Sampling a Base Relation**

**Figure 11:** A query using samplescan. A new logical operator SampleScan (WR/WoR) represents the operation.
3 Myria Operator Implementation

3.1 Physical Query Plans

A relational algebra plan only describes the algebraic language constructs of the query. To specify the actual implementation and access methods that the database will use, the logical plan is converted into a physical query plan. During this phase, the system decides how each logical operator will be computed in the database. In the Myria database system, RACO is responsible for converting the logical plan into a more-descriptive physical plan. In a distributed database like Myria, the physical query plan has additional complexity due to the communication sometimes required between the nodes.

Figure 12: A logical query plan (left) and its associated physical query plan (right). Each ‘Fragment’ represents a portion of the query plan that can be computed locally on each worker in the database. The solid arrows between the fragments represent a communication step between the machines.
3.2 Bernoulli & Poisson Sampling

In relational algebra, the ‘Select’ operator outputs tuples that match a given predicate. Unbounded sampling is implemented by introducing ‘random’ as an expression type that the ‘Select’ operator can use.

![Query Plan Diagram]

**Figure 13:** The corresponding physical query plan for running Poisson sampling. Because each Bernoulli trial is independent, there is no communication between the workers.

3.3 New Operators

In a single-instance database engine, implementing bounded-size sampling is trivial: generate $k$ random tuple indices between 0 and $inputSize$, and return only the tuples that match one of those indices. Even without knowing $inputSize$, algorithms like Reservoir Sampling make this problem fairly straightforward.

However, we are interested in sampling from a shared-nothing parallel database where each worker holds only a fraction of the total $inputSize$, and there is no guarantee on how that data is distributed across the workers.

Suppose you had a 20-node cluster and wanted to sample 1000 tuples from relation $R$. It would be incorrect to sample $(1000/20 = 50)$ tuples from each worker. There is no guarantee that the tuples are evenly distributed or that a worker contains any data from relation $R$.

To implement bounded-size sampling, we extend Myria to support a few new physical operators for use in our distributed sampling algorithm.
3.4 Distributed Sampling Algorithm

SamplingDistribution

A major component of distributed sampling is figuring out how many tuples each worker should sample locally. In order to consistently generate an appropriate sampling distribution across the workers, the system needs to be aware of how the data is distributed. With the tuples counts from each worker, the system can decide how many tuples each worker should sample.

We introduce a new operator ‘SamplingDistribution’ that accepts the tuple count from each worker, computes the desired random distribution across the workers, and outputs the effective sample size for each worker.

Sample

Given the outputted information from SamplingDistribution, each worker can begin sampling from its local input. At this point, each worker knows its local input size and the number of samples that was assigned to it.

We introduce a new operator ‘Sample’ that accepts the

3.4 Distributed Sampling Algorithm

At a high level, the overall algorithm for bounded-size sampling on a distributed system goes as follows:

1. Each worker computes the size of its local data partition
2. A single node $M$ collects the size information from the workers
3. Node $M$ runs SamplingDistribution on the given worker info
4. Node $M$ shuffles the effective worker sample size to the respective nodes
5. Each worker receives its sample size and begins sampling the $k_i$ tuples assigned to them

3.4.1 Sampling a Base Relation

If we are sampling from a relation that is already stored on disk, we get the benefit of being able to perform step 1 quickly. Each worker performs a COUNT(*) on its local partition of the relation. When it comes time to perform the actual sampling, the worker scans the relation for the data.

3.4.2 Sampling an Arbitrary Expression

Sampling an arbitrary expression has a complication - we need to both compute the stream size and later use that same stream to perform the actual sampling operation. This requires us to materialize the expression we want to sample - this is achieved by storing it into a temporary relation. After the expression is materialized, we can perform sampling following the same procedure as sampling from a base relation.
However, if we attempt to materialize the entire expression we want to sample, we will lose a lot of performance gain from sampling. Suppose you wanted to sample just 50 tuples out of an output that is returning 2 million tuples. Having every worker materialize its entire local copy would result in a drastic drop in performance. We make two observations here:

1. If we are trying to sample a total of \( k \) tuples, SamplingDistribution will never create a distribution that results in worker \( k \) sampling more than \( k \). The implication for this is that in the above example, each worker has to materialize at most only 50 of its tuples.

2. For simple random sampling with/without replacement, there is very low probability that the distribution would result in serious skew towards a single worker. If we are willing to accept an occasional failure, we can make the assumption that each worker only needs to sample about \((\text{SamplingMultiple} \times \text{TotalSampleSize}/\text{NumWorkers})\). This makes the assumption that the data is roughly evenly distributed - increasing \( \text{SamplingMultiple} \) decreases the chances of this assumption failing, but also increases performance cost.

We introduce an operator ‘SampledTempInsert’ that stream-samples and materializes an input.
Figure 14: The corresponding physical query plan for performing samplescan. Due to the multiple steps of the distributed sampling algorithm, the logical SampleScan operator results in a series of physical operators.
4 Query Plan Optimization

Every production-level relational database makes use of query optimization. Query languages are almost always declarative - the user specifies the *data* they want to extract, but not *how* that data will be extracted. This is in contrast to a typical procedural language like Java where the user must explicitly process the data. With a declarative query language, the database engine has a lot of freedom to find the most optimal way of returning the final result to the user. In our Myria system, RACO performs query optimization through a series of transformations known as rules.

