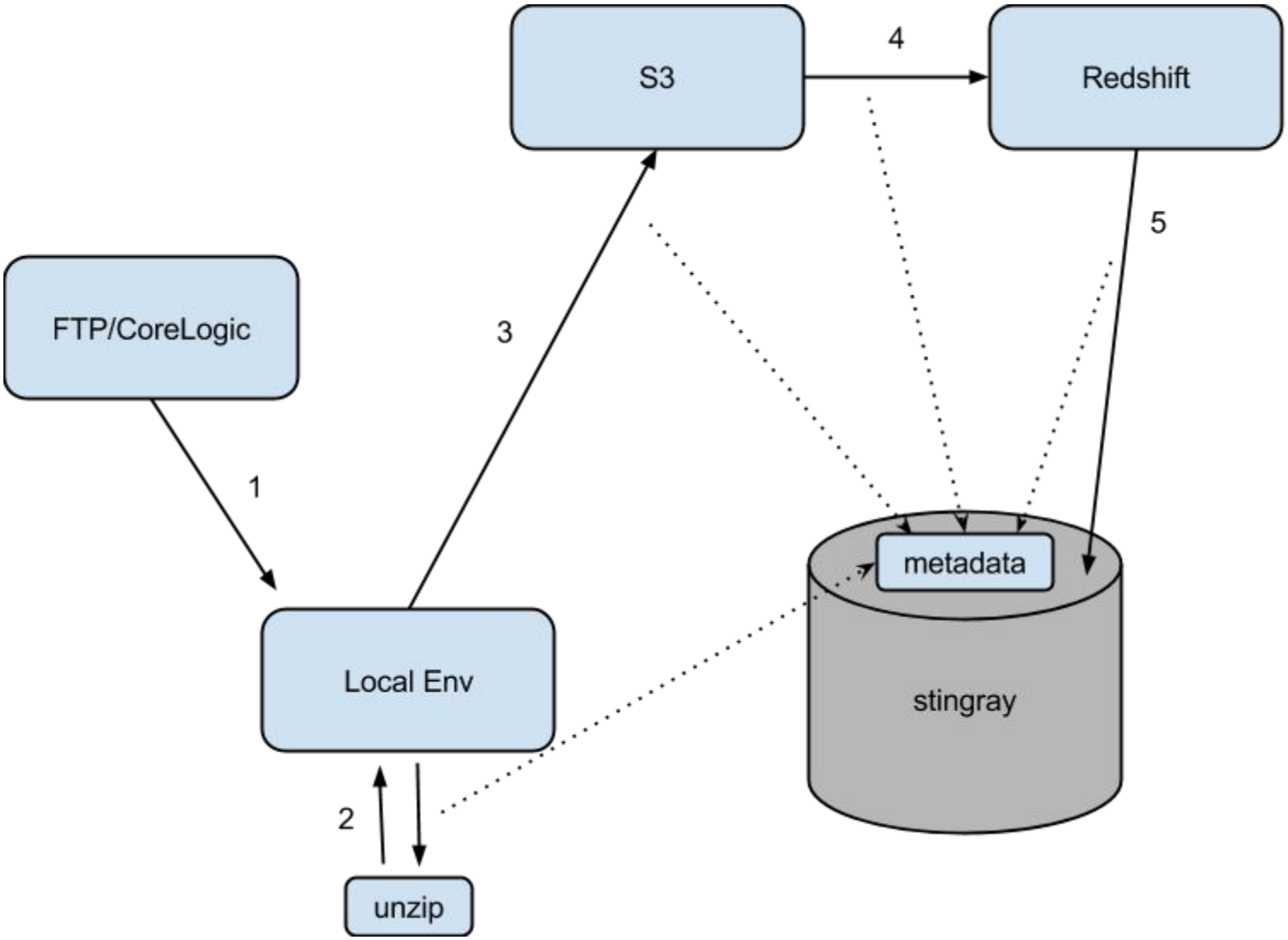
COLD Importer Overall Workflow Design



Overview

This doc gives overall detail of each step in the process of the COLD Importer. Other docs will give more specific documentation such as the schema design of each step. This however can be referenced to understand the big picture. For a visual of the process refer to the image above.

Step 1 - Download from FTP/CoreLogic Servers

The title is self-explanatory, in step 1 we simply will download the property files/data from the FTP server, which is owned by CoreLogic. In this step we simply need to specify a generic location to download the compressed files, so that it not only works on a local machine, but any server.

