Halt or Continue: Estimating Progress of Queries in the Cloud

Yingjie Shi, Xiaofeng Meng, and Bingbing Liu

School of Information, Renmin University of China, Beijing, China
shiyingjie1983@yahoo.com.cn, {xfmeng,liubingbing}@ruc.edu.cn

Abstract. With cloud-based data management gaining more ground by day, the problem of estimating the progress of MapReduce queries in the cloud is of paramount importance. This problem is challenging to solve for two reasons: i) cloud is typically a large-scale heterogeneous environment, which requires progress estimation to tailor to non-uniform hardware characteristics, and ii) cloud is often built with cheap and commodity hardware that is prone to fail, so our estimation should be able to dynamically adjust. These two challenges were largely unaddressed in previous work. In this paper, we propose PEQC, a Progress Estimator of Queries composed of MapReduce jobs in the Cloud. Our work is able to apply to a heterogeneous setting and provides a dynamically update mechanism to repair the network when failure occurs. We experimentally validate our techniques on a heterogeneous cluster and results show that PEQC outperforms the state of the art.

Keywords: progress estimate, PERT network, MapReduce, cloud.

1 Introduction

As a solution to manage big data with high cost performance, cloud data management system has attracted more and more attentions from both industry and academia, and it is now supporting a wide range of applications. These applications need operations on massive data, such as reporting and analytics, data mining and decision support. A large part of the applications are implemented through MapReduce[5] jobs. MapReduce integrates parallelism, scalability, fault tolerance, load balance into the simple framework, and MapReduce-based applications are very suitable to be deployed in the cloud which is composed of large-scale commodity nodes. Although data queries are sped-up in cloud-based systems, many queries cost a long period of time, even to several days or months[15]. So users want to know the remaining time of such long-running queries to help decide whether to terminate the query or allow it to complete. For them, some early results within a certain confidence interval which can be computed through online aggregation[7] are even better. The accurate feedback and online aggregation both depend on the progress estimate of queries. There are also some applications in which every query requires critical response time, such as advertisement applications, customer profile management of SNS(Social Network Site), etc. Estimating the remaining time of the queries is beneficial for scheduling them to guarantee their deadlines.

S.-g. Lee et al. (Eds.): DASFAA 2012, Part II, LNCS 7239, pp. 169–184, 2012.

In addition, providing the accurate remaining time of queries is also helpful to performance debugging and configuration tuning for applications in the cloud. Progress estimate of queries in the cloud first confronts challenges in parallel environment, such as parallelism and concurrency[11]. The characteristics of cloud raise this problem's complexity , common failures and environment heterogeneity are two main challenges.

In order to achieve high performance cost, cloud system is often constructed with cheap, commodity machines, and the platform is always of large scale (hundreds or thousands of nodes[4]), so failures are very common. Cloud systems support high fault tolerance and consider failure as normal situation. According to the usage statistics in [4], the average worker deaths per MapReduce job in Google is 5.0 with the average worker machines 268. It is reported in[8] that in a system with 10 thousand of super reliable servers (MTBF of 30 years), according to typical yearly flakiness metrics, servers crash at least twice with the failure rate of 2-4%. For commodity nodes in the cloud, the failure rate is even higher. So the challenge is that once a failure happens, the progress indicator has to cope with failures promptly to provide continuously revised estimates.

Scalability is one main advantage of cloud systems. With the development of applications and the increase of data volume, there is a trend that more nodes will be added into the existing system to get more computing capability. So it is difficult to keep all the nodes in the cloud belonging to the same hardware generation. This brings another challenge – heterogeneity. Different software configurations and concurrent workload further reduce the homogeneity of cluster performance [1]. As [14] reports, in the virtual environment, the performance COV (coefficient of variation) of the same application on the same virtual machines is up to 24%. A MapReduce job is always composed of several rounds of tasks, and the tasks during each round are executed parallel on different nodes. The heterogeneity and variation of performance results in that the task scheduling is composed of irregular task rounds, and it raises the complexity of estimating the progress.

We propose PEQC, a progress estimator of queries based on MapReduce jobs in the cloud. The main contributions of our work include:

1. We model every MapReduce task of the query with its duration and failure probability, and transform the query procedure into a stochastic PERT(Project Estimate and Review Technique) network by allowing the task duration to be random vairables;
2. We propose a method to compute the most critical path of the PERT network, which can present the execution of the whole query in the heterogeneous environment;
3. We provide an efficient update mechanism to repair the network and re-compute the critical path when failure happens;
4. We implement PEQC on Pig[19] & Hadoop[18] to verify its accuracy and robustness in presence of failures.

The rest of this paper is structured as follows. Section 2 describes the related work. Section 3 presents the problem modeling, and discusses the uncertainties and stochastic characteristics of task duration. In Section 4, we propose our solution to estimating progress of MapReduce queries, we also present the repair mechanism to react to common failures. Experimental validation of PEQC on Pig & Hadoop is presented in Section 5. We conclude the paper and discuss the future work in Section 6.

