Continuent Tungsten

Replicator Technical Whitepaper

Table of Contents

Basic Architecture	3
Components	8
Extraction from MySQL	19
Extraction from Oracle	21
Applying to JDBC Targets	23
Applying to Data Warehouse Targets with Batch Loading	27
Applying to Native Targets	30
Filtering	33
About Continuent	36

Copyright © 2017 Continuent Ltd

continuent

Basic Architecture

Tungsten Replicator is a high-performance database replication application that can extract data from various sources and apply that data to databases, data warehouses, NoSQL databases and messaging platforms. Tungsten Replicator is a complex piece of software that is based upon a few key and core design principles.

- Tungsten Replicator is optimized for replicating data from one data source to another as quickly as possible.
- Replicators are designed to replicate data continuously without the need or requirement to pause. In general, replicators are configured and left to execute without requiring monitoring or management.
- Data is exchanged using a core flexible architecture-, database- and data-neutral format.
- Data is replicated exactly between source and target, even if they are different database types, allowing for, and taking into account environment, storage, and data format differences. For example, a row in MySQL is replicated into Oracle as a row, but into Kafka as a message, and MongoDB as a BSON document.
- Data sources and targets are abstracted, so that it's possible to read from or write to any database or target.
- The basic unit of data exchange is the database transaction. The replicator always reads or writes an entire transaction.
- The basic unit of execution is the service. A single service reads from or writes to a data source.
- Each service is configured as a series of stages, configured into a pipeline. Data is moved from one stage to the next through the pipeline, one transaction at a time.
- Each replicator is capable of supporting multiple services within a single replicator instance.
- Within each stage, the data can be manipulated (filtered) by removing, modifying, or adding information.
- Each service is either online (actively operating) or offline (paused).
- Each replicator is by default configured to operate with full data integrity; in the event of failure transactions are rolled back or canceled, and errors cause the replicator to stop.

Collectively, this means that a replicator can:

- Replicate between databases in a heterogeneous or homogeneous deployment.
- Replicate from multiple sources and to multiple targets.
- Change, format, and filter information dynamically.
- Replicate data continuously and at high speed.
- Replication can be stopped, paused, and restarted without losing or skipping data.

In general, replicators are deployed as follows:

- A master replicator is deployed on the same host instance as the database and is configured to read data from a data source and generate the transactions.
- A slave replicator is deployed on the same host as the target database, and reads data from the master replicator over a network connection and applies the transactions to the target data store.

Therefore, there are typically at least two replicators in a deployment, and they are configured to extract and apply transactions from a source database to a target database. In each case, the unit of information transferred and used as a reference point is the transaction.

Copyright © 2017 Continuent Ltd

continuent

Transactions

The fundamental unit of information transfer in Tungsten Replicator is the transaction. Within most databases, the concept of a transaction is the update of one or more data objects in a manner that is considered to be consistent. Whether your database is transactionally capable or not, most data updates can be identified as a single operation. For example, transactions have distinct implications within MySQL or Oracle. Within a document-based and eventually consistent database such as MongoDB a 'transaction' is usually the operation of updating a single document.

Within data sources that support ACID (Atomicity, Consistency, Isolation, Durability) transactions, the rules and implications of ACID compliance can be duplicated. For databases where ACID compliance is not available, a 'best case' solution is used, for example, writing a record (and allowing the record to be rewritten if rollback is not supported). The exact semantics and method used depend on the databases being replicated.

As each transaction is read from a data source, the entire transaction and all the data created, deleted, or updated during that transaction are stored into the internal data format used by the replicator, the Transaction History Log (THL). Each transaction is given a monotonically increasing unique ID (the transaction sequence number inside the THL is called “seqno”). This ID can be used to identify the transaction, and is also used to identify the progress of the replicator. By using the sequence number it's possible to identify the replication point. If the replication stops (whether requested or due to an error) the transaction number (seqno) can be used to restart replication from the correct position without repeating or skipping data.

The implications of working on a transactional basis are:

- Transactions are distinct elements and supported blocks within many databases. For those data sources that support it, only full transactions are committed and verified on disk, and transactions that fail will be rolled back.
- Only complete transactions are read from a data source. Partial transaction reads are not supported.
- Only complete transactions are written to a target data source (where possible). For databases where ACID compliance is not supported, transactions may be repeated or re-processed.
- The current sequence number and position are only advanced when the replication of the transaction from the data source to THL, or from THL to the data target have completed in full.

The process of replicating data is therefore entirely centered around the transaction:

1. Read a completed transaction from a data source.
2. Convert the transaction into an architecture-neutral THL format.
3. Give the transaction a sequence number.
4. Record the current source replicator position along with the sequence number and data source.
5. Transfer the transaction to target replicator(s) as THL.
6. Apply transactions in the THL to a target data source.
7. Record the current target replicator position along with the sequence number and data source.

One feature of the transactional nature of the replicator is that replication can be started, paused and restarted using the sequence number (seqno) as a reference point.

Copyright © 2017 Continuent Ltd

Starting, stopping, restarting

Each replicator records the position based on the current role: the extracted source data is used on a master, and the data applied to the target for a slave. When extracting data, this information is used to identify the position within the source data where transactions have been read from. When applying data, the same information is written as each transaction is applied to the target database.

For example, within MySQL the current replicator position is recorded into a table within the database (trep_commit_seqno), using the current sequence number (of the data extracted) and source position (MySQL binary log file and position within the file).

If a master replicator is stopped, either because replication has been requested to be stopped (i.e. during maintenance) or due to a failure to apply, the current 'position' is recorded. When the replicator is restarted, the recorded position is retrieved and used to calculate where to begin reading from. For example, if the replicator is stopped when transaction sequence number 72 has been fully extracted, transaction reads will continue from sequence number 73 when the replicator restarts. More importantly, replication will continue by reading the next recorded transaction on the source database.

On a slave replicator, the THL is applied to a target database by transaction. When transaction 94 has been fully written, this status is recorded. If the replicator is stopped, when it restarts it will then apply sequence 95.

If the process of either extracting or applying a transaction fails, the transaction is not written to the THL, or committed to the target database. The use of the transaction in these cases makes not only for a useful logical boundary, it allows for a logical start/stop point. The transaction, as a single block of data, becomes the boundary for all operations.

Replicator roles

All Tungsten replicators are configured to have one of two roles, master or slave. There are some important attributes and distinctions between the two roles:

- Master – extracts data from a source and generates THL on disk, as well as serving THL requests from slaves over a network connection.
- Slave – reads THL from a remote replicator and applies the data to a target data source. A slave is also capable of serving the THL to another downstream replicator.

Functionality	Master	Slave
Stage dbms-to-q	Yes	No
Stage q-to-thl	Yes	No
Stage remote-to-thl	No	Yes
Stage thl-to-q	No	Yes
Stage q-to-dbms	No	Yes
Extracts from data source	Yes	No
Extracts from remote THL	No	Yes
Applies to data source	No	Yes
Shares THL over network	Yes	Yes

The stages are important in terms of understanding the operation of the replicators, since each stage is responsible for a specific type of data movement process. It is the combination of the stages that define the role of the replicator and what it is capable of.

Each stage performs one of the following operations:

- Converts source database information (MySQL binary log, Oracle data) into THL.
- Provides a queue to store or process data, for both ingesting and generating THL data.
- Converts THL data into a target database format to apply into a target data store.

However, each replicator can have multiple pipelines configured, but only one pipeline, and therefore one role, is enabled at any one time for each service. This means that a replicator can also switch roles at any time. For most normal deployments this does not happen, but when the replicator is configured as part of Tungsten Clustering, the ability to switch the replicator role and pipelines without needing to redeploy or change the configuration is a significant part of how manual switch and automatic failover is handled.

In the event of a role switch, the use of the transaction boundary is also exploited. In an environment where you are replicating between two MySQL servers that transaction boundary becomes the common point. If HostA is a master and HostB is a slave of that master, and roles change at transaction 1722, then 1722 becomes the reference point. When roles are switched on both hosts, HostB starts extracting transactions with sequence 1723 and HostA expects to apply sequence 1723 next.

The significance of the roles affects not only the operation of the replicator, it also has an impact on the abilities of the replicator and what information is exchanged. For example, when extracting data and adding information such as column names, this data can only be extracted from the source database from which the replicator is reading transactions.

The use of the individual stages and the pipelines used to define services within the replicator also makes it possible to create a pipeline that both extracts data and applies it to a target. The use of this direct method of replication, however, also imposes a different problem in the flexibility of the extraction and apply process.

Despite the differences between the roles, the underlying exchange of information between replicators is comparatively basic.

Replicator communication

All replicators are designed to be standalone in their operation. Although replicators either generate THL to be shared over the network, or read remote events from the network, the communication is not a required part of the operation.

A master replicator, for example, will read transactions from a source database and generate the THL for those transactions, regardless of whether a slave replicator is connected to the replicator and reading those events. This allows a master replicator to extract data and generate THL independently.

A slave replicator will continue applying data to a target database if there is local THL available even if the connection to the master has gone offline. This means that a slave can continue to apply data even if the source replicator or master database is offline.

In each case, the replicators operate independently. For replication to occur between the two replicators both must be online and able to communicate with each other, but both can operate without that communication taking place.

When a slave replicator first connects to a master replicator, some basic information is exchanged to ensure that the master replicator and THL match through the use of an EPOCH number, which is a sequence number in the THL. Once this information has been exchanged and established, the THL on the master is sent to the requesting slave in a continuous stream of THL data. Only when communication is broken, for example when going offline or failure, is the handshake information exchanged again.

The use of a single port makes management, configuration, and monitoring of the network used for the replicator very simple. This stream of information, and single channel of data, means that the THL can be sent over a single port and at a very high speed without a significant protocol overhead. The default port for communication between replicators is TCP port 2112. This can be changed, and can also be encrypted using TLS and SSL certificates.

