Deadline Queries: Leveraging the Cloud to Produce On-Time Results

David Alves, Pedro Bizarro and Paulo Marques
CISUC
University of Coimbra
Coimbra, Portugal
{dalves,bizarro,pmarques}@dei.uc.pt

Abstract—**MapReduce has become a widely used tool for computing complex tasks that process massive amounts of data in large clusters. Support for MapReduce tasks in cloud environments has been provided but it is left to users to make best guesses on the number of nodes needed for a task to complete within acceptable time. Moreover, the time a task will take to complete is often unknown beforehand. Previous research addressed this problem by establishing time constraints for query execution and, when needed, reduce the accuracy of queries using result approximation and/or sampling. However, in many situations reduced accuracy is not tolerable. In this paper we present FloodDQ, a MapReduce system that implements deadline queries–queries that must finish before a deadline, never discarding data or reducing accuracy. FloodDQ produces timely, accurate results by adaptively increasing or decreasing computing power, at runtime, towards completing execution within the specified deadline. In FloodDQ, users only specify a deadline and the input data. The system monitors the progress of the task and extrapolates whether it will complete on time. If the task is deemed to complete after the specified time, the system requests more nodes from an IaaS Cloud provider, and adds them to the computation. On the other hand, if the task is deemed to complete before the specified time the system quiesces and releases surplus nodes, cutting costs to a minimum. This paper describes FloodDQ's architecture for supporting deadline queries and presents experimental results where the system always meets the deadline in spite of changes to the number of nodes, size of data or existence of perturbations.**

Keywords: Elastic MapReduce, Deadline queries.

I. Introduction

A panoply of organizations produce data that must be processed within a timeline: Telcos and Banks often must process their daily records before the next business day; web scale companies often perform periodic tasks on operational data, e.g., log processing. Whether loose or strict, there is a clear benefit from producing the results of computations within a specific deadline. Prior work in DBMS query execution [1][2][3] has addressed this problem albeit at the expense of accuracy, effectively discarding data towards producing results within hard time constraints. Nevertheless a significant number of use cases require exact results and therefore cannot use such techniques. The underlying assumption in previous work has been that hardware resources are static and that the data must fit those resources processing capabilities if results are to be produced within the deadline. In our approach we drop this assumption, effectively allowing the system to add or release computing resources dynamically, *adapting the system, not the data*, *to the task's time requirements*. We specifically address large data processing computations, e.g., those run on large scale data processing systems such as Hadoop [4], and aim at producing accurate results within a deadline even when facing variable volumes of data and even for new, never previously executed tasks.

The problem is that adequately defining beforehand a number of nodes is very difficult for new tasks and/or variable data volumes, as different tasks have very different processing requirements, (e.g., due to data size, data content, data skew, etc.). This problem is relevant for private clusters where cluster resources must be shared by multiple simultaneous tasks, but it is even more relevant in cases where resources are rented, such as when using Amazon's Elastic Computing Cloud (EC2)[1]. In real scenarios [5], users commonly choose the number of nodes by making an educated guess based on previous experience, balancing politeness (in private, shared clusters) or cost (in public, rented clusters) with the computation speed they assume is reasonable. In order not turn the correct choice of number of nodes into guesswork users might rely on automated query execution time prediction techniques, but the current state of the art techniques either present significant error margin [5] (26%) or require in-depth runtime information that isn't always available, e.g., precise information on the compute cycles spent per input byte [6].

Our proposal to overcome these problems is to let users specify the desired time for a computation to finish instead of defining which resources are to be used for the computation.

In this paper we introduce a research system, FloodDQ, which implements our proposal. FloodDQ employs the MapReduce [7] programming model, which has been extensively used to tackle large scale data analysis. We

[1] http://aws.amazon.com/ec2/

[2] In practice we add an additional "slack" time, provided by

further extend the model to enable runtime and near-real-time up/down scalability, without loosing work or data, allowing to change the number of computing resources used dynamically. We use runtime query progress measurement data to provide feedback to a decision engine that asserts whether there is a surplus or lack of computational resources, towards meeting the deadline. The decision engine then requests/releases resources from a resource pool, i.e., either from a larger set of nodes in a private cluster or from an IaaS provider's resource pool (e.g., EC2). In our approach the system starts processing data and increases or decreases the number of nodes towards converging to the minimal number that will process all data within the defined time limit. Our solution does not require any pre-execution time knowledge of data volumes, data characteristics, or processing capabilities. Overall, the use of deadline queries allows a more efficient and cost-effective way of performing data processing on large-scale clusters when time and computing resources are key metrics that have to be managed. Moreover our implementation enables providing data processing services on a Platform-as-a-Service context without requiring user input regarding cluster size.

Contributions and Organization

The core contribution of this paper lies on our approach towards handling hard time limits for query execution. While previous approaches have assumed static computing power and addressed the time constraint by reducing the amount of data to process, we approach the problem in the other direction. We assume that data cannot be discarded, results must be precise, and vary the computing power towards meeting the deadline.

