System Restore in a Multi-Cloud Data Pipeline Platform

Long Wang†, Harigovind V Ramasamy‡, Valentina Salapura*, Robin Arnold†, Xu Wang†, Senthil Bakthavachalam†,
Phil Coulthard†^, Lee Surprenant†, John Timm†, Denis Ricard†, Richard Harper†, Ahut Gupta†
† IBM Watson, ‡ IBM Cloud, * IBM T.J. Watson Research Center
{wanglo, hvramasa, salapura, robin.arnold23, xuwang, senthilb, lmsurpre, johntimm, dricard, reharper, Ahut.Gupta}@us.ibm.com, ^coulthar@ca.ibm.com

***Abstract*—Data pipeline platforms hosting big data analytics can span multiple clouds. Backup and restore service is typically applied to deal with data corruptions in such platforms. This paper proposes a novel approach to providing consistency to the restored state of a multi-cloud data pipeline platform from its backups, and also presents the performance of the approach demonstrated in a dry run test.**

Keywords—data pipeline, system restore, backup, big data, analytics, multi-cloud, distributed system, hybrid cloud

I. Introduction

Services and applications are increasingly hosted by platforms which span multiple clouds and/or hybrid clouds, e.g., a combination of a private cloud and several public clouds. That is because an enterprise-owned private cloud is attractive for hosting confidential data while different public clouds provide the enterprise with flexible options of services with different Service Level Agreements (SLAs) regarding resiliency, security, performance, cost, etc.

Data pipeline platforms (e.g., AWS Data Pipeline [6] and IBM Watson Health Cloud [7]) play an important role for facilitating big data analytics. The data analytics service provided by the data pipeline platforms may run on top of multiple clouds and/or hybrid clouds spanning multiple geographical sites. It is essential to ensure the data in the components of the data pipeline platforms are consistent, especially when the datasets are highly critical or subject to regulatory compliance.

In this paper, we address the problem of providing consistency to a multi-cloud data pipeline platform when recovering from one of its backup states. Backup and recovery services are used in data pipeline platforms when recovering from errors such as data corruptions or partial data losses. The problem discussed in this paper resembles the classical problem of taking a consistent checkpoint of a distributed system (as a backup can be considered as a type of checkpoint). A classical solution to the problem requires coordinating all components during the checkpoint so that the global checkpoint of the components form a consistent state [4]. However, in reality, a data pipeline platform may be comprised of commercial softwares from different vendors. It is difficult and inefficient to coordinate amongst such diverse software that may potentially span multiple clouds in different geographical regions.

A commonly used solution for backing up heterogeneous systems is to quiesce/freeze the entire workload, and then back up all components after they are frozen. The disadvantage of this solution is that a backup of those components with rapid data ingestion speed may take quite long (e.g., more than half an hour in our data pipeline platform). However, the requirement of low Recovery Point Objective (RPO, see Section II for explanation of the term) in commercial data pipeline platforms demands high backup frequency. For example, the RPO of 15 minutes demands a backup be obtained at least every 15 minutes. A long outage for backup is rendered unacceptable by such a requirement. To shorten the outage, one may apply high-frequency checkpoint technologies which utilize the copy-on-write feature of the underlying systems to quickly create snapshots with little overhead, e.g., VM-μCheckpoint [5], Hadoop snapshot [8], and Logical Unit Number (LUN)-based snapshot provided by Tivoli Storage Manager [9]. However, applying these technologies for achieving a global consistent backup requires all components be handled by the same snapshot controller, e.g., by the same Reliability MicroKernel controller [5], in the same Hadoop cluster, or by the same LUN controller. Such a requirement cannot be met on multi-cloud data pipeline platforms using heterogeneous data stores.

