# Extending Map-Reduce for Efficient Predicate-Based Sampling

Raman Grover [1] , Michael J. Carey [2]
*Department of Computer Science, University of California, Irvine*
{ [1]ramang, [2]mjcarey}@ics.uci.edu

***Abstract*—In this paper we address the problem of using MapReduce to sample a massive data set in order to produce a fixed-size sample whose contents satisfy a given predicate. While it is simple to express this computation using MapReduce, its default Hadoop execution is dependent on the input size and is wasteful of cluster resources. This is unfortunate, as sampling queries are fairly common (e.g., for exploratory data analysis at Facebook), and the resulting waste can significantly impact the performance of a shared cluster. To address such use cases, we present the design, implementation and evaluation of a Hadoop execution model extension that supports incremental job expansion. Under this model, a job consumes input as required and can dynamically govern its resource consumption while producing the required results. The proposed mechanism is able to support a variety of policies regarding job growth rates as they relate to cluster capacity and current load. We have implemented the mechanism in Hadoop, and we present results from an experimental performance study of different job growth policies under both single- and multi-user workloads.**

## I. Introduction

The ability to process and analyse large quantities of data has become a key factor in determining the success of data-driven businesses. As storage has become relatively cheaper, enterprises are not just interested in collecting more data, but also maintaining large histories of data. The accumulated data may scale up to petabytes. Processing the data to derive useful information or find useful patterns is a challenging task. A significant part of the challenge derives from the massive size of the data. The collected data does not come with pre-built indexes to provide efficient traversal. Further, the data collection activity is unlikely to produce pre-sorted data or provide statistical information like histograms that describe the data. Thus, there is still a long way to go before the data is partitioned appropriately, useful indexes are constructed, or the data has been filtered to retain its useful parts. Any sort of data analysis or mining technique applied over the collected dataset is therefore likely to be expensive.

It is important to recognize that not everyone wishes to view data in the same way. Consider an example dataset that captures demographic information about each individual in the country. The dataset might record each person's name, date of birth, gender, residence, profession and salary. To a statistician interested in approximating the mean salary of female professionals in the state of California, a significant part of the data is unwanted. An appropriately-sized sample representing the female working population of the state would be sufficient to arrive at the required result. Different from such an analysis, a developer may just wish to test a new query against the dataset. Running the query against the whole dataset could incur a significant cost and a long wait time until the query returns results. This cost could be avoided by working with a small subset of data. Further, testing the query against all sorts of data may require imposing additional constraints on the values of the contained fields in the data.

Sampling has been established as an effective tool in reducing the size of input data to avoid the huge cost in any subsequent processing [9], [10]. For the scenarios described above, a random fixed sized sample will not suffice as each record in the returned sample is additionally required to satisfy some user-specified predicate(s). Sampling a dataset with additional predicates as an inclusion criterion will be referred to as predicate-based sampling in this paper. Predicate-based sampling is essentially equivalent to sampling the results of the relational selection operator and was studied in [9] in context of relational databases. Predicate-based sampling can be expressed as a SQL query as follows:

SELECT *attributes* FROM *dataset*
WHERE *condition* LIMIT k

Above, $condition$ represents the set of predicate(s) and $k$ is the required sample size. While predicate-based sampling can significantly reduce subsequent processing, it is a non-trivial task for the following reasons.

1) *Absence of Indexes:* The collected data does not arrive with pre-defined indexes. Evaluating the predicate(s) on a large volume of collected data would require a sequential scan that is likely to be resource-intensive.
2) *Wide range of possible predicates:* Applications or end-users could possibly apply a wide range of predicates that may not be known priori. Choosing a partitioning method or an index can be hard, as it may not be appropriate for the predicate(s) being used when sampling.

Predicate-based sampling as a pre-processing task is a widely occurring pattern inside Facebook. The data warehouse at Facebook stores more than 15PB of data and loads more than 60TB of new data everyday [15]. A large number of applications rely on processing the large quantities of collected data. These include reporting and data visualization tools as well as complex machine learning applications. It is imperative to develop a mechanism for obtaining the desired samples in a way that is acceptable both in terms of resource consumption and response time. The work presented in this paper began at Facebook and aimed at building an efficient mechanism that

would allow obtaining samples from very large datasets with response times independent of the size of the datasets.

At Facebook, a large part of data-processing is based on Map-Reduce [5] and runs extensively on Hadoop [6], a widely used open-source implementation of Map-Reduce. In industry, a typical Map-Reduce cluster consists of hundreds, if not thousands, of commodity machines. A Map-Reduce cluster is typically shared amongst a group of end-users and multiple jobs are executed in parallel. End-users typically use a high-level language (Hive [7] in the case of Facebook) to express their data-intensive tasks in terms of queries similar in style to SQL or relational algebra. A cluster has limited resources in terms of the available cores/disks and is thus configured with an upper bound on the number of map/reduce tasks that can be run in parallel. In such a constrained setting, it is desirable for each job to execute efficiently using minimal resources.

The Hadoop Map-Reduce implementation executes all jobs under the assumption that all input must be processed for producing the desired result. Job input files are divided into smaller partitions called input splits. Processing each split requires acquiring an empty map slot and using significant disk I/O and CPU cycles. Even if a job can potentially produce it's result from a fraction of the input, Hadoop has no mechanism for curtailing its execution and finishing early. (Section II-C contains more details regarding the inefficiencies that arise from Hadoop's rigid execution model.) This inefficiency might still be acceptable when handling smaller datasets of the order of a few hundred gigabytes; for larger jobs that need to process tera- or petabytes, however, these inefficiencies may require using a dedicated cluster, which may not be feasible.

Predicate-based sampling belongs to the class of jobs that may not require processing the whole input. Since the data distribution or predicate-selectivity is not known apriori, a job does not know a priori the amount of input that would suffice to produce the desired sample size. However, these characteristics can be estimated as data begins to flow through the system at runtime. As a job learns more about the input data, it can dynamically take decisions and consume input as and when required. Further, a job should ideally consider the current load on the cluster and adapt accordingly. Running on a lightly loaded cluster, a job can afford to consume larger amounts of input as the resources would otherwise be left idle. On a more heavily loaded cluster, a job shall be cautious in controlling its intake, as consuming additional input may cause significant delays for want of resources that have long wait times.

This paper offers the following contributions;

(1) *Mechanism for Incremental Processing*: We present a Map-Reduce based execution model that allows incremental intake of input by a job and gives the job the ability to control its growth. A job that dynamically takes decisions and self-regulates its growth is referred as a *dynamic* job in the rest of the paper.

(2) *Policies for Incremental Processing*: The mechanism to incrementally consume input can be used in a multitude of ways, each having different ramifications regarding runtime characteristics. We present a set of configurable policies that dictate the growth of a dynamic job and help the user in customizing job execution as per the requirements.

(3) *Implementation over Hadoop/Hive*:We provide an implementation of the proposed execution model where Hive job can be configured as *dynamic* and executed in accordance with a configured runtime policy.

(4) *Efficient Predicate-Based Sampling*: We apply incremental processing to the task of predicate-based sampling and obtain an efficient execution where response times depend on the desired size of the sample rather than the size of the input datasets.

(5) *Experimental Evaluation of Policies*: Performance experiments are reported that evaluate the policies under different workload settings and different degrees of skew in the input data. The experiments provide insights about what policy or policies work better and under what conditions.

The remainder of the paper is organized as follows. Section II gives an overview of Map-Reduce and describes a Map-Reduce based method for predicate-based sampling. Section III introduces the concept of an *Input Provider* and presents a modified Map-Reduce model that does not have the existing inefficiencies. Section IV describes our implementation in the context of a Facebook-like Hive/Hadoop setup. Section V describes an experimental evaluation of the policies. Section VI discusses the related work. We conclude in Section VII.

