Capturing Global Transactions from Multiple Recovery Log Files in a Partitioned Database System

Chengfei Liu*
University of South Australia
Adelaide, SA 5095, Australia
liu@cs.unisa.edu.au

Bruce G. Lindsay
IBM Almaden Research Center
San Jose, CA 95120, USA
bgl@almaden.ibm.com

Serge Bourbonnais Elizabeth B. Hamel Tuong C. Truong
IBM Silicon Valley Laboratory
San Jose, CA 95141, USA
{*bourbon, hameleb, tctruong*}*@us.ibm.com*

Jens Stankiewitz
Viterra Informationssyeme GmbH
44803 Bochum, Germany
jens.stankiewitz@viterra.com

Abstract

DB2 DataPropagator is one of the IBM's solutions for asynchronous replication of relational data by two separate programs Capture and Apply. The Capture program captures changes made to source data from recovery log files into staging tables, while the Apply program applies the changes from the staging tables to target data. Currently the Capture program only supports capturing changes made by local transactions in a single database log file. With the increasing deployment of partitioned database systems in OLTP environments, there is a need to replicate the operational data from the partitioned systems. This paper introduces a system called CaptureEEE which extends the Capture program to capture global transactions executed on partitioned databases supported by DB2 Enterprise-Extended Edition. The architecture and the components of CaptureEEE are presented. The algorithm for merging log entries from multiple recovery log files is discussed in detail.

1 Introduction

Data replication is a process of maintaining a defined set of data in more than one location. It involves copying designated changes from a source location to a target location, and synchronizing the data in both locations. Update propagation can be done within or outside the transaction boundaries. This classifies the data replication protocols into two categories: *synchronous (eager)* and *asynchronous (lazy)* [8]. The conventional correctness criterion for synchronous replication is *1-copy-serializability* [4]. Synchronous approach allows update propagation completed before the transaction commits, thus provides data consistency in a straightforward manner. However, the resulting communication overhead increases response time significantly. The asynchronous approach, on the other hand, delays the update propagation until after the transaction completes. The update propagation is normally implemented as a background process. This approach increases response time, but may cause inconsistency since copies are allowed to diverge.

The dangers of synchronous replication have been analyzed by Gray et al. [8], and since then the research efforts have been shifted towards asynchronous replication [6, 14, 3, 5]. Recently efforts have been made to shorten the response time by using group communica-

* Work done while the author was visiting IBM Silicon Valley Laboratory.

Proceedings of the 29th VLDB Conference, Berlin, Germany, 2003

tion primitives in synchronous approach [12, 11, 2].

Many commercial database systems are based on asynchronous replication model [10, 7]. Two strategies have been used in asynchronous approach: *push* and *pull*. The push strategy propagates the updates immediately after the commit of a transaction, while pull strategy propagates the updates at the client request.

DB2 DataPropagator is one of the IBM's asynchronous replication solutions for relational data. It adopts the pull strategy. In DB2 DataPropagator, the task of replication is divided into two separate programs: *Capture* and *Apply*. The Capture program captures the data changes from the log files of source database to staging tables, and the Apply program applies the data changes from the staging tables to the target database. A control center helps users to configure their replication tasks by defining the data source, target and specific requirements.

Current version of the Capture only captures changes made by local transactions in a single database log file. There is no support to the IBM partitioned database system DB2 Enterprise-Extended Edition (DB2 EEE) [9]. With the increasing deployment of partitioned database systems in OLTP environments, more operational data are being stored in the partitioned database systems. This increases the need for the ability to replicate data from DB2 EEE to other systems for data warehousing or backup/recovery purposes. As such, support of Capture in DB2 EEE is required.

In this paper, we present a system called CaptureEEE which enables DB2 DataPropagator and DB2 EEE to work together. CaptureEEE extends DB2 DataPropagator Capture by merging log entries from recovery log files belonging to multiple database partitions in DB2 EEE and then reconstructing global transactions by merging sub-transactions executed on these partitions. Like DB2 DataPropagator, CaptureEEE operates as an independent component. No change needs to be made for DB2 DataPropagator Apply.

The rest of the paper is organized as follows. Section 2 explains the principle of the DB2 DataPropagator Capture program. Section 3 introduces DB2 EEE, including the partition nodes, partitioning of tables, global transactions and partition recovery log files. Section 4 presents CaptureEEE, the work of capturing global transactions in DB2 EEE, a brief introduction of the prototyping is given. Section 5 concludes the paper.