Success of this step:

- compressed files all downloaded and stored in proper location
    - proper location: see Jian's comment

Validation of Success:

- Check: a decompressed file was downloaded and the download completed
    - If only a partial download of the compressed file (highly unlikely) we will simply start from the beginning with a fresh download
- Check the path to the file and see if it matches the structure set up
- Perform each the initial null/empty check below before the unzipping process begins
- Update metadata object for completion of step 1

Possible failures of this step:

- Connection failure to FTP servers:
    - before the download begins
    - anytime during the download
    - at the completion of the download
- Download file data is in the wrong format
- Downloads are stored in the wrong locations
- Compressed files are empty/null

Handling Errors:

- Connection failure to FTP servers - FTPServerNetworkDownloadException
    - Stop the download process on the current file
    - Manually check the connection and start again with the same download
    - Write to *cold_importer_property_file_processsing_status* table even on failure to keep track of time stamp of the initial download
    - Simply redownload the same file and start from the beginning
- Downloads are in the wrong format - FTPServerFormatDownloadException
    - If file is not .zip or .tar then try re downloading the file and again checking the status. More than likely this will be an error from FTP Server and not us
    - Still write to *cold_importer_property_file_processsing_status* and create row on initial download
- Downloads are stored in the wrong locations - FTPServerLocationDownloadException
    - Script to check the pwd of the download

        - Write mv command to put the downloaded zip in desired location
        - More than likely this will be an error on user credentials ssh into trunk
- Compressed files are empty/null
    - check the size of the downloaded file
    - If it meets initial threshold size and exists in the system then it passes the test
    - Won't actually know if the file content is null until we unzip

Step 2 - Decompress downloaded files

The title is self-explanatory for this step as well, but in step 2 we will decompress/unzip the downloaded file in it's current directory. In this step we simply need to navigate to the downloaded directory, and properly decompress the files to its intended format.

Success of this step:
- File is properly decompressed and stored in the same directory as downloaded
- Data file is now in intended data format and folder contains the metadata file and data file
- Metadata object is updated for this file to reflect these changes

Validation of Success:
- Check the folder of the decompressed download
    - Does the data file match the metadata file in size, number of entries, and format?
- Once this passes, and no errors occurred decompressing and opening the file, then update the metadata object for completion of step 2

Possible failures of this step:
- Files cannot be decompressed because they are empty or null
- Only part of the files get decompressed ?
- Decompressed files are stored in the wrong directory
- Files become lost once decompressed ?

Handling Errors:
- Files cannot be decompressed because they are empty or null - PropertyFileFormatDecompressException
    - try the process a second time and see if it was a malfunction
    - If still an error, operator inspect the file and the download source from FTP
    - More than likely an exception won't be thrown because this is prevented in step 1, but our script should identify if the file is empty to avoid uploading null data to S3

- Only part of the files get decompressed - PropertyFileFormatDecompressException

        - should be a rare error, only part of the file is unzipped?
        - More than likely this will be the result of the OS trying to unzip a file and it throws a system error - likely error in data format or file size
- Decompressed files are stored in the wrong directory - PropertyFileLocationDecompressException
    - like in step 1, after decompressing the file check the current directory to make sure it still matches -
- Files become lost once decompressed - PropertyFileLocationDecompressException
    - operator manually searches for the files
    - If not found, delete metadata entry for this file_id
    - restart the process for the file from step 1

Step 3 - Properties uploaded to S3

In this step all of the property data will be uploaded to S3 buckets. The importer for S3 will have to be configured to connect to the proper buckets that we own. The data uploaded to S3 will be raw and in the format from FTP. The reasoning behind this is that we need a copy of the raw data because FTP doesn't keep a copy of it. Data formatting will be down before storing into *stingray.*

Success of this step:

- All of the property data was successfully uploaded to the proper S3 bucket
- Bucket for each type of data that needs to be uploaded ~ 5 buckets

Validation of Success:

- Received successful code from S3 that our file was uploaded to the right bucket
- Once code is received, the metadata object is reflected that this step is complete

Possible failures of this step:

- Connection lost to S3 during uploading
    - No data was uploaded
    - Part of the data was uploaded
- Connection lost to S3 after uploads
- Data was uploaded to the wrong bucket
- Wrong data was uploaded
- Corrupted data was uploaded (null or empty)
- Data was leaked during the uploading process (security threat)

