IncReStore: Incremental Computation of MapReduce Workflows*

Ahmed E. Khalifa†+
Alexandria University
ahmed.elmorsy@alexu.edu.eg

Iman Elghandour+
Alexandria University
ielghand@alexu.edu.eg

Nagwa El-Makky
Alexandria University
nagwamakky@alexu.edu.eg

***Abstract*—Many applications in various industrial and research areas analyze large continuously evolving data. Big data analytics platforms such as MapReduce focus on distributed batch processing, and therefore, a query needs to be re-executed every time its input data evolve. In this paper, we present IncReStore, a system that incrementally computes queries on fast growing datasets by materializing query outputs and maintaining them. IncReStore runs in two modes: (1) Opportunistic IncReStore generates compensating queries on the fly during their execution to use previously materialized query outputs taking into account that data might have evolved; and (2) Active IncReStore automatically generates MapReduce jobs to update the materialized query outputs whenever the datasets that they depend on evolve. We have implemented IncReStore as an extension to Pig and Hadoop. Our experimental evaluation of IncRestore using the TPC-H benchmark shows significant speedups.**

I. Introduction

Analyzing large data has been the focus of many enterprises and research teams. Many of the applications employed by these groups fall in the categories of batch processing and ad-hoc analysis [15]. For over a decade, distributed batch processing platforms, such as MapReduce [4] and its open source implementation Hadoop [1], have been successfully used for batch and ad-hoc jobs. Nowadays, many applications require analyzing continuously evolving data, sometimes at high rates [2]. The common features of these applications are: (1) the input data evolve periodically or at random intervals, and (2) the execution performance of these applications is important.

Distributed batch processing platforms are not designed to process data that evolve at high rates. When new data are appended to the system, continuous and recurring queries have to be executed on the evolved datasets. This is inefficient because the processing platform ignores the outputs of these queries on older versions of the input datasets. Incremental approaches that are based on MapReduce have been developed to efficiently process evolving data [3], [10], [12], [16]. All these approaches have shown significant improvement over Hadoop in processing continuous queries. However, they are all query centered, which means that queries that are marked as continuous only benefit from the incremental query processing approaches offered by these systems. An alternative approach for incremental query processing is data centered, in which the platform keeps track of the datasets and the outputs of queries that consumes them as input. When the datasets evolve, the existing materialized outputs are consequently updated or scheduled for future update.

In this paper, we present IncReStore, a system that takes advantage of the materialized outputs of previously executed queries to incrementally compute queries on fast growing datasets. It leverages ReStore [5], a system that materializes the outputs of queries and sub-queries that it executes and reuses them to answer future queries. The new additional features provided by IncReStore is that it maintains query outputs that are materialized in the system whenever the datasets input to those queries evolve, and it is capable of incrementally processing queries using materialized outputs. This allows IncReStore to benefit three types of queries executed on evolving datasets: (1) new queries that can be rewritten to use materialized data, (2) continuous queries that need their outputs to be updated every time their input evolve, and (3) recurring queries that are executed periodically and therefore, IncReStore materializes their output, and when these queries recur, the new output is computed from the previously materialized output.

We present two modes of IncReStore for incrementally processing queries. The first mode is Opportunistic IncReStore that is useful for new and recurring queries with large recurrence intervals. When a query of these types is submitted for execution, IncReStore compiles the query into a query execution plan and identifies parts of this plan that can reuse existing materialized query outputs. However, since the datasets that are input to this query have changed, Opportunistic IncReStore rewrites the query execution plan to exploit the outdated materialized query outputs and the new delta data appended to the query input datasets. It additionally updates the materialized query outputs used to rewrite the query. The second mode is Active IncReStore that is useful for continuous queries and recurring queries with small recurrence intervals. When one or more datasets evolve, Active IncReStore spontaneously generates compensating queries that incrementally update continuous queries and all the materialized query outputs that depend on the updated datasets.

The main contributions of this paper are as follows:

- A framework that leverages the query reuse mechanism presented in ReStore [5], maintains the materialized query outputs when the input datasets of these queries change, and generates compensating queries (workflows of MapReduce jobs) to incrementally com-

*This work was supported by a Microsoft Azure Research Award
+ The authors were partially supported by SmartCI research center
†The author is now at Google Inc.

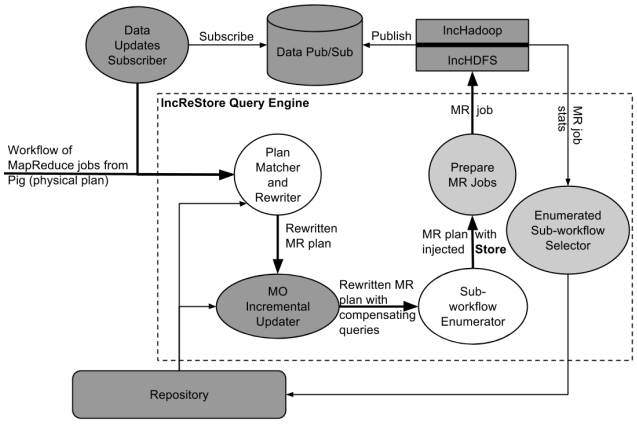

Fig. 1. IncReStore system architecture.

pute queries using these materialized query outputs using two approaches of incremental processing: opportunistic and active (Sections II and III).