2 DB2 DataPropagator Capture

The DB2 DataPropagator Capture program is a log based replication solution. It scans the recovery log file of a database sequentially and examines each recovery log entry. A log entry is a data structure that describes the action that was performed on an object within the database. In a DB2 recovery log file, each log entry is uniquely addressed by a *log sequence number (LSN)*, which is a relative byte address for the first byte of the log entry. A log entry consists of one common log entry header and a record. The common log entry header contains information detailing the log entry and transaction information, such as a *unique transaction identifier (TID)*. All log entries written by a single transaction contain the same transaction identifier. The transaction identifier ties together all log entries that belong to one single transaction.

A TID is stored in each *data manager (DM)* and *transaction manager (TM)* log entry. A transaction identifier for a log entry that updates, inserts, or deletes data is associated with a DM log entry. A transaction identifier for a log entry that commits or aborts a transaction is associated with a TM log entry. The end of a transaction is signaled with a *Commit* or *Abort* log entry, but the beginning of a transaction does not necessarily write a specific log entry, such as *BeginOfTransaction*.

2.1 Capturing Transactions

Since the log file of a database and its log entries provide information about database activities that happened in the past, complete transactions may be extracted from the log file. In particular, the Capture program extracts transactions by scanning the recovery log file of a database for specific log entries (e.g., log entries that belong to tables that are registered as replication sources within the database) and extracts essential information from the recovery log file.

Capture builds a transaction in memory, until the Capture program sees a Commit log entry for this transaction. The Capture program does not necessarily commit the transaction immediately to the staging tables. The commitment to the staging tables happens after a specified interval has elapsed, and this interval is referred to as a commit interval. Records describing the changes of each transaction within the memory of the Capture program are inserted into the staging tables. Two tables are used to record the transactions captured in the staging area: a *UOW (Unit Of Work)* table for keeping the transaction information, and a *CD (Changing Data)* table for keeping each change made by a certain transaction. By joining the UOW table with CD table, the Apply program is able to apply the changes made by the captured transactions to target data.

2.2 Restart Logic

The term *warm start* refers to the restart of a program, such as the Capture program, with the *warm start* option. The warm start option allows the Capture program to begin from where the Capture program had

stopped at its last termination. Thus, instead of reading the source recovery log file from the beginning, the Capture program continues reading the recovery log file from a previously saved position. In particular, during a warm start the Capture program continues reading the log file from a certain position, which has been stored during the last run of the Capture program.

When a transaction is performing some processing, the transactions is said to be *inflight*. When a commit or abort operation is received for a transaction, the transaction is committed or aborted, respectively, and is no longer inflight. If a commit operation of a transaction happens after a change data (CD) synch point is reached, the transaction is still treated as inflight. The minimum inflight LSN (*MinInflightLSN*) of all inflight transactions is stored at the CD synch point. This LSN ensures that the Capture program does not miss any log entries. If a warm start is required after the CD synch point, a program recognizes that it should start reading log entries for the transaction started with the MinInflightLSN.

Additionally, at the CD synch point, a max commit LSN (*MaxCommitLSN*) is stored. MaxCommitLSN is the LSN of the commit operation of the last transaction that has been processed into staging tables. This LSN ensures that the Capture program does not capture log entries from a single transaction more than once.

The term *cold start* refers to a situation in which the Capture program starts from scratch. All replicas are re-initialized and the Capture program reads the recovery log file of the database from the current end of the log. DB2 provides a log read application programming interface (API) to retrieve log entries from a recovery log file. The log read API returns log entries for tables that have been created with the *data capture changes* attribute. By calling the log read API, the Capture program retrieves log entries that are relevant for data replication. In this manner, overhead is minimized.

Restart is a feature within the Capture program that ensures that replication of changes can be resumed after a restart. The restart of the Capture program refers to a situation in which the Capture program has been manually stopped by the user or stopped without warning due to, for example, a hardware or software failure.

3 DB2 EEE

DB2 EEE extends a database manager to a partitioned database system. A partitioned database system is the collection of all *database partition servers (nodes)*, each has a database manager and a collection of data and system resources. In a partitioned database system, multiple database partition nodes can be assigned to a machine (or to multiple machines), and each database partition node houses a portion of the entire database. This portion of the database is known as a *database partition*. The fact that databases are partitioned across database partition nodes is transparent to users and applications.