This paper makes the following specific contributions:

- We extend the concept of hard time limits by introducing the notion that of *deadline queries*, totally accurate queries that must conclude execution before a specified deadline. Deadline queries fulfill their deadline by acquiring or releasing hardware resources dynamically. These are defined in Section II.
- We propose architecture for supporting deadline queries along with a real working system. FloodDQ supports deadline queries in an efficient and scalable way. The architecture of FloodDQ is described in Section III.
- An experimental evaluation of FloodDQ that shows advantages of this approach to select/project/aggregate queries in Section IV. In it we vary the number of starting nodes, the amount of data to process, the existence of perturbations and the length of the deadline. Overall, we show that in all these conditions the system is able to fulfill the specified deadlines.

II. Deadline Queries

In many situations data sampling or approximation as a means for controlling execution time is not a viable solution. For instance a query that processes a dataset to calculate a financial balance cannot discard any data. On the other hand, until now, totally accurate queries take an unknown amount of time to complete, depending on the query complexity, on data characteristics (e.g., size, content, skew, etc.) and on available computing power. We extend the concept of stipulating hard time limits to query execution to that of deadline queries: totally accurate queries that complete within a deadline.

More formally, a deadline query is one that, for any input data set D_{total}, and for any starting set of computing nodes C_{init}, will vary used resources between some minimum set of computing nodes C_{min} and some maximum set of computing nodes C_{max} such that: either i) the set of nodes used is minimized and the execution completes in a time T that is less than or equal to a specified deadline $T_{deadline}$, or ii) if it is not possible to meet the deadline even with C_{max} nodes then the query will use C_{max} nodes, minimizing T.

This definition assumes that all computing nodes have the same processing capabilities. Table 1 introduces the symbols used throughout this section.

TABLE I. Table of Symbols

Symbol	Meaning
D	Input data
$D_{current}$	Input data processed so far
D_{total}	Input data total amount
D'	Input data projected to have been processed by time T'
C	Computing nodes
C_{init}	Computing nodes initial amount
C_{max}	Computing nodes maximum amount
C_{min}	Computing nodes minimum amount
$C_{required}$	Computing nodes required to finish before $T_{deadline}$
T	Time
T_{total}	Actual time the query took to complete
$T_{current}$	Time elapsed since query start
$T_{deadline}$	Time of the deadline (relative to query start)
ΔT_{nn}	Expected mean time to obtain new nodes (configurable)
T'	Time until obtaining new nodes ($T_{current} + \Delta T_{nn}$)
I	The set of periodic reports since the last scaling operation
I_{size}	The number of reports in the I set
i	The index of a periodic report in the I set

A. *Meeting the Deadline*

Consider a query that returns the average value and average trading volume of all stocks, grouped by symbol:

```
SELECT symbol, avg(value), avg(volume)
FROM   StockTicks
GROUP  BY symbol;
```

Given a query plan, on a certain platform, the time a query takes to complete, T_{total}, is a function of the input data, D_{total}, and of the available computing nodes C. The challenge is to meet the deadline time $T_{deadline}$, such that $T_{total} \leq T_{deadline}$ in the presence of varying data (size, content, skew), possible system perturbations, different starting resources, faulty nodes, or any other parameters that may affect total execution time.

B. Estimating Progress and Computing Resources

Query progress estimation is a widely studied subject for single site query execution [8][9][10][11][12], however only very recently did alternatives appear for large-scale parallel computations. Parallax [13] and its evolution, ParaTimer [14], represent the current state of the art in progress estimation for MapReduce pipelines. ParaTimer uses critical-path progress estimation, i.e., it finds a critical data path in a graph of MapReduce sequences and then uses Parallax to estimate progress within each of the pipelines in the critical path. Similarly to Parallax/ParaTimer and to the original MapReduce paper we start by restricting our scope to single pipeline queries, which we present in this paper, leaving joining pipelines (joins) to future work. In our approach we have employed a different progress estimation for two reasons: i) our solution, albeit simple, has shown to be accurate in most cases with single pipeline queries consistently meeting the deadlines; ii) Parallax requires the input of pre-execution debug runs to estimate time remaining on yet to execute portions of the pipeline, a requirement that mismatches our objectives.

Summarily, the steps we take to assert whether the deadline will be met are:

1. Receive from each computing node periodic reports on the volume of consumed data.
2. Aggregate the reports from multiple nodes, for a given period, and calculate the sum of the individual reports.
3. Use linear regression to estimate when the total amount of data will be processed.
4. Make a decision, based on the result of step 3, on whether there is a need to increase or decrease the number of resources and request/release them. Adjust the computation accordingly and repeat from step 1.