![Diagram of Myria's query optimizer](image)

**Figure 15:** An example of Myria’s query optimizer. Notice that the user’s query specified to perform the selection predicate *after* the cross product, but the optimizer moved the selections *before* the cross product. This results in a significant improvement in performance without impacting the outputted result.

4.1 Flip

Our implementation of unbounded sampling introduces the ability to use random() in predicate expressions. We introduce a new conservative optimization rule:

A predicate containing a non-deterministic condition cannot be pushed through joins.

Intuitively, this can be understood by observing that sampling the output of a join is not equivalent to sampling the input to the join.
Example: suppose you wanted to perform a cross product between relations R and S and wanted to perform a Bernoulli sample on the result of that. If you did the Bernoulli sample before the join, a single tuple rejected from R would result in size(S) tuples to be left out in the final result. This is a violation of independence for Bernoulli/Poisson sampling.

```
1 T1 = scan(TwitterFollowers);
2 T2 = scan(TwitterFollowers);
3 T3 = cross(T1, T2);
4 Result = [from T3 where flip(.5) emit *];
5 store(Result, TwitterCrossed);
```

**Figure 16:** Demonstration of the rule that preserves correctness. Unlike the example in Figure 14, the Select is not pushed before the cross product. If the user wishes to run an unbounded sample on the relations before the cross product, their query must specify that.

### 4.2 Sample

The logical Sample and Samplescan operators also need to be considered for optimization rules. For the current implementation, we let RACO use its default conservatism to prevent the new operators from being moved through joins. As part of future work, it will be worth defining a set of optimization rules for sampling.
5 Results

To measure the performance, assess the usability, and discover interesting patterns with our new sampling operators, we set up a number of tests.

5.1 Test Setup

Tests were run on Amazon EC2 instances running the Myria database system. Two setups were tested:

1. A 2-node M3.large cluster consisting of 1 master and 1 worker instance
2. A 21-node M3.large cluster consisting of 1 master and 20 worker instances

The tests used the TwitterFollowers dataset – a dataset of Twitter users and their respective followers, for the subset of users that participated in tweeting about the Higgs boson discovery in July 2012. The dataset and more information can be found at:

snap.stanford.edu/data/higgs-twitter.html

5.2 Runtime Performance

An important measure for any implementation of a sampling algorithm is how well it performs relative to just scanning the entire dataset. There is inherent overhead in extracting a sample, but we expect runtime performance to be better for sampling less than $X\%$ of the population. One of the goals in these performance tests is to discover the approximate value of $X$. 
5.2 Runtime Performance

Figure 17: Runtimes reported for running the different sampling techniques on a single-worker cluster. Note how SampleWR and SampleWoR are significantly worse than the others because of the required materialization. In a single-worker cluster, the above distributed sampling algorithm is unnecessary - we could use a trivial random sampling technique.

Figure 18: Runtimes reported for running on a 20-node cluster. We see very good scalability in terms of cluster size compared to the single-node cluster.
5.3 Tuples per Second Performance

Another interesting measure for sampling operators is Tuples per Second - the effective rate at which the final result is being produced.

Figure 19

Figure 20
Overall, we see a few trends:

- Unbounded sampling (flip) performs consistently better than bounded sampling. This is not very surprising because each tuple has an independent inclusion probability, allowing each worker to perform the entire sampling process locally.

- SampleScan is consistently better than Sample due to the fact that Sample requires the overhead of having to materialize first.

- Without-replacement sampling pays a performance penalty relative to with-replacement sampling. This is because it is computationally more expensive to generate the target indices for without-replacement sampling.
6 Related Work

Literature on database sampling is almost as old as relational databases themselves. Below is a list of works related to the topic of database sampling.

Demonstration of the Myria Big Data Management Service

Computer Science & Engineering and eScience Institute, University of Washington

SIGMOD 2014 paper from the UW eScience group that discusses the design of the Myria system.

Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches

Cormode, Garofalakis, J. Haas, Jermaine

Overview of various sampling techniques used for large datasets.

Random Sampling from Databases

Olken

Detailed overview of how to perform database sampling in a number of settings.
7 Conclusion

As the amount of data continues to grow, it is important to be able to manage this complexity. Statistically-Random sampling offers a way to reduce query runtimes, return user-readable result sets, and opens up new interesting tasks for the database - like extracting bootstrapped samples from a dataset.

On Myria, we have seen a sizable improvement in performance when sampling less than about 70% of the dataset. Our system is also very scalable to the size of the cluster - with more worker nodes, the sampling algorithm improved relative to the baseline scan.

7.1 Future Work

There are a number of future additions and improvements that will be worth considering:

- Support sampling in Myria’s other supported languages - SQL and Datalog
- Establishing optimization rules for database operators.
- Making Sample and Samplescan equivalent to the end-user by using the sample syntax for both.
- Analyze and optimize the pre-sampling process for sampling from an arbitrary expression.
- Support for more sampling techniques, such as stratified sampling.