3.1 Node Configuration

DB2 EEE can be configured to execute on Massively Parallel Processing (MPP) shared-nothing hardware architecture, in which machines do not compete for resources. Each machine has exclusive access to its own disks and memory, and the database partition nodes that run on the machines communicate with each other through the use of messages. Below is an example of a configuration with three physical partition nodes. The first column shows the node number, the second the machine name, and the third the communication port of on that machine.

Example 1 *A configuration with physical partition nodes*

```
1 dolphin 0
2 sushi   3
3 tide    1
```

Another configuration is running multiple logical partition nodes, in which more than one database partition server runs on a machine. This configuration is useful when the system runs queries on a machine that has symmetric multiprocessor (SMP) architecture. Below is an example of a configuration with three logical partition nodes.

Example 2 *A configuration with logical partition nodes*

```
1 dolphin 0
2 dolphin 1
3 dolphin 2
```

It is also possible to have a mixed configuration of physical and logical partition nodes.

3.2 Table Partitioning

Once we defined the node configuration, we can create several *nodegroups* which are named subsets of the partition nodes in the configuration. Each nodegroup is defined within database partitions that belong to the same database. For example, we can create the following three nodegroups based on the configuration in either Example 1 or Example 2.

Example 3 *Defining nodegroups*

```
Create nodegroup n1 on nodes(1);
Create nodegroup n23 on nodes(2,3);
Create nodegroup n123 on nodes(1,2,3);
```

After nodegroups have been created, we can define some *tablespaces* for a database. For example, we create the following three table spaces.

Example 4 *Defining tablespaces*

```
Create tablespace s1 in nodegroup n1
 managed by system using('n1');

Create tablespace s23 in nodegroup n23
 managed by system using('n23');

Create tablespace s123 in nodegroup n123
 managed by system using('n123');
```

Now we can define tables for the database. For instance, we can define the following table t_1 in tablespace s_{123}.

Example 5 *A table definition*

```
Create table t1(col1 int,
                col2 varchar(50),
                col3 time,
                primary key(col1))
in s123 partitioning key(col1);
```

A table exists in a tablespace. The value of the *partitioning key* is used to map a row to a node by a hash function. In Example 5, the value of *col1* maps a row to a node defined in nodegroup n_{123}.

3.3 Global Transactions vs Local Transactions

In DB2 EEE, a transaction may update more than one partition node. A *coordinator node* of a transaction is the partition node where a transaction is issued; other partition nodes involved in this transaction are called *subordinator nodes*. A *local transaction* makes update only on the coordinator node. All other transactions are called *global transactions*. So there are two types of global transactions in DB2 EEE:
(1) A transaction which updates more than one node, thus has more sub-transactions;
(2) A transaction which makes update on only one node which is not the coordinator node.

In DB2 EEE, two types of transaction identifiers are used: a local transaction identifier (TID) which is the same as the TID in DB2 DataPropagator, and a *global transaction identifier (GTID)* for global transactions. GTID is associated with all TM log entries for global transaction management. All TM log entries for a same global transaction contain the same GTID value. TID is node-local. A subtransaction of a global transaction also uses TID in its DM and TM log entries.

DB2 EEE uses the presumed commit protocol (PrC) [1], a derivative of the two-phase commit protocol (2PC), to coordinate the commit of a global transaction.

In the following, we give an example of two global transactions.

Example 6 *Global transactions*

Two global transactions GTX_1 and GTX_2 are defined. Both inserting some tuples into table t_1 defined in Example 5. Suppose that values 4 and 88 will be mapped to node 1, values 2 and 888 to node 2, and value 1 to node 3. GTX_1 is issued on node 1 while GTX_2 on node 2. Also suppose the order of execution is $GTX_1.Insert$, $GTX_2.Insert$, $GTX_2.Commit$, $GTX_1.Commit$.

```
// Node 1 (GTX1)
Insert into t1 values
 (4,'gx1@1',current time),  // (1)
 (2,'gx1@2',current time),
 (1,'gx1@3',current time)
Commit;  // (4)

// Node 2 (GTX2)
Insert into t1 values
 (88,'gx2@1',current time),  // (2)
 (888,'gx2@2',current time)
Commit; // (3)
```

3.4 Log Files in DB2 EEE

A DB2 EEE database includes multiple database partitions, each with a recovery log file. The recovery log file structure on each partition node is same as in a non-partitioned DB2 database. However, extra TM log entries are created for global transaction menagement. These include *prepare*, *coordCommit*, and *subordCommit*. Like local transactions, DB2 EEE does not keep *BeginOfTransaction* entry for global transactions in recovery log files. For the second type of global transactions, DB2 EEE only generates prepare and subordCommit log entries, no coordCommit is generated in a recovery log file.