We now proceed to introduce our approach more thoroughly. At each period (every time interval *i*) the volume of consumed data from all the computing nodes is summed (D_i) and tagged with the current relative timestamp (T_i), forming a $<T_i,D_i>$ pair and added to the *I* set. At each time interval the decision engine's goal is to assert whether the deadline will be met and if there is a lack/surplus of resources. The first step is to use ordinary least squares, (a.k.a. simple linear regression) to estimate, based on the reports on the *I* set the current data processing rate ($Rate_{current}$):

$$\mathrm{Rate_{current}} = \frac{\mathrm{I_{size}} * \sum_{i\in I} T_i D_i - \sum_{i\in I} T_i * \sum_{i\in I} D_i}{\sum_{i\in I} {T_i}^2 - (\sum_{i\in I} T_i)^2}$$

We can find whether the deadline is to be met by using simple extrapolation to calculate the total time at the current rate:

$$\mathrm{T_{total}} = \mathrm{T_{current}} + (\mathrm{D_{total}} - \mathrm{D_{current}})/\mathrm{Rate_{current}}$$

If $T_{total} > T_{deadline}$ the deadline will be missed and the computation requires more nodes. On the other hand, if $T_{total} < T_{deadline}$[2] the computation may be using too many nodes. Thus the next step is to calculate the amount of nodes that will allow the computation to complete on time.

2 In practice we add an additional "slack" time, provided by configuration so that we can account for errors in estimation.

As releasing and acquiring nodes takes a certain time, ΔT_{nn}[3], before calculating the amount of nodes required, we must find the volume of data that will have been processed by the time new nodes arrive or old nodes leave. This time, *T'*, is given by:

$$\mathrm{T'} = \mathrm{T_{current}} + \Delta\mathrm{T_{nn}}$$

The data, *D'*, that will have been processed by time *T'* is given by:

$$\mathrm{D'} = \mathrm{D_{current}} + \mathrm{T'} * \mathrm{Rate_{current}}$$

After we have calculated *D'* the required data processing rate, $Rate_{required}$, to be able to meet the deadline is given by:

$$\mathrm{Rate_{required}} = \frac{\mathrm{D_{total}} - \mathrm{D'}}{\mathrm{T_{deadline}} - \mathrm{T'}}$$

Finally all that is missing is to translate the required rate into an actual number of required computing nodes. As previously mentioned we assume that all nodes have the same processing capability, therefore the required number of nodes, $C_{required}$, required to meet the deadline is given by:

$$\mathrm{C_{required}} = \text{ceiling}\left(\frac{\mathrm{C_{current}} * \mathrm{Rate_{required}}}{\mathrm{Rate_{current}}}\right)$$

As the number must be an integer, we use ceiling to find the required number of nodes, $C_{required}$. If $C_{current} < C_{required} < C_{max}$ we request $C_{required}$ - $C_{current}$ nodes. If $C_{current} > C_{required} > C_{min}$ we release $C_{current}$ - $C_{required}$ nodes, if $C_{required} = C_{current}$ we take no action. After an action of acquiring/releasing nodes has been taken and the nodes have joined/left the cluster, we dump the state of the decision engine. This process is repeated until the task completes.

III. Runtime And Execution

In this section we will introduce the runtime environment of FloodDQ and introduce all the relevant implementation details for deadline queries.

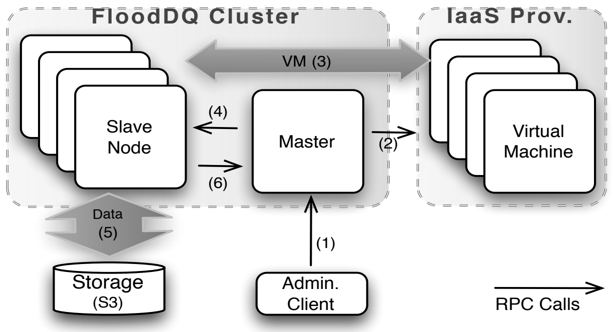

Figure 1. FloodDQ's architecture and execution overview

A. Runtime Overview and Architecture

FloodDQ's architecture follows the master/slave paradigm, both for simplicity and responsiveness in making provisioning decisions. MapReduce tasks are submitted to the master node and executed on a set of slave nodes. Execution starts with MapReduce computation submissions from administrative clients (1 in Figure 1). The

3 *ΔTnn* is actually two parameters one to acquire and one to release nodes; we mention only one for simplicity.

administration client provides the following information: 1) Initial number of nodes; 2) Data location; 3) Deadline; 3) Computation sequence description.