2 Related Work

There are two kinds of areas that are related to our work. First is the area of estimating the progress of SQL queries on single-node DBMS. [9] separates a SQL query plan into pipelined segments, which is defined by blocking operators. The query progress is measured in terms of the percentage of input processed by each of these segments. Its sub-sequent work [10] widens the class of SQL queries the estimator supports, and it increases the estimate accuracy by defining segments at a finer granularity. [3] decomposes a query plan into a number of pipelines, and computes the query progress through the total number of getnext() calls made over all operators. [2] characterize the query progress estimation problem introduced in [3] and [9] so as to understand what the important parameters are and under what situations we can expect to have a robust estimation of such progress. The work of this area focuses on partitioning a complex query plan into segments(pipelines) and collecting the statistics on cardinality information to compute the query progress. These techniques are helpful for estimating the progress of queries running on a single node. However, they do not account for the challenges brought by query parallelization, such as parallelism and data distribution.

Our work is also related to progress estimate of MapReduce jobs. We can classify the related work into two categories: estimating the progress of tasks during one MapReduce job, and estimating the progress of MapReduce pipelines. [17] estimates the time left for a task based on the progress score provided by Hadoop. This paper focuses on task scheduling in MapReduce based on the *longest approximate time to end* of every task, so it orders the task by their remaining times. It computes the progress rate through the progress score and the elapsed time of the task execution, and computes the remaining time of a task based on the progress rate. [17] provides a method to estimate the progress of a MapReduce task, however, there are also several challenges to estimate the progress of MapReduce jobs and MapReduce DAGs. Parallax[12] estimates the progress of queries translated into sequences of MapReduce jobs. It breaks a MapReduce job into pipelines, which are groups of interconnected operators that execute simultaneously. Parallax estimates the remaining time by summing the expected remaining time across all pipelines. It addresses the challenges of parallelism and variable execution speeds, without considering concurrent workloads. ParaTimer[11] extends Parallax to estimate queries translated into MapReduce DAGs. It estimates the progress of concurrent workloads through a critical path based on task rounds, which works well in a *homogeneous* environment. ParaTimer handles task failure through comprehensive estimation, which provides an upper bound on the query time in case of failure. However, ParaTimer assumes only one worst-case failure before the end of the execution, and it has to repeat all the steps in the estimate algorithm to adjust to failures. It may become inefficient when failures are very common, which is one characteristic of cloud. None of the above work handles heterogeneity in the estimation, which is also an important characteristic of cloud.

3 Problem Modeling and Stochastic Characteristics

The project evaluation and review technique (PERT) is widely used to estimate completion time of projects with concurrent workloads[6], we formulate the problem into a

stochastic PERT network. In this section, we discuss the problem modeling procedure and the stochastic characteristic.

3.1 Why stochastic PERT?

We take a query related to HTML document processing and log-file analysis as an example to discuss the problem. The query is to find the ad revenue of all the web page visitings with both page rank and duration in specific ranges. Table *Rankings* contains basic information of pageURLs, and table *UserVisits* models log files of HTTP server traffic. The SQL command is like:

```
SELECT sourceIP, destURL, adRevenue FROM Rankings, UserVisits WHERE
Rankings.pageRank > 10 AND UserVisits.duration > 1 AND
UserVisits.duration <= 10 AND Rankings.pageURL = UserVisits.destURL;
```

This query can be translated into three MapReduce jobs: job1 and job2 filter tuples according to the query conditions, they can be executed concurrently; job3 joins the tables and executes after job1 and job2. Suppose job1 is composed of 2 mappers and 1 reducer, job2 is composed of 6 mappers and 1 reducer, and job3 includes 1 mapper and 1 reducer. In the cluster there are 5 nodes with different hardware settings, and every node has 1 map slot and 1 reduce slot. The scheduling and execution of tasks in this query can be illustrated by a Gantt chart, as shown in Fig. 1.

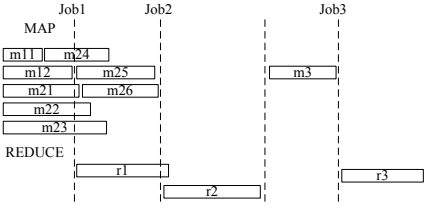

Fig. 1. Task Scheduling and Execution

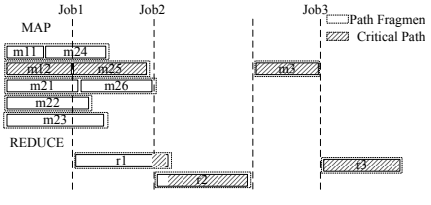

Fig. 2. The Critical Path Detected by ParaTimer

Though map tasks in one job execute the same operations on the same amount of data(one data block), their elapsed time is different because of the cluster heterogeneity. So even during one task round, time differences between tasks are obvious. ParaTimer[11] assumes that tasks in one round process at approximately the same speed. It detects the *path fragment* based on the task round, then composes them to a critical path. In the heterogeneous environment, ParaTimer does not find the actual critical path. The critical path ParaTimer detects is shown in Fig. 2, it is m12-m25-part of r1-part of r2-m3-r3. Not all the tasks on the path are actually *critical activities* that can directly impact total duration of the query, so monitoring these *fake critical tasks* is redundant. The critical path contains not only total tasks, but also some task fragments, which makes the estimation more complicated.

We can predict the duration of every task with sampling debug run or the execution performance history. However, the tasks do not execute exactly as planned due to the variation of hardware and software, which is called ”duration uncertainty”. So fixed task

durations cannot reflect the real execution of the whole query. During the field of operational research, stochastic PERT[16] is always used to solve planning problems and recognize uncertainty in the activity durations.We formulate every task of the MapReduce query into an edge of a DAG, and model the execution processing into a stochastic PERT network.