In the following, we give an example of DB2 EEE database recovery log files.

Example 7 *DB2 EEE database recovery log files*

Below are log entries generated from the execution of the two global transactions defined in Example 6. STX_{ij} stands for a subtransaction of GTX_i running on node j.

```
// Node 1
  STX11-Insert
  STX21-Insert
  STX21-Prepare
  STX21-SubordCommit
  GTX1-CoordCommit

// Node 2
```

```
  STX12-Insert
  STX22-Insert
  GTX2-CoordCommit
  STX12- Prepare
  STX12-SubordCommit

// Node 3
  STX13-Insert
  STX13-Prepare
  STX13-SubordCommit
```

4 CaptureEEE

In the DB2 DataProppagator Capture, only local transactions are captured from a single database recovery log file. So there is no problem of ordering, the total ordering of log entries can be easily obtained by reading the log entries from the single recovery log file sequentially. The earlier the commit log entry of a transaction is scanned, the earlier this transaction is captured. In DB2 EEE, however, we need to deal with multiple recovery log files. The ordering of the log entry reading and processing has to be worked out first. Regardless it is a local transaction or a global transaction, the transaction executed first must be captured and written to the staging table first.

In DB2 EEE, timestamps are generated using Lamport clock [13] which supports *partial ordering* among the events happened in DB2 EEE. The partial ordering together with the PrC version of two-phase commit logic deployed in DB2 EEE are crucial for capturing global transactions in right order.

The architecture of CaptureEEE is shown in Figure 1. CaptureEEE is composed of the following components: the *log reader*, the *transaction builder*, the *transaction merger*, and the *staging table inserter*. The log reader retrieves the log entries from multiple database recovery log files. The output of the log reader is a single log entry sequence which contains log entries from multiple log files. So the role of log reader is to merge the multiple recovery log files and construct a single logical recovery log file. From the output of the log reader, the transaction builder builds local transactions or subtransactions of a global transaction and stores them into a local transaction memory structure for use by the transaction merger. The transaction merger generates global transactions in the global transaction structure by mapping subtransactions in the local transaction memory to global transaction memory. The staging table inserter uses staging tables to store data of the captured local and global transactions from the local and global transaction memories, while restart table stores restart points for each one of multiple recovery log files.

In the following, we discuss each component of the architecture. In particular, we present the algorithm for merging the log entries in detail. The implementation of the prototype is also discussed.

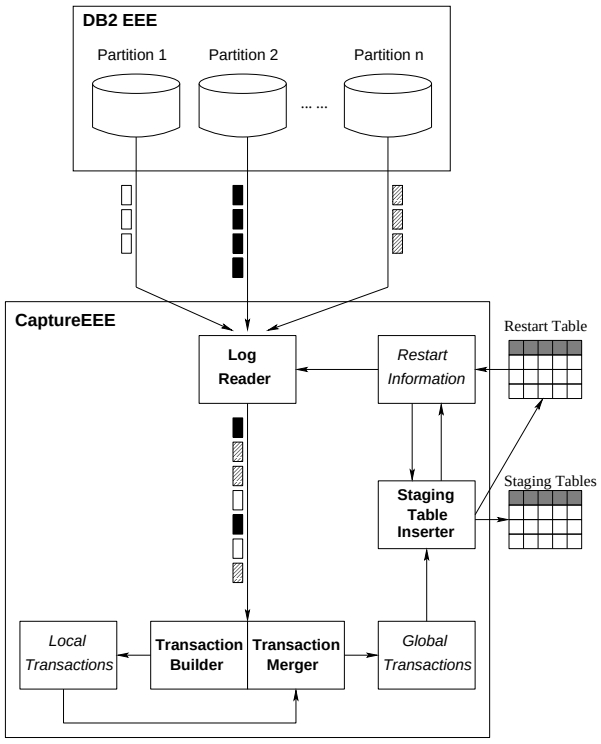

Figure 1: The Architecture of CaptureEEE

4.1 Merging Log Entries

Two approaches to log entry merging are available. The first approach uses a single log reader to scan all partition recovery log files and generate a single log entry sequence. The second approach uses a log reader for each partition node. In the second approach, all log readers can work simultaneously which is good, but additional synchronization effort is required on the simultaneously accessed recovery log files and this synchronization is complicated. In CaptureEEE, the first approach is applied.

