Architecture

Music/Cassandra Tables

Transaction Information Table (TIT)

This table contains the entities that are lock to perform transaction on partition, it contains an ordered array that contain information of the transactions.

Arow is associated with an specific partition.

There can be more than one row associated to a partition. But only one is active at any given time. There are two main reasons associated with creating a new TIT row: * Partition merge/split * The REDO array is too big, and its better to start a new row

This is a type of table, and there can be many instances of this table in Music. Reasons to have more than one table: * One table per replication policy, for example there could be one TIT table with replication factor of 3 and one TIT table with replication factor of 5. * Other policy requirements

Columns

- **Index**: UUID
  - Id of this row
  - **Note**: Later we could force the index to following a certain pattern, such that the rows of a given MDBC Server are located as close as possible to that server, without violating any other policy associated with the data.
- **Redo**: Array<Tuple<Text,UUID>>
  - Array (order-matters) of <TableName,Index> associated with the Redo Records that were applied to this partition
- **Partition**: UUID
  - Id of the partition associated with this transactiomns
- **LatestApplied**: Int
  - Integer associated with the latest RedoRecord applied from the Redo column into the data tables in Music.
- **Applied**: Boolean
  - Flag that indicates that this row Tx's were already commited to the data tables

Primary

(Partition)

Clustering

(Index)

Redo Records Table (RRT)

This table is the one that contains the TransactionsDigests.

There is one row per transaction.

There is no need to lock on this table.

This is an append/remove only table, no updates are going to be performed. * Removes are an optimization, not a correctness requirement * Removes are only executed when the transaction was completely applied to the data tables and all tables that are pointing to this row are already removed.

Columns

- **leaseid**: text
  - Id of the lease that was used to process the transaction associated with the row in TIT
- **leasecounter**: bigint
  - Transaction number (counter of the transactions performed so far using the lock in leaseid)
- **transactiondigest**: text
  - Serialized transaction digest, can be considered a blob

Primary

(leaseid,leasecounter)

Clustering

None

TableToPartition Table (TTP)

This table maps each table to the current (and previous) partitions.

Columns

- **Table**: Text
  - Name of the table to which partitions are being associated
- **Partition**: UUID
  - Current partition that holds this table
- **PreviousPartitions**: Set<UUID>
  - Name of all the previous partitions (inclusive of the current partition)

Primary

(Table)

Clustering

None

PartitionInfo Table (PI)

This table contains information about a partition. Contains information about the latest Tit row to be lock if this partition want to be hold. The tables associated with this partition and other information of the partition.

Columns

- **Partition**: UUID
  - Name of the partition that this rows describes
- **LatestTitTable**: Text
  - Name of the table that contains the latest TransactionInformation row associated with this partition
- **LatestTitIndex**: UUID
  - Latest index (row) in the previous table, that is currently is being updated with transactions in this partition
- **Tables**: Set<Text>
  - All tables that are contained within this partition
- **ReplicationFactor**: Int
- **CurrentOwner**: Text
  - URL address associated with the current owner

Primary

(Partition)

Clustering

None

RedoHistory Table (RH)

This table represents the Directed Graph that forms the history of REDO logs. Given that we create new REDO logs on each new repartition (or due to other reasons), we need to keep track of the changes and the order of the REDO logs. An example of the repartitions can be seen in figure 3.

Columns

- **RedoTable**:Text
- **RedoIndex**:UUID
- **PreviousRedo**: Set<Tuple<Text,UUID>>
- **Partition**: Text

Primary

(Partition)

Clustering

(RedoTable,RedoIndex)

Server Architecture

As shown in figure 1. The MDBC server is composed of the following components:

- **RunningQueries**
- **LocalStagingTable**
- **MusicInterface**
- **MusicSQLManager**
- **SQL**

