Balancing Reducer Skew in MapReduce Workloads using Progressive Sampling

Smriti R Ramakrishnan
Oracle Corporation
smriti.ramakrishnan@oracle.com

Garret Swart
Oracle Corporation
garret.swart@oracle.com

Aleksey Urmanov
Oracle Corporation
aleksey.urmanov@oracle.com

ABSTRACT

The elapsed time of a parallel job depends on the completion time of its longest running constituent. We present a static load balancing algorithm that distributes work evenly across the reducers in a MapReduce job resulting in significant elapsed time reductions.

Taking a user-specified model of reducer performance, our load balancer uses a progressive objective-based cluster sampler to estimate the load associated with each reduce-key. It balances the workload using *Key Chopping*, to split keys with large loads into sub-keys that can be assigned to different distributive reducers, and *Key Packing*, to assign keys with medium loads to reducers to minimize the maximum reducer load. Keys with small loads are hashed as they have little effect on the balance. This repeats until the user specified balancing objective and confidence level are achieved.

The sampler and load balancer have been implemented in the Oracle Loader for Hadoop (OLH), a commercial MapReduce application that employs Apache Hadoop to perform parallel data formatting and data movement into partitioned relational tables. We present the performance improvements we achieve in both OLH and in a MapReduce program for inverted index creation. The balancer works for arbitrary IID key distributions, the time used for sampling is small and our solution is very effective at reducing the elapsed time for the MapReduce jobs we explored.

Categories and Subject Descriptors
H.2.4 [**Database Management**]: Systems – *parallel databases*

General Terms
Algorithms, Measurement, Performance, Design, Theory

Keywords
Progressive Sampling, Load Balancing, MapReduce, Skew, Oracle Loader for Hadoop, Hadoop, Load model

1. INTRODUCTION

Good parallel program scalability relies on balancing the workload evenly across all available computational resources. We introduce a system we have developed to distribute work evenly among reduce tasks within a MapReduce job.

MapReduce [8] is a parallel programming paradigm where programmers specify *map* and *reduce* functions. The map function processes input key/value pairs (interchangeably called *records*), and generates an intermediate set of key/value pairs, which are processed by the *reduce* function that generates an output set of key/value pairs. The map function is applied to every input record and the reduce function is run once for each distinct intermediate key, hereafter called *reduce-key*, processing all values associated with that key in the map output (see Figure 1).

In typical parallel implementations of the MapReduce paradigm [40], a partitioning function is specified which partitions the mapper output records among the specified number of reducers. Apache Hadoop [1] is a software framework that enables writing scalable applications that process massive amounts of data in parallel on large clusters [40]. Hadoop includes an implementation of the MapReduce programming model [8].

The elapsed time of a MapReduce job is determined by the elapsed time of its map and reduce tasks. A single slow task can dominate a job's execution time; the stragglers problem [19]. Such delays can multiply in a chained job with sequential MapReduce steps, each stage adding its own delay as it waits for its last task to complete.

Skew in the runtime of a parallel task can be classified as data skew, computational skew, or machine skew [16]. Data skew is caused by the framework assigning tasks different amounts of data. Computational skew results from algorithms whose runtime depends on the content of the input rather than its size. Machine skew is caused by tasks being assigned non-uniform computing resources.

Skew in map task runtime is mitigated on Hadoop by the dynamic assignment of programmer-defined units of input data, called InputSplits, to map tasks. A map task that completes an InputSplit quickly and will be assigned another. Machine skew is mitigated on Hadoop through the use of virtual processors, where more powerful nodes support more "task slots" and speculative execution, where the same work may be assigned to more than one task and the output of the first task to finish is selected [40].

In this paper we address the problem of reducer data skew, that is, the assignment of intermediate records to reduce tasks so as to equalize their running time. Reducer data skew is of interest as reduce tasks can account for a significant portion of a job's elapsed time.

In Hadoop, the load on a reduce task comes from: (1) transferring its assigned mapper output records from each of the producing map tasks, (2) merging these mapper outputs to form the reducer input, (3) running the reduce function, and (4) writing the reducer output to the target data service. Activity (1) and some of (2) can be overlapped with map task execution but activities (3) and (4) can only start after all map tasks complete; forming a barrier between the map and reduce phases In addition activities (3) and (4) often operate serially within each reduce task. Data skew affects all four activities, not just the user-specified reduce function.

Reducer data skew can occur intrinsically if the mapper output contains reduce-keys with too many associated values, or by unlucky partitioning, if too many reduce-keys are assigned to the same reduce task. For instance, Hadoop's default partitioning function hashes the reduce-key to a reduce task. The default hashing partitioner works well when each key represents a very small portion of the overall workload. However it is not uncommon for the mapper output to have some reduce-keys with large loads and hashing these keys is likely to result in overloaded reduce tasks (see Figure 7). Furthermore, hashing does not work as a partitioning strategy in applications like sorting, which require range partitioning.

1.1 Use Cases

Consider the Hadoop implementation of the Terasort benchmark [26]; an application dominated by reduce tasks, where giving each reduce task the same number of records is important for good performance. For this reason, Hadoop's Terasort implementation contains an application-specific sampler that collects a user specified number of samples that it uses to generate a partitioning scheme. This scheme splits the key domain into intervals with roughly equal number of records and assigns these intervals to reduce tasks. This Terasort sampler is simple because Terasort uses an identity function as its mapper, so the mapper's output is the same as its input.

The Oracle Loader for Hadoop (OLH) is a commercial product [25] that runs a Hadoop MapReduce job to format, partition and potentially load data into an Oracle database table. OLH is described in Section 7.2. The OLH mapper formats each record and determines the Oracle database table partition it is assigned to. The OLH reducer writes its input to either an output file or it opens a connection to the target database and streams the data into the target table partition. Each OLH reduce task induces load both on the system where it is running and the external services it uses. Balancing the load assigned to reducers is important and beneficial for both the Hadoop system where the reducer is running and the database service the reducer is using.

Inverted Indexing is the quintessential MapReduce application, where the mappers parse a corpus of documents and the reducers create compressed indexes that record the position of each word in the corpus. Inverted indexes are commonly used to speed up search queries on large data collections e.g to index words in a document collection, sequences in a database of proteins, or signatures in a biometrics database. Most words appear very infrequently and can be hashed to an arbitrary reducer while some words appear frequently and must be chopped and partitioned carefully if we are to balance the reducer load (see section 7.3).

1.2 Contributions

We present an approach to reduce the runtime of MapReduce jobs by distributing the reducer workload evenly among the reduce tasks. Our approach requires little user interaction or user knowledge of statistics or sampling, is applicable to a wide variety of key distributions and applications, and quite effective in reducing both reducer data skew and the elapsed time needed for a MapReduce job.

The techniques we use to accomplish this goal are: (i) Statistical modeling of the reducer workload distribution from a set of samples, (ii) Progressive sampling which uses the model to determine whether we have taken enough samples to meet the user specified load balancing objective, (iii) Large key chopping, when allowed, to split reduce-keys whose load is too large to assign to any one reducer into several medium load reduce-keys using range- or hash-partitioning (iv) Medium key packing to assign medium load reduce-keys to reducers to minimize the maximum reducer load (v) Small key summarization to limit the size of the sample and the partitioning function, by tracking and packing only reduce-keys that have significant load and hashing the rest.

Subdividing keys with large record sets to balance skew is an old idea and has been used to parallelize sorts [2], joins [6, 7, 28], aggregates [21], and more recently to handle skew in specific MapReduce applications [16, 31] (see Related Work for details). This chopping of keys depends on reduce function being distributive, which is implicit in parallel MapReduce implementations of several algorithms, including sorting, because the outputs from different reducers are not physically concatenated, as we discuss in Section 3.5.1. Previous work on parallel joins relates the sample size needed to the quality of the chopping [2, 32]. Our work relates the sample size to the resulting quality of the balancing after the chopping and balancing is performed.

1.3 Results

We have implemented our algorithm on top of the open-source Apache Hadoop framework [1]. The load balancer is shipped as part of the Oracle Loader for Hadoop (OLH) product discussed earlier. We have also used this algorithm to perform load balancing in a variety of Hadoop programs with little change to the programs. We show test cases where the reducer loads are reduced by double digits, but reduce tasks aren't the only contributor to the elapsed time of a MapReduce job, and we observed elapsed time improvements between 20 and 400%. Sampling and load balancing overhead was negligible in all cases.

The rest of the paper is organized as follows: Section 2 characterizes the problem statement, Section 3 and 4 describe the algorithm, and Section 5 derives the stopping condition for the progressive sampler for general load models. Section 6 illustrates a linear load model, followed by results, related work and conclusions in Sections 7–10.

2. OVERVIEW

2.1 Load Balancing Goal

For a given MapReduce job, our goal is to generate a load balancing plan where it is highly probable that none of the R reducers is overloaded. That is, with probability greater than α, no reducer has actual load greater than (1+ δ) times its expected load. Assuming reducer loads are independent, it is sufficient that each reducer's probability of overload be greater than α1/R. Denoting the load on reducer r as a random variable RLr, we ensure that:

$$\forall r, \Pr(RL_r \leq (1+\delta)E(RL_r)) \geq \alpha^{1/R} \quad (1)$$

2.2 Characterizing reducer load

In this work we characterize the load on a reduce task based on the amount of data it processes, not the particular values present in that data. More specifically, the reducer load is modeled using: the number of reduce-keys it processes, the number of records it processes and the number of bytes needed to represent these keys and records; not on the content of the records (see Equation 7 for an example). For the use cases we consider: OLH, Sorting, and Inverted Index, the computational skew is negligible. In cases where computational skew is important, it is likely that data skew is also present, because the data must be accessed and merged by the reduce task before it invokes the user's potentially content-sensitive reduce function.

