Online Aggregation for Large MapReduce Jobs

Niketan Pansare[1], Vinayak Borkar[2], Chris Jermaine[1], Tyson Condie[3]
[1]Rice University, [2]UC Irvine, [3]Yahoo! Research
np6@rice.edu, vborkar@ics.uci.edu, cmj4@rice.edu, tcondie@yahoo-inc.com

ABSTRACT

In online aggregation, a database system processes a user's aggregation query in an online fashion. At all times during processing, the system gives the user an estimate of the final query result, with the confidence bounds that become tighter over time. In this paper, we consider how online aggregation can be built into a MapReduce system for large-scale data processing. Given the MapReduce paradigm's close relationship with cloud computing (in that one might expect a large fraction of MapReduce jobs to be run in the cloud), online aggregation is a very attractive technology. Since large-scale cloud computations are typically pay-as-you-go, a user can monitor the accuracy obtained in an online fashion, and then save money by killing the computation early once sufficient accuracy has been obtained.

1. INTRODUCTION

When running online aggregation (OLA) [10, 9, 11], at all times during query processing, a database system gives a user a statistically valid estimate for the final answer to an aggregate query, along with confidence bounds of the form: "with probability p, the actual query answer is within the range low to $high$". As the computation progresses, the bounds narrow, until (at query completion) the bounds are zero-width, indicating complete accuracy. The main benefit of OLA is that if an acceptably accurate answer can be arrived at very quickly (perhaps in a tiny fraction of the time needed to run the entire query), the query can be aborted, saving significant computer and human time.

Though OLA has arguably had quite a bit of scientific impact (stimulating significant subsequent research), its commercial impact has been limited or even non-existent. In our view, there have been two main reasons for this lack of adoption:

1. First, implementing OLA within a database engine would likely require extensive changes to the database kernel. OLA requires some sort of statistically quantifiable randomness within the database engine. Most OLA algorithms require that the blocks (or tuples) in a relation be processed using a "random" ordering, where "random" has a very stringent mathematical definition. Since this would require significant changes to most kernels and would wreak havoc with techniques widely-implemented by database vendors (such as indexing), vendors and kernel developers have justifiably viewed OLA with suspicion.

2. Second, the goal of saving human and computer time has never been as compelling as one might think. A user of an analytic database who writes a query that goes into a queue and finally makes it out into a big, production warehouse for evaluation has little motivation to kill the query early, even if the user is relatively happy with the results. Ending the query early might save some CPU cycles or disk bandwidth that can then be used by others, but the user who killed the query early may not benefit directly. Furthermore, the database hardware/software/maintenance costs in a self-managed system are not elastic, and do not decrease appreciably if many users decide to stop their queries early.

Significantly, we feel that these two impediments to widespread adoption of OLA may have become less important over time. The "We can't change the kernel" argument is less important at a time when people are implementing all sorts of new databases or data-oriented systems from scratch, particularly for large-scale, shared nothing cluster environments. The "Why stop early?" argument is also harder to make nowadays, given the current move into the cloud. When someone other than the end-user's organization is managing the compute infrastructure, as a query runs, dollars are quantifiable flowing from the end-user's organization and into the cloud. Now that there may be a real and observable cost associated with every CPU cycle consumed and byte transferred, the end-user will likely have to justify those costs to the management. It stands to reason that being able to achieve 99% of the accuracy in 10% of the time will become much more attractive under such a cost model. Thus, we feel that OLA is an old idea whose time has come.

Online Aggregation for Large-Scale Computing. Given the potential for OLA to be newly relevant, and given the current interest on very large-scale, data-oriented computing, in this paper we consider the problem of providing OLA in a shared-nothing environment. While we concentrate on implementing OLA on top of a MapReduce engine [7], many of our most basic research contributions are not specific to MapReduce, and should apply broadly.

Realizing OLA for large-scale, distributed computing is a challenging problem, and a simple extension to the classic work on OLA will not suffice. Classic work in OLA assumes that blocks or tuples are processed in a statistically random fashion, so that the set of data seen at any point in the computation is a random subset of the data in the system—if this is the case, then it is often easy to estimate the final answer using classic methods from sur-

Proceedings of the VLDB Endowment, Vol. 4, No. 11

vey (finite population) sampling theory [16]. The difference in a large-scale, distributed computing environment is the importance of elapsed time. In this type of environment, the basic unit of data that is processed is a *block*, which may contain millions of tuples and be a significant fraction of a gigabyte in size. When many machines are working in parallel, it is natural that there would be a lot of variation in the time taken to process each block. Some blocks could have a lot of data, and take longer to process. It is not unusual for machines to simply die, so they appear as if they have been processing a block forever. This variation in processing time is of tremendous importance if it is somehow correlated with the aggregate value of the block. Such correlation is to be expected: after all, blocks with a lot of data may have greater aggregate values, and take longer to process. In such a scenario, since those nodes that are processing large blocks with big aggregates are more likely to spend more time on those blocks, the set of blocks that have actually completed processing at any particular point are more likely to have small values, leading to biased estimates. This is an example of the well-known "inspection paradox" described by the renewal-reward theory [6]. Dealing with this in a principled fashion in a distributed environment is challenging, and requires innovation both in system design and in statistical analysis.

Our Contributions. We make the following contributions:

- We propose a system model that is appropriate for OLA over MapReduce in a large-scale, distributed environment.
- We describe in detail how we implemented our model in Hyracks [2].
- We discuss a Bayesian framework for producing estimates and confidence bounds within our model.
- We offer experimental evidence that our model produces accurate and usable estimates very quickly.

The remainder of this paper describes a MapReduce OLA library, implemented on top of Hyracks, for providing estimates for the aggregates: `SUM`, `COUNT`, `AVG`, `VARIANCE`, or `STD_DEV`. To use this library, the user writes a Hadoop-style map-reduce job (that conforms to the programming interface discussed in the appendix B) along with a class that is called-back by the MapReduce OLA library, every time an estimate is generated. Typically, this class can serve as an input to other map-reduce jobs or it can also pipe the estimates to a GUI front-end, thus providing a user interface similar to the classical online aggregation systems [10]. It is important to note that the programmer can halt the job when the estimation has reached a satisfactory level of confidence, à la CONTROL [1].

2. AN OPERATIONAL MODEL FOR OLA

2.1 Why an Operational Model?

The first step to getting OLA to work in a distributed, MapReduce environment is to define the abstract, operational model of the system. This model defines how data are processed in the system, and serves as a contract between the system implementors and the statistical analysis that underlies the OLA estimation process.

This operational model must meet two requirements:

1. First, the model must be amenable to statistical analysis. That is, at any point during the computation, it must be possible to take a snapshot of the system and to use that snapshot to predict the final output of the MapReduce program.
2. Second, the model must be amenable to implementation. It should impose little or no overhead on the system, so that there is little additional cost associated with running a Map Reduce OLA program, compared to a non-OLA MapReduce program. It should allow the actual implementation the freedom to deal with problems such as dead or slow machines and fluctuating resource availability, as well as allowing the implementation to take into account the physical placement of data in the system when assigning data to a CPU for processing. Furthermore, the model must be easy to implement, requiring little in the way of change or modification to the system it is built upon.

As discussed in the introduction to the paper, we note that that the "classic" OLA operational model (where data are simply processed in random order) is not directly applicable, because it ignores the "inspection paradox."

2.2 Our Model for OLA Over MapReduce

As such, we must define a somewhat more complicated operational model, whose key ideas are as follows.