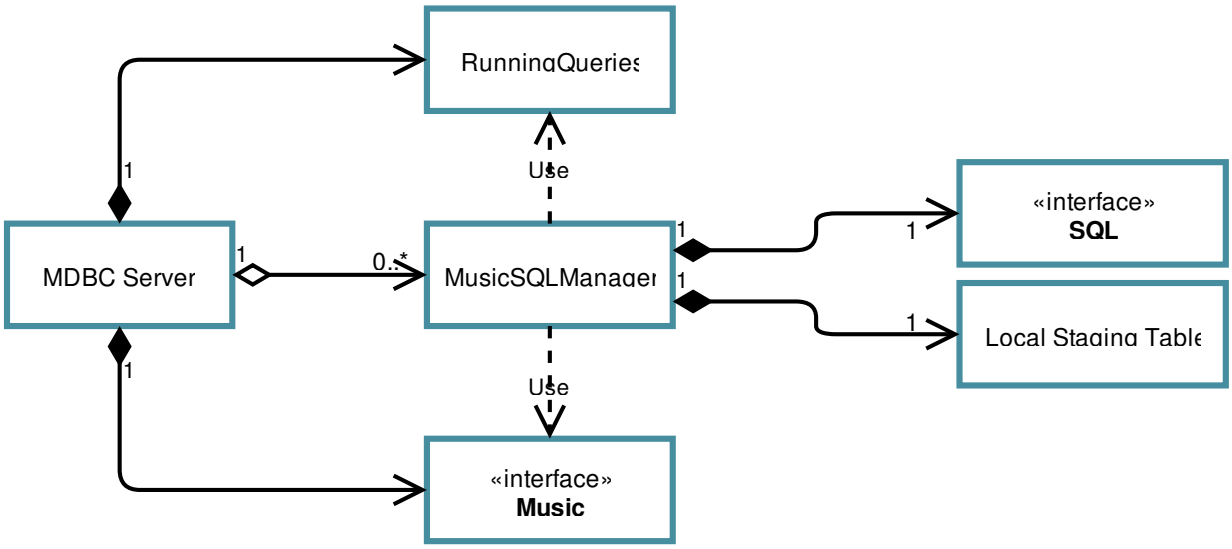

Fig 1: Server Architecture

RunningQueries

This is an in-memory data structure that contains the progress of each transaction being executed. Each transaction hold information about:

- **LTxId:** local transaction id, unsigned integer
- **CommitRequested** bool indicating if the user try to commmit the request already.
- **SQLDone:** bool indicating if sql was already commited, atomic bool
- **MusicDone:** bool indicating if music commit was alredy performed, atomic bool
- **Connection:** reference to a connection object. This is used to complete a commit if it failed in the original thread.
- **Timestamp:** last time this data structure was updated
- **RedoRecordId:** id of the redo record when it was already added to the RRT (it contains the info to the table and index)

LocalStagingTable

This is a serializable in-memory data-structure that contains the information about the changes that were updated by the SQL database for a given transactions. When the transaction is commited, this staging table is freeze and serialized to a string, and commited to Music. See the algorithms section.

There is one table per client connection.

It has one main operation:

```
1  void addOperation(String key, OperationType op, JSONObject oldValue, JSONObject
   newValue);
```

MusicInterface

This is the layer that interacts with Music. There is only one instance per MDBC server. It is in charge of holding locks, and executing get/puts to Music, following the corresponding locking mechanisms. This object is also used in the MusicSQLManager, so a reference of it is passed when a MusicSQLManager is created.

MusicSQLManager

When a connection is created from a new MDBC client, a Connection object is created. MusicSQLManager is the object that handle the main operations of a connection that translates operation between SQL and Music.

There is one object of this type for each client connection.

SQL or DBInterface

This interface the main operations that the MusicSQLMAnager performs with the SQL layer.

REDO Recovery Architecture

Figure 2 shows the main components used to recover using the REDO log. First we are going to analyze what is the REDO history associated with a given node, and then describe each of the components in figure 2