Reducer input statistics are determined by the job's input data and its map function, a combination that we are unlikely to have seen before, especially when running job chains, so instead of attempting to reuse old statistics we focus on computing new statistics efficiently.

We model the load on reducer *r* as the sum of the loads induced by the set *K*(*r*) of keys assigned to reducer *r*. We denote the load induced on the reduce task by a key *k* as L_k and estimate its expected value and variance from sampled statistics using a user-specified load model. Assuming the key loads are independent, we calculate:

$$RL_r = \sum_{k \in K(r)} L_k \tag{2}$$

$$\mathrm{E}(RL_r) = \sum_{k \in K(r)} \mathrm{E}(L_k) \tag{3}$$

$$\mathrm{Var}(RL_r) = \sum_{k \in K(r)} \mathrm{Var}(L_k) \tag{4}$$

2.3 Characterizing reducer data skew

To characterize reducer data skew, we classify reduce-keys into three types: Large, Medium and Small, based on the load they induce on the reduce tasks. Large keys have so many records or bytes that they are difficult to place on a single reducer without causing overload. Small keys contribute such a small load that they can be hashed over all the reducers with negligible probability of causing overload. Medium keys are in the middle: no single Medium key can cause a problem for a reducer, but we must be careful not to overload a reducer by assigning it the wrong combination of Medium keys. Medium keys may be naturally present in the data, or they may be generated by our Chopping phase.

Our algorithm corrects for skew in two ways. First, it chops Large keys into Medium keys. Second, it uses a packing algorithm which assigns Medium keys to reducers, a much better strategy than uniform hashing (see Figure 7). Chopping and packing together serve to reduce the maximum reducer load and are further described in Section 3.

2.4 Algorithm Features

We highlight the main features of our algorithm so we can categorize it into the large taxonomy of load balancing algorithms.

Sample mapper output: We sample the mapper output space by applying the map function to samples from the input data. This enables collection of load statistics on the mapper output. The map function is a black-box to the sampler. Existing Hadoop samplers [26] only sample the input space.

Progressive sampling: The sampler determines when its sample size is sufficient to generate a load balancing plan of desired quality. We provide a formal model which relates sample size to plan quality. Existing Hadoop samplers [26] require the user to specify the number of samples, and do not formalize the relationship between

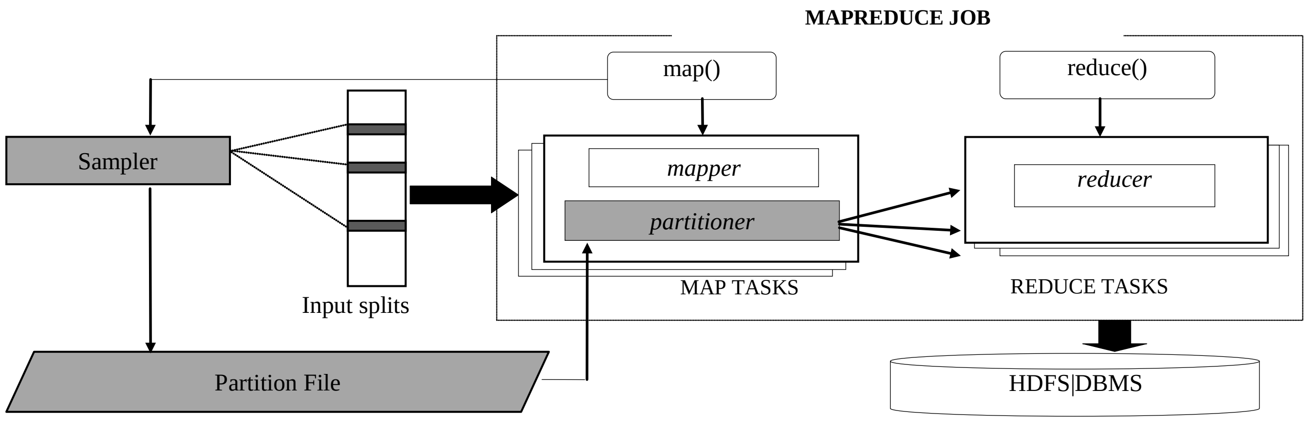

Figure 1: Workflow of Hadoop jobs with load balancing. The shaded boxes are new components added by our system. Our Partitioner replaces the job's original Partitioner which is not shown in this figure.

load balancing quality and sample size.

Small key summarization: We do not require the full reduce-key histogram. We only sample long enough to identify the Large and Medium keys and estimate their load.

Reduce-heavy jobs: Our system is most effective for MapReduce jobs where a significant part of the work performed is performed in the reduce tasks.

Static load balancing: We build on top of Apache Hadoop and treat key-to-reducer assignments as static. Other researchers (see Related Work) have modified Hadoop to support dynamic load balancing, work stealing or partition function modification and while these techniques can balance better, they can cause significant data movement [27], increasing elapsed time and causing deployment difficulties for end users. Static load balancing can also be used before any dynamic load balancing to reduce the expected amount of data moved.

Data skew: The algorithm handles reducer data skew, regardless of type, but it does not handle reducer computational skew.

Statistics collected by the sampler

1. **$\{X_{k1}, \ldots, X_{kM}\}$:** metrics that define key load and their expectations and variances: $E(X_{kj})$, $V(X_{kj})$. Typical metrics are the number of records and bytes per key

Inputs to the load balancer

1. Acceptable overload δ, Confidence level α
2. Load model [defaults to model of Section 6]
    a. Distribution of key load $\mathbf{L_k = f_k(X_{k1},\ldots, X_{kM})}$
    b. Expectation $E(L_k)$ as a function of $\mathbf{X_{ij}}$
    c. Variance $V(L_k)$ as a function of $\mathbf{X_{ij}}$
    d. Inverse CDF of reducer load distribution
    e. Large keys labeling criteria [defaults to §6.2.2.1]
    f. [Optional] Function returning sufficient sample size
3. [Optional] Limits on sampling time, percent data sampled

Load Balancing Algorithm

```
1. Compute the right hand side of Equation 5
2. Iterate Steps 3-6 until Equation 5 is satisfied
3. SAMPLE: collect ($n \geq 1$) records from ($s \geq 1$) splits
   a. Run the mapper on the sampled records
   b. Estimate* $E(X_{kj})$, $V(X_{kj})$
   c. Estimate* $E(L_k)$,$V(L_k)$ using the latest estimates of $E(X_{kj})$,$V(X_{kj})$
4. CHOP: split Large keys into Medium keys
5. PACK: compute a partitioning of keys to reducers using the latest estimates of $E(L_i)$, $V(L_i)$
   a. Estimate* E(reducerLoad), V(reducerLoad)
6. [Optional] Estimate the sufficient sample size N
```

(*we sample a minimum amount before estimation)

Figure 2: Sampler Pseudocode

Configurable load model: The load model is given as a configurable per-job callback that estimates reducer load based on the sampled statistics.

Packing random variables: The load induced by a key is a random variable since it is determined by sampling. The random variables are used as the weights in our packing algorithm.

3. ALGORITHM

Figure 1 illustrates the workflow of a MapReduce job that uses our load balancing system. The load balancer runs before the MapReduce job starts. It generates a balanced partitioning plan, -which contains a partitioning of sampled keys to reducers. This plan is made available to the partitioning phase of the MapReduce job before the job starts. This sequence is executed for every new MapReduce job.

The load balancing algorithm iterates through three steps: Sampling, to gather statistics on the reduce-key load distribution, Chopping, to split Large Keys into two or more Medium Keys, and Packing to generate a partitioning of Medium keys to reducers so as to minimize the maximum expected reducer load. The chopper and packer always consider the most recent sampled statistics and key load estimates.

The input to the load balancer consists of a user-specified model of key and reducer load and a quality requirement for load balancing. The load model specifies callbacks for the balancer to compute the expectations and variances of key load. The sampler maintains sample statistics about the number and size of records per key, which are made available to the load model via callbacks. Section 6 describes a load model that we found sufficient for the Oracle Loader for Hadoop and for inverted index creation. This workload function can be parameterized to model a large-class of MapReduce applications.

The algorithm stops when the plan's maximum estimated reducer load satisfies the quality requirements set by Equation 1. The corresponding stopping condition is derived in Section 5.

To summarize, our load balancer iterates through sampling, chopping and packing, updates the estimated reducer load statistics at intervals, and stops when the quality criteria for reducer load balancing is satisfied. The number of samples collected at that point is the sufficient sample size to achieve the probabilistic guarantee of Equation 1. The intervals at which estimates at recalculated are determined by the size of the chunks of data that we generate for sampling (input splits in Hadoop) and the cost of computing the new chopping and packing. The balancer also supports early stopping by letting the user specify limits on the sampling time or percent data sampled. Figure 2 describes this sampling pseudocode.

3.1 Sampling

The goal of the progressive sampler is to collect enough samples to meet the load balancing goal of Equation 1. This number of samples is not known beforehand, and can vary with the number of reduce tasks and the reduce load distribution, both of which can change for every MapReduce job. By not requiring the user to specify the number of samples needed, we reduce the cognitive load on the user allowing them to focus on the quality of the result rather than the process to achieve it.

Our sampler efficiently samples the input space using cluster sampling. It directs Hadoop to give it small chunks, called InputSplits (see Section 3.1.1). It converts the input space sample into reduce-key space sample by executing the user's map() function on each input record. Since the map() function is a black-box, this is similar to the database problems of query optimization on virtual table columns or grouping on the output of user defined functions [23].

The sampler stops sampling when it has collected enough samples to pass a hypothesis test on its reducer load estimates (Section 5), or when the number of samples reaches a user-specified maximum.