3.2 PERT Modeling

In this section, we discuss the problem modeling. First, we give the problem definition:

Definition 1. *[Problem Definition] Given the MapReduce job DAG $D(N,E)$ of Query Q in the cloud, return accurate and real-time estimate of the query progress T.*

The execution plan of a query is represented by a MapReduce job DAG D, during which the nodes N represent the jobs, and the arcs E represent the job's logical relationships. The modeling can be defined as follows:

Definition 2. *[Modeling to PERT network] Given DAG $D(N,E)$ of Query Q, construct a PERT network $G(U,\xi)$. Node set $U=\{u_i\}_{i=1}^{n}$ represents the completing of tasks; Edge set ξ is composed of two subsets: $EP=\{ep_{ij}\}$ represents execution procedure of a task, with edge weight ω_{ij} denoting the task duration; $ER=\{er_{ij}\}$ only represents logical relationship of tasks,with ω_{ij} equal to zero.*

In $G(U,\xi)$, nodes represent completing of one or more tasks, and there are two kinds of edges: *EP* and *ER*, $EP \bigcup ER=\xi$. *EP* represents the execution procedures of tasks, the direction of arcs represent precedence relation between two nodes, and the edge weights represent task durations, which are considered as random variables in PEQC; *ER* doesn't represent real task execution, they only represent the precedence relationship of nodes, and the edge weights are always fixed values - 0. The precedence relationship of tasks results from two sequences: logical sequence between jobs and execution sequence of tasks on the same slot. The logical sequence between jobs is from query plan *D*, if two jobs have the relationship: job1→job2, then all the map tasks of job2 must execute after all the reduce tasks of job1. There can only be one task executing on one slot at the same time, so tasks assigned to the same slot have the the precedence relationships. Given the execute plan and task scheduling of the query example, we can construct the PERT network as Fig. 3 shows. The precedence relationship of m3 and r1 is determined by the job DAG, while the precedence relationship of m11 and m24 is determined by the scheduler that assigns them to the same slot.

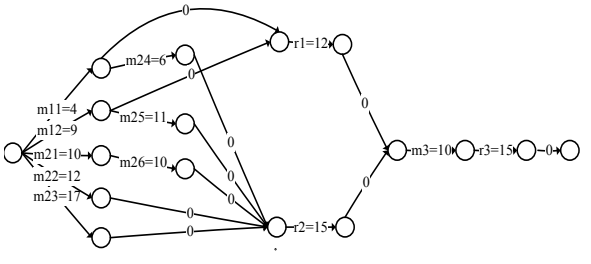

Fig. 3. PERT Network

3.3 The Stochastic Characteristics

There are two kinds of uncertainties during the query execution procedure: duration uncertainty and scheduling uncertainty. The duration uncertainty will change PERT network by alter the weight of arcs, while the scheduling uncertainty will change the structure of PERT network. We discuss duration uncertainty and the stochastic characteristics it brings in this section, the algorithm reacting to scheduling uncertainty will be shown in Section 4.4.

Suppose there are m tasks in the query, let $T = \{T_i\}_{i=1}^{m}$ represents the duration of every task in query Q. We can estimate T_i from sampling debug or running history of tasks on the node, however, tasks will never execute with the durations we expected. It is unrealistic to associate T_i with a deterministic value, so we treat T_i as a random variable which obeys some distribution. We assume all the duration of tasks have independent distributions. Let $P = \{P_i\}_{i=1}^{r}$ represents the path between initial node U_1 and terminal node U_n, where r denotes the cardinality of set P. D_i represents the duration of P_i, where $D_i = \sum_{ep_k \in P_i} (T_k)$. According to the central limit theorem (CLT), D_i approximately obeys normal distribution [8]. The duration D* of the PERT network is $D^* = max_{p_i \in P}(D_i)$, then P^* will be the critical path. Let μ_k, σ_k^2 represent the mean and variance of task duration T_k on one path P_i. Then the expected value and variance of D_i are:

$$E(D_i) = E(\sum_{ep_k \in P_i} T_k) = \sum_{ep_k \in P_i} E(T_k) = \sum_{ep_k \in P_i} \mu_k \tag{1}$$

$$\sigma^2(D_i) = \sigma^2(\sum_{ep_k \in P_i} T_k) = \sum_{ep_k \in P_i} \sigma^2(T_k) = \sum_{ep_k \in P_i} \sigma_k^2 \tag{2}$$

The expected value of D* is $E(D^*) = max_{p_i \in P}(\mathrm{E}(D_i))$. Given a deadline t for the query, we can compute the probability for completing the query before t according to the distribution characteristic and parameters of D*: $P\{D^* < t\} = \int_{-\infty}^{t} f(x)dx$.

4 Proposed Solution

In this section, we introduce our three-step solution to estimating query progress in the cloud. First, PEQC constructs the PERT network of tasks depending on environment resources and scheduling strategy. Secondly, PEQC chooses the most critical path from PERT network. Thirdly, it computes the progress of tasks on the critical path to represent the query progress. Since common failures affect the task scheduling and query progress, we also discuss the update mechanism to respond to failures with less time cost and changes.

4.1 Constructing the PERT network

Given the execution plan DAG of MapReduce jobs for one query, we can compute a topological order of the job sequence. We decompose every job into map tasks and reduce tasks and compute the task topology sequence, which is used as the input of constructing PERT network. As discussed in Section 3.2, the precedence relationship between tasks stems from job DAG and task scheduling. Although we demonstrate PEQC

with FIFO scheduler in this paper, it can be extended to other schedulers with some manageable modifications. Under the FIFO policy, the scheduler maintains a waiting queue of jobs sorted by arrival time and job priority. The scheduler assigns available slots to tasks from the job at the head of the queue. When allocating slots to tasks from one job, it also follows the *data locality* rule, which means that the available slot is preferentially allocated to the task that operates on data in the same node of this slot. We mimic the schedule policy and compute the relationships between tasks scheduled on the same slot.