As DB2 EEE can be configured to execute on an MPP hardware architecture, no central clock is available to support the ordering of events. In DB2 EEE, Lamport clock is used to generate a timestamp which provides a partial ordering to events. The local clock of each database partition node gets synchronized when the partition node receives a message from other partition nodes. This is also used in implementing the PrC version of the two-phase commit protocol to provide the right timestamps for both coordinator nodes and subordinator nodes. For the TM log entries *prepare*, *coordCommit*, and *subordCommit* of any global transaction t, the following is always true (ts stands for timestamp).

```
ts(t.prepare) < ts(t.coordCommit)
< ts(t.subordCommit)
```

Using the partial ordering and the two-phase commit logic, the log entry sequence can be generated such that the timestamps appearing in log entries of the sequence are always in ascending order. A log entry sequence for the multiple recovery log files shown in Example 7 is given as follows.

Example 8 *A log entry sequence of the multiple recovery log files shown in Example 7*

```
STX11-Insert
STX21-Insert
STX12-Insert
STX22-Insert
STX13-Insert
STX21-Prepare
GTX2-CoordCommit
STX21-SubordCommit
STX12-Prepare
STX13-Prepare
GTX1-CoordCommit
STX12-SubordCommit
STX13-SubordCommit
```

The log reader uses the timestamps associated with the TM log entries in multiple recovery log files to generate the single log entry sequence. The log reader reads the log entries from the partition node with the current minimum timestamp of all unprocessed TM log entries. The set of partition nodes participated in the DB2 EEE source database defined in the configuration forms the node list of the log reader. The log reader needs to establish connection to all partition nodes in the node list first. Then switch the connection to the node if the partition node owns the minimum unprocessed timestamp. Once a partition node is connected, the log reader calls the log read API to retrieve the log entries of that node.

Before we give the algorithm for log merging, we first look at where the log reader starts and stops.

4.1.1 Restart Logic

In CaptureEEE, there are multiple restart positions as there are multiple recovery log files. We choose to store one MinInflightLSN for each partition node, but a single MaxCommitTimestamp is used for all nodes. This provides a good balance between the needed time for updating the restart table and the time necessary for the restart. In DB2 EEE, LSNs cannot be used to determine the ordering of the log entries belonging to different recovery log files. Therefore, a single MinInflightTimestamp is required. However, it is complicated to use a single MinInflightTimestamp to replace one MinInflightLSN for each partition node.

For a warmstart situation, the MinInflightLSN for each partition node is retrieved from the *restart table* as the start LSN for the partition node. For a coldstart situation, current active LSN (of the last log entry of a recovery log file) is set as the start LSN for each partition node. The last unprocessed timestamp for each partition node is set to 0 for both warmstart and coldstart situations.

4.1.2 End Of Log

For Capture in DB2 DataPropagator, when it hits the *EOL* (End Of Log), the log reader returns the EOL code since there is no more log entries to read. However this is no longer the case in DB2 EEE since there are multiple log files. In DB2 EEE, time difference may exist among the local clocks of the partition nodes. When the log reader hits the EOL of the recovery log file of one partition node, it is possible that many log entries in the recovery log files of other nodes have not yet been read. However, it is also possible that the next TM log entry to be written on the node with the EOL reached by the log reader keeps the minimum unprocessed timestamp. Therefore, based on the partial ordering implemented in timestamps associated with the TM log entries, the log reader is unable to switch to other nodes when the log reader hits the EOL of the log file of one node.

Is the log reader able to continue the delivery of more log entries to the single log entry sequence when it hits the EOL of the log files of some partition nodes? We analyze all possible situations between the partition nodes with EOL seen to the log reader and the partition node with the current minimum unprocessed timestamp, and come up with the following propositions.

Proposition 1 *When the log reader hits EOL of any partition node n_1, the log reader can switch to the partition node n_2 which has the minimum unprocessed timestamp if the unprocessed log entry with this minimum timestamp is a* commit.

Proposition 2 *When the log reader hits EOL of any partition node n_1, the log reader can switch to the partition node n_2 which has the minimum unprocessed timestamp if the unprocessed log entry with this minimum timestamp is a* coordCommit, *and either n_1 is not in the participant list of the coordCommit.*

Proposition 3 *When the log reader hits EOL of any partition node n_1, the log reader can switch to the partition node n_2 which has the minimum unprocessed timestamp if the unprocessed log entry with this minimum timestamp is a* prepare.