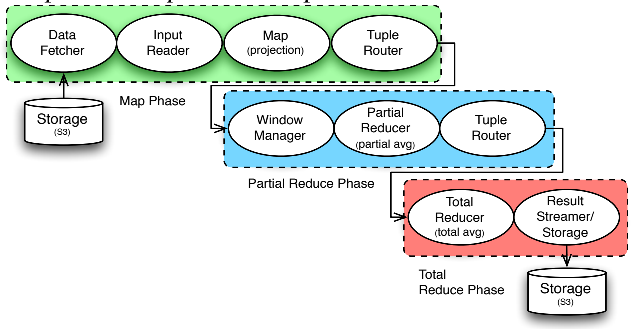

Figure 2. A logical view of a streaming MapReduce dataflow

Just as with traditional MapReduce jobs, the master creates data partition pointers, based on a partitioning function, and places these in a queue. The master node then composes a logical plan for the computation. Figure 2 depicts a logical plan for the query introduced in Section II.A. Based on the initial number of nodes this logical plan is then transformed to a physical plan that will execute in the slave nodes, depicted in Figure 3. To start execution the master requests the configured initial number of nodes issuing provisioning commands to the IaaS provider (2 in Figure 1). After the IaaS provider has allocated and booted the virtual machines (3 in Figure 1), the master node deploys the corresponding portion of the physical plan to each node (4 in Figure 1). The slave nodes request data partitions from the master and start fetching data from a storage location (5 in Figure 1). The storage service may vary; in this paper we have used Amazon's Simple Storage Service (S3)[4]. Slave nodes periodically send heartbeats with embedded progress metrics such as data consumed at each period (6 in Figure 1), which the master uses to adaptively make decisions regarding the provisioning requirements, as described in Section II.B. When it is predicted that more nodes are required to process the data within the deadline the master requests them the IaaS provider and adds them to the computation. If, on the other hand, it is predicted that there are surplus nodes, then those nodes are released.

B. *Streaming MapReduce and Dataflows*

We employ MapReduce [7] as the core computational model. In particular we developed a streaming version of MapReduce, in which data is continuously streamed between operators without intermediate materialization. Streaming MapReduce, originally introduced in [15], presents several advantages such as enabling online aggregation, i.e., allowing reducers to perform work without having to wait for the full map output. Recent work [16], which used streaming MapReduce, demonstrated the advantages of the model, but while this work focused on a static provisioning setting we leverage streaming MapReduce in a dynamic provisioning environment.

[4] http://aws.amazon.com/s3

Dataflow composition is performed programmatically by adding Map and Reduce operators to a sequence that, when processed by the master node, will form a logical plan, as can be seen in Figure 2. Each sequence may contain several instances of Map or Reduce operators that will be aggregated into the following phases:

Map Phase - Computations always start with a Map phase, as they must at least transform the raw data into a tuple stream. In this Map phase, data is retrieved from storage, transformed and passed to one or more Map operators.

Partial Reduce Phase – Executing a part of one or more Reduce operators, a Partial Reduce phase operates on a window of tuples. The purpose of the Partial Reduce phase is mainly to reduce network traffic by "reducing" data before it is sent to the Total Reduce phase.

Total Reduce Phase– Executing the final part of a Reduce operator, Total Reduce phase produces the final Reduce output (e.g., performing the total SUM on the output of the multiple partial SUMs).

C. *Elastic Scalabiltiy and Optimizations*

When new nodes must be added to the cluster, the master node expands the physical plan to account for the additional nodes. Each node is then given a new portion of the plan to execute, connects to the already present nodes in the cluster, fetches partitions from the master and starts processing data. If on the other hand nodes must be released, then the leaving nodes will first finish to process the partitions that are in progress, flush all state to downstream nodes and will only then leave the cluster. In no case is data discarded or work lost.

It is important to note that, at its current stage, FloodDQ does not support state migration. This means that the amount of total reducers (the only locations with permanent state) must be maintained constant during the whole execution of a query. This is usually not a problem as the number of nodes executing Total Reducers is often much lower (typically there will be only 1 total Reducer) than the number of nodes executing Mappers and Partial Aggregations.

Partial aggregation (PA) stands for the introduction of an intermediate phase in a reduce phase that aggregates results for only a portion of a key group, leaving it to the total reduce phase to perform the total aggregation on the intermediate aggregation results. Windowed PA means that a PA operator processes a *window* of tuples at a time. The reason why we use windowed PA is that a large class of aggregation operators, e.g., `MIN`, `MAX`, `SUM`, `AVG`, etc., require the entire input data set to produce an accurate final result for a given key, meaning that the entire set for that key group must be sent to the same machine. This presents two problems: 1) Data Skew - if key groups have very different cardinalities then there is the possibility of data skew, i.e., some nodes are assigned key groups with much higher cardinality than other nodes, causing them to overload and/or become bottlenecks. 2) Data Migration - when adding/releasing computing resources whole key groups would have to be migrated to other machines, causing augmented bandwidth consumption and imposing a variable time delay that might affect meeting the required deadline. PA solves problem 1 as it commonly reduces the input data set on input data to another set whose cardinality is, at most, the number of different keys. Windowed PA also solves problem 2 because there is never the need to migrate the state of a PA operator. New nodes can immediately start to compute partial aggregations and leaving nodes only need to flush the internal state of the PA operators to the Total Reducers.

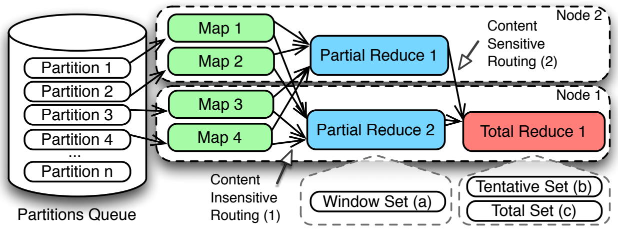

Figure 3. A physical view of a streaming MapReduce dataflow

In FloodDQ we employ two different data routing operators, one that is content insensitive (1 in Figure 3) to route data to and from Map operators and to Partial Reducers, and one that is content sensitive (2 in Figure 3) that routes data to the Total Reducers. For the former we use the load-balanced routing operator introduced in RiverDQ [17], which routes data based only on the queue size of consumer operators. For the latter we implemented a part of the Flux [18] routing operator, specifically we implemented the mechanism Flux introduced to deal with transient data skew, the transient skew buffer (TSB). Similarly to the content-insensitive router the TSB allows to keep forwarding data to consumers based on queue size, but maintains separate queues for different consumers.