We assume that data have been organized into storage units that we refer to as *blocks*. A "block" is nothing more than an arbitrary subset of the data in the system; typically, we would expect a block to contain tens or hundreds of megabytes of data. We allow for the possibility that the data may have been organized into blocks in an adversarial fashion that the OLA software cannot control (that is, some blocks may be very large or very small, or may contain all of the data with the greatest aggregate values). However, all of the methods we describe in the paper are also compatible with the case where the data have been placed into blocks in a fully random fashion. In that case, the aggregate values associated with each block would likely have low variance, and so our methods will converge much more quickly than if packed in an adversarial fashion. But our methods apply to both cases.

At the time that the OLA computation begins, all of the blocks are logically ordered in a statistically random fashion, into a single queue of blocks. We assume the existence of a `GetNext()` operation that iterates through the blocks in the queue in order.

As in other MapReduce implementations, we assume a central *scheduler* whose job is creating mappers and reducers, supplying them with data, and scheduling all of them on the physical system hardware. When the scheduler decides that there are enough system resources to process the next block, it makes a call to `GetNext()` to obtain the identifier for a random block. The scheduler must schedule the block given to it by `GetNext()`; it cannot schedule the blocks out of order. After some arbitrary delay, this block is assigned to a mapper, at which time the block is "processed" by the mapper. This processing includes dead time while the block is read from disk and transferred over the network, and it includes all of the necessary processing of the actual bytes in the block. Once the scheduler has assigned a block to some mapper, it then calls `GetNext()` to obtain another unprocessed block to assign to another mapper. We assume that the scheduler only assigns one block at-a-time to each mapper, so that the processing times are independent across each mapper.

Although the scheduler is not allowed to schedule blocks out of order, nor can it have more than one block that has been obtained from `GetNext()` that has not been scheduled, it may wait as long as it wants to call `GetNext()`, and it may also wait as long as it wants to schedule a block once it has been obtained by `GetNext()`. This flexibility is important because it allows the scheduler to wait for a physical mapper to become available that is located close to some physical copy of the block.

2.3 Taking a Snapshot

When it is time for the statistical analysis software to estimate the final answer to the query, a snapshot must be taken of the system. This snapshot consists of all of the statistics that will be used by the software to compute its estimate. These statistics are collected on a per block basis. For block i, they include:

1. The status of the block. This status is either `done` if the block has been fully processed, `processing` if the block is being processed by a physical mapper, or `unassigned` if the block has been obtained by `GetNext()` but is waiting to be assigned to a mapper.
2. If the block's status is `done`, then for each group in the block, the snapshot contains $x_{i,j}$, which is the value obtained by aggregating all records in the block that fall in group j.[1]
3. The snapshot contains t_i^{sch}, which is the time taken to assign the block to a mapper. If the block's status is `unassigned`, then a lower bound on t_i^{sch} is given; this is the time elapsed waiting for assignment.
4. If the block's status is not `unassigned`, then the snapshot contains the IP address (machine) of the mapper, as well as where the block was physically obtained from; this takes the value `local` (if the block was read from the same machine as the physical mapper), `rack` (if it was read from a different machine on the same rack), or `dist` (if it was read from a machine on a different rack).
5. If the block's status is not `unassigned`, the snapshot also contains the time t_i^{proc}, which is the time required to process the block. If the block's status is `processing`, then t_i^{proc} is the time taken since the mapper was first given the block by the scheduler.

Why this model? Why these statistics? We end this section by considering some of the intuition behind the operational model.

From an implementor's point of view, the model is compelling because it allows for a lot of freedom. The scheduler is able to process blocks on whatever machine it chooses. It can wait to schedule a block if no appropriate machine is available. It can ramp up the computation over time by adding more physical mappers, or ramp it down by simply not asking for blocks. The only real constraint is that when the scheduler makes a call to `GetNext()` to obtain a new block, it *must* assign the block it is given.

From a statistical point of view, the model has been designed with one singular goal in mind: at the time that a snapshot is taken, we wish to be able to (reasonably) view the x_i values associated with each of the blocks that have been received by some call to `GetNext()` as a set of independent, identically distributed (iid) samples from a random variable with distribution function $f(x_i)$. As with any finite population, by randomly permuting the population and then traversing the items in order, we produce a set of (approximately) iid samples from a distribution where:

$$f(x_i) = \frac{\sum_j I(x_i = x_j)}{n} \quad (1)$$

In this expression, n is the size of the population and the function I returns the value 1 if the boolean argument is true, and 0 otherwise.

[1]For simplicity and clarity in the rest of the paper, we will drop the j and assume that all measured quantities, times, and statistics refer to a single group for which we have decided to perform estimation. The extensions to the multi-group case are straightforward—they require collecting all of the statistics on a per-group basis—and will only serve to complicate our notation.

The word "approximately" is necessary only because of the small correlation induced by the fact that the population is finite.

If this were the entire story, then we would essentially be done: we would obtain a set of iid samples from $f(.)$, and hundreds of years of statistical theory would tell us exactly how to infer the various characteristics of $f(.)$ and estimate the final query result.

Unfortunately, an added complication is that we have the so-called "inspection paradox" to deal with. While the aggregate values associated with blocks that have *been obtained* by `GetNext()` can be seen as iid samples from $f(.)$, the aggregate values associated with the blocks that have *been fully processed* and have observable values cannot. That is, it may be the case that the time taken to schedule or to process a block is correlated with its contents. Thus, when we take a snapshot, some non-random set of the blocks returned by `GetNext()` may not yet have completed processing. For example, waiting for long time to schedule may, in fact, be the result of a block having a high aggregate value—the block may be in a busy part of the cluster and so it is difficult to schedule, but the reason that part of the cluster is busy could very well be that its blocks have more bytes, and hence higher aggregate values. Note that even in the case where blocks are uniformly sized, there may still be a correlation between processing time and aggregate value—imagine, for example, that the blocks on a slow machine tend to have a high aggregate value.

To take this into account, we allow for the scheduling and processing times to be correlated with the actual aggregate value, and we assume that the set of values associated with the blocks returned by `GetNext()` are samples from a three-dimensional distribution $f(x_i, t_i^{sch}, t_i^{proc})$.

By using this three-dimensional distribution function, it will allow us to make predictions about the x_i values that we have not seen, but for which we have information about t_i^{sch} and t_i^{proc}, and hence we can deal with the inspection paradox in a principled fashion. For example, if we have a particular block that has been processed for 125 seconds, where it took 5 seconds to schedule, we *can* correctly view x_i as a random sample from the distribution $f(x_i | t_i^{sch} = 5, t_i^{proc} \geqslant 125)$, thereby neutralizing the inspection paradox. This is precisely why all of the various timings are collected during the snapshot: they must be taken into account when estimates and confidence bounds are produced.

While this is a fairly thorough introduction to the intuition behind the model and the associated statistical considerations, the actual estimation process will be described in detail subsequently.

3. IMPLEMENTING THE MODEL

In this section we describe our implementation of the OLA model in Hyracks [2]. Hyracks is a new open source project that supports map and reduce operations, along with higher level relational operations such as filter (selection), projection, and join. The Hyracks architecture is similar to Hadoop—it has a single master node for submitting jobs (queries) and housing the task scheduler, which executes tasks on worker nodes running in the cluster. Hyracks tasks support read and write operations in HDFS [3], which we leverage to store the input to the map tasks and the output of the reduce tasks. Like Hadoop, when a client submits a MapReduce job, Hyracks assigns a single map task to a given block in the input data, and creates a configurable number of reduce tasks that are assigned specific groups using some partitioning function.

We modified the Hyracks implementation in two ways. First, we created a single queue containing the blocks in the input data. The order of the blocks in the queue is uniformly shuffled using the `java.util.Collections.shuffle` routine from the Java Standard Library. When Hyracks schedules a map task, it assigns the

current block at the head of the queue. The map task's execution time includes the time to obtain its assigned block from HDFS, the execution of the map function on each input record, and the execution of the combiner on the complete map function output. In this work we ignore performance issues involving locality; although we do account for block locality in our model. In future work, we plan on investigating locality scheduling techniques reminiscent to Delay Scheduling [18].