The actual transmission of the information over this port is done as quickly as the THL can be generated. One effect of this process is that a single network interface could become saturated if the data can be extracted from a source database quickly enough. There is no way to throttle the network communication, since this would ultimate negate the high-performance characteristics.

Replicator management

The replicator provides a management interface through the Java JMX interface. This interface supports local and remote control over a network. This allows all aspects of the replicator to be controlled both from a command-line interface and other compatible tools. Since the replicator is a core part of the Tungsten Clustering product, it is this interface that allows the Tungsten Clustering manager to control the replicator (i.e. bring the replicator online or offline), or change the roles and operational stance.

For standalone deployments, the replicator is controlled and managed through a single command-line interface tool called trepctl. This single tool supports all the required functionality to set the replicator status, roles, modes, and control the operation.

Installation and configuration of the replicator and all the associated tools, including the filters, command-line tools and other components, is done via a powerful tool called tpm (Tungsten Package Manager).

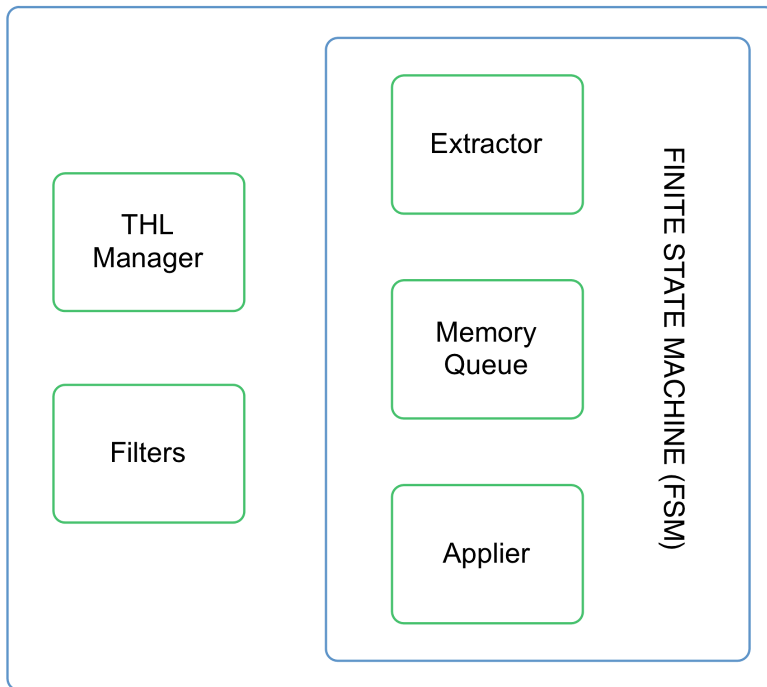
In the majority of situations these tools and utilities are used very sparingly. The replicator has been designed to self-manage processing and replication. Once installed, configured and launched, a replicator will continue to operate until it is unable to due to factors outside its control, such as a data source or network failure. The replicator automatically manages the connections, operational log files and the generated THL files.

Ultimately this means that the replicator does not require active management to keep it running. Only in the event of a failure or configuration change are changes to the replicator operation required.

Copyright © 2017 Continuent Ltd

Components

The replicator is composed of different components that work together in order to support the overall service. It should be noted that the components described in this section work together to support a single 'service', that is, the extraction or application of data from or to a single source. A single replicator can support multiple services.



The individual components work together as part of the configured service pipeline. Pipelines and their role in execution will be described in further detail later in this document, but it is important to understand the process of extracting or applying data is only possible through this pipeline and stage approach.

There are six main components to the replicator.

Three are directly related to specific stages of the replication process:

- Extraction - extract data from a data source.
- Internal Queuing - store the transaction information in memory.
- Applier - apply the data to a target data source.

The remaining three components support the stage processing model:

- Filters - enable modification of the transaction stream.
- THL Management - store and manage the THL.
- Finite State Machine - manages the replicator processing state.

We will take a brief look at these components before digging into specific details.

Extractors

When reading data from a source database, the entire process needs to be coordinated and managed. The replicator must know where the data was extracted from, how to continue extraction in the event of a pause or restart of the replicator, and how to manage the source system so that the log, state and progress can also be managed.

The current list of available extractors:

- MySQL up to 5.7, including Oracle MySQL, Percona MySQL and MariaDB, open source and commercial
- Oracle, using Change Data Capture (CDC), for Oracle 10i and 11i, open source and commercial
- Oracle, using Tungsten Redo Reader, for Oracle 9i through to 12c

The role of the extractor is to take the data from the source system and resolve that data into a transaction to be stored within the THL. This process is critical to get right because the information must be extracted and stored in such a way that the data can be perfectly reproduced on a target system, whether the target system is the same database as the source or a completely different database and environment.

For example, when reading from a transactional database the information can be extracted as a statement (if we know it's being applied to an identical system), or as row-based data if the target database is unknown. If the data is a completely different format, then it must be extracted in a way that enables it to be reconstituted. This may also require extracting session-based information, variables, settings, and even column names and primary key data.

Internal queuing mechanism

The replicator has to move data between stages and also make that data available to different components. For convenience, this is handled by placing the stream of transactional data into an internal queue. The role of the queue is to provide a cache and buffer the data. This is used both to provide the information to other replicators over the network, and to provide a ready list of transactions to be written to disk or applied to a target database.

Because the replicator is designed to operate at such a high rate of speed, it cannot rely on continually reading information from disk only when the next stage of the pipeline has finished processing an item. In order to process the data so quickly, the internal queue is always kept primed with new transactions, and this operation happens in the background, independent of the extraction or apply process. The internal queue can also provide a useful staging area for performance filtering without interrupting the core extraction or apply process.

Appliers

The appliers take the incoming stream of transactions and apply that information to the target database or system. Like the extraction process, the role of the applier is to translate the architecture and database neutral format of the THL into a suitable target structure.

For example, when replicating row-based information to a transactional JDBC target, the data must be reconstructed into a suitable INSERT statement. But when applied to a document-based database, such as MongoDB, the data must be converted into a BSON document. The applier is entirely responsible for this process of translating the information into the correct format, and ultimately writing the data to the target.

As part of the transactional nature, the applier component is also responsible for ensuring that the data has been correctly written and that it is safe to mark that transaction as replicated, which enables the replicator to proceed with the next transaction.

Copyright © 2017 Continuent Ltd

Filters

Filters enable data within the stream of transactions to be changed. Filters perform modifications to each transaction as the data moves through each stage of the replicator. A filter can add information or data to a transaction, it can modify the data in a transaction, or it can filter and remove the transaction entirely from the data stream.

One or more filters can be associated with each stage. The filter mechanism, although part of the stage and transaction stream, effectively sits outside the overall process since there can be multiple filters and even multiple instances of the same filter on each stage.

THL management and storage

All data that is extracted from a data source or that is applied to a target database must have been stored in the THL format. A THL management component provides both the mechanism for translating the information into the THL format, and the method for serializing that information so that it can be stored on disk or exchanged over the network. For a slave replicator, the information also needs to be read from disk for those occasions when the master replicator is unavailable.

For the information stored on disk, the THL manager not only serializes and deserializes the transactions between the in-memory queue and on-disk representation, it also manages the THL files. Services can be configured so that THL is automatically removed after a set period of minutes, hours or days, but the THL is only removed if it has been successfully applied.

Finite state machine

The finite state machine (FSM) is the component that manages the execution of the pipeline and the replicator as a whole. The FSM has been designed so that a replicator service is always in a known, single, fixed state, and that the existence of an active state helps to define and identify the current state of replication itself.

The FSM manages these states and each service will either be in the active state, or transitioning to or from that state. There are two primary states:

- Online - the replicator is actively extracting data from a source or applying data to a target.
- Offline - the replicator is idle.

All movements between the different states, for example transitioning from offline to online are executed on the boundary of a single transaction. Therefore, as described earlier in this document, a single transaction is either entirely processed (and committed) or the process fails. This state based model ensures the status, and progress, of the replication is known and identifiable at all times.

Within the two major states, there are a number of different sub-states, which either define a transition or identify further information about the current state. For example:

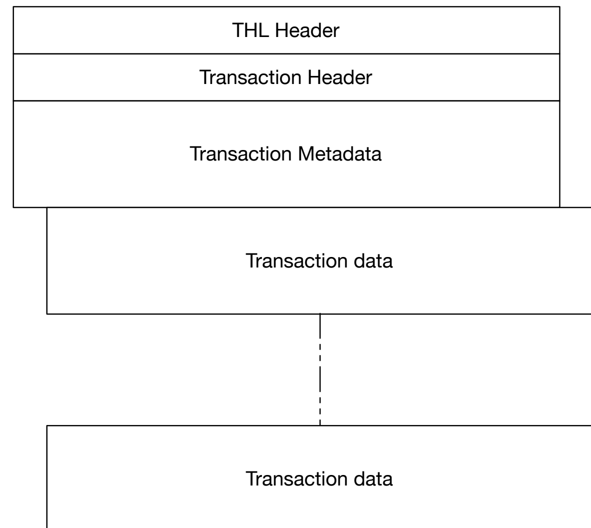
- Going-Online:Synchronizing - indicates that a slave replicator is waiting for the master replicator to come online.
- Going-Online:Provisioning - indicates that a master replicator is extracting source data from a data source before replicating change data.
- Offline:Error - indicates that replication has failed due to a replication error.

These states are vital to the operation as they both indicate the state and available next state or step in the process.

Copyright © 2017 Continuent Ltd

THL

The transaction history log (THL) is the core data exchange component within the replicator. All data extracted is written to THL, and all data applied is converted from the THL format into the native format of the target database.



THL has been specially designed to be as flexible and adaptable as possible to store the required information, while remaining as format- and architecturally- neutral as possible. The role of the THL is store the information extracted from a database in such a way that it can be perfectly replicated on a target system, regardless of whether the target system is the same as the source, or different. As described elsewhere in this document, the role of THL is to store a representation of the data without knowing what the target database environment or system is.