As none of the prior technologies achieve our goal and a general solution for a consistent backup/restore of a multi-cloud distributed system is unavailable, we designed a novel system restore approach by leveraging certain architectural characteristics of data pipeline platforms, specifically the characteristic that data are dispatched to the components for processing from a data pipeline queue. Each data item is assigned a unique identifier by the pipeline queue and the identifier is propagated to all downstream components. The system restore approach combines the restoration of individual components' backups (which may be inconsistent) and a replay of data from the data pipeline into the components to iron out inconsistencies. Particularly, each component is able to tell from the data item identifier whether or not the data item has been processed and saved in its data store (by looking up that identifier in the data store). Only unprocessed data items are processed by the component during replay.

We verified the consistency of the restored state in a dry run test by validating the data against pre-specified consistency rules. Moreover, the replay-based approach supports a near-zero RPO. With certain performance optimizations described in Section IV, the system restore time in the dry run test is less than 12 hours for restoring tens of terabytes of data, well satisfying the Recovery Time Objective (RTO, i.e. the downtime of the system before it is recovered) of the test platform.

The main contributions of the paper are: (i) the proposal of a novel system restore approach that addresses the challenging problem of providing consistency to the restored state of a multi-cloud data pipeline platform by leveraging specific architectural characteristics of the data pipeline, (ii) the presentation of performance in a dry run test, which demonstrates the effectiveness and efficiency of the proposed approach.

II. Data Pipeline Consistency Problem and Challenges

Figure 1 illustrates the architecture of our data pipeline platform. *Data Input* represents the inputs fed to the platform from many various data sources. For example, for a medical data pipeline platform, the data input may include patient and practitioner information from hospitals and clinics, medication information from pharmacies, standard descriptions of medication's therapeutic effects from the Food and Drug Administration, etc. *Data PreProcessors* are a number of processes distributed across multiple servers which perform specific types of pre-processing of the input data and correlation of the data from different sources.

Many *producers* put the pre-processed data into a data pipeline queue, and a number of *consumers* read data from the queue, send them to the downstream *data processors* and *data stores*. There are multiple data stores, one for each type of the input data. So *DataStore1, DataStore2, DataStore3* are data

978-1-7281-3032-3/19/$31.00 ©2019 IEEE
DOI 10.1109/DSN-Industry.2019.00012

stores of different types of data from potentially different vendors, cloud providers, or open-source technologies. A data store for one type of data has several replicas which may be in a primary-standby topology or an active-active topology in order to ensure data persistency and/or high availability of the system.

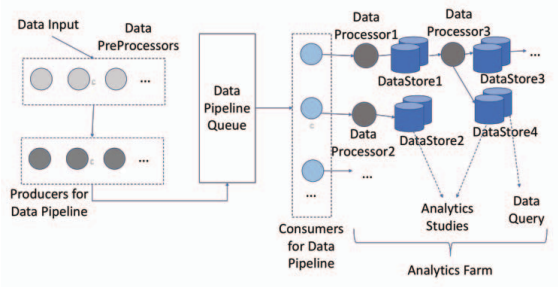

Figure 1: Architecture of the Data Pipeline Platform

The data pipeline consumers are of different types. Each type of consumer reads certain type of data from the data pipeline queue and sends them to a *data processing chain*. As illustrated in Figure 1, a data processing chain consists of a sequence of *<Data Processor i, DataStore j>* pairs, where a data processor represents operations such as data cleansing, data mapping, data integration, or data transformation. Output of the processed data from a *Data Processor* is stored in a *DataStore*. Data analysis and queries can then be performed on the data stores. The figure shows three data processing chains:

{<Data Processor1, DataStore1>, <Data Processor3, DataStore3>}, ...
{<Data Processor1, DataStore1>, <Data Processor3, DataStore4>}
{<Data Processor2, DataStore2>}

We use the term *System Restore* to denote the activity of restoring one or more constituent data stores from backups in a manner that ensures consistency of the entire data pipeline platform. As illustration, consider a data pipeline for processing patient medical data; regulatory compliance (e.g., HIPAA [11], GxP [12]) requires data backup to be implemented for the data stores in the pipeline. If data in a single data store gets corrupted, one may attempt restoration to a prior known good state, i.e., from the last backup with no data corruption. However, restoring only a single data store in this system from the backup to some prior state may cause the system as a whole to become inconsistent. For example, consider a data pipeline for a medical application which includes data stores for records of patient visits to doctors, data stores for records of patients, and data stores for records of doctors. For the consistency of the entire data pipeline platform, if there exists a record of a patient's visit to a doctor V_{PD}, there should also exist a record of the patient P, and a record of the doctor D. If the patient record datastore is restored from a prior backup that excludes P, and the visit datastore is restored from a prior backup that includes V_{PD}, then the data pipeline platform is an inconsistent state.