## II. Predicate Based Sampling Using Map-Reduce

### A. Map Reduce Overview

Map-Reduce treats data as a list of (key,value) pairs and expresses a computation in terms of two functions: *map* and *reduce*.

$$map(k_1, v_1) \rightarrow list(k_2, v_2)$$
$$reduce(k_2, list(v_2)) \rightarrow list(k_3, v_4)$$

The *map* function, defined by the user, takes as input a key-value pair and outputs a list of intermediate key-value pairs. All intermediate values corresponding to the same intermediate key are grouped together and passed to a *reduce* function. The *reduce* function, also defined by the user, processes each key and the associated list of values and produces a list of key-value pairs that form the final output. Figure 1 shows the data flow in a Map-Reduce based execution.

Input data is loaded into a file or files in a distributed file system (DFS) wherein each file is partitioned into smaller chunks, also called input splits. A Map-Reduce computation begins with a Map phase where each input split is processed in parallel by as many map tasks as there are splits. Each input split is a list of key-value pairs. A map task applies the user-defined *map* function to each key-value pair to produce a list of output key-value pairs. The output key-value pairs from a map task are partitioned on the basis of their key. Each partition is then sent across the cluster to a remote node in the *shuffle* phase. Corresponding partitions from the map tasks are merged and sorted at their receiving nodes. For each key,

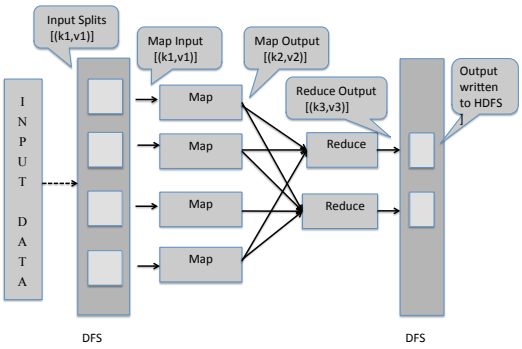



Fig. 1. Data flow in Map-Reduce

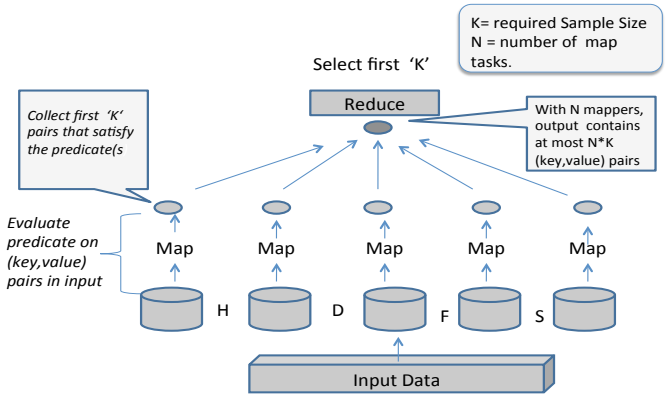



Fig. 2. A Map-Reduce based method for Predicate-Based Sampling

the associated values are grouped together to form a list. The key and the corresponding list are given to the user-specified *reduce* function. The resulting key-value pairs are written back to the DFS and form the final output.

### B. Implementing Predicate-Based Sampling

Obtaining a predicate-based sample can be done in an "embarrassingly parallel" fashion wherein the input dataset is broken down into partitions and each partition is searched in parallel for candidate records that satisfy the predicate(s). The candidate records found across the partitions are brought together and processed further to produce a random sample of the required size. It is straightforward to realize a naive Map-Reduce based method for predicate-based sampling. The map and reduce logic are shown in Algorithms 1 and 2 respectively.

**Algorithm 1** Predicate-Based Sampling: Map Logic

```
k ⇐required size for the sample
foundRecords ⇐ 0
k_dummy ⇐ dummy key
for all (key,value) pairs (k_i, v_i) in input do
  if foundRecords < k then
    if value satisfies the predicate then
      foundRecords++
      output (k_dummy, v_i) pair
    end if
  end if
end for
```

**Algorithm 2** Predicate-Based Sampling: Reduce Logic

```
k ⇐required size for the sample
{Comment: As Map phase uses a single key k_dummy, the Reduce
task receives a single [k_dummy,list(value)] pair.}
sampledRecords ⇐ list(value)
if sampledRecords.size() <= k then
  output sampledRecords
else
  Output the first k (k_dummy,value) pairs from sampledRecords
end if
```

The map phase uses a dummy key $k_{dummy}$ for each map output. The map function evaluates the predicate on the value part of each input (key,value) pair. If the predicate evaluates to true, the map task outputs a ($k_{dummy}$,value) pair. The map phase needs to output $k$ such pairs, $k$ being the required size of the sample. Since each input partition gets processed in isolation, each map task attempts to output a maximum of $k$ pairs, as it is possible that that none of the other map tasks output any desirable results. Since all map outputs share the same key ($k_{dummy}$), the lone reduce task receives a single key and a list containing the values that were found to satisfy the predicate. The list may contain a minimum of zero and a maximum of $N * k$ values, $N$ being the number of map tasks. If the size of the list exceeds $k$, the Reduce task selects the *first* $k$ [1] values from the list. Each chosen value is paired with the dummy key and forms a part of the final result.

### C. Inefficiencies in Naive Implementation

Although the method, just described can give the required sample, it is not efficient in terms of response time or resource consumption. This section describes the execution model of Hadoop and explains the inefficiencies in this scenario.

A Hadoop cluster is built from a large number of commodity machines that form a shared-nothing architecture. It has a fixed set of resources in terms of the total number of cores/disks and available bandwidth between nodes. A Hadoop cluster is pre-configured with a bound on the number of concurrent map tasks that it will attempt to execute per node. While the number of reduce slots required by a job is typically small, the number of map slots required for a Map-Reduce job is proportional to the size of the input. For a job that needs to process a massive quantity of input data, the required number of map slots can be much greater than the configured capacity of the cluster. In this case, only a small fraction of the map tasks can be scheduled to run in parallel. The remaining map tasks are placed inside a queue as they wait for the availability of map slots. In a shared environment where there are other concurrent jobs competing for map slots, the wait time for acquiring a map slot can be significant. This delays the completion of the Map phase. Since the Reduce phase cannot begin until the completion of the Map phase, the job as a whole is delayed as well.

[1] One could do a "random" k instead, to get more random results, in cases where more randomness is desired.

Irrespective of resource constraints in a cluster, Hadoop executes all jobs under an assumption that all input splits must be processed for the job to achieve its goal. The strategy has its worst effect when the assumption is not true and the input data is huge. In middle of the map phase, a job might have already processed sufficient input to form the result, but it has no mechanism to convey its success to Hadoop. Instead of continuing to the reduce phase, the job stalls, waiting for more map slots when it doesn't "need" them. When map slots become available, the job processes the remaining partitions at the cost of significant disk I/O and CPU cycles. Compared to its needs, the job does more work, consumes excess resources, and takes longer to complete. Note that the wasted resources could have served other jobs and helped them to finish earlier. Over-utilization of resources can thus have a significant impact on the system throughput.

Hadoop considers the user-defined *map* and *reduce* functions as black boxes and has a very limited understanding of the semantics of the job. A Hadoop job has no say in the execution strategy and hence possible optimizations known to the job are not made use of; the execution plan is fixed prior to job execution, leaving little or no opportunity for dynamic modifications or adaptations in accordance with the availability of resources in the cluster. By forming a shared-nothing architecture across hundreds if not thousands of machines, Hadoop surely provides a mechanism for processing petabytes of data. However, as seen in the case of 'predicate-based' sampling, its benefits can be lost if significant time and resources are wasted in doing futile work that was not truly required to compute the desired result.