The major features of the standardized THL environment are:

- Identification of each transaction with a unique ID (seqno).
- Correlation between the unique ID and an identifiable extraction point from the source database.
- Ability to store metadata and change data.
- Architecture neutral storage format. The data is recorded in a way that handles big-endian/little-endian differences, and the data can be replicated between different architectures without loss or corruption.
- Transactional semantics are not lost. Data is always stored on a transaction-by-transaction basis.
- Data can be split across multiple files.
- Data can be split across multiple fragments in the transaction.

The THL consists of only one block of the first three items, and one or more blocks of the transactional data. Keep in mind that a single transaction may consist of one or more updates to one or more tables or schemas. Although the management and header information is the same for the entire transaction, the transactional data may consist of one, hundreds, or even thousands of statements or rows of data.

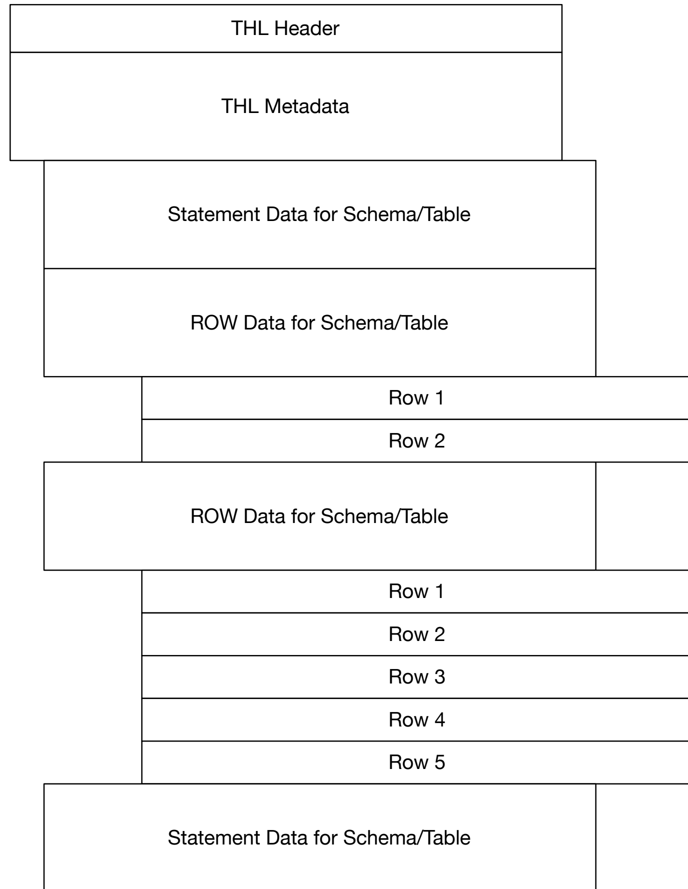
The size of an individual THL event is dependent on the size of the transaction, and for reasons of size, verification, checksums and overall size management, a single transaction may be split into one or fragments.

The thl command-line tool provided with Tungsten Replicators allows you to see the raw THL information. For example, here's the output from a statement-based THL event:

```
SEQ# = 2 / FRAG# = 0 (last frag)
- TIME = 2017-06-03 07:21:01.0
- EPOCH# = 0
- EVENTID = mysql-bin.000022:0000000000001078;218904
- SOURCEID = demo-c11
- METADATA = [mysql_server_id=1;dbms_type=mysql;tz_aware=true;service=alpha;shard=msg]
- TYPE = com.continuent.tungsten.replicator.event.ReplDBMSEvent
- SQL(0) = SET INSERT_ID = 3
- OPTIONS = [##charset = ISO8859_1, autocommit = 1, sql_auto_is_null = 0, foreign_key_checks = 1,
unique_checks = 1, sql_mode =
'NO_ENGINE_SUBSTITUTION,NO_AUTO_CREATE_USER,ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,ERROR_FOR_DIVISION_BY_ZE
RO,NO_ZERO_DATE,NO_ZERO_IN_DATE', character_set_client = 8, collation_connection = 8, collation_server = 8]
- SCHEMA =
- SQL(1) = insert into msg.msg values (0,'Hello')
```

Copyright © 2017 Continuent Ltd

THL and transactional headers



The core of the THL is the structure of the transaction and additional metadata and environmental information that goes along with it. This header information exists for all transactions, regardless of the underlying transactional data. In the previous example, the first three lines contains the THL header:

```
SEQ# = 2 / FRAG# = 0 (last frag)
- TIME = 2017-06-03 07:21:01.0
- EPOCH# = 0
```

The header first defines the THL sequence number (2 in this case), and the fragment, alongside whether the fragment is the last for this transaction. The time is the original time the transaction was committed to the source data store, and the EPOCH number is used as a validation check to ensure the THL is from the same master.

```
- EVENTID = mysql-
bin.000022:0000000000001078;218904
- SOURCEID = demo-c11
```

The next section contains information about the source data, the EVENTID and the SOURCEID. EVENTID is the data store reference from where the information was loaded. In the example above, we show the MySQL binary log file and byte position within it where this transaction started. The source ID is usually the hostname.

Metadata

The metadata portion within the THL contains information about the database environment in which the data was extracted. The information is represented as a list of key/value pairs. From the example, information about the MySQL server, include the ID, whether time zones had been configured, and the database type have been recorded.

```
- METADATA = [mysql_server_id=1;dbms_type=mysql;tz_aware=true;service=alpha;shard=msg]
```

The metadata in this section is common to the transaction as a whole and is most often used to store information about the source and any processing that may have occurred. The replicator also uses this area to record whether the data has been filtered or modified in some way, for example, by adding primary key information, column data, or other settings.

Transactional data

The actual transactional data is contained within the final block and the exact format depends on whether the information was extracted as a statement or row-based statement, or a combination of the two. A single transaction, and THL event, could consist of multiple SQL statements and ROW events. For all statement and row events, the schema is always recorded outside of the actual transaction data. For row-based events, the information contains both schema and table information.

Statement-based events

For statement-based events, the transactional data contains the information required to execute the same statement from the source database on the target database. At the current time, MySQL to MySQL deployments are the only environment in which statement based replication is supported.

The information contains within a statement block includes the statement environment, and the DML statement that generated the data change. For example:

```
- SQL(0) = SET INSERT_ID = 3
- OPTIONS = [##charset = ISO8859_1, autocommit = 1, sql_auto_is_null = 0, foreign_key_checks = 1,
unique_checks = 1, sql_mode =
'NO_ENGINE_SUBSTITUTION,NO_AUTO_CREATE_USER,ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,ERROR_FOR_DIVISION_BY_Z
ERO,NO_ZERO_DATE,NO_ZERO_IN_DATE', character_set_client = 8, collation_connection = 8, collation_server =
8]
- SCHEMA =
- SQL(1) = insert into msg.msg values (0,'Hello')
```

There are multiple SQL statements; the first (SQL(0)) sets the insert ID (auto-generated primary key), and the MySQL database environment settings such as whether auto commit has been enabled, and the SQL mode supported. This information has all been extracted verbatim from the information written into the MySQL binary log.

The actual DML statement is in SQL(1). Note that SCHEMA is blank; this is because the SQL statement has been executed on the source database with an explicit schema/table reference. Multiple statements would have their own SQL and OPTIONS blocks.

The benefit of statement-based replication is that the size of the replication is much smaller, and therefore requires less time, less disk, and less bandwidth. Statement-based cannot replicate data to non-MySQL platforms as the statements cannot be rewritten for different targets, nor applied to non-SQL or transactional systems.

Row-based events

With row-based events, the information stored is more descriptive and substantial than with statement-based events. This is because the actual data, rather than the statement that generated the changes, is stored in the THL. The changes are recorded, including both the operation type (i.e. INSERT or UPDATE) and the data itself. Additionally, because a single transaction could consist of many changes of each type into a number of different tables and schemas, each individual operation needs to be recorded as a separate block, one for each table updated.

The following information can be stored for each row entry:

- Index (position within the original row)
- Column name
- Column data type
- Length of datatype (for character or binary fields)
- Unsigned (for numeric values only)
- BLOB type (which typically requires special handling)
- Description

Copyright © 2017 Continuent Ltd

For example:

```
- ROW# = 0
  - COL(index=1 name= type=4 [INTEGER] length=8 unsigned=false blob=false desc=) = 1
  - COL(index=2 name= type=0 [NULL] length=0 unsigned=false blob=false desc=) = NULL
  - COL(index=3 name= type=0 [NULL] length=0 unsigned=false blob=false desc=) = NULL
  - COL(index=4 name= type=93 [TIMESTAMP] length=0 unsigned=false blob=false desc=) = 2017-08-10 16:48:31.0
  - COL(index=5 name= type=0 [NULL] length=0 unsigned=false blob=false desc=) = NULL
  - COL(index=6 name= type=0 [NULL] length=0 unsigned=false blob=false desc=) = NULL
  - COL(index=7 name= type=4 [INTEGER] length=8 unsigned=false blob=false desc=) = 0
  - COL(index=8 name= type=12 [VARCHAR] length=0 unsigned=false blob=false desc=) = MASTER_ONLINE
  - KEY(index=1 name= type=4 [INTEGER] length=8 unsigned=false blob=false desc=) = 1
```

Technically row-based information is actually stored in two separate blocks, called the row-values and the row-keys. This is a misnomer, because the row-values are the new information for the row, while the row-keys are the old information. That is, for an UPDATE event, the key-values contain the version of the row before the update, and the row-values the version after the update has been completed. The old data is required to do a look-up on the information to ensure we are updating the correct row if no primary key is provided.

These two sides of the row-event data can be used according to the different operations. An INSERT contains only row-values (new data), UPDATES contain both, and DELETES contain only key-values, the values required to lookup the correct row to be deleted.