Among all the jobs of one query, there are some jobs that can execute concurrently, which we call a *job batch*. More precisely:

Definition 3 (Job Batch). *Given the execution plan DAG $D(N,E)$ of a query Q, a job batch J, is a set of jobs $j \in N$ that can execute concurrently with no precedence relationships among them.*

The DAG $D(N,E)$ can be broken into several job batches, which execute sequentially. The job batches can be computed from *D(N,E)*, or can be gotten directly from the MapReduce job launcher. Algorithm 1 computes the task scheduling using job batch as the unit, because only tasks during the same job batch can execute concurrently and influence the task scheduling of each other. In Algorithm1, every task in the input sequence contains a list of dependencies, which includes its precursor tasks according to the job DAG. We adopt two arrays called *MapSlots* and *ReduceSlots* to store scheduling information for map slots and reduce slots, respectively. Every element of the array denotes a task slot, and it is associated with three attributes: pointer to the currently scheduled task, sum of length of tasks executed on this slot, number of tasks scheduled on this slot. Both *MapSlots* and *ReduceSlots* contain information for one job batch, so they are set to initial status at the beginning of scheduling tasks of every job batch(line4). Take the query in Section 3.1 for example, the task sequence is m11, m12, r1, m21, m22, m23, m24, m25, m26, r2, m3, r3. There are two *job batches*: job1 and job2 compose JB1, and job3 composes JB2. The initial state of MapSlots[] is listed in Table 1.

Table 1. Initial State of MapSlots

	slot1	slot2	slot3	slot4	slot5
pointer	null	null	null	null	null
length	0	0	0	0	0
round	0	0	0	0	0

Table 2. MapSlots State after Round One

	slot1	slot2	slot3	slot4	slot5
pointer	m11	m12	m21	m22	m23
length	4	9	10	12	17
round	1	1	1	1	1

Algorithm 1 schedules jobs from the head of the job sequence. It chooses the slot with the smallest length from MapSlots (line8), then arranges a task to this slot. During the tasks belonging to the scheduling job, the one that operates data on the same node with this slot is scheduled preferentially (line8). If there are no such tasks in the scheduling job, it chooses the task at the head of task queue. Table 2 shows the state of MapSlots[] when all the slots have been arranged with one task. Next the algorithm chooses the slot with the smallest length - slot1, and arrange a task from m24, m25, m26 according to data locality. After arranging a new task to one slot, Algorithm1 first adds a new node and edge of this task into the AdjList(line9-11), then it deals with the task's dependencies.

Algorithm 1. Construct PERT Network

```
   input : TaskSequence T: task topology sequence, it maintains six variables for each task t
           ∈ T: t.name, t.type, t.length, t.dependency, t.failureprob, t.nodeip.
   output: AdjList[]:PERT network
 1 Set AdjList[0].data = Start; AdjList[0].out = null;
 2 current = 0; /*current represents the processing node in the network*/;
 3 for all jobbatch JB in T do
 4     MapSlots.setInitial(); ReduceSlots.setInitial();
 5     for all job J in JB do
 6         Slots = MapSlots;
 7         while existing unscheduled task in J do
 8             s = getShortestSlot(Slots); t = getLocalTask(s, J);
 9             t.length = getLength(t, s); t.failureprob = getFailureProb(t.s);
10             AdjList[++current].data = t.name; AdjList[current].adj = null;
11             Edge e ← new Edge(t.name, t.length, t.failureprob, null);
12             if t.dependency != null then
13                 if length( t.dependency ) ==1 && Slots[s].pointer==null then
14                     addEdge(getEdge(t.dependency), e); /*add e consecutively after t's
                       dependency task*/
15                 else
16                     AdjList[++current].data = virtual_t.name; AdjList[current].out = e;
17                     for i in t.dependency do
18                         Edge ed = new Edge(virtual_t.name, 0,0,null );
19                         ed.out = AdjList[i].out; AdjList[i].out = ed
20                 if Slots[s].pointer!=null then
21                     Edge ed = new Edge(virtual_t.name, 0,0, null);
                       addEdge(Slots[s].pointer, ed);
22             else addEdge(start, e); /*make the network's start node tail of e*/
                   Slots[s].length += T.length; Slots[s].round++; Slots[s].pointer = e ;
23             if (all map tasks in J are scheduled) then Slots = ReduceSlots;
24 AdjList[++current].data = End; AdjList[current].out = null;
25 for i with AdjList[i].out == null do
26     Edge ed = new Edge(End, 0, 0, null ); AdjList[i].out = ed;
```

The dependency tasks of task t stem from two precedence relationships: the logical relationship from job DAG is included in the dependency attribute of t, and the precursor task of t on slot s can be gotten from MapSlots[s] or ReduceSlots[s]. If a task has more than one dependency task, PEQC creates a virtual node which represents the event of all the dependency tasks' completing, the weights of all the arcs pointing to the virtual node are set to 0(line16-19). After scheduling all the tasks in the sequence, Algorithm1 adds edges from the nodes without out edges to the end node (line20-22). PEQC adopts an adjacency list to store the PERT network, because the netwok is not too dense, and adjacency list is more convenient in the network update. After constructing the PERT network for the task execution of the query, the remaining work of PEQC will be all operations on the graph.