Our second modification involves running the estimator in the reduce task during the shuffle phase. In the shuffle phase, the reduce task is continuously receiving the output of completed map tasks. The output of a map task includes a *data* file containing the groups assigned to the reduce task and a *meta-data* file containing timing and locality information. If the map output contains no groups for a given reduce task then an empty data file is given along with a complete meta-data file. The meta-data file contains the block identifier, the time it took to schedule the block and the block locality relative to the map task execution: machine-local, rack-local, or distant. Also included is the map task IP address, start time and end time. Finally, we include the time when the estimator is called on the reduce task. The reduce task executes the estimator when it receives a new map output. The location of all data and meta-data files received thus far is given to the estimator when it is called. After the estimator completes, its output can be written to HDFS or forwarded to a downstream operator using the user-defined call-back class described in the introduction.

One issue that caused us some headache during the debugging of our system is that a reasonably synchronized global time must be maintained for the system. Since block processing times are typically on the order of minutes, this synchronization need only be accurate to within a few seconds. But a significant drift can indeed cause problems. The reason is that when a block arrives at the reducer, the total processing time is computed by subtracting the time that the block is received from the time that estimation began. Likewise, at estimation time, the reduce task performing the estimation must subtract the start time for the block from the current time to obtain a lower bound on the total processing time for the block. Due to the way we implemented this originally, these lower bounds were not consistent with the total processing time recorded for each block—the bounds tended to be much too large. Since a correlation between processing time and aggregate value had been observed, the result was that at estimation time the system "guessed" that the aggregate value for these unfinished blocks was very large, and significant over-estimates routinely occurred.

4. ESTIMATION

In this section, we consider how estimates and confidence bounds for those estimates can be obtained. As intimated previously, this is a challenging problem, as we must take into account processing times as well as observed aggregate values in order to circumvent the inspection paradox.

4.1 Overview

We will apply a Bayesian approach for estimation [13]; for brevity, this section will assume that the reader has some very basic familiarity with Bayesian statistics. The Bayesian approach has several obvious benefits for this particular problem. Most significant is the fact that the inspection paradox "goes away" under the Bayesian approach if one takes into account the time spent waiting for each block to be processed as observed data.

In standard Bayesian fashion, we will first describe a stochastic, parametric process that we imagine was used to produce the "observed" as well as the "hidden" data. The "observed data" will collectively be referred using variable $\mathbf{X}$. This set includes all of the known aggregate values and processing times. Our generative process will also produce a set of unobserved variables collectively referred to as Θ. Θ includes any data that is unobserved (for example, the processing time for a block that has not yet finished)—this data is collectively referred to as $\mathbf{Y}$—as well as any unknown parameters required by the generative process (for example, the mean aggregate value per block). In Bayesian fashion, we will then attempt to infer the distribution $P(\Theta|\mathbf{X})$, which is referred to as a *posterior distribution* for Θ. Then, given $\mathbf{X}$ as well as $P(\Theta|\mathbf{X})$, it is possible to obtain a posterior distribution over the actual query result, which can be used to obtain confidence bounds that are reported to the user.

Note that the discussion in this section is directly applicable only to `SUM` and `COUNT` queries, which are both evaluated by simply summing x_i values (in the `SUM` case, x_i will contain the total aggregate value for the block, and in the `COUNT` case, x_i will contain the tuple count for the block). Extensions to other aggregates such as `AVG`, `VARIANCE` and `STD_DEV` are straightforward; in general they require that we maintain zeroth, first and second moments for each block[2].

4.2 Generative process

To obtain the data that we must analyze to produce estimates and confidence bounds, we imagine that the following steps are repeated, once for each of the n blocks in the system:

1. $\mathbf{Z}_i \sim \text{Normal}(\mu, \Sigma)$
2. $(\mathbf{X}_i, \mathbf{Y}_i) \leftarrow \text{PostProcess}(\mathbf{Z}_i)$

"$\sim$" should be read as "is sampled from". After this process has been repeated n times (once for each block)—our goal is then to infer the posterior distribution for Θ using $\mathbf{X}$.

This process requires some additional explanation. We begin by describing the vector $\mathbf{Z}_i$. If there are m machines being used to execute a query, we imagine that associated with the ith block is a vector $\mathbf{Z}_i$ with $3m + 2$ entries, which contains both observed and hidden data. $\mathbf{Z}_i$ takes the form:

$$\mathbf{Z}_i = \langle x_i, t_i^{sch}, t_{i,1}^{loc}, t_{i,1}^{rack}, t_{i,1}^{dist}, t_{i,2}^{loc}, t_{i,2}^{rack}, t_{i,2}^{dist}, \ldots t_{i,m}^{loc}, t_{i,m}^{rack}, t_{i,m}^{dist} \rangle$$

The vector has the following components:

1. x_i is the value that is obtained when the block is aggregated.
2. t_i^{sch} is the time required to schedule the block, once it has first been selected for scheduling.
3. $t_{i,j}^{loc}$ is the time taken to actually process the block by a mapper on machine j, given that the block is to be read locally from machine j.
4. $t_{i,j}^{rack}$ is the time taken to process the block by a mapper on machine j, given that the block is to be read from a machine on the same rack as machine j.
5. $t_{i,j}^{dist}$ is the time taken to process the block by a mapper on machine j, given that the block must be read from a machine on a different rack.

[2]The zeroth, first and second moments are count, sum and sum of squares respectively.

Note that $\mathbf{Z}_i$ has $(3m + 2)$ dimensions, rather than the three dimensions one might expect after reading Section 3 of the paper. The reason is that we do not have a single processing time distribution; rather, we have $3m$ such distributions, with three distributions for each machine, depending on where the actual data comes from. This provides for a very fine-grained model, where processing times can differ from machine to machine.

Also note that we assume that $\mathbf{Z}_i$ is normally distributed, with mean vector μ and covariance matrix Σ. At first glance, assuming normality may seem questionable, but in practice this is not a particularly significant assumption because we are aggregating over many $\mathbf{Z}_i$ values—one for each block. Assuming normality here is similar to appealing to the Central Limit Theorem [12] when applying more traditional, non-Bayesian methods.

Finally, note that in step (2) of the generative process, $\mathbf{Z}_i$ is "post-processed" to actually produce the observable data $\mathbf{X}_i$ that is associated with the ith block. This removes the data from $\mathbf{Z}_i$ that could not/should not be observed, and puts this unobservable data into $\mathbf{Y}_i$. For example, given that each block is processed only once, no one is ever going to observe both $t_{i,1}^{loc}$ and $t_{i,5}^{loc}$ for a given block—we might imagine that both values exist, but they will never be observed together. Hence, $\mathbf{X}_i$ will never contain both of these values, and one or the other must end up in $\mathbf{Y}_i$.