## III. Towards Higher Efficiency and Lower Response Time

Given the limitations in Hadoop's execution model, it is fairly straightforward to realize that the naive Map-Reduce based method of Section II-B would require excessive resources and have significantly large response time. We need a mechanism that allows us to derive desired samples from increasingly large quantities of data (tera or petabytes) using minimal resources so that sampling jobs can be run concurrently with other jobs on a shared cluster. We propose such a mechanism in this section. The core map and reduce logic remains the same as described earlier.

In proposing an alternate execution model, it is important to identify the desirable features in the current execution model and not lose them. Hadoop's execution model has the following notable features:

- *No Synchronization Required Between Mappers:* Map tasks do not have any dependence on each other and can run in parallel without needing any sort of synchronization or message-passing between them.
- *Abstraction between Hadoop and Job Semantics:* Hadoop abstracts itself from the *map* and *reduce* logic, treating them as black boxes. This decoupling provides a generic design that does not impose any restriction as to what can go inside the Map and Reduce functions as long as they conform to the prescribed declarations.

Hadoop maintains complete knowledge of the cluster state. Cluster parameters like the capacity in terms of map/reduce slots and the current load are monitored by Hadoop. On the other hand, a Map-Reduce job understands what it wants to do with data and when it needs to process additional data, but it has no knowledge of the state of the cluster. The execution framework and the job must both somehow collaborate to take decisions based upon their combined knowledge. Sections III-A and III-B describe additional features to help Hadoop obtain a resource-efficient optimal execution.

### A. An Incremental Processing Mechanism

As described earlier, a major pitfall in Hadoop's execution model is the underlying assumption that every job will need to process its whole input to produce the required result. When this assumption is not true, the result is an inefficient execution leading to wastage of resources. Hadoop has limited understanding of the semantics of a job and hence the decisions related to the growth of a job such as to how the job wants to consume its input are best understood and taken by the job and not Hadoop. Further, the decisions may depend upon the actual input data and its characteristic properties like distribution or selectivity. As the job may not have sufficient knowledge of the data, such decisions are best taken *dynamically* as the job progresses and not statically prior to execution. As mentioned earlier, a job that dynamically controls its intake of data will be referred as a *dynamic* job in this paper.

Each *dynamic* job may have its own custom logic to assess its progress and incrementally add input when required. We introduce the concept of an *Input Provider* into Hadoop's execution model. An *Input Provider* contains the logic for making dynamic decisions regarding the intake of input by the job. The *Input Provider* is provided by the job in addition to the map and reduce logic. As stated earlier, the input to a Hadoop job is a set of DFS input partitions. A *dynamic* job begins to execute as a small job, intending to process only a subset of the input partitions. The initial set of input partitions and all subsequent increments (if required) are decided by the *Input Provider*. As the map phase progresses, the execution framework, at regular intervals of time, invokes the *Input Provider* and provides it with statistics about the output produced by finished mappers, the status of the job, the current load, and the availability of map slots in the cluster. Collection and reporting of these statistics is an existing feature in Hadoop and does not introduce any additional code or overhead.

The Input Provider can respond to the information it receives in three possible ways, as described below and summarized in Figure 3.

1) In a favorable case, the *Input Provider* may discover that the job does not need to process additional input. The *Input Provider* responds with an *"end of input"* status indicating that the job need not consume additional input partitions. The map tasks that are currently in progress are allowed to complete, but the *Input Provider* is not

invoked further and the job then proceeds to the *shuffle* phase. Thereafter, the job behaves like any other job.

2) In contrast, the *Input Provider* may realize the need to process additional input. It may respond with an *"input available"* message and provide to Hadoop the list of additional partitions to be processed next.
3) Alternatively, the *Input Provider* may decide that "wait and see" is the best decision for now. In this case, it will postpone adding input and wait until its next invocation to reassess job's progress. The *Input Provider* responds with a *"no input available"* message in this case.

The final number of map tasks for a *dynamic* job remains uncertain until the *Input Provider* responds with an *"end of input"* message. As the *reduce* phase needs to process all intermediate values for a given intermediate key, it cannot begin until all scheduled map tasks have completed. For a *dynamic* job, this requirement is necessary but not sufficient, as the *Input Provider* may add additional input. The execution framework does not begin the reduce phase until the *Input Provider* has responded with the *"end of input"* message.

The *Input Provider* for a *dynamic* job attempts to minimize the quantity of input that is processed by the job in order to obtain the desired result. It gives the job an opportunity to exploit any optimizations arising from the hidden characteristics of the input data. More importantly, the runtime characteristics are no longer just a function of the size of the input data, but can now depend on the characteristics of the data being processed. Such an execution model allows a job to work with larger datasets even in an environment that offers limited resources. At the same time, by deferring decisions to runtime, the job can control its growth in accordance to the load on the cluster and judiciously utilize resources as per their availability.

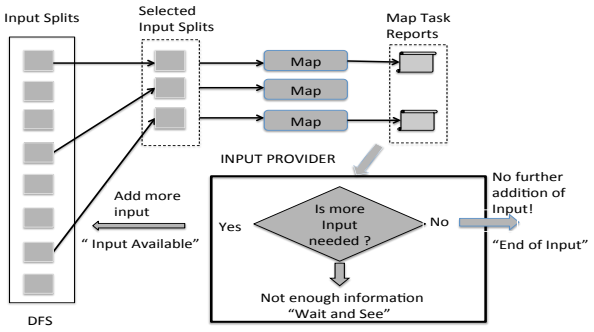



Fig. 3. Incremental processing of input

### B. A Mechanism Needs a Policy

The previous section described a mechanism based on introducing the notion of an *Input Provider* that allows a job to consume its input in an incremental manner. A job may use this mechanism in multiple ways, each having different ramifications on the runtime behaviour of the job and on the throughput of the cluster. In controlling the intake of input data, the decisions need to take into account the capacity of the cluster, the current (or expected load) on the cluster, an acceptable response time and, an acceptable cost in terms of the resources used. A job's decisions can have a significant impact on the progress of other concurrent jobs on the resulting throughput of the cluster.

A job may adopt a conservative approach, where it attempts to obtain the required result with minimal use of resources. Such a job will add minimal input at each step. At the other extreme a job could follow an aggressive approach and add all input in a single step. A low priority job may follow a conservative approach, restricting itself to small additions in input to leave resources idle for higher priority jobs. Under conditions of a heavy workload and limited availability of resources, a job may adopt a conservative approach in order to avoid adding larger amounts of input that may lead to increased contention and delays. A conservative approach may also help when much of the input data is expected to have desirable characteristics such that a small fraction will suffice to produce the required result. In constrast, under an agressive approach, the job behaves like any other job wanting to process all its input. A job may do so when its input is small or when the cluster has abundant resources that are better used than left idle. A dynamic job could also adopt an approach in between these two extremes. In order to have maximum flexibility and enable a job to use the system to its own needs, we have formulated a set of policies that govern the dynamic decisions taken by a job. A policy is defined by three parameters that are described next.

1) ***Evaluation Interval***: As described earlier, a dynamic job is configured with an *Input Provider* that evaluates the job's progress and assesses the need to add any additional input. $EvaluationInterval$ defines the time interval between each evaluation. Evaluating a job's progress and assessing its need to process additional input may incur a cost that jobs may not want to pay at short intervals. On the other hand, evaluating progress at longer time intervals may result in unnecessary waits by the job if its already-added input has been processed and the *Input Provider* needs to make a decision regarding further input.