D. *Partial Fault Tolerance*

FloodDQ has partial support for fault tolerance: it allows node failures on all nodes except the few nodes (typically one) running Total Reduce Operators. Total support for fault tolerance is left as future work. When fault tolerance is enabled each data input tuple is tagged with the id of the input partition that originated it (`p_id`). This information is passed on to downstream operators. If the downstream operator is a Map then the operator processes the input tuple and tags the corresponding output tuple (if any) with the same `p_id`, i.e., `((p_id,key),value))`. Partial Reducers produce a partial results grouped by both `p_id` and `key`, i.e., `map((p_id,key),bag(value))`.

When a partition is finished Map operators send a *partition end punctuation,* i.e., a `(P_END,p_id)` control tuple, to all downstream operators, which causes them to flush the state of that particular partition and to forward the punctuation to their downstream operators. Nodes executing total reduces maintain two in-memory sets: the tentative set (b in Figure 3), containing `(p_id,map(key,bag(value)))` entries and the total set (c in Figure 3) containing `map(key,bag(value))` entries. When tuples are received from partial reduces they are further reduced and saved onto the tentative set. When partition end punctuations are received the corresponding `(p_id,bag(key,(value)))` set of tuples is merged into the total reduction result.

The master receives periodic heartbeats from each node enabling it to detect when failures occur. When failures do occur the master node signals all reachable nodes to cut communication with the failing nodes. Finally the master sends all remaining nodes a set of partition id's to be invalidated, which causes total reduce operators to discard the relevant portion of the tentative set and re-adds the invalidated partitions to the to Process queue.

A final note: due to space constraints, we do not show experiments with node failures in the next section.

IV. PERFORMANCE EVALUATION

In this section we describe the performance evaluation of FloodDQ. The experimental setup is described first in Section A. Next, in the following experiments we vary:

- The number of starting nodes (Section B);
- The dataset size (Section C);
- The existence of perturbations (Section E); and
- The length of the deadline (Section D).

A. *Experimental Setup*

The experiments were run using our prototype, FloodDQ[5], on top of Amazon Elastic Compute Cloud (Amazon EC2). The rented nodes used in the experiment belonged to the Amazon EC2 European (Ireland) cluster and consisted in between 1 and 27 nodes of so called m1.large instances. Each m1.large instance is a virtual machine corresponding to a machine with 2 dual-core Xeon CPUs, with a 2.66 GHz clock speed, and 7.5 GBytes of RAM.

The running query was an aggregation query identical to the query of Section II.A:

```
SELECT symbol, avg(value), avg(volume)
FROM   StockTicks
GROUP  BY symbol;
```

The dataset used consisted in between 10 GBytes and 32 GBytes of real stock data[6]. The dataset was stored as plain text files in Amazon Simple Storage Service (Amazon S3).

The metrics reported in the experiments are:

- **Estimated time left** (in seconds): this is the time that FloodDQ estimates is left for execution to finish. The estimated time left, together with running time and the deadline value, is used by FloodDQ to determine if more nodes are needed of if nodes can be released;
- **Number of nodes:** this is the current running number of nodes being used during query execution. The number of nodes shows FloodDQ scaling up or down in reaction to the computed estimated time left.

Unless otherwise noted, the base experiment has a deadline of 15 minutes (or 900 seconds), processes 32 GB of data, and starts with a single node. All experiments were

[5] Available at http://flood.dei.uc.pt

[6] Purchased from http://www.grainmarketresearch.com/

run successfully 3 times, only the best run is shown, for clarity.

B. Varying Initial Amount of Resources

In this experiment, the query starts with a varying number of nodes: 1, 5, or 10 nodes. Figures 4 and 5 plot the running estimated time left and number of nodes being used, respectively, for this experiment. As can be seen from Figure 4 and Figure 5, shortly after the initial ramp-up, FloodDQ has a good estimate on the remaining time left and determines that it needs around 6 nodes to terminate the execution within the deadline. Thus all experiments converge to this number. During execution, and due to small variations on machine performance, FloodDQ further changes the number of nodes up or down. In all cases the query terminates under the 900-second deadline (finishing up between 807 and 889 seconds).