Proposition 4 *When the log reader hits EOL of any partition node n_1, the log reader can switch to the partition node n_2 which has the minimum unprocessed timestamp if the unprocessed log entry with this minimum timestamp is a* subordCommit, *and the subtransaction does not participate in a global transaction which n_1 is the coordinator and the coordCommit has not been processed by the log reader of n_1.*

We explain the above propositions.

If the log entry with the minimum unprocessed timestamp is *commit*, this is a local transaction executed independently from n_1, therefore, the minimum unprocessed timestamp has no conflict with the upcoming log entries in n_1, including a TM log entry with a timestamp less than the minimum unprocessed timestamp of n_2.

If the log entry with the minimum unprocessed timestamp is *prepare*, it will cause no conflict. Since *prepare* happens before *coordCommit*, it has no impact on the commit of a transaction.

If the log entry with the minimum unprocessed timestamp is *coordCommit* or *subordCommit*, the conditions stated in the propositions are required to check in case any conflict may cause with the upcoming log entries in n_1.

These propositions can be used to help the log reader continue the delivery of more log entries in most situations. However, the single log entry sequence generated this way may not satisfy the ascending timestamp property since the above propositions are based on the following fact: the partial ordering is a sufficient condition but not a necessary condition for the ordering of events. In CaptureEEE, however, the ascending timestamp property is used for other purposes, e.g., recording the commit sequence of processed transactions. As such, we solve the EOL problem by the direct internal support from DB2 EEE engine. Whenever the log read API hits the end of a recovery log file, the log read API returns both the EOL return code and a virtual timestamp to the caller. The virtual timestamp is the timestamp DB2 EEE would have written at the time the EOL situation happened if a timestamp would have been needed.In addition, this virtual timestamp is *smart* since it gets bigger when the log reader hits the same EOL next time, this allows log entries in other nodes go through.

4.1.3 Log Entry Merging Algorithm

We consider a list $N = [n_1, n_2, \cdots, n_k]$ of k partition nodes. Each partition node has a recovery log file containing a sequence of DM and TM log entries. A timestamp is associated with the following TM log entries: *commit* for local transaction commit, *prepare* for subordinator prepare, *coordCommit* for coordinator commit, and *subordCommit* for subordinator commit.

Let the variable *curNode* point to the current partition node. For each partition node n_i $(1 \leq i \leq k)$, we define the following variables:
lastUnprocTS: to keep the timestamp of the last unprocessed TM log entry;
logEntryOnHold: to point to the last unprocessed TM log entry;
conn: to keep the handle for a connection to the partition node.

In the following, we give the algorithm for log entry merging. In the algorithm, the function *getCurNode*() returns a pointer to the partition node with $min(n_i.lastUnprocTS)$. In case there are more than one node own $min(n_i.lastUnprocTS)$, choose the node with the smallest node number or choose one randomly.

INPUT: A list $N = [n_1, n_2, \cdots, n_k]$ of k partition nodes, each has a recovery log file.
OUTPUT: A merged single log entry sequence S in the ascending order of timestamps.

Step 1 Initialization
Set NIL to curNode.
For each node n_i $(1 \leq i \leq k)$:
- Set 0 to $n_i.lastUnprocTS$;
- Set NIL to $n_i.logEntryOnHold$;
- Establish a connection to the node and assign the handle to $n_i.conn$.

Step 2 Switch to the right node
Call getCurNode() and set the returned value to curNode, switch node connection to curNode.conn.

Step 3 Process the log entry on hold
If curNode.logEntryOnHold is not NIL, append the log entry pointed by curNode.logEntryOnHold to S and set NIL to curNode.logEntryOnHold.

Step 4 Get next log entry
Read next log entry from curNode.conn.

Step 5 Handle EOL
If EOL is returned, process the EOL as follows:
- Set new EOL virtual timestamp to curNode.lastUnprocTS;
- Call getCurNode() to find new curNode;
- If the new curNode hits EOL last time or curNode = getCurNode(), return EOL (EOL of the single logical log entry sequence is reached!);
- Set getCurNode() to curNode;
- Go to Step 3.

Step 6 Process TM log entries with new timestamp
If the returned log entry is a commit, a prepare, a coordCommit, or a subordCommit TM log entry, process the returned TM log entry as follows:
- Set the new timestamp associated with the log entry to curNode.lastUnprocTS;
- Set the pointer to the log entry to curNode.logEntryOnHold;
- Call getCurNode() to find new curNode;
- If curNode ≠ getCurNode(), set getCurNode() to curNode, switch node connection to curNode.conn;
- Go to Step 3.