In fact, there are four different ways in which the "post - processing" will be performed, depending upon the state of block i at the time that the estimation is performed:

i **in case 1: (No information)** $\mathbf{X}_i = \langle\rangle$
In this case, the block has not been chosen by the scheduler and so no information is available. $\mathbf{X}_i$ is empty, and $\mathbf{Y}_i = \mathbf{Z}_i$.

i **in case 2: (Scheduling)** $\mathbf{X}_i = \langle \lfloor t_i^{sch} \rfloor \rangle$
In this case, the block is at the head of the scheduler's queue and is waiting for a map task to be assigned to it. Thus, we have a lower bound on the scheduling time, denoted by $\lfloor t_i^{sch} \rfloor$. This is simply the amount of time the block has been waiting to be scheduled. Again in this case, $\mathbf{Y}_i = \mathbf{Z}_i$.

i **in case 3: (Scheduled and processing)** $\mathbf{X}_i = \langle t_i^{sch}, \lfloor t_{i,W_i}^{L_i} \rfloor \rangle$
In this case, a map task has been assigned to the block and processing has begun. Thus, we have access to an exact value for t_i^{sch}. We also have W_i, which is the identity of the machine on which the block is being processed, and L_i, which is the locality information for the block (loc, $rack$, or $dist$). Finally, we know $\lfloor t_{i,W_i}^{L_i} \rfloor$, which is a lower bound on the processing time for the block—one can view $t_{i,W_i}^{L_i}$ as being equivalent to t_i^{proc} from Section 3. In case 3, $\mathbf{Y}_i$ contains everything in $\mathbf{Z}_i$ except for t_i^{sch}.

i **in case 4: (Scheduled and processed)** $\mathbf{X}_i = \langle x_i, t_i^{sch}, t_{i,W_i}^{L_i} \rangle$
In this case, the map task has finished processing the data and the aggregate value has finally arrived at the reducer. Hence, in addition to W_i and L_i, we know exact values for the scheduling time, the processing time, and the aggregate value for the block. Here, $\mathbf{Y}_i$ contains everything in $\mathbf{Z}_i$ except for the three values in $\mathbf{X}_i$.

4.3 Prior Distributions

To make our model fully Bayesian, we must supply priors on μ and Σ. In our implementation, each $\mu_k \sim \text{InvGamma}(1, 1)$ (where k refers to the kth dimension in $\mathbf{Z}_i$). The inverse Gamma distribution is a standard, uninformative prior for values that must be non-negative—it makes sense to have non-negative means for all of the time values in the $\mathbf{Z}_i$ vector. It will also usually make sense to have a non-negative mean for x_i; if not, then another suitable, uninformative prior can be used.

Handling the covariance matrix Σ is a bit trickier. The standard prior distribution for a covariance matrix is the inverse Wishart distribution, because it is "conjugate" for the normal. This means that under certain conditions, upon observing the output from a normal distribution with an inverse Wishart prior on the covariance, the posterior on the covariance is still inverse Wishart. Conjugacy is convenient because it can make inference much easier. Unfortunately, these "certain conditions" are not met in our application because we do not always have actual observations from the normal—we may only know, for example, that the processing time has a lower bound (if we are in "case three" from the previous subsection). Thus, we choose to use an application-specific prior that is easily factorizable; that is, where we can easily write the marginal distribution for each entry in the covariance matrix. This makes deriving a Gibbs sampler for inference much easier (see the next subsection). Specifically, we let $\sigma_k \sim \text{InvGamma}(1, 1)$, where $\Sigma_{k,k} = \sigma_k^2$. Then, we assume that the following process is used to generate the rest of Σ:

while *true* **do**
  for k_1 = *1 to (3m+2)* **do**
    for k_2 = k_1 + *1 to (3m+2)* **do**
      $\rho_{k_1,k_2} \sim \text{GenBeta}(-1, 1, 1, 1)$;
      $\Sigma_{k_1,k_2} = \Sigma_{k_2,k_1} = \rho_{k_1,k_2} \times \sigma_{k_1} \times \sigma_{k_2}$;
  if Σ *is positive-definite* **then**
    break;

Algorithm 1: Generation of the covariance matrix Σ

Here, GenBeta$(-1, 1, 1, 1)$ refers to a generalized Beta$(1, 1)$ distribution, stretched to cover the range from -1 to 1 (rather than the usual 0 to 1). What this process does is to essentially sample a correlation ρ for each of the pairs of variables in $\mathbf{Z}_i$, and to then check whether a valid covariance matrix has been obtained (one that is positive definite). If it has not, then the whole process is repeated again. The PDF for Σ can then be written as:

$$P(\Sigma) \propto \begin{cases} 0 \text{ if } \Sigma \text{ is not positive-definite} \\ \left(\begin{array}{c} \prod_k \text{InvGamma}(\sigma_k|1,1) \times \\ \prod_{k_1,k_2} \text{GenBeta}(\rho_{k_1,k_2}|-1,1,1,1) \end{array} \right) \text{otherwise} \end{cases} \tag{2}$$

4.4 Posterior Distribution

In this subsection, we tackle the problem of obtaining a formula for the desired posterior distribution, $P(\Theta|\mathbf{X})$. Recall that $\mathbf{X} = \bigcup_i \{\mathbf{X}_i\}$, and the unobservable data set Θ contains $\mathbf{Y} = \bigcup_i \{\mathbf{Y}_i\}$, as well as the normal parameters μ and Σ.

From elementary probability, we know that:

$$P(\Theta|\mathbf{X}) = \frac{P(\mathbf{X}|\Theta)P(\Theta)}{P(\mathbf{X})} \tag{3}$$

This means that there are three quantities that we must derive expressions for: $P(\mathbf{X}|\Theta)$, $P(\Theta)$, and $P(\mathbf{X})$.

We deal with $P(\mathbf{X}|\Theta)$ first. From the generative process, we know that $P(\mathbf{X}|\Theta) = \prod_i P(\mathbf{X}_i|\Theta)$. We can easily write an expression for each $P(\mathbf{X}_i|\Theta)$, which will depend upon the case that holds for block i:

i **in case 1: (No information)** $P(\mathbf{X}_i|\Theta) = P(\langle\rangle|\Theta) = 1$ since $\mathbf{X}_i$ is empty.

i **in case 2: (Scheduling)** Here, we have only a lower bound on the scheduling time. Thus, $P(\mathbf{X}_i|\Theta) = P(\langle \lfloor t_i^{sch} \rfloor \rangle|\Theta) = 1$ if $t_i^{sch} \geqslant \lfloor t_i^{sch} \rfloor$, and 0 otherwise since this is impossible.

i **in case 3: (Scheduled and processing)** In this case, $P(\mathbf{X}_i|\Theta) = P(\langle t_i^{sch}, \lfloor t_{i,W_i}^{L_i} \rfloor \rangle|\Theta) = \text{Normal}(t_i^{sch}|\mu, \Sigma, \mathbf{Y}_i)$ if $t_{i,W_i}^{L_i} \geqslant \lfloor t_{i,W_i}^{L_i} \rfloor$, and 0 otherwise since this is again impossible.

i **in case 4: (Scheduled and processed)** Here, we evaluate a normal distribution: $P(\mathbf{X}_i|\Theta) = P(\langle x_i, t_i^{sch}, t_{i,W_i}^{L_i} \rangle|\Theta) = \text{Normal}(x_i, t_i^{sch}, t_{i,W_i}^{L_i}|\mu, \Sigma, \mathbf{Y}_i)$.

Now, we move onto deriving an expression for $P(\Theta)$. From the last few subsections, we have:

$$\begin{aligned} P(\Theta) &= P(\mathbf{Y}|\mu, \Sigma)P(\mu)P(\Sigma) \\ &= P(\mu)P(\Sigma)\prod_i P(\mathbf{Y}_i|\mu, \Sigma) \\ &= P(\Sigma)\prod_j \text{InvGamma}(\mu_j|1, 1)\prod_i \text{Normal}(\mathbf{Y}_i|\mu, \Sigma) \end{aligned} \tag{4}$$

where an explicit formula for $P(\Sigma)$ was given previously.

This gives us expressions for $P(\mathbf{X}|\Theta)$ and $P(\Theta)$. In standard Bayesian fashion, we ignore $P(\mathbf{X})$, which would be very difficult to compute since it would involve integrating over Θ. But since $P(\mathbf{X})$ does not depend upon Θ, it is merely a normalizing constant that is necessary for the total mass of $P(\Theta|\mathbf{X})$ to be one, and is not needed to compare the relative merits of candidate Θ values.