This can be seen in the example below where a different block of row changes has been recorded for three separate tables.

```
- OPTIONS = [foreign_key_checks = 1, unique_checks = 1, time_zone = '+00:00', ##charset = UTF-8]
- SQL(0) =
 - ACTION = INSERT
 - SCHEMA = msg
 - TABLE = msg
 - ROW# = 0
  - COL(1: ) = 73661367
  - COL(2: ) = Insert a value
- OPTIONS = [foreign_key_checks = 1, unique_checks = 1, time_zone = '+00:00', ##charset = UTF-8]
- SQL(1) =
 - ACTION = UPDATE
 - SCHEMA = msg
 - TABLE = msgb
 - ROW# = 0
  - COL(1: ) = 1
  - COL(2: ) = Update an row
  - KEY(1: ) = 1
  - KEY(2: ) = Update an entry
- OPTIONS = [foreign_key_checks = 1, unique_checks = 1, time_zone = '+00:00', ##charset = UTF-8]
- SQL(2) =
 - ACTION = DELETE
 - SCHEMA = msg
 - TABLE = msgc
 - ROW# = 0
  - KEY(1: ) = 2
  - KEY(2: ) = Deleted value
```

The use of row-based events and extraction is what enables the heterogeneous functionality within the replicator. Because the information has been represented as raw data, it is possible to take the information and reproduce it on a target database irrespective of the target environment or data structure and interface. For example, when applying to a JDBC target an INSERT INTO statement can be constructed with the right fields and values. For an update, an UPDATE statement. For this to operate correctly, the replicator on the applier side must get metadata from the target database to understand the target fields so that this information can be used to construct the statement.

There are some useful side effects of this process that can be taken advantage of:

- The target database is not required to have an identical structure, providing the information can be written into the target.
- The table type could be different, for example within MySQL a table could be InnoDB or TokuDB, enabling migrations.
- Character sets could be different (if compatible).
- The size and definition of fields does not have to be identical. Data could be replicated into larger fields, or TEXT or BLOB in place of simple CHAR or VARCHAR.
- Indexes could be different.
- Order of fields could be different, or there could be more fields on the target than the source.

All of the above differences are possible without needing the filtering mechanism within the replicator to perform any changes. Instead, just the use of row-based extraction provides this functionality because of the way the data is ultimately applied into the JDBC targets.

When applying to a document-based database, the values can be assembled into the target document structure, such as JSON or BSON document. This flexibility enables the data to be effectively formatted or applied into the target database irrespective of what that structure or environment is like. However, using row-based events for heterogeneous apply into non-JDBC targets requires some changes to the structure of the information.

To understand that, let's look at the structure of the individual row events within the THL...

Row-based inserts

A row-based insert contains the information to be inserted into the record. In most cases, it is the raw information and consists of a list of all the fields in the source table and what data was inserted. If the information was NULL or not defined, this is also represented:

```
- ACTION = INSERT
 - SCHEMA = msg
 - TABLE = msg
 - ROW# = 0
  - COL(1: ) = 73661367
  - COL(2: ) = Insert a value
```

Note how the information stored here does not contain column name information. For JDBC targets this information is not required if the data is in the same order on the source and target tables.

However, when applying to heterogeneous targets, the column name can be useful and even required. For example, when replicating to a document-based target the column name is required so that the field name within the document is available.

Also note that there is no index or primary key information. Again, for JDBC this is often not required for a simple insert, but a document database or message system like Kafka uses the primary key information to identify the document or message ID.

Depending on the data source on the extraction side, this information may be needed or required when applying to heterogeneous or batch targets. Oracle, for example, includes this information automatically. On MySQL, the column name and primary key information must be separately updated within the THL using filters.

The result is THL that contains the column names and the key portion of the row update now explicitly contains the list of primary keys:

```
- ACTION = INSERT
 - SCHEMA = msg
 - TABLE = msg
 - ROW# = 0
  - COL(1: id) = 73661367
  - COL(2: msg) = Insert a value
  - KEY(1: id) = NULL
```

Note the actual value is not populated in the 'key', instead it is just a placeholder to identify which column(s) are the primary key.

Row-based updates

As noted earlier, a row-based update operation contains the old values of the incoming data before the update (in the key-values) and the new data (in the row-values):

```
- ACTION = UPDATE
 - SCHEMA = msg
 - TABLE = msgb
 - ROW# = 0
  - COL(1: ) = 1
  - COL(2: ) = Update an row
  - KEY(1: ) = 1
  - KEY(2: ) = Update an entry
```

Using this information, it should be possible to use the old values to lookup the row in the target database and then update it with the new data.

It's possible with the combination of information to either model a version of the old row of data, or the new row of data. Even if the incoming row only contains changed fields, an entire version of the record can be reconstructed by merging the old and changed data together.

However, in a heterogeneous applier this is inefficient, and for batch appliers, unworkable. Performing full lookups across a target table or environment could be expensive. Instead, the primary key and column name filters are enabled on an extractor used for heterogeneous replication, and the key-values information is used to indicate the primary key information which is then used as the more efficient lookup value.

Row-based deletes

For a delete operation, only the data to be deleted is recorded, that is, the old data required to lookup the row:

```
- SQL(2) =
 - ACTION = DELETE
 - SCHEMA = msg
 - TABLE = msgc
 - ROW# = 0
  - KEY(1: ) = 2
  - KEY(2: ) = Deleted value
```

Thus, only the key-values of the THL are populated.

Again, for a true heterogeneous replication environment this information becomes the primary key lookup data.

Copyright © 2017 Continuent Ltd

Immutability

Extracted THL cannot be modified or altered in any way. Once the data has been extracted and written to disk, the THL is considered immutable. The THL therefore becomes an exclusive record of the transactions from the source system.

The only way that data can be modified is during the replication stream inside the pipeline stages. This means that data can be extracted from a source database and recorded into the THL without any modifications, then used as the source data to be applied to a variety of targets where the content and data can be filtered or changed to suit the destination environment.

Thus, THL extracted from MySQL can be written directly to a MySQL slave to create a perfect copy, while being modified during replication into Kafka to filter out tables not needed, and modifying data to be compatible with the target Kafka queues. The original source material stays in place with the changes only occurring within the applier replicators.

The role of the THL in this case is that it can act as a canonical reference source for the information extracted. The immutability however does mean that if additional information or data is needed, it must be added by the replicator performing the extraction before the THL is written to disk.

For example, for heterogeneous targets, the THL must have the column name and primary key information added to the THL before the data is written to disk. There is no other source for this information than the source database used during extraction.

Storage

Storage of the THL information on disk is organized as one or more files; each file contains one or more transactions, dependent on the transaction size and the configured size of the THL files. For example:

```
-rw-rw-r-- 1 mc mc   100067271 Aug 11 02:25 thl.data.0000000001
-rw-rw-r-- 1 mc mc   100170874 Aug 11 02:31 thl.data.0000000002
-rw-rw-r-- 1 mc mc    67169531 Aug 11 02:32 thl.data.0000000003
```

This can be compared to the THL index showing the sequence numbers for each file:

```
LogIndexEntry thl.data.0000000001(0:1233)
LogIndexEntry thl.data.0000000002(1234:2444)
LogIndexEntry thl.data.0000000003(2445:3192)
```

The size of the THL files can be managed to allow for differences in the storage environment and properties, and also to cope with transactional sizes and potential recovery scenarios. Transactions are not split across THL files, as this would increase the potential for corruption if only partial THL files were distributed.

THL files are also self-managed by the replicator using a simple expiry mechanism. For example, if the expiry is configured for five days, THL files are deleted once the last transaction in them is more than 5 days old.

The expiry works differently depending on whether the replicator is an extractor or an applier:

- On an extractor, the expiry always operates and deletes THL files accordingly. This happens whether or not a downstream, slave replicator has transferred the content.
- On an applier, the files are only deleted if the data has been applied to the target, in conjunction with the expiry setting. For example, if the replicator is ten days behind, but the expiry is set to five days, no THL will be deleted.

THL can also be viewed and manually purged, but the purging can only take place at the beginning or end of a given file, a single event cannot be removed from the middle of a given THL file.

Copyright © 2017 Continuent Ltd

THL position during extraction and apply

The replicator is able to stop, start, and reset the position from which replication takes place, whether this is during the process of extraction or applying data. For any replicator, the information is stored within a suitable structure usually within the source or target database. This helps to retain transaction consistency and ensure that the database extraction and target apply are in sync with each other. For example, on MySQL and Oracle the position is recorded within a tracking table within the database, called trep_commit_seqno. This table contains both the seqno and extraction position.

The replication position is recorded within the THL as the EVENTID, and is then associated with the THL sequence number. The replicator always stops on a transaction boundary, i.e. a single THL sequence number. In MySQL, the extraction position is the binary log file and byte position within that file. For Oracle, it is the system change number (SCN).

Once this information is stored within the THL, the replication can be paused, or go offline, or stopped altogether by recording the THL sequence number in the corresponding tracking table. When the replicator is placed online, or restarted, the same table is consulted and used to identify the restart position.

However, because this information cannot always be relied upon, a logical sequence of sources is used to determine the position.

On the extractor, the semantics are:

8. Consult the stored event extraction position in the source database table and use the stored EVENTID.
9. Check the THL stored on disk and find the EVENTID in the last stored transaction.
10. Start extracting with the current source database position (binlog or SCN).

This ensures that even if the source database-tracking table has been removed or corrupted, then the extraction can continue from a known reference point.

For an applier, the position is computed slightly differently because the source of the information is not the same. The sequence number, rather than extraction position, is used as the point of reference:

1. Consult the stored position (EVENTID) in the target database-tracking table and get the associated sequence number (seqno).
2. Check for the sequence number in the stored THL and request the next THL position from the upstream master replicator.
3. Request the first stored sequence number from the remote master.

For verification purposes, when requesting THL from a master, if the information is known from the stored position table, then the EPOCH, or sequence number when the master went online, is used to verify that the THL source and stored THL match.

Should it be necessary, the extraction or apply position can be explicitly modified and set before the replicator is placed online, which allows for recovery or replay scenarios.