- An extended version of the Hadoop Distributed File System (HDFS) that publishes information about new data that are appended to the datasets stored on it and caches this newly appended data in the memory of the Hadoop cluster nodes, in addition to an extended version of Hadoop that schedules workflows of MapReduce jobs on the cluster nodes caching the new data to update query outputs materialized in the system. (Section IV).
- An implementation of IncReStore and an extensive set of experiments that evaluates the new proposed frameworks and approaches (Section V).

II. Overview of IncReStore

Pig [6] compiles an input query into a workflow of MapReduce jobs. IncReStore extends Pig as it leverages ReStore [5], a system that improves the execution performance of workflows of MapReduce jobs by materializing the intermediate results of executed workflows and reusing them for future workflows submitted to the system. However, datasets that are input to queries evolve, and hence materialized query outputs become outdated and we need to recompute the outputs of continuous and recurring queries. IncReStore addresses this challenge by tracking data changes, updating materialized query outputs, and consequently using these materialized outputs to incrementally process queries.

In Section II-A, we briefly describe ReStore. Next, in Section II-B, we present the architecture of IncReStore and its two operation modes.

A. Overview of ReStore

ReStore [5] is a system that materializes and manages the results of previous computations of MapReduce jobs and sub-jobs to reuse them for future queries submitted to the system. ReStore extends Pig [6], which is a dataflow language processor that translates a query written in a high level language, Pig Latin, into a workflow of MapReduce jobs. The input to ReStore is the workflow of MapReduce jobs generated by Pig for an input query. ReStore performs the following: (1) it rewrites the MapReduce jobs to reuse outputs previously materialized in the system (`Plan Matcher and Rewriter`), (2) it stores the outputs of executed jobs in the workflow for future reuse (`Enumerated Sub-workflow Selector`), and (3) it creates more reuse opportunities by storing the outputs of selected sub-jobs/sub-workflows (`Sub-workflow Enumerator`). The outputs of computations (jobs, sub-jobs, workflows) that are materialized by ReStore for the input query are stored in the ReStore repository along with metadata about their execution statistics and the physical plans that are used to generate them. Figure 1 shows the system architecture of IncReStore including the components inherited from ReStore.

ReStore evicts the queries and their outputs that are materialized in its repository if the input datasets of these queries are modified. Therefore, ReStore mainly supports applications that process static data. IncReStore extends ReStore to allow it to maintain the materialized query outputs and to incrementally process queries. Note that the outputs of the queries materialized in the repository can represent the entire query submitted to ReStore or part of the query (sub-query). In the rest of the paper, we refer to it as materialized query since a part of a query submitted to ReStore/IncReStore (sub-query) is a query on its own.

B. IncRestore System Architecture

Figure 1 shows the system architecture of IncReStore, the new components added to Pig, and how it is integrated with IncHadoop / IncHDFS (our extended versions of Hadoop and HDFS as we describe them later in this section and in Section IV). IncRestore leverages the main components of ReStore [5] to materialize query outputs and reuse them to rewrite future queries. IncReStore takes the workflow of MapReduce jobs compiled by Pig and rewrites it to reuse any query outputs that were previously executed and materialized in the system. If the input datasets of the query that IncReStore is materializing its output have evolved, the query's materialized output becomes invalid. IncReStore updates this materialized output to reflect the new delta data that have been appended to the datasets. Additionally, IncReStore incrementally process queries using the materialized query outputs maintained in its repository. Note that we are using dark and light grey colors in Figure 1 to highlight the new components that we have added to or changed from ReStore, respectively.

IncReStore, similar to ReStore (Section II-A), keeps a repository to manage the materialized outputs of queries that were previously executed in the system. For each materialized query output, IncReStore keeps the following: (1) the physical query execution plan that corresponds to this query (organized as a workflow of MapReduce jobs), (2) the output path on HDFS, (3) statistics about the execution of this query on Hadoop (for example, execution time and output size), and (4) its input datasets, their paths on HDFS, and metadata about them. Note that every query execution plan stored in the repository has one or more input datasets and one output. A general query (i.e. Pig query) submitted to the system can have multiple outputs. However, we generate multiple entries in the repository, one for each materialized output. Therefore, we consequently store in any entry of the repository the part

of the query that produces this specific output in the form of its physical execution plan.

In addition to the components that IncReStore inherits from ReStore that are described in Section II-A, IncReStore has the following two new components: (1) `Materialized Output Incremental Updater` and (2) `Data Updates Subscriber`. Additionally, we extend two components of ReStore as follows: (1) `Prepare MapReduce Jobs` to inject information in the MapReduce jobs about the parts of the data to process. This is useful for the queries that we run to update the materialized query outputs; and (2) `Enumerate Sub-workflow Selector` to only materialize parts of the queries that are submitted to and executed in the system and ignore those used to maintain existing materialized query outputs. Moreover, for Active IncReStore to perform its functionality, we extended Hadoop and HDFS to allow Hadoop to process new data appended to a dataset stored in HDFS while these new data are cached in the memory of the cluster nodes. We propose two IncReStore modes that are suitable for various types of queries: Opportunistic IncReStore and Active IncReStore.