The topic of data consistency of distributed systems was studied for transactional systems [1, 2]. ACID properties guarantee atomicity, consistency, isolation, and durability of database transactions despite errors and failures. Those four properties characterize the transaction paradigm, which has influenced many aspects of development in database systems. Data consistency of distributed systems has also been deeply investigated in the context of distributed shared memory systems with various consistency models (e.g., eventual consistency, serial consistency) [3].

While previous works in those topics are relevant to System Restore, they do not address the specific problem we had to solve in the context of building a data pipeline platform due to multiple constraints. The first constraint is the usage of heterogenous components from different vendors on different clouds. The second constraint is the need for low RPO and low RTO. RPO specifies how much data might get lost after restoration from a failure, in terms of the interval of time during which the data generated might get lost. RTO specifies the duration of time when a service may be down after a failure, before the service is recovered to be operational. The third constraint is the need for data stores to be available even when a backup activity is ongoing because the backup may take a while. The fourth constraint is special considerations for certain data stores, e.g., some data stores can be fully regenerated from data in other data stores. The final constraint is consideration of data model dependencies among various data stores along the data pipeline.

III. System Restore Approach

The system restore approach combines component-specific backup solutions (to achieve high performance and low overhead) with a restore orchestration process that restores the components' states from backups while ensuring consistency to the restored state of the entire platform. The triggers for a system restore are manifold, e.g., partial data losses and data corruption. No matter the reason, a system restore in a commercial cloud platform is typically a business decision that is not taken lightly and after ruling out alternative localized recovery approaches.

A. System Backup

The system is comprised of data stores placed in different clouds. There are two kinds of data stores: *cache datastores* whose contents can be regenerated from other data stores by applying specific data processing chains and *source datastores* whose contents cannot be regenerated. Cache datastores are used in the data pipeline platform for performance enhancement. It is difficult to coordinate the backups of these data stores to achieve a globally consistent checkpoint like in traditional checkpoint solutions for distributed systems. We do backup of all data stores individually at approximately the same time, say 12am and 12pm of Coordinated Universal Time (UTC) every day. We do not have to provide strict time synchronization among the data stores on different clouds, sites, and/or time zones. Table 1 lists the categories of the data stores and their corresponding backup mechanisms. The backup mechanisms in the table are all application-level solutions because storage-level or kernel-level solutions usually do not have as good performance as application-level ones [10].

Table 1: Backup Mechanisms for Data Stores

Category of Data Stores	**Backup Mechanism**
Files	Incremental backup based on the *rsync* tool
Relational databases (e.g., Oracle, IBM DB2, MySQL)	The specific backup utility provided by the database software, e.g. Oracle Data Pump Export, the "`backup database <db name>`" command for IBM DB2.
NoSQL databases (e.g., MongoDB)	The specific backup utility provided by the database software, e.g. "mongodump" for MongoDB.
Hadoop files and databases	Combination of multiple Hadoop backup mechanisms, e.g., HDFS snapshot capability, HBase ExportSnapshot tool
Metadata and configurations of certain software	Custom tools and utilities are developed to capture state of these data stores.
Miscellaneous documents and data for onboarded user / customer accounts	Custom tools and utilities are developed to capture state of these data stores.