Handling Errors:

- Connection lost to S3 during uploading: AWSNetworkS3UploadException
    - receive error code from S3 that there was a failure
    - check the code and network connection to configure
    - this step is atomic, so if it doesn't all safely load to the proper bucket, then no permanent harm done
    - retry the step
- Connection lost to S3 after uploads - AWSNetworkS3UploadException
    - if after upload, the upload should have persisted, and should have received a status code confirming it from S3
    - operator perform check to network and see if it can get back online
- Data was uploaded to the wrong bucket - AWSLocationS3UploadException
    - check the confirmation code from S3 to detect the bucket upload
    - operator manually checks the S3 bucket that the code responded with to see if the file is there - making sure it wasn't an error from S3
    - check the code to see the bucket the file was suppose to be uploaded to
    - delete the file from the improper bucket and restart the process of uploading to the right bucket
- Wrong data was uploaded - AWSFileS3UploadException
    - check the confirmation code from S3 to the the contents of the file that was just uploaded
    - operator manually checks the S3 bucket to see if the wrong file is actually in the bucket
    - check the code/metadata objects to see if the right file_id was timestamped last
    - delete the wrong file from the bucket, and start the upload again for the right file
- Corrupted data was uploaded - AWSFileS3UploadException
    - check the confirmation code from S3 to see if the data was empty or null
    - check the bucket in S3
    - check the file contents to see if the data is empty or null or if just the upload is pulling empty or null
    - This error is probably the result of another error happening with it either being the wrong file or null file
- Data was leaked during the uploading process (security threat) - AWSSecurityFileS3UploadException
    - check the confirmation code from S3 to see if they detected this
    - probably going to be an outside resource seeing the error and notifying us
    - review the upload code to S3 and also the FTP data
    - most likely will need to be thoroughly reviewed by operator

Step 4 - Copy Properties from S3 to Redshift

This step will successfully try to transfer all of the newly added properties that don't exist in Redshift from S3. It will make sure that the files don't exist already in Redshift by reading from the metadata object *last_copy_to_redshift.*

Success of this step:

- *cold_importer_s3_processing_status* <= *cold_importer_redshift_processing_status*
- All of the data from *cold_importer_s3_processing_status* was successfully loaded into Redshift. After the load *last_uploaded_to_s3* == *cold_importer_redshift_processing_status*
- Data is stored in the proper tables in Redshift

Validation of Success:

- Check the timestamps of the metadata tables for *cold_importer_s3_processing_status* and *cold_importer_redshift_processing_status*
- Check the confirmation code from Redshift to see if the atomic copy command went through and the data rests in the right cluster
- Check the corresponding clusters in Redshift to see if the file now sits there
- Update the metadata object reflect the update in the pipeline

Possible failures of this step:

- *cold_importer_s3_processing_status* > *cold_importer_redshift_processing_status* - meaning there are later timestamps of uploads to S3 then there are copies to Redshift. This means a copy to Redshift was skipped
- Connection failure between S3 and Redshift
- Data is stored in the wrong cluster in Redshift
- Corrupted data is copied into Redshift
- Data was leaked during the copy process (security threat)
- Wrong data from S3 was copied into Redshift

Handling Errors:

- *cold_importer_s3_processing_status* >*cold_importer_redshift_processing_status* - AWSTimeRedshiftException
    - check the timestamp to see if the result is actually true
    - investigate if both of the latest timestamps point to the same file_id object
    - if they do check the console output to see if there was an error the first time when trying to copy the file to Redshift
        - If so, ignore this error, it should have a later timestamp
        - Basically only look at the earliest timestamp in the *cold_importer_redshift_processing_status* for a particular file_id, because this tells you the FIRST time the copy command was initiated