Copyright © 2017 Continuent Ltd

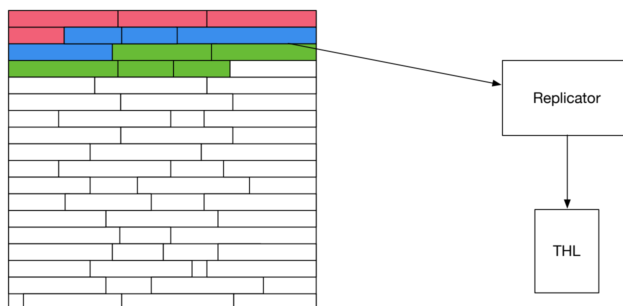
Extraction from MySQL

MySQL is one of the primary sources and targets for the Tungsten Replicator. MySQL supports the generation of a binary log. This is a sequential list of the individual transactions and corresponding transactional data written to the MySQL server in sequence. Because MySQL generates this information in the order of the events being written, it is possible to replay and reapply the individual transactions on a target MySQL server and reproduce the data originally applied to the source server.

Architecture

The architecture of the MySQL binary log read extraction is actually very simple, as seen in the diagram to the right. The binary log is a sequential list of transactions from the source database, and so it is already a good source of information. All the replicator needs to do is parse the content of the binary log and translate the information into THL.

Fortunately, the MySQL binary log is a sequential list of changes, and so each transaction stored within the binary log is read and extracted as a single transaction to be stored within the THL. This results in a one-to-one relationship between the stored transaction within the binary log, and a stored transaction with the THL.



The process of reading the binary log content is constant. Of all the available extraction methods, the MySQL is probably the easiest to understand because it is a simple one-to-one mapping of the source recorded transactional information and the transactions recorded in the THL.

Reading the binary log

Reading from the binary log by the replicator is a simple process of tailing the end of the log file looking for transactions that can be extracted and translated into THL. The process of reading the log is a constant stream, always waiting for new writes and extracting and reading the new transaction as quickly as possible.

Positional information (the binary log filename and byte position within the log file) is used as the reference point for reading data. This position is attached to each THL sequence number so that replication can gracefully stop and restart reading at the correct position.

Reading the binary log from disk has some advantages over reading the data through other methods. The primary benefit is that the information can be read directly without requiring the database to be running or active, and without requiring any additional load on the source database. It also means that if the database fails, the replicator is still able to read fully committed transactions on disk. Since the replicator is a key part of the Tungsten Clustering product, such considerations are vital to ensure the switches and failover work correctly.

Reading from disk also has the benefit that in a fully running and operational system, the data written to disk is likely to still be within the operating system disk cache, and is therefore available in memory instead of via an expensive disk read.

The content of the binary log contains full transactional information, recorded as either statement-based or row-based events. In MySQL, transactions are recorded within the binary log according to a combination of the settings and the type of statement.

Copyright © 2017 Continuent Ltd

Three modes are supported:

- Statement-based logging, which records the original SQL statement executed.
- Row-based logging, which records the row-based changes that resulted from an executed statement.
- Mixed logging, which uses the most efficient/data-integrity-safest operation depending on the original executed statement.

This is ultimately stored and represented within the binary log as a row or statement based operation, tagged against the schema and table information.

Metadata and environment

The content of the binary log contains information both about the transaction and the environment in which the statement was executed. This includes active SQL modes, environment variables, timestamp and other data required when the statement would be re-executed on the slave MySQL node. This metadata environment is required because these values have the potential to change the operation, i.e. the SQL mode could change the way MySQL reacts to data types or failures.

By looking at the raw MySQL binary log and the THL generated, it can be seen how the metadata and environment for the transaction are recorded and stored so that replication can be achieved on another MySQL target.

An example raw MySQL binary log extract can be seen below.

```
SET TIMESTAMP=1502415176/*!*/;
SET @@session.pseudo_thread_id=3275/*!*/;
SET @@session.foreign_key_checks=1, @@session.sql_auto_is_null=0, @@session.unique_checks=1,
@@session.autocommit=1/*!*/;
SET @@session.sql_mode=1436549152/*!*/;
SET @@session.auto_increment_increment=1, @@session.auto_increment_offset=1/*!*/;
/*!\C utf8 *//*!*/;
SET @@session.character_set_client=33,@@session.collation_connection=33,@@session.collation_server=8/*!*/;
SET @@session.lc_time_names=0/*!*/;
SET @@session.collation_database=DEFAULT/*!*/;
```

Below is the corresponding content within the THL metadata:

```
- METADATA =
[mysql_server_id=1;dbms_type=mysql;tz_aware=true;strings=utf8;service=alpha;shard=sales20;tungsten_filter_
columnname=true;tungsten_filter_primarykey=true;tungsten_fil
ter_enumtostring=true]
- TYPE = com.continuent.tungsten.replicator.event.ReplDBMSEvent
- OPTIONS = [foreign_key_checks = 1, unique_checks = 1, time_zone = '+00:00']
```

Local and remote reading

The binary log is an artifact that exists and is generated by the MySQL server for the purposes of replay/backup and replication. As such, native replication within the MySQL server is provided by generating the binary log on the master, then sending the generated log content over a network connection to a slave MySQL server.

The preferred mode of extracting data from MySQL is to read the binary log files directly, but the replicator is also capable of connecting to the MySQL server as a replication client, reading the binary log remotely in the exact same fashion as a native MySQL slave.

This allows the replicator to operate either by reading the files directly or by reading them entirely remotely over a network. This can be used in situations where direct reading is not supported or allowed to due to tighter security situations, or in environments where direct access to the MySQL server is available, such as Amazon Web Services RDS deployments.

Copyright © 2017 Continuent Ltd

continuent

Binlog management

In most situations, the replicator will avoid making modifications or changes to the way that MySQL manages and organizes its binary logs. This means that operations such as flushing logs, switching binary logs, or purging old logs continues to be handled by MySQL.

However, there are situations when the replicator must manage the binary logs to ensure the integrity of the data, particularly when the replicator is operating within a Tungsten Clustering environment. For example, during a switch operation inside the cluster, a new MySQL binary log can be triggered in order to speed up the process of identifying the correct replication point.

The replicator can also stop, start and reconfigure native replication in order to assist in the identification of the start position to ensure that the binary log and THL information match, particularly during role or THL changes.

Because all of these operations are normally only allowed within a privileged environment on a server where the Tungsten user has SUPER privileges, binlog management can be disabled. This allows effective, albeit limited, operation in AWS RDS environments where SUPER privileges are not supported.

Extraction from Oracle

Two methods of extracting data from Oracle are supported in Tungsten Replicator, Change Data Capture and Redo Reader. The older, Change Data Capture (CDC) method is supported for Oracle 10g and Oracle 11i sources. CDC is a built-in function of the Oracle database that captures database changes into a duplicate set of tables, and it is these tables which are used as the source of replication information, since they quite literally contain a list of all the changes in the source database.

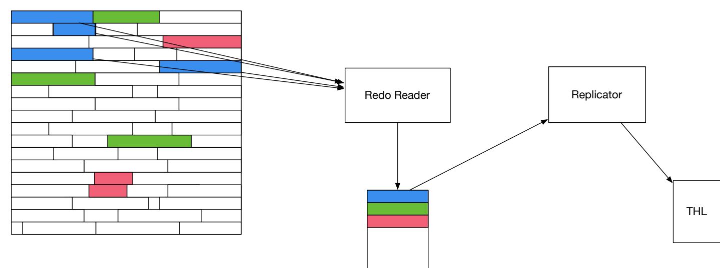
The Redo Reader method is newer and supports a much wider range of Oracle sources, from Oracle 9g to Oracle 12c and the operation more closely matches the binary extraction from the MySQL binary log. The redo reader accesses transaction information directly from the logs generated by Oracle, and this makes it both faster and less intrusive, with less impact on the source database than the CDC method.

Whether you use Redo Reader or CDC, the resulting data extraction from Oracle is always in row-based format once written into THL, and therefore automatically usable for all homogeneous and heterogeneous replication needs.

Architecture for Redo Reader

The Oracle Redo Reader extracts information from the Oracle Redo and Supplemental logs directly by parsing their raw content. The redo and supplemental logs are not sequential or ordered in the same way as the MySQL binary log, and transactional information is distributed around the log files. Therefore, the extraction process is more complicated and happens through two separate stages:

- A redo reader process reads the contents of the redo and supplemental logs and generates an interim log, called the plog, which contains a sequential list of transactions created from the distributed transaction information.
- The replicator reads from the contents of the PLOG files to extract a sequential list of transactions that are translated into the THL format.



Copyright © 2017 Continuent Ltd

The Redo Reader and Replicator work together to perform the extraction, but also work independently. This means that a redo reader process can operate without the replicator running.

The redo reader handles the extraction, identification and translation of information into the appropriate format for the replicator to be able to read the data, while the replicator handles the translation of the extracted data into the THL format for replication apply and re-distribution to other downstream slaves.

Direct and off-board replication

Because the redo reader process is separate from the Replicator, two separate deployment methods are available, direct and off-board. In a direct deployment, the redo reader and Replicator are installed on the same host as the Oracle database instance, and the redo log and supplemental log files are read directly.

In an off-board deployment, the redo reader component is installed on the Oracle database instance, with the Replicator installed on a different host. The redo reader component is supported on the same range of operating systems and CPU architecture platforms as Oracle, whereas the replicator has been certified only on a select group of Linux hosts.