Step 7 Process DM and other TM log entries
Append the log entry to S, and go to Step 4.

Below is an example which illustrates how the above algorithm works for generating the single log entry sequence from the multiple recovery log files in Example 7.

Example 9 *Log entry sequence generated from the multiple partition log files in Example 7 using the algorithm*

```
STX11-Insert
STX21-Insert
STX21-Prepare (hold)
STX12-Insert
STX22-Insert
GTX2-CoordCommit (hold)
STX13-Insert
STX13-Prepare (hold)
STX21-Prepare (delivery)
STX21-SubordCommit (hold)
GTX2-CoordCommit (delivery)
STX12-Prepare (hold)
STX31-SubordCommit (delivery)
GTX1-CoordCommit (hold)
STX12-Prepare (delivery)
STX12-SubordCommit (hold)
STX13-Prepare (delivery)
STX13-SubordCommit (hold)
GTX1-CoordCommit (delivery)
STX12-SubordCommit (delivery)
STX13-SubordCommit (delivery)
```

The algorithm scans each partition recovery log file once. It checks at the TM log entry for *commit*, *prepare*, *coordCommit*, or *subordCommit*. If the timestamp associated in the TM log entry is no longer the minimum unprocessed timestamp, the log entry is on hold and the log reader switches to the node which has the current minimum unprocessed timestamp. The unprocessed log entry on hold is delivered first, if any.

4.2 Building and Merging Transactions

From the single log entry sequence generated by the log reader, the transaction builder and merger are able to build both local and global transactions by processing the log entries from the single log entry sequence. The transaction builder takes care of all DM log entries and all TM log entries for local transactions, while the transaction merger takes care of all TM log entries for global transactions.

4.2.1 Building Local Transactions and Subtransactions

The transaction builder has the same logic as the Capture program of DB2 DataPropagator. Its role is to capture each local transaction as well as each subtransaction belonging to a global transaction from the single log entry sequence returned by the log reader. For each DM log entry, the transaction builder checks the TID of that log entry. The TID is used to identify both a local transaction and a subtransaction belonging to a global transaction. If the transaction with this TID is not in the local transaction memory, a new entry needs to be created for the transaction with the TID. The DM log entry is always appended to the end of the transaction. All log entries of a local transaction or a subtransaction comes from the same physical recovery log file. For a Commit TM log entry, the transaction builder knows that a local transaction has been captured and this transaction is ready in the local transaction memory for inserting to the staging tables at the next CD synch point.

For an Abort TM log entry, the transaction builder clear the local transaction in the local transaction memory. The transaction did not make any changes to the source database since the DB2 had performed a rollback to undo all updates made by this transaction.

4.2.2 Merging Subtransactions

While the TID is used to build a local transaction or a subtransaction of a global transaction, the GTID is used to merge subtransactions with the same GTID to a global transaction. A GTID is only included in the TM log entries for global transaction management, i.e., the prepare, coordCommit, and subordCommit log entries.

The transaction merger works on the set of subtransactions that is created by the transaction builder. By the time a prepare log entry for global transactions is processed, all the DM log entries of the subtransaction on the subordinator node have already been processed by the transaction builder and put into the local transaction memory.

The main task of the transaction merger is to process three types of TM log entries for global transactions:

- *Prepare*
  For a *prepare* log entry, the transaction merger checks the GTID of that log entry. If the global transaction with this GTID has not been created in the global transaction memory, a new entry is created for the global transaction with the GTID. The transaction merger also checks the TID associated with the *prepare* log entry and maps the subtransaction identified by TID in the local memory to the global transaction identified by GTID in the global transaction memory.

- *CoordCommit*
  For a *CoordCommit* log entry, the transaction merger takes both GTID and TID from this log entry and maps the subtransaction (if any) identified by the TID to the global transaction identified by GTID. To this point, we know that all the

subtransactions of the global transaction identified by GTID have been captured and mappings from all subtransactions to the global transaction have been done. The global transaction is ready for publishing to staging tables.

- *SubordCommit*
  If more than one subtransaction are involved in a global transaction, the transaction merger does nothing with a *subordCommit* log entry. A *subordCommit* log entry is useful for a global transaction which involve only one subtransaction on a subordinator node (type 2 global transaction). In this case, this global transaction is processed as a local transaction.