        - the timestamps after that only occur if there was an error and should be there afterwards because the table is append only
        - But when trying to copy for the first time for a particular file_id, need to check the latest timestamp to the table, as this should not violate the *cold_importer_s3_processing_status* <= *cold_importer_redshift_processing_status* for a particular file_id
- Connection failure between S3 and Redshift - AWSNetworkRedshiftException
    - check the confirmation code from Redshift about the failure
    - operator checks the network to see if it is an easy fix
    - make sure the code is targeting the right network
- Data is stored in the wrong cluster in Redshift - AWSLocationRedshiftException
    - check the confirmation code from Redshift about the location the data was actually stored
    - check the code to see the target cluster in redshift the copy command initaited
    - delete the file from the wrong cluster and initiate the copy command on the proper one
- Corrupted data is copied into Redshift - AWSFileCopyRedshiftException
    - if file is null or empty
    - check the confirmation code from Redshift to see if the transaction went through
    - check the redshift cluster to see if the faulty data exists
    - checks the S3 bucket from origin of copy to see if data is faulty there as well
    - might need to start process from beginning of pipeline for file if it has been faulty throughout it's lifetime
- Data was leaked during the copy process (security threat) - AWSSecurityRedshiftException
    - check the confirmation code from Redshift to see if they detected the exception
    - inspect the data to see what kind of information was leaked
    - check the FTPServer location file to see if it was a problem from the source
    - shutdown the process to determine where in the pipeline the information is leaking
- Wrong data from S3 was copied into Redshift - AWSFileCopyRedshiftException
    - check the confirmation code from Redshift to see what data the copy command was performed on
    - check the code to see the target for the copy command
    - check the S3 bucket to make sure the file we want copied exists in the proper bucket
    - Delete file from the cluster and perform copy command again with the file intended

Step 5 - Import properties into stingray

In this step we will format the data correctly to be stored into the property tables in *stingray*. After reformatting they will be stored there. It makes sure that it's not restoring old data into stingray by reading from the metadata table *cold_importer_stingray_processing_status.*

Success of this step:

- *cold_importer_redshift_processing_status* <= *cold_importer_stingray_processing_status*
- The data has to be formatted correctly
- Property data is stored correctly into stingray, and in the proper tables

Validation of Success:

- check the metadata tables to see if *cold_importer_redshift_processing_status* <= *cold_importer_stingray_processing_status* for the first time importing into *stingray*
- get success code that all of the data was stored properly into stingray without any errors - should be atomic process
- update metadata that the pipeline has completed for this file_id

Possible failures of this step:

- *last_copy_to_redshift* > *last_inserted_into_stingray* - meaning that there are timestamps in which copies to Redshift were made without ever inserting into *stingray*.
- Property file is null or damaged
- Property file cannot be formatted into what the *stingray* table requires
- Data was leaked during the copy process (security threat) or corrupted data imported

Handling Errors:

- *cold_importer_redshift_processing_status* > *cold_importer_stingray_processing_status* - StingrayFileImportTimeException
    - check the timestamp to see if the result is actually true
    - investigate if both of the latest timestamps point to the same file_id object
    - if they do check the console output to see if there was an error the first time when trying to import into *stingray*
        - If so, ignore this error, it should have a later timestamp
        - Basically only look at the earliest timestamp in the *cold_importer_stingray_processing_status* for a particular file_id, because this tells you the FIRST time the copy command was initiated
        - the timestamps after that only occur if there was an error and should be there afterwards because the table is append only
        - But when trying to import for the first time for a particular file_id, need to check the latest timestamp to the table, as this should not violate the

cold_importer_s3_processing_status <=
cold_importer_redshift_processing_status for a particular file_id

- Property file is null or damaged - StingrayFileImportException
    - check the response code from stingray db to see the details - transaction is atomic so no harm should be permanently stored
    - check the corresponding metadata and file in redshift and s3 to see if there were earlier issues with this same problem
    - operator open file in Redshift and inspect
    - try to reimport the file once concluded that it seems to be non-null or damaged
- Property file cannot be formatted into what the *stingray* table requires - StingrayFileImportFormatException
    - check the response code from stingray to see what the mapping issue is - most likely a type conversion
    - check the *stingray* schema to see what we are trying to cast/format the file data into
    - make judgment based on if the schema needs to change or if middleware code needs to be written to massage/serialize the data in the proper format
    - make adjustments based on those scenarios and try to reimport the file
- Data was leaked during the copy process (security threat) or corrupted data imported - StingrayFileImportSecurityException
    - most likely either losing data in transition Redshift ---> stingray or the file sitting in Redshift cluster contains harmful information trying to attack our system
    - Check the response code from stingray to see if the transaction went through - hopefully it did not
    - Check the file contents throughout the pipeline to see where it acquired corrupted data
    - Shut down the pipeline momentarily to make sure the system is not at risk
    - Check the connection between Redshift cluster and *stingray* to see if is at risk for harmful infiltration
    - every scenario for this exception will be unique