The current sampler runs inside the client, a single-process, multi-threaded application. While it is possible to run the sampler as a MapReduce job, this only makes sense when the amount of data it needs to process is large enough to amortize the job start up cost, which we have not found to be the case in our use cases. However, after doing some sampling, we can use the technique of Section 6.3 to estimate how much additional sampling will be needed in order to meet our load balancing criteria. A high enough estimate can be used to trigger a two phase sampling approach.

3.1.1 Implementation on Hadoop

Hadoop provides an API to read data in logical units called InputSplits [40]. The Hadoop framework defines InputFormat and RecordReader interfaces that users must implement in order to create InputSplits, and parse input splits into records. The Hadoop framework assigns one InputSplit to each map task. Our sampler uses the same interface to read data randomly spread through the job's input.

Our sampler configures the user's InputFormat to produce many small InputSplits and processes these splits in a random order. After processing a split, either entirely or partially, based on the size of the split, each sampler thread updates a global state with the statistics it has collected from this split and recalculates the probability that the stopping condition has been reached. The first thread that realizes that that we have sampled enough to meet the hypothesis, signals a stop.

3.1.2 S plit-based sampling

The Hadoop API for reading data is designed for optimal mapper operation: reading large data blocks from a local map task. Existing support for sampling in Hadoop consists of interfaces to pick random rows from each InputSplit, which saves CPU time, not I/O. The sweet spot for our sampler is to sample a few dozen 4MB data chunks chosen throughout a multi-TB input set. Since we need so little data we want to avoid starting a MapReduce job and since we are sensitive to data clustering, we want to read data from many random smallish InputSplits.

To do this, we configure the InputFormat to produce many small InputSplits. However Hadoop's InputSplit interface returns a materialized list of splits with all the split metadata. This list must fit in the memory of the node where the job is submitted, which is not necessarily on the Hadoop cluster. The materialized list is not an issue for Hadoop jobs, since block sizes are large and splits typically match the block size. However, materialized lists can be problematic for a sampler that prefers to sample from a large number of small splits. We overcome this by using the JVM heap size to constrain the number of InputSplits to request from the InputFormat.

3.2 Chopping

Large keys are identified based on the reduce-key histogram, and are split into Medium keys of equal load using range partitioning on a secondary key, or hash-partitioning based on the entire record.

When range-partitioning is used, the balancer performs total-order partitioning on the values sampled in the Large key; it sorts the set of sampled values and picks Qk-1 values to act as partition points or pivots, where Qk is the desired number of Medium keys for the kth Large key.

The number of Medium keys for a Large key depends on the average reducer load, which is computed by dividing the total estimated load by the number of reducers. The balancer exposes several options to configure the chopper. A user may disable chopping, specify a minimum size for a chopped key (analysis of variance not shown here), or specify the criteria for labeling a key as a Large key. These are implemented using Hadoop configuration properties or callbacks into the user-specified load model. In the Oracle Loader for Hadoop, a key k is Large if its load is greater than the average reducer load, and the size of a chopped Medium key is a configurable fraction of the average reducer load.

Since chopping decisions are made based on the sampled values and since they use sampled estimates of load, chopping adds an additional variance to the key load estimates. Estimating this variance is an integral part of our analysis, and is detailed in Section 6 and the Appendix.

3.3 Packing

By the end of the Chopping phase, every Large key has been chopped into several Medium keys. In this phase, all Medium keys are assigned to reducers using a greedy bin-packing algorithm that seeks to minimize the expected maximum reducer load across all reducers. Details on the packing algorithm are in Section 4.

3.4 Using the plan in a MapReduce job

The assignment of Large and Medium reduce-keys to reducers is made available to the map tasks in a partition file, similar to the one used by Hadoop's Terasort and the Pig system [11]. Map tasks process input key-value pairs and produce reduce-key and reduce-value pairs. The map tasks lookup the reduce-key in the partition file. If the reduce-key is designated as a Large key, the associated reduce-value is used to determine the correct reducer. This involves either range partitioning or hash partitioning as the chopping strategy. For hash partitioning, the partition file entry contains a mapping of hash value to a reducer index. For range partitioning it also contains the pivot reduce-values used for total-order partitioning the reduce-values of this reduce-key. Both alternatives are repeatable in case of mapper failure.

If the reduce-key is designated as a Medium key, the key entry in the partition file contains the reducer index it has been assigned. If the key was not sampled by the sampler or it was determined to be a Small key it will not be present in the partition file. At map time, these keys are assigned to reducers via hashing. Given our independence assumption on reduce-keys (Section 3.5.3), any reduce-key not present is very likely have small load and thus very unlikely to cause data skew.

Hadoop Implementation

The partition file is distributed to all the cluster nodes using Hadoop's DistributedCache mechanism. Our system configures the MapReduce job with a new Partitioner class that replaces the original Partitioner class and references the partition file. The job is then submitted for execution. The map tasks use the new Partitioner to distribute load among reducers. The workflow is illustrated in Figure 1.

3.5 Algorithm Assumptions

3.5.1 Chopping requires distributive reduce function

To ensure the equivalence of the reduce function R after applying the chopping procedure, R must be distributive. We reuse the word distributive from the database literature on parallel group-by-aggregates [21], because the key partitioning between the map and reduce phases is similar to the relational group-by.

In our context, the reduce() function is the aggregation operator and Large key chopping generates a partitioning of the reduce-values associated with a reduce-key. A reducer distributes over union if:

$$reduce(k, \bigcup P_i) = \bigcup reduce(k, P_i)$$

where Pi are the partitions into which records with key k have been chopped. If a reducer distributes over union, we can use hashing to assign records to partitions. A weaker condition is for a reducer to distribute over concatenation, in which case we must use ranges to partition the records associated with each key.

Our balancer allows the user to specify whether the reducer is not distributive, in which case no chopping is allowed; distributive over union, which means we can use cheap hash based chopping; or distributive over concatenation, which initiates range partitioning of the records. Chopping, while improving reducer balance, adds an additional term to variance of reducer load and thus increases the number of samples that need to be taken.

In a MapReduce context it is typical to delay the physical execution of combination operators until the result is consumed, typically by another MapReduce step. For example, the output of a sorting step will typically not concatenate its output into a single file, but instead generate multiple files, each storing a non-overlapping run.

Joins require work to fit into the distributive model and we plan to support them in the future. One approach we are considering is based on the fragment-replicate (FR) approach for parallel hash joins [7]. In this approach, any reduce-key that is chopped must be chopped carefully: only the records coming from one of the input sources can be chopped while all the records from the other sources must be replicated to every reducer handling that key. To do this, the sampler and the Partitioner must be extended to detect the logical source of their input records.

3.5.2 Reduce-key sampling

The ideal way to sample the mapper output is to materialize the entire mapper output and sample it. However this is prohibitively expensive so instead we choose random InputSplits from the mapper input, which generally correspond to runs of data in a file, and execute the mapper on these runs.

As we require statistics from the mapper output space, not the input space, we run the map() function on the input sample and collect statistics on the resulting output data. This is adequate as long as there are no "computational bombs" hidden in the input. A computational bomb is a low probability input record that causes the mapper to generate a huge number of output records. Such a bomb is likely to be missed by the sampler but it can have a huge effect on the map output space distribution.

3.5.3 Cluster sampling

Sampling chunks of data is known as block-sampling or cluster-sampling. Cluster sampling is equivalent to simple random sampling if the keys are distributed independently among the chunks or blocks of data. Any correlation of keys with blocks violates this independence assumption and decreases the effective sample size, so the sampling procedure must be corrected [3, 20]. Most database samplers are cluster samplers since reading contiguous disk blocks is more I/O efficient than reading random individual records. We plan to address potential cluster correlation and its effect on cluster sampling in future work.

3.5.4 Reducer load model

Our algorithms for variance estimation and bin packing require that combining the load generated by separate reduce-keys be additive. Estimation of Medium key load for keys generated by key chopping requires that the load for a single reduce-key be decreasing with respect to both bytes and records. Small key summarization requires that the load model rely only on global statistics, not per reduce-key statistics. All these assumptions are met by the linear load model of Section 6, which we have found to be very commonly applicable in practice.

4. PACKING ALGORITHM

We can map our packing problem to the minimum makespan problem on identical parallel machines from classical scheduling theory [29]. Makespan is the maximum completion time across all machines. The objective is to minimize the maximum load over multiple identical tasks. This problem is strongly NP-hard with has several heuristic solutions [8].

4.1 Packing of random variables

Sampling adds a new spin on the packing problem. Since key loads are estimated from sampled information, they have sample means and variances.

The packing algorithm itself can either be sampling-agnostic or sampling-aware. A sampling-agnostic packer does not know that its input key loads have variances since the sampler and packer essentially treat each other as black boxes. We implement a sampling-agnostic packer by sending the sample means of key loads as input to the packer. The packer generates a key-to-reducer partitioning that optimizes its scheduling heuristic, and the sampler samples until it can provide good confidence intervals on the packer-determined reducer loads. The advantage of being sampling-agnostic is that we can change the packing algorithm without changing the sampling workflow. For instance, we can upgrade to packers with better worst-case performance ratios or run times [5, 15] or change the packer input to be a function of the means and variances e.g. “key load = E(key load) + 2Var(key load)”.

In the future, we plan to study sampling-aware packing, in which the objective function understands that loads have variances. A useful objective would be to maximize the probability that each reducer’s load is less than some percentage of the average load.

4.2 Packer implementation

The packer used in this paper is sampling-agnostic and accepts as input the mean value of key loads, as estimated by sampling. It uses the best-fit decreasing bin-packing heuristic which considers units in the same order as the LPT scheduling heuristic (longest processing time first), which orders jobs in non-increasing order of processing time, and assigns each job to the least loaded machine [22].