New queries and recurring queries with large recurrence intervals benefit from Opportunistic IncReStore. The objective of this mode of IncReStore is to update materialized query outputs only when it uses them to rewrite a query. For every query submitted to IncReStore, `Plan Matcher and Rewriter` searches the repository for materialized query output(s) that can be used to rewrite the input query and uses the matched materialized outputs to rewrite the query. If the matched materialized outputs are invalidated because the input datasets used to generate them have changed since the materialized outputs last update, IncReStore takes the change in the input datasets into consideration. Therefore, `Materialized Output Incremental Updater` rewrites the query execution plan to consider the new delta appended to the input datasets. The new query rewriting exploits both the outdated materialized query outputs and the new added data. Next, the query is executed using Hadoop and as an additional benefit, the materialized query outputs are updated. Finally, if it is the first time that IncReStore sees this query, the enumerated sub-workflow selector component of IncReStore chooses parts of the query to materialize their outputs.

Furthermore, we need to update the output of continuous queries once new data are streamed to the system. Queries recurring at fast rates also benefit from actively updating the materialized query outputs that they can use frequently to avoid any delays at the time of their execution. Active IncReStore addresses these requirements by immediately updating the materialized query outputs once a dataset that affects them evolves. Our solution includes an extended version of Hadoop/HDFS that caches new data appended to datasets stored on them and executes queries to update the affected materialized query outputs. IncReStore uses `Data Updates Subscriber` to subscribe its materialized query outputs to changes made to the datasets that they depend on. When new data are added to one of the datasets stored in IncHDFS, it caches the new data in the memory of the nodes of the cluster and sends a message to the `Pub/Sub` server that this dataset is updated. Next, `Data Updates Subscriber` submits dummy queries to IncReStore to force it to update the affected materialized query outputs. The updating process of the materialized query outputs follows the same procedure as the one we use for Opportunistic IncReStore. Next, we describe how IncReStore updates materialized query outputs and our extensions to Hadoop and HDFS.

III. Incrementally Updating Materialized Query Outputs

The first phase of IncReStore is `Plan Matcher and Rewriter`, which searches the IncReStore repository for materialized query outputs that can be used to rewrite an input query, and then rewrites the query to use these outputs. `Materialized Output Incremental Updater` checks the freshness of the materialized outputs that `Plan Matcher and Rewriter` used to rewrite the query. If one or more of these materialized outputs depend on input datasets that have evolved, it rewrites the query to include additional operations to update the outdated materialized outputs and incrementally compute the output of the input query from them. We also refer to this rewritten execution plan as the compensating query.

We add a new feature to the physical operators built in Pig. We now specify the start and end offset of the input data that this operator consumes. This creates three modes of operation for a physical operator: (1) ALL: the operator processes all the data in the dataset, (2) NEW: the operator processes new delta data appended to the dataset, and (3) OLD: the operator processes data in the dataset before appending the delta data to it. `Materialized Output Incremental Updater` specifies one of these modes for each operator in the physical plan used to update the materialized output.

In order to generate the execution plan for updating an outdated materialized output, `Materialized Output Incremental Updater` scans the physical plan of the materialized output starting from the root (`Store` operator) and generates an update plan for each operator. The techniques used for generating update plans for various operators are similar to those presented in [14] for SQL. We specify for each operator whether it is consuming OLD, NEW, or ALL data pipelined from its predecessor operator. We differentiate between four types of relational operators:

- `Load`, `Store`, `Filter`, `Project`, `Union`, and `Split`. We update these operators to process NEW data. Their output is appended to the output of the same operator on OLD data.
- `Join` and `Cross`. These operators have two inputs pipelined from a predecessor operator in the execution plan. The physical plan to update a `Join` operator includes the `Union` of the outputs of the following four `Join` operations: (1) joining OLD from the two inputs, (2) joining NEW from the two inputs, (3) joining NEW from the first input and OLD from the second input, and (4) joining OLD from the first input and NEW from the second input.
- `Order`, `Distinct`, and `Limit`. The operator is replaced with the `Union` of the operator executed on OLD and the operator executed on NEW, followed by re-applying the operator on the output.

- `Group` and `Cogroup`. The operator is replaced with a merge of the output of the operator on NEW and the output on OLD. The outputs of these operators are represented as maps of group keys to bags of elements. We merge the outputs by merging the bags of elements that belong to the same group key.

Moreover, IncReStore supports aggregation operators such as `Sum`, `Average`, and `Count`. We store additional metadata about aggregation operators. When we create an execution plan for the compensating query that updates the materialized outputs, we use this metadata to calculate the result of the aggregation operators instead of re-executing it. For example, we store the total sum and number of dataset records for an `Average` operator. Next, we calculate the sum of NEW, add it to the value stored in the repository, and divide it by the total number of records in ALL.

We note that the final plan subsumes the plan that was previously executed on OLD data, and therefore this part of the plan is replaced with the previously materialized output. The other part of the plan is the one we execute to update the materialized output.