4.2 Computing the Critical Path

The *job batches* of a query partition $G(U,\xi)$ into k sub-networks:$SG=\{SG_i(U_i,\xi_i)\}_{i=1}^{k}$. Let P_i^* represents the critical path of SG_i, D_i^* represents the duration of P_i^*, then the critical path of the whole network is $P^* = \bigcup_{i=1}^{k}(P_i^*)$, the duration is $D^* = \sum_{i=1}^{k}(D_i^*)$. PEQC computes the critical path of each sub-network and connects them to form the critical path of the whole network. PEQC adopts the *divide and rule* mechanism for two reasons: computing the critical paths of all the sub-networks parallel;re-computing the critical path of some job batch instead of the whole network when failure happens.

Given a PERT network using the duration expected value as the arc weight, there can be several critical paths whose durations are equal, which we call *candidate critical paths*. How to find the *most critical path* that can represent the execution of the whole query from these candidate critical paths? We discuss how the task failure can influence the path length first. When a task failure happens on one slot, the task duration on this slot is changed to the time interval between its start time and failure time, which is shorter than the expected task duration. So task failure can result in the duration reduction of path where the failed task is. However, the critical path is always the longest path in the network. If there are task failures on the path, its duration may be exceeded by other paths. We use a metrics called *path reliability* to measure the probability that a candidate critical path maintains the longest path.

Definition 4 (Path Reliability). *Suppose a candidate critical path P_c includes m tasks, the failure rate of task i is denoted by F_i, then the reliability of P_c is: $P_c = \prod_{i=1}^{m}(1-F_i)$.*

A task failure can be caused by many reasons. However, modelling the failure probability of tasks is beyond the scope of this paper. So we do not address this problem and assume predefined task failure rate to focus on the other factors. Among all the candidate critical paths of each sub-network, PEQC chooses the one with the biggest path reliability, then connects them together as the critical path of the whole query.

4.3 Estimating the Progress

Tasks on the critical path represent the query progress. They can be classified into three types: completed tasks, running task, and pending tasks. Suppose there are n pending tasks of the query, the remaining time of the query is: $T_{remaining} = T_{running} + \sum_{i=1}^{n}(T_i)$. For the pending tasks that haven't executed, PEQC adopts their duration expected values as the estimate time. The elapsed time of the running task can be evaluated through its actual execution speed to provide more accurate estimate. Existing methods break map or reduce task into pipelines and sum the elapsed time of every pipeline as the task estimate[17][12]. In our experiment, we adopts the finish time estimate method of [17], which estimates the time left for a task based on the progress score provided by Hadoop, as $(1-ProgreeScore)/ProgressRate$, and the *ProgressRate=ProgressScore*/elapsed time t.

4.4 Reacting to Failures

PEQC constructs a PERT network to identify how the MapReduce query behaves before its execution, we call it *baseline scheduling*. However, the actual jobs do not execute

exactly as the baseline scheduling because of common failures. Predicting accurately all the failures to happen in the execution is unrealistic, so PEQC provides reactive update algorithm to adjust to the changes caused by failures.

PEQC has to "repair" the whole network and re-compute the critical path whenever a task failure comes up. An intuitive method is to reconstruct the PERT network and repeat the regular steps of Algorithm1. However, this method involves much unnecessary work and costs much time when failures are very common. We can partition the baseline PERT network into three parts depending on the time when failure occurs, just as Fig. 4 shows(failure happens at 3s): executed part contains tasks that have completed or being executed when failure occurs; scheduling part contains tasks that haven't executed and their scheduling is affected by the failure; pending part contains tasks that have not been executed, and their scheduling is not changed by this failure. Tasks in the scheduling part have two characteristics: they belong to the same *job batch* with the failure task, and they have not been executed when the task failure happens. The update algorithm only changes the scheduling part instead of the whole PERT network.

Algorithm 2. Updating PERT When Task Failure Happens

input : AdjList[]: baseline PERT network; T: TaskSquence; tf: failed task; TP_f: time point when failure happens.
output: Updated PERT network

1 $Schedule$.setInitial();
2 $start\_task$ =$T.getNext(Schedule.getLastTask())$; $slot$ = $Schedule$.getSlot(tf);
3 $RTP_f = TP_f - ST$; /* ST: start time of job batch; RTP_f: relative failure time to ST */
4 $Schedule[slot].current\_lengh = Schedule[slot].scheduling\_lengh = RTP_f$;
5 $CandidateSlots$ = getShortestSlot($Schedule - slot$);
6 ls = getLocalSlot($CandidateSlots, tf'$); /* tf' represents the re-executing task of tf*/
7 $AdjList[++length].data = tf'.name$; $AdjList[length].adj$ = null;
8 Edge e = new Edge($tf'.name, tf'.length, tf'.failureprob, null$);
9 addEdge($s.executing\_task$, e);
10 $ls.scheduling\_task = tf'$;
11 $ls.scheduling\_length += tf'.length$;
12 **for** *edge in getFollowingEdge(tf)* **do**
13 **if** $edge.name \in tf.dependency$ **then**
14 moveEdge($edge, tf'$); /*move $edge$ consecutively after tf', update the $scheduling\_length$ and $scheduling\_task$ of two corresponding slots*/
15 **for** all job J ***between*** $start\_task$ ***and*** end_task **do**
16 s = getShortestSlot($Schedule$); t = getLocalTask(s,J);
17 $edge$ = getEdge(t); moveEdge($edge, s.scheduling\_task$);
18 **for** *edge* ***in*** *getFollowingEdge(t)* **do**
19 **if** $edge.name \in t.dependency$ **then**
20 moveEdge($edge$,t)