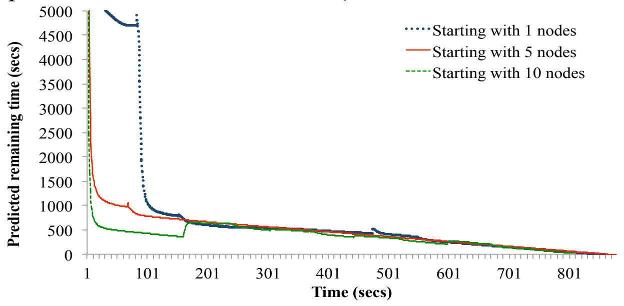

Figure 4. Estimated time left; varying initial resources

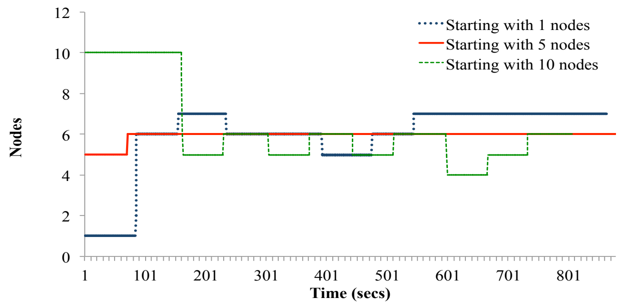

Figure 5. Number of nodes with variation on initial resources

C. Varying Dataset Size

In this experiment the dataset size is varied from 10 GBytes to 32 GBytes. Figures 6 and 7 plot the running estimated time left and number of nodes being used, respectively.

As it can be seen in Figure 6, regardless of the dataset size, and in spite of all runs starting with just one node, all queries finish at about the same time, just below the desired deadline. This is accomplished, as shown in Figure 7, by an increase in the number of used nodes. For example, processing 32 GBytes required an increase from one to seven nodes while processing 10 GBytes required just two nodes.

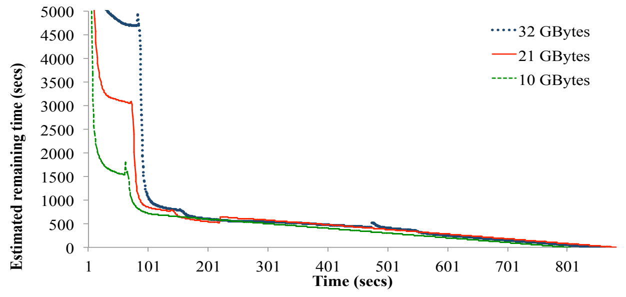

Figure 6. Estimated time left with variation on dataset size

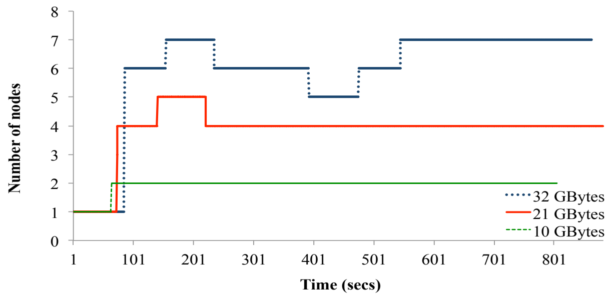

Figure 7. Number of nodes with variation on dataset size

D. Varying the Deadline

In the next experiment we varied the deadline itself, while keeping fixed the initial number of nodes (1) and the dataset size (32 GBytes). The results are presented in the next two Figures, 8 and 9.

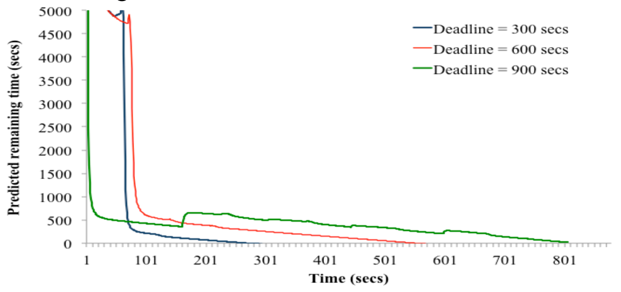

Figure 8. Estimated time left for varying deadlines

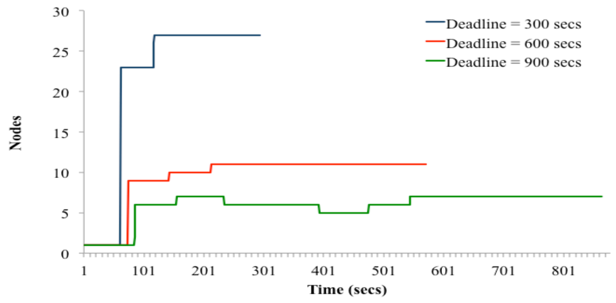

Figure 9. Number of nodes for varying deadlines

As expected, the shorter the deadline the sooner will queries complete. At the same time, the shorter the deadline, the more nodes will be requested to compensate for a shorter deadline.

E. *Dealing with Perturbations*

In this experiment we run a query similar to the ones in the previous sections but with 7 starting nodes (the expected number of nodes to finish the query under the deadline without requiring further changes). However, at about 30 seconds into execution a major perturbation was introduced in the system: in all 7 nodes a set of processes was run to consume CPU and IO using the `stress` command (10 workers per node stressing CPU and 10 other workers per node stressing IO). This perturbation lasted for 5 minutes (300 seconds) and is represented in both figures below.