2) ***Work Threshold***: The *Input Provider* for the job assesses the job's progress and evaluates its need to add additional input. Between successive evaluations, if a job has not done enough new work in terms of finishing new map tasks, it may not be worthwhile for the input provider to re-evaluate and make new decision. $Work\ Threshold$ puts a lower bound on the additional work to be done by a job between successive evaluations. The work done is measured in terms of the number of partitions processed, and the threshold is expressed as its percentage of the total number of input partitions for the job.

3) ***Grab Limit***: $GrabLimit$ puts an upper bound on the number of new input partitions that can be added in a single step. As each partition needs an empty map slot, $GrabLimit$ limits the additional demand to "grab" map slots that will be made by the job at each step.

The strategy adopted by a job in controlling its intake of input can be modeled by varying the parameters just described.

| Policy | Description | Work Threshold (% Total Input Size) | Grab Limit TS = *total map slots*, AS = *available map slots* |
|---|---|---|---|
| **Hadoop** | **Hadoop**'s default behaviour | - | Infinity |
| **HA** | **H**ighly **A**ggressive policy | 0 | max (0.5 * TS, AS) |
| **MA** | **M**id **A**ggressive policy | 5 | (AS != 0) ? 0.5 * AS : 0.2 * TS |
| **LA** | **L**ess **A**ggressive policy | 10 | (AS != 0) ? 0.2 * AS : 0.1 * TS |
| **C** | **C**onservative policy | 15 | 0.1 * AS |

TABLE I
POLICIES FOR INCREMENTAL PROCESSING OF INPUT

As an example, the policy adopted by Hadoop in executing a job is modeled by setting the $GrabLimit$ to infinity, signifying that there isn't an upper bound on the number of input partitions that a job will add in a single intake. Hence, a job ends up adding all its input in a single step. As all the input is added initially, the remaining parameters are not needed. We will refer to this setting as 'Hadoop' policy in the rest of the paper. The 'Hadoop' policy is at one end of the spectrum, representing an aggressive approach. The parameters can also be varied to form a less aggressive approach. We have used them to model model a set of four more policies (Table I), each differing from the others in its approach to adding incremental input. As an example, consider the MA policy. MA requires at least 5% of the total number of input partitions be processed between each successive evaluation by the *Input Provider*. At each evaluation, the *Input Provider* evaluates the $GrabLimit$ as either being one-half of the available map slots (AS) or one-fifth of the total map slots (TS) in the cluster and restricts the addition of input accordingly. The $Evaluation\ Interval$ is irrelevant for the 'Hadoop' policy, but it was fixed at 4 seconds for other policies in the experiments presented later.

## IV. HADOOP SAMPLING IMPLEMENTATION

In this section we show how to use the proposed *Input Provider* mechanism for incrementally ingesting input in Hadoop in order to produce a resource efficient execution of predicate-based sampling. The map and reduce logic remain unchanged, but the job is now configured to execute as a *dynamic* job and has an associated *Input Provider* that controls its intake of input.

Hadoop provides the notion of a JobConf as the primary interface for describing a Hadoop job. A JobConf object consists of a set of configuration parameters. We extended this set with additional parameters (listed below) that are required to characterize a map-reduce job as '*dynamic*'.

| Parameter | Description |
|---|---|
| dynamic.job | A boolean flag, set as $true$ for dynamic jobs |
| dynamic.job.policy | The name of the policy to use in controlling job's growth |
| dynamic.input.provider | An implementation of the InputProvider interface |

Job submission in Hadoop follows a client-server model wherein a JobConf instance is submitted to a *JobClient*. The submitted JobConf is sent across to a server side daemon called the *JobTracker* that manages the lifecycle of the job. Since the *Input Provider* is pluggable external logic, a buggy implementation could potentially cause significant overheads on the server side or in the worst case bring down the *JobTracker*, which is a single point of failure for the cluster. We thus chose to have the *Input Provider* exist as a client-side entity like the *JobClient*. The *Input Provider* is thus initialized on the client side by the *JobClient*. As part of its initialization, the *Input Provider* is provided with the set of input partitions that form the complete input for the job. The *JobTracker* then remains agnostic of the existence of the *InputProvider* or of the policy being used for execution.

To begin with, the *Input Provider* provides the *JobClient* with the set of input splits that form the initial input to the job. The initial input and each subsequent increment (if required) is chosen randomly with a uniform distribution from the set of un-processed input partitions. This is done to introduce randomness in the produced sample. Both the initial input and any subsequent increment (if required) is limited by the $GrabLimit$, as defined for the policy in use. As the job progresses, the JobClient, at regular intervals of time ($EvaluationInterval$), retrieves all information regarding the status of the job and the load on the cluster from the JobTracker. If the job has made sufficient progress, as required by the policy, the JobClient invokes the Input Provider and provides it with the job status and statistics that summarize the current load on the cluster. The job status includes additional statistics such as the number of records processed and the number of output tuples produced by the completed map tasks.

As described earlier in Section II-B, the map logic outputs each key-value pair found to satisfy the set of predicate(s). At each evaluation point, if the number of produced map outputs is found to exceed the required sample size, the *Input Provider* stops adding input. Otherwise the Input Provider assesses the need to add input. In evaluating the amount of additional input that would suffice, the *Input Provider* needs to take into account the "expected" output from the pending map tasks. To do so, given the number of input records processed so far and the number of matching records found among them, the *Input Provider* estimates the predicate selectivity for the input data. The *Input Provider* takes into account the expected output volume from pending map tasks based on the estimated selectivity and computes the number of additional records that will likely need to be processed in order to achieve the required sample size. Note that each split may vary in the number of records contained in them. Thus, given the splits and the total input records processed so far, the *Input Provider* computes the expected number of records in each split. Using this estimation, the Input Provider is able to arrive at the

required number of splits that need to be processed next by the job in order to reach the desired sample size.

Predicate-based sampling can be expressed as a single Map-Reduce job. In order to exploit new possible optimizations from incremental processing of input, the job needs to be configured as a *dynamic* job. As stated ealier, at Facebook, end-users use Hive as a high-level language for expressing their data-intensive tasks. We have modified the Hive compiler so that the constructed JobConf has the "dynamic.job" flag set to true and the "dynamic.input.provider" parameter set to the class name for the class that implements the *Input Provider* interface. The required size of the sample, is set in the JobConf and subsequently used in initializing the *Input Provider*. As described in Section III-B, a dynamic job executes under some policy that dictates its intake of input. The available policies are defined in a policy.xml file. As of now, the Hive syntax does not allow specifying the policy as part of the query, but Facebook may provide such support in a future Hive release [8]. Hive does allow setting of configuration parameters explicitly from the command line interface. The end-user is currently required to choose amongst the configured policies (which are listed in the policy.xml file) by setting the "dynamic.job.policy" parameter accordingly.

## V. Experimental Evaluation

So far we have described our new mechanism for incremental processing of input and showed how to apply it to the task at hand - obtaining fixed-size, predicate-based samples from an un-indexed dataset. In order to evaluate the effectiveness of the proposed method and study the performance characteristics of our various policies (Table I) under different degrees of skew in the input and under different user workloads, we conducted a set of experiments that we describe next.

### A. Experimental Setup

We ran experiments on a 10-node IBM x3650 cluster. Each node had one Intel 2.26GHz processor with four cores, 12GB of RAM, and four 300GB hard disks. Thus the test cluster consisted of a total of 40 cores and 40 disks. We used Hadoop-0.20.2 as the base version and applied the modifications to provide support for dynamic jobs. On the client side, we used Hive-0.5.0 and modified the compiler as described, to set the required parameters in the constructed JobConf.