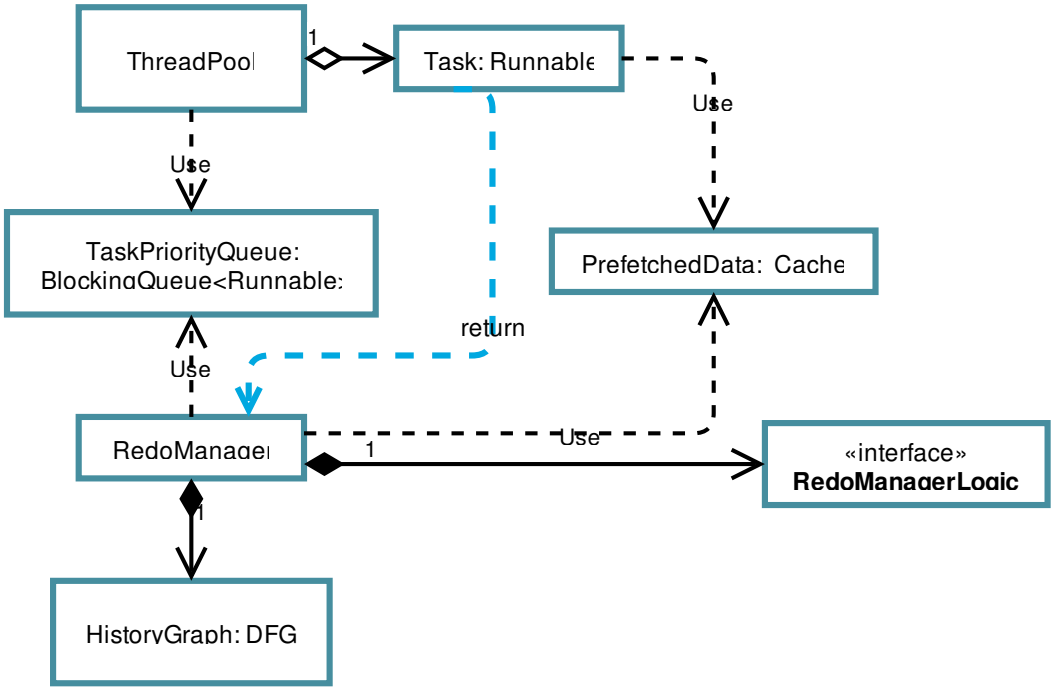

Fig 2: Redo Spec Architecture

REDO History

Given that new Redo log rows are created in TIT each time, the system is repartitioned, then a history is created as shown in figure 3. Each node represents a give TIT row, and the graph is a directed acyclic graph.

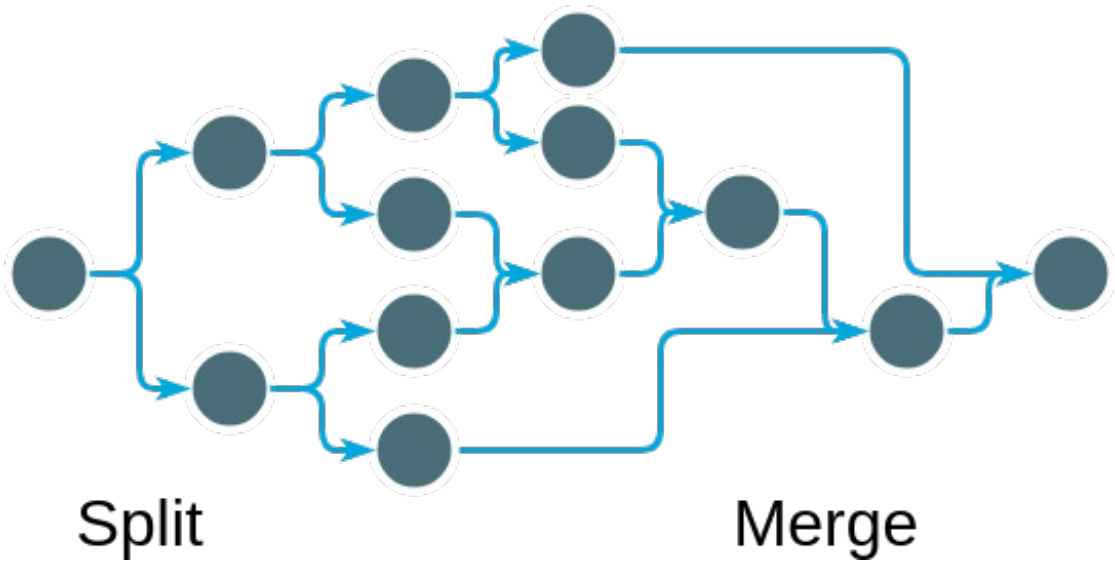

Fig 3: Redo History Directed Graph

Properties

- The are only two types of repartition: split in two or merge two.
- Each partition can only be splitted into two partitions
- Only two partitions can be merged into a new partition
- Sometimes one node is transformed into a new node, without split or merge. This can happen due to the reasons explained in previous sections, such as a TIT row being too big.
- Each partition is going to have a new different name (e.g. use UUID to name them)
- On partition a new TIT row is created

- The replication policy used is the max of all the tables that are contained within the partition.