It is interesting to note that shortly after the perturbation, FloodDQ scales the query a little, from 7 to 8 nodes. This makes sense because, in the very initial stages of the perturbation, the average performance per node (taking into account the recent history) is not that bad. However, as the perturbation continues to affect the performance (and the performance history of the nodes), and as soon as FloodDQ is able to take a new decision, more nodes will be requested. In fact, since all 7 nodes were heavily perturbed, both in terms of CPU and IO usage, one would expect FloodDQ to request up to 14 nodes (that is, 7 new, non-perturbed nodes). As Figure 11 shows, that is indeed what happens. Later, when the perturbation disappears, the number of used nodes returns to 7 as expected.

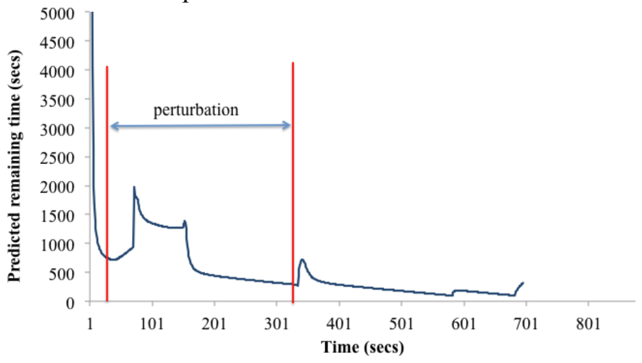

Figure 10. Estimated time left; perturbation experiment

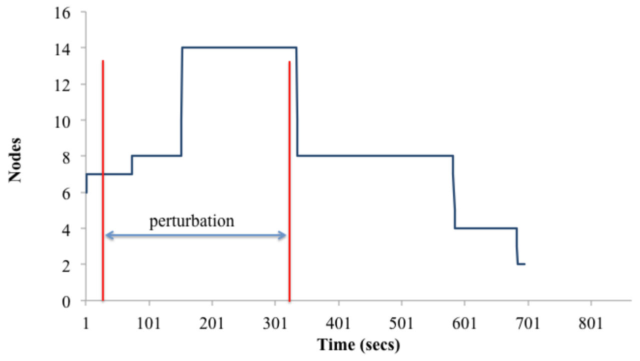

Figure 11. Number of nodes; perturbation experiment

Although not explicitly shown, this experiment also suggests that the system is able to adapt to unevenly distributed workloads, i.e., where nodes are submitted to uneven loads. This is due to the usage of content insensitive routing and partial aggregation, which guarantees that load, up to the total reduce, is distributed based on capacity and not on the data itself.

V. Related Work

Imposing hard time limits on the execution of single, large, queries is not a new concept. Hou W. *et al.* [1] first introduced the concept in 1989. This seminal work addresses hard time limits in query execution, specifically COUNT aggregations. In Hou's approach, sampling is used to reduce the amount of data to be processed by the COUNT aggregation, returning approximate query results. Contrasting with our approach, we add or reduce computing nodes to meet the deadline. Online aggregation [19] is another approach that, while not defining strict time limits, addresses the problem of long-running queries taking an unknown amount of time to complete. The novelty in this approach was making it possible for users to interactively specify *running confidence intervals* for the answers. Again, unlike our work, Online Aggregation effectively allowed to trade-off accuracy for speed, while FloodDQ maintains accuracy but increases or decreases used resources.

More recently, research at Oracle [2] further explored the concept of hard time limits. The novelty of the approach is the broadening of the concept to different types of aggregators and presenting a "working system". In this case hard time limits are met by using known aggregate query execution statistics and augmenting queries: with a ROWNUM clause (for partial results) and a SAMPLE clause (for approximate results). This approach presents the drawback of having to define the values of ROWNUM and SAMPLE clauses beforehand and thus was improved in [3] to allow progressive estimation of aggregates.

Early work on real-time databases addressed hard-time limits for concurrent query execution [20], by introducing real-time concurrency control and query scheduling mechanisms. Such work, however, relies on accurate knowledge of past executions and is mostly directed to assuring real-time constraints in concurrent query scenarios. In contrast, in our work the data initially unstructured, the operators have initially unknown execution profiles and focus is on the execution of a single, relatively long running, query.

To our knowledge, all previous work on hard-time limits on single query execution focused on the production of partial or approximate results. In our approach we do not sacrifice accuracy for speed; instead we use dynamic resource allocation to increase/decrease the computation capability of the cluster. In our approach, when estimates suggest that a deadline will be missed, we effectively scale the system so that it processes data faster, instead of reducing the amount of data to be processed. As results always present maximum accuracy, the concept of hard time limits is no longer solely applicable to approximate aggregates. In fact, in our approach, hard time limits may be applied to any generic select/project/aggregate computation. Moreover, in our approach, there is no need to know beforehand the performance of query execution, or the data characteristics. We argue that our approach is fundamentally different thus the use of the “new” term *deadline queries/computations*.