5. PROGRESSIVE SAMPLING

The sampler’s goal is to collect the minimum number of samples that will guarantee the load balancing goal of Equation (1). The sampler and packer run iteratively until enough samples have been collected. The output of the packer is a load balancing plan which contains the assignment of sampled Medium keys to reducers.

Since loads are estimated from sampled information, they have sample means and variances. This is an important observation. It implies that we can compute a target maximum reducer load variance. We use this fact to determine when to stop sampling.

Given a key partitioning, the load balancer estimates the expectation and variance of each reducer’s load using Equations 3-4. Applying a hypothesis test on the reducer load distribution RL, the sampler can stop when the variance for every reducer is small enough to guarantee the hypothesis in Equation 1. Equation 5 arises from a hypothesis test on the RL distribution, to compute the target variance that will accommodate a maximum overload of $(1+\delta)$ at a confidence level of $(1-\alpha)$.

$$\forall r, \frac{Var(RL_r)}{E(RL_r)^2} \le \left(\frac{\delta}{D^{-1}_{(1-\alpha)}} \right)^2 \tag{5}$$

where $D^{-1}_{(1-\alpha)}$ is the inverse CDF at probability=(1-α). The hypothesis test passes, and sampling stops, when the variance of each reducer's load satisfies Equation 5. The number of samples collected at that point is the sufficient sample size to achieve the probabilistic guarantee of Equation 1.

An important point about this hypothesis test is that we directly test the reducer loads. We do not perform hypothesis tests on the chopped key loads before packing them, even though they are also random variables with finite variances. Our approach differs from the key chopping approaches in [6, 26, 28], which test the variance of chopped keys. Testing the reducer load tends to require fewer samples in practice, since unlike variances, standard errors do not add linearly. Distributed hash table systems like CHORD [35] also exploit this fact.

Our implementation also performs additional hypothesis tests on the collected statistics, like record size, to ensure confidence about these estimates. These metrics are treated as constants in our calculations once they have small enough variance.

Table 1: Notation for the linear load model

Name	Description
T	Estimated # of map-output records
T_k	Estimated # of map-output records with key k
B_k	Estimated # bytes in T_k records
N	# records sampled from the mapper output
N_k	# sampled records with reduce-key k
L_k	Load generated by reduce-key k
Q_k	Number of chopped keys from Large reduce-key k
L_{qk}	Load of a single chopped key from Large reduce-key k. The chopped key has T_{qk} records and B_{qk} bytes in those records.
RL_r	Load assigned to reducer r

6. EXAMPLE: REDUCER LOAD MODEL

Here we describe the reducer load model we find sufficient for our purposes including the Oracle Loader for Hadoop product, and the inverted index creation (see experiments in Section 7). This utilizes a linear workload function that can be parameterized to model a large-class of MapReduce applications. Using this model we derive the reducer load variance as required by the stopping condition in Equation 5. We also derive an expression for the sufficient sample size, and a lower bound on the smallest maximum overload that can be guaranteed by the sampler.

Alternately, our load balancer accepts a user-provided oracle to define a load model with user-specified variance as detailed in Figure 2.

6.1 Load model

In our linear model, the load Lk generated by a single reduce-key k is a weighted linear function of the number of records with the key (Tk), and the number of bytes in these records (Bk). ck, ct, and cb are application-specific weights:

$$L_k = c_k + c_t T_k + c_b B_k \tag{6}$$

Substituting the linear key loads into Equation 2, the reducer load becomes a weighted linear combination of the number of reduce-keys assigned to this reducer, and the number of records and bytes in those keys. If K(r) is the set of keys assigned to reducer r:

$$RL_r = \sum_{k\in K(r)} L_k = c_k|K_r| + c_t \sum_{k\in K(r)} T_k + c_b \sum_{k\in K(r)} B_k \tag{7}$$

A useful technique to determine the weights is to assign them comparable units e.g. seconds per key, record, and byte:

a) c_k represents the cost of starting the processing of a new reduce-key. This may entail the execution of a new SQL statement, or the creation of a new file.
b) c_t represents the per record cost of processing or sorting the records associated with a given record key.
c) c_b represents the cost of processing the bytes in the records associated with a reduce-key, which typically depends on the cost of moving records or performing a calculation on them.

6.2 Variance of key load

From Equations (3) and (4), the reducer variance is the sum of the key load variances of the keys assigned to the reducer. In this section, we will derive an expression for key load variance. The balancer uses these key variances to compute the reducer load variance from Equations 3-4, which are plugged into Equation 5 to test the stopping condition for the sampler.

Sampling contributes two types of variance to the key load. The first is the sampling variance in estimating key load. This applies to all sampled keys. The second type of variance only applies to Large Keys that get chopped into Medium Keys. Since we use the sampled data to determine the pivots for chopping, it adds a variance to the Medium key load. We will derive expressions for both variances.

6.2.1 Sampling variance

Let T_k denote the number of records with key k in the entire dataset. Let N_k denote the number of sampled records with key k. If T is the total number of records in the entire data, N is the total number of samples, and p_k is the probability that a record data has key k, we can model N_k ~ Binomial(p_k, N) and T_k ~ Binomial(p_k, T). The expectation and variance of T_k is just the binomial variance $Tp_k(1- p_k)$. For datasets with a large number of records per key, we can approximate binomial T_k by a normal distribution.

To illustrate a simple example of variance derivation, we will also model B_k as a normal random variable B_k ~ Normal(μ_b,σ_b^2).

$$L_k = c_k + c_t T_k + c_b \sum_{1}^{T_k} \mathrm{N}(\mu_b, \sigma_b^2)$$

6.2.2 Additional variance due to chopping

Chopping adds additional variance to the chopped key's load, since chopping operates on sampled key-value pairs. Our algorithm partitions each sampled Large key k into Q_k Medium keys of equal load L_{qk}. Based on Equation 6, the key load L_{qk} is the weighted sum of the number of records in the Medium key (T_{qk}) and the number of bytes in those records (B_{qk})

$$L_{qk} = c_k + c_t T_{qk} + c_b B_{qk} \tag{8}$$

Continuing our example where B_{qk} is a normal random variable, the chopped key load L_{qk} is a sum of normals, with expectation and variance as derived in Appendix A, with $c_1 = (c_k + c_b\mu_z)^2$ and $c_2 = c_t^2\sigma_z^2$.

$$\mathrm{E}(L_{qk}) = c_k + \sqrt{c_1}E(T_{qk}) \tag{9}$$

$$\mathrm{Var}(L_{qk}) = c_1\mathrm{Var}(T_{qk}) + c_2 E(T_{qk}) \tag{10}$$

6.2.2.1 Variance depends on the chopping strategy

All that remains to complete our example is to model the distribution of T_{qk}, the estimated number of records in the q^{th} chopped key, which depends on the number of chopped keys for key k (Q_k), which is configurable and can be application-specific.

We model T_{qk} ~ Binomial(D_k, T_k), where T_k is the total estimated records for key k and D_k is the binomial probability. The variance of T_{qk} is thus the variance of a hierarchical binomial variable. The binomial probability

D_k is itself a random variable whose value is determined by the chopping strategy, and by the desired number of chopped keys (Q_k). Using [36], the Appendix derives a distribution for D_k for the range-partitioning chopping strategy.

To complete our example, we will use the definition of Large and chopped keys implemented in the Oracle Loader for Hadoop. OLH defines a key k as being Large if its load L_k is greater than the average reducer load A=L/R, and chops Large keys into Q_k = ceiling(s.L_k/A) Medium keys, s ≥ 1. Appendix A contains a derivation for E(T_{qk}) and Var(T_{qk}) based on this OLH model, which we plug into Equations 9-10 to get the variance of chopped key load:

$$\mathrm{E}(L_{qk}) = c_k + \sqrt{c_1}\frac{T}{R} \tag{11}$$

$$\mathrm{Var}(L_{qk}) = \frac{(c_1+c_2)TRp_k + c_1T(1-2p_k)}{R^2p_k} + \frac{c_1T^2}{RN} \tag{12}$$

6.3 Expression for sufficient sample size

An advantage of the linear model is that it has a simple closed-form expression for sufficient sample size. While the sampler's stopping condition in Equation (5) does not need a computable sample size, knowing the size in advance is useful in practice for:

(a) Early stopping: the algorithm can stop early if the required sample size is too large for sampling to complete within user-specified memory or time limits.

(b) Better resource usage: the balancer can start up a parallel MapReduce job for sampling if the required sample size is large enough to justify the overhead of starting the job.

To derive the sample size we can substitute the expectation and variance of reducer load into Equation 5, and solve for the sufficient sample size for a given reducer N(r). The minimum sample size is the maxr(N(r))

$$N(r) = \frac{K_r c_1 T^2}{RK_r^2 f_0^2 v_{\max}^2 - R\sum_{k\in K(r)} f_1(c_1,c_2,T,R,p_i)} \tag{13}$$

A final note is that it is possible to consider a finite population correction (FPC) [4] when the sample size is a large proportion of the total. We do not discuss FPC in this paper since the required sample size was small in all our experiments (<2% of data size).

6.4 Lower bound on the overload

Reducer load variance is minimized when N=T and the sampler has read all the data, a situation we want to avoid in practice! Plugging this minimum variance into Equation 5 gives us a minimum attainable overload $\delta_{\min}$. A non-zero $\delta_{\min}$ implies that the load balancer will never be able to generate a partitioning plan of user-specified quality δ if $\delta < \delta_{\min}$. To handle this case, in our implementation we allow the load balancer to be configured to either conservatively quit without generating a plan, or to generate a best-effort plan with its specified time and memory resources.