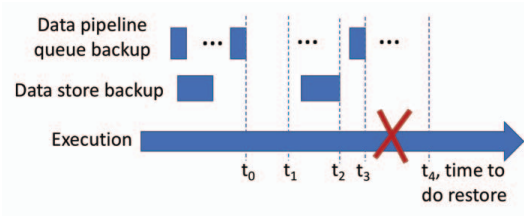

Figure 2: Backup Timeline

Besides backup of the data stores, we also perform backup of the data pipeline queue using a custom data pipeline consumer that saves the state of the data pipeline queue onto backup storage. The data pipeline queue implements a data retention policy (e.g., retain all queue data from the last two weeks); old data are purged from the queue. The special data pipeline consumer implements a continuous streaming-style backup process; however, it splits the backup into small chunks and each chunk has the backup timestamp, for facilitating the restore process.

Figure 2 shows sample timelines for the backups of the data pipeline queue and the data stores. The data pipeline queue's state is a sliding window along the execution timeline as expired data keeps getting purged based on the retention policy. Though the figure shows only one data store, the timelines for the backups of the other datastores are similar, i.e. their backups have same frequency and are started at approximately the same time. So, each box in the datastore backup timeline in Figure 2 represents one batch of backups spanning all the datastores. We mark the timestamps of the backups by the completion time. Thus, t_0 and t_3 are the completion timestamps for two backups of the data pipeline queue. t_2 denotes the completion time for the backup that completed last in one batch of datastore backups. t_1 is the time before any backup in that batch was started.

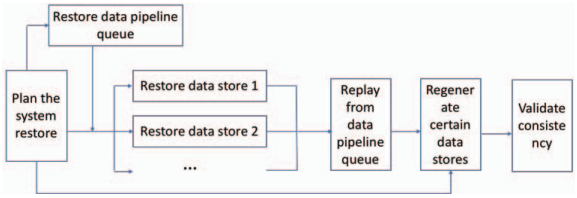

Figure 3: The System Restore Process

B. System Restore

The restore orchestration process is an automated restore engine that helps ensure consistency across the data stores in the data pipeline platform during a restore activity (Figure 3). The process implements the logic behind the recovery plans shown in Table 2. The process is executed in a maintenance window with the workload stopped.

Step 1: Plan the system restore.

Not all failures require restoration from backups for recovery. For example, a failure of a server may be recovered by either restarting the server or failing over the workload to another replica of the server; the site-level or cloud-level failure due to a disaster can be recovered by failing over the data pipeline platform to another site or cloud. Usually failures that require restoration from backups are associated with data corruption. There are different failure scenarios that involve data corruption, each meriting its own recovery plan, as shown in Table 2. In Figure 2, time t_4 denotes the time when a restore is attempted due to the occurrence of one of the failure scenarios sometime between t_3 and t_4.

Table 2: Recovery Plans for Different Failure Scenarios

Failure Scenario	Recovery Plan
1. Failure of component(s) resulting in corruption or destruction of *cache datastore(s)*	a) Regenerate the failed cache data stores from the source data stores. b) Validate consistency.
2. Failure of component(s) resulting in corruption or destruction of *source datastore(s)*	a) Restore *all* source data stores from their backups at t_2, when the backups of the data stores were taken immediately before the failure. b) Do the replay from the existing data pipeline queue, starting at t_1 and ending at the end of the queue (i.e. t_4). c) Regenerate all cache data stores from the restored source data stores. d) Validate consistency.
3. Failure of component(s) resulting in corruption or destruction of the data pipeline queue	a) Restore the data pipeline queue from its backups with timestamps ranging from t_0 to t_3. b) Restore *all* source data stores from their backups taken at t_2. c) Do the replay from the restored data pipeline queue, starting at t_1 and ending at t_3. d) Regenerate *all* cache data stores from the restored source data stores. e) Validate consistency.
4. Two or more of the above failure scenarios	Select one recovery plan above that handles all the failure scenarios. For example, the recovery plan for Scenario 2 can also handle Scenario 1; the recovery plan for Scenario 3 can handle both Scenario 1 and Scenario 2.

Step 2: Restore from backup.