IV. In-Memory Incremental Processing of MapReduce Jobs

Opportunistic IncReStore operates with the Hadoop / HDFS stack to incrementally update the materialized query outputs. It employs the `Materialized Output Incremental Updater` described in Section III to update the materialized outputs when they are used to rewrite other queries. This makes Opportunistic IncReStore a best fit for updating materialized outputs that are not frequently accessed. However, for materialized outputs that are accessed frequently or every time the input datasets are updated (e.g., concurrent queries), it is better to actively update the materialized outputs momentarily when the input datasets evolve. To achieve this design objective, we need to detect that new data are appended to a dataset in HDFS, and update all the materialized outputs that depend of this dataset. There are two techniques to detect updates in datasets stored in HDFS: (1) Pull-based solution, in which IncReStore periodically checks all the datasets stored on HDFS and that are related to materialized outputs stored in its repository to detect the outdated ones; and (2) Push-based solution, in which HDFS sends notification to IncReStore when a dataset evolves. The pull-based solution adds unnecessary overhead due to the redundant requests. Therefore, we choose the push-based solution.

Another design goal that we want to achieve is that IncReStore efficiently updates the materialized outputs when the input datasets to their queries evolve. However, Hadoop reads the input of a job from HDFS and writes its output to HDFS. To address this challenge, we extend HDFS to cache new data appended to a dataset in the memory of the cluster nodes and extend Hadoop to assign job tasks that updates the materialized outputs to run on the nodes that caches these data. Next, we describe how we extended Hadoop and HDFS to meet these design goals.

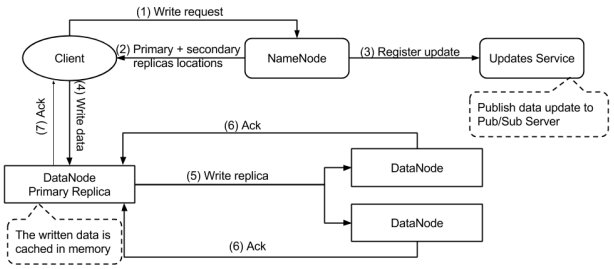

Fig. 2. Control flow of write operation in IncHDFS.

A. IncHDFS

HDFS follows the Google File System (GFS) [7] in its mechanism of writing data. It employs a master/slave architecture, where the master is the NameNode and the slaves are the DataNodes. Metadata about the DataNodes and the data stored on them are kept at the NameNode. A data file stored on HDFS is divided into data blocks. Each data block is replicated and stored on multiple DataNodes. One of these replicas is marked as primary, and the DataNode storing it becomes responsible for maintaining the consistency of all replicas of this particular data block. We introduce a new lightweight service called UpdateService in IncHDFS to run on the NameNode and to hold metadata about new data updates to existing datasets. This metadata includes where HDFS caches the new data. UpdateService is also responsible for sending push notifications about datasets updates. The control flow of the write operation as modified by IncHDFS is shown in Figure 2. The steps are as follows:

1) A client contacts the NameNode to request appending data to a specific dataset.
2) For each data block to be appended to a dataset, the NameNode replies to the client with the locations of the DataNodes to store the new data block on them including which of them is the primary replica.
3) The NameNode sends a message to the UpdateService to notify it about the requested update of the dataset. The UpdateService logs metadata about the dataset and the DataNode with the primary replica for each new data block. The UpdateService sends a push notification about this update to IncReStore.
4) The client pushes the data to all location of replicas (not shown in the figure). Then, the client sends a write request to the DataNode with the primary replica. This DataNode caches the new data block in its memory.
5) The DataNode with the primary replica forwards the write request to all DataNodes with replicas stating where to append the new data block.
6) The DataNodes write the data on disk then send acknowledgement to the primary replica that the write operation is completed.
7) The DataNode with the primary replica replies with an acknowledgment to the client.

The main extensions to HDFS implemented in IncHDFS are (1) publishing notifications about data changes by the new UpdateService module, that we added to the NameNode (Step 3), and (2) caching the new appended data to the DataN-

odes storing the primary replica (Step 4). Each data block is cached only in the memory of the DataNode containing the primary replica to guarantee that this solution works for large appended data and small caches at the DataNodes. However, if the DataNode storing the primary replica fails, the cached data are lost, and we read the data from the disk of one of the stored replicas. Additionally, we note that the allocated cache at each DataNode has a limited size. Therefore, if the cache becomes full, some of the cached data blocks are lost and they are read from disk. Finally, We note that changes we made to the HDFS mechanism does not affect its fault tolerance, and therefore IncHDFS acts the same way as HDFS in case of failures. Next, we describe in Section IV-B how IncHadoop schedules tasks to run on nodes where data are cached.

B. IncHadoop

Our main objective is to make Hadoop executes the MapReduce jobs generated by Active IncReStore to update the materialized outputs on the same cluster nodes where the new data are cached. We call the extended version of Hadoop that performs this IncHadoop. This new objective introduces two challenges: (1) IncHadoop need to schedule the tasks that executes the MapRduce jobs submitted by Active IncReStore on the same nodes that caches this new data in their memory and (2) we need to allow IncHadoop tasks to read data from memory instead of disk (local or remote). Addressing these challenges allows low latency in-memory processing of data while they are being written on IncHDFS. We extend Hadoop as follows:

- We update the Hadoop scheduler to schedule the tasks of the MapReduce jobs submitted by Active IncReStore on the same nodes where the appended data blocks are cached. To schedule the tasks of a MapReduce job on these DataNodes, the IncHadoop scheduler queries the UpdateService for the locations of these DataNodes. It then schedules the tasks to be executed on these DataNodes. This approach achieves low latency compared to reading the data from disk.
- We add a new InMemoryInputFormat to IncHadoop, which extends the FileInputFormat of Hadoop. This new input format allows IncHadoop tasks to read their input from the memory of the DataNodes on which they are scheduled to run instead of reading their input data splits from disk.