The left bound ary of scheduling part can be determined by the event nodes representing the completion of executing tasks. The right bound ary is composed of the event nodes that represent the completion of reduce tasks owing to the same job batch with the

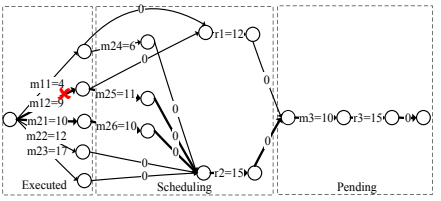

Fig. 4. Task Failure Happens

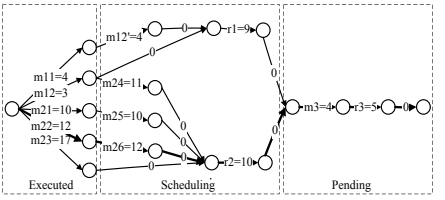

Fig. 5. The Task Scheduling after Failure

failed task. After determining the start task and end task from the task scheduling queue, Algorithm2 reschedules the tasks between the start task and end task, and updates the network. PEQC maintains two arrays called *MapSchedule* and *ReduceSchedule* for every *job batch*, which store the execution and scheduling information for map slots and reduce slots respectively. Every element in the array contains four variables: *executing_task* represents the task that is executing on this slot; *scheduling_task* represents the last scheduled task of this slot; *current_length* represents the duration sum of tasks executed on this slot; *scheduling_length* represents the duration sum of tasks scheduled on this slot. At the beginning of the update algorithm, *scheduling_task* is set to the *executing_task*, and *scheduling_length* is set to *current_length*(line1). The failure task affects the scheduling of tasks of the same type with it, the processing procedure after failed map task and reduce task are the same except the scheduling arrays they used. During the scheduling queue T, the start task is the one following the last task of the executed part (line2), and the end task is the one before the first task of the pending part. For the corresponding slot of tf, its *current_length* should be changed to the relative time of TP_f to the start time of the job batch (line3-4). Algorithm 2 schedules tf' first because failure task always has the highest priority (line5-6). After finding the proper slot ls, it creates a new node and a new edge for tf', the new node is added at the end of Adjlist (line7-8), and the new edge is added after the *executed_task* of ls (line9-11). The edges following *tf* which represent the logical precedence of the query, are moved consecutively after the new task tf' (line12-14). After processing tf', Algorithm2 reschedules the tasks between *start_task* and *end_task*(line15-20). The re-computed task scheduling after failure is shown in Fig. 5.

The previous method focuses on updating the network when a task failure happens. When node failure happens, all the executing tasks on the failure node have to be re-executed on other nodes. If a map task has completed on the node before failure happens, but the job the map task belongs to hasn't completed, then this map task also has to be re-executed. Because the output results of map task are stored on the local node, and the output results are unavailable after the node failure. The updating algorithm after node failure should take three things into consideration: first, there may be more than one failed task to be re-executed; second, the slots on the failure node should be disabled when re-scheduling tasks; third, the scheduling of tasks in the pending part may be changed because of the disabled slot. The pending part may include several *job batches*, and the updating can execute parallel.

5 Evaluation

In this section we evaluate PEQC, and compare it with ParaTimer and the default estimator of Pig from two aspects: the precision of query estimate and the resilience reacting to failures. The experiment is implemented on Pig 0.8.0 and Hadoop 0.20.2.

5.1 Experimental Setup and DataSet

All the experiments are run on a heterogeneous cluster of 31 nodes connected by a 1Gbit Ethernet switch. One node serves as the namenode of HDFS and jobtracker of MapReduce, and the remaining 30 nodes act as slaves. There are four levels of hardware equipment of all the nodes in the cluster, the setup details are shown in Table 3. The master node which acts as the jobtracker belongs to levelIII.We set the block size to 64M, and configure Hadoop to run 1 mapper and 1 reducer per node.

Table 3. Testbed Setup

	LevelI	LevelII	LevelIII	LevelIV
CPU	Quad Core 2.33GHz	Quad Core 2.66GHz	Quad Core 4GHz	Quad Core 2.13GHz
RAM	7GB	8GB	4GB	4GB
Disk	1.8TB	2TB	2TB	500GB
No. of Node	4	10	14	2
Node Type	PC	PC	PC	Server

In the experiment we adopt the query introduced in Section 3.1, which contains three MapReduce jobs. We perform the tests on two datasets with different data size. In dataset1, the data size of *Rankings* and *UserVisits* are 2G and 24.5G; in dataset2, the data size of the two table are 1.4G and 61G. We adopt the data generation method from[13]. It first generates a collection of random HTML documents, then generates the data of *Rankings* and *UserVisits*. The task number of every job is listed in Table 4. We estimate the distribution parameters of task duration from the task running history of every node, Table 5 shows the distribution parameters of map task in Job2 on four nodes, they belong to four different hardware levels.