### B. Test Data and Queries

The dataset for our experimental evaluation was derived from the LINEITEM table from the TPC-H [16] dataset. Each row in the table captures the sale of an item and includes attributes such as quantity, price, discount, tax, etc. In order to evaluate the proposed method on differently sized data, we generated LINEITEM data at scales of 5, 10, 20, 40 and 100. Table II summarizes the properties of the generated datasets. We desired a balanced distribution of load across the 40 disks and hence required the input data to be evenly distributed across the disks with no replication.

Since the input is partitioned across multiple disks, each partition could contain a varying number of records that turn out to match the predicate(s) for a given Hive query. Under a skewed distribution of matching records across the partitions, the Input Provider can make significant error(s) in estimating the selectivity. The Input Provider relies on the estimated selectivity to determine the increments in the input. In the case of an under-estimation, the Input Provider may add more than the required amount of input, thus causing delays in completion and an excessive use of resources. On the contrary, an over-estimation may produce insufficient results and require the Input Provider to add additional input many times. A uniform distribution of the matching records is favorable, but not the typical case. Obviously end-users or applications may use a wide variety of predicates when sampling. As data is collected and partitioned without prior knowledge of the predicate(s), it is likely to end up having a skewed distribution with a large fraction of the matching records occurring in just a small fraction of the partitions. It is thus imperative to evaluate the proposed mechanism and each policy under different degrees of skew in the input data. We need to produce a partitioned LINEITEM dataset such that for a given predicate, the distribution of matching records across the paritions is skewed. The method for producing such a partitioned dataset is described next.

TABLE II
Generated Datasets

| Scale | Size(GB) | Number of Rows (million) | Number Partitions on HDFS |
|---|---|---|---|
| 5 | 3.8 | 30 | 40 |
| 10 | 7.8 | 60 | 80 |
| 20 | 15.8 | 120 | 120 |
| 40 | 37.6 | 240 | 240 |
| 100 | 78.7 | 600 | 600 |

*Modeling data skew:* Suppose that we have a dataset and a predicate $p$ with selectivity $\rho$. If $T$ is the total number of records in the dataset, then the number of records that satisfy $p$ is given by $T * \rho$. To generate data, we need to assign each matching record to an input partition to contain the record. To do so with a skewed distribution across $N$ partitions, the assignment of a matching record to an input partition was treated as a random variable drawn from a Zipfian [11] distribution.

Zipf's law states that out of a population of $N$ elements, the frequency of elements of rank $k$, $f(k; z, N)$, is:

$$f(k; z, N) = \frac{\frac{1}{k^z}}{\sum_{n=1}^{N}(\frac{1}{n^z})}$$

Following the Zipfian distribution, the frequency of occurrence of an element is inversely proportional to its rank. In the current context, let:

1) $N$ = total number of input partitions;
2) $k$ be their rank; partitions are ranked as per the number of records in the partition that satisfy the given predicate;
3) $z$ be the value of the exponent characterizing the distribution.

For every matching record, we draw its containing input partition from the described Zipfian, thus resulting in a skew.

TABLE III
PREDICATE AND ASSOCIATED DISTRIBUTION

| Column | Predicate | Distribution |
|---|---|---|
| ExtendedPrice | $ExtendedPrice < 1000$ | Uniform ($z$ = 0) |
| Discount | $Discount > 0.01$ | Moderate Skew ($z$ = 1) |
| Tax | $Tax > 0.05$ | High Skew ($z$ = 2) |

We varied the Zipfian parameter $z$, setting it to be 0, 1, and 2, thus giving a zero, moderate and high skew, respectively.

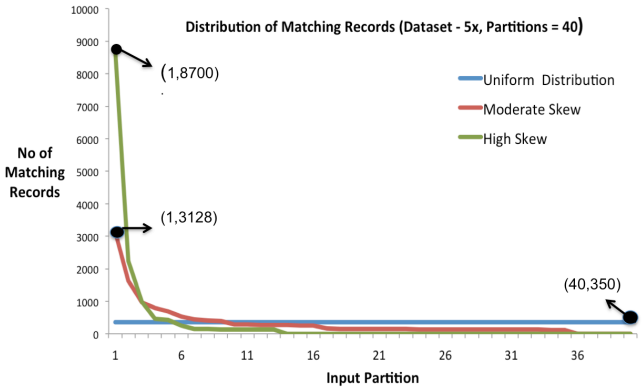



Fig. 4. Distribution of Matching Records across Input Partitions, shown for 5x Input Data

As stated earlier, the dataset for our experiments is the LINEITEM table. Corresponding to each degree of skew ($z$ = 0, 1, 2), we chose an arbitrary column and formed a corresponding predicate. Table 3 summarizes the set of predicates and the associated skew in the distribution of matching records across the input partitions. The overall selectivity of the dataset to each predicate was fixed at 0.05%, while skew was varied in the manner just described.

Having generated the distribution of matching records across the partitions, we then modified the other records in each partition accordingly to ensure that the remaining records contained random values not satisfying the predicate. Figure 4 shows the distribution of matching records across partitions, for each degree of skew ($z$ = 0, 1, 2), for 5x (scale) input data. Recall from Table II, that 5x input gets partitioned into 40 partitions when stored in HDFS. With 5x input and a predicate selectivity of 0.05%, we have 15,000 matching records out of total of 30 million records. As shown in Figure 4, with zero skew, we get an equal number of matching records (350) in each partition. For higher degrees of skew, we have a fewer partitions that contain a larger fraction of the matching records. When $z$ = 2, we have 8700 of the 15,000 mtaching records being contained in a single partition. In the case of $z$ = 1, we have 3128 matching records being contained in a single partition. Similar distributions were generated for each scale of input data (10, 20, 40 and 100).

As stated earlier, predicate-based sampling from a dataset can be expressed as a single Map-Reduce job. The runtime characteristics and performance of the Hadoop job will depend on a number of factors. These include the size of the dataset, the policy in use, the selectivity of the predicate, the physical distribution of the matching records within the dataset, and the load on the cluster. In order to understand the role of each factor, we divide our experiments into three sections. In our first experiment, we studied single-user performance as a function of the input dataset size and skewness in data. Next, we evaluated performance under a homogeneous multi-user workload followed by an evaluation under a heterogeneous workload. For generating workloads, we used a workload generator [2] that allowed us to model a group of end-users that work in parallel using Hive.

We used a fixed sample size of 10,000 for all experiments. The Hive query for each job followed the template:-

SELECT ORDERKEY, PARTKEY, SUPPKEY
FROM LINEITEM
WHERE *predicate* LIMIT 10000

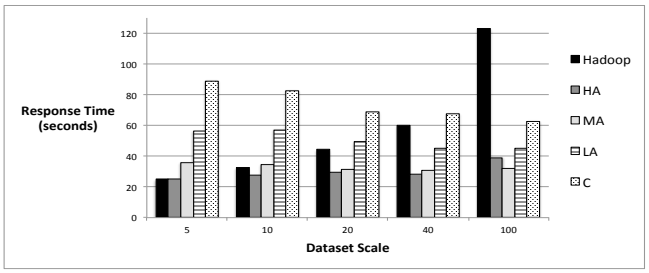
For varying the degree of skew in the distribution of the matching records, we used the corresponding *predicate* listed in Table III.