Node

Each node in the graph contains the following information

- **RedoTable**: info from RH table
- **RedoIndex**: info from RH table
- **Tables**: tables that are going to be recovered in that node, obtained from partition in RH table.
- **TIT Metadata Read**: it is a boolean flag that indicates that the associated metadata was already downloaded.
- **ParentsDoneCounter**: it is a counter that is increased each time one of the parents of the node was successfully applied.
- **LastAppliedChange**: last change applied from this Redo Index to this local MDBC server.
- **TxDigestDownloaded**: boolean flag that indicates that the TxDigest download task was already downloaded

All this information is used to implement the algorithms to parallelize the REDO operations.

Redo Recovery Execution

We decomposed the recovery algorithm into multiple independent tasks that is going to be executed by a pool of threads. There are three main types of tasks:

1. Download Metadata (embarasingly parallel)
2. Download Transaction Digests (emabarasingly parallel)
3. Apply Digests (Sequential with concurrency)

To save memory usage, and avoid pulling too much data before it is actually consume, we cannot perform this tasks in this same order. Additionally prefectched data can be erased, if there is too much data downloaded that was not commited in time.

RedoManager

This is the entity in charge of defining the next tasks to execute and to initialize the REDO recovery and all the required data structures, including the thread pool.

RedoManagerLogic

This is the logic within the redo manager. It is created to allow extension of the algorithms for selecting the tasks.

Interface

```
void init(HistoryGraph history);
List<Task> getInitialTasks();
List<Task> setCompleted(Task completedTask);
Boolean isDone();
```

PrefetchedData

This is going to contain the results of tasks of type 1. and 2. If the recovery algorithm overprovisioned memory, the RedoManager can request to delete the least important data (further to the right in figure 3), and it would return the task that was used to generate it. Such that it can be readed to the TaskPriorityQueue

Interface

```
Metadata getMetadata(int NoodeIdx);
Digest getTransaction(int NodeIdx, String table, UUID row);
Task deleteLessImportant();
void addMetadata(int NodeIdx, Metadata meta, Task task);
void addDigest(int NodeIdx, String table, UUID row, TxDigest tx, Task task);
long getSize();
```

HistoryGraph

This is the object that is going to model the REDO history, and will be used to create the plan for recovery. Additionally it would hold the required data to select the next tasks to perform by the RedoManager. It contains basic graph operations used by the RedoManager.

Interface

```
List<Nodes> GetRoots();
List<Nodes> GetParents(Node node);
List<Nodes> GetChildren(Node node);
List<Nodes> GetChildren(List<Node> node);
Node GetNode(String id);
void deleteTableFromPredecessor(String id, String table);
void increaseParentCompleted(String id);
void readyToRun(String id);
void setMetadataDownloaded(String id);
void setDigestDownloaded(String id);
void setRestored(String id);
```

TaskPriorityQueue

This is a special type of priority queue, used to hold the following tasks to be executed by the thread pool. The priority is defined by a combination of operation and the node that is associated.

The tasks have the following priorities: 1. DownloadMetadata: regular 2. DownloadDigests: lowest 3. ApplyDigest: highest

This priorities means, that we focus on applying the digest more than downloading, tasks of type 3 are only created when all the related data was already downloaded, and the predecessors were already applied.

The nodes have the following priorities. We enumerate the nodes in the HistoryGraph using Breath-first search, the lower the number, the higher the priority. Intuitively, what this means is that nodes closer to the root, e.g. that are going to be executed first in the recovery, are downloaded first.