IncHadoop and IncHDFS are essential for Active IncReStore to update its materialized query outputs once their input datasets are updated. IncHDFS caches the new data blocks appended to its datasets in the memory of its DataNodes and publishes information about the datasets that have evolved and the data nodes caching the new data blocks. The `Data Updates Subscriber` keeps track of which materialized outputs are affected by which datasets and generates dummy queries and submits them to Active IncReStore to trigger it to update the materialized outputs that are affected. Active IncReStore prepares the MapReduce jobs that update these materialized outputs using the approach described in Section III and submits them to IncHadoop. Finally, IncHadoop schedules the tasks that execute these MapReduce jobs on the DataNodes with cached data blocks. We have effectively instrumented IncHadoop to perform in-memory incremental data processing.

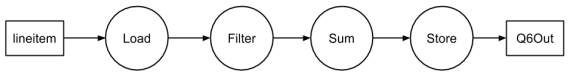

Fig. 3. Q6 Physical plan.

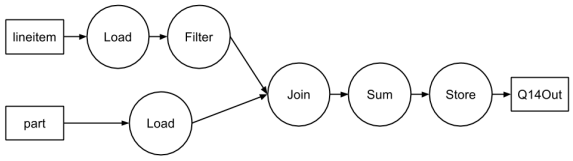

Fig. 4. Q14 Physical plan.

V. Experiments

IncReStore extends ReStore [5], which is based on Pig 0.10, to add support for incremental query processing. IncHadoop and IncHDFS extend Hadoop 2.4.1 and its HDFS module, respectively. In this paper, we focus on datasets that evolve by appending new data to them, which is supported by Hadoop.

We conducted our experiments on a cluster of 15 Microsoft Azure[1] virtual machine instances (nodes) of type A2. Each node has 2 cores, 3.5 GB of memory, and two disks of sizes 60GB and 80GB. Each node is running Ubuntu 14.04 LTS. HDFS and IncHDFS are configured to create 3 replicas of each data block. One of the nodes of the cluster is dedicated to run the *Name Node* and the *Resource Manager*.

We use the TPC-H benchmark in our experiments. We generated a data instance of size 350 GB using the TPC-H data generator and we divided this instance into multiple chunks each is 50 GB to evaluate incremental updates as described in Section V-A. In the experiments, we use the 22 queries of the TPC-H benchmark[2]. However, for the experiment discussed in Section V-A, we focus on two queries (Q6 and Q14) to test specific features of IncReStore. Our main metric is the average execution time of queries as measured by Hadoop and IncHadoop. Each reported result in our experiments is based on the average of five runs.

A. Effectiveness of Incremental Query Processing in IncReStore

In this section, we evaluate the effectiveness of the two modes of IncReStore: opportunistic and active for incremental query processing. We use two TPC-H queries (1) query Q6 as an example of aggregation queries and (2) query Q14 as an example of join queries. Figures 3 and 4 show the physical execution plans for Q6 and Q14, respectively. Initially, we store a 50 GB of the data on HDFS (IncHDFS in case of Active IncReStore), and we execute each query using four different platforms: Pig, ReStore, Opportunistic IncRestore, and Active IncRestore. In the case of ReStore and IncReStore, query and sub-query outputs are materialized for future use. Next, we incrementally append data to those stored on HDFS and used by the queries, 50 GB each time. IncReStore objective is to

[1]Available at: portal.azure.com

[2]TPC-H queries written for Pig are available at: https://www.cs.duke.edu/starfish/mr-apps.html

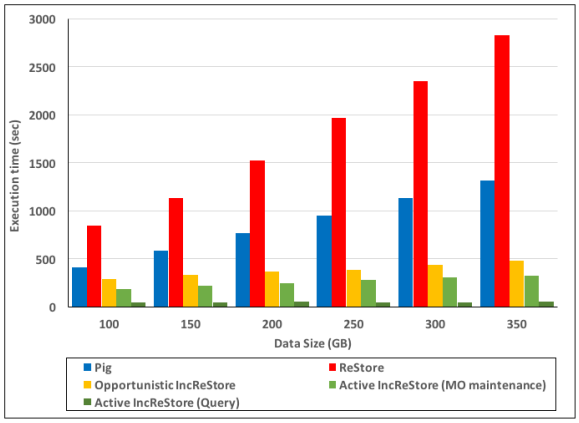

Fig. 5. Q6 Query execution times.

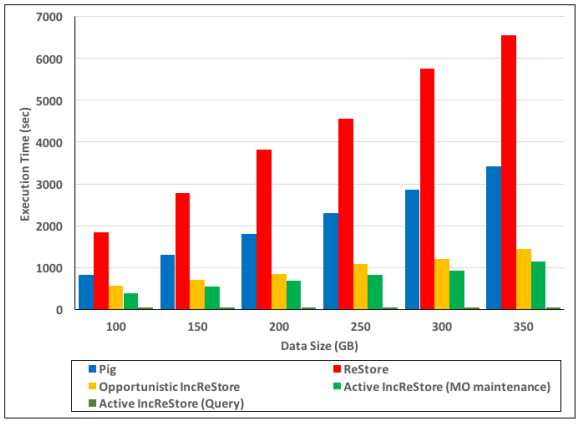

Fig. 6. Q14 Query execution times.

enhance the execution time of the queries that frequently recur, and therefore, we re-execute the queries each time new data are appended using the different platforms and measure the execution time taken by each of these platforms.