### C. Single-User Workload

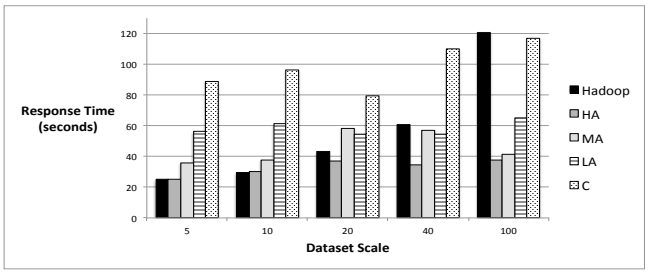
In order to study the role of the dataset size and its associated skew, we began by evaluating the performance characteristics of the various policies on a cluster with no other concurrent jobs. Each node in the cluster was configured to support four concurrent map tasks. With no other jobs competing for map slots, a Hadoop job could execute a total of 40 map tasks in parallel. We experimented with five differently sized datasets, three different degrees of skew, and the five different policies from Table I, giving a total of seventy-five different combinations. Figure 5 summarizes the job response time results for each such combination. In Figure 5, graphs (a), (b) and (c) correspond to the case of zero, moderate, and high skew, respectively, and provide a comparison between the response times with each policy for different dataset sizes. Figure 5 (d) provides insight into the amount of work done by the jobs by showing the number of partitions processed per job under each policy for the case of moderate skew. All numbers are averages taken over 5 runs. Let us see what we can learn from these initial results.

1) *Response time under the default Hadoop policy:* Under the Hadoop policy, the response time is not influenced by the physical distribution (skew) of the matching records in the dataset. This is as expected, as under the Hadoop policy, all input partitions are required to be processed; there is no opportunity to partially process the input data to collect just the required number of matching records. The completion time for the job is largely determined from the time consumed in the Map phase, which takes as much time as is required to apply the map function to every input partition. The Hadoop policy's response time is thus governed by the size and the total number of input partitions. Thus response time can be seen to increase with the input data size. As described earlier, this behavior makes the execution model of Hadoop poor when sampling large datasets.

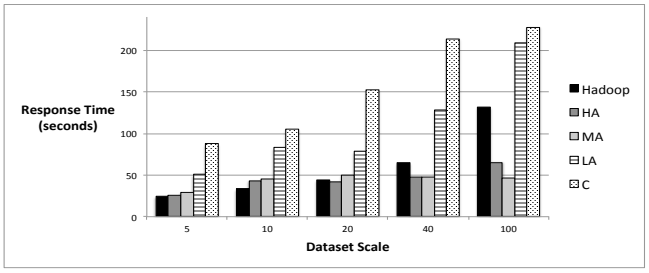
2) *Effect of data skew:* Recall that the policy in use puts an upper bound on the amount of (additional) input that is added in each step. This limit is highest for 'Hadoop' which adds all



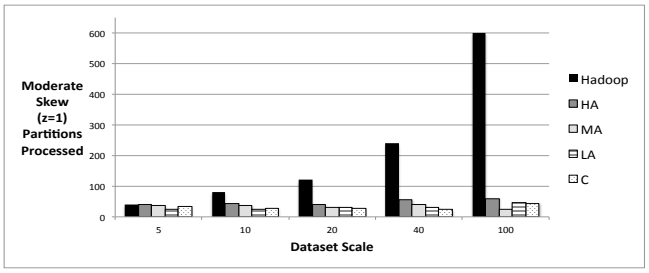

(a) Uniform Distribution




(b) Moderate Skew




(c) High Skew




(d) Moderate Skew: Partitions Processed

Fig. 5. (a), (b) and (c) Response Time (seconds) under each policy for Different Dataset Sizes and different degrees of skew. (d) Work done in terms of partitions processed in the case of moderate skew.

input up front, and it decreases across HA, MA, LA and C in that order. Under C, the job acts in a conservative manner and takes input in small increments. Since the cluster is not being shared, under 'C', the job is unable to use the cluster to its full capacity, leaving available map slots idle. Conservatism has the worst effect when the physical distribution of the matching records is highly skewed. This is because, under a high degree of skew, a large number of input partitions will yield no results. Since a conservative approach processes less data in each step, it can require many steps to collect the desired number of matching records. In contrast, the aggressive approach adopted by HA utilizes the cluster to its maximum capacity by adding enough incremental input to use all available map slots. By adding larger amounts of input, it is able to overcome the effect of increased skew, as by processing larger amount of data in each step, the probability of discovering matching records increases. This is why HA and MA generally perform better than the other policies on the otherwise idle cluster.

3) *Using the idle cluster "wisely"*: On an idle cluster, a more aggressive approach to utilizing resources provides better response times. As a policy becomes less aggressive, idle resources in the cluster are left unexploited so that other potential jobs, arriving in the future, can make use of them. Since the cluster is not being shared, the conservative approaches do not use the cluster to its maximum capacity. Response time therefore worsens as one moves towards less agressive policies such as LA and C. The Hadoop policy adopts the most aggressive approach of all, but it is unable to outperform HA. This is because the Hadoop policy demands resources in excess of the cluster's maximum capacity. As shown in Figure 5 (d), the number of partitions processed (and thus the resources consumed) under the Hadoop policy is much higher as compared with the other policies.

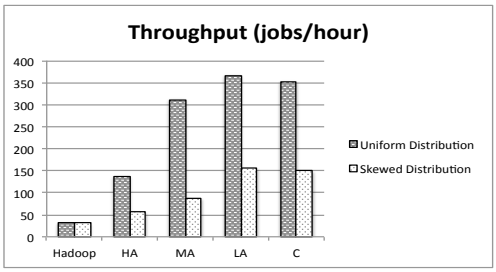
In production, Hadoop clusters are almost never used in an exclusive mode with no other concurrent jobs. Performance is thus better evaluated in a more practical scenario where the resources offered by the cluster are shared amongst a group of users.

### D. Homogeneous Multiuser Workload

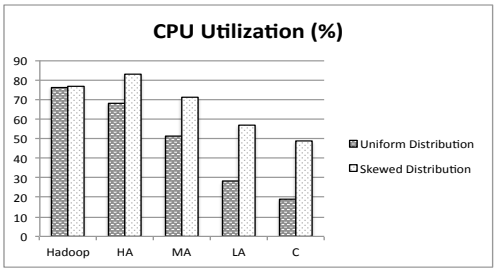
Our next set of experiments are based on multiuser workload settings and evaluate the impact of the policies on the throughput of the cluster for these workloads. In our multiuser workload setting, the cluster capacity parameters are increased to have 16 map slots per node as against the 4 used in our single-user experiment. The number 16 was arrived at by trying different settings with the objective of achieving maximum throughput. We modeled a group of 10 concurrent users where each user submits a query and waits for its completion before submitting another query (the same query again). Each of the ten users submit the same query, but each works against a different copy of the dataset (scale=100x) to ensure that each query requires fetching its input from the disk and does not leverage the buffer cache populated by some other query. In a given workload run, all jobs used the same *Input Provider* and policy. Each workload was run for a sufficiently long duration to obtain steady state throughput.