4.5 Putting It All Together

Since our goal is to produce estimates and confidence bounds for the actual query result, we are not interested in the posterior distribution $P(\Theta|\mathbf{X})$ for its own sake. Rather, we will use $P(\Theta|\mathbf{X})$ to produce estimates and confidence bounds for the answer.

To describe how this is done, note that given a possible value for Θ—combined with the visible data $\mathbf{X}$— we have access to each and every x_i value in the database. Thus, given a particular Θ as well as $\mathbf{X}$ it is very easy to compute the query answer as:

$$Q(\Theta, \mathbf{X}) = \sum_i x_i \tag{5}$$

Then by integrating $P(\Theta|\mathbf{X})$ over all possible Θ, we obtain various statistics describing the eventual query result. For example, the following gives us the expected value of the query result:

$$\int_\Theta P(\Theta|\mathbf{X})Q(\Theta, \mathbf{X})d\Theta$$

And we can obtain the lower end l for a 95% confidence bound on the query result by computing Λ and l so that:

$$\begin{aligned} &\int_{\Theta\in\Lambda} P(\Theta|\mathbf{X})d\Theta = 0.025 \text{ where} \\ &\mathbf{max}_{\Theta\in\Lambda}\{Q(\Theta, \mathbf{X})\} \leqslant l \text{ and } \mathbf{min}_{\Theta\in\bar{\Lambda}}\{Q(\Theta, \mathbf{X})\} \geqslant l \end{aligned} \tag{6}$$

The upper end could be computed in a similar fashion.

Unfortunately, performing this sort of computation exactly is difficult. The difficulty is often circumvented using so-called "Markov Chain Monte Carlo" (MCMC) methods [15] that sample directly from a distribution such as $P(\mathbf{X}|\Theta)$. In our case, we apply a particular MCMC method called a *Gibbs sampler* to the problem [4]. The samples obtained from a Gibbs sampler are easily used to compute expected value and confidence bounds. For example, we can run the sampler to produce several hundred candidate Θ values, and then average the associated query results—this is equivalent to the expected value computation described above. Cutting off the top 2.5% and the bottom 2.5% of the set of query results, and then taking the highest and lowest remaining results, gives 95% confidence bounds on the query answer.

5. EXPERIMENTS

In this section, we describe a set of experiments on the software that we have developed. Our experiments are designed to answer the following questions: Can the confidence bounds that our system reports be trusted? How important is it to take into account the correlation between processing time and data value in both synthetic and real data? How important is choosing blocks in a statistically randomized order? In a realistic setting, is the system able to produce accurate results quickly?

5.1 Experiment One

Basic Setup. In the first experiment, we run our version of Hyracks with OLA over six months of data from the Wikipedia page traffic data set (available at http://aws.amazon.com/datasets/4182), with the simple goal of counting the number of Wikipedia page hits on a per-language basis over those six months. Six months of Wikipedia data take up about 220GB (compressed), and are stored in 3,960 blocks. We run our software on an eleven node cluster, with one master node and ten slaves. Each machine has four disks, four cores running at 2.3 GHz, and 12GB of RAM. We use 80 mappers and ten reducers (with one reducer available for each of the ten languages that are to be counted). The entire MapReduce process takes approximately 46 minutes to run to completion.

To demonstrate the relative importance of the different components of our software, we run three different versions of our OLA software. The first uses every method described in the paper. In the second version, randomization of blocks is not performed by the scheduler, so blocks are scheduled in arbitrary (but non-random) order. In the third version, randomization is used, but the (possible) correlation between processing/scheduling time and the aggregate value is not taken into account by the system, leaving the software vulnerable to the "inspection paradox" described earlier.

Results. A subset of the observed results are given in Figures 1, 2, and 3. Figure 1 shows the posterior distribution of possible query results computed by our system, for the English language, at various times during the MapReduce task (we show results after 10% of the task is complete, after 20% is complete, after 30%, and so on). Two posterior distributions are plotted: one computed running the first version (all features from the paper), and a second computed running the second version (no randomization). Each plot also shows the true query result for the English language. Figure 2 is similar to Figure 1, but it shows the results for the French language. Figure 3 also shows results for the French language, but it shows the computed posterior distribution for the query result after a much smaller portion of the task has completed: 1%, 2%, 3%, and so on. This plot also shows both version one of our software, and version three (randomization, but no correlation).

Discussion. It is clear that without randomization, severe bias is possible, and so confidence bounds obtained without a random scheduling order are useless. With randomization, the confidence bounds from Figures 1 and 2 seem remarkably accurate. It is significant that the bounds obtained are quite narrow, very quickly. Take the English language. After 10% of the blocks have been processed, the bounds go from approximately 4.1 to 4.4×10^{10}, which represents a possible error of only $\pm 3\%$. For many applications, it may be acceptable to simply kill the computation with this level of accuracy. In general, convergence could be made to happen even more quickly by increasing the number of blocks used to store the same data set, though this could have a negative effect on the overall processing time of the MapReduce job. Even for this particular data set (where the correlation between processing time and aggregate value is quite weak) there is still a clear benefit to taking into ac-

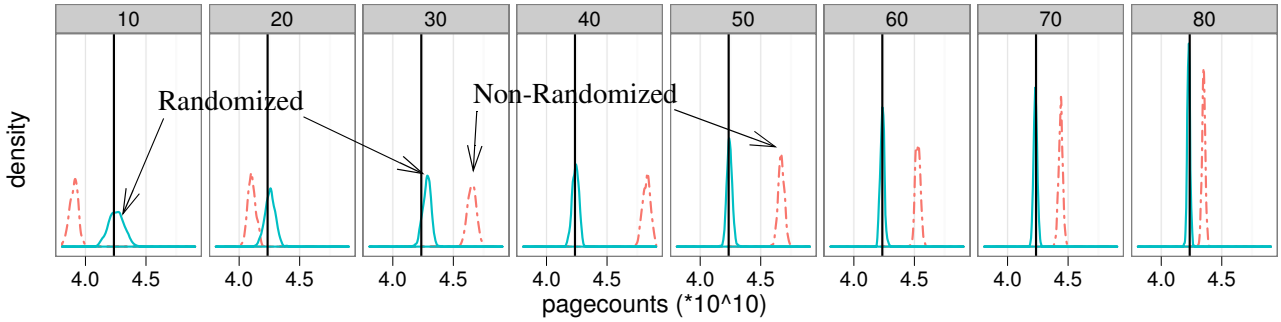

Figure 1: Posterior query result distribution for number of Wikipedia page hits over the English language, at various query completion percentages, using both a randomized and arbitrary block ordering. The actual query result is a vertical black line.

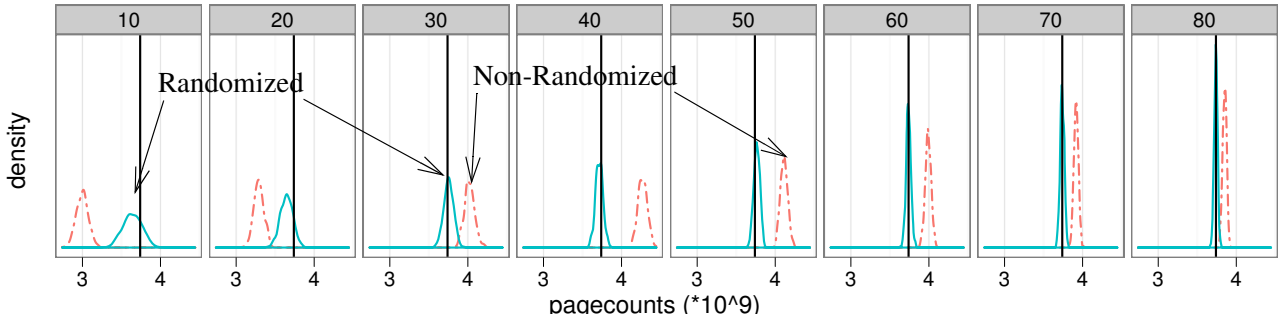

Figure 2: Identical to Figure 1, except for the French language.

count the correlation, particularly when only a very small fraction of the data has been processed. Consider the plots corresponding to finishing 3% and 4% of the MapReduce job. Without correlation, the posterior distribution almost totally misses the actual query result. Taking into account the correlation, the distribution is neatly bisected by the correct result.