Table 4. No. of Tasks

	Mappers Dataset1	Reducers Dataset1	Mappers. Dataset2	Reducers Dataset2
Job1	393	27	970	66
Job2	32	3	21	2
Job3	406	27	998	67

Table 5. Distribution Parameters

	Node1	Node6	Node11	Node30
Mean Value	12.60	8.19	7.30	9.60
Variance	0.77	0.72	0.46	0.99

5.2 Accuracy Evaluate

In this experiment, we evaluate the estimate accuracy and the time overhead of getting the first estimate result in the heterogeneous environment. We obtain progress estimate every 10 seconds and examine three metrics: mean estimate error, max estimate error[3][11], and time overhead. Pig reports the progress as the percentage of the query completed, PEQC and ParaTimer provide the remaining time. In order to compare them with the same metric, we transform the percentage of query f reported by Pig into remaining time: $t_{remaining} = \frac{(1-f)(t_i-t_0)}{f}$. t_0 represents the time when the query is submitted, t_i represents the time when this estimate is reported. Let t_e represents the estimated remaining time at time t_i, t_n represents the time when the query actually completes, then the estimate error at time t_i is: $error_i = |\frac{t_i+t_e-t_n}{t_n-t_0}|$.

Fig. 6 and Fig. 7 illustrate the remaining query time estimated by the progress indicator over time on two datasets. There are four lines in each figure. The almost straight line represents the actual remaining time. In Fig. 6, there are two turning points on Pig's curve: *t1*(140s) and *t2*(380s). Pig's indicator assumes all the jobs execute sequentially with the same weight. Job1's mappers occupied the slots before *t1*, so the estimate is pessimistic. After that Job1 and Job2 execute concurrently, the estimate changes to be optimistic. Job3 starts approximately at *t2*, and its duration accounts for about one-third of the whole query's time. So after *t2*, the estimate is close to the actual remaining time. ParaTimer's estimate becomes close to the actual line at *t3*(448s), when there are only the reducers of Job3 executing. The critical path ParaTimer detects after *t3* is close to the actual one. PEQC computes the critical path from PERT network, and it supplies relatively stable estimate. In general, PEQC's estimate is slightly optimistic. This is because PEQC computes the critical path based on the mean value of every task duration, which is different from the task's duration in the actual query execution. The trend of every curve in Fig. 7 is similar to that of Fig. 6. The metrics of tests on two datasets are shown in Fig. 8. Pig fetch the job progress from Hadoop directly without any overhead. PEQC has to construct the PERT network and compute the critical path before estimating, and ParaTimer has to compute the path fragment and compose the critical path, which costs more time than PEQC.

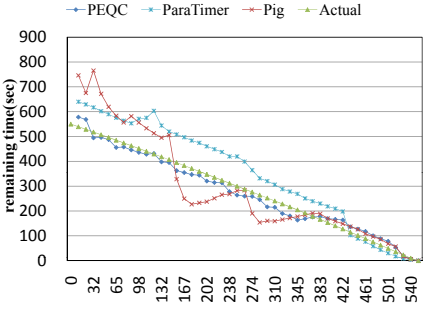

Fig. 6. Estimated Result (Dataset1)

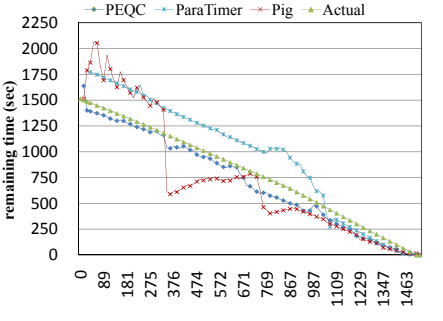

Fig. 7. Estimated Result (Dataset2)

	Dataset1			Dataset2		
	PEQC	ParaTimer	Pig	PEQC	ParaTimer	Pig
Mean Error	4.2%	10.3%	12.3%	4.7%	13.1%	14.1%
Max Error	10.6%	31%	44.9%	11.7%	24.1%	33.5%
Overhead(s)	1.05	6.45	0	3.40	18.52	0

Fig. 8. Metrics of Accuracy Test

	Task Failure			NodeFailure	
	PEQC	Paratimer	Pig	PEQC	Pig
Mean Error	4%	15.4%	16.2%	4.5%	15.3%
Max Error	8.6%	28.4%	38.2%	23.4%	39.8%
Overhead(s)	1.02,0.87,0.54	6.42,6.2,6.31	0	2.9	0

Fig. 9. Metrics of Failure Test

5.3 Robustness to Failures

In this section, we conduct two tests to evaluate the estimators' robustness to task failures and node failures. The metrics we adopt are: mean estimate error, max estimate error, the time overhead of reacting to each failure. Fig. 10 shows the results of task failure test. ParaTimer provides an additional estimate called *PessimisticFailureEstimate*, which assumes a single worst-case task failure will occur. We run this test on dataset1 and fail three tasks at 134s, 225s, and 377s. The metrics are shown in Fig. 9. When task failure occurs, PEQC only recomputes part of the network, so it cost less time than ParaTimer.

Fig. 11 shows the results of node failure. ParaTimer does not support estimate in presence of node failures, we conduct this test on PEQC and Pig on dataset2. We make two node failures(two nodes in LevelI) by cutting off their network connections to the cluster concurrently. The node failure happens at about 275s, when Job1 is executing. There are 28 slave nodes left in the cluster after the failure, all the executing tasks and completed map tasks of Job1 have to be re-executed, and the scheduling of the remaining tasks have to be changed. The metrics of this test is shown in Fig. 9. It costs PEQC 2.9s to repair the network and re-compute the critical path. Though tasks in both *JB1* and *JB2* have to be re-scheduled, and their critical paths have to be re-computed, PEQC can do the repair work in parallel.