In our multiuser experiment, we evaluated the overall throughput for the cluster under each policy. In addition, we monitored the CPU utilization (%) and disk reads (Kbs/sec) at 30 second intervals on each node of the cluster. We first evaluated performance under a uniform distribution of matching records and then repeated the experiment for the case of a highly skewed distribution ($z$ = 2). The results are summarized in Figure 6, where we show for each of our policies (from Table I), the observed throughput (jobs/hour) and the corresponding resource usage in terms of CPU utilization and disk reads. The CPU utilization and disk read results are

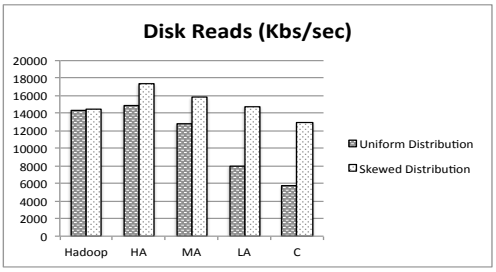
averaged over the 40 cores and 40 disks that constitute the cluster. Note that the policies are shown along the x-axis in the figure. Following are the notable observations:




(a) Throughput (jobs/hour)




(b) CPU Utilization




(c) Disk Reads

Fig. 6. Performance characteristics under homogeneous workload (a) Observed throughput (jobs/hour) under each policy, (b) and (c): Resource usage (% CPU utilization and disk reads(Kbs/sec))

1) *Performance characteristics using the default Hadoop policy:* The 'Hadoop' policy follows the most aggressive approach in consuming input and gives the least throughput. As the Hadoop policy requires processing all of the input, the demand for map slots by a single job well exceeds the capacity of the cluster. Jobs spend a significant time waiting for map slots to become available, thus delaying completion and decreasing the throughput, and they also use excess resources to process the entirety of the input. It is worth noting that the average CPU utilization (%) and Disk Reads (Kbs/sec) are highest under Hadoop despite the low throughput under the same policy. This confirms an inefficient execution with significantly greater amount of work being done per job.

2) *Uniform distribution:* With a uniform distribution of the matching records across the partitions, the Input Provider is able to form a good estimation of the predicate selectivity from processing of just a fraction of the input. The total number of partitions that need to be processed to form the required sample can thus be well predicted at an early stage from the estimated selectivity. However subsequent increments to the input are limited as per the $GrabLimit$ defined by each policy. Under a very aggressive policy, the job has a higher $GrabLimit$ and may add more input in the beginning than what is required. This leads to increased waste and contention for resources, as a job processes more than the required amount of input. A more conservative policy with a smaller $GrabLimit$ adds less input at each step and thus avoids the over-addition of input. As a result, the throughput in Figure 6 increases as a policy becomes less aggressive along HA, MA, LA as $GrabLimit$ decreases. C is slightly worse due to being more conservative than needed.

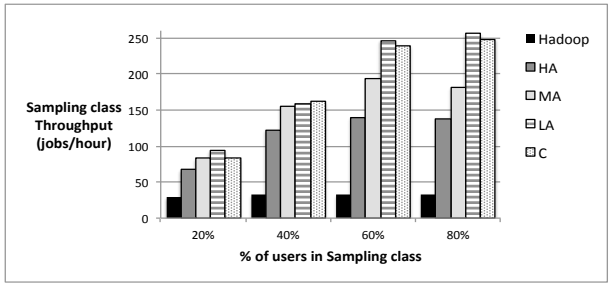
3) *Highly skewed distribution:* When the distribution of matching records across the input partitions is skewed, a large number of partitions may not contribute any records that satisfy the predicate(s). A significant amount of resources and time is expended in processing these partitions. This results in higher CPU utilization, increased disk reads, and a lower throughput in comparison to the case of a uniform distribution. As expected, the Hadoop policy throughput is not affected by the presence of skew in the input.

### *E. Multiuser Heterogeneous Workload*

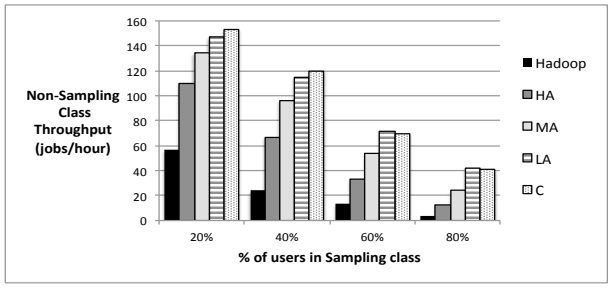
In practice, users have different needs and hence the workload on a production cluster is expected to be heterogeneous in terms of the kinds of jobs running concurrently. At Facebook, for example it is typical to have a fraction of the end-users interested in obtaining predicate-based samples while the rest are executing other kinds of jobs. The runtime policy used by the users in obtaining the desired samples can impact the response time for other kinds of concurrent jobs. As a step towards understanding the effect of the policy in use, we categorized users in our next workload into two classes: Sampling and Non-Sampling. The predicate used for sampling jobs corresponds to a uniform distribution of the matching records across the partitions. Non-Sampling users submit basic $select\text{-}project$ queries with a selectivity of 0.05%. The dataset used was again a copy of the LINEITEM table (100x).

As resources are shared amongst the jobs, one can intuitively argue that a resource-efficient execution strategy for sampling jobs should positively impact the progress of the other class of concurrent jobs. The extent of impact will depend upon the fraction of jobs belonging to the Sampling class and the policy that controls their execution. We varied the fraction from 0.2 to 0.8. The results are shown in Figure 7, where we show the per-class throughput for each class of jobs, measured with each of the policies C, LA, MA, HA and Hadoop used by the jobs belonging to the Sampling class. Key observations include the following:-

1) *Sampling class throughput*: As shown in Figure 7 (a), the Sampling class throughput increases as the fraction of users in Sampling class increases, which is as expected. For a given fraction of users in the Sampling class, the throughput results show similar trends (as policy is varied) as were observed in our earlier homogeneous workload experiment where all users




(a) Sampling Class: Throughput (jobs/hour)




(b) Non-Sampling Class: Throughput (jobs/hour)

Fig. 7. Heterogeneous Workload: using Default Scheduler (a), (b)

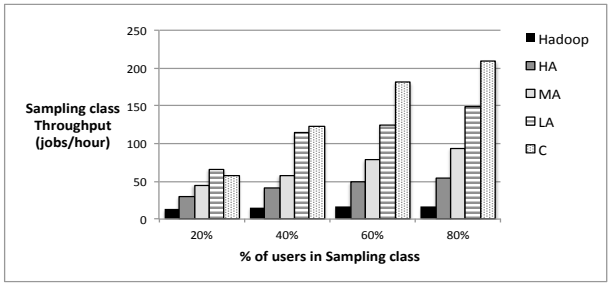
belonged to the Sampling class.

2) *Non-Sampling class throughput*: As shown in Figure 7 (b), the throughput for Non-Sampling class is definitely influenced by the policy adopted by the *dynamic* jobs in 'Sampling' class. The throughput for Non-Sampling class is least when the Sampling class follows the Hadoop policy. This is expected, as the under the Hadoop policy, Sampling class jobs execute in an inefficient manner, causing greater contention for map slots and consuming an excess of resources. A shift to a less aggressive policy by the Sampling class provides a boost to the throughput of the Non-Sampling class. With a small fraction (20%) of users in Sampling class, the use of policy LA as against Hadoop raises the throughput for Non-Sampling class by a factor of 3. This factor increases to as high as 8 when a significant fraction (80%) of users belong to the Sampling class. For either class of users, a conservative approach (C/LA) adopted by the *dynamic* Sampling jobs gives the maximum throughput as against an aggressive approach (HA/Hadoop).

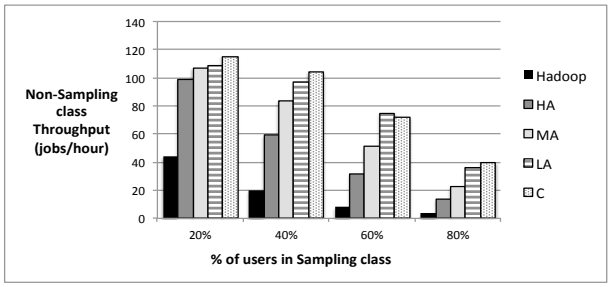
### *F. Scheduler Impact*

In a shared environment, the strategy used by the framework in allocating the available map/reduce slots and distributing the available resources across jobs is expected to have a significant role in determining the throughput of the system. While the scheduling policy works towards improving a throughput by optimum allocation of resources, a *dynamic* job plays a helping hand by limiting the demand for map slots. In Hadoop, the task of assigning empty slots to the pending tasks is handled by the *TaskScheduler*. The default implementation provided by Hadoop is based on FIFO, with slots being assigned in order of a job's submission timestamp. One of the prominently used alternate scheduler implementations is the Fair Scheduler [17] developed by researchers at U.C Berkeley and Facebook. In order to check the invariance of our conclusions to the scheduling policy, we repeated our last experiment using the FairScheduler as against the default scheduler provided in Hadoop. The results are summarized in Figure 8.