Figures 5 and 6 show the result of comparing the execution times of the TPC-H queries Q6 and Q14, respectively on the four platforms when the data grows from 50 GB to 350 GB. Pig is not aware that it has executed the queries before on part of the data, and therefore it re-executes the entire query. ReStore materializes outputs from previous query runs, but since the query input data have changed, these materialized query outputs are invalidated, and ReStore re-executes the queries. Therefore, the execution time taken by Pig and ReStore increases linearly when the data sizes increase. However, the execution time taken by both Opportunistic IncRestore and Active IncRestore for different input query sizes is slightly affected by the change of the data size because it employs the proposed techniques for incremental query execution.

It is shown in [5] that the execution time of queries taken by ReStore is lower than that taken by Pig because ReStore only incurs overhead for materializing outputs the first time a query is executed and it benefits from these outputs in subsequent runs reducing the total execution time. However, if the data are evolving, as shown in this experiment, ReStore performs poorly because the overhead of materializing outputs is incurred with every execution of the query. IncReStore

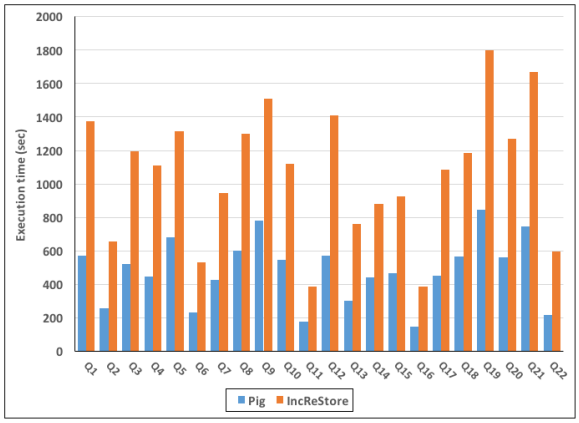

Fig. 7. Overhead introduced by IncReStore.

leverages the query outputs reuse in ReStore while addressing the evolution of the data. Both modes of IncReStore materialize outputs the first time the queries are executed, and incrementally update these materialized outputs when the data evolve (Active IncRestore) or when a query access outdated materialized outputs (Opportunistic IncRestore).

Figures 5 and 6 show that the speedups of Opportunistic IncReStore are 4 and 2.4 for the TPC-H Q6 and Q14, respectively, and that the speedups of Active IncRestore are 15 and 4 for the TPC-H Q6 and Q14, respectively. In both cases we are calculating the speedup as the query execution time taken by IncReStore (opportunistic or active) divided by the query execution time taken by Pig. The speedups of Active IncRestore are higher than Opportunistic IncReStore because most of the processing is done offline when the data are incrementally updated to subsequently update the materialized outputs that the query can use. In the figures, note that dark light green bars represent the execution of the query using already updated materialized outputs, and the light green bars represent the execution time required for updating these materialized outputs. Moreover, the IncReStore speedup for the join query (Q14 as shown in Figure 6) is less than the speedup for the aggregation query because the update plan for a join query is more complex (Section III).

B. Overhead Introduced by IncReStore

In this experiment, we evaluate the overhead introduced by IncReStore when it executes an input query for the first time to materialize its outputs and to add meta-data about the query in the repository. We execute the 22 queries of the TPC-H benchmark on a dataset of size 50 GB using Pig and incReStore.

Figure 7 shows the execution times of the 22 TPC-H queries taken by Pig and IncReStore. The difference in the execution times on the two systems reflects the overhead incurred by IncReStore for materializing query and sub-query outputs. The overhead (the execution time of the query taken by IncReStore divided by the execution time of the query taken by Pig) introduced for different queries ranges between 1.8 and 2.9 with and average of 2.2. Given the gain we achieve when we use IncReStore by incrementally maintaining

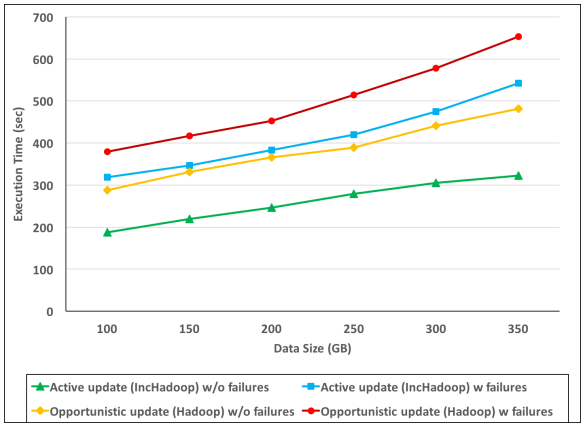

Fig. 8. Fault tolerance maintained by IncHadoop.