5.2 Experiment Two

Basic Setup. This experiment is somewhat similar to the first, except that our goal is to try to determine, in a systematic fashion, how accurate the computed posterior distribution is when used to compute 95% confidence bounds. Since testing accuracy requires many, many repetitions of the MapReduce task, instead of actually running the task in a real cluster, we use a simulator. In our simulation, a random aggregate value is associated with each block, and random processing times are associated with each block as well; the correlation between aggregate value and processing time is set to be 0.7. Under the same setup as above (80 mappers, 3,960 blocks), we repeat the MapReduce and the estimation process 100 times. Every time that the estimation is re-run, we consider several different task-completion percentages (1% done, 2% done, 3% done, and so on). At each task-completion percentage, we use the computed posterior distribution to obtain 95% confidence bounds by "chopping off" the top 2.5% and bottom 2.5% of the posterior distribution. We then determine whether or not the actual query result is within the 95% confidence bounds, and compute the fraction of the time that the query result is not within the 95% confidence bounds over each of the 100 repetitions. If our estimation process worked perfectly, then the 95% confidence bounds would cover the actual answer 95% of the time, with a 5% error rate. This whole process is repeated twice: once using the full estimation process, and a second time ignoring the possibility of correlation between processing time and aggregate value.

	Percentage of MapReduce task complete						
	2%	3%	4%	5%	10%	20%	30%
w corr	.03	.01	.06	.05	.05	.02	.05
w/o corr	.70	.63	.62	.61	.37	.22	.13

Table 1: Fraction of confidence bounds that are "incorrect".

Results. The results are given in Table 2, and are mostly self-explanatory. For each of the listed task-completion percentages, the fraction of "incorrect" 95% confidence intervals is given, for both of the software versions.

Discussion. As can clearly be seen, the confidence bounds computed when taking into account correlation are accurate, coming very close to the expected 5% throughout query execution. On the other hand, the results obtained without taking into account the correlation are very poor, particularly when only a small fraction of the MapReduce task has been completed. This mirrors the results shown in Figure 2, but under much more extreme circumstances.

6. RELATED WORK

OLA has been studied for some time in the context of classic, SQL databases [10, 9, 11, 1] and more recently for peer-to-peer systems [17]. But, in context of MapReduce, the only work that considers OLA (without ignoring MapReduce's open programmability and fault tolerance) is the Hadoop Online Prototype (HOP) system [5]. HOP supported OLA queries by executing reduce tasks at data dependent intervals e.g., on 10%, 20%,. . . ,90% of the data. The query estimate assumed a uniform sample of the input data but made no modifications to enforce this in the Hadoop scheduler. This led to significant error in their estimates. To compensate, the authors modified the query to contain extra parameters that in-

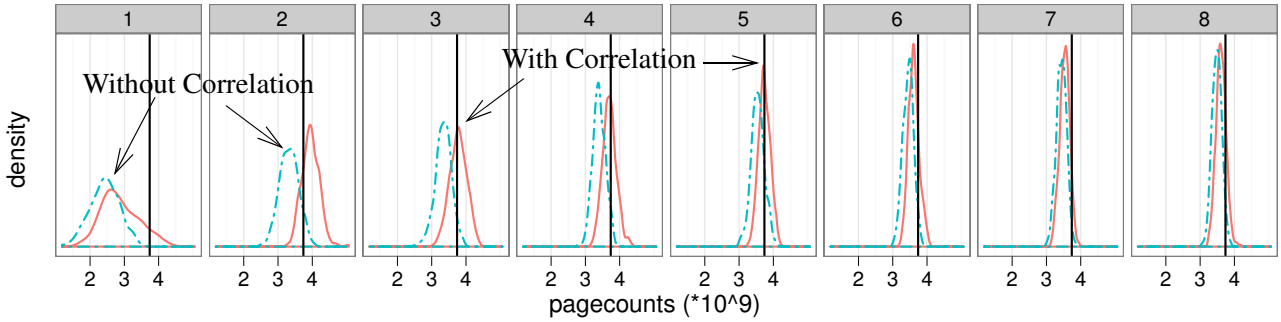

Figure 3: Posterior query result language for the French distribution, at various query completion percentages, taking into account and ignoring correlation between aggregate value and processing time.

dicated how many samples of a particular aggregate group were present, and scaled the estimate accordingly.

An alternative to OLA is precomputed synopsis, where the system uses summary statistics (computed prior to the execution of the query) to provide approximate answers [8, 14].

7. FUTURE WORK

Our current scheduling policy greedily assigns the block at the head of the randomized queue to the first mapper available. Clearly, this destroys locality and hence introduces overhead. However, since we ran our tests on a single rack cluster, a performance study measuring this overhead would be mostly uninformative. In subsequent work, we will study the performance impact by running the experiments on a multi-rack cluster.

Locality scheduling in the context of online aggregation is a direction that we plan to investigate further. Scheduling computation near the data is the primary optimization in today's MapReduce systems [7, 2]; today's schedulers care most about rack locality. Indeed at Yahoo!, the performance gains of machine to rack locality is negligible; others have also drawn this conclusion [18].

We would also like to take into account external constraints on the scheduler. For example, we may wish to schedule only those tasks from the highest priority jobs. Fortunately, it seems that we might be able to explore such issues without changing our statistical model for OLA, since our model allows for a block to be held for an arbitrary amount of time before it is scheduled. Our model allows for that scheduling time to be correlated with any properties of the block (such as the block's aggregate value, or its physical location). Thus, we can implement scheduling policies that hold the block at the head of the queue until a suitable location for processing the block opens up; our statistical model remains valid in this case.

8. CONCLUSION

Like the earlier works on Online Aggregation, we focus on single-table query plans involving "Group By" aggregations, which is precisely the workload targeted by MapReduce. The focus of our work here is to develop a model that accounts for biases that can arise when estimating aggregates in a cluster environment. This model allows us to export "early returns" of query aggregates that are statistically robust.

Acknowledgements: Pansare and Jermaine were supported by the NSF under award 1007062. Borkar was supported by the NSF under award 0910989 and by a Facebook Fellowship award. The idea for OLA over MapReduce came out of early discussions with Joe Hellerstein and Peter Haas. We would like to thank Peter Haas and the anonymous reviewers for providing useful suggestions.

9. REFERENCES

[1] R. Avnur, J. Hellerstein, B. Lo, C. Olston, B. Raman, V. Raman, T. Roth, and K. Wylie. Control: continuous output and navigation technology with refinement on-line. In *SIGMOD conference*, pages 567–569, 1998.

[2] V. R. Borkar, M. J. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE*, pages 1151–1162, 2011.

[3] D. Borthakur. *The Hadoop Distributed File System: Architecture and Design*, 2007.

[4] G. Casella and E. George. Explaining the gibbs sampler. *The American Statistician*, 46:167–174, 1992.

[5] T. Condie, N. Conway, P. Alvaro, J. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *NSDI Conference*, pages 21–21, 2010.

[6] D. Cox. *Renewal Theory*. Methuen and Co, New York, 1970.

[7] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI '04*, pages 137–150, December 2004.

[8] P. B. Gibbons and Y. Matias. New sampling-based summary statistics for improving approximate query answers. *SIGMOD Rec.*, 27:331–342, June 1998.