4.3 Inserting to Staging Tables

At each CD synch point, the staging table inserter outputs the captured transactions into staging tables as well as the restart information into restart table.

Once a local transaction is captured in the transaction builder or a global transaction is captured in transaction merger, the information (e.g., commit timestamp, transaction identifier, transaction type) of the transaction is appended in a queue ready for use by the staging table inserter. For a local transaction, the task of insertion is the same as the Capture program. For a global transaction, log entries of all its subtransactions need to be inserted into staging tables as a single transaction. The information of a global transaction (including GTID) is recorded in UOW table, while log entries of all subtransactions are inserted into the CD table associated with the GTID.

For restart information, as introduced in Section 4.1.1, CaptureEEE choose to store one MinInflightLSN for each node, but a single MaxCommitTimestamp is used for all partition nodes.

The output of the staging table inserter matches that of the Capture program to ensure compatibility to the existing Apply program. For example, the structure of the staging tables remains unchanged.

4.4 Implementation of the Prototype

A prototype of CaptureEEE has been implemented on DB2 UDB V8. This prototype is implemented as an extension of the Capture program of DB2 DataPropagator. Therefore, CaptureEEE behaves the same as Capture, e.g., the single logical log entry sequence generated by the log reader of CaptureEEE emulates the single physical log file in the Capture program.

The components of CaptureEEE are implemented as independent threads. The log reader thread produces single log entry sequence independently from the transaction builder/merger thread which consumes this log entry sequence. The transaction builder/merger thread produces local/global transactions in local/global transaction memory independently from the staging table inserter which takes these transactions as input and outputs them into staging tables. The context management provided in DB2 is used to switch the connection from one node to another.

Preliminary performance analysis has been done on this prototype of CaptureEEE. It turns out that for a logical node configuration, there is only minor performance drop. However, for a physical node configuration, the number of log entries processed per second drops around 20% for adding a new physical partition node. This is caused by the overhead to switch nodes by the log reader.

5 Conclusion

With the increasing deployment of partitioned database systems like DB2 EEE in OLTP environments, there is a need to replicate the operational data from these partitioned systems to other systems for the purposes, say data warehousing or backup/recovery. In this paper, we addressed issues of data replication in partitioned database systems. We introduced CaptureEEE which is an extension of the Capture program of DB2 DataPropagator. This extension allows the transactions executed in DB2 EEE to be captured and replicated. The architecture and the components of CaptureEEE were presented. In particular, the algorithm for merging the log entries was discussed in detail. In CaptureEEE, the partial ordering in timestamps generated using Lamport clock and the PrC version of the two-phase commit protocol were used to guide the merging of log entries and the merging of subtransactions into global transactions.

In the future, the multiple log reader approach will be investigated to catch up with the performance penalty caused by the single log reader approach.

References

[1] Y. Al-Houmaily, P. Chrysanthis, and S. Levitan. An argument in favour of presumed commit protocol. In *Proceedings of the International Conference on Data Engineering*, pages 255–265, 1997.

[2] Y. Amir and C. Tutu. From total order to database replication. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 494–, 2002.

[3] T. A. Anderson, Y. Breitbart, H. F. Forth, and A. Wool. Replication, consistency, and practicality: Are these mutually exclusive? In *Proceedings of the SIGMOD Conference*, 1998.

[4] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison Wesley, Massachusetts, 1987.

[5] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz. Update propagation protocols for replicated databases. In *Proceedings of the SIGMOD Conference*, 1999.

[6] P. Chundi, D. J. Rosenkrantz, and S. S. Ravi. Deferred updates and data placement in distributed databases. In *Proceedings of the International Conference on Very Large Data Bases*, 1996.

[7] Oracle Corporation. *Oracle8i(tm) Advanced Replication*, 2000.

[8] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of the SIGMOD Conference*, 1996.

[9] IBM. *DB2(tm) Universal Database Enterprise-Extended Edition: Quick Beginnings V7*, 2000.

[10] IBM. *DB2(tm) Universal Database: Replication Guide and Reference V7*, 2000.

[11] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-r, a new way to implement database replication. In *Proceedings of the International Conference on Very Large Data Bases*, pages 134–143, 2000.

[12] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *TODS*, 25(3):333–379, 2000.

[13] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), 1978.

[14] E. Pacitti, P. Minet, and E. Simon. Fast algorithms for replica consistency in lazy master replicated databases. In *Proceedings of the International Conference on Very Large Data Bases*, 1999.