Interface

```
void AddOp(Task task, PriorityLevel prio, String nodeId);
Task GetNextTask();
Boolean freezeBelowLevel(PriorityLevel prio);// This function is only used when the memory is overprovesioned, and we don't want to allow more tasks of type 1 and 2, but is left generic, for future use
Boolean restart();// To be used after a freeze was applied
```

ThreadPool

Normal Java ThreadPool. It's composed of a set of available threads that will run the threads that are stored in the TaskPriorityQueue.

Task

This inherits from Runnable. And it executes one of three types of tasks that was presented in the section Redo Recovery Execution. When this operation is over, it indicates the RedoManager, that the task was succesfully completed. Additionally it call the corresponding functions in the HistoryGraph

Algorithms

This section describes the main operations that are performed in an ETDB system.

Bootup

```
def boot(tables):
        # This function get the set of partitions that are currently associated with the tables
        # and also the set of all the partitons that have been associated with those tables
        P,oldP = getAllPartitionsFromTTP(tables)
        # Lock all partitions, this is done using the Music Interface
        for p in P:
                locked = lockPartition(p)
                # If lock was not succesful
```

```
                if not locked:
                        # Explained in another section
                        result = requestLockRelease(p)
                        if not result:
                                raise exception
                        else:
                                locked = lockPartition(p)
                                if not locked:
                                        raise exception
        # Merge all partitions, explained in another section, using MusicInterf
ace
        mergePartitions(P)
        # Pull data from Music data tables, using the MusicSQLManager
        pullDataFromMusic(tables,P)
        # Apply Redo, using the Redo Recovery
        RedoRecovery(tables,P,oldP)
```

Request Lock Release

```
def requestLockRelease(p,ownershipType,priority):
        # Obtain the url of the owner from the PI table
        owner = getCurrentOwnerFromPi(p)
        # Request ownership using GRPC
        # Current owner receives the query and using the ownership type and pri
oriy, it decides is if release it
        # Releases the lock if required
        # Replies with decision
        result = remoteOwnershipRequest(owner,p,ownershiptType,priority)
        return result
```

Partition Merge

```
def partitionMerge(PartionInfoArray)
```

Transaction Commit

1. Query is submitted to the mdbc client
2. Local transaction Id is generated for query submitted (LTxId)
3. Element is appended to local data structure RunningQueries
    - This element contains:
        - **LTxId:** local transaction id, unsigned integer
        - **RedoRecordId:**
        - **CommitRequested** bool indicating if the user try to commmit the request already.
        - **SQLDone:** bool indicating if sql was already commited, atomic bool
        - **MusicDone:** bool indicating if music commit was alredy performed, atomic bool
        - **Statement:** reference to a statement object. This is used to complete a commit if it failed in the original thread.
        - **Timestamp** last time this data structure was updated
4. If required data is fetched from MUSIC
    - Pull the set of keys
    - Do an atomic gets on each of the keys
        - **Why atomic gets and not locking the key**
            - If the mdbc node can hold all the required data for a node, then it would need to pull the data only once from MUSIC, given that not outside entity could have a lock for the range
5. Query is executed in the local mdbc SQL Tx Database
    - Each new databae change (insert,write,delete) is saved into a local shadow table.
6. Client send a commit request to mdbc driver, RunningQueries data structure is updated
7. Generate a local Commit Id (CommitId), unsigned integer that is monotonically increasing.
    - This is going to be problematic when the integer overflows. We can solve this in two ways.
    - One is releasing the transaction lock and obtaining it, and then reinitailizing the local commit id.
    - The other one is having logic to using numeric types that can hold bigger values.
8. Commit transaction to Music
    - Push new row to RedoLogTable
    - The submission contains the following components:

        - **LeaseId**: id associated with the lease related to the transaction table;
        - **Counter**: is the CommitId generated
        - **Operations**: is an array of two strings, , which represents the old values of the columns and the new values of the columns
        - The primary key for this in cassandra:
            - Partition: LeaseID
            - Clustering: Counter
9. Mdbc commits to local SQL database
10. SQL DB returns commit result
11. If Tx commits succesfully: -> Mdbc assigns Id to transaction

Redo Recovery to MDBC Server

Redo Commit to Data Table