When restoring from backup is needed, Step 1 helps decide which data stores should be restored, whether the data pipeline queue should be restored, and which backups of them should be used. Then the restore mechanisms for the data stores and/or the data pipeline queue are applied. The restore mechanisms are the corresponding solutions for the backup mechanisms listed in Table 1. It is worth noting that when we restore the data pipeline queue from a sequence of backups (or chunks), we should merge them together into a complete data set and then restore the data pipeline queue with this data set.

Step 3: Replay data from data pipeline queue.

The key for ensuring consistency in the recovered platform state rests in a controlled replay of the data items in the data pipeline queue. Replaying a data item from the data pipeline queue involves activating *all* data processing chains on that data item. Some data items may not be relevant for a data processing chain; activating such data processing chains on that data item will result in no changes to the data stores within those chains. Within an *activated and relevant* data processing chain, if the backup used for restoring a data store already reflects a state where a data item was already processed and placed in the data store, then that data item is skipped during the replay; only otherwise, should that data item be processed and the result stored in the data store. For this purpose, each data item in the data pipeline queue has a unique identifier (created by the pipeline queue and placed in the data item as an extra payload attribute) that is carried over in the data stores of all the relevant data processing chains.

Additional considerations during the replay procedure are listed below:

(1) The data pipeline queue should be cleaned before the replay. The streaming of the data pipeline is stopped by the system restore, and as a result, the data pipeline queue has partial data, i.e. data that are part of stopped transactions. So, prior to replay, we scan the data pipeline queue, identify such partial data, and remove them from the queue. A complete set of

references of a transaction's data items are placed in each of the data items to facilitate the cleaning process.

(2) Each data item has a timestamp indicating when the data item is fed into the platform. During normal processing, the timestamp is just the system clock time. However, in order to meet certain compliance requirements (e.g., for data pipeline platforms subject to GxP standards applicable to pharmaceutical companies), the old timestamp reflected in the backup must be retained during replay of the data item, and the current system clock time cannot be used.

(3) Certain metadata and helper attributes are generated dynamically during processing, e.g. record IDs in a database. Such data are not obtainable in a deterministic manner from the data pipeline queue. So, these new metadata/attributes need to be handled carefully to avoid injecting inconsistency with the data restored, and corrections/updates are made when needed.

Step 4: Regenerate cache datastores.

After restoring the source data stores, the values in the cache data stores may not be consistent with the restored source data stores. Therefore, it is necessary to regenerate the cache data stores from the source data stores.

Step 5: Validate consistency.

As the last step, the restore orchestration process performs consistency validation by checking the contents of the data stores against a number of pre-specified rules. These rules are formulated offline using domain knowledge. A sample (and trivial) rule may specify that: if the system holds a record for a patient visit to a doctor, the records for the doctor (practitioner) and the patient must also exist in the system in the respective data stores. In reality, these rules are much more complex.

IV. Performance in Dry run Test

The main performance metrics of the system restore process are RPO and RTO. The frequency of the data store backup in our dry run test environment is every 12 hours, but the replay step will populate the data of the data stores until (i) the current time if the data pipeline queue is not failed, or (ii) the time t_3 in Figure 2 if the data pipeline queue suffered from a failure (Scenario 3 in Table 2). In case (i), the RPO is nearly zero. In case (ii), as the data pipeline queue is backed up at high frequency (every few minutes), the RPO provided by our solution is a few minutes. We may reduce the RPO further for case (ii) if we increase the frequency of the data pipeline queue backup, but a few minutes is sufficient for meeting the RPO value in the SLA of our test platform (15 minutes).