Similar to the results obtained using the default scheduler, the throughput for either class is increased when a conservative approach is adopted by the Sampling class jobs. Comparing the results shown in Figures 7 and 8, it is worth noting that, for either class of users, the overall throughput achieved falls on switching to the Fair Scheduler. In order to understand this behaviour, we categorized the map tasks as local (those that read input from the local disk) and non-local (those that read input from a disk on a remote node) and measured 'locality' as the % of map tasks that are 'local'. In addition, we measured 'slot occupancy' as the % of map slots occupied at any time. What we found is that Fair Scheduler achieved a locality of 88% and a 'slot occupancy' of 18%. The corresponding numbers for the default scheduler were 57% and 44%. Higher 'locality' is desirable to avoid reading across the network, but achieving higher 'locality' may require the scheduler to wait until an empty map slot is available at a node that has an input partition on its local disk, thus producing delays in execution. Regardless we see again that the Input Provider helps and that a fairly conservative policy is best when system is loaded.




(a) Sampling Class: Throughput (jobs/hour)




(b) Non-Sampling Class: Throughput (jobs/hour)

Fig. 8. Heterogeneous Workload: Using Fair Scheduler (a), (b)

## VI. Related Work

As stated earlier, predicate-based sampling is synonymous with sampling the results of the relational selection operator. Sampling in the context of relational databases was a topic of interest in the early 1990's. Olken described methods for sampling the results of selection operator in context of relational

tables [9]. The described methods relied on indices for efficient traversal of the relation being sampled. Lipton et al. [12] used sampling as an effective method for estimating predicate selectivity. The method described could handle complex selection predicates such as arithmetic expressions, but required an index on some column of the relation (that need not be the column to which selection is applied). Effective methods for producing random samples from databases using B+ trees were also described in [1], [9] and [14]. Not surprisingly, there is a common theme here; the proposed methods for sampling all assume that the data has been loaded into relations and that indexes are available. In contrast, here we have studied sampling when the data has no indices and resides in a filesystem instead of a database. The work presented here simultaneously estimates predicate selectivity as well as forms a sample containing the results of the selection operator.

In the context of Map-Reduce, high-level languages such as Hive [7] and Pig [13] allow users to execute queries against a sample of the data instead of the whole dataset. This is not sufficient for obtaining a fixed-size, predicate-based sample as the predicate evaluated over the sample may produce an empty result. A possible technique to ensure non-empty output is to form a biased sample as suggested in [4], but that method also relies on indexes created in advance. Finally, in [3], Babu discussed methods to optimize Map-Reduce jobs and discussed the need for developing an efficient sampling harness that could be used to build partial statistics for input and intermediate data in a Map-Reduce pipeline. Our proposed mechanism for incremental processing and its implementation could be used as a seed towards building such a sampling harness.

## VII. Conclusion and Future Work

While predicate-based sampling can easily be expressed today as a Map-Reduce job, the resulting inefficiencies in the Hadoop execution model results in unacceptable response time and resource usage. This is because the basic Hadoop execution model assumes the need to process all of the input data, and hence its runtime characteristics (response time/resource usage) depend on the overall size of the input. This paper presented a modified execution model and an implementation that supports incremental processing of input, wherein a job is fed with input data as and when required by the job. An *Input Provider* associated with the job is provided with information related to its progress and the existing load on the cluster. This helps the job to adapt its growth in accordance with the availability of resources in the cluster.

In order to provide flexibility in using the mechanism for incremental processing, we formulated a set of policies that govern the growth of a job and its dynamic decisions related to its intake of input. An end-user can also choose to form new policies and choose a policy on a per-job basis. We modeled Hadoop's execution strategy as a policy (Hadoop) and formed a set of additional policies (C, LA, MA, HA) that differed in the criteria used in adding input to a job. We modeled different degrees of skew in the input data and provided experimental results under different workload conditions.

Experiments on an idle cluster with no concurrent jobs showed that a conservative policy such as C does not utilize the cluster to its capacity and results in a higher response time. In contrast, in a more realistic multi-user setting, a conservative policy was seen to consume the least resources and produced the highest throughput. Our LA policy emerged as a good overall policy to use in both homogeneous and heterogeneous workload settings. In all cases, the Hadoop policy resulted in the least throughput with maximum utilization of resources. Hadoop's execution model (or policy) was confirmed to be very wasteful for obtaining predicate-based samples from large datasets.

In our implementation, the policy for execution is chosen statically, prior to job execution. As part of future work, it could be interesting to implement a more flexible model wherein a job could decide and change the policy at runtime, based on the discovered characteristics of the input data together with the existing load on the cluster.

The work presented in this paper is in process of being integrated into the Hive/Hadoop system at Facebook.

## VIII. Acknowledgements

This project is supported by NSF IIS award 0910989 and a grant from the UC Discovery program. We would like to thank Joydeep Sen Sharma from Facebook (where much of the engineering work presented here was performed) for useful guidance and feedback. We would also like to thank UCI MS alumnus Vandana Ayyalasomayajula for providing us with the workload-generator used in our experiments.

## References

[1] G. Antoshenkov. Random sampling from pseudo- ranked B+ trees. In *Proc*. VLDB 2002, pp. 375-382
[2] V. Ayyalasomayajula. HERDER: A Heterogeneous Engine for Running Data-Intensive Experiments and Reports. M.S Thesis. University of California-Irvine, 2011
[3] S. Babu. Towards automatic optimization of mapreduce programs. In *Proc*. SoCC 2010 pp. 137-142.
[4] S. Chaudhuri, R. Motwani, and V. Narasayya. On random sampling over joins. In *Proc*. SIGMOD 1999, pp. 263274.
[5] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc*. OSDI 2004, pp. 137-150.
[6] Hadoop. http://hadoop.apache.org.
[7] Hive Website. http://hadoop.apache.org/hive
[8] https://issues.apache.org/jira/browse/HIVE-2004
[9] F. Olken. Random Sampling from Databases. PhD thesis, University of California Berkeley, 1993
[10] F. Olken and D. Rotem. Simple Random Sampling from Relational Databases. In *Proc*. VLDB 1986, pp. 160-169.
[11] D. E. Knuth. The Art of Computer Programming, Vol 3: Sorting and Searching. Addison-Wesley, Reading, MA, 1973.
[12] R. J. Lipton, J. F. Naughton, D. A. Schneider, and S. Seshadri. Efficient sampling strategies for relational database operations. *Theoretical Computer Science* 1993: pp. 195- 226.
[13] C. Olston, B. Reed et.al. PigLatin: A not-so-foreign language for data processing. In *Proc.* SIGMOD 2008, pp 1099-1110.
[14] F. Olken and D. Rotem. Sampling from B+ trees. In *Proc* VLDB 1989, pp. 269-277.
[15] A. Thusoo, Z. Shao et al. Data warehousing and analytics infrastructure at Facebook. In *Proc.* SIGMOD 2010, pp. 1013-1020.
[16] TPC-H *Benchmark Specification* http://www.tpc.org
[17] M. Zaharia, D. Borthakur et al. Job Scheduling for Multi-User MapReduce Clusters. UC Berkeley Technical Report UCB/EECS-2009.