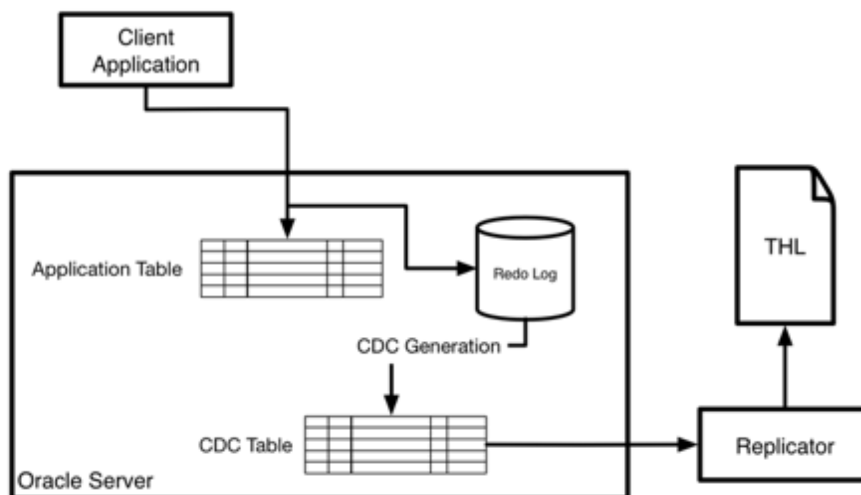
Using the off-board method it is possible to extract from a much wider range of Oracle databases than would be supported with the native Replicator. Using off-board replication also makes it easier to deal with the relatively small impact of reading the redo logs and the replication process by moving that onto another host entirely.

Architecture for CDC

The Oracle Change Data Capture (CDC) technology is provided by the Oracle database itself, although the exact methods and solutions differ based on the Oracle edition being used. The underlying principle is for the database to copy data changes into a set of tables that sit alongside the source tables being replicated.

The CDC tables, that is, the tables that contain a list of changes in the source tables, are then read by the replicator in order to extract the information and then that data is appended to the THL as standard THL events. There is a one-to-one mapping for the tables that are being monitored, and the CDC tables that contain the change information.

Setting up CDC requires settings within the Oracle database to be enabled and the tables to be monitored configured and added to the CDC process.



Copyright © 2017 Continuent Ltd

The actual duplication of the changes is handled internally within the database using one of two methods:

- Synchronous - on Standard and Enterprise Editions of Oracle, database triggers are used to copy the changes on the monitored tables. Each time an operation is performed on the monitored table, a trigger is fired and the changes copied to the CDC table. This model is considered to be synchronous, in that the changes written to the monitored tables are synchronously copied to the CDC tables as part of the original transaction. This method does, however, introduce an overhead to the original database operation.
- Asynchronous - on Enterprise Edition only, a separate process within Oracle that parses the list of changes stored in the redo logs and uses that information to update the CDC tables with the list of changes. Changes to the tables for applications operate at full speed because no triggers are fired, but asynchronous mode is potentially slower because changes are only represented within the CDC tables after transactions have been committed.

The CDC method overall is efficient, but generally slower than direct reading of the redo logs due to the nature of the CDC table generation and the subsequent extraction of the data by the replicator. The overall process operates as follows:

11. Application inserts transaction into the database.
12. Change is copied to CDC tables.
13. Replicator reads the CDC tables and translates into THL.
14. CDC data in the tables is deleted.

Because continually reading the CDC tables would itself represent an overhead on the Oracle server, the replicator will delay reading if data has not been written for some time. If there have been few changes, it could take up to a minute before the replicator tries to read the CDC tables.

In addition to the delay, CDC is also limited in terms of the data types it supports, and increases both the database load and storage requirements as additional copies of the data must be stored within the CDC table temporarily during extraction.

Applying to JDBC Targets

JDBC targets are comparatively easy to apply data to because JDBC has a standardized method for applying data built around the semantics of a typical SQL statement. Applying into such a target is therefore a case of reconstituting a statement that will perform the corresponding action. Because the JDBC interface works within the transactional basis, reproducing the source transaction on the target is straightforward.

Beyond the basics of developing the statement to be execute, the other consideration is building the environment in which the statement should be executed, and then recording the transaction and replication state.

Copyright © 2017 Continuent Ltd

Architecture

The basic architecture of applying to a MySQL target is to connect to the database using a JDBC connections using the appropriate JDBC driver, build a suitable statement, and then apply the required statement and operation to the database as part of each incoming transaction. Building a suitable JDBC statement is relatively easy, since it is possible to create templated forms of the appropriate statements and to fill in the necessary fields. For example:

For an INSERT operation:

```
INSERT INTO table (collist) VALUES (values)
```

For an UPDATE operation:

```
UPDATE table SET column = value WHERE column = oldvalue
```

For a DELETE operation:

```
DELETE table WHERE column = value
```

In addition to the construction of a suitable statement, the replicator creates the environment in which the statement should be executed (i.e. setting auto increment values, time zone and other settings). The applier also stores the current status information into the appropriate tracking table.

Thus, the typical database applier process is:

1. Start transaction (BEGIN statement).
2. Build the database environment and settings to enable the transaction update.
3. Execute statement or rebuilt ROW statement to target database for each row of the course database.
4. Commit the transaction (COMMIT) statement.
5. Record the progress into the trep_commit_seqno tracking table on the target node.

This process is repeated on a transaction-by-transaction basis.

In addition to the basic process, building and deploying an applier for each environment requires supporting the following elements:

- Build the basic templates for the INSERT/UPDATE/DELETE statements. Although the basics are the same for all databases and environments, minor differences in elements such as the WHERE clauses, table, column and other definitions all require a slightly different format.
- Configure the methods for specifying rows, columns and schemas. For example, some databases use a period to separate a schema and table while some use an underscore.
- Configure the commands and settings to configure the environment for executing the statement. In MySQL, this is supported through SET statements, but others may use functions or stored procedures.
- Configure the nomenclature and commands required for handling transactions.
- Set the table definition, format, and update semantics for the trep_commit_seqno tracking table and other management tables used by the replicator to monitor and control replication.

The exact tools, methods and operations for a typical JDBC applier are common, although minor differences exist.

Copyright © 2017 Continuent Ltd

Applying to MySQL

MySQL is a special case when it comes to replication due to the fact that the MySQL extraction process reads and stores MySQL statements as well as MySQL row-based events. This means that the exact method used to apply data into MySQL depends on the method and storage format of the incoming data:

- If the incoming data stored within the THL is statement-based (and therefore from a MySQL source), the statement can be applied directly to a MySQL database target as the source MySQL statement that was executed on the slave.
- If the incoming data in the THL is a row-based event, whether from MySQL or Oracle, the data is applied as a row-based JDBC statement by building a suitable JDBC/SQL statement.

In a MySQL-to-MySQL replication topology, the replicator can extract and apply data whether it had been recorded as statements, rows or in the mixed methodology supported by the MySQL binary log. For non-MySQL targets, applying row-based events is based on the JDBC SQL statement semantics.

The only other major facet of application of data within the MySQL environment is the reproduction of the MySQL environment. As noted when the data was extracted on MySQL, a huge range of environmental information is extracted from the MySQL binary log to provide information and data about the SQL mode, auto increment, time zone, character set etc. information.

All of this is recreated on the target during replication with the settings applied as efficiently as possible. In a typical MySQL replication deployment, many settings are unlikely to be changed regularly between statements or rows so the current settings and environment are cached. In the event of a loss of a connection, or when bringing the replicator online or during startup, the entire environment is recreated and set as a whole using the metadata stored within the THL.

There are some additional benefits when applying to MySQL:

- Within MySQL-to-MySQL deployments, DDL statements are replicated and applied. Therefore, table creation, column updates, etc. are all extracted and applied verbatim. This allows full replication of data comparable to the native MySQL replication.
- Row-based data-application can be optimized to improve performance. For example, if changes have been made to the same table using multiple statements within the course of a single transaction, this can be optimized into a single INSERT/UPDATE/DELETE statement.

When performing heterogeneous deployments and replication, the environmental metadata that can be shared and replicated is limited by what has been extracted and what can be applied. Certain settings, such as time zones and character sets are supported, but non-MySQL extraction will obviously not support the MySQL-specific SQL_MODES and other settings, and so the default MySQL global settings are used instead.

Applying to Oracle

When applying data into Oracle, the source THL information is going to be recorded as row-based events, since Oracle extraction always stores information in rows, and heterogeneous replications requires row-based storage.

The process of applying the data into Oracle therefore matches the basic JDBC target as outlined above. The data is embedded into different statement templates and then executed as a basic statement. Oracle supports transactional semantics, and basic environment settings, and the status and positional data is stored within a table within the Oracle database so that the state and transaction can be managed.

Copyright © 2017 Continuent Ltd

Parallel apply

Due to the transaction support offered by JDBC environments, the replicator supports the ability to apply data to JDBC targets in parallel, using multiple threads to separate the data on a transaction-by-transaction basis using the active schema to apply the data, and then making use of a round-robin thread apply basis.

For example, given 6 sequential transactions within the source THL across four different schemas, the data could be applied using three separate threads and applied as follows:

Transaction 1 (Schema A, Table A) applied on thread 1

Transaction 2 (Schema C, Table A) applied on thread 2

Transaction 3 (Schema A, Table B) applied on thread 1

Transaction 4 (Schema B, Table C) applied on thread 3

...

In order to protect transactional consistency and avoid transaction ordering and isolation causing problems, the transactions are divided across the threads on schema boundaries. This operates as efficiently as possible, while also ensuring parallel threads are not updating the same schemas out of the original source order.

Thus, in the above example, transactions 1, 2, and 4 are applied simultaneously, even though they are technically out of order from the incoming sequential transaction stream. This only possible because the replicator can identify and prevent execution of updates that would affect the same schema/table.

To further avoid problems, the replicator also monitors the apply progress and controls the number of parallel threads being used during the apply process to prevent non-sequential execution of transactions creating too much of a distance between each transaction. For example, if your application updates multiple schemas, but there are more transactions within one schema than others, one thread may end up 10 or 20 transactions ahead of the next thread.

Ultimately, the replicator will attempt to throttle back the parallel threads and limit the drift, although this reduces the performance gained.

When using parallel apply, the following aspects should be kept in mind:

- The number of schemas in your target database. If you have only one schema, parallel apply will not help.
- The transaction load should be evenly distributed across all your schemas.
- The number of parallel applier threads enabled should be tuned to support the maximum number of transactions of your target database, number of schemas, and transaction load.
- Care should be taken with large or long-running transactions in single schemas because it introduces larger separation between threads.