An active-active or multi-active topology is widely used for providing low RTO for high availability or disaster recovery, however, the types of failures that require backup and restore, like data corruption, cannot be handled by high availability or disaster recovery solutions, because data corruption propagates to all replicas almost immediately. Instead, we minimize the RTO value by adopting the following optimizations:

- Automating the system restore process in Figure 3.
- Restoring all source data stores, and if needed, the data pipeline queue, in parallel. So, the total restoration time is the time spent on the data store which takes most time for restoration. In our dry run test environment, it took less than 10 hours to restore the largest data store which has 20TB data.
- Minimizing the amount of replay work by selecting t_1 close to the start of the backup at t_2 (see Figure 2). But t_1 should be sufficiently ahead before the start of the backup (say, 10 min) to cover the time differences on different machines and different clouds, so that we can avoid the complications of time synchronization. As data stores are backed up every 12 hours, the largest amount of data to be replayed are 12 hours of data in the pipeline. In our dry run environment, it took less than 2 hours to replay 12 hours of data (typical amount of ingested data per day is much less than which saturates the pipeline's capability).
- Performing the regeneration of the cache data stores and the validation after the service is back online, so the two steps do not count towards the RTO. This optimization is feasible because (a) the cache data stores act as cache for the real source data stores, and are only created and used for enhancing performance, and (b) the consistency validation is a read-only operation that can run in parallel with the real workload. Before the regeneration, the data processors query and update only the source data stores; after regeneration, the data processors can consult cache data stores.

The total system restore time in our dry run test is less than 12 hours, which well satisfies the RTO of the test platform. Furthermore, the effectiveness of the restore is demonstrated by the consistency validation step. We constructed 18 consistency rules that cover the most critical relationships among the data stores in our platform. The exact rules are not presented here as they bear confidential information, but, as an example, a simple rule has the following format (there are more complicated rules): *if data i is in store x then data j has to be in store y*. The validation of the restored datastores against the rules did not report any violation of the rules in our dry run test.

V. Conclusions

Data pipeline platforms such as AWS Data Pipeline and IBM Watson Health Cloud are playing an important role for hosting big data analytics. Due to manifold reasons, restoration from backup may be needed to recover the platforms, and it is critical for their restored state to be consistent across the large number of components of the platforms. Because of the heterogeneity of the components (from different vendors, on different clouds) and the low RPO requirement, prior-art technologies for taking a global consistent backup or conducting consistent restoration of a general distributed system do not apply. We proposed a system restore approach, by leveraging the architectural characteristics of data pipeline platforms, to provide consistency to the state of multi-cloud data pipeline platforms restored from backups. Our dry run test validated the effectiveness of the approach and demonstrated it satisfies the required RPO and RTO values.

References

[1] I. Traiger, J. Gray, C. Galtieri, B. Lindsay: Transactions and consistency in distributed database systems, ACM Trans. on Database Systems, 1982.

[2] D. Parker, G. Popek, G. Rudisin, A. Stoughton, B. Walker, E. Walton, J. Chow, D. Edwards, S. Kiser, C. Kline: Detection of Mutual Inconsistency in Distributed Systems, IEEE Transactions on Software Engineering, Volume: SE-9, Issue: 3 , May 1983.

[3] M. Mizuno, M. Raynal, J. Zhou: Sequential Consistency in Distributed Systems, Dagstuhl Seminar on Distributed Systems 1994.

[4] L. Wang, K. Pattabiraman, Z. Kalbarczyk, R.K. Iyer, L. Votta, C. Vick, A. Wood: Modeling coordinated checkpointing for large-scale supercomputers, Dependable Systems and Networks 2005.

[5] L. Wang, Z. Kalbarczyk, R.K. Iyer, A. Iyengar: VMμCheckpoint: Design, Modeling, and Assessment of Lightweight InMemory VM Checkpointing. IEEE Trans. on Dependable and Secure Computing, 2015

[6] https://aws.amazon.com/datapipeline/

[7] https://www.ibm.com/cloud/healthcare

[8] https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsSnapshots.html

[9] Tivoli Storage Manager Overview, https://www.ibm.com/support/knowledgecenter/ru/SSGSG7_7.1.1/com.ibm.itsm.srv.doc/c_tsmintro.html

[10] L. Wang, R. E. Harper, R. Mahindru, H. V. Ramasamy: Disaster Recovery for Cloud-Hosted Enterprise Applications, Cloud Computing 2016.

[11] https://www.hhs.gov/hipaa/index.html

[12] https://en.wikipedia.org/wiki/GxP