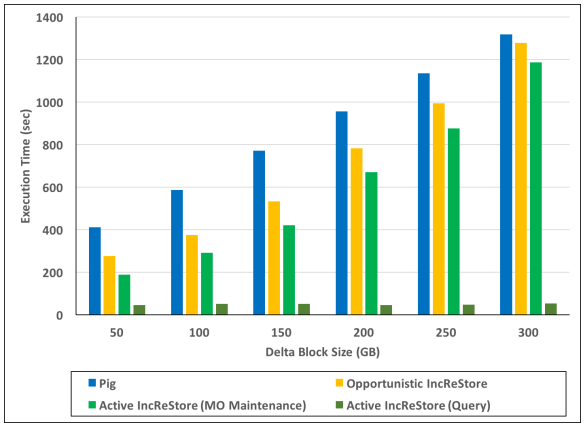

Fig. 9. Q6 execution time using different delta block sizes.

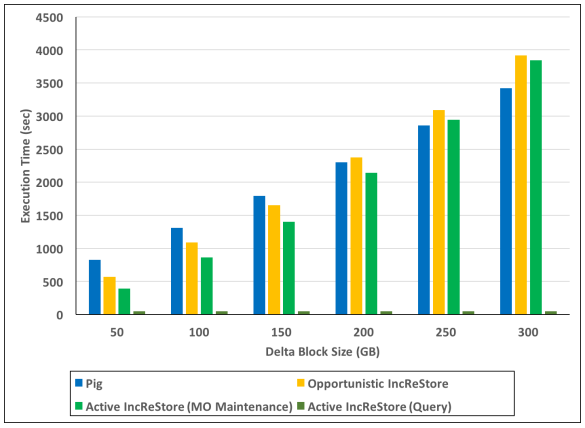

Fig. 10. Q14 execution time using different delta block sizes.

the materialized outputs and using them for future queries as discussed in V-A, the overhead can be tolerated.

C. IncHDFS/IncHadoop Fault Tolerance

Both IncHDFS and IncHadoop maintain the same fault tolerance capabilities of HDFS and Hadoop, respectively. The main difference between Hadoop and IncHadoop is that IncHDFS caches newly appended data for IncHadoop to execute jobs that update materialized query outputs. To evaluate the performance of IncHadoop in this case, we design this experiment as follows. IncReStore first executes the TPC-H query Q6 on data size of 50 GB stored on IncHDFS to generate the initial materialized outputs for the query. Next, we keep appending data chunks of size 50 GB each until the total size of the data becomes 350 GB. Each time we append data, Active IncReStore triggers updating materialized outputs by starting a job to be executed by IncHadoop. To simulate node failures, we kill a random slave node in the cluster every 10 seconds during the query execution. For comparison, we repeat the experiment using Opportunistic IncReStore which runs on top of Hadoop/HDFS.

In Figure 8, we compare the execution times taken by IncHadoop to update the materialized outputs generated for Q6 in these two cases: (1) failures and (2) no failures. As expected, the execution time increases when there is failures occurring due to rescheduling failed tasks. However, the slow down in the execution of the query, which is calculated as the ratio of the execution time of IncHadoop (no failures) divided by the execution time of IncHadoop (failing nodes), is on average 0.7. In addition of the node failures experienced by Hadoop, we identify a new type of failure in IncHadoop, which is a failure of a node that is caching a new data block. In this case, the new scheduled task reads the data block from disk instead of the cached copy. The latency in the execution is due to reading from disk and rescheduling the task. It is important to note here that the execution times incurred by Active IncReStore in the presence of failure is less than that incurred by Opportunistic IncReStore in the presence of failure (represents Hadoop fault tolerance) and close to Opportunistic IncReStore with no failure.

D. Effect of Size of Appended Data on IncReStore

IncReStore materializes outputs of executed queries and sub-queries for future use, and it maintains these materialized outputs when new data are appended to their inputs. The question we are trying to answer using this experiment is when it is better to use IncReStore to incrementally update these materialized outputs and when it is better to execute the query on the new data. Initially, we execute the the TPC-H queries Q6 and Q14 on data of size 50 GB. We then append blocks of the data of various sizes that range from 50 GB to 300 GB to the original 50 GB, and we execute the queries each time and measure the total time needed.

Figures 9 and 10 show the effect of the size of data appended to the datasets on the time required for incrementally updating the materialized query outputs that are used by queries Q6 and Q14, respectively. As expected, when the size of the data appended to the datasets increases, the speedup of IncReStore as compared to Pig decreases until it becomes more beneficial to execute the query on the updated data than to employ incremental processing techniques on the priory materialized outputs. For Q14, which is a join query, this decision should be made when the appended data size is 4 times the original data size (Figure 10). Figure 9 shows that Q6, which is an aggregation query, would always benefit from incremental query processing because the execution of the query on the old data is saved.

VI. Related Work

MapReduce [4] distributed computing platforms such as Hadoop and dataflow SQL-like language processors such Pig [6] have been successfully used for batch processing of complex analysis tasks. However, they are not suitable for continuous and frequently recurring queries on data that evolve. IncReStore materializes and maintains outputs of the queries that are executed and also intermediate outputs of some parts of them.