Finally, the replicator attempts to most efficiently handle and manage the process of going online and offline, ensuring that transactions in mid-execution across multiple threads complete properly. Each parallel thread has its own status and progress marker stored within the apply table, and when adjusting the number of threads the table is consulted and 'condensed' into the reduced number of threads.

Copyright © 2017 Continuent Ltd

Applying to Data Warehouse Targets with Batch Loading

Data warehouse targets are databases that may support JDBC and may have traditional row-based storage, but are usually optimized for column-based, rather than row-based, storage. This means that inserts and updates are better optimized when loading larger quantities of rows simultaneously. For this reason, the simpler single transaction-based updates, especially if the transaction contains a small number of rows, are inefficient.

In many cases, it's possible to load thousands, even hundreds of thousands of rows in the same period of time that a single row update could be made. Data warehouse targets are also traditionally very bad at performing delete operations, since the role of a delete in data warehouses is an anathema. However, since the replicator is responsible for replicating data, including deletes, all source data operations must be applied into the target.

The solution to this problem is to batch multiple transactions from the source database together into a single operation when it is applied to the target. This way, instead of applying only singe rows, thousands of rows can be modified at one time. To further improve the overall process, the size of the batches and the frequency can be controlled, and to maximize the performance, the data is written to a file and imported, as data warehouses are optimized for that process.

For batch loading to work, the source data must have a primary key, because in order to perform updates each row in the data must be uniquely identifiable. Without a primary key, it becomes impossible to accurately identify the data, and it also makes it difficult to identify the individual rows within the change data so that inserts and deletes can be performed.

Architecture

The basic process is as follows:

1. Generate a batch file (using the character-separated format) containing the rows to be inserted/updated/deleted.
2. Connect to the target data warehouse and import the file.
3. Combine the existing stored data (the base table) with the incoming data (the staging data).

The first stage is exclusively handed by the replicator; the latter stages are normally handled by the replicator, but rather than being handled directly, are in fact managed through a JavaScript script, with the JavaScript environment forming part of the replicator applier process.

The use of JavaScript makes the batch loading very flexible, and easy to modify and adapt for new targets without having to write new, custom, appliers directly within the replicator. It also allows for a significant amount of customization for different areas of the process, i.e. allowing copies of the change data, or custom imports.

The first step is to generate a file that contains the change data. A single file is created for each table within each schema. This file has some specific characteristics:

- Each row is prepended with information about the row data, including the transition ID, operation type (insert, update etc.), and commit timestamp.
- It contains all the raw row data.
- In the default mode, it contains inserts and deletes as single rows. Updates are represented by two rows, a delete to remove the 'current' version of the information, and an insert to add the new version of the information.

The generation of a file with these characteristics actually creates a very precise source of all of the changes to a table that can then be used to track all of the updates. This could be used for auditing or historical recording of database changes. But for the basic process, it contains an effective list of all the changes.

For the loading process, the native CSV import is used to get the data into a staging table where the record of all the changes is stored until it can be merged into a target table. At this point, because we are batching updates, the CSV that has been generated may contain multiple changes to the same row of data. How this is resolved is handled by the final stage.

The final part of the process is to perform a merging, or materialization, process. This converts the list of all the transaction changes into the 'base' data, that is a targeted and resolved version. This process is:

1. Select all of the rows in the staging table, and delete all the rows in the base table marked to be deleted.
2. Insert the 'last' version of each row marked as an insert. The last version is the one with the highest THL sequence number. If there are 10 changes to a single transaction, the last change in a batch will always be the version of the row that matches that transaction, because it was the active version at the time the row was extracted.

Data is batched into the generated CSV files when written out according to two rules:

- The number of rows in the incoming stream,
- The period since the last transaction.

For example, you could set 10,000 rows and 30 seconds, and the transaction would be applied to the target when either threshold is reached. So, if 10,000 rows were batched, but only 1 second had elapsed, the transactions would be committed to the target.

Obviously, this batching of information introduces a potential latency and delay for replication, since the interval could be as high as the period interval setting. In general, the combination of the two settings should help to control the overall replication latency.

There are minor differences between different target data warehouses as to how this process works, but the basic mechanics are always the same.

Applying to Vertica

When applying to HP Vertica, the JavaScript batch loading script handles the loading of information directly into Vertica from the generated CSV files. The actual process is as follows:

1. Generate a CSV file.
2. Connect to Vertica over JDBC.
3. Start a transaction.
4. Use the COPY command to import the CSV into a staging table.
5. Delete everything in the base table marked as a delete in the staging table.
6. Insert the latest version from the staging table of the data into the base table. This uses nested SQL statements within Vertica to select the correct rows.
7. Commit the transaction.

With Vertica, the convenience of having a JDBC interface to the database makes it very easy for the replicator to apply the data. The Vertica applier is probably the simplest and most straightforward of the batch applier processes for this reason.

Copyright © 2017 Continuent Ltd

Applying to Redshift

Amazon Redshift is a column-store optimized database that works and operates in a similar fashion to Vertica, while operating entirely within the Amazon Web Services (AWS) cloud environment. Within the replicator, the basic process is identical to Vertica, except that for Redshift, the CSV file must be uploaded to Amazon's S3 storage service before it can be imported into Redshift.

For Redshift, the process is therefore:

1. Generate a CSV file.
2. Upload the CSV file to S3.
3. Connect to Redshift over JDBC.
4. Start a transaction.
5. Use the Redshift COPY command to import the data from S3.
6. Delete the base table data.
7. Insert the latest version of the new data.
8. Commit the transaction.

For Redshift, due to the nature of the S3 platform, the process can be configured so that the CSV data is kept and not deleted, so that you have a long-term history of all the database changes. You can also configure it so that the data is imported into a long-term table.

Applying to Hadoop

Apache Hadoop, and all the different flavors of Hadoop such as Cloudera or MapR, all operate on a very similar basis. A clustered environment that supports a distributed file system, HDFS, and a distributed computing processing platform that uses Map Reduce process in order to parse the information stored.

Unlike most other traditional databases, the storage format for information on Hadoop is generally a text (CSV or other) file rather than a specific format. This is because HDFS is an append-only file system, which means that changes to files on disk are impossible, you can only write a new version of the data and delete the old version.

For database replication, the replicator performs slightly differently with Hadoop in that the replicator stops once the data has been stored within HDFS. The materialization process is handled by a completely separate set of tools, because the Map Reduce nature of the Hadoop environment, while efficient for very large volumes of data, can take seconds or even minutes to complete. If the materialization was handled internally by the replicator, it would also slow down the replication of the raw data. Instead, the replicator is dedicated to getting the data into HDFS as quickly as possible.

For replicating into Hadoop therefore the replicator operates as follows:

1. Generate a CSV file.
2. Copy the generated CSV file into HDFS.

A separate tool operates using the Hive database environment to provide the materialization process. As a separate unit, it can operate on a different cadence to the replicator applying the data into HDFS. This can be tuned so that specific tables are materialized at different rates and frequencies. But it also means that the materialization process within Hadoop can make use of the entire Hadoop Map Reduce environment. In a high-volume situation, this means that hundreds of thousands of nodes can be used to merge the data across many gigabytes or terabytes of data.

Copyright © 2017 Continuent Ltd

Applying to Native Targets

The application of data to native targets requires different considerations and operational processes than when applying to either JDBC or batch environments. Whereas JDBC and batch targets are typically row-based databases that often support SQL and/or JDBC interfaces, native targets include databases and environments that may not use rows or SQL as their interface.

However, native targets allow for the application of data into different types and forms of database or storage mechanisms, which makes them practical if you want to apply data into message platforms, or 'eventually consistent' NoSQL targets such as MongoDB.

Native targets have the following differences and considerations:

- Applying to a native target requires modification of the incoming data into a format supported by the target. For example, translating the row-based information into a document base, or message.
- Conversion of the information may be necessary to cope with differences in data types. For example, dates, times, binary data.
- Transactional boundaries may be difficult to achieve, because not all targets support ACID compliance.

The major underlying difference between native targets and both batch and JDBC targets is that the information must be applied using the native API or interface to the target system. We cannot use SQL or even batch-based methodologies - instead, the replicator has to be explicitly adapted to support the new target. This means that for a native applier target there is no common architecture; instead, the method and process for applying into the target environment are entirely reliant on the target in use.

Applying to MongoDB

MongoDB is a document-based eventually-consistent database where data is represented as a document - that is, a single representation of data where the information is stored as a list of a key value pairs. Values can themselves be further key value pairs. Each individual document is stored within a single database, called a collection, and each document is identified by a unique document ID.

To apply the data into MongoDB, each incoming record is converted into a MongoDB BSON document, which is similar the JSON format. Each document is identified using the primary key if incoming data has one, or using a self-generated ID if not.

Changes to the database are modeled as follows:

- An insert creates a BSON document and adds it to the document collection with either the primary key or self-generated ID as the document ID.
- An update creates a BSON document with the new version of the information, and replaces the stored version, either using the primary key as the document ID or by performing a full table lookup using the old data as the search parameters.
- A delete removes the entry from the collection using the primary key as the document ID or by performing a full table lookup using the old data as the search parameters.

Applying data into MongoDB is often used as a method of storing cached data and effectively replicating data into the target can be achieved with very low latency because the data is updated instantly through the native applier and API.

Applying to Elasticsearch

Elasticsearch is a text and database indexing platform rather than a pure database. It's designed to take incoming information and index it to provide super quick data lookups, without necessarily providing full access to the information. Therefore, it sits alongside a database as part of the front-end application interface and cache.

Applying data into Elasticsearch requires translation of the incoming data from the THL into a JSON format which defines the content that will be indexed and a reference to be able to load the original from the source database.