7. EXPERIMENTS AND RESULTS

We tested the performance of the load balancing algorithm with two Hadoop applications: (a) Oracle Loader for Hadoop (OLH), a commercial Oracle product that leverages Hadoop to prepare and load records in parallel into a partitioned Oracle database table (b) a MapReduce application that builds an inverted index over a document collection.

We tested both applications on skewed data, and compared the elapsed time of the jobs with and without load balancing. We also measured the maximum reducer load factors. The load factor of a reducer is the ratio of its observed load to the average (perfectly balanced) reducer load. Elapsed time reflects practical gains but it is harder to isolate cluster specific side-effects like contention for a single disk-arm. Load factor is a more stable

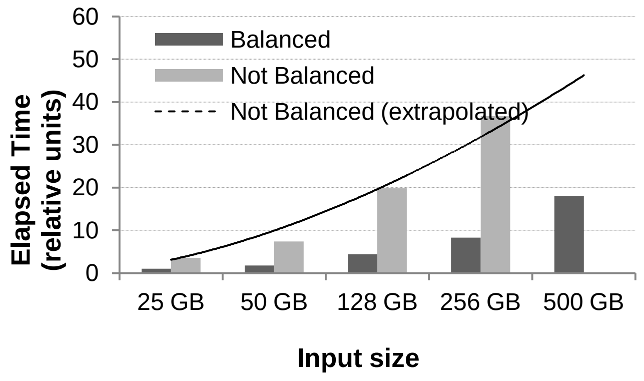

Figure 3: Oracle Loader for Hadoop on skewed data ran 3x-4x faster with load balancing. Speedups increased with increasing input size. Without load balancing, the largest experiment of 500GB had not completed even after running for 3X longer than its balanced counterpart (outside the plotted region) and was killed (Experiment: OLH writing Datapump-format files to HDFS on cluster #3; 128 reducers; 128 reduce-keys (i.e. database partitions) with Zipf(1) skew)

measurement that reflects the quality of the algorithm independent of external factors like resource contention. We report both metrics.

We found that load balancing was most effective for large input data sizes with reduce data skew and a large percentage of the total work in the reduce phase.

In our test data, jobs without load balancing had load factor as high as 10 or 20, which is equivalent to a 100-200% overload. Load balancing reduced the load factor to <1.5, which is only 5% overload As a result, load balanced jobs finished up to 10-60% faster; a significant gain for long-running jobs.

In our experiments, the overhead due to sampling and load balancing was negligible. The sampler progressively determined the sufficient sample size for jobs of different skews and input sizes without user input. The load balancer spent less than 1% of the total job time in the sampling phase since it always sampled just enough data to achieve the required confidence guarantees.

7.1 Hadoop configuration

Cluster configuration: We ran experiments on multiple development and production Hadoop clusters. Cluster #1 had eight UltraSPARC nodes each with 8GB RAM and 2x73GB disks, and interconnected with Gigabit Ethernet. Cluster #2 had seven x86_64 nodes each with 48GB RAM, 1x500GB disk, and interconnected with InfiniBand. Cluster #3 was a production cluster of 15 nodes, each with 48GB RAM, 12x3TB disks and interconnected with InfiniBand.

Hadoop job configuration: Hadoop jobs can be tuned with several parameters to optimize job performance. We focused on varying the number of reducers since this parameter largely affects the reducer load skew. We kept all other parameters constant. The optimal number of reducers for a job depends on the application and on cluster constraints. For a MapReduce job that loads into a database, it depends on the degree of parallelism supported by the database, the number of database partitions to be loaded, and the number of available reducer slots. The Hadoop user guide recommends using 0.95 or 1.75 times the maximum available reducer slots. Load balancing resulted in significant improvements across all settings.

Sampler Configuration: All balancer parameters are exposed as Hadoop Configuration properties [40]. The user can configure the maximum acceptable overload (δ), the confidence level that the balancer is within that margin of error (α), and the reducer load model (Figure 2). The user can also specify limits on sampling resource utilization e.g. maximum number of samples, and maximum sampling time.

7.2 Oracle Loader for Hadoop

Our first test application was the Oracle Loader for Hadoop (OLH), a product that loads records from the Hadoop ecosystem into a partitioned Oracle database table. OLH can be used as the load step of an ETL process (Extract-Transform-Load). OLH can insert data directly into a table using the JDBC or OCI protocols. Alternately, it can write data to HDFS for offline bulk loads, in text-delimited format or in a proprietary Oracle format (Datapump) which enables faster bulk loads into the Oracle database.

OLH input data may contain records destined for multiple database partitions. The Oracle database supports a wide range of partitioning strategies to improve load balancing, reduce locking, and improve parallelism. The database determines the partition for a record by applying a partitioning function to its partitioning each record based on partitioning columns defined in the table schema.

OLH is implemented as a Hadoop MapReduce job. OLH uses the database partition ID as the reduce-key for a record, and the record acts as the reduce-value. This setup allows all records from a single database partition to be processed by the same reducer, but can result in unbalanced reducer loads when the input data contains more records for certain database partitions than others.

Our balancer alleviates this problem and reduces the overall execution time of the OLH Hadoop job for skewed inputs. It balances load across OLH reducers by assigning records for a Large database partition across multiple reducers, while maintaining load efficiency by optionally sorting records by primary key within each chopped partition. This chopping is possible as the Oracle database supports efficient parallel inserts into the same database partition.

When the load-balancing feature is enabled, OLH first executes our load balancer in its sampling phase. If the sampler detects a large database partition in the input, it assigns the records in that partition to more than one reducer using either total-order partitioning on the primary key or weighted hashing (Chopping and Packing). After the balancer has finished execution, OLH starts up its MapReduce job to load the input data. The OLH partitioner reads the partition file generated by the balancer, and assigns records from each chopped database partition to the pre-determined reducers for that partition.

OLH Test data: we generated several GB of test data in which each row had a partitioning column, and several payload columns containing random uniform primitive database types and variable length strings of length U[1,200] bytes. We skewed the partition sizes in the input by drawing the partitioning column data from Zipf (exponent=1) or Poisson (lambda=0.5) distributions.

Results: we ran OLH to load data in JDBC and Datapump modes, with and without load balancing. In JDBC mode, OLH loaded records into an Oracle database using JDBC. In Datapump mode, it wrote Datapump files to HDFS. We set the number of reducers equal to the number of database partitions.

Figure 3 shows elapsed time speedups on an OLH-Datapump experiment to load data containing records from 128 database partitions, skewing partition size per a Zipf(exponent=1) distribution, with the largest partition containing ~18% of the data. We ran OLH with #reducers=#partitions, and with balancing was disabled, OLH assigned one database partition to each reducer. This resulted in a maximum reducer load factor of ~18%, corresponding to the largest partition; a 95% overload. When we enabled load balancing, the load factor dropped to 1.06, a mere 6% overload. Load balancing resulted in up to 80% faster job execution times.

The improvements in elapsed times increased with larger skew. We ran OLH-JDBC experiments on Zipf(10,1)

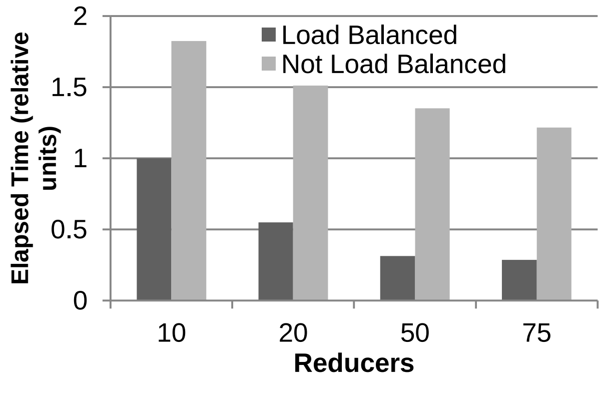

Figure 4: Building an inverted index on skewed data ran 2x-4x faster with load balancing. Speedups increased as we increased the number of reducers (Experiment: 24 GB data; cluster #3; 1000 distinct words with a Zipf(1) word frequency distribution)

and Poisson(lambda-0.5) skewed data. The largest database partition in the Poisson data was 2x larger, and elapsed time improvements correspondingly 2-3x higher, than for the Zipf data.

7.3 Inverted Index

Our second test application was a MapReduce application that builds inverted indexes on a document collection. In a MapReduce implementation, the reduce-key is a word and the reduce-value is a list of locations where the word occurs. The map phase emits <word, location> pairs, sorted by location per word, and the reduce phase aggregates all the locations for a word into a list. Keys are skewed when some words in the input occur more frequently than others

Test data: We generated a synthetic document collection and skewed the word frequency according to a Zipf distribution with exponent=1. This exponent models the frequency distribution of words in English.

Results: We ran several experiments in which we varied the data size from 400MB to 24 GB, and the number of reducers from 10 to 75. Load balanced jobs ran 2-4x faster than unbalanced jobs and the speedups improved as we scaled up the number of reducers (Figure 4). The speedups also scaled well as we increased the input size: on 240 GB of data load balancing gave us 6x speedups in elapsed time.

7.4 Benefit of progressive sampling

To test the benefit of progressive sampling, we measured sample size and sampling time while scaling up the input data size and the number of reducers used to run the Hadoop job.

A benefit of progressive sampling was that sampling time was relatively constant as we increased the input size, while fixing the data skew distribution and number of reducers (Figure 5). The sample size scaled linearly as we

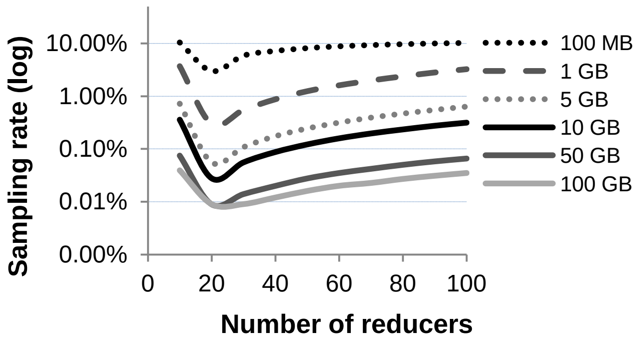

Figure 5: Sample size scaled linearly with the number of reducers, and stayed relatively constant with input size (OLH, 10 DB partitions, Zipf(10,1) skew, cluster #2).