There have been several research works that addressed incremental processing for (1) MapReduce (e.g., [3], [10], [16]), (2) dataflow systems running on top of MapReduce (e.g., [12]), and (3) distributed computing platforms other than MapReduce (e.g., [8], [9], [13], [17]). The work that is related most to ours is Nova [12]. Nova is a new layer on top of Pig in the Hadoop stack that stores outputs of continuously running queries. It employs techniques similar to view maintenance to update the query outputs. In contrast, IncReStore is tightly integrated with Pig/ReStore, which allows it to leverage cost optimizations implemented in their logic and a full-fledged query rewriting mechanism using materialized outputs from previous query runs. Moreover, IncReStore has two modes that actively/lazily update the materialized outputs when the data evolve.

Each of Redoop [10], Incoop [3], and IncMR [16] extends Hadoop to add support for incremental processing of recurring queries. However, these systems allow queries to benefit from either previous query outputs (Redoop) or map outputs (Incoop, IncMR). On the contrary, IncReStore is integrated with Pig/ReStore which allows more optimization opportunities and matching capabilities. Moreover, IncReStore allows queries executed for the first time in the system to reuse materialized outputs generated as a result of executing different queries.

Spark [17] is a cluster computing framework that stores data objects called Resilient distributed datasets (RDDs) in the memory of cluster nodes to allow reusing them in different iterations of the queries. However, data reuse is limited to queries running within the same session, and Spark does not currently automate the discovery and reuse of already computed RDDs.

For the Dryad execution platform, Nectar [8] (earlier related contributions [9] and [13]) builds a server to cache outputs of shared jobs and sub-jobs executed in the system. It supports incremental processing to update materialized data. However, Nectar lacks the optimizations and matching capabilities provided by IncReStore. Moreover, IncReStore is tailored for MapReduce systems.

To achieve the most efficient operation of IncReStore, we have extended Hadoop/HDFS to cache data on cluster nodes and execute MapReduce jobs on them to update materialized outputs. A related work is, Tachyon [11], which supports high throughput in-memory caching of the file system and uses the lineage technique to guarantee fault tolerance. This deems Tachyon a general purpose platform, however, IncHadoop/IncHDFS is a tailored light-weight hack of Hadoop/HDFS that allows HDFS to temporarily cache new data added to it until MapReduce jobs finish processing these data.

VII. Conclusions

Many applications require executing continuous and recurring queries on incrementally updated data. It is the responsibility of the system to perform this computation efficiently and with the least possible latency. In this paper, we present IncReStore, a system that incrementally executes new, continuous, and recurring queries that have input datasets that evolve over time. IncReStore exploits materialized query outputs of previous runs of the input query or other queries. When the data evolve, IncReStore maintains these materialized query outputs by generating update plans that integrates outputs on the newly added data and previously materialized data. This maintenance procedure is executed when a materialized output is outdated and is needed for rewriting a query in the Opportunistic IncReStore mode. To further reduce the latency of IncReStore and to make it suitable for continuous queries, we have extended Hadoop/HDFS to cache new data updates in the memory of cluster nodes and execute queries on the cached data to update the materialized outputs. This technique is employed by the Active IncReStore mode. We evaluate our implementation and demonstrate the effectiveness of the proposed approaches using a comprehensive experimental study. We show that the execution of the continuous queries on IncReStore can be 4 times faster in case of Opportunistic IncReStore and 15 times faster in case of Active IncReStore than Pig that re-executes the queries.

References

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

[2] M. Ahuja et al. Peta-scale data warehousing at Yahoo! In *Proc. ACM SIGMOD*, pages 855–862, 2009.

[3] P. Bhatotia et al. Incoop: MapReduce for incremental computations. In *Proc. ACM SOCC*, pages 7:1–7:14, 2011.

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

[5] I. Elghandour and A. Aboulnaga. ReStore: Reusing results of MapReduce jobs. *Proc. VLDB Endow. (PVLDB)*, 5(6):586–597, 2012.

[6] A. F. Gates et al. Building a high-level dataflow system on top of Map-Reduce: the Pig experience. *Proc. VLDB Endow. (PVLDB)*, pages 1414–1425, 2009.

[7] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proc. ACM SOSP*, pages 29–43, 2013.

[8] P. K. Gunda et al. Nectar: Automatic management of data and computation in datacenters. In *Proc. OSDI*, pages 75–88, 2010.

[9] B. He et al. Comet: batched stream processing for data intensive distributed computing. In *Proc. ACM SOCC*, pages 63–74, 2010.

[10] C. Lei et al. Redoop infrastructure for recurring big data queries. *Proc. VLDB Endow. (PVLDB)*, 7(13):1589–1592, 2014. (demo).

[11] H. Li et al. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proc. ACM SOCC*, pages 1–15, 2014.

[12] C. Olston et al. Nova: continuous pig/hadoop workflows. In *Proc. ACM SIGMOD*, pages 1081–1090, 2011.

[13] L. Popa et al. DryadInc: Reusing work in large-scale computations. In *Proc. HotCloud*, 2009.

[14] X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. *IEEE TKDE*, 3(3):337–341, 1991.

[15] A. Thusoo et al. Data warehousing and analytics infrastructure at Facebook. In *Proc. ACM SIGMOD*, pages 1013–1020, 2010.

[16] C. Yan et al. IncMR: Incremental data processing based on MapReduce. In *Proc. IEEE CLOUD*, pages 534–541, 2012.

[17] M. Zaharia et al. Spark: Cluster computing with working sets. In *Proc. HotCloud*, volume 10, page 10, 2010.