The interface to Elasticsearch is a REST API and this requires some differences from a traditional database when applying data:

- Storing documents requires using a specific URL that includes the index name, index type, and a unique reference number. The resulting URL is idempotent.
- The index name is by default set by the incoming schema name.
- The index type is by default set by the incoming table name.
- The unique URL reference should be based on the incoming primary key data, or a document reference can be automatically generated.
- The content is based on a JSON document.
- The JSON document contains metadata (including the schema, table and version information).
- The JSON document also contains the raw incoming data.
- When updating or inserting information, the process must use a PUT operation if the full URL and ID has already been created, or use a POST if a document ID is to be generated. Removing requires a DELETE operation, using the unique URL reference for the document.

Because applying data into Elasticsearch is usually adjunct to the main database storage, there are a number of options to control how the information goes into Elasticsearch:

- Document ID can be configured to use the primary key and/or primary key and table and schema details.
- Delete errors (i.e. due to mismatched document IDs) can be ignored.
- Update error (i.e. due to mismatched document IDs) can be ignored.
- Index name can be set explicitly in place of using the schema name.
- Index type name be set explicitly in place of using the table name.
- Schema, table and source commit time can be embedded into the metadata.
- Replicating data into Elasticsearch using the Tungsten Replicator works as follows:
- One Elasticsearch document is created for each incoming row of data from the THL.
- All data is fully replicated, including inserts, updates and deletes.
- Data can be written either locally or remotely directly into the Elasticsearch instance.

Copyright © 2017 Continuent Ltd

The resulting information is added into the Elasticsearch index with a format similar to below:

```
curl http://$HOSTNAME:9200/test/messages/99999?pretty
{
    "_index" : "test",
    "_type" : "messages",
    "_id" : "99999",
    "_version" : 1,
    "found" : true,
    "_source" : {
          "msg" : "Hello Elasticsearch",
          "committime" : "2017-06-06 19:09:20.0",
          "id" : "99999",
          "source_table" : "messages",
          "source_schema" : "test"
    }
}
```

The benefit of replicating data into Elasticsearch this way is that the data can be moved and replicated very efficiently. The use of the REST API in this fashion also opens up the possibility to apply the data into other tools that use a REST API for storing and updating information.

Progress information is stored into Elasticsearch as a unique document within a given index name and type.

Applying to Kafka

Kafka is a high-performance, distributed, message bus system. It can handle a high volume of messages because the actual distribution of the message data is entirely distributed across the cluster of the messaging platform.

The format of a Kafka message consists of:

- A given topic name - this is used to separate different streams of information. Within the replicator, the default topic name is derived from the incoming schema name and table name.
- A message ID - this is an explicit reference for the message so that it can be identified if required. By default, the replicator uses a combination of the schema, table and primary key information.
- The message content - Kafka sends 'bare' messages, in that the content is just a payload to the message, but for interoperability JSON and other serialized formats are used. The replicator translates the incoming data into a JSON formatted message, with the data embedded into a metadata payload.

Unlike other targets, Kafka is not a database, and so when the data is replicated into Kafka instead of inserting, updating and deleting data, the messages sent via Kafka are instead a message that contains the change data.

For example, the following message is an example of one sent via Kafka with Tungsten Replicator:

```
{
   "record" : {
      "msg" : "Hello World",
      "id" : "2384751"
   },
   "_source_table" : "msg",
   "_optype" : "INSERT",
   "_seqno" : "3194",
   "_committime" : "2017-08-15 18:21:32.0",
   "_source_schema" : "test"
}
```

The meta data fields contain the schema, sequence number, commit time and other information and the inclusion of this data can also be configured.

An update is represented like this:

```
{
   "_source_schema" : "test",
   "_committime" : "2017-08-15 18:22:09.0",
   "_source_table" : "msg",
   "_seqno" : "3195",
   "_optype" : "UPDATE",
   "record" : {
      "msg" : "Goodbye",
      "id" : "2384751"
   }
```

The replication works in a similar fashion to the Elasticsearch native applier:

- One Kafka message is created for each incoming row of data from the THL.
- All messages are tagged and identified with the source operation type, schema and commit time.

Progress information, data and sequence number information is stored in Zookeeper, which is a required component for Kafka deployments.

Filtering

Within every stage within the replicator process it is possible to attach one or more filters. Each stage executes the filters in the order defined, and different stages and operations within the system can only operate and use certain filters. For example, you can only add column name and primary key information to THL at the stage where the data is extracted, because for the filter to operate it must have access to the source database.

The role of the filters and purpose of the filtering process is to allow for information to be modified and altered according to the needs of the replication deployment in process.

Filtering architecture

The filter process enforces the following rules on each filter:

- A filter only ever takes in a single THL event at a time, and only returns either the modified THL event or nothing (to indicate the event has been removed).
- A filter can only operate on the THL and event information contained within it.
- A filter is unaware of the database where the data came from.
- A filter is unaware of the data in the database.
- A filter has access through JDBC where available to the source or target database.
- A filter has no concept of the replicator, pipeline or configured elements.

Although filters have to operate within these very confined environments, they are actually very powerful:

- A filter can remove events from the THL
- A filter can remove individual rows from a single THL event.
- A filter can modify the statement in an event.
- A filter can modify the row data within an event.
- A filter can add rows to an event.

Copyright © 2017 Continuent Ltd

- A filter can remove rows from an event.
- A filter can add or remove columns from rows in an event.
- A filter can introduce new events into the THL stream.

This means that filters can perform a huge variety of different tasks, but are best employed when they are used to directly modify or alter the data within the THL event being filtered. Augmenting, adding, or constructing new information within a filter is inefficient. For example, a filter can modify the data within the THL, but it cannot change what information was extracted.

Although a filter has access to the underlying database, it would be expensive to use the filter to add or augment information from the extracted data during replication, because you would have to perform a database lookup. This means that a filter is an inefficient method to use to emulate a typical Extract Transform and Loading (ETL) process.

Instead, filters should be used as a method to remove, simplify, or reformat the information already contained within the THL so that it is practical or usable within its target environment.

For example, filters can be used to:

- Filter out schemas, tables, or even columns from individual tables to remove the information so that it is not replicated.
- Change the format of data, for example translating strings into different formats, changing dates into different date formats (EPOCH into DATETIME), or anonymizing data.
- Appending additional information to data, such as source database names or source hosts so that the data can be more easily filtered on the target.

Regardless of what the filters do, the role of the filter is always to change the streamed information in the path of the replicator.

Filtering points

Filters are applied within a stage within the overall pipeline of the replicator, and also within different replicators. Some filters are only workable at some points, and other filters change the data within the replication stream before the THL reaches the next stage:

- Filters operate within a specific stage in the replicator.
- Multiple filters within the same stage in a replicator operate in order.

The order can be significant both within the stage and across replicators. For example, a filter that adds the column name information to the THL must have been executed before a filter that removes data based on the column name - otherwise there is no way to identify which column to filter.

The application of filters at different points also changes the nature of the THL, and it must be remembered that THL, particularly on disk, is considered a canonical version of the THL and transactional information. The role of the filter is to alter that transactional data to suit the needs of the replication stream.

For example, adding a filter in the master changes the THL for all downstream replicators, which limits the flexibility. Using a filter on a slave replicator at the point before it creates its own local copy of the THL forces the THL on disk to be permanently changed, while filtering/changing the data at the point of applying the data to the database enables you to change the data only for the target environment, without affecting the THL stored on disk

This is best understood by examining a number of different scenarios where filtering is being used. For example, if the user wants to replicate data between two MySQL servers separated by a WAN connection, filtering out tables that are not needed on the target database.

Copyright © 2017 Continuent Ltd

One solution would be to filter out the tables within the master replicator where the data is extracted. This way the data is not in the THL stored on disk, and never transferred to the downstream replicator. This has the following implications:

- Data transferred over the WAN will be reduced, thereby improving the replication latency and time.
- Extracted THL is not suitable for DR purposes, since the THL will never contain a full copy of the extracted data.
- Wanting to replicate filtered tables is impossible, especially historically, which could present a problem if you need to add missing data from the already extracted data stream.

Examining how this affects different THL filtering can be seen by examining how that affects the THL data:

Host	Stage	Filter Affect	Recommended Filters
Master	Binlog-to-q	Modifies the data during extraction	Filters that change/augment the source data stream, such as column names or primary keys.
Master	q-to-thl	Modifies the data before it's converted to THL and stored on disk	Filtering, but only if you can guarantee the tables or data will not be required in the target.
Slave	remote-to-thl	THL downloaded from a master before storage on disk as THL	Removing schemas, tables, columns.
Slave	thl-to-q	THL before it moves to the applier stage	Filters that modify the data or structure of the THL. For example, the filters used to reassign THL events to shards for parallel apply operate in this stage.
Slave	q-to-dbms	Only affects transactions/data as written into the target database/environment	Filters for modifying data specifically for the target database environment such as changing data format/encoding.

Care should therefore be taken to ensure that filters are used and applied correctly according to the environment and data replication requirements. Using the filters at the wrong point could limit or even corrupt the data as it gets replicated into the target environment.

JavaScript filters

The majority of filters within Tungsten Replicator are written in Java, as is the rest of the Tungsten Replicator software, but the replicator also supports a JavaScript environment based on the Mozilla Rhino implementation of the JavaScript engine. This allows for filters to be written and executed using JavaScript.

The JavaScript environment has full access to the Java classes and objects that are exposed through the replicator, which means that the filter can access and process the native Java objects that make up the THL stream of data as exposed within the replicator.

The main benefit of the JavaScript environment is that it allows for very rapid filter development, and also means that users can customize the filters to their own requirements.

Copyright © 2017 Continuent Ltd

About Continuent

Continuent is a leading provider of database clustering and replication, enabling enterprises to run business-critical applications on cost-effective open source software. Continuent Tungsten provides enterprise-class high availability, globally redundant data distribution and real-time heterogeneous data integration in cloud and on-premises environments. Our customers represent the most innovative and successful organizations in the world, handling billions of transactions daily across a wide range of industries.

For more information on our products and services, please visit www.continuent.com, email us at sales@continuent.com or call us at (800) 270-9035, and follow us on Twitter @Continuent.

Copyright © 2017 Continuent Ltd