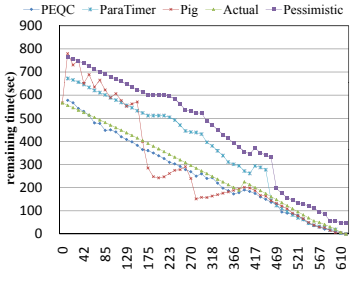

Fig. 10. Estimated Result (Task Failure)

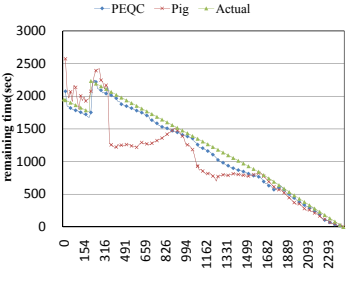

Fig. 11. Estimated Result (Node Failure)

6 Conclusion and Future Work

In this paper, we propose PEQC, a progress indicator of queries composed of MapReduce jobs in the cloud. PEQC focuses on solving challenges brought by two features of cloud: environment heterogeneity and common failures. PEQC models the task

execution of a whole query into a stochastic PERT network. It adopts partial update mechanism to react to the task failures. Based on our implementation on Pig & Hadoop on a heterogeneous cluster, PEQC provides promising remaining time estimate of a query in the cloud, and can repair the PERT network when failure happens in acceptable time. There are also some inherent characteristics within MapReduce that give rise to difficulties of the problem, such as speculative execution and data skew in the reduce phase. We will judge the tradeoff between algorithm complexity and processing time, and make PEQC more robust to these challenges in the future work.

Acknowledgements. This research was partially supported by the grants from the Natural Science Foundation of China (No. 91024032, 91124001, 61070055, 60833005), the Fundamental Research Funds for the Central Universities, and the Research Funds of Renmin University of China (No. 11XNL010, 10XNI018), National Science and Technology Major Project (No. 2010ZX01042-002-003).

References

1. Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D., Silberschatz, A., Rasin, A.: HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. In: 35th ACM Conference of Very Large Databases, pp. 922–933. ACM Press, New York (2009)
2. Chaudhuri, S., Kaushik, R., Ramamurthy, R.: When can we trust progress estimators for SQL queries. In: 25th ACM International Conference on Management of Data, pp. 575–586. ACM Press, New York (2005)
3. Chaudhuri, S., Narassaya, V., Ramamurthy, R.: Estimating progress of execution for SQL queries. In: 24th ACM International Conference on Management of Data, pp. 803–814. ACM Press, New York (2004)
4. Dean, J.: Experiences with mapreduce, an abstraction for large-scale computation. In: PACT, p. 1. IEEE Press, Washington (2006)
5. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. In: OSDI, pp. 137–150. ACM Press, New York (2004)
6. Malcolm, D.G., Roseboom, J.H., Clark, C.E., Fazar, W.: Application of a technique for research and development program evaluation. Operations Research 7(5), 646–669 (1959)
7. Hellerstein, J.M., Haas, P.J., Wang, H.J.: Online Aggregation. In: 17th ACM International Conference on Management of Data, pp. 171–182. ACM Press, New York (1997)
8. Dean, J.: Designs, lessons and advice from building large distributed systems. In: Keynote from LADIS 2009 (2009)
9. Luo, G., Naughton, J.F., Ellmann, C.J., Watzke, M.: Toward a progress indicator for database queries. In: 24th ACM International Conference on Management of Data, pp. 791–802. ACM Press, New York (2004)
10. Luo, G., Naughton, J.F., Ellmann, C.J., Watzke, M.: Increasing the accuracy and coverage of SQL progress indicators. In: 21st IEEE International Conference on Data Engineering, pp. 853–864. IEEE Press, Washington (2005)
11. Morton, K., Balazinska, M., Grossman, D.: ParaTimer: A progress indicator for mapreduce DAGs. In: 30th ACM International Conference on Management of Data, pp. 507–518. ACM Press, New York (2010)
12. Morton, K., Friesen, A., Balazinska, M., Grossman, D.: Estimating the progress of MapReduce pipelines. In: 26th IEEE International Conference on Data Engineering, pp. 681–684. IEEE Press, Washington (2010)

13. Pavlo, A., Rasin, A., Madden, S., Stonebraker, M., DeWitt, D., Paulson, E., Shrinivas, L., Abadi, D.J.: A comparison of approaches to large-scale data analysis. In: 29th ACM International Conference on Management of Data, pp. 165–178. ACM Press, New York (2009)
14. Schad, J., Dittrich, J., Quian-Ruiz, J.: Runtime measurements in the cloud: observing, analyzing, and reducing variance. J. Proc. of VLDB Endowment 3(1), 460–471 (2010)
15. Schatz, M.C.: CloudBurst: highly sensitive read mapping with MapReduce. Bioinformatics 25(11), 1363–1369 (2009)
16. Shogan, A.W.: Bounding distributions for a stochastic pert network. Networks 7(4), 259–381 (1977)
17. Zaharia, M., Konwinski, A., Joseph, A.D., Katz, R., Stoica, I.: Improving MapReduce performance in heterogeneous environments. In: OSDI. ACM Press, New York (2008)
18. The Hadoop Website, `http://hadoop.apache.org`
19. The Pig Website, `http://pig.apache.org`