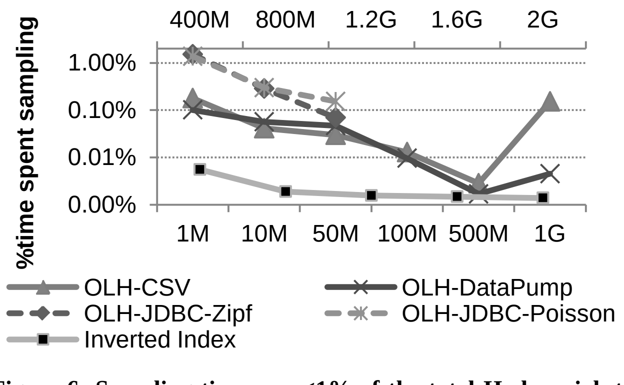

Figure 6: Sampling time was <1% of the total Hadoop job time, and scaled sub-linearly with dataset size. The lower X-axis labels input sizes for Oracle Loader for Hadoop. The upper X axis labels dataset sizes for Inverted Index (Zipf(100,1) skew, 100 reducers, cluster #2).

increased the number of reducers (Figure 5), but the sampling overhead was negligible since both sample size and time were very small (<1% of total job time, Figure 6).

The sampling overhead was negligible, even when the load balancer was enabled on uniform data with no skew (results not shown).

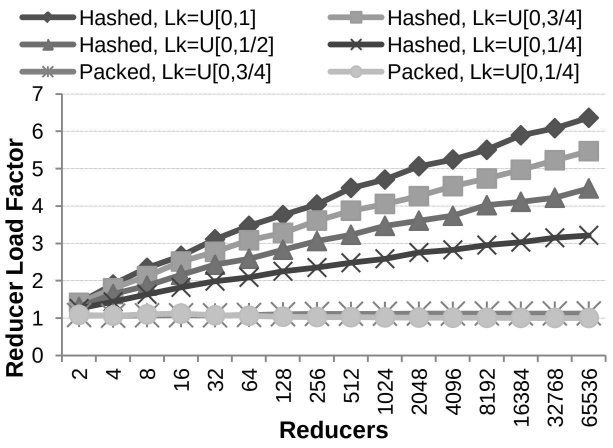

Figure 7: Comparing uniform hashing versus packing as a partitioning strategy for assigning keys to reducers. The reducer load factors from uniform hashing progressively degrade as we increase the number of reducers and the maximum key load *M*. Key loads are generated from U[0, *M*].

7.5 Benefit of packing over hashing

We ran simulations to illustrate the benefit of our packing strategy over a uniform hashing strategy for partitioning keys to reducers. We modeled the scenario in which there are no Large keys, or when chopping has already split all the Large keys into Medium keys. Considering *R* reducers, we randomly generated Medium key loads from a uniform distribution $L_k \sim$ U[0,*M*], with the maximum key load $M \leq 1.0$ and total key load $\sum L_k = R$. We assigned these keys to reducers via packing and via hashing and measured the maximum reducer load for each strategy averaged over multiple runs. We expected packing to better at balancing Medium keys, due to the non-zero probability of hash collisions. Figure 7 shows the results of the simulation. While packing consistently found a balanced partitioning, uniform hashing produced progressively worse load factors as we increased the number of reducers *R* and the maximum key load *M*.

8. DISCUSSION

When is a key infrequent enough to be Small? The criteria for Large key chopping is quite clear and well motivated: When the load on a reducer is larger than the average reducer load *L*/*R*, it must be split. The criteria for deciding when a key is Medium and must be packed vs. when it is Small and can be hashed, is less clear. But fortunately we can use the same criteria we used to determine that we have sampled enough data to determine when the load for a reduce-key is small enough to allow the key to be classified as Small.

It is helpful to think of the overhead of Small key processing as an additional load term, call it L_s that should be added to each RL_r calculated by the packer. Each L_s incorporates the cost of processing the Small keys that are randomly assigned to this reducer. The worst case for Small variance is when all the Small keys have the maximum Small size which we will call *sco*, for Small cutoff. Let us assume the worst case: that we have *j* times *R* Small keys each with *sco* records for some *j*. So $L_s = sco$ Binomial($1/R$, $j\ R$) and Var(L_s) = $sco^2\ j$. Let us choose *k* so that the number of Medium records is $k\ R\ sco$. So the total number of records is $T = (k+j)R\ sco$. Now we can calculate the variance of the Small load factor as Var(L_s / (T/R)) = $j\ (k+j)^{-2}$ and choose *j* and *k* so that the total reducer load factor satisfies our hypothesis test in Equation 5.

Note that the choice of *j* and *k* is somewhat arbitrary. We can minimize the variance of the reducer load by choosing large *k* and a small *j*, doing more bin packing and less hashing but this requires recording a large number of Medium keys in the partition file. Performance will be better served by choosing a small *k* and a large *j*, and storing a small number of Medium keys in the partition file.

What happens downstream of Hadoop? In this paper, we focus on balancing reducer data skew. A related problem is that of addressing skew in the target system downstream of the Hadoop job. For instance, the PNUTS [33] system range-partitions data across a shared-nothing cluster and guarantees that data remains balanced across machines after bulk inserts. The bulk loader: (a) determines partitions over the union of new and existing data by sampling both sets and (b) optimally moves existing data to maintain balanced partitions. Balancing existing partitions was not a concern for us in the Oracle Loader for Hadoop (OLH) since the target Oracle database has its own static partitioning scheme that is not changed by the data loader, and can balance parallel queries on skewed partitions. The Oracle database has a partitioning advisor, but that is a different product than OLH. The dynamic partitioning schemes used in 'NoSQL' stores like HBase or PNUTS are quite different than the static partitioning scheme used inside an RDBMS. While our sampler can be modified to accept metadata from the target system about existing partitions, a general solution for evenly repartitioning new and existing partitions simultaneously from within a Hadoop job is not as immediately obvious. Generalizing this problem of combining old meta-data and new samples is an interesting topic for future research.

9. RELATED WORK

Skew in parallel databases: has been widely studied in the database literature, largely in the context of parallel joins [7, 39]. Random sampling techniques in relational databases [24] have historically been used to estimate key histograms for query optimization [10, 13, 14, 18] and for load balancing in parallel databases [6, 7].

The approach of chopping keys was used to optimize parallel sorting and parallel joins, and was accompanied by work on determining the minimum sample size required to estimate equal-size partitions [2, 32]. These results have been incorporated in recent MapReduce systems. The PNUTS system has a bulk loader [33, 34] which uses sampling to generate balanced partitions on the target data store and uses the sufficient sample size result from [32]. Pig [11] and Hive [37] are high-level languages for executing MapReduce programs on Hadoop and include implementations of skew joins. Pig version 0.5 has a skew join implementation [28] based on [7]. It uses a sampling MapReduce job to determine the key histogram and splits keys such that they fit into reducer memory. Hive has a skew join implementation that uses bucketing and accepts the degree of skew as input from the user. As far as we know, neither system automatically determines the sufficient sample size or stopping condition for sampling.

MapReduce Skew: Skew was largely ignored in the early days of MapReduce. The first efforts were domain-specific and did not have general load models: Kolb et al [31] did not use sampling to estimate loads, and Kwon et al [16] partitioned data to fit in task memory based on sampling, but did not discuss sample size or sampling variance. Recent approaches are more general, but require modification to the Hadoop framework and do not provide load balancing guarantees. Gufler et al [12] propose a system that counts bytes and tuples to measure load, but does not use sampling. They modify the Hadoop JobTracker to aggregate statistics from all records and all mappers. They acknowledge that this approach does not scale to large numbers of keys, so they only maintain partition-level statistics, which is a less granular than key-level statistics, and an approximation. We are able to accurately maintain key-level statistics because our balancer only requires data about Large and Medium keys, and simply hashes Small keys. Kwon et al [17] perform dynamic load balancing to mitigate computational and assignment-based skew in map and reduce tasks. Dynamic load balancing is good for handling computational skew, but requires more data movement, which can get expensive. Our static load balancing can also be applied before dynamic load balancing starts, to reduce the expected amount of data moved.

A recent paper by Vernica et al [38] is not specific to skew-mitigation but describes an extension to Hadoop that adds *adaptive* mappers and partitioners that exchange messages via a central coordinator. Our progressive sampler could complement this extension by running directly inside the job's map tasks.

Hashing: Finally, the problem of determining which keys can be safely hashed without causing overload has broad applications. Fan et al [9] determine the minimum cache size required to guarantee a worst-case maximum load on backend servers, by mapping the problem to a binomial distribution analysis [30].

10. CONCLUSIONS AND FUTURE WORK

We presented a static load balancing system to mitigate reducer data skew for MapReduce applications, and a progressive sampling technique to automatically determine the minimum number of samples required to achieve high confidence on the balanced reducer workload.

Our solution is domain-agnostic, it can work with many existing Hadoop applications and infrastructures and requires no user intervention other than enabling the balancer for a job. The statistical model used to determine the stopping condition can be parameterized to be applicable to a wide range of applications. We illustrated closed form solutions for a linear load model in some detail.