Estimating the progress of executing queries is something that our algorithm needs in other to decide whether to scale the system up or down. Previous work in this area [8][9][10][11][12] focused mostly on single site execution and heavily relied on previously collected operator statistics, which aren’t always available in the largely User-Function based MapReduce paradigm. Only very recently [13][14] has work appeared that addressed execution in MapReduce frameworks and even this work relied on pre-computation debug runs and was not directed at the streaming, dynamically provisioned, setting we employ.

Recently, Amazon launched an “Elastic MapReduce”[7] service, in which MapReduce computations are executed in the Apache Hadoop framework and over its EC2 product. This service, however, contrary to our approach is not dynamically provisioned in runtime according to the computation requirements, and it does not impose a deadline on the production of results. Instead it simply allows users to specify a cluster size that will be automatically allocated to the computation, allowing for inadequate provisioning.

VI. Conclusions and Future Work

In this paper we proposed a new type of queries: deadline queries, where used resources are varied (obtained or release) in order to guarantee that the query is able to terminate within a deadline regardless of the number of initial nodes, the size of the dataset, or the existence of perturbations. This approach is different from past work that guarantees hard-time limits on queries by using sampling or approximations. We showed an experimental evaluation of the system on a variety of scenarios. In all cases, FloodDQ, our research prototype, was able to terminate the query under the deadline constraints with the minimum set of nodes possible.

Future work will focus on queries with additional complexity, such as those that use joins. This will require us to build on the way that FloodDQ estimates query progress to account to more complex dataflows. Another future concern is that of better estimating total completion time by learning from observation the time it takes since a new (virtual machine) node is requested until it is granted. Future work will also focus on complete fault tolerance, especially in tolerating faults when they happen close to the deadline.

Aknowledgement

This work was partially supported by the Portuguese Research Agency – FCT, through a UT-Austin|Portugal PhD scholarship (SFRH/BD/51081/2010), CISUC (R&D Unit 326/97), and the Aeminium Project (CMU|Portugal project ref. CMU-PT/SE/0038/2008).

References

[1] Hou, W., Ozsoyoglu, G., and Taneja, B. Processing aggregate relational queries with hard time constraints. In SIGMOD '89.

[2] Hu, Y., Sundara, S., and Srinivasan, J. Supporting time-constrained SQL queries in oracle. In VLDB '07.

[3] Hu, Y., Hou, W., Sundara, S., and Srinivasan, J.. An experimental study of time-constrained aggregate queries. In EDBT '10.

[4] Hadoop. http://hadoop.apache.org/.

[5] Kavulya, S., Tan, J., Gandhi, R., and Narasimhan, P. An Analysis of Traces from a Production MapReduce Cluster. In CCGrid ‘10.

[6] Wang, G., Butt, A R., Pandey, P., and Gupta, K. A simulation approach to evaluating design decisions in MapReduce setup. In MASCOTS ‘09.

[7] Dean, J. and Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. In Usenix Annual Technical Conference (2004).

[8] Chaudhary, S., Kaushik, R., and Ramamurthy, R. When can we trust progress estimators for SQL queries? In SIGMOD '05.

[9] Chaudhary, S., Narasayya, V., and Ramamurthy, R. Estimating progress of execution for SQL queries. In SIGMOD '04.

[10] Luo, G., Naughton, J., Ellmann, C., and Watzke, M. Increasing the accuracy and coverage of SQL progress indicators. In ICDE ’05.

[11] Luo, G., Naughton, J., Ellmann, C., and Watzke, M. Toward a progress indicator for database queries. In SIGMOD '04.

[12] Mishra, C. and Koudas, N. A Lightweight Online Framework For Query Progress Indicators. In ICDE 2007.

[13] Morton, K., Friesen, A., Balazinska, M., and Grossman, D. Estimating the progress of MapReduce pipelines. In ICDE ’10.

[14] Morton, K., Balazinska, M., and Grossman, D. ParaTimer: a progress indicator for MapReduce DAGs. In SIGMOD '10.

[15] Logothetis, D. and Yocum, K. Ad-hoc data processing in the cloud. Proceedings of the VLDB Endowment, VOL. 1, NO.2 (Aug 1, 2008).

[16] Condie, T., Conway, N., Alvaro, P., Hellerstein, J., Elmeleegy, K., and Sears, R. MapReduce online. In NSDI'10.

[17] Arpaci-Dusseau, R., Anderson, E,. and Treuhaft, N. Cluster I/O with River: Making the fast case common. In IOPADS ‘99.

[18] Shah, M., Hellerstein, J., Sirish C., and Franklin, M. Flux: an adaptive partitioning operator for continuous query systems. In ICDE ’03.

[19] Hellerstein, J., Haas, P., and Wang, H. Online aggregation. In SIGMOD '97.

[20] P S Yu, Kun-Lung Wu, Kwei-Jay Lin, and S H Son. On real-time databases: concurrency control and scheduling. Proceedings of the IEEE. VOL. 82, NO. 1 (1994), 140-157.

[7] http://aws.amazon.com/elasticmapreduce/