[9] P. Haas and J. Hellerstein. Ripple joins for online aggregation. In *SIGMOD Conference*, pages 287–298, 1999.

[10] J. Hellerstein, P. Haas, and H. Wang. Online aggregation. In *SIGMOD Conference*, pages 171–182, 1997.

[11] C. Jermaine, S. Arumugam, A. Pol, and A. Dobra. Scalable approximate query processing with the dbo engine. In *SIGMOD Conference*, pages 725–736, 2007.

[12] E. L. Lehmann and G. Casella. *Theory of Point Estimation*. Springer, second edition, 1998.

[13] A. O'Hagan and J. J. Forster. *Bayesian Inference*. Arnold, second edition, 2004.

[14] F. Olken and D. Rotem. Maintenance of materialized views of sampling queries. In *Proceedings of the Eighth International Conference on Data Engineering*, 1992.

[15] C. Robert and G. Casella. *Monte Carlo Statistical Methods*. Springer, second edition, 2004.

[16] C. Särndal, B. Swensson, and J. Wretman. *Model Assisted Survey Sampling*. Springer, New York, 1992.

[17] S. Wu, S. Jiang, B. C. Ooi, and K.-L. Tan. Distributed online aggregations. *Proc. VLDB Endow.*, 2:443–454, August 2009.

[18] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys*, pages 265–278, 2010.

APPENDIX

A. OVERVIEW OF MAP-REDUCE

MapReduce is a programming model for performing aggregate computations over large data sets. The programmer specifies a *map* function that processes input records and produces a list of intermediate key/value pairs, and a *reduce* function that is called once for each distinct map output key and associated list of intermediate values. Optionally, the programmer can supply a *combiner* function, which is applied to the intermediate results between the map and reduce steps. The combiner interface is similar to reduce functions. Combiners are typically used to perform "pre-aggregation," which can reduce the amount of network traffic when the map and reduce steps are executed in a distributed environment.

The MapReduce architecture consists of a query processing layer, that is based on a dataflow of map and reduce operations, and a *Distributed File System* (DFS), which stores the input to the map function and the output of the reduce function. The intermediate data is typically stored on the local file system. The query processing layer consists of a single *master* node and many *worker* nodes [7]. The master is responsible for accepting *jobs* that specify the user-defined functions and automatically parallelizing those functions into units of work called *tasks*. Each task is assigned a portion of the relevant input and is individually scheduled on the worker nodes. Worker nodes are assigned a fixed number of *slots* for executing tasks (e.g., two maps and two reduces). A heartbeat protocol between each worker and the master is used to update the masters' bookkeeping state of running tasks, and drive the scheduling of new tasks: if the master identifies free worker slots, it will schedule further tasks on the worker.

A map task is assigned a portion of the input file called a *split*. By default, a split corresponds to a single DFS block (64MB by default), so typically the total number of file blocks determines the number of map tasks. The execution of a map task is divided into two phases. The *map* phase reads the assigned split from the DFS, parses it into records (key/value pairs), and applies the map function to each record. After the map function has been applied to each input record, the *commit* phase executes the combiner function (if given) and registers the final output with the worker, which will then inform the master that the task has finished executing.

The execution of a reduce task is divided into three phases. In the *shuffle* phase, the reduce task receives its assigned key range from the output of each map task. This phase typically runs concurrently with the map tasks in a pipelined fashion. After receiving its partitions from all map tasks, the reduce task enters the *group* phase. This is commonly performed via sorting techniques. Finally, the *reduce* phase invokes the user-defined reduce function for each distinct key and associated list of values.

B. OLA PROGRAMMING INTERFACE

In this section we describe the interface that Hyracks exports for running online aggregation (OLA) queries. We begin with the programming interface, which is very similar to conventional Hadoop[3], except for three key differences:

1. The user's `Mapper`, `Reducer` and `Combiner` classes have to obey the contract given in appendix B.1.
2. The user can optionally write a class that implements the `EstimateReceiver` interface (see appendix B.4). This will be called whenever a new estimate is generated. We provide a default implementation of this interface that simply writes the estimates to HDFS.
3. The user also needs to specify few additional properties in the job configuration file (see appendix B.2 and B.3).

B.1 Map, Reduce and Combiner Contracts

In this subsection, we describe the input and output of the mapper, the combiner, and the reducer in our OLA implementation. Our system supplies default combiner and reducer implementations; the mapper is always user-supplied.

Currently, our OLA system designed to handle computations that are equivalent to the following SQL:[4]

```
SELECT AGG(g(y))
FROM MY_TABLE AS y
GROUP BY h(y)
```

To implement such a query, the user would first need to supply a mapper. The mapper would simply read in the data, and then output a stream of $\langle h(y), g(y)\rangle$ pairs.

Now consider the combiner. For the time being, we restrict ourselves to the case where `AGG` is `SUM`. Section 2.3 of the paper describes a number of statistics that must be collected by the system to perform OLA over such a query. The various timing values are hidden from the Hyracks programmer who uses our system, and are collected automatically. Thus, assuming for a moment that the system is restricted to handling `SUM` queries, the only one of these parameters that a Hyracks programmer would actually be concerned with producing is $x_{i,j}$. Recall that $x_{i,j}$ is the aggregate value for group j over block i, which would be produced (along with the group key j) by the combiner. Thus, the output of the combiner whose input *key* is j and input *values* are $\langle g(y_1), g(y_2), ...\rangle$, is:

$$x_{i,j} = \sum_k g(y_k) \tag{7}$$

along with the key for group j.

Likewise, the reducer accepts a set of $x_{i,j}$ values (all having the same group key j), aggregates them and then outputs a final aggregate value along with the group key.

In our current implementation, the group key type must be `Text`, and $x_{i,j}$ must be of type `Double`. These types are a bit restrictive, and we plan to generalize this in the future.

While the prior discussion is adequate to handle `SUM` queries, (and, if $x_{i,j}$ is binary, it suffices for `COUNT` queries), in order to support `AVG`, `VARIANCE` and `STD_DEV` we need some additional information. Note that $x_{i,j}$ is the so-called "first moment" of the distribution of the values in block i. To handle the additional aggregates, we generalize $x_{i,j}$ to $x^m_{i,j}$ so that:

$$x^m_{i,j} = \sum_k g^m(y_k), \tag{8}$$

where $x^m_{i,j}$ denotes the mth moment of the values in the block. To support the full set of five aggregate functions described above, we need the zero-th, first, and second moments of each block. Thus, the user-supplied combiner does not only output $x_{i,j}$ (which is equivalent to $x^1_{i,j}$), it must actually output the triple $\langle x^0_{i,j}, x^1_{i,j}, x^2_{i,j}\rangle$ in a comma separated vector (CSV) record format. This triple is then used by the reducer to produce the final aggregate.

[3]We will use the terms *Hadoop* and *Hyracks* interchangeably since Hyracks supports the Hadoop API.

[4]Note that while this query does not contain an explicit `WHERE` clause, a boolean predicate can implicitly be present in $g(y)$ by making this function return zero if the record in question does not match a selection condition. We also plan to generalize beyond such simple queries to support multiple aggregation operations, as well as joins.

	Mapper	Combiner	Reducer
Input key	N/A	j	j
Input value	N/A	y_k	$\langle x^0_{i,j}, x^1_{i,j}, x^2_{i,j} \rangle$
Output key	j	j	j
Output value	y_k	$\langle x^0_{i,j}, x^1_{i,j}, x^2_{i,j} \rangle$	Final value

Table 2: `Mapper`, `Reducer`, `Combiner` class interfaces.

The inputs and outputs for the mapper, combiner, and reducer are summarized in the table 2.