We presented our solution in general terms: distributive reduce functions, a broadly applicable minmax scheduling objective and user-defined workload models. Further, we carried out a detailed analysis of sampling variance, bin packing and Small key distributions to provide probabilistic guarantees on the quality of the load balancing. We also formalized the conditions for correctness of the chop-pack transforms used by the balancer.

The load balancer is implemented in the Oracle Loader for Hadoop product. With load balancing enabled, Hadoop jobs with skewed reduce-keys showed a marked reduction in the maximum reducer load factor, and had corresponding speedups in execution time. The sampler was adaptive to different skews, dataset sizes and cluster configurations, and sampling overhead was minimal even when balancing was enabled on jobs with no skew.

In future work**,** we plan to increase the applicability of our work to a wider range of MapReduce applications and address correlations using cluster sampling.

11. ACKNOWLEDGMENTS

We thank the Oracle Loader for Hadoop team for their continuing input and feedback, Gabriel Balan, Alan Wood and Geng Li for providing reviews of early drafts, and Russ Brown and Jacco Draiijer for guidance with the Hadoop clusters.

12. REFERENCES

[1] Apache Hadoop: *http://hadoop.apache.org*.

[2] Blelloch, G.E. et al. 1991. A comparison of sorting algorithms for the connection machine CM-2. *Proceedings of the 3rd annual ACM symposium on Parallel Algorithms and Architectures - SPAA '91* (New York, New York, USA, Jun. 1991), 3-16.

[3] Chaudhuri, S. et al. 2004. Effective use of block-level sampling in statistics estimation. *Proceedings of the 2004 ACM SIGMOD international conference on Management of data - SIGMOD '04* (New York, New York, USA, Jun. 2004), 287.

[4] Cochran, W.G. 1977. *Sampling Techniques*. Wiley and Sons, Inc, New York.

[5] Coffman, J. et al. 1978. An Application of Bin-Packing to Multiprocessor Scheduling. *SIAM Journal on Computing*. 7, 1 (1978), 1.

[6] DeWitt, D.J. et al. Parallel sorting on a shared-nothing architecture using probabilistic splitting. *[1991] Proceedings of the First International Conference on Parallel and Distributed Information Systems* 280-291.

[7] DeWitt, D.J. et al. 1992. Practical Skew Handling in Parallel Joins. (Aug. 1992), 27-40.

[8] Dean, J. and Ghemawat, S. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*. 51, 1 (Jan. 2008), 107.

[9] Fan, B. et al. 2011. Small cache, big effect. *Proceedings of the 2nd ACM Symposium on Cloud Computing - SOCC '11* (New York, New York, USA, Oct. 2011), 1-12.

[10] Ganguly, S. et al. 1996. Bifocal sampling for skew-resistant join size estimation. *ACM SIGMOD Record*. 25, 2 (Jun. 1996), 271-281.

[11] Gates, A.F. et al. 2009. Building a high-level dataflow system on top of Map-Reduce: the Pig experience. *Proceedings of the VLDB Endowment*. 2, 2 (Aug. 2009), 1414-1425.

[12] Gufler, B. et al. 2011. Handling Data Skew in MapReduce. *CLOSER* (2011), 574-583.

[13] Haas, P.J. et al. 1995. Sampling-Based Estimation of the Number of Distinct Values of an Attribute. (Sep. 1995), 311-322.

[14] Haas, P.J. et al. 1996. Selectivity and Cost Estimation for Joins Based on Random Sampling. *Journal of Computer and System Sciences*. 52, 3 (Jun. 1996), 550-569.

[15] Hochbaum, D.S. and Shmoys, D.B. 1987. Using dual approximation algorithms for scheduling problems theoretical and practical results. *Journal of the ACM*. 34, 1 (Jan. 1987), 144-162.

[16] Kwon, Y. et al. 2010. Skew-resistant parallel processing of feature-extracting scientific user-defined functions. *Proceedings of the 1st ACM symposium on Cloud computing - SoCC '10* (New York, New York, USA, Jun. 2010), 75.

[17] Kwon, Y. et al. 2012. SkewTune. *Proceedings of the 2012 international conference on Management of Data - SIGMOD '12* (New York, New York, USA, May. 2012), 25.

[18] Larson, P.-A. et al. 2007. Cardinality estimation using sample views with quality assurance. *Proceedings of the 2007 ACM SIGMOD international conference on Management of data - SIGMOD '07* (New York, New York, USA, Jun. 2007), 175.

[19] Lin, J. 2009. The Curse of Zipf and Limits to Parallelization: A Look at the Stragglers Problem in MapReduce. *Proceedings of the 7th Workshop on LargeScale Distributed Systems for Information Retrieval LSDSIR09 at SIGIR 2009* (2009).

[20] Lohr, S.L. 2010. *Sampling: Design and Analysis*.

[21] M. Tamer Ozsu, P.V. 2011. *Principles of Distributed Database Systems*.

[22] Maciej Drozdowski 2009. *Scheduling for Parallel Processing*. Springer Publishing Company, Inc.

[23] Michael Stonebraker, J.M.H. 1998. *Readings in Database Systems*. Morgan Kaufmann.

[24] Olken, F. and Rotem, D. 1995. Random sampling from databases: a survey. *Statistics and Computing*. 5, 1 (Mar. 1995), 25-42.

[25] Oracle Loader for Hadoop: *http://www.oracle.com/technetwork/bdc/hadoop-loader*.

[26] O'Malley, O. 2008. *TeraByte Sort on Apache Hadoop*.

[27] Paton, N.W. et al. 2008. Autonomic query parallelization using non-dedicated computers: an evaluation of adaptivity options. *The VLDB Journal*. 18, 1 (Jan. 2008), 119-140.

[28] Pig Skew Join Specification: *http://wiki.apache.org/pig/PigSkewedJoinSpec*.

[29] R L Graham, E L Lawler, J K Lenstra, A.H.G.R.K. 1979. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of Discrete Mathematics*. 5, 2 (1979), 287-326.

[30] Raab, M. and Steger, A. 1998. Balls into Bins - A Simple and Tight Analysis. (Oct. 1998), 159-170.

[31] Rahm, L.K. and A.T. and E. 2012. Load Balancing for MapReduce-based Entity Resolution. *International Conference on Data Engineering (ICDE)* (2012).

[32] Seshadri, S. and Naughton, J.F. 1992. Sampling Issues in Parallel Database Systems. (Mar. 1992), 328-343.

[33] Silberstein, A. et al. 2008. Efficient bulk insertion into a distributed ordered table. *Proceedings of the 2008 ACM SIGMOD international conference on Management of data - SIGMOD '08* (New York, New York, USA, Jun. 2008), 765.

[34] Silberstein, A.E. et al. 2011. A batch of PNUTS. *Proceedings of the 2011 international conference on Management of data - SIGMOD '11* (New York, New York, USA, Jun. 2011), 1101.

[35] Stoica, I. et al. 2001. Chord. *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications - SIGCOMM '01* (New York, New York, USA, Aug. 2001), 149-160.

[36] Swart, G. 2004. Spreading the Load Using Consistent Hashing: A Preliminary Report. (Jul. 2004), 169-176.

[37] Thusoo, A. et al. 2009. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*. 2, 2 (Aug. 2009), 1626-1629.

[38] Vernica, R. et al. 2012. Adaptive MapReduce using situation-aware mappers. *Proceedings of the 15th International Conference on Extending Database Technology - EDBT '12* (New York, New York, USA, Mar. 2012), 420.

[39] Walton, C.B. et al. 1991. A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins. (Sep. 1991), 537-548.

[40] White, T. 2009. *Hadoop: The Definitive Guide*. O'Reilly.

APPENDIX

A. VARIANCE OF CHOPPED KEY LOAD

In this section, we derive the variance of chopped key load. Recall from Equation 8 in the main paper, that the load L_{qk} generated by the q^{th} chopped Medium key of the k^{th} Large key is:

$$\begin{aligned} L_{qk} &= \alpha + \beta T_{qk} + \gamma B_{qk} \\ B_{qk} &= Sum(Z, T_{qk}) \\ V(B_{qk}) &= T_{qk} V(Z) \end{aligned}$$

with $Sum(X, n)$ as the sum of n variables distributed as X. We assume that L_{qk} is IID.

Methodology: Our general approach is to define several levels of hierarchical random variables. Since L_{qk} is defined in terms of $T_{qk} \sim Bin(D_k, T_k)$, a hierarchical binomial in the main paper, we will need to derive the distribution of D_k and substitute upwards to derive the variance of T_{qk} and L_{qk}.

First, let us derive $V(L_{qk})$ conditional on T_{qk}. For large values, the binomial variable T_{qk} can be approximated as a normal variable. Continuing the example from the main paper, we assume the number of bytes B_{qk} is distributed normally, so $Z \sim N(\mu_b, \sigma_b^2)$. So, L_{qk} is a sum of independent normals and the variance of L_{qk} is the sum of normal variances

$$\begin{aligned} E(L_{qk}|T_{qk}) &= \alpha + \beta T_{qk} + \gamma T_{qk} E(Z) \\ &= \alpha + \sqrt{c_1} T_{qk} \\ V(L_{qk}|T_{qk}) &= \gamma^2 T_{qk} V(Z) = c_2 T_{qk} \end{aligned}$$

writing $c_1 = (\beta + \gamma E(Z))^2$, $c_2 = \gamma^2 V(Z)$.

Next, we can factor in T_{qk} as a random variable with its own variance. Using conditional expectation to factor in the expected and variance of T_{qk}:

$$\begin{aligned} E(L_{qk}) &= E_{T_k}(\alpha + \sqrt{c_1} T_{qk}) \\ &= \alpha + \sqrt{c_1} E(T_{qk}) && (1) \\ V(L_{qk}) &= V_{T_k}(E(L_{qk}|T_{qk})) + E_{T_k}(V(L_{qk}|T_{qk})) \\ &= V_{T_k}(\alpha + \sqrt{c_1} T_{qk}) + E_{T_k}(c_2 T_{qk}) \\ &= c_1 V(T_{qk}) + c_2 E(T_{qk}) && (2) \end{aligned}$$

We will now derive E(T_{qk}) and Var(T_{qk}).

B. VARIANCE OF T_{QK}, THE CHOPPED KEY RECORD LOAD

We will derive an expression for the probability distribution of T_{qk}. Given T_k, the number of records that have key k, we model T_{qk} as Binomial(D_k, T_k). Intuitively, the number of records T_{qk} is drawn from a binomial distribution of T_k records and probability equal to the length of the interval generated by the q^{th} partitioning element normalized to [0,1]. From the main paper, T_k and N_k are binomial variables. We can write out the expectation and variance of T_{qk} using the expectation and variance of a hierarchical binomial (see Equations 15-16):

$$\begin{aligned} T_{qk} &\sim Bin\,(D_k, T_k) && (3) \\ E(T_{qk}) &= E(T_k)E(D_k) && (4) \\ V(T_{qk}) &= E(T_k)E(D_k)(1 - E(D_k)) \\ &\quad + (E(T_k)^2 - E(T_k) + V(T_k))V(D_k) \\ &\quad + E(D_k)^2 V(T_k) && (5) \end{aligned}$$

B.1 Variance of D_k, the chopping probability

Next, we derive the probability distribution of the binomial probability D_k, when range-partitioning is used as the chopping strategy.

To model this variable, we map the problem of partitioning the reduce-values for key k into equal groups into the problem of chopping the unit line [0,1] into groups of equal length. We model the D_k distribution as the (scaled) distribution of this group-length.

First, note that we can map the N_k reduce-values for key k onto points in the [0,1] space: Let $\{x_1 \dots x_{N_k}\}$ be the sorted set of reduce-values. Each x_i is drawn by sampling the unknown reduce-value distribution F. Since we know that if Y has a U[0,1] distribution then the inverse CDF $F^{-1}(Y)$ is distributed as F, the inverse CDF generates a one-to-one mapping between each x_i and a value y(i) on the [0,1] interval. The length of each interval $|y(i) - y(i-1)|$ is hence generated from a random partitioning of [0,1] into $N_k + 1$ intervals.

From [36], the length of the interval $|y(i) - y(i-1)|$ is IID distributed according to $Min(U, N_k)$, where $Min(U, n)$ is defined as the smallest of n elements drawn from U[0,1]. $Min(U, n)$ has the following probability distribution:

$$\begin{aligned}
f(Min(U,n)) &= n(1-x)^{(n-1)}dx \\
E(Min(U,n)) &= \frac{1}{(n+1)} \\
Var(Min(U,n)) &= \frac{n}{(n+1)^2(n+2)}
\end{aligned}$$

The process of chopping groups the $N_k + 1$ intervals into Q_k equal-size groups. Q_k is the number of Medium keys in a Large key, and is application-specific. The expected number of intervals in each group is $N_k + 1/Q_k$. The length of a group is the sum of the lengths of $N_k + 1/Q_k$ consecutive intervals. This group-length distribution is the distribution of D_k.

B.1.1 Treating N_k and Q_k as constants

We will first treat N_k and Q_k as constants for simplicity. The group-length (D_k) is the sum of $N_k + 1/Q_k$ consecutive intervals, and is modeled as:

$$\begin{aligned}
X_k &= Min(U, N_k) \\
D_k &\sim Sum\,(X_k, ((N_k+1)/Q_k)) \\
E(D_k) &= ((N_k+1)/Q_k)E(X_k) \\
V(D_k) &= ((N_k+1)/Q_k)V(X_k)
\end{aligned} \tag{6}$$

B.1.2 Treating N_k and Q_k as random variables

Next, let us treat N_k and Q_k as random variables with sampling variances. N_k is a binomial variable, per our original model. In general, the number of Medium keys Q_k must also be modeled as a random variable, since its value will depend on the sampled key load L_k.

When N_k and Q_k are random variables, we must use conditional expectation to derive the distribution of D_k (first w.r.t. Q_k, then w.r.t. N_k)

$$\begin{aligned}
E(D_k) &= E_{N_k}(E_{Q_k|N_k}(E(D_k|N_k, Q_k))) \\
&= E_{N_k}(E_{Q_k|N_k}(\frac{N_k+1}{Q_k}E(X_k)) \\
&= E_{N_k}(E_{Q_k|N_k}(\frac{1}{Q_k}))
\end{aligned} \tag{7}$$

$$\begin{aligned}
V(D_k) &= V_{N_k}(E(D_k|N_k)) + E_{N_k}(V(D_k|N_k)) \\
&= V_{N_k}(E_{Q_k|N_k}(\frac{1}{Q_k})) + E_{N_k}(V_{Q_k|N_k}(\frac{1}{Q_k})) \\
&+ E_{N_k}(E_{Q_k|N_k}(\frac{1}{Q_k N_k}))
\end{aligned} \tag{8}$$

Next, we must determine the variance of Q_k. We will define Q_k as defined in the Oracle Loader for Hadoop application (OLH). Our load balancer is implemented in OLH, and chops a Large key into Medium keys of size equal to the average reducer load. Hence, $Q_k^{OLH} = \lceil L_k/(L/R) \rceil$. Here, we show the derivation of the simple case when $Q_k = \lceil T_k/(T/R) \rceil$, the case when the workload is equal to the record load ($c_k = c_b = 0$ in Equation 6) since it results a simple closed-form solution for Var(T_{qk}). The derivation for $Q_k = \lceil L_k/(L/R) \rceil$ is similar, albeit more tedious, and we do not include it here due to lack of space. Using the moments of a reciprocal variable (18-19) and dropping the ceiling constraint on Q_k:

$$\begin{aligned} E_{Q_k}(1/Q_k) &= (T/R)E(1/T_k) \\ &\approx (T/R)(\frac{1}{Tp_k} + \frac{Tp_k(1-p_k)}{T^3p_k^3}) \\ &= (T/R)\frac{Tp_k + 1 - p_k}{T^2p_k^2} \\ &\approx (T/R)\frac{1}{Tp_k} = \frac{1}{Rp_k} \qquad (9) \\ V_{Q_k}(1/Q_k) &= (T^2/R^2)(V(T_k)/E(T_k)^4) \\ &\approx (T^2/R^2)\frac{(1-p_k)}{T^3p_k^3} \\ &\approx \frac{(1-p_k)}{R^2Tp_k^3} \qquad (10) \end{aligned}$$

Substituting $E(1/Q_k)$ and $V(1/Q_k)$ back into Equations (7-8) for D_k:

$$\begin{aligned} E(D_k) &\approx E_{N_k}(\frac{1}{Rp_k}) = \frac{1}{Rp_k} \\ V(D_k) &= 0 + \frac{1-p_k}{R^2Tp_k^3} + \frac{1}{Rp_k}E_{N_k}(1/N_k) \\ &= 0 + \frac{1-p_k}{R^2Tp_k^3} + \frac{1}{Rp_k}(\frac{1}{Np_k} + \frac{1-p_k}{N^2p_k^2}) \qquad (11) \\ &\approx \frac{1-p_k}{R^2Tp_k^3} + \frac{1}{RNp_k^2} \qquad (12) \end{aligned}$$

Assumptions: In Equation 11, we assume that the covariance of Q_k and N_k is zero. In Equation 12, we assume that $\frac{1-p_k}{N^2p_k^2} \ll \frac{1}{Np_k}$, which is a reasonable approximation for $p_k \geq 0.1$ (simulation results not presented due to space constraints). Substituting $E(D_k)$ and $V(D_k)$ back into Equations (4-5) for T_{qk}:

$$E(T_{qk}) = \frac{Tp_k}{Rp_k} = \frac{T}{R} \qquad (13)$$

$$V(T_{qk}) = \frac{T(Rp_k - 1) + 2T(1-p_k)}{R^2p_k} + \frac{T^2}{RN} \qquad (14)$$

C. HIERARCHICAL BINOMIALS

When the parameters of a binomial B are variables X,Y:

$$E(\mathrm{B}(X,Y)) = E_x(E_y(E(B|X,Y))) = E(X)E(Y) \qquad (15)$$

$$\begin{aligned} V(\mathrm{B}(X,Y)) &= V(\mathrm{B}(E(X),E(Y))) \\ &- E(Y)V(X) + E(Y)^2V(X) \\ &+ E(X)^2V(Y) + V(X)V(Y) \qquad (16) \end{aligned}$$

D. VARIANCE OF A RECIPROCAL

To calculate the variance of 1/X, we use the formula for the mean of an arbitrary function g() of variable X:

$$\begin{aligned} \langle g(X)\rangle &= g(\langle X\rangle) + 1/2!d^2/dX^2 g(\langle X\rangle)c_2 \\ &+ 1/3!d^3/dX^3 g(\langle X\rangle)c_3 + \\ &+ 1/4!d^4/dX^4 g(\langle X\rangle)(c_4 + 3c_2^2) \end{aligned} \qquad (17)$$

where $c_1, \ldots$ are the cumulants of the distribution of X. The mean and variance of g(X)=1/X approximated up to the second order cumulants, with E=E(X) and V=V(X)

$$\langle 1/X\rangle = 1/E + (V/E^3) + (3V^2/E^5) \qquad (18)$$

$$\begin{aligned} V(g(X)) &= V/E^4 + 8 * V^2/E^6 - 6 * V^3/E^8 \\ &- 9V^4/E^{10} \qquad (19) \end{aligned}$$