B.2 Specifying the aggregate

Internally, our OLA library provides implementations of the combiner and reduce functions for the five aggregates described previously. We provide a helper class `OAJobConfigurer` that allows the user to indicate the aggregate choice, and which asks the Hyracks engine to use the default implementation of combiner and reduce functions for that aggregate. Use of this class is demonstrated in the following code snippet:

```
Job job = new Job();
job.setMapperClass(mapperClass);
OAJobConfigurer.configure(job, "SUM");
...
master.submitJob(job);
```

Listing 1: Usage of OAJobConfigurer class.

The first line creates a new job configuration object, which is submitted to the master scheduler by the last line. The configuration parameters are set by the client prior to the final submission of the job. Here, we select the (user-defined) mapper function implementation and also the (default) `SUM` aggregate implementation from the OLA library. It should be noted that the clients can still implement their own reduce (and combiner) function as long as it conforms to contract given in the appendix B.1. This is useful, for example, when one needs to implement a `HAVING` clause for a group-by aggregate.

B.3 Configuration parameters

Next, we describe the configuration parameters exported by the system. These parameters must be set by the client prior to the final job submit call to the master scheduler. Our goal is to give confidence bounds of the form "with probability p, the actual query answer is within the range from low to $high$", where the bounds are updated each ms milliseconds during the execution of the query.[5]

The user can specify this using the following properties in the job configuration file:

1. `mapreduce_online.estimation_interval` = ms
   This property specifies the number of milliseconds after which the Bayesian estimator should be called by the Hyracks engine. If the value is large, the estimator is rarely called and it improves the performance of map-reduce job. This is because the Bayesian estimator is CPU intensive code and will compete with the map-reduce workers for CPU cycles. On the other hand, if the value is low, the estimator is effectively called after the processing of every block, hence giving more estimates to the user.
2. `mapreduce_online.confidence_interval` = $range$
   Another way to improve the performance is to ask the Hyracks engine not to call the estimator code once a desired accuracy is reached. That is, estimation is stopped once the width of the interval returned by the Bayesian estimator is less than or equal to the specified range. To disable this feature (i.e. to obtain estimates until the end of the map-reduce job irrespective of accuracy), the user has to set this value to `0`.
   A feature slated for future work, which could potentially reduce the cost of estimation further, is to allow the user to switch the scheduling policy from random to a more conventional locality-based scheduling, once all the groups have reached the desired accuracy.
3. `mapreduce_online.confidence_level` = $conf$
   The system defaults to $conf = 95\%$.
4. `mapreduce_online.reduce_operator` = `AGG`
   Any one of the five aggregates defined previously can be used for `AGG`.
5. `mapreduce_online.estimate_receiver.class` = `RC`
   `RC` gives the name of the class that will be called back by the Hyracks engine to communicate the estimates with the user. This class has to implement the `EstimateReceiver` interface, described in the appendix B.4.

[5]The probability p is fixed during the execution of the query, and the estimator outputs the confidence interval $[low, high]$ based on the value of p.

B.4 EstimateReceiver interface

Whatever mapper, combiner, and reducer are specified by the user, the output of the various combiners is sent both to the user- or system-supplied reducer, and to our Bayesian estimation code which uses the combiner-supplied moments to estimate the final answer to the specified aggregate query. Somehow, the system needs to do something with these estimates. It is the `EstimateReceiver` class that is tasked with accepting an estimate from the OLA engine and processing it as necessary. The `EstimateReceiver` class , for example, might write the estimate to a file, or it might display the result graphically in a web page so that the user can see the progression of the computation.

The interface for this class is:

```
public interface EstimateReceiver {
  public void Open();
  // [low, high] is the confidence interval
  public void NextGroup(Text key,
    Double estimate, Double low, Double high);
  public void Close();
  public void NoMoreBlocks();
}
```

Listing 2: EstimateReceiver interface.

Whenever the estimator finishes processing, the Hyracks engine calls `Open` function and then calls `NextGroup` for each group. Finally, it calls `Close` to notify that there are no more groups found in the current estimation cycle. If the job is completed (i.e. there are no more blocks remaining to be processed) or if all the groups have reached desired accuracy, the Hyracks engine will call `NoMoreBlocks` method.

B.5 Example

To make this interface concrete, we describe the steps involved to implement the following simple SQL query:

```
SELECT SUM(page_hits)
FROM WIKIPEDIA_LOG
GROUP BY LANGUAGE
```

The above SQL query uses the Wikipedia page traffic data set, that stores each log entry in CSV format. The two fields in the log that we are interested in, are the language of the Wikipedia page

pointed to by that entry (LANGUAGE) and the number of times that page has been viewed (page_hits). The above query returns the total number of page hits for each language in the Wikipedia data set. This query can be implemented by a single MapReduce job.

To implement this query, the user-supplied map function parses the data, and then outputs the language (the key) and the number of page hits. The reduce function accepts a single language and list of page hit values. It sums the page hit values and writes a result (key-value) record to the output. Finally, the WikipediaMain class stitches the job together and submits it to the master scheduler for execution.

```
public class WikipediaMapper extends
  Mapper<LongWritable, Text, Text, DoubleWritable> {
  // Line number is the key and
  // the content of the line is the value
  public void map(LongWritable key, Text value, Context
       ctx)
    throws IOException, InterruptedException {
      // LANG and PAGE_HITS are integer constants
      // that depend on the format of the log entry
      Array[String] fields = value.split(",");
      ctx.write(fields[LANG],
  Double.parseDouble(fields[PAGE_HITS]));
  }
}

public class WikipediaCombiner extends
  Reducer<Text, DoubleWritable, Text, Text> {
    public void reduce(Text key,
      Iterable<DoubleWritable> values, Context ctx)
      throws IOException, InterruptedException {
        double sum = 0;
        for (DoubleWritable val : values) {
            sum += val.get();
        }
        ctx.write(key, new Text("," + sum + ","));
    }
}

public class WikipediaReducer extends
  Reducer<Text, Text, Text, DoubleWritable> {
    public void reduce(Text key,
      Iterable<DoubleWritable> values, Context ctx)
      throws IOException, InterruptedException {
        double sum = 0;
        for (Text val : values) {
    Array[String] moments = val.toString().split(",");
    sum += Double.parseDouble(moments[1]);
        }
        ctx.write(key, new DoubleWritable(sum));
    }
}

public class WikipediaMain {
  public static void main(String[] args)
    throws Exception {
    ...
    Job job = new Job();
    job.setMapperClass(WikipediaMapper);
    job.setCombinerClass(WikipediaCombiner);
    job.setReducerClass(WikipediaReducer);
    ...
    master.submitJob(job);
  }
}
```

Listing 3: Sample Hadoop implementation of page-hit query.

To obtain online estimates for the same program, the user needs to modify the WikipediaMain class to specify the aggregate function (as discussed in appendix B.2) and set the job configuration properties (as discussed in appendix B.3). Note, that using this default configuration, the user need not write the reducer and the combiner class.

The user also needs to specify an extra class WikipediaEstimateReceiver that handles the estimates returned by the OLA library.

```
public class WikipediaMain {
  public static void main(String[] args)
    throws Exception {
    ...
    // Set job configuration properties
    Configuration conf = new Configuration();
    conf.setProperty("mapreduce_online.reduce_operator",
         "SUM");
    conf.setProperty("mapreduce_online.estimate_receiver.
         class", WikipediaEstimateReceiver);

    // Specify aggregate
    Job job = new Job(conf);
    job.setMapperClass(WikipediaMapper);
    OAJobConfigurer.configure(job, "SUM");
    ...
  }
}

public class WikipediaEstimateReceiver
  implements EstimateReceiver {
  ...
  public void NextGroup(Text key, Double estimate, Double
       low, Double high) {
    // Do something with estimate
    // Example: display it to a GUI
  }
}
```

Listing 4: OLA implementation of page-hit query.